ИСП Институт Системного Программирования
Российской Академии наук

Труды Института Системного Программирования

Том 22

Москва 2012

ИСП **Учреждение Российской академии наук**
Институт системного программирования РАН

Труды Института Системного Программирования

Том 22

Под редакцией
академика РАН В.П. Иванникова

Москва 2012

УДК51

Труды Института системного программирования: Том 22.
/Под ред. В.П. Иванникова/ – М.: ИСП РАН, 2012. – 492 с.

В двадцать втором томе Трудов Института системного программирования публикуются статьи, содержащие результаты работ, проводимых в Институте в 2011-2012 годах.

ISBN 978-0-543-57631-6

© Институт Системного Программирования РАН, 2012

Содержание

Предисловие.....5

Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения
А.И. Аветисян.. 11

Планирование команд и конвейеризация циклов на современных архитектурах
А.И. Аветисян.. 19

Программная конвейеризация циклов на платформе ARM
Р. Жуйков, Д. Мельник, Р. Бучацкий 33

Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM
Р. Жуйков, Д. Плотников, М. Варданян.....49

Задача локального распределения регистров во время динамической двоичной трансляции
К. Батузов 67

Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU
К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян.....77

Восстановление структуры бинарных данных по трассам программ
А.И. Аветисян, А.И. Гетьман.....95

Инкрементальный анализ исходного кода на языках C/C++
В.О. Савицкий, Д.В. Сидоров 119

Комбинированный (статический и динамический) анализ бинарного кода
А.Ю.Тихонов, А.И. Аветисян 131

Подход к восстановлению потока управления запутанной программы
И.Н. Ледовских, М.Г. Бакулин 153

Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования
В.Н. Игнатьев 169

Перспективы виртуализации высокопроизводительных систем архитектуры x64
А. О. Кудрявцев, В. К. Кошелев, А. И. Аветисян 189

Использование стандарта OpenCL для программирования ПЛИС
А. Белеванцев, А. Меркулов, В. Платонов 211

Оптимизация расчётов в пакете OpenFOAM на GPU
А. В. Монаков 223

Финальные модели спецификации
И. Бурдонов, А. Косачев 233

Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов
С. П. Вартанов, Д.В. Сидоров 281

Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux
М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов..... 293

Интерполяция формул с кванторами в CSIsat на основе инстанцирования
В. С. Мутилин В. С., М. У. Мандрыкин. 327

Анализ типовых ошибок в драйверах операционной системы Linux
В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов 349

Автоматизация построения расписаний для периодических систем реального времени
А. Третьяков 375

Системы рекомендаций: обзор современных подходов
А.Г. Гомзин, А.В. Коршунов 401

Случайные графы, модели и генераторы безмасштабных графов
М.М. Берновский, Н.Н. Кузюрин..... 419

Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ
В.А.Захаров, Т.А. Новикова 435

Об одной задаче Коффмана-Шора, связанной с упаковкой прямоугольников в полосу
М.А. Трушников..... 456

Сравнение сложностей задач нахождения базиса Гребнера идеала и решений этого идеала
А. В. Шокуров..... 463

П р е д и с л о в и е

В 22-м томе Трудов ИСП РАН публикуются статьи сотрудников Института (главным образом, молодых специалистов), в которых описываются результаты научных исследований, полученные в конце 2011-го – начале 2012 гг.

Том открывается двумя статьями А.И. Аветисяна. Первая из них называется «Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения» и содержит описание метода двухэтапной компиляции программ на языках общего назначения (Си/Си++), основанного на компиляторной системе LLVM и позволяющего проводить оптимизации программ с учетом профиля пользователя и особенностей его целевой машины, а также организовывать развертывание программ в облачном хранилище с дополнительной оптимизацией и поиском дефектов программ.

Во второй статье А.И. Аветисяна – «Планирование команд и конвейеризация циклов на современных архитектурах» – предлагается метод планирования команд и конвейеризации циклов, основанный на расширяемой архитектуре планировщика, содержащей компоненты выявления и использования параллелизма на уровне команд.

В статье Романа Жуйкова, Дмитрия Мельника и Рубена Бучацкого «Программная конвейеризация циклов на платформе ARM» описывается адаптация для архитектуры ARM алгоритма поворотного модульного планирования, используемого в компиляторе GCC. Для обеспечения поддержки архитектуры ARM были в этот алгоритм были внесены значительные изменения, которые позволили ускорить ряд тестовых приложений на 3-4%.

Статья Романа Жуйкова, Дмитрия Плотникова и Мамикона Варданяна «Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM» посвящена описанию системы для автоматической настройки параметров компиляции, предназначенной для использования на встраиваемых платформах. С использованием этой системы, в частности, были выявлены недочеты в работе компилятора GCC, приводящие к генерации неоптимального кода для платформы ARM.

Кирилл Батузов представил статью «Задача локального распределения регистров во время динамической двоичной трансляции», в которой рассматриваются возможные улучшения алгоритма локального распределения регистров в QEMU, эмуляторе аппаратного обеспечения, основанного на использовании динамической двоичной трансляции. Практические результаты показали, что дальнейшее улучшение алгоритма возможно только при изменении имеющихся ограничений, накладываемых на этот алгоритм.

К. Батузов, П. Довгалюк, В. Кошелев и В. Падарян в своей статье «Два способа организации механизма полносистемного детерминированного

воспроизведения в симуляторе QEMU» предлагают два подхода к реализации полносистемного детерминированного воспроизведения в симуляторе QEMU, которые отличаются тем, какие компоненты виртуальной машины включаются в воспроизведение. Оцениваются достоинства и недостатки этих подходов, оценивается уровень накладных расходов при записи журнала невоспроизводимых событий..

В статье А.И. Аветисяна и А.И. Гетьмана «Восстановление структуры бинарных данных по трассам программ» рассматривается задача восстановления форматов бинарных данных. Предложены подходы к решению данной задачи, а также описывается реализация этих подходов в рамках разрабатываемой системы восстановления форматов.

Статью «Инкрементальный анализ исходного кода на языках C/C++» написали В.О. Савицкий и Д.В. Сидоров. В статье описывается метод построения статического анализатора кода, позволяющий существенно сократить время повторного поиска дефектов для языков C\C++. Используется свойство малого отношения количества лексем из исходного файла к количеству лексем из заголовочных файлов. Метод реализован в программном продукте для среды разработки MS Visual Studio.

Статья А.Ю.Тихонова и А.И. Аветисяна «Комбинированный (статический и динамический) анализ бинарного кода» посвящена обсуждению проблем анализа программ в бинарном коде для распознавания алгоритмов составляющих их функций, выяснения особенностей реализации алгоритмов, обнаружения недокументированных возможностей. В ряде случаев статический анализ может не дать результатов. Предлагается использовать в таких случаях в дополнение к статическому анализу динамический анализ (анализ трасс программы). Описывается разработанная и реализованная авторами среда динамического анализа бинарного кода TrEx и ее интеграция с известной средой статического анализа Ida Pro.

И.Н.Ледовских и М.Г.Бакулин являются авторами статьи «Подход к восстановлению потока управления запутанной программы». В статье описывается подход к восстановлению потока управления программы, запутанного комбинацией различных преобразований – от вставки недостижимого кода с помощью непрозрачных предикатов до виртуализации. Приводятся результаты тестирования прототипной реализации данного подхода на модельном примере, запутанном с применением различных инструментальных средств защиты.

В статье В.Н. Игнатьева «Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования» отмечается, что для достижения переносимости, надёжности и безопасности программ на C и C++ могут вводиться дополнительные ограничения на язык и стиль программирования. Предлагается новый метод формализации и классификация таких ограничений, описывается система их автоматической проверки, основанная

на применении статического анализа, а также способ интеграции с распространёнными системами сборки проектов.

А.О. Кудрявцев, В.К. Кошелев и А.И. Аветисян в статье «Перспективы виртуализации высокопроизводительных систем архитектуры x64» исследуют перспективы применения технологий виртуализации в области высокопроизводительных вычислений на платформе x64. Рассматриваются основные причины падения производительности при запуске параллельных программ в виртуальной среде. Подробно рассматриваются системы виртуализации KVM/QEMU и Palacios. В качестве тестовых пакетов используются HPC Challenge и NAS Parallel Benchmarks.

Статья Андрея Белеванцев, Алексея Меркулова и Владимира Платонова «Использование стандарта OpenCL для программирования ПЛИС» посвящена использованию стандарта OpenCL для облегчения программирования логических интегральных схем (ПЛИС), применяемых в качестве акселератора в гетерогенной вычислительной системе. Описывается схема реализации подмножества стандарта, обеспечивающая обмен памятью и управление выполнением задач на ПЛИС в предположении, что ПЛИС связана с центральным процессором через шину PCI-express.

Статья «Оптимизация расчётов в пакете OpenFOAM на GPU» представлена А.В. Монаковым. В статье рассматривается задача повышения скорости расчётов в пакете OpenFOAM за счёт переноса части вычислений на графические акселераторы (GPU). Приводится краткий обзор пакета и анализ переноса на GPU метода сопряжённых градиентов. Описаны несколько оптимизаций, часть из которых специфична для рассматриваемой реализации, а часть применима не только для GPU. Приводятся первичные результаты тестирования производительности.

Целью статьи Игорять Бурдонова и Александра Косачева «Финальные модели спецификации» является выделение подмножества трасс спецификации, достаточного для генерации полного набора тестов. Такое подмножество авторы назвали *финальной трассовой моделью спецификации*. Традиционная LTS-модель удобна тем, что является способом конечного представления регулярных множеств трасс. Для представления финальной трассовой модели в работе предлагается разновидность LTS, названная финальной RTS (Refusal Transition System). Такая модель обладает целым рядом свойств, полезных для генерации тестов.

В статье С. П. Вартанова и Д.В. Сидорова «Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов» предложена оптимизация алгоритма проверки выполнимости булевых формул DPLL (Davis — Putnam — Logemann — Loveland) с помощью кэширования промежуточных результатов при решении задачи нахождения входных данных для неинтерактивных программ. Дополнительная информация для оптимизации алгоритма запоминается на одном из предыдущих запусков алгоритма. Возможность подобного рода

модификации алгоритма основана на особенности последовательностей проверяемых формул.

Мандрыкин М.У., Мутилин В.С., Новиков Е.М. и Хорошилов А.В. – авторы статьи «Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux». В работе анализируются методы, используемые в современных инструментах статической верификации Си программ. Представлен обзор текущих возможностей инструментов в части поддержки конструкций языка Си, масштабируемости, видов проверяемых свойств и достоверности результатов анализа. Рассмотрены особенности применения инструментов статической верификации для анализа исходного кода драйверов устройств операционных систем и описаны существующие системы верификации драйверов, построенные на основе таких инструментов.

Статья В.С. Мутилина и М.У. Мандрыкина «Интерполяция формул с кванторами в CSIsat на основе инстанцирования» посвящена реализации на основе инструмента CSIsat интерполяции Крейга для формул с кванторами, заданных в рамках комбинации теорий вещественной линейной арифметики и неинтерпретируемых функций. Предложен способ реализации интерполирования таких формул с помощью инстанцирования подкванторных выражений с использованием внешнего SMT-решателя CVC3. Описываются изменения, которые были внесены в интерполятор и решатель для реализации поддержки кванторной интерполяции. Приводятся результаты тестирования модифицированного интерполятора как на задачах, полученных из набора SMTLIB, так и на тестах, специально сгенерированных для этих целей.

Статью «Анализ типовых ошибок в драйверах операционной системы Linux» представили В.С. Мутилин, Е.М. Новиков и А.В. Хорошилов. В статье предлагается методика выявления и классификации типовых ошибок и соответствующих им правил на основе изменений, вносимых в драйверы операционной системы Linux. Приводятся результаты применения данной методики, обсуждаются полученная классификация и распределение типовых ошибок по классам.

В статье Андрея Третьякова «Автоматизация построения расписаний для периодических систем реального времени» рассматривается проблема построения расписаний для систем, имеющих жесткие ограничения по времени работы. Эта проблема является одной из основных при проектировании систем авионики. В работе проанализированы существующие алгоритмы, выделены их недостатки и предложено собственное решение, удовлетворяющее необходимым требованиям. Описываются разработанные инструменты для визуализации, редактирования и валидации создаваемых расписаний.

А.Г. Гомзин и А.В. Коршунов написали статью «Системы рекомендаций: обзор современных подходов». Статья представляет собой обзор основных алгоритмов, используемых в системах рекомендаций. Рассмотрены методы

коллаборативной фильтрации, методы, анализирующие содержимое объектов, и методы, использующие базы знаний. Все рассматриваемые методы имеют свои недостатки. В статье предложены способы комбинирования методов построения рекомендаций, позволяющие избавиться от этих недостатков.

Статья М.М. Берновского и Н.Н. Кузюрина «Случайные графы, модели и генераторы безмасштабных графов» посвящена обсуждению моделей случайных графов, описывающих реальные сети, которые возникают в различных областях: биологии, компьтерных науках, инженерии, социологии. Особое внимание уделено моделям, описывающим социальные сети.

В статье В.А.Захарова и Т.А. Новиковой «Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ» описывается новая модификация алгоритма проверки логико-термальной эквивалентности программ, основанной на операции вычисления точной нижней грани в решетке конечных подстановок. Показано, что трудоемкость предложенного алгоритма оценивается величиной $O(n^6)$, где n - размер анализируемых программ.

Статью «Об одной задаче Коффмана-Шора, связанной с упаковкой прямоугольников в полосу» представил М.А. Трушников. В статье предложен новый онлайновый алгоритм упаковки прямоугольников в полосу, существенно превосходящий известный алгоритм Коффмана-Шора по качеству получаемой упаковки.

В статье А. В. Шокурова «Сравнение сложностей задач нахождения базиса Гребнера идеала и решений этого идеала» сравниваются сложности задач нахождения решения системы алгебраических уравнений и базисов Гребнера идеалов этих систем.

Том завершается статьей Я.А. Хетагурова «О построении аппроксимирующих функций характеристик системы». В этой статье выявляется взаимное влияние характеристик на показатели системы, части, устройства посредством построения общих математических моделей (ММ), использующих одинаковые характеристики систем, частей, устройств. Для получения уравнений ММ применяются линейные и гиперболические аппроксимирующие зависимости, основанные на данных характеристик систем, частей и устройств, близких по назначению и применению.

Академик РАН В.П. Иванников

Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения

Арутюн Аветисян <arut@ispras.ru>

Аннотация. В статье описывается метод двухэтапной компиляции программ на языках общего назначения (Си/Си++), основанный на компиляторной системе LLVM и позволяющий проводить оптимизации программ с учетом профиля пользователя и особенностей его целевой машины, а также организовывать развертывание программ в облачном хранилище с дополнительной прозрачной оптимизацией и поиском дефектов программ. Особенностью метода является применимость к языкам общего назначения и использование общей компиляторной инфраструктуры на всех этапах оптимизации и развертывания программы.

Ключевые слова: динамическая оптимизация, LLVM, уязвимость, облачное хранилище.

1. Введение

Широко применяемые статические оптимизации и учет профиля программы на машине разработчика являются основными видами оптимизации программ на языках Си и Си++. Очевидно, что при этом возникают сложности с точным учетом особенностей поведения пользователя и архитектуры его машины, поскольку при сборке на машине разработчика эти особенности либо не могут быть выяснены, либо требуется поддержка многих версий собранной программы для точной подгонки под необходимую архитектуру. Для решения этих задач, как впервые предложено в [1, 5], предлагается разделить этап изначальной сборки программы с применением машинно-независимых оптимизаций и этап окончательной специализации программы для конкретной машины пользователя и поведения пользователя. Такой подход мы называем методом двухэтапной компиляции.

Для осуществления двухэтапной компиляции необходимо распространение программы в объектных файлах, содержащих ее внутреннее представление, сохраняющее информацию высокого уровня и позволяющее проводить профилирование и машинно-зависимую оптимизацию программы. Наличие общей инфраструктуры двухэтапной компиляции позволяет решать следующие задачи:

- Учет точного профиля пользователя при динамической оптимизации на его машине, т.е. программа оптимизируется на конкретном наборе входных данных для данного конкретного запуска, а разные запуски программы могут приводить к различным оптимизациям. При этом значительно упрощается процесс разработки без потерь в производительности получаемой программы;
- Учет параметров машины пользователя при адаптивной оптимизации на его машине. Машинно-зависимые оптимизации, требующие информации об особенности архитектуры пользователя (распределение регистров, планирование команд, упреждающая загрузка, векторизация и т.п.) могут выполняться по-разному, существенно (десятки процентов) влияя на производительность;
- Оптимизация во время простоя, т.е. применение обоих описанных выше подходов во время простоя машины пользователя по тем же сохраненным данным профиля. Можно реализовывать разные оптимизационные стратегии на все время жизни программы (например, регулярная оптимизация по новым данным либо пересборка по поступлении сильно отличающегося профиля пользователя);
- Сокращение затрат на распространение и поддержку программы (достаточно поддерживать одну версию программы, при сборке которой применялись лишь машинно-независимые оптимизации). Конечно, для языка Си невозможно добиться портируемости, схожей с Java, из-за того, что особенности целевой машины попадают в программу на самых ранних этапах компиляции (а именно – на этапе препроцессирования). Разумнее говорить о портируемости единой версии программы между вариантами одного семейства архитектур (например, ARM) либо между архитектурами с совпадающими размерами базовых типов данных (например, ARM и x86). Тем не менее, созданием специальной “виртуальной” переносимой целевой архитектуры и компиляцией пользовательских программ под нее, а потом – их “подгонкой” под реальную целевую архитектуру возможно частично ослабить эти ограничения. Вопросом для исследований остается, во-первых, доля реальных программ, для которых можно добиться портируемости таким образом, а, во-вторых, необходимые дополнительные затраты на сглаживание разницы между реальной и виртуальной архитектурами и, как следствие, падение производительности. Примером подобного подхода является проект Google Portable Native Client [4].

Основой для реализации метода двухэтапной компиляции может служить LLVM [2, 3] – популярная компиляторная инфраструктура с открытыми исходными кодами на языке Си++. В рамках LLVM реализованы статический компилятор, компоновщик, виртуальная машина, JIT-компилятор и другие библиотеки. Функционирование системы обеспечивается единым внутренним

представлением, которое может быть представлено в текстовом виде, в виде структур данных в оперативной памяти, а также в двоичном виде как бит-код. Этот бит-код может быть сохранен в промежуточных объектных файлах для дальнейшей оптимизации, в том числе динамической. При этом возможно использовать все предоставляемые LLVM возможности по обработке внутреннего представления (включая различные анализы, трансформации и т.п.).

Важно упомянуть отличия предлагаемого метода от похожих подходов, использующихся в динамических языках, особенно на платформе .NET [6] с применением на машине пользователя компилятора NGEN [7] для генерации из байт-кода MSIL бинарной программы для нужной целевой архитектуры. Во-первых, платформа .NET ограничивает использование языков общего назначения, например, поддерживая язык C++/CLI [8], который по сути является отдельным динамическим языком, лишь происходящим от Си++. Более того, компилятор NGEN может не поддерживать преобразование некоторых методов байт-кода MSIL в бинарный код, полагаясь для таких случаев на обычный JIT-компилятор. Во-вторых, насколько можно судить, при компиляции NGEN не использует собранный профиль конкретного пользователя, ограничиваясь статическими оптимизациями. Наконец, преимущество нашего подхода в использовании единой инфраструктуры LLVM на всех этапах компиляции, от оптимизаций на машине разработчика до машины пользователя, что позволяет переиспользовать код при поддержке динамических, статических оптимизаций, оптимизаций во время простоя, а также переиспользовать компоненты поддержки профиля.

Далее в разделе 2 описывается реализация метода двухэтапной компиляции на базе LLVM, рассчитанная в том числе на применение на встраиваемых устройствах с процессором ARM. Раздел 3 посвящен концепции совместного использования облачного хранилища и двухэтапной компиляции для повышения производительности и безопасности программ. Раздел 4 заключает статью.

2. Метод двухэтапной компиляции

В предлагаемой реализации метода двухэтапной компиляции на первом этапе приложение компилируется на машинах разработчиков специальным набором компиляторных инструментов на базе LLVM, при этом выполняются лишь машинно-независимые оптимизации. Результат компиляции сохраняется в бит-кодовых файлах LLVM, дополнительно автоматически генерируется информация об устройстве программного пакета и о схеме его инсталляции. На втором этапе программа оптимизируется на машине пользователя, возможно, с учетом его поведения и особенностей его вычислительной системы. Поддерживается несколько режимов работы: а) автоматическая кодогенерация бинарной программы, оптимизированной под конкретную архитектуру, и инсталляция программы с помощью сохраненной на первом этапе информации; б) динамическая оптимизация программы во время её

работы с учетом собранного профиля пользователя с помощью реализованной инфраструктуры сбора профиля на основе пакета Oprofile и JIT-оптимизации на основе JIT-компилятора LLVM (подробнее см. [5]); в) оптимизация программы с учетом профиля пользователя во время простоя системы для экономии ресурсов (idle-time optimization).

Опишем разработанную схему двухэтапной компиляции для программ, система сборки которых основана на использовании утилит configure и make. Доработка инструментов LLVM производилась из-за того, что изначально в LLVM не предусмотрены средства прозрачного, автоматического получения бит-кода с учетом зависимостей между модулями, а также отсутствует поддержка динамического связывания модулей с бит-кодом. На первом этапе происходит генерация установочного пакета, содержащего в себе модули бит-кода и скрипты автоматического развертывания. Вместо использования оригинальной утилиты configure предлагается использовать специальную обертку configure-proxy, осуществляющую необходимые подстановки и вызывающую оригинальную утилиту, результатом работы которой является скрипт make.sh. Принцип работы сгенерированного скрипта для сборки аналогичен – вызывается утилита make с передачей всех необходимых параметров, а дополнительно конфигурируются вызовы компилятора LLVM-GCC. Для компиляторов реализованы обертки, основанные на том же подходе – для каждого вызова изначального компилятора с помощью конфигурации, заданной измененными скриптами сборки, запускается компилятор LLVM-GCC или Clang, позволяющий получить (параллельно с изначальным объектным файлом) и файл с бит-кодовым представлением. Для того, чтобы не влиять на ход сборки программы и не модифицировать скрипты сборки, необходимо полностью сохранять все исходные результаты сборки.

Помимо этого, необходимые изменения внесены в компиляторы переднего плана и компоновщик – эти изменения позволяют отследить зависимости между отдельными модулями программы. После окончания компиляции программы с помощью скриптов пост-обработки происходит, создание инсталляционного пакета на основе сгенерированных зависимостей. Инсталляционный пакет содержит файлы с бит-кодом, файлы, помеченные как зависимости на этапе постобработки, и скрипты компиляции и установки.

На втором этапе во время установки программы существует две альтернативы – статическая компиляция или использование динамической компиляции. В обоих случаях все происходит прозрачно для пользователя за счет создания скриптов компиляции и установки на первом этапе – в зависимости от заданного режима работы скрипты вызывают либо статический компилятор LLVM для получения объектного кода по файлам с бит-кодом, либо вызывает интерпретатор и JIT-компилятор LLVM, который обеспечивает сборку необходимых частей программы во время ее работы.

При экспериментальной проверке корректности работы созданной системы двухпроходной компиляции на системах x86 и ARM дополнительно была

проведена оптимизация компонент LLVM для их более быстрой работы и потребления меньшего объема памяти, что существенно для встраиваемых архитектур. При кодогенерации программ на платформе ARM было достигнуто сокращение использование памяти на 1.6-10.9% и времени компиляции на 10-20%.

3. Облачное хранилище и двухэтапная компиляция

Метод двухэтапной компиляции позволяет оптимизировать программы пользователя на его машине как динамически, через создание соответствующих JIT-компиляторов, так и во время простоя программы, с учетом собранного профиля, отражающего поведение пользователя, и характеристик его машины. Тем не менее, для мобильных устройств часто полная оптимизация программ на устройстве является затруднительной, требуя большого количества ресурсов. Возможно избежать этих затрат путем переноса второго этапа компиляции на сервер приложений, а на устройство возложить лишь задачу сбора и передачи профиля на сервер.

Для организации такого сервера требуется новый метод распространения приложений. Такое распространение предлагается организовать через «облачное хранилище» приложений нового поколения, обеспечивающее как переносимость программ в рамках одного семейства процессорных архитектур ARM, так и высокую степень надежности и безопасности хранимых приложений. Для построения облачного хранилища необходимо использование предложенной схемы двухэтапной компиляции для получения представления программы в виде бит-код файлов LLVM, а также метаинформации об устройстве приложения и о схеме его установки (см. рисунок 1).

На первом этапе двухпроходной компиляции создается установочный пакет приложения с бит-кодом LLVM и указанной выше метаинформацией. После помещения пакета приложения в хранилище для обеспечения безопасности приложение может быть автоматически проверено инструментами среды Svace [9] на наличие критических ошибок и уязвимостей (используя бит-код LLVM, а также информацию о связях программы), а также в случае необходимости может быть проведен более глубокий динамический анализ, например, сборкой приложения и запуском в песочнице, соответствующей целевому устройству. Таким образом, обеспечивается необходимая степень безопасности приложений.

Для обеспечения продуктивности приложения после проверки его пакета на безопасность по запросу загрузки приложения конкретным пользовательским устройством в хранилище осуществляется второй этап схемы двухэтапной компиляции и генерируется установочный пакет с объектным кодом для конкретного устройства. После этого во время жизни приложения на устройстве по желанию пользователя может быть организован сбор динамического профиля на устройстве и сохранение его на диск [5]. В

периоды слабой загруженности устройства собранный профиль может быть передан в облачное хранилище для оптимизации приложения с его использованием, а на устройство загружена новая оптимизированная версия приложения.

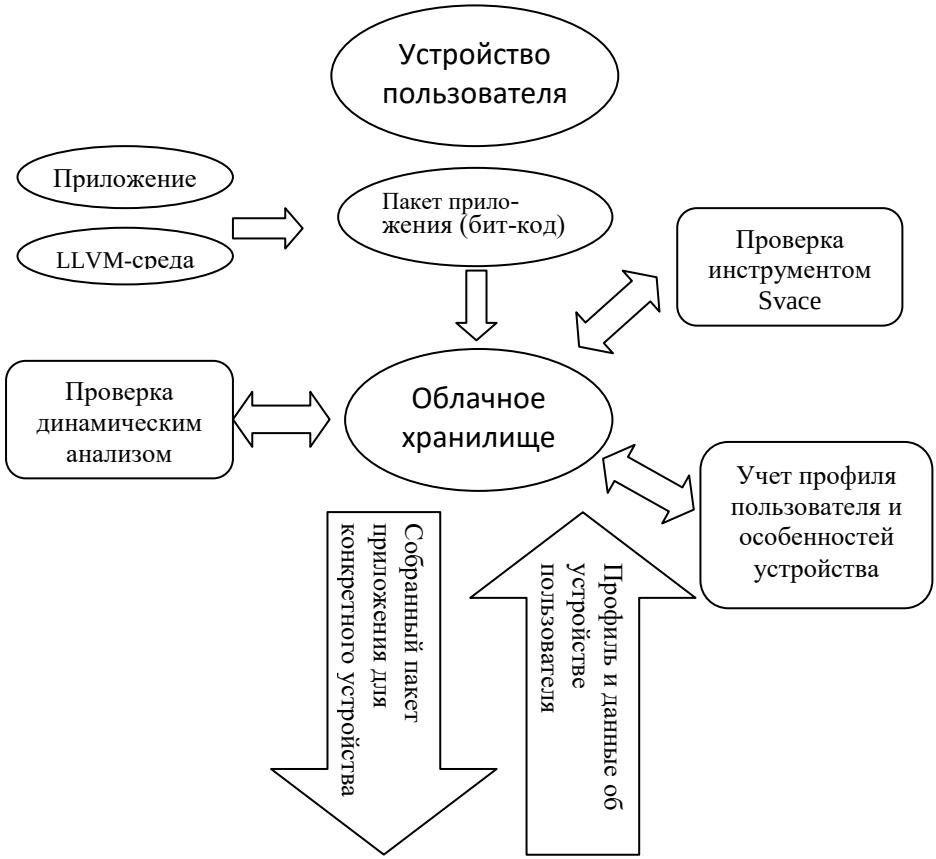


Рис. 1. Устройство облачного хранилища приложений.

Данный подход может быть использован для приложений как с открытым, так и с закрытым исходным кодом без каких-либо опасений со стороны разработчиков, так как восстановление исходного кода приложения по бит-код файлам LLVM является затруднительным. Отметим, что похожий подход к распространению программ широко применяется для динамических языков типа Java, но без дополнительной оптимизации и проверки на безопасность

программ, помещенных в хранилище, что и обеспечивает перспетивность предлагаемой схемы распространения приложений.

4. Заключение

В данной статье был рассмотрен метод двухпроходной компиляции для программ, созданных на языках общего назначения, и позволяющий повысить как производительность программ учетом профиля конкретного пользователя и особенностей его машины, так и безопасность программ проверкой на дефекты и уязвимости в облачном хранилище. В настоящее время в ИСП РАН ведутся работы в двух направлениях – по улучшению методов динамической оптимизации на базе двухпроходной компиляции с применением собранного профиля и подменой кода на стеке и по разработке и реализации форматов развертывания и сборки приложений для облачного хранилища.

Список литературы

[1] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, том 17, 2009 г. стр. 31-50.

[2] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[3] Компиляторная инфраструктура LLVM. http://llvm.org/

[4] Portable Native Client Introduction. http://www.chromium.org/nativeclient/pnacl/building-and-testing-portable-native-client

[5] А.И. Аветисян, К.Ю. Долгорукова, Ш.Ф. Курмангалеев. Динамическое профилирование программы для системы LLVM. Труды ИСП РАН том 21, 2011, стр. 71-82.

[6] Инфраструктура .NET. http://msdn.microsoft.com/en-us/netframework/aa496123

[7] Компилятор NGEN. http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx

[8] Stanley B. Lippman. Pure C++: Hello, C++/CLI. MSDN Magazine, Visual Studio 2005 Guided Tour, 2006. http://msdn.microsoft.com/en-us/magazine/cc163681.aspx

[9] Арутюн Аветисян, Андрей Белеванцев, Алексей Бородин, Владимир Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН том 21, 2011, стр. 23-38.

Two-stage compilation for optimizing and deploying programs in general purpose languages

Arutyun Avetisyan <arut@ispras.ru>
ISP RAS, Moscow, Russia

Abstract. We describe the approach for two-stage compilation of C/C++ programs using the LLVM compiler infrastructure that allows optimizing programs taking into account the user profile and his/her target machine features, as well as deploying programs in a cloud storage while transparently optimizing and checking for defects. The notable features of the approach are its applicability to programs written using general-purpose languages and utilizing the common compiler infrastructure on all optimization and deployment stages.

On the first stage (host machine) the build process is transparently captured and the LLVM bitcode files are generated with the proper support for archive/library files. LTO optimizations are also performed at this time, and the program dependencies and installation structure (e.g. libraries, installed resource or documentation files) are also captured. On the second stage (target machine) the final code generation and installation is performed with the transparent path translation. Either static code generation or dynamic JIT-based compilation is supported.

As mentioned, a cloud-based deployment strategy can also be used allowing for additional features like defect and vulnerability checking while the cloud store has access to the deployed programs in the LLVM bitcode form.

While optimizing LLVM toolchain for working on ARM embedded devices we have achieved the memory consumption reduction of up to 10% and 10-20% compile time decrease.

Keywords: dynamic optimization, LLVM, security vulnerability, cloud storage.

References

[1]. A. Belevantsev, D. Zhurikhin, D. Melnik. Kompilyatsiya programm dlya sovremennykh arkhitektur. [Program compilation for modern architectures] Trudy ISP RAN [The Proceedings of ISP RAS], volume 16, p. 31-50, 2009. (In Russian)

[2]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[3]. LLVM Compiler infrastructure, http://llvm.org/

[4]. Portable Native Client Introduction, http://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client

[5]. A.I. Avetisyan, K.U. Dolgorukova; Sh.F. Kurmangaleev. Dinamicheskoe profilirovanie programmy dlya sistemy LLVM [Dynamic profile collection for LLVM]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 71-82 (in Russian)

[6]. The .NET infrastructure, http://msdn.microsoft.com/en-us/netframework/aa496123

[7]. The NGEN Compiler, http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx

[8]. Stanley B. Lippman. Pure C++: Hello, C++/CLI. MSDN Magazine, Visual Studio 2005 Guided Tour, 2006, http://msdn.microsoft.com/en-us/magazine/cc163681.aspx
[9]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm. [Using static analysis for finding security vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 23-38 (in Russian).

Планирование команд и конвейеризация циклов на современных архитектурах

Арутюн Аветисян <arut@ispras.ru>

Аннотация. В статье предлагается метод планирования команд и конвейеризации циклов, основанный на расширяемой двухкомпонентной архитектуре планировщика – выявления и использования параллелизма на уровне команд. Компонент выявления параллелизма основан на подходе селективного планирования и состоит из ядра, поддерживающего перенос команд с созданием компенсационных копий, и модулей, реализующих дополнительные преобразования команд, включая спекулятивное и условное выполнение. Компонент использования параллелизма заключается в наборе эвристик, организующих выбор наилучшей команды для планирования на данной итерации планировщика. При описании разработанных компонент делается упор на улучшения базового подхода, необходимые для реализации предлагаемого метода в промышленном компиляторе. Приводятся экспериментальные результаты на платформах Intel Itanium и ARM.

Ключевые слова: планирование команд, конвейеризация циклов, спекулятивное выполнение, условное выполнение.

1. Введение

Для достижения высокой продуктивности вычислений в рамках одного ядра современного многоядерного процессора необходимо применение машинно-зависимых оптимизаций, позволяющих эффективно использовать ресурсы процессора. Такие оптимизации с необходимостью должны оперировать некоторой моделью ресурсов процессора и выполняться в дополнение к хорошо известным машинно-независимым оптимизациям (удаление избыточных выражений, оптимизация индуктивных переменных, вынос инвариантов и т.п.). Существует три основных машинно-независимых оптимизаций – автоматическая векторизация кода, планирование команд и конвейеризация циклов. Центральное место среди них занимает планирование команд, в комплексе с которым может выполняться и конвейеризация циклов. Векторизация кода среди этих оптимизаций стоит особняком, т.к. требует в основном хорошего анализа зависимостей по данным между массивами, анализа алиасов, и модели векторных устройств и регистров целевого процессора, в остальном представляя собой техническую задачу. Планирование команд, напротив, является NP-полной задачей, и основной сложностью при разработке алгоритма планирования является, во-первых,

создание инфраструктуры, поддерживающей все необходимые для выявления параллелизма преобразования команд, а во-вторых, создание эвристик, демонстрирующих хорошую производительность на репрезентативном наборе тестов. Последнему вопросу часто уделяется мало внимания при разработке алгоритмов планирования, а ведь именно правильное использование выявленного параллелизма дает в результате улучшение производительности программы.

В данной статье описывается разработанный нами метод планирования команд, основанный на известном подходе селективного планирования. К сожалению, базовая схема селективного планирования в том виде, как она изложена в литературе [9], не позволяет применить ее в промышленном компиляторе типа GCC [10], а также применить ее к архитектурам с явным параллелизмом команд и встраиваемым архитектурам. Основными недостатками, повлекшими за собой необходимость разработки нового метода, являются:

- Предполагаемая применимость базовых преобразований команд планировщика ко всем командам внутреннего представления компилятора. На практике для реальных программ это предположение оказывается неверным;
- Отсутствие механизма выбора лучшей на данном шаге команды для планирования из множества доступных, т.е. не разработана упомянутая выше задача использования выявленного параллелизма;
- Большая сложность, выражающаяся в значительном росте времени компиляции на реальных программах. Так, первоначальная версия алгоритма, реализованная нами для компилятора GCC, замедляла его в два раза. Для достижения приемлемой производительности оказалось необходимым переработать ядро алгоритма – изменить анализ зависимости по данным, анализ жизни переменных, базовые механизмы преобразования команд;
- Отсутствие преобразований, необходимых для выявления параллелизма на современных архитектурах с явным параллелизмом команд типа Itanium (спекулятивное выполнение) и на встраиваемых архитектурах типа ARM (условное выполнение).

Несмотря на описанные недостатки, алгоритм селективного планирования имеет и достоинства: во-первых, следование общей схеме двухпроходного планирования сверху вниз и четкое разделение шагов по выявлению и использованию параллелизма программы позволяет на основе селективного планирования построить метод с расширяемой архитектурой, в котором легко можно добавлять новые преобразования команд, увеличивающие набор доступных для планирования команд и, следовательно, выявляющие больше параллелизма, а также добавлять новые эвристики выбора команды для планирования, улучшая использование параллелизма. Во-вторых, идея селективного планировщика о нескольких точках планирования (т.н. *фронте*

планирования) и возможности одновременного планирования команд в этих точках делает равноправными команды из разных базовых блоков и путей выполнения – одна точка планирования неизбежно отдает предпочтение командам из тех блоков, в которые эта точка попадает раньше. Следовательно, для регионов планирования без четко выраженных «горячих" путей можно строить в среднем лучшие расписания. Наконец, поддержка фронта планирования позволяет организовать конвейеризацию циклов с использованием той же инфраструктуры планировщика, причем с поддержкой сложного потока управления внутри цикла. Конечно, как будет показано далее, эвристики планирования при этом также нужно доработать.

Поэтому алгоритм селективного планирования был положен нами в основу предлагаемого в разделе 2 метода планирования команд и конвейеризации циклов, позволяющего добавлять новые преобразования команд и эвристики выбора лучшей команды. Раздел 3 посвящен добавлению к разработанному планировщику спекулятивного и условного выполнения команд. Раздел 4 представляет экспериментальные результаты, а раздел 5 заключает статью.

2. Метод планирования команд и конвейеризации циклов

2.1. Общая схема планирования

Итак, предлагаемая нами схема планирования по организации планировщика следует двухпроходному планированию сверху вниз, сохраняя поддержку нескольких точек планирования и разделяя этапы выявления и использования параллелизма. Самый верхний уровень алгоритма планирования выглядит следующим образом:

1. **Построение ациклических регионов планирования** по данной процедуре программы.
2. Планирование каждого региона отдельно:
    a. **Обход фронтом планирования региона** в топологическом порядке. При этом в точках разветвления потока управления фронт переходит сразу на все блоки-последователи, а в точках слияния – уже запланированные участки кода не получают новых точек планирования.
    b. **Для каждой точки планирования**, при наличии свободных ресурсов:
        i. **Выявление параллелизма** (сбор доступных для планирования команд в данной точке).
        ii. **Использование параллелизма** (выбор лучшей команды с помощью набора эвристик или переборных алгоритмов).
        iii. **Планирование выбранной команды**. При этом обновляются структуры данных планировщика,

необходимые для первых двух шагов по сбору команд и выбору лучшей.

Наиболее интересная часть схемы заключается в цикле планирования пункта 2b, работающего для каждой точки планирования до заполнения ресурсов процессора на текущем моделируемом такте, и именно эту часть мы рассмотрим подробнее (см. рисунок 1). Пока стоит отметить, что расширение возможностей планировщика и учет реальных программ может изменить и остальные части. Например, построение регионов в пункте 1 и их обход в пункте 2 может быть расширен так, что планировщик будет поддерживать конвейеризацию вложенных циклов – для этого для каждого отдельного цикла строится свой регион, а обход регионов осуществляется по гнездам циклов, в порядке от внутренних к внешним. При этом необходимо учесть, что после планирования внутреннего цикла состав региона, соответствующего внешнему циклу, может измениться. Учет реальных программ может выразиться в том, что, например, отдельные виды циклов нельзя конвейеризовать (при наличии несводимого потока управления, возникающего из-за обработки исключений или применения операторов перехода), либо планируется настолько большая процедура, что максимальный размер региона (как по количеству базовых блоков, так и команд в них) необходимо ограничить.



Рис. 1. Схема основного цикла планировщика.

При выявлении параллелизма (см. рисунок 1) основную часть нагрузки несет ядро планировщика, обеспечивающее сбор готовых для планирования команд

(возможно, с учетом одного или нескольких преобразований) и перенос команды в точку планирования с поддержанием корректности программы. В схеме алгоритма верхнего уровня это соответствует шагам 2b-i и 2b-iii. Ядро ответственно за два базовых преобразования команд – *клонирование* команд (т.е. возможность обеспечить такой перенос команды, который потребует создания *компенсационных копий*) и *унификацию* команд (т.е. при доступности одинаковых команд на разных путях выполнения планируется лишь одна копия команды), используя в основном данные анализа зависимостей для выполнения своих действий. Остальные преобразования команд обеспечиваются *модулями* расширения – это *переименование регистров* и связанный с ним *перенос через копии* (уже предложенные в [9] преобразования), *спекулятивное* выполнение, *условное* выполнение. Модули обращаются к ядру за необходимой информацией о свойствах команд, жизни регистров и т.п. для проведения преобразования, при этом модули могут добавлять свою информацию в структуры данных, поддерживаемые ядром. Конвейеризация циклов также может рассматриваться как модуль, но с точки зрения корректности получаемой программы в основном обеспечивается правильным построением регионов и выбором их порядка обхода.

Использование параллелизма (шаг 2b-ii) заключается в поддержке набора эвристических алгоритмов, приоритизирующих собранные команды, после чего для выдачи выбирается наиболее приоритетная команда. Возможен дополнительный перебор элементов множества готовых команд для упорядочивания выдаваемых команд в пределах одного цикла, как в [12]. Как и в случае модулей, применение эвристик, основанных на информации о критическом пути команды, относительных вероятностях выполнения отдельных команд, жизни регистров и т.п., требует обращения к ядру для поддержания всех необходимых данных в актуальном состоянии. Вообще, в описанной схеме расширение планировщика заключается в написании нового модуля или эвристики выбора, при этом у ядра запрашивается необходимая информация либо через ядро организуется ее сбор при обходе планируемого региона.

Таким образом, мы обобщили базовую схему селективного планирования, предложив организовать планировщик с явно выраженными частями выявления и использования параллелизма, который легко расширяется написанием новых модулей или эвристик выбора. В следующем подразделе мы опишем новые алгоритмы ядра планировщика, которые понадобились по сравнению с базовым подходом [9] для применения к реальным программам.

2.2. Алгоритмы ядра планировщика

К улучшениям алгоритмов ядра планировщика можно отнести выбор структур данных, учет различных типов команд, улучшения анализа зависимостей по данным, улучшение планирования выбранной команды и поддержка модулей переименования регистров и конвейеризации циклов. Опишем кратко

основные сделанные улучшения (более подробно каждое из сделанных изменений описано в [1-4, 13].

Структуры данных планировщика хранят информацию для регионов, базовых блоков, команд и *выражений* (правых частей команд) в массивах, которые индексируются по идентификатором соотвестствующих элементов. Наиболее интересным является понятие *виртуальной* команды, хранящей все данные о виде команды, не связанные с ее местом в регионе, и являющейся кэширующим контейнером для представления одной и той же команды в разных частях региона в процессе ее подъема к точке планирования. Виртуальная команда вместе с данными о ее конкретной точке нахождения называется *операцией* и является элементом множества доступных команд. Следовательно, при добавлении модулями потоково-нечувствительной информации она хранится как данные виртуальной команды, а потоково-чувствительной – как данные операции.

Необходимость учета различных типов команд, как уже упоминалось, возникает из-за неприменимости преобразований ко всем без исключения командам. Например, переименование регистров применимо только к выражениям, т.к. нужно выделить правую и левую части команды для генерации соотвествуюших пересылок. Аналогично, для некоторых команд нельзя создать компенсационные копии (для ассемблерных команд, атомарных командщ и некоторых других), т.к. невозможно гарантировать, что копия команды будет иметь ту же семантику. Тип команды указывает на применимые к ней преобразования, является потоково-нечувствительной информацией и хранится в виртуальной команде.

Улучшения анализа зависимостей направлены на две компоненты: уточнение зависимости и ускорение анализа. Уточнение необходимо из-за знания того, из какой части команды происходит зависимость, и достигается добавлением *событий* анализатора (найдена зависимость по памяти/регистрам, чтению/записи и т.п.) [4]. Подпиской на события анализатора планировщик может выполнять различные действия – отслеживание частей команд, породивших зависимость, стоимостей зависимостей и т.д. Ускорение анализа получается за счет кэширования наиболее частых результатов анализа – наличия/отсутствия зависимости – битовыми множествами, содержащими идентификаторы соответствующих команд, и за счет поддержки специальных контекстов анализатора, хранящих эффект разбора данной команды [3].

Этап планирования выбранной команды требует поиска всех команд, которые могли быть унифицированы в данную. Изначальный подход селективного планирования [9] уже включает в себя средства для ускорения этого поиска, сохраняя промежуточные множества готовых команд в начале каждого базового блока. Для преобразованных команд этого оказывается недостаточно, так как при поиске необходимо отменять сделанные преобразования ровно в тех точках региона, при протаскивании через которые они были впервые применены. Для ускорения этой части алгоритма мы

разработали механизм кэширования преобразований, который при выполнении преобразования некоторой операции на некоторой команде записывает обе формы операции – изначальную и преобразованную – в вектор *истории преобразований*, сохраняемый для каждой команды из региона. При поиске запланированной операции, зная текущую обрабатываемую команду региона, легко найти в кэше старую версию искомой команды. Тонкостью при реализации кэша является тот факт, что к моменту поиска операции команда, на которой она была изначально преобразована, уже может быть сама запланирована и перемещена со старого места в регионе. Также возможно, что отмену преобразования нужно будет делать по достижении при поиске компенсационной копии команды, на которой была преобразована операция. Поэтому поиск в кэше осуществляется не по идентификатору команды, а по хэшу ее виртуальной команды, который одинаков для всех компенсационных копий одной команды. При создании компенсационных копий векторы историй переносятся на эти копии с оригинальной команды.

Поддержка переименования регистров ядром требует отслеживания регистров, которые не могут быть использованы для переименования при перемещении команды из-за нарушений интервалов жизни регистра, а также из-за ограничений целевой архитектуры, возникающих при планировании уже распределенного на регистры кода. Наиболее важным является первый случай, и для него выполнено два улучшения. Во-первых, частый случай доступности оригинального целевого регистра также кэшируется в потоково-чувствительных данных операции. Во-вторых, для контроля над агрессивностью планировщика ведется учет регистрового давления. В каждой точке региона ядром поддерживается значение текущего регистрового давления, которое изменяется на пути перемещения вверх планируемой команды. Предполагается, что влияние команды на регистровое давление не зависит от ее места в программе (что не совсем точно), а значит, хранится в виртуальной команде. При передвижении команды наверх обновляется значение максимального регистрового давления вдоль соответствующего пути переноса, и, если это максимальное значение превысит текущее сохраненное значение регистрового давления в некоторой точке пути, то оно обновляется. Для ограничения роста давления сохраненное значение (хранящееся отдельно для каждого регистрового класса для большей точности) сравнивается с максимальным числом доступных регистров, и для некоторого значения перемещение команды запрещается.

Конвейеризация циклов с точки зрения ядра планировщика из предложенной нами схемы является особым преобразованием, поскольку поддерживается уже имеющимся преобразованием клонирования команд за исключением единственной детали – запретом сбора доступных для планирования команд вдоль обратного ребра региона-цикла. Разрешением сбора команд для планирования в начале цикла, в том числе уже запланированных, достигается конвейеризация. Конечность планирования обеспечивается запретом перехода

фронта планирования на запланированный код и ограничением на максимальное количество раз планирования команды [3].

2.3. Оценка выгоды перемещения при конвейеризации

Особеностью поддержки конвейеризации является трудность оценки влияния конвейеризованных команд на расписание цикла в целом [1]. Из-за важности этой проблемы и отсутствия ее решения в базовом алгоритме [9] мы кратко обсуждаем ее и в настоящей статье. Рассмотрим следующий типичный цикл:

```
Загрузка данных
Использование данных
Переход на начало цикла
```

После планирования загрузок и некоторых использований планировщик может обнаружить «дыры» в расписании, которые не могут быть заполнены командами, находящимися ниже. В этот момент он рассматривает для конвейеризации команды сверху, и если они подходят, то происходит следующее:

```
<-- Уже запланированный -->
<-- Старое место загрузки, «дыра» -->
Использование данных <-- уже запланировано -->
Загрузка <-- здесь был свободный слот -->
Незапланированный код
Переход на начало цикла
```

Был использован свободный слот для конвейеризации загрузки, но при этом создана «дыра» среди уже запланированных команд, которая потенциально могла бы быть заполнена одной из команд, использующих результат загрузки (запланированной или незапланированной).

Другая проблема состоит в том, что команды из начала цикла могут быть выданы слишком рано после конвейеризованных командами в конце цикла, от которых они зависят. В том же примере после конвейеризации загрузки на ее старом месте в случае Itanium возникает команда проверки, которая выполняется сразу после команды перехода на начало новой итерации, что достаточно близко для того, чтобы вызвать задержку. Чтобы решить эту проблему, между этими двумя командами необходимо больше запланировать других команд либо в начале цикла, либо загрузка должна быть помещена перед одной из команд, использующих значение с предыдущей итерации.

Мы пытаемся улучшить конвейеризованное расписание, выполняя дополнительный проход планировщика над конвейеризованным циклом, но на этот раз уже с отключенной конвейеризацией. Такая техника помогает уплотнить конвейеризованное расписание, заполняя появляющиеся «дыры» за

счет увеличения времени компиляции. Для решения второй проблемы мы используем две различные эвристики. Во-первых, в некоторых случаях можно предсказать, произойдет ли задержка на команде проверки. Это происходит, когда спекулятивная загрузка планируется на одном такте с последней командой перехода в цикле. Пусть N – номер этого такта, C – номер такта, на котором планируется команда проверки. Тогда расстояние между загрузкой и проверкой будет C+1 тактов, т.к. один такт необходим для команды перехода. Когда это расстояние больше либо равно L, латентности загрузки, можно выполнять спекуляцию. Во-вторых, мы стараемся увеличить промежуток между загрузкой и проверкой, откладывая планирование этой проверки с помощью уменьшения ее приоритета и давая другим командам больше шансов быть запланированными между загрузкой и проверкой, уменьшая вероятность задержки на команде проверки.

3. Поддержка спекулятивного и условного выполнения

3.1. Спекулятивное выполнение

Модуль поддержки спекулятивного выполнения опирается на теоретические результаты из [6] и использует выполненные улучшения анализа зависимостей, описанные выше и в [3, 5]. Согласно предложенной в [6] модели спекулятивного выполнения необходимо определить легальные возможности для спекулятивного преобразования и вероятности выполнения зависимостей по данным и по управлению, после чего с помощью утверждений 1-3 из [6] можно судить о выгоде каждого конкретного спекулятивного перемещения. Покажем, как в условиях промышленного компилятора можно максимально близко реализовать такой подход.

Мы используем описанные выше события анализа зависимостей для маркировки зависимостей как нуждающиеся в спекулятивном выполнении по данным либо по управлению для их преодоления (*статусы* зависимостей). После этого для поддержки целевой архитектуры необходимо уметь отвечать на вопрос, может ли данная команда быть выполнена спекулятивно определенным образом, и если да, то как выглядит ее спекулятивная форма. Так как разные виды спекулятивного выполнения могут быть последовательно применены при проносе команды наверх, то спекулятивный *статус* команды должен быть частью данных операции. Ядро планировщика обеспечивает корректное слияние статусов при слиянии множеств доступных команд вместе с остальными данными операций для того, чтобы при унификации спекулятивной и неспекулятивной команды была сгенерирована правильная команда для планирования. При создании компенсационных копий спекулятивных команд ядром также учитывается ее текущий статус, а на месте оригинальной команды создается соответствующая команда проверки.

Описанные изменения достаточны для выполнения легальных спекулятивных преобразований, но необходимо еще отобрать выгодные преобразования. Для этого те же события анализа зависимостей генерируют не только статусы, но и вероятностные оценки выполнения конкретной зависимостей. Из-за невозможности профилирования зависимостей по данным применяется набор эвристик [5] – различие между прямым доступом к элементам полей структуры и через указатели, различие указателей-параметров функции, различие баз указателей ведет к малой вероятности зависимости. Для зависимостей по управлению используются либо статические вероятности, предсказанные по потоку управления, либо данные профилирования. Аналогично, ядро планировщика обеспечивает слияние вероятностей зависимостей по управлению при перемещении команды наверх.

При планировании спекулятивной команды ее стоимость вместо формул по типу [8] отражается в наборе эвристик сортировки, заключающийся в предпочтении обычных операций спекулятивным, предпочтении спекуляции по данным более «вероятной» (как правило) спекуляции по управлению, сравнении спекулятивных команд одного типа по их вероятности. В условиях промышленного компилятора этих эвристик оказывается достаточно для приближенной оценки выгодности каждого конкретного преобразования, что показывается экспериментальными результатами – можно настроить вероятностные отсечки таким образом, чтобы получить выигрыш в среднем по репрезентативному набору тестов.

Не освещенным ранее вопросом, специфичным для нашего планировщика, является связь между проводимыми преобразованиями команд. Так, спекулятивная команда может быть как перенесена через копию, так и запланирована с созданием компенсационных копий. Для переименования регистров в спекулятивной команде необходимо сначала выполнить переименование, чтобы команда спекулятивной проверки получила правильный номер целевого регистра [4].

3.2. Условное выполнение

Поддержка условного выполнения при планировании команд описана нами в работах [7,8], здесь мы кратко остановимся на вопросах, связанных с использованием разработанной нами схемы планирования для реализации модуля условного выполнения. В отличие от спекулятивного выполнения, где в данных операции нужно сохранять ее спекулятивный статус, для условного выполнения нужно сохранять предикат, которым защищена операция. Далее, определение типов команд и левых/правых частей ядром также учитывает записанный предикат, отбрасывая его по необходимости. Главной особенностью является применение условного выполнения, т.е. наложение предиката, не при проносе команды через другую команду, а при слиянии ядром множеств готовых команд в точках слияния потока управления. В момент слияния ядро определяет, контролируется ли данный условный

переход предикатом, и при положительном ответе передает управление модулю условного выполнения, который накладывает выделенный предикат на те команды из текущего множества готовых команд, которые могут быть переведены в условную форму согласно описанию целевой архитектуры.

При поиске запланированной команды и ее перемещении наверх к точке планирования (этап 2b-iii в разделе 2) необходимо, как и для других преобразований, отменять условное выполнение, т.е. удалять контролирующий предикат соответствующих команд при спуске через команду условного перехода. Для этого преобразование в форму с условным выполнением записывается в истории преобразований команды, при этом в качестве команды, на которой произошло преобразование, записывается соответствующий условный переход, что позволяет автоматически отменить его при спуске через этот переход. Кроме того, нужно преобразовывать текущую операцию в условную форму при подъёме через условный переход, если выбранная команда также была преобразована на этом условном переходе. Чтобы решить эту задачу, мы разработали механизм повторения преобразований команд, также использующий историю преобразований. Этот механизм позволяет повторно произвести все преобразования, произошедшие над выбранной для планирования командой в процессе вычисления множества доступных команд, над искомой командой в процессе подъёма от найденной первоначальной команды. Наконец, при поиске команды при спуске через условный переход, контролируемый предикатом искомой команды, необходимо продолжать поиск только на одной ветви, соответствующей этому предикату: так как перенос команды через переход с получением условной формы не изменяет условий, при которых эта команда будет выполнена, то на ветви с обратным предикатом искомая команда никогда не будет выполнена, поэтому на этом пути нельзя удалять такие же команды, если они там есть.

Как и для спекулятивного выполнения, вопрос о комбинации условного выполенния с другими преобразованиями разрешается положительно – можно переименовывать команду в условной форме, а также дополнительно преобразовывать в спекулятивную форму и выполнять подстановку через копии. Отслеживание предиката команды также позволяет более точно оценивать интервал жизни ее целевого регистра (подробнее в [7]). Наиболее интересным результатом является использование условного выполнения вместо спекулятивного при конвейеризации циклов – это делает возможной конвейеризацию доступов в память на архитектурах, не поддерживающих необходимое для этого спекулятивное выполнение (ARM).

4. Экспериментальные результаты

Мы выполнили запуск стандартного промышленного набора тестов SPEC CPU 2000 [11] на архитектуре Intel Itanium с использованием компилятора GCC версии 4.6, в котором присутствует реализация нашего алгоритма планирования. По результатам запуска версии компилятора с поддержкой

спекулятивного выполнения и всех изменений раздела 2, реализованных в первую очередь, пакет тестов SPEC FP ускорился на 3.5% [1], отдельные тесты – до 10.9%. Это показывает хорошие результаты выявления параллелизма, обычно присутствующего на тестовых программах с плавающей точкой. Результаты на SPEC INT в среднем схожи со старым планировщиком, также улучшенным нами в результате работ [14], так как для т.н. целочисленных тестовых программ основное содержание параллелизма достается имеющейся по результатам [14] поддержкой спекулятивности. Затраты на время компиляции составляют 15-18%, что является нормальным показателем для высокого уровня оптимизации на платформе с явно выраженным параллелизмом, и эти результаты позволили включить разработанный планировщик команд по умолчанию для компилятора GCC на платформе Itanium для уровня оптимизации –O3, после его включения в основную ветвь компилятора, начиная с версии 4.4.

При запуске версии компилятора с поддержкой условного выполнения и результатов по улучшению анализа указателей из [5] средние результаты тестов SPEC FP улучшились в среднем на 4%, один тест сработал некорректно (в последних версиях поддержки условного выполнения данный тест был исправлен). При этом существенным достижением являлось отсутствие тестов, на которых производительность заметно падает, но при этом максимальное ускорение снизилось с 10.9% до 8.9%.

При тестировании разработанного планировщика на наборе программ (пакеты SQLite, CLucene, matmult, x264, Evas) на архитектуре ARM со всеми улучшениями, описанными в разделах 2 и 3, были получены ускорения от 1% до 3% в среднем по тестам этих программ. Наибольшое ускорение составляло до 20%, но при этом наблюдались и падения производительности в 5-8%. В целом, аккуратной настройкой планировщика можно добиться стабильного среднего ускорения для набора тестов, но из-за ограниченного параллелизма платформы ARM агрессивное планирование может ухудшить отдельные тесты, и требуется взвешенный подход при использовании планировщика для каждой конкретной программы на ARM.

5. Заключение

В данной статье был рассмотрен алгоритм планирования, созданный на базе селективного планировщика, который позволяет эффективно выявлять и использовать параллелизм на уровне команд с помощью набора преобразований, важнейшими из которых является переименование регистров, спекулятивное и условное выполнение, конвейеризация циклов. Выполненные нами улучшения алгоритма направлены, во-первых, на поддержку современных архитектур типа Itanium и ARM, а во-вторых, на возможность применения подхода к реальным программам в условиях промышленного компилятора GCC. Экспериментальные результаты показывают, что реализованные преобразования позволяют использовать

планирование после распределения регистров с мощью, близкой к алгоритмам, работающим до распределения регистров, с приемлемыми затратами по времени компиляции, которые позволили включить планировщик в основную ветвь компилятора GCC, начиная с версии 4.4.

Список литературы

[1] Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik. GCC instruction scheduler and software pipelining on the Itanium platform. 7th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-7). Boston, MA, USA, April 2008. http://rogue.colorado.edu/EPIC7/avetisyan.pdf

[2] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In Proceedings of GCC Developers' Summit 2006, Ottawa, Canada, June 2006, pp.1-12.

[3] Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, and Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. In Proceedings of GCC Developers' Summit 2007, Ottawa, Canada, July 2007, pp. 7-21.

[4] Andrey Belevantsev, Dmitry Melnik, and Arutyun Avetisyan. Improving a selective scheduling approach for GCC. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007. http://sysrun.haifa.il.ibm.com/hrl/greps2007/

[5] Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin. An Approach for Data Propagation from Tree SSA to RTL. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007.

[6] А.А.Белеванцев, С.С.Гайсарян, В.П.Иванников. Построение алгоритмов спекулятивных оптимизаций. Журнал Программирование, N3 2008, с. 21-42.

[7] Д. Мельник, А. Монаков, А. Аветисян. Поддержка команд с условным выполнением в селективном планировщике команд. Труды ИСП РАН, том 21, 2011 г.

[8] Dmitry Melnik, Alexander Monakov, Andrey Belevantsev, Tigran Topchan, Mamikon Vardanyan. Improving Selective Scheduler Approach with Predication and Explicit Data Dependence Support. 8th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-8), 2010.

[9] Soo-Mook Moon and Kemal Ebcioglu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853-898, November 1997.

[10] GCC, GNU Compiler Collection. http://gcc.gnu.org

[11] SPEC CPU 2000. http://spec.org/cpu2000/

[12] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In Proceedings of GCC Developers' Summit, Ottawa, Canada, June 2003.

[13] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, том 17, 2009 г. стр. 31-50.

[14] Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for Itanium. In Proceedings of GCC Developers' Summit 2005, Ottawa, Canada, June 2005.

Instruction scheduling and software pipelining for modern architectures

Arutyun Avetisyan <arut@ispras.ru>
ISP RAS, Moscow, Russia

Abstract. We describe the approach for instruction scheduling and software pipelining based on a two-stage extensible architecture of detecting and using the available instruction level parallelism. The detection stage is based on a selective scheduling approach and consists of a kernel supporting instruction movement with bookkeeping code creation and instruction unification, and of modules that implement additional instruction transformations such as register renaming, control and data speculation, predication (conditional execution). The usage stage is a number of heuristics for choosing the best instruction for schedule on the given scheduler iteration. Among the usual critical path heuristics, we introduce execution probability heuristics to control speculative movements, register renaming limitations with tracking register pressure, and the additional scheduling pass quickly removing the schedule holes possibly introduced by software pipelining. We concentrate on the basic approach improvements that were needed for implementing the suggested method in an industrial compiler. We also show experimental results of the scheduler on Intel Itanium and ARM platforms: Itanium speedup achieves 4% on average for SPEC FP 2000 tests with up to 10% for individual tests, while initial ARM support achieves 1-3% speedup for smaller programs.

Keywords: instruction scheduling, software pipelining, speculation, predication.

References

[1]. Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik. GCC instruction scheduler and software pipelining on the Itanium platform. 7th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-7). Boston, MA, USA, April 2008. http://rogue.colorado.edu/EPIC7/avetisyan.pdf

[2]. Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In Proceedings of GCC Developers' Summit 2006, Ottawa, Canada, June 2006, pp.1-12.

[3]. Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, and Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. In Proceedings of GCC Developers' Summit 2007, Ottawa, Canada, July 2007, pp. 7-21.

[4]. Andrey Belevantsev, Dmitry Melnik, and Arutyun Avetisyan. Improving a selective scheduling approach for GCC. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007. http://sysrun.haifa.il.ibm.com/hrl/greps2007/

[5]. Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin. An Approach for Data Propagation from Tree SSA to RTL. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007.
[6]. Belevantsev, A.A., Gaisaryan, S.S., Ivannikov, V.P. Construction of speculative optimization algorithms. Programming and Computer Software, V.34, I.3, 2008, pp. 138-153. DOI 10.1134/S036176880803002X
[7]. D. Melnik, A. Monakov, A. Avetisyan. Podderzhka komand s uslovnym vypolneniem v selektivnom planirovshhike komand. [Support for Conditional Execution in Selective Scheduling.] Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, p. 103-118, 2011. (In Russian)
[8]. Dmitry Melnik, Alexander Monakov, Andrey Belevantsev, Tigran Topchan, Mamikon Vardanyan. Improving Selective Scheduler Approach with Predication and Explicit Data Dependence Support. 8th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-8), 2010.
[9]. Soo-Mook Moon and Kemal Ebcioglu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853-898, November 1997.
[10]. GCC, GNU Compiler Collection. http://gcc.gnu.org
[11]. SPEC CPU 2000 tests, http://spec.org/cpu2000/
[12]. Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In Proceedings of GCC Developers' Summit, Ottawa, Canada, June 2003.
[13]. A. Belevantsev, D. Zhurikhin, D. Melnik. Kompilyatsiya programm dlya sovremennykh arkhitektur. [Program compilation for modern architectures] Trudy ISP RAN [The Proceedings of ISP RAS], volume 16, p. 31-50, 2009. (In Russian)
[14]. Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for Itanium. In Proceedings of GCC Developers' Summit 2005, Ottawa, Canada, June 2005.

Программная конвейеризация циклов на платформе ARM

Роман Жуйков <zhroma@ispras.ru>, Дмитрий Мельник <dm@ispras.ru>
Рубен Бучацкий <ruben@ispras.ru>

Аннотация. В данной работе описывается проделанная с компилятором GCC работа по адаптации имеющегося алгоритма поворотного модульного планирования для архитектуры ARM. Имеющийся в компиляторе алгоритм может работать только с циклами определенного вида, а на платформе ARM нет подходящей инструкции для организации циклов этого вида. Были выполнены предварительные оценки производительности с помощью реализации шаблона псевдо-инструкции для платформы ARM. По результатам этой проверки было решено расширить возможности алгоритма для оптимизации более общего вида циклов. Разработана и реализована поддержка циклов, счетчик которых меняется как арифметическая прогрессия. Для добавления такой поддержки были внесены значительные изменения в алгоритм поворотного модульного планирования. По результатам тестирования выявлено ускорение части тестовых приложений до 3-4%.

Ключевые слова: оптимизация программ; программная конвейеризация циклов; поворотное модульное планирование; архитектура ARM.

1. Введение

В современном мире широкое распространение получили различные встраиваемые системы. В связи с этим все больше возрастают требования к качеству кода для встраиваемых процессоров. Одним из способов получения более качественного кода является адаптация имеющихся многоплатформенных компиляторов к архитектурным особенностям встраиваемых платформ.

Цель данной работы – адаптировать имеющуюся в компиляторе GCC[1] программную конвейеризацию циклов для процессоров архитектуры ARM. Сложный оптимизационный алгоритм в данный момент работает лишь на небольшом классе циклов, причем такие циклы встречаются на очень ограниченном наборе платформ. Планируется расширить класс циклов, рассматриваемый оптимизацией поворотного модульного планирования.

Дальнейшее изложение построено следующим образом. В разделе 2 описывается программная конвейеризация циклов. Раздел 3 рассказывает об имеющейся реализации алгоритма поворотного модульного планирования.

Раздел 4 посвящен адаптации данной оптимизации для платформы ARM. В разделе 5 описаны полученные экспериментальные результаты.

2. Программная конвейеризация циклов

Программная конвейеризация циклов преобразует тело цикла таким образом, что на выполнение могут выдаваться команды из разных итераций исходного цикла, которые выполняются параллельно, подобно конвейеру. Рассмотрим пример:

```
for i = 1 to N
  A(i);
  B(i);
  C(i);
  D(i);
End
```

В этом примере A(i), B(i), C(i), D(i) обозначают инструкции, которые обрабатывают элемент данных i, зависящие друг от друга. Другими словами, A(i) должно быть выполнено перед тем, как может начать выполняться B(i). Например, A может загружать данные из памяти в регистр, B и C – выполнять арифметические операции над данными, а D – записывать полученный результат обратно в память. Предположим, что не существует зависимости между операциями для различных значений i. То есть, A(2) может начинаться до завершения D(1). Без программной конвейеризации, операции будут выполняться в следующем порядке:

```
A(1); B(1); C(1); D(1); A(2); B(2); C(2); D(2); A(3);…
```

Допустим, что на выполнение каждой инструкции уходит один такт процессора (без учета увеличения счетчика цикла). Также допустим, что в процессоре есть одно функциональное устройство, которое выполняет инструкции A и D (в примере – устройство работы с памятью) и одно функциональное устройство, которое выполняет инструкции B и C (в примере – арифметическое устройство). Тогда в рассмотренном случае на выполнение каждой итерации уходит четыре такта (при этом на каждом такте одно из функциональных устройств процессора простаивает, так как нет готовых для выполнения инструкций).

Теперь рассмотрим преобразованный цикл, который полностью использует все функциональные устройства процессора:

```
A(1);
B(1);
for i = 2 to N
  A(i), C(i - 1);
  B(i), D(i - 1);
```

```
end
C(N);
D(N);
```

Для него порядок выполнения инструкций будет следующим:

```
A(1); B(1); A(2), C(1); B(2), D(1); A(3), C(2); B(3), D(2); …
```

Здесь инструкции, разделенные ‘;’, выполняются на разных тактах процессора, а инструкции, разделенные ‘,’, – на одном. При этом можно легко проверить, что в среднем на выполнение каждой итерации тратится по два такта процессора.

Из этого примера видно, что при наличии возможности параллельного выполнения большого числа инструкций применение этой оптимизации может оказаться очень выгодным. Однако оно может быть связано с многими трудностями. В частности, для корректного преобразования цикла часто необходимо отдельно обрабатывать часть инструкций с нескольких первых и последних итераций – это ведет к увеличению размера кода (иногда в разы превышая размер тела цикла). Также, время жизни некоторых регистров может превышать длину одной итерации (например, какая-то инструкция использует значение переменной, вычисленное на предыдущей итерации), что означает необходимость использования временных регистров для хранения таких значений и, следовательно, создания новых операций для заполнения этих регистров. Эти дополнительные операции могут свести на нет все преимущества от применения конвейеризации циклов.

Было доказано[2], что проблема конвейеризации циклов за оптимальное количество тактов и с использованием минимального количества дополнительных регистров – NP-трудная. Поэтому на практике используются различные эвристические подходы. Один из таких подходов реализован в селективном планировщике команд[3], разработанном в отделе компиляторных технологий ИСП РАН. Реализация планировщика настроена главным образом на использование на процессорах Intel Itanium, однако может применяться и на других платформах (ARM и *x86-64*).

Еще одним вариантом алгоритма конвейеризации циклов является поворотное модульное планирование, реализация которого в компиляторе GCC описана в следующем разделе. Этот подход одновременно ищет такую расстановку инструкций с нескольких итераций цикла, чтобы минимизировать число тактов процессора, проведенных внутри цикла, и стремится минимизировать количество дополнительных регистров, пытаясь расставить инструкции цикла так, чтобы инструкции, устанавливающие и использующие значения регистров, были как можно ближе друг к другу.

3. Поворотное модульное планирование

Модульное планирование – это класс алгоритмов программной конвейеризации циклов, который упрощает задачу программной конвейеризации, используя одно общее расписание для всех итераций цикла и фиксированную частоту выдачи новых итераций.

Поворотное модульное планирование (Swing Modulo Scheduling) – это эффективный алгоритм из класса алгоритмов модульного планирования, целью которого является максимизация частоты выдачи новых итераций, а также минимизация регистрового давления, т.е. количества одновременно живущих регистров в одной итерации цикла.

В конвейеризованном цикле расписание для одной итерации цикла поделено на несколько стадий таким образом, что выполнение всех стадий по одной из последовательных итераций происходит одновременно. То есть, если, к примеру, тело цикла поделено на три стадии, то одновременно будут выполняться первая стадия i-й итерации, вторая стадия (i–1)-й итерации и третья стадия (i–2)-й итерации. Количество тактов между началом выполнения одной стадии и началом выполнения следующей, то есть количество тактов в одной стадии, называется интервалом запуска. На рисунке 1 для примера показан конвейеризованный цикл из пяти итераций, в теле которого три инструкции. Тело конвейеризованного цикла состоит из трех стадий и занимает четыре такта.

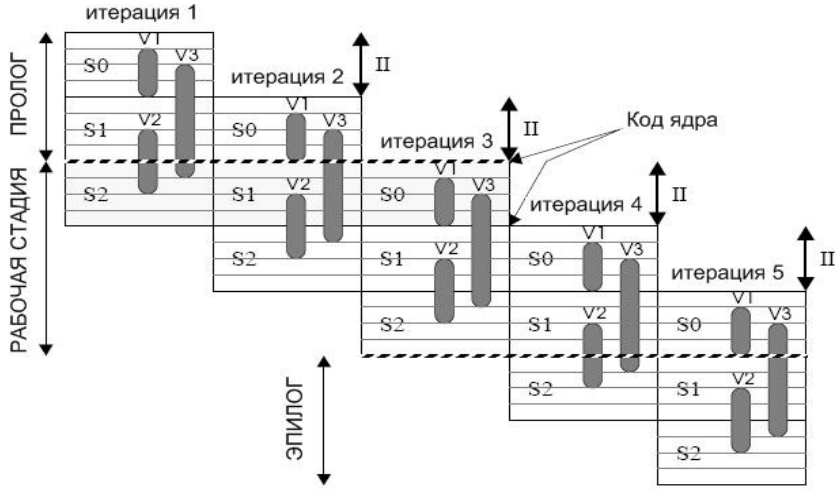


Рис. 1. Пример конвейеризованного цикла.

Очевидно, что чем меньше интервал запуска, тем чаще выдаются новые итерации, тем меньше времени выполняется цикл, то есть тем эффективнее

конвейеризация. Поэтому основной задачей алгоритма ПМП является нахождение такого минимального интервала запуска, чтобы ему соответствовало правильное расписание тела цикла. Проход, реализующий данную оптимизацию в компиляторе GCC, включается опцией –fmodulo-sched и работает перед проходом распределения регистров. Суть алгоритма заключается в последовательном переборе увеличивающихся значений интервала запуска, начиная с минимального и до максимального, и попытках нахождения правильного расписания для цикла при каждом значении интервала запуска.

Более формально, алгоритм выглядит следующим образом:

1. Построение графа зависимостей по данным.
2. Определение минимального и максимального интервалов запуска.
3. Определение порядка следования узлов.
4. Планирование тела цикла.
5. В случае успешного планирования всех узлов расписание готово, иначе необходимо увеличить размер интервала запуска и повторить фазу планирования. Если значение интервала запуска достигает своего максимального предела, то считается, что применение модульного планирования к данному циклу не даст выигрыша, и работа алгоритма заканчивается. В этом случае цикл будет обработан обычным планировщиком.
6. Генерация копий переменных.
7. Генерация пролога и эпилога.

Полное описание алгоритма приводится в работе [4]. Далее дано краткое описание фаз алгоритма для обозначения особенностей реализации поворотного модульного планирования в компиляторе GCC.

3.1. Построение графа зависимостей по данным

Для тела цикла строится *граф зависимостей по данным*, учитывающий как зависимости внутри тела цикла, так и зависимости между итерациями. При анализе зависимостей по данным используется анализ алиасов, а также данные о стоимости (латентности) инструкций из модели процессора и обычного алгоритма планирования ациклических участков кода.

3.2. Определение минимального и максимального интервалов запуска

Интервал запуска ограничивается снизу двумя факторами [5]: рекуррентными соотношениями в графе зависимостей (в графе зависимостей есть рекуррентное соотношение, если в нем существует простой цикл, то есть в исходном цикле используются значения, вычисленные на предыдущих итерациях) и ограничениями на ресурсы процессора.

Для максимального значения интервала запуска подходит длина любого правильного расписания для тела цикла. То есть, алгоритм пытается конвейеризовать цикл до тех пор, пока выполнять полученный цикл может получиться выгоднее, чем оригинальный цикл.

3.3. Фаза вычисления порядка следования узлов

Фаза вычисления порядка следования узлов использует построенный граф зависимостей и дополнительные атрибуты узлов и создает упорядоченный список, содержащий все узлы графа. Этот порядок указывает, в какой очередности узлы графа зависимостей будут анализироваться фазой планирования.

Фаза вычисления порядка следования узлов стремится учесть два фактора в процессе работы:

1. Давать больший приоритет инструкциям, расположенным на критических путях. Под критическим путем здесь подразумевается путь максимальной длины в ациклическом графе, который получается из графа зависимостей отбрасыванием зависимостей между итерациями. Тот факт, что на инструкции, которые окажутся в списке последними, будет больше ограничений, учитывается тем, что предпочитаются инструкции с меньшей мобильностью.
2. Уменьшать максимальное количество одновременно живых регистров. Для этого каждый узел помещается как можно ближе и к своим предкам, и к своим потомкам. Однако от порядка, в котором рассматриваются инструкции, может зависеть возможность такого планирования.

Для достижения обоих этих целей фаза упорядочивания узлов графа использует специальный обход графа зависимостей[6].

3.4. Фаза планирования

Фаза планирования инструкций рассматривает инструкции в порядке, заданном фазой упорядочивания и пытается вставить инструкции в частичное расписание цикла как можно ближе к соседним инструкциям, которые уже были запланированы. Инструкции планируются по-разному в зависимости от того, какие из их соседей уже присутствуют в частичном расписании.

Если для какого-либо узла не найдено подходящее место в расписании, то интервал запуска увеличивается на единицу, и фаза планирования повторяется заново с этим увеличенным значением интервала запуска, которое предоставит больше возможностей для составления правильного расписания. Одной из особенностей реализации алгоритма поворотного модульного планирования в GCC, влияющей на время его работы, является то, что фаза упорядочивания выполняется всего один раз, даже в том случае, когда фаза планирования выполняется несколько раз.

3.5. Генерация копий переменных

Этот этап необходим для того, чтобы из расписания тела цикла, сгенерированного на предыдущей фазе, получить программу, которая будет корректно выполняться. Например, если существуют регистры, срок жизни которых превышает длину интервала запуска, на которой было достигнуто успешное планирование, это означает, что инструкция, определяющая значение регистра, будет выполнена более одного раза до момента, когда это значение понадобится инструкции-потребителю, а, следовательно, правильное значение будет потеряно. Для таких регистров определяется количество копий, которые необходимо сгенерировать, и генерируются сами копирующие инструкции, после чего каждое использование исходного регистра заменяется копией, содержащей правильное значение. Создание копий переменных включается отдельной опцией компилятора GCC –fmodulo-sched-allow-regmoves. Когда данная опция выключена, расписание цикла составляется таким образом, чтобы избежать необходимости создания инструкций копирования.

3.6. Генерация пролога и эпилога

На этой фазе генерируется пролог и эпилог для получения окончательного расписания и корректной программы. Когда количество итераций цикла не известно на этапе компиляции есть несколько подходов. Например, можно выполнять отдельный эпилог в зависимости от количества итераций в, либо сохранить первоначальный цикл и выполнять его в случае, если количество итераций слишком мало, чтобы поток выполнения дошел до тела цикла (другими словами, данных недостаточно, чтобы загрузить конвейер). В реализации GCC, на которой основывается эта работа, применяется второй подход, так как он не приводит к резкому росту размера исполняемого кода. Предположим, при планировании тело цикла было поделено на три стадии. Это означается, что одновременно будут выполняться первая стадия i-й итерации, вторая стадия (i–1)-й итерации и третья стадия (i–2)-й итерации. Соответственно, в пролог цикла будут вынесены (скопированы) инструкций из первой и второй стадии первой итерации, а также дополнительно из первой стадии второй итерации. Аналогично в эпилог будет выноситься третья стадия предпоследней итерации, а также вторая и третья стадия последней итерации. В общем случае, если есть n стадий и они нумеруются с единицы, то инструкции i-й стадии будут n-i раз скопированы в пролог и i-1 раз скопированы в эпилог.

4. Адаптация алгоритма оптимизации для платформы ARM

Поворотное модульное планирование в компиляторе GCC использует шаблон *do-loop*, для распознавания циклов, в которых управляющая часть содержит все инструкции, изменяющие или использующие определенный регистр-

счетчик, а число итераций может быть изменено с помощью изменения значения этого регистра до цикла.

Do-loop – это цикл с уменьшающимся на единицу счетчиком, условие выхода из которого выглядит как “счетчик цикла равен нулю (или единице)”. На разных архитектурах этот шаблон определяется разными способами. На процессорах семейства *x86* есть инструкция “loop <метка>”, которая делает все вместе – уменьшение, сравнение и ветвление. Она использует регистр cx для хранения числа итераций. Похожая ситуация на процессорах Intel Itanium архимектуры *ia64*: существует специальный регистр ar.lc, который может быть использован только для этой цели, и инструкция br.cloop. Все другие поддерживаемые GCC платформы не имеют готовой инструкции, выполняющей все три действия, но компилятором допускается другой вариант организации *do-loop* цикла с инструкцией декремента отдельно от последней инструкции в теле цикла. Последняя инструкция, соответственно, должна выполнять сравнение и ветвление. Примером такой архитектуры является платформа Cell SPU. На ней есть инструкция “biz <регистр>, <*метка*>”, что означает “branch if zero”, то есть переход на метку происходит только при равенстве регистра нулю.

В алгоритме поворотного модульного планирование к *do-loop* циклам еще несколько требований. Счетчик цикла не должен использоваться в других инструкциях, вне управляющей части цикла. Цикл должен состоять из одного базового блок и не содержать вызовов других функций и процедур, а также других инструкций с побочными эффектами.

Возвращаясь к платформе ARM, на ней нет подходящей аппаратной инструкции для создания шаблона *do-loop* циклов. Однако прежде чем реализовывать поддержку алгоритма ПМП для более общего вида циклов, было решено сначала оценить возможности конвейеризации на процессорах ARM. Для этого необходимо было создать в машинно-зависимой части компилятора псевдо-инструкцию *do-loop* для платформы ARM.

4.1. Создание шаблона псевдо-инструкции

Были опробованы два способа организации шаблона. Оба подразумевают эмуляцию ассемблерной инструкции, которая состоит из двух реальных инструкций процессора. Пусть *reg* — регистр для счетчика цикла. Первый способ – использовать “sub *reg*, *reg*, 1” для декремента и пару “cmp *reg*, 0; bne <метка>” как инструкцию ветвления. Второй способ – просто взять пару инструкций “subs reg, reg, 1; bne <метка>” в качестве последней инструкции цикла. Второй способ должен работать быстрее, но при его реализации возникла следующая проблема с распределением регистров: в некоторых случаях при распределении регистров регистр-счетчик сбрасывается на стек, после чего не находится подходящего регистра для восстановления. Такая ситуация допускается алгоритмом распределения регистров, причем для правильного корректирования программы компилятором в этом случае

необходимо описать в машинно-зависимой части компилятора процесс замены инструкции на другие, использующие оперативную память вместо регистра.

Поскольку нашей целью была лишь предварительная оценка возможностей алгоритма поворотного модульного планирования на платформе ARM, реализация такого процесса замены инструкций была слишком трудоемкой. Поэтому был выбран для реализации первый способ создания шаблона псевдо-инструкции с помощью инструкций sub (для декремента на 1), cmp и bne. Данный вариант удалось запустить после внесения необходимых изменений в работу конечного автомата, описывающего состояние процессора в архитектурно-зависимой части GCC для платформы ARM. При тестировании оказалось, что полученная реализация генерирует неверный код для библиотеки *eina* (одна из библиотек для набора тестов *expedite*[7]). В процессе исследования этой проблемы было решено отказаться от использования псевдо-регистра и выбрать один конкретный аппаратный регистр на ARM (например, r5), чтобы использовать только этот регистр для *do-loop* циклов.

Использование одного аппаратного регистра допустимо, так как поворотное модульное планирование поддерживает работу только с *do-loop* циклами, являющимися самыми внутренними в группе вложенных циклов. Идея использования одного выбранного аппаратного регистра позволила избежать проблемы с распределением регистров и вернуться к использованию второго способа организации шаблона *do-loop* с помощью пары инструкций subs и bne. Дополнительно, было добавлено условие для генерации псевдо-инструкций с *do-loop* шаблоном только в случае, если выставлен уровень оптимизации не ниже первого (-O1) и включена опция –fmodulo-sched, отвечающая за ПМП в GCC.

После этих изменений, алгоритм ПМП заработал на платформе ARM следующим образом. В первую очередь, когда оптимизационный проход *do-loop* обрабатывает циклы, где число итераций может быть вычислено до начала цикла, такие циклы переформировываются для использования *do-loop* шаблона. До начала цикла, в пролог, добавляется инструкция “mov r5, <*1+число_итераций*>”. А две последние инструкции в цикле становятся “subs r5, r5, 1; bne <*метка–начало цикла*>” вместо исходной проверки условий выхода из цикла. После такого преобразования цикл становится подходящим для анализа с помощью имеющейся реализации ПМП, если он удовлетворяет другим обычным ограничениям - состоит из одного базового блока, который не содержит вызовов. Оптимизационный проход ПМП происходит при первой работе планировщика, до распределения регистров. Если найдено расписание команд, лучшее, чем исходное, цикл помечается, чтобы не быть перепланированным после распределения регистров. Соответственно, в успешно запланированных с помощью ПМП циклах аппаратный регистр r5 всегда оказывался занят для управляющей части цикла.

Экспериментально сравнивалась производительность базовых GCC оптимизаций уровня 2 (-O2) и этих же оптимизаций с опцией –fmodulo-sched. Измерялись результаты на тестовой плате OMAP3EVM. Тестовый набор для *sqlite*[8] показал замедление на 2%: один тест на 3% ускоряется, но другие замедляются на 1-4%. Тестовый набор *expedite* для библиотеки растеризации *libevas*[7] ускоряется на 0.25%. При этом отдельные тесты показывают результаты от замедления на 4% до ускорения на 5%.

Дополнительно было проведено тестирование набора небольших тестов *aburto* и исследованы полученные результаты. Тест *heapsort* ускоряется на 7%. Также увеличивается производительность простого умножения матриц и умножения с транспонированной второй матрицей на 2% и 4% соответственно. Тест *flops* замедляется на 2.5%. На других тестах производительность не изменилась.

Исследование ассемблерного кода и журналов компилятора GCC позволило сделать вывод, что ускорение программы происходит, если планировщик успешно составил новое улучшенное расписание для «горячего» цикла. Поворотное модульное планирование лучше распределяет нагрузку между итерациями, старается разнести максимально далеко друг от друга зависимые по данным инструкции с соседних итераций. Напротив, когда цикл «горячий» и планирование оказалось неуспешным, получается замедление, связанное с вставкой инструкций *do-loop* шаблона до прохода планировщика. Эти инструкции остаются в коде цикла независимо от того, было ли успешно составлено расписание. И, поскольку между этой парой инструкций есть дополнительная зависимость по данным, (ветвление зависит от флага, выставляемого сравнением), они дают небольшую задержку и приводят к замедлению. Один из возможных способов исправить эту ситуацию – реализация ПМП без требования *do-loop* шаблона, что требует больших изменений в платформо-независимой части ПМП. Логика алгоритма планирования во многом опирается на тот факт, что счетчик цикла не используется внутри цикла, кроме управляющих инструкций цикла.

4.2. Циклы с декрементом счетчика

Следующим этапом адаптации была реализация работы алгоритма ПМП на циклах, не использующих *do-loop* инструкций, счетчик которых ведет себя аналогично случаю *do-loop* циклов. Предполагается, что счетчик уменьшается каждую итерацию ровно на единицу, а цикл заканчивается при достижении счетчиком нуля. Также в этом случае необходимо отказаться не только от самого требования *do-loop* шаблона, но и разрешить использование (чтение) счетчика в инструкциях вне управляющей части цикла. Как следует из описания алгоритма планирования, в итоговом запланированном цикле число итераций уменьшается, часть инструкций копируется в пролог и эпилог. И если в *do-loop* циклах можно просто соответствующим образом поменять значение счетчика перед циклом, то в данном варианте такой подход оказывается недопустим.

Было выбрано простое решение - запланировать инструкцию декремента вместе с другими инструкциями, как будто она не относится к управляющей части цикла, а в конце проверить корректность полученного расписания. В силу особенностей реализации достаточно проверить, что инструкция декремента попала первую стадию в новом расписании, и отменить преобразование цикла в противном случае. Такой вариант реализации позволяет выиграть до 20% производительности на простейших тестовых примерах, однако не влияет на производительность настоящих тестов, поскольку циклы с таким поведением счетчика встречаются в реальных программах крайне редко.

Итак, исследования показали, что на тестовых программах планирование циклов с помощью ПМП позволяет значительно улучшить производительность на процессорах ARM. Поэтому было решено реализовать поддержку в алгоритме ПМП циклов более общего вида, чем *do-loop* циклы.

4.3. Поддержка циклов нового вида

Еще раз отметим, что ПМП оптимизация в компиляторе GCC работает до распределения регистров. Поэтому далее в данном разделе под словом “регистр” будет подразумеваться псевдо-регистр на этапе работы алгоритма ПМП, а не аппаратный регистр.

Опишем новые требования к циклу, под которые попадает значительно большее количество циклов в реальных приложениях. Цикл по-прежнему должен содержать только один базовый блок, и не иметь внутри вызовов и других инструкций с побочными эффектами. Однако для управляющей части и целочисленного регистра-счетчика цикла допускается большая свобода, чем в *do-loop* циклах.

Выход из цикла должен происходить при достижении равенства в сравнении счетчика с инвариантной в цикле величиной, то есть, либо с константой, либо с регистром, значение которого установлено до цикла и не меняется в теле цикла. Внутри цикла счетчик должен изменяться ровно одной инструкцией, которая прибавляет или вычитает ненулевую константу. Другие инструкции могут использовать счетчик, но не могут менять его значение. Получается, что счетчик цикла всегда соответствует элементам некоторой арифметической прогрессии.

Для поддержки циклов нового вида были внесены значительные изменения в оптимизацию поворотного модульного планирования. Необходимо отметить, что циклы, подходящие под новую модель, могли встретиться не только на платформе ARM, но и на других платформах. Примером такой платформы является распространенная на персональных компьютерах архитектура *x86-64*, которая является одной из приоритетных платформ для компилятора GCC. В связи с этим, в процессе нижеописанных работ проводились дополнительные проверки корректности и качества кода при компиляции для *x86-64*.

4.4. Корректировка конечного значения счетчика в цикле

В первую очередь опишем, как была решена проблема, возникшая в пункте 4.2. Часть инструкций с помощью копирования выносится в пролог и эпилог. После этого преобразования необходимо изменить управляющую часть цикла так, чтобы количество итераций уменьшилось, но все инструкции, использующие счетчик, получили его корректное значение. Будем считать, что построено новое расписание цикла, все инструкции поделены на стадии, стадии нумеруются с нуля. Пусть STAGE — номер стадии, на которую в новом расписании попала инструкция сравнения счетчика с финальным значением FIN, которое может быть как регистром, так и константой. Инструкции со стадии STAGE будут скопированы в эпилог ровно STAGE раз. Сама инструкция сравнения не должна копироваться в эпилог, ее результат не будет там использоваться, поскольку в эпилог не выносится инструкция ветвления. Тем не менее, ровно на STAGE итераций раньше необходимо выполнить выход из нового цикла, чтобы инструкции в эпилоге получили корректное значение счетчика.

Для корректирования управляющей части цикла необходимо сделать следующее: в инструкции сравнения, нужно заменить FIN на новое значение по формуле FIN2 = FIN — STAGE * STEP, где STEP — шаг изменения счетчика на каждой итерации. В случае, когда FIN представляет собой регистр, для нового значения FIN2 необходимо перед циклом завести новый регистр, в котором произвести соответствующие вычисления. Если же FIN — константа, которая записывается в виде непосредственного значения в инструкции сравнения, необходимо заменить эту константу на новую. Иногда новая константа FIN2 не может быть записана в виде непосредственного значения, так как на ARM набор таких констант ограничен специфическим образом. В этом случае для хранения такой константы лучше также выделить отдельный регистр, вместо того, чтобы на каждую итерации цикла получать константу с помощью нескольких инструкций.

4.5. Подсчет числа итераций цикла во время выполнения

Как уже было описано ранее, в случае, когда нет возможности определить количество итераций цикла во время компиляции, алгоритм поворотного модульного планирования копирует исходную версию цикла, прежде чем изменить ее. Получается, что в коде есть две версии цикла, кроме того, в код добавляется проверка условия на число итераций. Если итераций оказывается недостаточно, чтобы заполнить пролог и эпилог, и хотя бы один раз выполнить тело оптимизированного цикла – происходит переход на исходную версию цикла. В обычном случае, когда итераций много, будет выполняться оптимизированная версия цикла. Рассмотрим, как выглядит это условие, при котором должна выполняться оптимизированная версия цикла. Число итераций цикла должно быть строго больше чем число стадий в построенном расписании цикла. В случае *do-loop* цикла число итераций можно легко

узнать, зная начальное значение счетчика цикла. Тем самым, остается добавить инструкцию сравнения этого регистра с числом стадий (константа, известная в момент компиляции) и переходить на оптимизированную версию цикла только при достаточном количестве итераций.

Для нового типа циклов с постоянным шагом счетчика, ситуация несколько сложнее. Для правильного подсчета числа итераций необходимо учитывать возможные переполнения счетчика цикла, которые могут усложнить способ подсчета количества итераций. Кроме того, переполнения могут сделать цикл бесконечным. В случае, когда переполнений не происходит, число итераций можно подсчитать, поделив разность конечного и начального значения счетчиков цикла на шаг счетчика. Приведем два примера циклов, к которым не применимо такое решение.

```
for (unsigned int x = 0x0; x != 0x6F80919A; x += 0xEDCBA987) ...; //этот цикл имеет 22 итерации
for (i = 0; i <= n; i++) …;
```

Если в теле цикла не изменяются беззнаковые целочисленные переменные i и n, то число итераций в нем равняется n+1. Однако, в случае, когда n представляет собой максимально возможное для заданного типа беззнаковое число, состоящее из всех единиц в двоичной записи, цикл становится бесконечным.

Конечно, подобные ситуации встречаются редко, но необходима их корректная обработка в оптимизации поворотного модульного планирования. Можно было бы просто отбросить циклы, в которых происходит переполнение целочисленных регистров. В процессе изучения того, как отличить такие циклы от обычных, было найден имеющийся в GCC механизм для анализа циклов - функция get_simple_loop_desc, которая позволяет получить выражение, вычисляющее число итераций через значения регистров во время выполнения программы. Для циклов, анализ которых невозможен, компилятор будет отказываться от оптимизации. Оказалось, что на платформах с *do-loop* инструкцией данная функция неприменима для циклов, на которых было выполнено *do-loop* преобразования. Данный случай был рассмотрен отдельно, чтобы не уменьшать множество циклов, для которых применима оптимизация поворотного модульного планирования.

Функция get_simple_loop_desc может определить условия, при которых цикл становится бесконечным. Соответственно, в этом случае до подсчета количества итераций цикла в код добавляется проверка этих условий. В получившемся коде оптимизированная версия цикла выполняется только в случае, когда количество итераций конечно и превосходит число стадий в расписании. В других случаях происходит переход на исходную версию цикла.

4.6. Учет зависимостей по регистру флагов

Некоторые изменения были внесены так же в построение графа зависимостей по данным. Поскольку в *do-loop* циклах регистр-счетчик использовался только в управляющей части цикла, условие выхода из цикла проверялось до начала оптимизации, и далее инструкция сравнения исключалось из анализа. Для поддержки циклов нового вида необходимо было вычислять зависимости, как для инструкции сравнения, так и для инструкции ветвления – последней в теле цикла. Это связано с тем, что теперь было необходимо принимать во внимание другие инструкции, которые могли читать значение регистра-счетчика, или даже менять значение регистра флагов, который используется инструкцией ветвления.

На данном этапе было обнаружено, что для некоторых тестов на платформе *x86-64* происходит генерация некорректного кода. В отличие от ARM, где многие арифметические инструкции имеют две версии — с выставлением флагов и без (например, sub и subs), на *x86-64* почти все арифметические инструкции изменяют флаги. Иногда в новом расписании цикла между сравнением регистра счетчика с финальным значением и инструкцией ветвления, организующей переход в начало цикла, вставлялась другая арифметическая инструкция, которая меняла значение флага нулевого результата. Для исправления данной ошибки была добавлена реализация генерации необходимых ребер в графе зависимостей по данным.

Кроме того, была выявлена и исправлена еще одна ошибка, касающаяся регистра флагов. В некоторых случаях неверно определялся регистр, который порождает зависимость по данным между двумя инструкциями, из-за чего неверно генерировались копии переменных. Стоит отметить, что регистр флагов на платформах ARM и *x86-64* не может быть скопирован, поэтому для зависимостей по этому регистру важно правильно расставить нужные анти-зависимости в графе, чтобы в итоговом расписании не возникало необходимости копировать регистр флагов.

5. Результаты тестирования

Все проделанные работы были выполнены на компиляторе GCC. Использовалась текущая разрабатываемая сообществом версия компилятора 4.7. Сравнивалась производительность тестов на обычной оптимизации –O2 с результатами, полученными с разработанной поддержкой нового вида циклов при включенных опциях –O2, –fmodulo-sched и –fmodulo-sched-allow-regmoves.

Тестирование на платформе ARM выполнялось на тестовых платах EBV Beagle Board и Samsung C110. Обе платы с процессором ARM Cortex-A8. Тестовый набор для *sqlite* показал ускорение на 3%. Тестовый набор *expedite* для библиотеки растеризации *libevas* в среднем показывает такие же результаты, при этом отдельные тесты показывают результаты от замедления

на 3% до ускорения на 4%. Дополнительно было проведено тестирование умножения матриц из целых и вещественных чисел с помощью различных алгоритмов. Для матриц из целых чисел среднее ускорение составило 3%, а для случая чисел с плавающей точкой производительность увеличилась на 1%.

Тестирование на платформе x86-64 проводилось на персональном компьютере с процессором Intel Pentium IV, поддерживающим 64 битное расширение EM64T. Для умножения матриц из чисел с плавающей точкой с помощью различных алгоритмов ускорение составило в среднем 1%. При этом прирост производительности составил 1.5% для самой простой реализации умножения, 2.5% для реализации с временной переменной, и около 3% для алгоритма с транспонированной второй матрицей и алгоритма, где поменяна вложенность внутренних циклов. В то же время реализация методами Maeno Toshinori и Dan Warner замедлилась на 3% и 5% соответственно.

Было проведено тестирование на промышленном наборе тестов SPEC CPU2000[9], в среднем производительность изменилась незначительно. Наибольшее ускорение получено на тесте 300.twolf , оно составило 1.7%. На тестах 252.eon и 178.galgel производительность снизилась на 2%.

6. Заключение

В рамках данной работы был проведен анализ имеющегося в компиляторе GCC алгоритма программной конвейеризации циклов ПМП для выявления возможностей его использования на платформе ARM. По результатам этой работы было обнаружено, что планировщик может работать только с циклами очень узкого класса. Были исследованы возможности отказа от этого ограничения и реализована поддержка имеющимся алгоритмом циклов нового типа. Проделанная работа позволяет применять данную оптимизацию на таких распространенных платформах, как ARM и *x86-64*.

Были сделаны необходимые улучшения, позволяющие алгоритму конвейеризации циклов корректно работать на циклах нового вида. В частности, был улучшен алгоритм создания пролога и эпилога, а также значительно усложнен механизм генерации проверки условий для выполнения оптимизированной версии цикла. Также реализовано внесение в граф зависимостей по данным необходимых изменений для генерации компилятором корректного кода.

По результатам тестирования получено значительное увеличение производительности на некоторых приложениях. Среди дальнейших работ планируется проверить работу алгоритма на других платформах, а также исследовать причины снижения производительности некоторых тестов при конвейеризации циклов.

Список литературы

[1] Веб-сайт Gnu Compiler Collection. http://gcc.gnu.org/

[2] M.R. Garey, D.S. Johnson. “Computers and Intractability: A Guide to the Theory of NP-completeness”. San Francisco: W. H. Freeman & Company Publishers. 1979.

[3] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, Том 16, 2009, стр. 31-50.
[4] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. “Lifetime-sensitive modulo scheduling in a production environment”. Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001.
[5] B.R. Rau. “Iterative modulo scheduling: An algorithm for software pipelining loops”. In Proc. of the 27th Annual International Symposium on Microarchitecture, pp. 63-74. November 1994.
[6] Mostafa Hagog and Ayal Zaks. Swing Modulo Scheduling in GCC. In Proceedings of the GCC Developer's Summit 2004, pp 55-64, Ottawa, Canada.
[7] Веб-сайт Enlightenment Foundation Libraries http://www.enlightenment.org/p.php?p=about/efl
[8] Веб-сайт SQLite http://www.sqlite.org/about.html
[9] Веб-сайт Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/

Loops software pipelining on ARM platform

Roman Zhuykov, ISP RAS, Moscow, Russia zhroma@ispras.ru
Dmitry Melnik, ISP RAS, Moscow, Russia dm@ispras.ru
Ruben Buchatskiy, ISP RAS, Moscow, Russia ruben@ispras.ru

Abstract. This article describes improvements we made in the implementation of swing modulo scheduling (SMS), a well-known software pipelining technique, in the GNU Compiler Collection (GCC) for ARM platform. Prior GCC implementation required a loop being pipelined to conform to the do-loop pattern, which needs a special hardware instruction. However, such hardware instruction is absent on ARM. First we implemented a "fake" do-loop instruction in the ARM backend, which helped us to verify whether GCC SMS implementation is profitable on ARM. Then we designed and implemented support for loops which loop counter varies as an arithmetic progression. In do-loops the loop counter must be used only in control part of the loop, and we allow reading loop counter register by other loop instructions. For such loops we improved the algorithm for creating prologue and epilogue as well as implemented much more complex algorithm of verification conditions for entering performance-optimized version of the loop. Also we made necessary changes in data dependency graph to generate correct code. When dependency graph is built we create additional anti-dependencies between instructions which use flag register. The resulting performance improvement is 3-4% for selected test applications on ARM platform. For x86-64 platform, performance results are mostly neutral, with exception of 2-3% improvement on matrix multiplication tests.

Keywords: program optimizations; loop software pipelining; swing modulo scheduler; ARM.

References

[1]. Gnu Compiler Collection website. http://gcc.gnu.org/
[2]. M.R. Garey, D.S. Johnson. "Computers and Intractability: A Guide to the Theory of NP-completeness". San Francisco: W. H. Freeman & Company Publishers. 1979.
[3]. A. Belevantsev, D. Zhurikhin, D. Melnik. Kompilyatsiya programm dlya sovremennykh arkhitektur [Program compilation for modern architectures], Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 31-50 (in Russian).
[4]. J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. "Lifetime-sensitive modulo scheduling in a production environment". Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001. doi: 10.1109/12.910814
[5]. B.R. Rau. "Iterative modulo scheduling: An algorithm for software pipelining loops". In Proc. of the 27th Annual International Symposium on Microarchitecture, pp. 63-74. November 1994. doi: 10.1145/192724.192731
[6]. Mostafa Hagog and Ayal Zaks. Swing Modulo Scheduling in GCC. In Proceedings of the GCC Developer's Summit 2004, pp 55-64, Ottawa, Canada.
[7]. Enlightenment Foundation Libraries website http://www.enlightenment.org/p.php?p=about/efl
[8]. SQLite website http://www.sqlite.org/about.html
[9]. Standard Performance Evaluation Corporation website. http://www.spec.org/cpu2000/

Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM

Роман Жуйков <zhroma@ispras.ru>,
Дмитрий Плотников <dplotnikov@ispras.ru>,
Мамикон Варданян <mamikon@ispras.ru>

Аннотация. В данной работе описывается созданная в ИСП РАН система для автоматической настройки параметров компиляции, разработанная для использования на встраиваемых платформах. Система включает в себя средства анализа полученных результатов. С помощью этих средств были выявлены недочеты в работе компилятора GCC, приводящие к генерации неоптимального кода для платформы ARM. Были выявлены причины генерации неоптимального кода, а также рассмотрены и реализованы различные подходы к улучшению оптимизаций. Также была проведена работа по улучшению других оптимизаций компилятора GCC, недостатки которых были обнаружены при ручном анализе ассемблерного кода. Получено значительное увеличение производительности выбранных тестовых приложений на платформе ARM.

Ключевые слова: генетические алгоритмы; оптимизация программ; векторизация; предварительная загрузка данных; архитектура ARM.

1. Введение

В современном мире широкое распространение получили различные встраиваемые системы. В связи с этим все больше возрастают требования к качеству кода для встраиваемых процессоров. Одним из способов получения более качественного кода является адаптация имеющихся многоплатформенных компиляторов к архитектурным особенностям встраиваемых платформ.

Цель данной работы – рассмотреть потенциальные возможности такой адаптации на примере компилятора GCC[1] с открытым исходным кодом и процессоров архитектуры ARM. Данное исследование выполняется в рамках проводимых в отделе компиляторных технологий ИСП РАН работ по оптимизации программ для современных вычислительных архитектур[2]. Планируется добиться генерации более качественного кода для встраиваемых систем без внесения значительных изменений в инфраструктуру компилятора GCC.

GCC является многоплатформенным компилятором и содержит множество оптимизаций и их параметров, которые могут влиять на итоговый машинный код. В основном, большинство оптимизаций настроено на получение качественного кода для платформ x86 и x86-64. Учитывая сложность GCC, существуют большие возможности для его улучшения, путем настройки компилятора для одной конкретной архитектуры. Улучшения могут быть нескольких видов. Во-первых, могут быть существенные недостатки при генерации кода для менее популярных архитектур. Исправления для них обычно требуют добавления новых шаблонов инструкций или настройки платформозависимых параметров. Во-вторых, машинно-независимые оптимизации могут быть не оптимально настроены для определенной архитектуры, в этом случае для достижения лучшего результата может быть изменено их поведение по умолчанию. Например, при распределении регистров могут использоваться разные ограничения для различных архитектур. Наконец, есть возможность написать новую оптимизацию для учета особенностей конкретной архитектуры. Например, в ИСП РАН была реализована поддержка спекуляции в планировщике в соответствии с особенностью Intel Itanium. Однако, такой вид улучшения может потребовать большое количество времени и ресурсов.

Производительность сгенерированного кода имеет первостепенное значение для компиляторов. Для достижения хорошей производительности обычно недостаточно реализовать платформозависимые оптимизации, также необходима настройка уже существующих платформонезависимых оптимизаций. Настройка для архитектуры ARM выполнялась с использованием нескольких тестовых приложений, для каждого из которых была сделана попытка добиться максимально возможной производительности на целевой платформе. Для улучшения работы компилятора с тестовыми приложениями можно использовать ручной анализ ассемблерного кода. Такой анализ требует большого количества времени, поэтому мы создали инструмент, который позволяет выявить часть недостатков в оптимизационных платформонезависимых преобразованиях, приводящих к генерации неоптимального кода на выбранных тестовых приложениях.

APTT (Automatic Performance Tuning Tool, Инструмент для Автоматической Настройки Производительности) позволяет автоматически подбирать лучшие опции и параметры компилятора для заданной программы и наборов входных данных. Разработанный инструмент основан на ACOVEA[3] (Analysis of Compiler Optimizations via an Evolutionary Algorithm), которая позволяет искать лучшие опции при помощи генетических алгоритмов, но имеет ряд недостатков. Они были исправлены одновременно с расширением функциональности.

В данной статье описываются найденные в компиляторе GCC недочеты и выявляются причины генерации неоптимального кода, а также рассматриваются различные подходы к улучшению оптимизаций.

Дальнейшее изложение построено следующим образом. В разделе 2 описываются используемые тестовые приложения. Раздел 3 рассказывает о разработанной системе для автоматического подбора опций компиляции. Раздел 4 посвящен проделанной работе над оптимизацией преобразования ветвлений в GCC. В разделе 5 описываются найденные в оптимизации предварительной загрузки данных недочеты и способы их исправления. Раздел 6 содержит описание изменений, касающихся автоматической векторизации циклов.

2. Тестовые приложения

Для тестирования оптимизаций компилятора GCC были выбраны три библиотеки с открытым исходным кодом. Эти библиотеки широко используется на современных встраиваемых системах с процессорами ARM. Примерами таких устройств являются мобильные телефоны и планшеты, пользовательские оболочки которых реализованы с использованием функциональности нижеописанных библиотек.

Первая библиотека – *libevas*. По результатам профилирования, на выполнение кода этой библиотеки уходит более 95% времени выполнения набора тестов *expedite* для EFL. Enlightenment Foundation Libraries [4] - набор свободно распространяемых графических библиотек для оконного менеджера Enlightenment. Используемый нами набор тестов *expedite* был урезан для оптимизации времени тестирования и получения большей стабильности результатов, без уменьшения покрытия всех главных функции *libevas*, для которых наиболее важно улучшение производительности.

Еще одна библиотека для тестирования – *sqlite* [5]. Она представляет собой легковесную встраиваемую реализацию реляционной базы данных, которая может использоваться для хранения пользовательской информации во встраиваемых системах. В качестве тестов для *sqlite* использовался набор из 20 тестов, содержащий большую часть операций языка SQL – включая добавление, изменение, поиск и удаление данных, и разнообразное выполнение комбинаций этих операций в виде единой транзакции. Для уменьшения влияния накладных расходов при работе с базой данных использовалась виртуальная файловая система в оперативной памяти.

В качестве третьего тестового приложения использовался *webkit* [6]. Это инфраструктура с открытым исходным кодом для отображения веб-страниц. Сейчас он используется в нескольких браузерах и других программах, нуждающихся в отображении и редактировании содержимого веб-страниц. Для тестирования производительности использовался набор тестов SunSpider для JavaScript, включенный в репозиторий *webkit*. К сожалению, рассмотренные нами тесты, измеряющие время визуализации HTML-кода, оказались очень нестабильными. Время различалось до 10-15% от запуска к запуску, несмотря на все попытки уменьшить накладные расходы, например организацию чтения страницы из оперативной памяти.

Мы адаптировали все библиотеки вместе с их тестами для использования с разработанным инструментом APTT, который описан в следующем разделе.

3. Реализация автоматизированного инструмента настройки

Для автоматического выявления недостатков в оптимизационных платформонезависимых преобразованиях, была решено создать систему, которая позволяет автоматически подбирать лучшие опции и параметры компилятора для заданной программы и наборов входных данных. Такая система позволяет улучшать работу компилятора следующим образом. Предположим, какие-то опции или параметры компиляции увеличивают производительность некоторого набора приложений на данной платформе. Можно оценить влияние найденных опций на генерируемый код в сравнении с оптимизацией по умолчанию, и соответствующим образом изменить поведение компилятора для этой платформы. Например, если из результатов тестирования следует, что отключение какой-либо машиннонезависимой оптимизации влечет увеличение производительности, то необходимо отключить данный оптимизационный проход на используемой платформе, или выяснить и устранить причины, по которым этот проход приводит к генерации неоптимального кода.

В инструменте ACOVEA реализован поиск лучшего набора опций компиляции с использованием генетического алгоритма. Основная идея генетического алгоритма использует схему естественного отбора. Сначала создается первое поколение особей – множество произвольных случайных наборов опций, для которых измеряется производительность. Далее среди наборов, которые позволили получить лучшие результаты тестов, производится скрещивание — обмен случайной частью набора опций. Из полученного таким образом второго поколения особей для скрещивания вновь выбираются только наборы опций, показавшие лучшую производительность. После создания заданного пользователем количества поколений поиск считается завершенным и в качестве результата эволюции предлагается тот набор опций, который дает самую лучшую производительность.

Разработанная нами система APTT выполняет эволюционный поиск параметров компилятора с использованием модифицированной версии ACOVEA и обеспечивает удобную инфраструктуру для сборки, запуска приложений, проверки и анализа результатов настройки. С помощью сбора дополнительной информации об использовании опций компилятора во всех поколениях генетического алгоритма получаем более детальные данные для анализа. Кроме того, была организована поддержка кросс-компиляции, что позволяет производить работу генетического алгоритма отбора и компиляцию тестовых приложений на одном компьютере, а запуск тестов на другом компьютере, например, тестовой плате с процессором ARM.

Инструмент APTT состоит из двух основных скриптов - aptt-host и aptt-target. Первый скрипт выполняет основные функции по поиску и фильтрации опций: подготовка структуры каталогов, запуск генетического алгоритма, анализ полученных результатов. Для реализации возможности кросс-компиляции, исходные тексты ACOVEA были модифицированы так, чтобы не производилась сборка и запуск тестового приложения напрямую. Теперь строка опций компиляции передается специальному скрипту host-acovea-proxy. Это скрипт выполняет кросс-компиляцию приложения с указанными опциями и посылает сообщение, что тестовое приложение готово к запуску. Сообщение обрабатывается скриптом aptt-target, запущенным на тестовой плате. После выполнения набора тестов aptt-target отсылает сообщение с результатами скрипту host-acovea-proxy, он в свою очередь передает результаты генетическому алгоритму.

Далее в этом разделе описывается еще несколько особенностей системы APTT, которые были реализованы для удобства ее использования при настройке компилятора GCC для платформы ARM.

3.1. Единая структура для развертывания приложений

Система для автоматической настройки состоит из одного многоядерного x86 компьютера для кросс-компиляции необходимых приложений и нескольких тестовых плат, связанных таким образом, что все устройства имеют один общий каталог через сетевую файловую систему NFS. APTT использует определенную структуру каталогов для хранения оригинальных источников приложения, общих ресурсов (библиотек/инструментов, которые сами не являются предметом для оптимизации, но необходимы целевому приложению для запуска), скриптов и каталогов для сборки, запуска приложения и хранения информации о его профиле (если включен запуск с профилированием). По своей структуре данная организация каталогов имеет аналогию с наследованием в объектно-ориентированных языках программирования, позволяя выносить общие действия и настройки на верхние уровни. Для того, чтобы добавить новые приложения для настройки, необходимо скопировать структуру каталогов из шаблона, после чего настроить файлы конфигурации и сценариев для конкретного приложения. Такая структура облегчает добавление новых тестов для настройки и позволяет реализовать часто выполняемые задачи в системе настройки.

3.2. Параллельная сборка и выполнение

Поддержка параллельной компиляции и выполнения значительно ускоряет процесс работы генетического алгоритма. Параллелизм используется на двух уровнях. Во-первых, APTT позволяет компилировать приложения во время ожидания результатов выполнения ранее скомпилированного приложения. Во-вторых, допускается использовать несколько тестовых плат в процессе работы, запуская целевые приложения одновременно на всех платах.

Еще одной особенностью является то, что тестовые платы, используемые для настройки, не обязательно должны быть одной и той же модели, одного производителя или с одинаковой скоростью процессора. Параллелизм на уровне плат осуществляется на уровне популяций генетического алгоритма, и напрямую сравниваются друг с другом только результаты производительности, полученные в одной популяции, то есть на одной тестовой плате. Эволюция происходит независимо в каждой популяции. Однако заложена возможность миграции между популяциями, которые позволяют лучшим комбинациям опций GCC из каждой популяции распространять себя в другие популяции, таким образом, продолжая их конкуренцию с "местными видами" из этой популяции. Обычно рекомендуется использовать от 2 до 4 популяций для оптимизации. Для оптимизации нагрузки количество популяций должно быть кратно числу используемых тестовых плат. В наших тестах мы провели настройку приложений на трех различных тестовых платах (TI OMAP 3540, EBV Beagle Board и Samsung C110, все с процессором ARM Cortex A8) и результаты, полученные по итогам оптимизации, были корректны и воспроизводимы на любой из трех плат.

3.3. Поддержка компиляции с профилированием

Структура APTT позволяет с его помощью настраивать приложения с учетом оптимизаций с профилированием, а не только со статическими оптимизациями. Оптимизации с профилированием включаются в GCC следующим образом: сначала приложение компилируется с опцией -fprofile-generate, затем при исполнении оно сохраняет информацию о профиле приложения (например, считает выполнения циклов, вероятности выполнения базовых блоков/путей потока управления и т.д.) в .gcda файлы, которые, как правило, находятся в каталоге для сборки приложения. После этого приложение перекомпилируется с опцией -fprofile-use, используя ранее собранную информацию о профиле. В автоматических инструментах оптимизация с профилем приводит к затратам двойного времени, необходимого для оценки того же количества особей. Задача системы управления очередью в APTT —позволить чередование выполнения двух этапов (сбора профиля и окончательной оценки) в различных каталогах сборки так, чтобы свести к минимуму время простоя тестовых плат, потому что обычно время запуска теста превышает время сборки приложения.

3.4. Быстрая перекомпиляция

Чтобы перекомпилировать приложение от запуска к запуску, допускается создание отдельного скрипта rebuild-pool, который должен перекомпилировать приложение с новыми опциями. Иногда тестовые приложения содержат большое количество кода и их компиляция с нуля с новыми опциями занимает длительное время. Если это время превышает время запуска теста на тестовой плате, для ускорения процесса тестирования

можно изучить профиль выполнения тестового приложения и определить файлы, содержащие функции, выполнение которых занимает большую часть времени работы приложения. Обычно количество таких файлов достаточно мало по сравнению с размером всего приложения. После чего можно внести в скрипт rebuild-pool только необходимые файлы. Остальные файлы приложения при этом будут всегда скомпилированы с опциями по умолчанию. Такой подход для тестового приложения *webkit* позволяется сократить время перекомпиляции с 15 до 3 минут, обеспечивая практически те же результаты оптимизации по итогам работы генетического алгоритма.

3.5. Сокращение итоговой строки опций компилятора

Набор флагов, полученных в результате работы генетического алгоритма, может быть избыточным: некоторые флаги могут в контексте других флагов не изменять поведение компилятора, но они будут по-прежнему присутствовать в результирующей строке опций. Мы написали программу, которая пытается уменьшить количество флагов для лучших особей за счет удаления тех опций, присутствие которых не влияет на итоговые бинарные файлы. Она пытается достичь этого путем последовательного удаления опций по одной, этот процесс может занять достаточно длительное время.

После использования инструмента на наших тестах мы смогли сократить лучшие по производительности наборы опций до не более чем 40 значимых опций, несмотря на то, что полный набор опций GCC для настройки состоит более чем из 200 опции. Существует потенциальная возможность еще сильнее сократить полученную строку опций за счет исключения из нее опций, которые мало влияют на производительность. Однако данный метод требует множества запусков целевого приложения на тестовых платах для определения точного вклада каждой из опций в итоговую производительность, а также, в некоторых случаях, ручной оценки существенности данного вклада.

3.6. Инструменты анализа результатов

Для облегчения анализа результатов при автоматической настройке мы разработали несколько скриптов – например, анализатор результатов эволюции и графический инструмент для проверки процесса эволюции. Скрипты используют записанные в виде файлов результаты запуска, в которых для каждой строки опций компиляции хранится результат ее производительности, а также номер текущего поколения и популяции.

Первый скрипт направлен на изучение булевых опций, которые могут иметь только два состояния - включена либо выключена. Скрипт по файлу с результатами всех запусков строит таблицу с опциями и параметрами GCC, упорядоченными по их полезности, которая рассчитывается как средняя производительность запусков, при компиляции которых использовалась данная. Кроме того, он показывает, сколько раз встречается каждая опция и

вычисляет ее среднее положение в логе запуска. Первое значение показывает, насколько часто встречается данная опция у особей, а второе показывает, насколько "зрелой" является каждая опция. Большее среднее значение позиции означает, что опция имеет тенденцию чаще появляться в конце эволюционного процесса. Чем больше эти два значения для опции, тем значимее выгода при использовании этой опции. Кроме того, этот инструмент помогает диагностировать те опции или параметры, которые вызывают ошибки при компиляции, либо приводят к генерации некорректного кода. При неудачном выполнении тестовой программы ставится заведомо самая плохая оценка производительности у данной особи (например, время выполнения 10^9 секунд). Опции, вызывающие ошибки, могут быть легко идентифицированы по их низкому положению в таблице результатов, отсортированной по среднему времени выполнения особей с такой опцией. Необходимо заметить, что такие опции или их комбинации незначительно влияют на работу генетического алгоритма, поскольку быстро исчезают в процессе эволюции.

Второй скрипт предназначен для проверки эволюции числовых параметров. Он представляет собой графический инструмент, который по информации о запусках тестовой программы генерирует некоторое количество веб-страниц с набором диаграмм. Они отображают, как распределение значений параметров варьируется между поколениями и популяциями для каждого настраиваемого параметра, что позволяет увидеть прогресс генетического алгоритма по поколениям. Наблюдение за процессом эволюции в такой визуальной форме помогает разработчику выбрать значения параметров, которые обеспечивают последовательное улучшение производительности.

Также, как часть APTT, мы разработали скрипт, который выбирает заданное число лучших строк опций и параметров компилятора, среди найденных в процессе эволюции, и перепроверяет их достоверность на указанной тестовой плате. После чего генерируется отчет в формате, совместимом с Excel, в котором показаны результаты работы тестов, в том числе относительный прирост производительности по сравнению с базовой оптимизацией –O2.

По результатам работы APTT с компилятором GCC на платформе ARM на используемых тестах были выбраны оптимизации компилятора GCC для дальнейшего исследования и ручного анализа. В следующих разделах рассматриваются самые важные улучшения, произведенные в оптимизациях компилятора GCC.

4. Преобразование ветвлений

Для тестового приложения *sqlite* инструмент APTT показал что опция -fno-if-conversion дает значительный прирост производительности. Данная опция выключает используемую по умолчанию оптимизацию преобразования ветвлений[7], при которой условные переходы удаляются, а все команды в базовых блоках, выполнение которых зависело от этого перехода, защищаются соответствующим предикатом. Преобразование применяется

только к тем базовым блокам, которые не содержат вызовов функций и длинных цепочек зависимостей по данным. Такое преобразование уменьшает количество сбросов конвейера и дает другим оптимизациям (например, планировщику инструкций) большую свободу в преобразовании кода и нахождении параллелизма на уровне инструкций.

Преобразование ветвлений требует поддержки со стороны процессора: необходимо наличие условной формы инструкций, которые выполняются, только если предикат выполнен, и работают как инструкция nop в противном случае. На платформе ARM любая инструкция может быть сделана условной по значению флагов. Например, moveq выполнит mov только если установлен флаг нулевого результата.

Были исследованы причины того, что эта оптимизация приводит к снижению производительности на ARM. Изучение получаемого ассемблерного кода показало, что проблему можно свести к следующему примеру:

```
int f(int a) {
 int z;
 if (a)
  z = 0x66667777;
 else
  z = 0x99998888;
 return z;
}
```

а)

```
ldr   r3, .L4
ldr   r2, .L4+4
cmp   r0, #0
moveq r0, r2
movne r0, r3
bx    lr
```

б)

```
cmp   r0, #0
ldrne r0, .L4
ldreq r0, .L4+4
bx    lr
```

в)

Рис 1. Пример неоптимального преобразования ветвления. а) Исходный код на C; б) ARM-ассемблер для -O2; в) ARM-ассемблер для -O2 -fno-if-conversion

В коде ассемблера на рис. 1*(б)* две загрузки из памяти выполняются всегда (без предикатов), хотя в дальнейшем используется результат только одной из них. В коде на рис. *1(в)* такой недостаток отсутствует, поэтому он выполняется быстрее. Причина создания данных двух загрузок – длинные целые константы в исходном коде на C.

Опция –fif-conversion включает в GCC оптимизационные проходы *ce1* и *ce2*, которые выполняются до распределения регистров. Аналогичный оптимизационный проход *ce3*, включаемый опцией –fif-conversion2, выполняется после распределения регистров. Причина изучаемой проблемы была выявлена в оптимизационном проходе *ce1*. Используется несколько методов преобразования ветвлений, и один из них – это замена присваиваний из базовых блоков ветвления в одно присваивание, которое использует RTL-выражение if-then-else. Впоследствии данное выражение преобразуется в

условную команду пересылки, как в примере выше. Компилятор пытается предсказать, будет ли возможно создать условную команду пересылки. Если аргумент – длинная константа, которая не может быть подставлена в инструкцию как непосредственное значение, простая условная команда пересылки не подходит, и используется безусловная загрузка длинной константы с последующей условной пересылкой из регистра. Для вышеописанного примера, RTL-код выглядит следующим образом:

```
r137:SI=0xffffffff99998888
r138:SI=0x66667777
cc:CC=cmp(r135:SI,0x0)
r133:SI={(cc:CC==0x0)?r137:SI
:r138:SI}
```

а)

```
r3:SI=0x66667777
r2:SI=0xffffffff99998888
cc:CC=cmp(r0:SI,0x0)
(!cc:CC) r0:SI=r2:SI
 (cc:CC) r0:SI=r3:SI
```

б)

Рис 2. RTL-код при преобразовании ветвлений
а) после прохода ce1 ; б) после прохода split2

В этом коде, выражение “r133:SI={(cc:CC==0x0)?r137:SI:r138:SI}” описывает условную команду пересылки, которая преобразуется в RTL-код на рис. 2*(б)* оптимизационным проходом *split2*.

Если включен только один оптимизационный проход *ce3*, работающий после распределения регистров, то к моменту его работы компилятор уже создает необходимые инструкции загрузки длинных констант в регистр и может их преобразовать в условную форму.

Очевидным решением кажется отключение ранних оптимизационных проходов преобразования ветвлений и включением только *ce3* прохода с помощью опций –fno-if-conversion –fif-conversion2. Однако, данная комбинация отключит часть полезных преобразований, выполняемых оптимизационным проходом *ce1*. Например, *ce1* может преобразовать выражение “x > 0 ? x : 0”, которое обычно требует трех инструкций (сравнение, условное ветвление и пересылка) в инструкцию “bicle r8, r0, r0, asr #31” (ее семантика такова: очистить в регистре r0 те биты, которые равны 1 во втором аргументе, и сохранить результат в регистре r8; здесь второй аргумент равен r0 asr #31, который из определения арифметического сдвига равен 0 либо 0xffffffff, в зависимости от знака числа в регистре r0).

Решение о том, следует ли использовать промежуточный регистр при загрузке константы вне условной пересылки, принимается в функции emit_conditional_move. Но для архитектуры ARM, решение иногда оказывается неверным, например в вышеописанном случае. Данный недочет был исправлен, причем были сохранены преимущества оптимизаций выполненных проходом *ce1*, но с защитой от создания безусловных загрузок. До вызова функций emit_conditional_move производится проверка

необходимости использования промежуточного регистра, и если такая необходимость есть, то преобразование на оптимизационном проходе *ce1* отменяется, и только во время работы прохода *ce3* загрузки преобразуются в предикатную форму. Описанные изменения включаются с помощью опции -fif-conversion-no-unconditional-moves.

Результаты тестирования показали, что с введенной опцией компилятор для некоторых тестов создает улучшенный код, и не было найдено тестов, на которых наблюдается снижение производительности. Таким образом, мы улучшили работу на платформе ARM платформонезависимой оптимизации преобразования ветвлений в компиляторе GCC. Ускорение для тестовых приложений оказалось небольшим, для *sqlite* оно составило 0.5%. Причина этого в том, что подходящие условные инструкции не встречаются в горячих местах кода в исследуемых тестовых приложениях.

5. Предварительная загрузка данных

Предварительная загрузка данных (prefetching)[8] – технология, позволяющая ускорить работу программ за счет уменьшения количества промахов кэша. На многих современных архитектурах есть инструкция предварительной загрузки. Эта инструкция фактически является подсказкой процессору, она не влияет на логику работы программы. Предварительная загрузка работает почти как обычная инструкция загрузки из памяти, с той лишь разницей, что данные не попадают ни в один регистр. Запускается процесс передачи данных из памяти в кэш, и через некоторое число тактов (в это время выполняются другие инструкции) данные оказываются в кэше. При последующем обращении к этим данным пропадает необходимость длительного ожидания работы с памятью. Один из вариантов использования предварительной загрузки, предлагаемый в GCC, это использование функции __builtin_prefetch. На платформе ARM ее вызов будет заменен на инструкцию предварительной загрузки pld для адреса, переданного функции в качестве параметра.

Также в GCC есть оптимизационный проход, подключаемый с помощью опции -fprefetch-loop-arrays, который автоматически добавляет операции предварительной загрузки для обращений к элементам массива в циклах. В нем происходит анализ всех обращений в памяти происходящих на одной итерации цикла. В зависимости от множества параметров целевой платформы часть обращений выбирается для предварительной загрузки. Поскольку данные в кэше обновляются блоками по размеру линии кэша, обычно в оптимизационном проходе производится разворачивание цикла, чтобы для одного обращения к памяти в терминах исходного цикла было необходимо выполнить одну инструкцию предварительной загрузки на каждой итерации развернутого цикла.

5.1. Настройка параметров предварительной загрузки данных

Оптимизация предварительной загрузки данных в GCC имеет несколько параметров, контролирующих ее работу. Параметры определяют аппаратные характеристики кэша, такие как размер линии кэша, количество одновременно выполняемых загрузок, время задержки при передаче данных из памяти в кэш второго уровня и другие. Анализ результатов тестирования библиотеки *libevas* с помощью APTT показал, что эти параметры, и их интерпретация в компиляторе для генерации кода, сильно влияют на производительность.

В первую очередь, следует отметить два параметра оптимизации предварительной загрузки данных, появившиеся в GCC версии 4.5. Это prefetch-min-insn-to-mem-ratio и min-insn-to-prefetch-ratio. Они были созданы в первую очередь для более консервативной оптимизации предварительной загрузки данных на некоторых тестах набора SPEC CPU 2000 [9] на процессорах архитектуры x86. Первый параметр устанавливает ограничение на минимальное соотношение числа инструкции в цикле к числу обращений к памяти в нем, подразумевая, что если нет инструкций для выполнения на процессоре во время ожидания операций с памятью, то не будет улучшения производительности от предварительной загрузки. Но для многих циклов это не так, например обычный цикл копирования массива в памяти почти не содержит инструкций для долгого выполнения на процессоре, но может быть значительно ускорен за счет предварительной загрузки данных.

Второй параметр отключает предварительную загрузку в циклах с неизвестным числом итераций, если соотношение между числом предварительных загрузок и общим числом инструкций превосходит значение этого параметра. Поскольку без использования профилирования число итераций цикла редко оказывается известно, этот параметр успешно отключает предварительную загрузку на библиотеке *libevas*, устанавливая слишком высокое ограничение (значение по умолчанию равно 10). Можно сказать, что оба эти параметра оказываются неуместными, по крайней мере, на приложениях, исследованных в данной работе. Тестирование с помощью APTT показало, что оптимальным значением этих двух параметров на платформе ARM является ноль, это позволяет использовать все возможности оптимизации предварительной загрузки.

5.2. Оценка дистанции предварительной загрузки

С помощью APTT было проведено исследование влияния параметра prefetch-latency на производительность тестовых приложений. Данный параметр должен означать количество тактов процессора, которое необходимо для предзагрузки данных из памяти в случае, когда эти данные отсутствуют в кэше. Теоретически, оптимальное значение prefetch-latency должно зависеть от конкретного используемого аппаратного обеспечения, но не от запускаемых программ. Однако, результаты работы APTT показали, что

оптимальное значение данного параметра значительно отличается для выбранных тестовых приложений.

Оказалось, что в оптимизации предварительной загрузки prefetch-latency используется для вычисления дистанции предварительной загрузки. Дистанцией предварительной загрузки называется, в случае платформы ARM, число байт, добавляемое к аргументу инструкции предварительной загрузки pld по сравнению с обычной загрузкой ldr для обращения к той же области в памяти на той же итерации. Для подсчета дистанции предварительной загрузки компилятор пытается оценить время, требуемое для выполнения одной итерации цикла. Далее, зная шаг у каждого из обращений к памяти и задержку предварительной загрузки prefetch-latency, вычисляется дистанция предварительной загрузки. Ее максимально точное вычисление очень важно, ведь в случае, когда она слишком мала, данные не успеют оказаться в кэше к нужному моменту времени, а в случае, когда дистанция слишком велика, они могут быть стерты из кэша в результате записи туда других данных. Однако, реализованные в GCC вычисления не очень точны при определении времени выполнения одной итерации цикла. В частности, выполняется оценка времени на исходном (без развертывания итераций) теле цикла, после чего полученное значение умножается на коэффициент развертывания. Получается, что инструкции изменения счетчика цикла и ветвления подсчитываются несколько раз, хотя после развертывания большая часть из них будет удалена. В результате компилятор получает дистанцию предварительной загрузки большую, чем это требуется для архитектуры, особенно на коротких циклах. Это создает излишний дефицит кэша и не позволяет настраивать параметр prefetch-latency, поскольку его реальное влияние зависит от размера и шага конкретного цикла.

Механизм оценки дистанции предварительной загрузки был улучшен с помощью подсчета временной оценки для уже развернутого цикла вместо исходного. На простых тестовых циклах выигрыш производительности составляет до 10%. Тем не менее, даже после улучшения, возможно получение более высокой производительности тестов при выборе оптимального значения параметра prefetch-latency с помощью APTT.

5.3. Уменьшение регистрового давления

При анализе ассемблерного кода, сгенерированного при включенной оптимизации предварительной загрузки, было обнаружено, что дополнительный регистр используется как аргумент в каждой инструкции предварительной загрузки pld, в то время как требуемый адрес предварительно загружаемого места в памяти можно легко вычислять относительно аргументов инструкции загрузки ldr, для которой делается предварительная загрузка. Как следствие, требуется значительно больше регистров, причем разворачивание цикла усугубляет эту проблему.

Увеличение числа используемых регистров легко увидеть на следующем примере:

```
                        .L3:                .L3:
for (i=0; i<n; i++)     mov r1, r2, asl     ldr ip, [r2, #0]
  s+=a[i]*b[i];         #2                  add r1, r1, #1
                        ldr r5, [r3, r4]    ldr r4, [r3, #0]
                        ldr r6, [r3, ip]    cmp r1, r5
                        add r2, r2, #1      pld [r2, #116]
                        cmp r2, r8          pld [r3, #116]
                        add r7, r1, r4      add r2, r2, #4
                        add r1, r1, ip      add r3, r3, #4
                        pld [r7, #116]      mla r0, r4, ip,
                        pld [r1, #116]      r0
                        add r3, r3, #4      bne .L3
                        mla r0, r6, r5,
                        r0
                        bne .L3
```

Рис 3. Пример излишнего регистрового давления.
а) Исходный код на C; б) ARM-ассемблер до исправления;
в) ARM-ассемблер после исправления

Удалось выяснить, что источник проблемы – оптимизация отслеживания изменений скалярных переменных (scalar evolution optimization pass), которая анализирует поведение скалярных переменных в циклах. Оказывается, данная оптимизация не позволяет разложить такое выражение, как &a[i] на сумму &a[0] и i, для дальнейшего анализа отдельно каждой части выражения. Этот недостаток является причиной того, что в дальнейшем при оптимизации индукционных переменных цикла оказывается невозможным вычисление адреса &a[i]с использованием того же регистра, что и для обращений к a[i].

Были разработаны улучшения для оптимизации скалярных переменных в циклах. Стало допустимым разложение адресного выражения. Эффект от внесенных изменений не ограничивается влиянием на оптимизацию предварительной загрузки данных, но также позволяет оптимизациям счетчиков цикла находить более полное множество индукционных переменных в циклах, где используются обращения к глобальным массивам или массивам внутри структур.

Улучшение оптимизации дает прирост производительности библиотеки *libevas* на процессоре ARM в среднем на 3.4% и до 11% на некоторых тестах. Кроме того, на 1% ускоряется набор тестов с использованием чисел с плавающей точкой SPEC FP 2000 на архитектуре x86 на процессоре Core 2 с использованием профилирования и следующих опций оптимизации: -O3

-ffast-math -march=native –fsched-pressure –fschedule-insns -fprefetch-loop-arrays. Для целочисленных тестов SPEC INT 2000 производительность не изменяется.

6. Автоматическая векторизация

Отдельно от предыдущих работ был выполнен ручной анализ кода, создаваемого компилятором GCC на платформе ARM при включенной автоматической векторизации[10]. GCC поддерживает автоматическую векторизацию для многих SIMD архитектур, включая технологию ARM NEON[11]. Векторизация не используется по умолчанию на уровнях оптимизации ниже третьего (-O3). Для включения оптимизации с помощью векторных инструкций NEON на платформе ARM необходимо указание следующих опций компилятора: -mfpu=neon –mfloat-abi=softfp –ftree-vectorize. В GCC оптимизационные проходы, производящие векторизацию, работают над внутренним представлением Tree-SSA. Для каждого цикла строится граф зависимостей по данным между инструкциями, в том числе с разных итераций цикла. В этом графе строятся компоненты сильной связности. Компоненты, состоящие из одной вершины-инструкции, означают что действия, выполняемые этой инструкцией на разных итерациях цикла, могут быть выполнены параллельно. В дальнейшем, компоненты сильной связности топологически сортируются. Это позволяет рассмотреть дополнительный параллелизм и возможности для векторизации, например, выполнив разделение цикла или перестановку вложенных циклов.

Были найдены и исправлены два недостатка в реализации автоматической векторизации в компиляторе GCC касающиеся генерации кода для векторной технологии NEON.

6.1. Выравнивание данных

При создании компилятором GCC векторизованной версии цикла, в код добавляются проверки условий, при которых будет выполняться эта векторизованная версия. Например, если цикл производит операцию копирования памяти memcpy(a, b, N), и он автоматически векторизуется для использования векторных инструкций NEON, обрабатывающих четыре четырехбайтовых слова за одну операцию, то для выбора выполняемой версии цикла проверяется условие, что (a | b) & 15 == 0. Данное условие разрешает выполнение векторизованной версии только в том случае, когда оба указателя выровнены по 16 байтам. На некоторых архитектурах (и даже некоторых версиях инструкций NEON, не поддерживаемых сейчас в GCC) на самом деле требуется, чтобы данные в векторных операциях были выровнены по размеру вектора, но инструкция загрузки vldr в NEON требует лишь выравнивания по размеру элемента (4 байта). Это условие выполняется на практике почти всегда, тем самым позволяя чаще выполнять более быструю версию цикла с векторными инструкциями. В таком варианте реализации для большей части циклов в библиотеке *libevas* лишь в редких случаях выполняется медленная

обычная версия цикла, тем самым давая прирост производительности 8.36% на наборе тестов *expedite*.

6.2. Поддержка непосредственных значений в векторных инструкциях сдвига

Оказалось, что GCC для архитектуры ARM не позволяет создание векторных инструкций сдвига с непосредственным значением последнего аргумента – величины сдвига, не смотря на наличие таких инструкций в NEON. Как следствие такого недостатка, компилятор вынужден сначала создавать вектор из четырех одинаковых значений в векторном регистре NEON, и потом использовать векторную инструкцию сдвига. Сгенерированный код автоматически векторизованного цикла получается неоптимальным.

В GCC в архитектурно зависимую часть для ARM был добавлен шаблон для поддержки векторных операций сдвига с непосредственно подставляемым значением величины сдвига. После этих изменений автоматическая векторизация оптимально обрабатывает операции сдвига. Полученные результаты не содержат замедлений, набор тестов *expedite* для библиотеки *libevas* ускоряется в среднем на 0.5%, и до 3% улучается производительность на конкретных тестах.

7. Заключение

Нами был разработан инструмент APTT на базе ACOVEA для автоматической настройки опций компилятора, который использует генетические алгоритмы. При этом функциональность ACOVEA была существенно доработана, что позволило сократить время работы инструмента, например при помощи параллельной компиляции и выполнения. Кроме того, доработки облегчили анализ итогового результата за счет фильтрации строки с опциями.

Используя результаты APTT и ручного анализа кода ассемблера, были разработаны исправления к нескольким оптимизациям компилятора GCC, включая автоматическую векторизацию, предварительную загрузку и преобразование ветвлений. Данные улучшения позволяют компилятору GCC генерировать более качественный код для платформы ARM и значительно увеличивают производительность некоторых тестовых приложений. Например, набор тестов *expedite* для библиотеки *libevas* ускоряется в среднем на 12% при использовании улучшенной версии компилятора GCC. Это означает, что более тонкая настройка компилятора для заданной встраиваемой платформы позволяет добиться генерации более качественного кода.

Следует также отметить, что созданный инструмент APTT полезен не только поиска плохо работающих оптимизаций компилятора, но может также использоваться простыми пользователями для повышения производительности различных приложений. APTT с помощью генетического алгоритма позволяет для заданного приложения и набора тестов получить

опции компиляции, существенно повышающие производительность приложения на используемой платформе. Мы сравнили на платформе ARM производительность выбранных нами тестов с использованием обычного компилятора GCC, а также нового улучшенного варианта GCC, причем с использованием лучшего набора опций, найденного с помощью APTT. Итоговое улучшение производительности для выбранных приложений без использования профилирования составило 30% для *libevas*, 9% для *sqlite*, и 14% для *webkit*. С использованием профилировщика, ускорение *sqlite* составило 28%.

В дальнейшем планируется реализация собственного инструмента для эволюционного поиска, для того, чтобы получить независимость от ACOVEA. Также необходимо дальнейшее усовершенствование инструментов для фильтрации значимых опций и построения отчетов.

Список литературы

[1] Веб-сайт компилятора GCC. http://gcc.gnu.org

[2] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, Том 16, 2009, стр. 31-50.

[3] Веб-сайт ACOVEA. http://www.coyotegulch.com/products/acovea/

[4] Веб-сайт Enlightenment Foundation Libraries. http://www.enlightenment.org/p.php?p=about/efl

[5] Веб-сайт SQLite. http://www.sqlite.org/about.html

[6] Веб-сайт WebKit. http://www.webkit.org

[7] S. Pop, R. Yazdani, Q. Neill “Improving GCC’s auto-vectorization with if-conversion and loop flattening”, Proceedings of the GCC Developers Summit 2010, pp. 89-96.

[8] C. Yang, C. Li, F. Wang “Performance Improvements for GCC Using Architecture Features on IA-64”, Proceedings of the GCC Developers Summit 2005, pp. 199-208.

[9] Веб-сайт Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000

[10] D. Nuzman, A. Zaks. "Autovectorization in GCC - two years later", Proceedings of the GCC Developers Summit 2006, pp. 145-158.

Веб-сайт ARM. Технология NEON.
http://www.arm.com/products/processors/technologies/neon.php

Automatic tuning of GCC optimization passes for ARM platform

Roman Zhuykov, ISP RAS, Moscow, Russia zhroma@ispras.ru
Dmitry Plotnikov, ISP RAS, Moscow, Russia leitz@ispras.ru
Mamikon Vardanyan, ISP RAS, Moscow, Russia mamikon@ispras.ru

Abstract. Modern compilers can work on many platforms and implement a lot of optimizations, which are not always tuned well for every target platform. In the paper we present a tool for automatic tuning of compilation parameters that was developed in ISP RAS. The tool uses genetic algorithm to evolve the best compiler optimization parameters. It supports cross-compilation, parallel build and execution on remote targets, and support for two-tier compilation mode for profile-guided optimizations. Also it provides the tools for automated analysis of the tuning results to help identify the most important compilation parameters.

We used our tool to tune GCC parameters for ARM platform, and demonstrated how it can be used to improve performance of several popular applications. Using the tool's automated analysis of the tuning results, we found several deficiencies in GCC compiler optimizations. The affected optimizations include if-conversion (conversion of conditional branches into predicated form), prefetching (adding of data prefetching instructions in loops), and autovectorization. We identified the reasons for suboptimal code generation, and improved affected optimizations accordingly. Also, we describe a few other improvements that we made to GCC based on the results of manual assembly code analysis. Described improvements to GCC yield significant speedup for chosen test applications on ARM platform.

Keywords: genetic algorithms; program optimizations; vectorization; data prefetching; ARM

References

[1]. Gnu Compiler Collection website. http://gcc.gnu.org/
[2]. A. Belevantsev, D. Zhurikhin, D. Melnik. Kompilyatsiya programm dlya sovremennykh arkhitektur [Program compilation for modern architectures], Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 31-50 (in Russian).
[3]. ACOVEA website. http://www.coyotegulch.com/products/acovea/
[4]. Enlightenment Foundation Libraries website. http://www.enlightenment .org/p.php?p=about/efl
[5]. SQLite website. http://www.sqlite.org/about.html
[6]. WebKit website. http://www.webkit.org
[7]. S. Pop, R. Yazdani, Q. Neill "Improving GCC's auto-vectorization with if-conversion and loop flattening", Proceedings of the GCC Developers Summit 2010, pp. 89-96.
[8]. C. Yang, C. Li, F. Wang "Performance Improvements for GCC Using Architecture Features on IA-64", Proceedings of the GCC Developers Summit 2005, pp. 199-208.
[9]. Standard Performance Evaluation Corporation website. http://www.spec.org/cpu2000
[10]. D. Nuzman, A. Zaks. "Autovectorization in GCC – two years later", Proceedings of the GCC Developers Summit 2006, pp. 145-158.
[11]. ARM website. NEON technology. http://www.arm.com/products/processors/technologies/neon.php

Задача локального распределения регистров во время динамической двоичной трансляции

Кирилл Батузов <batuzovk@ispras.ru>

Аннотация. QEMU — эмулятор аппаратного обеспечения, использующий динамическую двоичную трансляцию в своей работе. В данной статье рассматриваются возможные улучшения алгоритма локального распределения регистров в QEMU. Изменения были произведены на практике, и результаты показали, что дальнейшее улучшение алгоритма возможно только через изменение существующих ограничений на него.

Ключевые слова: распределение регистров, динамическая двоичная трансляция, QEMU.

1. Введение

QEMU — эмулятор аппаратного обеспечения, который позволяет запускать программы и операционные системы, написанные для одних машин, на других. QEMU имеет два режима работы: режим эмуляции системы и режим эмуляции приложения. В первом режиме эмулируется все аппаратное обеспечение, и в полученной виртуальной машине загружается своя операционная система. Во втором режиме эмулируется только процессор. Этот режим позволяет запускать отдельные пользовательские приложения, написанные для одной архитектуры, на другой [1].

В своей работе QEMU использует динамическую двоичную трансляцию. При этом гостевой код дизассемблируется в некоторое внутреннее представление, и все необходимые изменения производятся на уровне внутреннего представления. Затем из внутреннего представления генерируется машинный код для заданной архитектуры.

В данной статье рассматривается задача распределения регистров во время генерации кода по внутреннему представлению.

2. Обзор существующих алгоритмов локального распределения регистров

Задача распределения регистров состоит в назначении переменных внутреннего представления на регистры. Ее можно разбить на две подзадачи:

- выбрать, какие данные должны находиться на регистрах в каждой точке программы, и поместить их на так называемые псевдорегистры — виртуальные регистры, число которых не ограничено, и каждый из которых содержит ровно одно значение,
- поставить в соответствие псевдорегистрам реальные.

При этом если распределение регистров происходит в рамках одного базового блока, то такое распределение называется локальным распределением регистров. Если при распределении учитываются несколько базовых блоков, то распределение регистров называется глобальным [6]. В данной работе рассматривается только задача локального распределения регистров.

Целью алгоритмов распределения регистров обычно является получение как можно более быстро работающего кода. Однако время работы фрагмента программы зависит от многих факторов, и непосредственно измерять его бывает тяжело. Поэтому эффективность алгоритмов распределения регистров часто оценивают по количеству сгенерированных в результате чтений из памяти и записей в память [3].

Задача локального распределения регистров является NP-трудной [5]. Известно несколько алгоритмов ее точного решения: через сведение к задаче линейного программирования [4] либо поиском кратчайшего пути в ациклическом взвешенном ориентированном графе [3]. Однако эти алгоритмы работают достаточно медленно, и вместо них на практике применяются различные эвристики, такие как:

- Первый дальний (Furthest-First, FF) – когда необходимо поместить псевдорегистр на реальный, выбирается любой свободный. Если свободного нет — то освобождается тот, ближайшее использование содержимого которого находится как можно дальше.
- Первый чистый (Clean-First, CF) – аналогично предыдущему, но для освобождения имеют приоритет регистры, значение которых совпадает со значением соответствующей переменной, записанным в памяти.

Эвристический алгоритм FF сочетает в себе высокую скорость работы и получает достаточно близкое к оптимальному решение [4, 5]. Опишем его более подробно.

1. Рассматривается очередная инструкция (код просматривается в прямом порядке, от начала к концу).
2. Для всех псевдорегистров, которые еще не распределены на регистры, выбираются произвольные свободные регистры.

3. Если свободных регистров недостаточно, то некоторые регистры освобождаются. Для освобождения выбираются те регистры, ближайшее использование содержимого которых находится как можно дальше.
4. Алгоритм переходит к следующей инструкции и возвращается на шаг 1.

3. Анализ возможных улучшений существующего в QEMU алгоритма

QEMU обрабатывает гостевой код небольшими кусками — блоками трансляции. Код гостевой системы транслируется сначала во внутреннее представление, а затем из внутреннего представления в машинный код процессора, на котором запущен эмулятор. Внутри блок трансляции состоит из одного или нескольких базовых блоков и представляет собой ациклический участок кода с одним входом.

В качестве внутреннего представления в QEMU выступает массив инструкций и их аргументов. Аргументами могут быть временные, локальные, глобальные переменные или константы. Временные, локальные и глобальные переменные отличаются областью видимости: базовый блок для временных, блок трансляции для локальных и весь код для глобальных. Глобальные переменные используются, в частности, для эмуляции регистров гостевой архитектуры — каждому регистру ставится в соответствие глобальная переменная [1].

Прежде чем для инструкции внутреннего представления может быть сгенерирован код, ее операнды должны быть распределены на регистры. Процесс распределения регистров в QEMU совмещен с генерацией кода: сразу после того, как для операции внутреннего представления были распределены регистры, для нее генерируется машинный код. Таким образом, изменить выбор регистра впоследствии уже невозможно, и алгоритм распределения регистров в QEMU является однопроходным. Распределение регистров в QEMU локальное. Таким образом, переменная может находиться на регистре только внутри базового блока, а в его конце она должна быть записана обратно память.

Алгоритм распределения регистров в QEMU может быть описан следующей последовательностью шагов.

1. Рассматривается очередная инструкция (код просматривается в прямом порядке, от начала к концу).
2. Если инструкция являет вызовом функции, то все глобальные переменные сохраняются в память, регистры, которые могут быть испорчены вызываемой функцией освобождаются, аргументы располагаются на стеке и регистрах в соответствии с соглашениями о вызовах. Алгоритм переходит к шагу 6.

3. Для всех аргументов, которые еще не распределены на регистры, выбирается подходящий (с учетом ограничений целевой архитектуры) свободный регистр. Если подходящих свободных регистров несколько — то берется наиболее приоритетный. Приоритеты регистров статические и выражаются в порядке их просмотра при выборе подходящего [2].
4. Если на шаге 3 свободных регистров не нашлось, то произвольный не используемый в данной инструкции регистр освобождается, а его содержимое сбрасывается в память. Шаг 3 повторяется.
5. Если данная инструкция является последней в базовом блоке, то все глобальные переменные, находящиеся на регистрах, сохраняются в память.
6. После того, как распределение регистров для данной инструкции завершено, для нее генерируется машинный код и записывается в выходной буфер.
7. Если переменная, содержащаяся в регистре, оказалась мертва после данной инструкции, то этот регистр помечается как свободный.
8. Алгоритм переходит к следующей инструкции и возвращается на шаг 1.

Теперь рассмотрим возможности применения в QEMU алгоритмов, описанных в части 2 данной статьи.

QEMU производит распределение регистров во время выполнения программы, поэтому время работы самого алгоритма также важно учитывать. По соотношению времени работы и качества получаемого кода одним из лучших является алгоритм FF. Для его реализации в рамках существующего алгоритма необходимо внести изменения в четвертый шаг алгоритма.

Однако наличие во внутреннем представлении QEMU вызовов функций, которые не рассматриваются в алгоритме FF, накладывает два дополнительных ограничения:

1) вызываемая функция может читать или изменять глобальные переменные, поэтому все они должны быть сохранены из регистров в память,
2) содержимое некоторых реальных регистров может быть испорчено вызываемой функцией в соответствии с соглашениями о вызовах.

В QEMU эти проблемы частично решены. Корректность генерируемого кода обеспечивается шагом 2 алгоритма. Неравноправие регистров, которое вносит вторая проблема, учитывается в приоритетах регистров при распределении на третьем шаге [2].

В данной статье предлагается следующее улучшение: сохранять глобальные переменные в память не перед самим вызовом функции, а сразу после их последнего использования до этого вызова. На границах базовых блоков поступать аналогично.

Данные глобальные переменные обладают тем свойством, что они живы, занимают регистр, но не могут быть использованы без предварительной их записи в память. Таким образом, предлагается производить такую запись как можно раньше, освобождая регистр для других нужд. В случае, когда регистров не хватает, данная эвристика позволяет сэкономить запись в память, поскольку алгоритм выбора регистра для освобождения может выбрать другой регистр, который еще можно использовать. И в случае, когда еще есть свободные регистры, эта эвристика тоже может экономить сбросы регистров в память. Это связано с неоднородностью реальных регистров в QEMU — некоторые из них сохраняют свои значения после вызовов, а некоторые нет. Таким образом, освобожденный эвристикой регистр может оказаться более «удобным» для хранения переменной, чем имеющийся свободный регистр.

В описанным выше алгоритме FF сходная ситуация, когда глобальная переменная жива, но не может быть использована без записи в память, может возникать на границах базовых блоков. Однако в той модели

- все регистры равноправны и любой может быть использован вместо любого другого,
- такие переменные имеют наиболее высокий приоритет при освобождении регистра в случае их нехватки — ни одна переменная не может иметь использование дальше, чем граница базового блока.

Таким образом, необходимости в подобной эвристике в алгоритме FF не было.

4. Изменения в алгоритме распределения регистров в QEMU

В ходе данной работы в алгоритме распределения регистров в QEMU были реализованы две эвристики, а также внесены изменения в анализ времени жизни переменных для поддержки данных изменений.

Анализ времени жизни переменных, который существует в QEMU для удаления мертвого кода, был модифицирован так, чтобы для каждого операнда каждой операции внутреннего представления сохранять, где данный операнд в следующий раз используется. Для этого в нем, при проходе от конца кода к началу, поддерживается массив, в котором для каждой переменной внутреннего представления записано, в какой последней встреченной операции она использовалась. Аналогичным образом для каждой инструкции внутреннего представления определяется ближайший следующий за ней вызов функции.

При распределении регистров для каждого регистра в каждый момент времени поддерживается информация, в какой инструкции внутреннего представления будет следующее использование содержимого данного регистра. Эта информация используется в следующих правилах:

- если в результате нехватки регистров какой-то регистр должен быть освобожден, то выбирается тот регистр, ближайшее использование содержимого которого произойдет как можно позже;
- если в некоторый момент времени регистр содержит глобальную переменную и ближайшее использование этой переменной находится в другом базовом блоке или после вызова функции, то переменная сбрасывается в память, а регистр освобождается;
- если регистр может быть испорчен в результате вызова функции, а ближайшее использование его содержимого произойдет после вызова функции, то содержимое данного регистра сбрасывается в память, а регистр освобождается.

5. Экспериментальные результаты

Для оценки эффективности произведенных изменений в QEMU было реализовано профилирование генерируемых сбросов в память. Для каждого сброса профилировщик определяет причину: нехватка регистров, затирание регистров вызываемой функцией, сброс глобальных переменных перед вызовом функции, сброс глобальных переменных перед концом базового блока.

В таблицах 1 и 2 приведены результаты тестирования. Тестирование проводилось на архитектуре x86, в качестве гостевой архитектуры также использовалась x86. Первые три теста запускались в режиме эмуляции приложения, последний тест представляет собой загрузку операционной системы GNU/Linux в режиме эмуляции системы.

	Нехватка регистров	Конец базового блока	Вызов функции: глобальные переменные	Вызов функции: затираемые регистры	Всего
date	647 (4.5%)	5480 (38.2%)	7640 (53.2%)	582 (4.1%)	14349
bzip2	1469 (7.9%)	6042 (32.5%)	10055 (54.0%)	1041 (5.6%)	18607
linpackc	514 (4.4%)	4455 (37.7%)	6360 (53.8%)	487 (4.1%)	11816
Linux boot	11765 (4.2%)	105556 (37.6%)	149284 (53.1%)	14390 (5.1%)	280995

Таб. 1. Результаты профилирования причин сброса регистров оригинального алгоритма распределения регистров в QEMU.

	Нехватка регистров	Конец базового блока	Вызов функции: глобальные переменные	Вызов функции: затираемые регистры	Всего
date	102 (0.7%)	5535 (39.0%)	8349 (58.8%)	224 (1.6%)	14210
bzip2	317 (1.7%)	6188 (33.6%)	11588 (62.9%)	324 (1.8%)	18417
linpackc	52 (0.4%)	4491 (38.4%)	6978 (59.6%)	179 (1.5%)	11700
Linux boot	1139 (0.4%)	106543 (39.0%)	160513 (58.8%)	4730 (1.7%)	272925

Таб. 2. Результаты профилирования причин сброса регистров измененного алгоритма распределения регистров в QEMU.

В первых трех тестах видно существенное снижение сбросов регистров в память из-за нехватки регистров и из-за затирания регистров при вызовах функции (в соответствии с соглашением о вызовах). С другой стороны, количество сбросов глобальных переменных на границах базовых блоков и на вызовах функций увеличивается. Это связано с тем, что в новом алгоритме такие ситуации обнаруживаются как можно раньше, и происходит сброс содержимого регистров в память. В старом алгоритме глобальные переменные сохранятся на регистрах до самого конца базового блока или вызова функции. Однако до этого момента они могут быть сброшены по причине нехватки регистров. Тем не менее, общее количество сбросов регистров в трех первых тестах уменьшилось на 1%.

Четвертый тест содержит достаточно много недетерменизма, поэтому сравнивать работу двух алгоритмов по числу сгенерированных сбросов в память на нем некорректно. Он приведен для того, чтобы продемонстрировать, что распределение причин сброса регистров для режима эмуляции системы остается таким же, равно как и влияние изменений в алгоритме распределения регистров.

Из полученных результатов видно, что почти 60% общего числа сгенерированных сбросов составляют сбросы глобальных переменных перед вызовами функций, более 33% со сбросом глобальных переменных на границах базовых блоков. На долю остальных причин приходится не более 3.5%. Таким образом, дальнейшие улучшения алгоритма, направленные на более удачный выбор регистров при распределении, не принесут заметных результатов. Для улучшения алгоритма распределения регистров необходимо

либо ослаблять ограничения, связанные со сбросом глобальных переменных перед вызовами функций, либо переходить к алгоритму глобального распределения регистров. При этом первое может дать гораздо более сильный эффект, чем второе.

Снятие или ослабление ограничений в общем случае, когда про вызываемую функцию ничего не известно, невозможно. Однако в QEMU большое количество вызовов связано с небольшими вспомогательными функциями самого QEMU (например, функция helper_rdtsc, реализующая инструкцию rdtsc архитектуры x86). Если информацию про то, какие глобальные переменные используют данные функции, передать в алгоритм распределения регистров, то станет возможным сохранять только нужные глобальные переменные.

6. Заключение

В данной работе были рассмотрены возможные улучшения алгоритма локального распределения регистров в QEMU. Рассматривались как существующие алгоритмы, так и новые, специфичные для QEMU, эвристики. Выбранные эвристики были реализованы, и было проведено исследование их эффективности. По результатам этого исследования стало ясно, что дальнейшее улучшение выбора регистров не принесет результатов без снятия существующих в QEMU ограничений на алгоритм распределения регистров: он является локальным алгоритмом распределения регистров, и все глобальные переменные должны быть сохранены в память перед вызовом функции.

Список литературы

[1] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Дата обращения: 20.03.2012.

[2] К. Батузов, А. Меркулов. Оптимизация динамической двоичной трансляции. Труды Института системного программирования РАН, том 20, 2011 г.

[3] Wei-Chung Hsu, Charles N. Fisher, James R. Goodman. On the Minimization of Load/Stores in Local Register Allocation. IEEE Transactions on Software Engineering, vol 15, No. 10, October 1989.

[4] Vincenzo Liberatore, Martin Farach-Colton, Olrich Kremer. Evaluation of Algorithms for Local Register Allocation. Lecture notes in Computer Science, vol 1575,1999.

[5] Martin Farach, Vincenzo Liberatore. On Local Register Allocation. DIMACS Technical Report 97-33, July 1997.

[6] Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий. Второе издание. Вильямс, 2008.

Local Register Allocation Problem in Dynamic Binary Translation

Kirill Batuzov
ISP RAS, Moscow, Russia
batuzovk@ispras.ru

Abstract. QEMU is a generic machine emulator based on a dynamic binary translation approach. This paper evaluates possible improvements to the local register allocation algorithm in QEMU. The local register allocation problem is NP-hard, but there are several approximate heuristic algorithms. Among them Furthest-First and Clean-First heuristics are known to produce a near optimal result in linear time.

Local register allocation algorithm in QEMU scans instructions in direct order. For each instruction operand that is not yet stored on a register, an empty register is allocated. If there are no empty registers, then a spill of an arbitrary register is generated. On basic block ends and helper functions' calls all global variables are also spilled.

Two improvements for the register allocation algorithm are proposed in this paper. The first is to improve spill choices by using the heuristic from the Furthest-First algorithm. The second is to aggressively spill global variables which will be spilled anyway because of helper function calls and basic block ends. Both improvements were implemented and tested on practical workloads. Profiling results show that the amount of generated spills was reduced by approximately 1%.

With these improvements, over 96% of spills are generated due to function calls and basic block ends. This implies that the further improvements to this algorithm are not possible without altering its existing constraints.

Keywords: register allocation, dynamic binary translation, QEMU

References

[1]. QEMU — Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Access date: 20.03.2012.

[2]. K. Batuzov, A. Merkulov. Optimizatsiya dinamicheskoj dvoichnoj translyatsii [Optimizations in Dynamic Binary Translation]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2011, vol. 20, pp. 37-50 (in Russian).

[3]. Wei-Chung Hsu, Charles N. Fisher, James R. Goodman. On the Minimization of Load/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, vol 15, No. 10, October 1989. doi: 10.1109/TSE.1989.559775

[4]. Vincenzo Liberatore, Martin Farach-Colton, Olrich Kremer. Evaluation of Algorithms for Local Register Allocation. *Lecture notes in Computer Science*, vol 1575, 1999. doi: 10.1007/978-3-540-49051-7_10

[5]. Martin Farach, Vincenzo Liberatore. On Local Register Allocation. *DIMACS Technical Report 97-33*, July 1997.

[6]. Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2006.

Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU

К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян
{batuzovk, Pavel.Dovgaluk, vedun, vartan}@ispras.ru

Аннотация. В работе представлены два подхода к реализации полносистемного детерминированного воспроизведения в симуляторе QEMU, которые отличаются тем, какие компоненты виртуальной машины включаются в воспроизведение. Оба предложенных подхода позволяют воспроизводить выполнение гостевой ОС и прикладных программ без модификации их кода. Они не предполагают никаких ограничений на аппаратное окружение виртуальной машины. В частности, поддерживаются устройства, использующие в своей работе DMA. Оцениваются достоинства и недостатки этих подходов, оценивается уровень накладных расходов при записи журнала невоспроизводимых событий.

Ключевые слова: QEMU, детерминированное воспроизведение, программный симулятор

1. Введение

За последнее десятилетие виртуальные машины получили широкое распространение среди пользователей, как Linux, так и Windows. Основным сценарием использования можно считать развертывание виртуальной платформы (virtual appliance) на которой устанавливается некоторое ПО, требующее определенное окружение, отличное от имеющегося на физической машине, либо намеренная изоляция установленного ПО от внешнего мира. Массовое использование виртуальных машин началось благодаря тому, что их производительность достигла достаточно высокого уровня. Такие техники, как бинарная трансляция и аппаратная виртуализация, позволяют получать накладные расходы в пределах нескольких десятков процентов. Помимо того, развитый графический интерфейс и устойчивость работы бесплатных виртуальных машин (VMware Player [1], Virtual Box [2], QEMU [3, 4]) значительно упростили их эксплуатацию, рядовой пользователь имеет возможность начать эксплуатировать машину без длительного предварительного обучения, непосредственно после скачивания дистрибутива с сайта разработчика.

Но виртуальные машины вошли не только в обиход обычных пользователей, но и самих разработчиков. Отладка ядерного кода еще в начале нулевых годов начала мигрировать с обычных программных отладчиков, использующих возможности уровня ОС и аппаратуры, на отладчики, работающие с виртуальными машинами. Такой подход к отладке необходим для поиска и устранения сложных ошибок, проявление которых зависит, в том числе и от влияния отладчика на изучаемую систему. В этом случае связка отладчик/виртуальная машина успешно применяется для отладки Гейзенбагов, ошибок в драйверах устройств, межпроцессном и межтредовом взаимодействии, отладки ПО при кроссплатформенной разработке. Последнее становится все более актуальным, поскольку последние годы стремительно растет рынок мобильных устройств. Ведущие производители предлагают разработчикам набор программных инструментов, в состав которых обязательно входит виртуальная машина, используемая для отладки.

Еще одним востребованным сценарием использования виртуальных машин является расследование инцидентов нарушения информационной безопасности и криминалистическая экспертиза. Широкое распространение информационных технологий привело к необходимости соблюдения определенных политик информационной безопасности при работе с персональными данными и секретной информацией. Виртуальные машины могут применяться как для ограничения действий, так и для фиксации происходящего в информационной системе с целью последующего анализа. Применяемые ранее механизмы журналирования действий пользователя привели к идее развития этого подхода, когда начальное состояние системы и журнал действий позволяют воспроизвести работу всей эмулируемой системы внутри виртуальной машины с высокой точностью.

Данная статья посвящена вопросу организации детерминированного воспроизведения работы программного симулятора (Рис. 1). Существуют закрытые коммерческие симуляторы [5], в которых данный механизм присутствует и используется для расширения возможностей отладки. Однако среди программ с открытым исходным кодом еще не было представлено соответствующей разработки, пригодной для промышленной эксплуатации. В статье рассматриваются два подхода к реализации детерминированного воспроизведения в симуляторе QEMU, оцениваются перспективы их развития.

Дальнейший материал организован следующим образом. В разделе 2 приводится сравнение известных механизмов детерминированного воспроизведения в программных симуляторах (виртуальных машинах). Раздел 3 описывает структуру симулятора QEMU. В разделе 4 описываются различные подходы, реализующие детерминированное воспроизведение в рамках симулятора QEMU. В разделе 5 рассматриваются детали реализации этих подходов в QEMU. В разделе 6 приводятся результаты численных экспериментов. В последнем разделе делаются заключительные выводы.

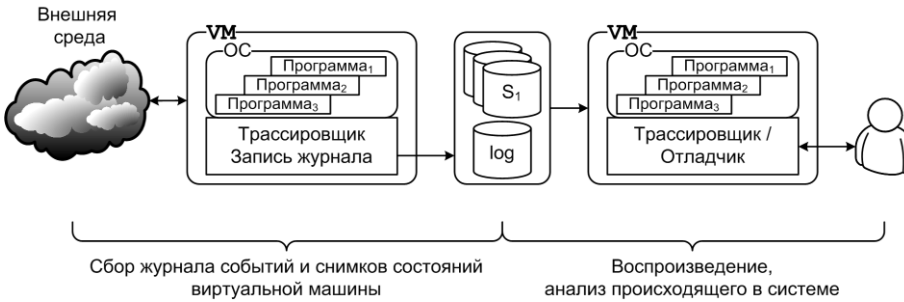


Рис. 1. Детерминированное воспроизведение работы виртуальной машины.

2. Обзор средств трассировки

Известно несколько проектов, в которых детерминированное воспроизведение работы виртуальной машины используется для достижения перечисленных ранее целей.

В 2002 году был опубликован один из первых результатов: проект ReVirt [6], проводившийся в Университете Мичигана. Детерминированное воспроизведение на основе виртуальной машины было успешно применено для анализа работы зловредного кода.

В качестве основы была взята виртуальная машина UMLinux [7], записывается и воспроизводится состояние виртуальной аппаратуры, на которой работает гостевая операционная система. Виртуальная машина представляет собой процесс хостовой системы, для доставки в виртуальную машину прерываний от периферийных устройств и исключений используются сигналы. Реализованный механизм записи и воспроизведения использует аппаратные счетчики производительности, специфические для архитектуры x86, и требует доработки ядра хостовой операционной системы. Начальное состояние представляет собой образ диска в момент перед запуском системы, промежуточные состояния сохранятся как обычные контрольные точки, синхронизированные со счетчиком выполнившихся команд. Недетерминированные события: асинхронные аппаратные прерывания, пользовательский ввод и входящие сетевые пакеты. Помимо того, определенные машинные команды, вносящие недетерминированность (rdtsc, rdpmc), эмулируются виртуальной машиной, результаты их работы заносятся в журнал. В данной работе был достигнут малый уровень накладных расходов на саму виртуализацию (от 0% до 50%), накладные расходы на снятие журнала составляют примерно 5% по сравнению с обычной работой виртуальной машины. Размер журнала (после сжатия gzip) составляет порядка 200МБ в сутки, не считая сетевой трафик.

Сотрудниками компании VMware в 2007 году была представлена публикация [8], где описывался ReTrace, механизм детерминированного воспроизведения работы всей вычислительной системы: в работе определялся набор событий в виртуальной машине, который позволял для заданного начального состояния воспроизводить работу всей вычислительной системы с точностью до машинной инструкции. Поток записываемых в журнал событий достаточно компактен, авторами работы был достигнут результат в 5 байт на тысячу инструкций, замедление работы виртуальной машины составило примерно 5-10% при включенном сборе журнала. Полная загрузка и выключение виртуальной машины под управлением ОС Windows уместилась в сжатом журнале размером 776 KB. Прототипная реализация ReTrace использовала на втором этапе симулятор QEMU. Использовалась крайне упрощенная виртуальная машина, события доставлялись в систему состоящую только из центрального процессора и виртуальной памяти.

Вскоре была опубликована еще одна работа [9] с описанием системы динамического анализа бинарного кода Aftersight, построенной на основе механизма ReTrace, но непосредственного доступа к описанной системе предоставлено не было.

В результате приведенных исследований, в коммерческий продукт VMware Workstation 6 была включена технология Replay Debugging, предоставляющая пользователю интерактивный отладчик во время воспроизведения. Базовая технология ReTrace была доведена до промышленного уровня, но возможность получать детальную трассу временно присутствовала только в отладочной версии Workstation 6, и впоследствии была отключена.

Поддержка данной технологии была остановлена при переходе виртуальных машин VMware на аппаратную виртуализацию. Можно предположить, что причиной этому стала высокая сложность переноса технологии, особенно в части перехвата асинхронных событий в периферии, т.е. отслеживание непосредственных операций чтения/записи данных ой работы устройств с памятью.

Проект XenLR [10] (2008 г., Huazhong University of Science & Technology) реализует на основе Xen 3.10 крайне ограниченную виртуальную машину, в ней присутствует только два источника недетерминированных событий: клавиатура и таймер. Работает в таком окружении модельная операционная система MiniOS. Накладные расходы на снятие журнала фактически нулевые, объем сохраняемых данных составляет 1.4МБ в сутки. Данную работу следует рассматривать исключительно как подтверждение концепции, практическое применение из-за скудности виртуальной машины едва ли возможно.

Проект ExecRecorder [11] использует виртуальную машину Bochs для полносистемного воспроизведения работы. Создание промежуточных контрольных точек происходит через системный вызов fork, последующее переключение на контрольную точку происходит как активация соответствующего остановленного процесса. Предложенная система не

поддерживает многоядерные системы и DMA-устройства, помимо того, сам симулятор Bochs вносит высокие накладные расходы на виртуализацию. Размер записываемого журнала составляет около 5.5ГБ в сутки.

Наиболее близкой работой является проект FREE [12], ведущийся в Национальном Университете Чиао Тун, Тайвань. В числе организаций, поддерживающих проект, присутствует известный центр информационной безопасности TRUST [13], расположенный в США.

В рамках данного проекта детерминированное воспроизведение реализуется в симуляторе с открытым исходным кодом QEMU. К числу недетерминированных событий относят аппаратные прерывания, чтение данных из устройств (как через порты, так и через отображение в память), DMA-транзакции. Причем вопрос поддержки последнего типа событий решен не был, в виртуальной системе не должно быть устройств, осуществляющих непосредственную запись данных в память.

При сборе журнала поддерживается счетчик инструкций, отражающий количество выполненных виртуальной машинной команд. Записываемые в журнал события снабжаются номером шага, что позволяет их доставлять при воспроизведении в соответствующие моменты времени. Состояние периферийных устройств при воспроизведении не восстанавливается, что позволяет сделать заключение о границе, по которой прошло детерминированное воспроизведение: как и в случае с ReTrace воспроизведение затрагивает только центральный процессор и оперативную память.

Несмотря на ограничения, гостевой системой может являться полноценная ОС, такая как Linux или Windows. Размер журнала относительно невелик и составляет около 500МБ в сутки. Накладные расходы на трассировку в публикации не приводятся.

Помимо рассмотренных работ нельзя не отметить соответствую возможность в коммерческом программном симуляторе SimNOW [5]. В состав симулятора входит «устройство» XTR, которое сохраняет начальное состояние виртуальной машины (дамп физической памяти, состояние процессора), а затем записывает в журнал все обращения к памяти, включая «DMA-транзакции», проходящие через северный мост. Сохраненные данные впоследствии могут быть использованы для воспроизведения работы системы и анализа ее состояния средствами плагинов-анализаторов, подключаемых к виртуальной машине через штатное API. К недостаткам следует отнести крайне медленную работу записи журнала, замедление составляет более пяти порядков, по сравнению с реальной системой, и значительный размер журнала. Положительной стороной является высокая достоверность симуляции.

Исходя из результатов рассмотренных работ, можно заключить следующее. Механизм полносистемного детерминированного воспроизведения, неоднократно разрабатывался и включался в состав различных виртуальных

машин. Наилучшие результаты были получены в коммерческом симуляторе VMware Workstation, но на данных момент поддержка этого механизма остановлена. Промышленная эксплуатация детерминированного воспроизведения поддерживается коммерческим симулятором SimNOW.

Известно несколько открытых работ, предлагающих данный механизм, но перспективы доведения этого механизма до соответствующего технологического уровня (промышленной эксплуатации) не ясны.

Одна из работ [12] предлагает использовать в качестве основы для реализации полносистемного детерминированного воспроизведения программный симулятор QEMU, но в ней не решен ряд важных вопросов, в частности поддержка «DMA»-устройств, воспроизведение состояния периферийных устройств, и, главное, интеграция предложенного механизма с направлением развития симулятора.

Исходя из характеристик рассмотренных систем, можно сделать вывод о приемлемых характеристиках детерминированного воспроизведения: размер журнала составляет несколько ГБ в сутки, а накладные расходы при сборе журнала должны не превышать нескольких десятков процентов.

3. Структура симулятора QEMU

QEMU — эмулятор вычислительных систем, использующий в своей работе динамическую двоичную трансляцию. Основной цикл работы QEMU состоит из 4 этапов (рис. 2):

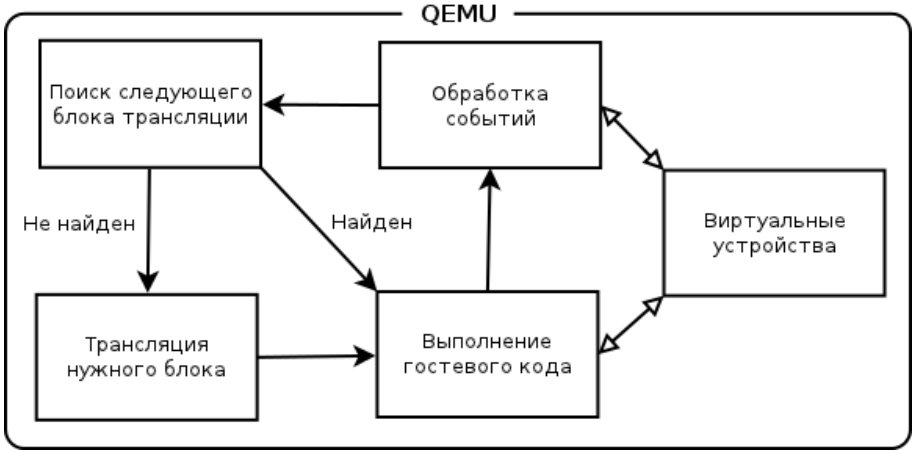


Рисунок 2 – Структура симулятора QEMU.

- выполнение оттранслированного гостевого кода,
- обработка событий, произошедших в системе,

- поиск следующего участка кода в кэше трансляций,
- трансляция нужного участка, если он не был найден в кэше.

Также QEMU создает виртуальные устройства, которые эмулируют работу реальной аппаратуры. Это позволяет ему выполнять код операционных систем и прикладных программ без каких-либо модификаций.

Теперь рассмотрим возможные источники недетерминизма в этой системе. Их можно разбить на две категории: внешние, возникающие из-за недетерминированности поступающих извне через виртуальные устройства данных, и внутренние, возникающие в результате недетерминированных взаимодействий внутри самого QEMU.

Дело в том, что в своей работе современные версии QEMU используют несколько потоков. Потоки могут создавать виртуальные устройства для того, чтобы асинхронно выполнять длительные синхронные операции. Начиная с версии 0.10.50 обработка событий и выполнение гостевого кода могут быть разнесены в различные потоки. Начиная с версии 1.0 такое поведение является единственным возможным и не может быть отключено. Потоки при взаимодействии используют различные механизмы, включая сигналы. Момент прихода сигнала является недетерминированным событием и повлиять на него из пользовательского приложения без модификации операционной системы невозможно.

Более того, сейчас ведутся исследования, целью которых является параллельное выполнение кода на разных ядрах эмулируемого процессора в различных потоках в QEMU [14]. Соответствующие изменения пока не включены в основную ветку разработки QEMU и в данной статье они рассматриваться не будут.

4. Сравнение двух подходов к детерминированному воспроизведению в QEMU

Детерминированное повторное воспроизведение программы основывается на записи всех «внешних» недетерминированных событий и обеспечении детерминированности всех «внутренних» процессов. Два рассматриваемых в данной статье подхода различаются тем, где проходит граница между «внутренними» процессами и «внешними» событиями.

Оба подхода, рассматриваемые в данной работе, воспроизводят поведение всей операционной системы с помощью эмулятора виртуальных машин QEMU. QEMU эмулирует центральный процессор гостевой системы, а также все периферийные устройства. Для передачи информации между устройствами и процессором используются виртуальные аналоги системной шины, прерываний и портов ввода-вывода. Некоторым устройствам, таким как сетевой адаптер или контроллер USB, может быть нужен доступ к внешним, по отношению к виртуальной машине, ресурсам. Для каждого типа таких устройств QEMU предоставляет интерфейс для взаимодействия с

основной ОС: сетевой интерфейс, позволяющий виртуальной машине получать доступ к реальным сетям, проброс USB для доступа к физическим USB устройствам и так далее.

На рисунке 3 схематично изображены некоторые устройства, поддерживаемые QEMU и их взаимодействие. Общее количество устройств достаточно велико и не может быть изображено на одной схеме. Пунктирные линии 1 и 2 на данной схеме ограничивают ту часть виртуальной машины, которая будет детерминировано воспроизводиться в первом и втором подходах соответственно.

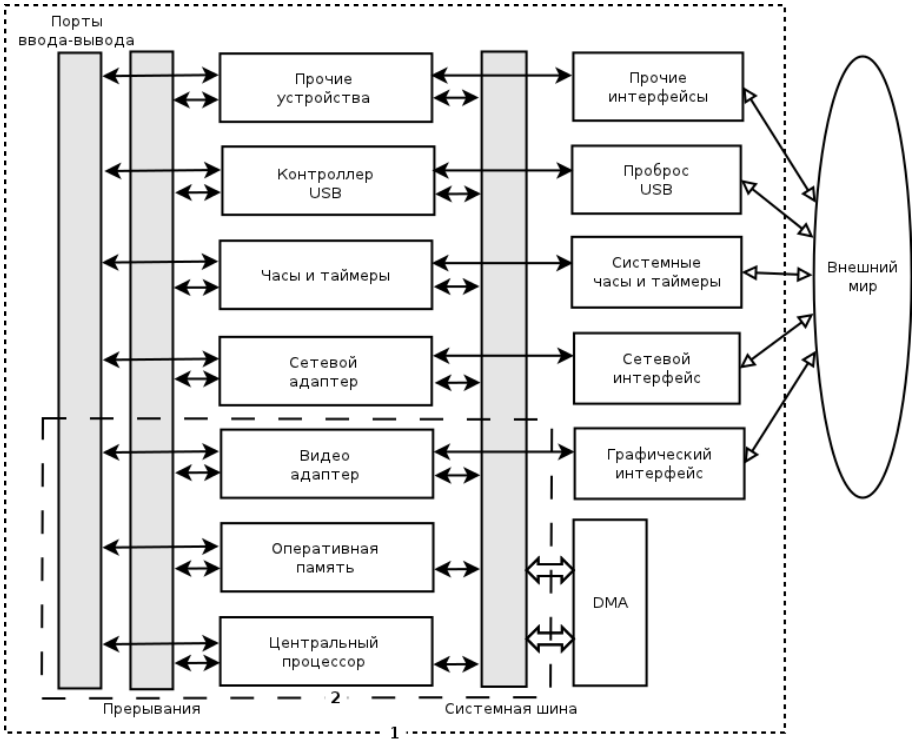


Рис. 3 – Взаимодействие различных частей QEMU.

В первом подходе предлагается детерминировано воспроизводить работу всей виртуальной машины. Граница между «внешними» событиями и «внутренними» процессами при этом проходит по интерфейсам QEMU для общения с основной ОС. В каждом из них необходимо реализовать запись и воспроизведение всех «внешних» событий, проходящих через данный интерфейс. Во «внутренней» части QEMU необходимо реализовать счетчик инструкций (для обеспечения связи времени прихода недетерминированного

события и момента исполнения программы) и модифицировать аппарат прерываний так, чтобы во время воспроизведения прерывания приходили ровно в тот же момент выполнения программы, когда они пришли во время записи журнала событий.

Подход показал свою состоятельность на версии QEMU 0.13.0, однако он имеет сложности с переносом на более новые версии.

Преимуществами такого подхода являются:

- возможность воспроизводить состояние всей виртуальной машины, включая периферийные устройства,
- наглядность записываемых событий — каждое событие отображается на понятное пользователю событие, такое как приход сетевого пакета или нажатие кнопки на клавиатуре,
- очень низкая зависимость от конфигурации гостевой системы — достаточно только реализовать подсчет выполненных инструкций.

Однако он также обладает некоторыми недостатки:

- интерфейсов взаимодействия с основной ОС в QEMU достаточно много, они меняются и часть из них, например, проброс USB-устройств в гостевую систему, существенно зависит от основной ОС,
- появление нового интерфейса, например, позволяющего подсоединить виртуальную машину как USB устройство к основной ОС, потребует его доработки для использования с детерминированным воспроизведением,
- «внутренняя» часть очень большая и обеспечить ее детерминированность сложно.

Последние версии QEMU существенно используют многопоточность. В QEMU версии 1.0.1 выполнение гостевого кода и обработка событий разнесены в отдельные потоки, также отдельные потоки заводятся для реализации асинхронного ввода-вывода через синхронные средства. Различные потоки взаимодействуют между собой с помощью сигналов. Обеспечение детерминированности такой системы является задачей, сопоставимой по сложности с исходной. Таким образом последний недостаток является ключевым.

Второй подход предлагает минимизировать количество устройств, поведение которых воспроизводится. К ним относятся:

- центральный процессор — именно он выполняет гостевую программу и обеспечивает ее детерминированное исполнение,
- оперативную память, чтобы не увеличивать размер журнала излишне сильно за счет чтений и записей в нее гостевой программой,
- видео адаптер, чтобы во время воспроизведения пользователь мог следить за «ходом» исполнения программы.

Второй подход решает проблему недетерминизма «внутренних» процессов, вынося наиболее сложные части во «вне». Все процессы, являющиеся внутренними для второго подхода, выполняются в одном потоке. Также достоинством данного подхода является его независимость от значительной части периферийных устройств и их интерфейсов взаимодействия с основной ОС. Устройства, которые не используют прямую запись в оперативную память, могут быть использованы с детерминированным повторным воспроизведением без необходимости их дополнительной модификации.

Недостаткам данного метода является более сильная зависимость от конфигурации гостевой машины. Основная работа делается в процессоре, также необходимо модифицировать все устройства, использующие DMA транзакции - прямую запись в оперативную память гостевой системы. Вторым ограничением является плохая наглядность записанных в журнале событий. Она решается сохранением соответствия между записанными в журнале событиями и событиями, наблюдаемыми пользователем. Третьим ограничением является то, что при воспроизведении поддерживается только состояние процессора и оперативной памяти, состояние периферийных устройств выпадет из рассмотрения. Помимо того, возможно, потребуется переработка или написание собственного механизма сохранения состояний системы, т.к. конфигурация виртуальной машины будет отличаться при записи и воспроизведении и встроенным механизмом снимков состояния нельзя будет воспользоваться.

5. Детали реализации детерминированного воспроизведения в QEMU

Реализация первого подхода была подробно описана в статье [15], поэтому данный раздел будет посвящен второму подходу.

Второй подход рассматривает взаимодействие программного кода операционной системы с аппаратным окружением как недетерминированное внешнее событие. При этом все внутренние процессы, в случае однопроцессорной системы, будут детерминированными. Для однозначного определения момента возникновения недетерминированного события будем включать в информацию о взаимодействии количество выполненных инструкций с момента начала работы системы.

Реализация метода строится на том, что различных способов взаимодействия с аппаратным окружением у ОС фиксированное и очень небольшое количество. Полный набор этих способов является архитектурно-зависимым. В частности, для аппаратной платформы i386\x86-64 имеют место взаимодействия через:

- контроллер памяти (MMU),
- порты ввода\вывода (PIO),
- прерывания,
- машинно-зависимые регистры (MSR),

- специальные инструкции, такие как RDTSC, CPUID.

Нужно отметить, что хотя полный набор и является архитектурно-зависимым, некоторые его составляющие, а именно контроллер памяти и прерывания, на которые приходится большая часть взаимодействий, могут быть реализованы архитектурно-независимым способом.

5.1. Подсчёт количества выполненных инструкций

В своей работе QEMU использует динамическую двоичную трансляцию. Команды целевой архитектуры транслируются во внутреннее представление, из которого в последствии генерируется машинный код для реальной архитектуры. Для подсчета количества выполненных инструкций предлагается завести глобальную переменную внутреннего представления для хранения соответствующего значения. Во время трансляции во внутреннее представление необходимо в него добавлять код, который увеличивает этот счетчик на 1 при каждом выполнении каждой инструкции. Таким образом получается достаточно легковесный счетчик инструкций.

5.2. Машинно-зависимые регистры и специальные инструкции

Так как чтение и запись машинно-зависимых регистров производится с помощью специальных инструкций (RDMSR и WRMSR для архитектуры x86), то машинно-зависимые регистры и специальные инструкции будут рассмотрены вместе. Обращение к машинно-зависимым регистрам и выполнение специальных инструкций в QEMU транслируется в вызовы вспомогательных функций. Вспомогательные функции реализуют операции, выражение которых на языке внутреннего представления будет слишком громоздким и запутанным. Они написаны на языке Си и являются частью исходных кодов QEMU. Предлагается модифицировать функции, отвечающие за машинно-зависимые регистры и специальные инструкции таким образом, чтобы значения, полученные на первом проходе, сохранялись в журнал недетерминированных событий, а во время воспроизведения считывались из него.

5.3. Контроллер памяти и порты ввода\вывода

В QEMU за контроллер памяти и порты ввода\вывода отвечает одна и та же подсистема работы с памятью. В ней вся память рассматривается как набор областей. Области памяти бывают четырёх типов: контейнер, псевдоним, RAM и MMIO. Каждая область памяти имеет размер и поддерживает операцию адресации, семантика которой зависит от типа области. Рассмотрим подробнее каждый из типов.

- Область памяти типа контейнер включает в себя другие области по различным смещением. При адресации к такой области происходит переадресация к области, содержащейся по

соответствующему адресу. Например, если область типа контейнер «C» содержит в себе область «A» по смешению «OFFSET», то при адресации к области «C», по адресу «ADDR», такому что «OFFSET» <= «ADDR» < «OFFSET» + «Длина A», произойдёт переадресация к области A по адресу «ADDR» - «OFFSET».

- Область типа псевдоним является ссылается на часть другой области по определённому смещению. Адресация для такой области работает следующим образом. Если область «A» является псевдонимом области «B» по смещению «OFFSET», то при адресации к области «A» по адресу «ADDR» произойдёт переадресация к области «B» по адресу «OFFSET» + «ADDR».
- Область памяти типа RAM ассоциирована с некоторым указателем из адресного пространства эмулятора QEMU. При адресации к области памяти типа RAM происходит разыменование указателя по соответствующему адресу.
- Область памяти типа MMIO ассоциирована с обработчиком доступа, который вызывается при адресации к этому типу области памяти.

Изначально эмулятор QEMU инициализирует только две области памяти типа контейнер: SYSTEM и IO. SYSTEM представляет собой область памяти, непосредственно доступную гостевой операционной системе при операциях работы с памятью, а IO - область памяти, доступную посредством инструкций доступа к портам ввода\вывода. Области памяти создаются, добавляются и удаляются динамически, по ходу работы гостевой операционной системы. Например, процесс добавления оперативной памяти к гостевой операционной системе выглядит следующим образом.

1. Создаётся новая RAM область памяти, размер области равен размеру оперативной памяти.
2. Выделяется память для оперативной памяти виртуальной машины.
3. Задаётся соответствие между новой областью и выделенной памятью.
4. Новая область добавляется в область SYSTEM по соответствующему смещению.

Теперь, при обращении гостевой операционной системы по адресам со «смещение» по «смещение + размер оперативной памяти» будет происходить обращение к ранее выделенной памяти.

Виртуальные устройства создают как RAM, так и IO области памяти, причём часть областей создаётся и добавляется при инициализации виртуального устройства эмулятором QEMU, а часть и при его инициализацией гостевой операционной системой. Поэтому необходимо сохранять информацию обо всех созданиях и добавлениях\удалениях областей памяти. Кроме того, в том

случае, если RAM область была создана с заранее инициализированной памятью (например, это ROM-память видеокарты), необходимо сохранять данные, которыми она была инициализирована. В случае же IO областей, необходимо сохранять результаты всех операций.

Самым распространённым способом передачи большого объёма данных между устройством и операционной системой является прямой доступ к памяти. Поэтому виртуальные устройства QEMU имеют возможность писать в память гостевой операционной системы напрямую по указателю, никак не прерывая при этом работу гостевой операционной системы. Для сохранения данных, записанных напрямую, предлагается инструментировать каждое виртуальное устройство отдельно.

5.4. Прерывания

В эмуляторе QEMU аппаратные прерывания обрабатываются в основном цикле QEMU. При этом, хотя сам цикл и является аппаратно-независимым, код для обработки прерываний различается в зависимости от аппаратуры. За вызов соответствующих обработчиков прерываний отвечает переменная interrupt_request. Предлагается сохранять значение переменной interrupt_request перед условным оператором, определяющим наличие необработанного прерывания, в том случае, если её значение изменилось по сравнению с предыдущим. Кроме этого, необходимо также сохранять номер аппаратных прерываний. Для этого предлагается инструментировать код обработчика аппаратных прерываний, которой также находится в основном цикле QEMU.

5.5. Воспроизведение журнала

Для воспроизведения выполнения по журналу событий предлагается создать в QEMU собственную конфигурацию виртуальной машины RETRACER. В отличии от обычных конфигураций, она не создает периферийных устройств, зато воссоздает структуру памяти из журнала первого прохода. Чтобы гарантировать синхронность выполнения, размер блока трансляции будет установлен равным одной инструкции, а основной цикл будет инструментирован таким образом, что после каждой выполненной инструкции происходил вызов обработчика виртуальной машины RETRACER. Данный обработчик проверяет наличие асинхронных событий типа «создание области памяти», «прямая запись в память», «доставка прерывания» и производит необходимые действия. При такой реализации «асинхронные события», с точки зрения гостевой операционной системы будут происходить тогда же, когда и при первом проходе.

6. Результаты численных экспериментов

Тестировался симулятор QEMU, в котором детерминированное воспроизведение было реализовано первым способом, т.е. высокоуровневые события доставлялись в полную виртуальную машину.

Эксперименты проводились на персональном компьютере Intel® Core™ i5-2500 CPU @ 3.30GHz (4 CPUs), 8192MB RAM под управлением 64-разрядной ОС Windows 7 Enterprise.

В ходе экспериментов оценивались две наиболее важные характеристики детерминированного воспроизведения: накладные расходы, возникающие при снятии журнала и скорость роста журнала.

На Рис. 4 показаны накладные расходы, возникающие при записи журнала. Измерения проводились на четырех примерах, для каждого примера сравнивается время работы двух режимов работы симулятора. Первый режим (Базовая версия, светло серый цвет столбцов) предполагает, что доставка прерываний может происходить в произвольном месте, второй режим допускает доставку прерываний только на границах блоков трансляции (темно серый цвет столбцов). Сплошная заливка столбцов соответствует обычному выполнению примера, заливка с градиентом – снятию журнала.

Возникающее при трассировке замедление сильно варьируется как в первом, так и во втором режиме. Однако доставка прерываний только на границах стабильно улучшает скорость. Наибольшее замедление стабильно показывает трассировка ICQ, тогда как загрузка ОС происходит даже несколько быстрее, при включенном журнале. Объяснения для этого эффекта пока не найдены. В целом, замедление варьируется в интервалах от 10 до 110% для первого режима и от 0 до 90% для второго.

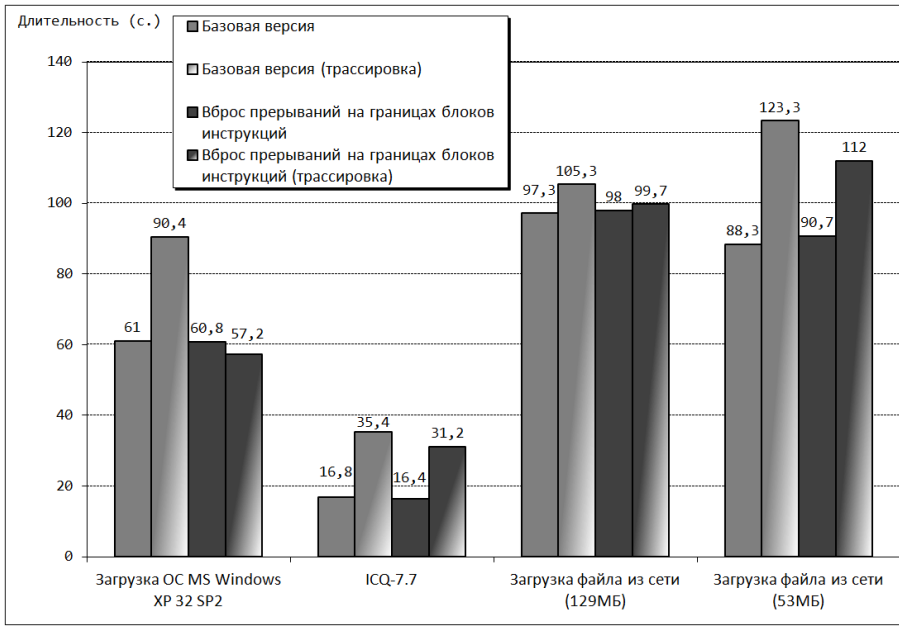


Рис. 4. Сравнение накладных расходов, возникающих при снятии журнала.

На рис. 5 и 6 показаны графики со скоростью роста журнала для гостевых систем Windows и Linux соответственно. Характер графиков одинаков, после начального этапа загрузки ОС, когда объем записываемых в журнал данных достаточно велик (в отдельные моменты времени достигает нескольких МБ в секунду), прирост размера журнала стабилизуется: и в том и в другом случае в секунду пишется порядка 50КБ данных. Большой объем данных в начальные моменты объясняются тем, что загружаемая ОС активно опрашивает шины в поисках доступного оборудования, а затем проводит инициализацию найденных устройств. Если для Windows выход на стабильную скорость роста происходит примерно через две минуты, то для Linux этот момент наступает несколько позже (шестая минута). Поскольку измерения делались для гостевых систем находящихся в состоянии покоя, пользователь не совершал никаких действий, соединение с сетью отключено, полученная скорость роста должна рассматриваться как нижняя граница. Дополнительный вклад будут вносить действия пользователя (клавиатура и мышь), а также входящий сетевой трафик. Если первым можно пренебречь, то второе неизбежно увеличит размер журнала. Однако достигнутая скорость для состояния покоя достаточно мала и сравнима с показателями аналогичных систем: 50КБ/с за сутки создают журнал размером около 4ГБ.

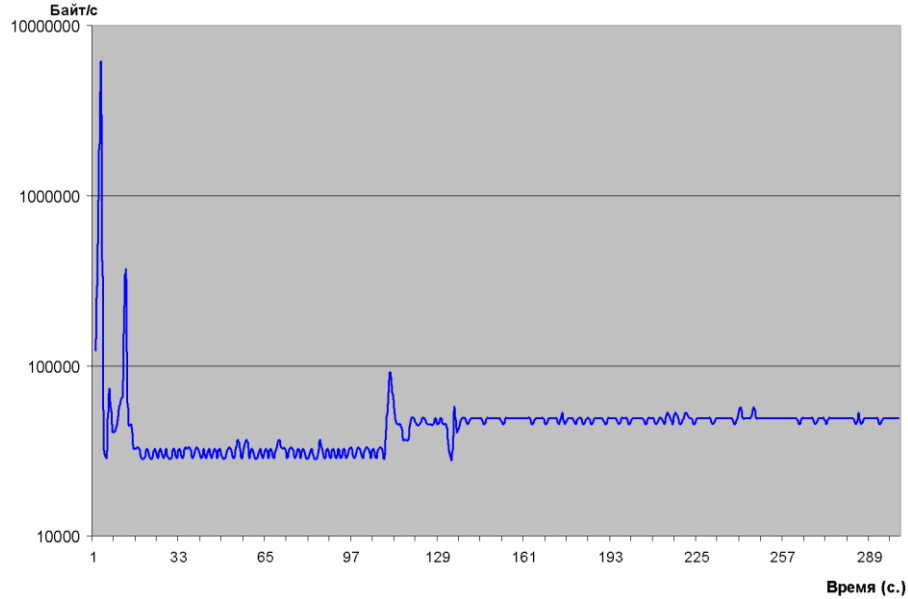


Рис. 5. Скорость роста журнала при загрузке Windows XP.

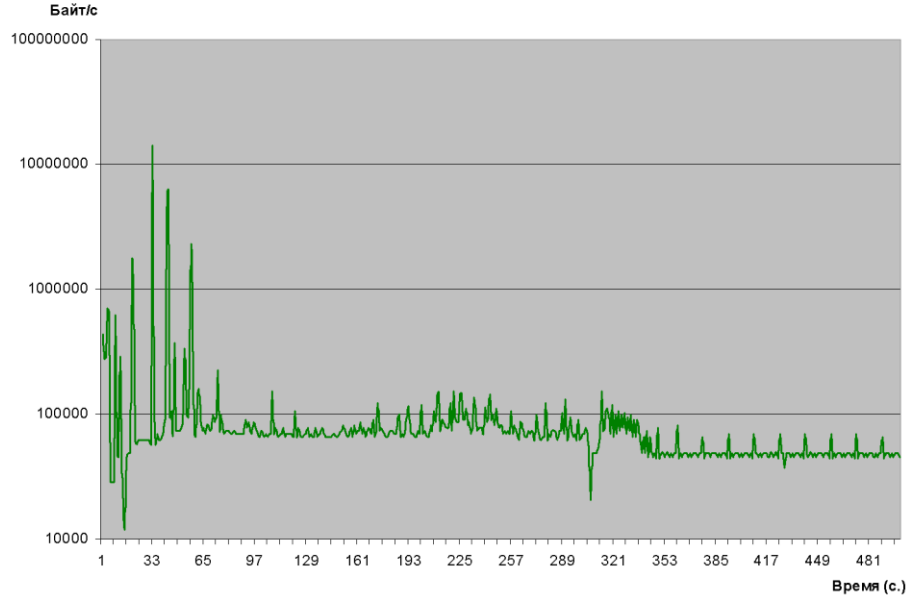


Рис. 6. Скорость роста журнала при загрузке Kubuntu v9.04.

7. Заключение

В работе был рассмотрена внутренняя организация программного симулятора QEMU и предложены два подхода к реализации в рамках данного симулятора механизма детерминированного воспроизведения.

Оба подхода обладают соответствующими достоинствами и недостатками. Накладные расходы у подхода, предполагающего доставку событий в полную виртуальную машину, показывают приемлемые значения. Оба подхода позволяют включать в состав конфигурации виртуальной машины устройства, использующие DMA.

Воспроизведение работы всей системы позволяет реализовать полносистемную отладку с возможность перемещения обратно во времени, а также интегрироваться в процесс анализа бинарного кода.

Список литературы

[1] VMware Player http://www.vmware.com/products/player/ дата обращения 2 апреля 2012

[2] Virtual Box https://www.virtualbox.org/ дата обращения 2 апреля 2012

[3] F. Bellard. QEMU, a fast and portable dynamic translator. // In USENIX 2005 Annual Technical Conf. pages 41–46, Apr. 2005.

[4] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page дата обращения 2 апреля 2012

[5] R. Bedicheck. SimNow: Fast platform simulation purely in software. In Hot Chips 16, Aug. 2004.

[6] Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. // ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.

[7] K. Buchacker, V. Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. // Proc. of Sixth IEEE International Symposium on High Assurance Systems Engineering (HASE'01), 2001 pp.0095

[8] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. // Workshop on Modeling, Benchmarking and Simulation (MoBS), June 2007.

[9] Jim Chow, Tal Garfinkel, Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. // Proceedings of the 2008 Annual USENIX Technical Conference, June 2008. pp. 1-14.

[10] Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. // In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology (2008). pp. 149-154.

[11] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T.Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. // Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71

[12] Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. // The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.
[13] The Team for Research in Ubiquitous Secure Technology (TRUST). http://www.truststc.org/ дата обращения 2 апреля 2012
[14] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung. PQEMU: A Parallel System Emulator Based on QEMU. // IEEE 17th International Conference on Parallel and Distributed Systems, 2011.
[15] Pavel Dovgalyuk. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. // Proc. of 16th European Conference on Software Maintenance and Reengineering, 2012.

Two approaches to full-system deterministic replay in QEMU

K. Batuzov, P. Dovgaluk, V.Koshelev, V. Padaryan
{batuzovk, Pavel.Dovgaluk, vedun, vartan}@ispras.ru

Abstract. *In the paper we evaluate two approaches to full-system deterministic replay. Both of them allow replaying guest OS and applications without modifications. Deterministic replay is intended for debugging and dynamic analysis of system core, multithreaded and non-deterministic applications, cross-platform applications, and devices' drivers.*
Presented approaches differ by boundary line dividing non-deterministic "outer world" and deterministic part of the simulator. All inputs from the "outer world" are written into the log to allow latter replaying of deterministic part. The common thing in both approaches is that deterministic part includes CPU, RAM, and video adapter. This allows debugging, tracing, and analyzing of the replayed code.
In the first approach "outer world" is presented by inputs – keyboard input, USB devices, mouse, network adapter, and microphone. All virtual peripheral devices should be deterministic in this approach. In the second approach all emulator parts except CPU, RAM, and video adapter are considered external. This means that all interactions between CPU and virtual peripheral devices (including IO, MMIO and DMA transactions) are written into the replay log.
The first approach has the following pros: one can replay whole virtual machine for devices' drivers debugging; relatively low number of changes within the simulator; low usage of storage for replay log. The drawback of this approach is the need to support for all external interfaces of the virtual machine. The second approach is completely opposite – it requires changes only in the several interfaces, but every virtual device with DMA interface should be modified. It also generates bigger replay logs.
We evaluated time and space overheads for the first approach. Slowdown is very low – it is lower than 1% in loading Windows XP scenario and is about 25% for network operations. Replay log growth speed for basic guest OS execution is about 50kb per second.

Keywords*: QEMU, deterministic replay, full-system emulator*

References

[1]. VMware Player http://www.vmware.com/products/player/
[2]. Virtual Box https://www.virtualbox.org/
[3]. F. Bellard. QEMU, a fast and portable dynamic translator. In USENIX 2005 Annual Technical Conf. pages 41–46, Apr. 2005.
[4]. QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page

[5]. R. Bedicheck. SimNow: Fast platform simulation purely in software. In Hot Chips 16, Aug. 2004.
[6]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224. doi: 10.1145/844128.844148
[7]. K. Buchacker, V. Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. Proc. of Sixth IEEE International Symposium on High Assurance Systems Engineering (HASE'01), 2001 pp.0095
[8]. Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. Workshop on Modeling, Benchmarking and Simulation (MoBS), June 2007.
[9]. Jim Chow, Tal Garfinkel, Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. Proceedings of the 2008 Annual USENIX Technical Conference, June 2008. pp. 1-14. doi: 10.1109/FCST.2009.55
[10]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology (2008). pp. 149-154. doi: 10.1109/FCST.2008.31
[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T.Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71. doi: 10.1145/1181309.1181320
[12]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.
[13]. The Team for Research in Ubiquitous Secure Technology (TRUST). http://www.truststc.org/
[14]. Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung. PQEMU: A Parallel System Emulator Based on QEMU. IEEE 17th International Conference on Parallel and Distributed Systems, 2011. doi: 10.1109/ICPADS.2011.102
[15]. Pavel Dovgalyuk. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. Proc. of 16th European Conference on Software Maintenance and Reengineering, 2012. doi: 10.1109/CSMR.2012.74

Восстановление структуры бинарных данных по трассам программ

А.И. Аветисян, А.И. Гетьман
{arut, thorin}@ispras.ru

Аннотация. В данной работе рассматривается задача восстановления форматов бинарных данных. Предложены подходы к решению данной задачи, а также описывается реализация этих подходов в рамках системы восстановления форматов, разрабатываемой в ИСП РАН. Особенности одного подхода заключаются в методах восстановления сетевых сообщений и файлов, входящих и исходящих потоков бинарных данных. Помимо того, в работе предложен подход, основанный на механизме описания моделей функций, позволяющий решать задачу восстановления формата с гораздо большей точностью за счёт привлечения знаний пользователя об особенностях работы конкретной программной среды.

Ключевые слова: динамический анализ; бинарные трассы; восстановление форматов; анализ протоколов

1. Введение

Использование трасс, как источника данных о программе, подразумевает динамический подход к её анализу. Особенностями данного подхода являются, с одной стороны высокая точность, так как известна вся последовательность выполнявшихся инструкций, и значения всех регистров в процессе их исполнения, что, в свою очередь, даёт возможность вычислить все буферы данных (их адреса и размеры), с которыми работала программа. Ограничение данного подхода кроется во фразе «с которыми работала программа», так как для анализа в качестве источника данных доступна только та часть программы (её графа потока управления), которая выполнялась на тех данных, поданных на вход программе в процессе снятия с неё трассы. То же самое касается и обрабатываемых программой буферов памяти – известны только те буферы, с которыми программа работала на фиксированных входных данных. Ниже в данной работе будут приведены конкретные примеры, как эти ограничения влияют на точность анализа, с точки зрения восстановления структуры бинарных данных. Указанные ограничения (во всяком случае, полнота покрытия графа потока управления) могут быть частично сняты с привлечением некоторых аспектов статического анализа, однако при этом обычно резко падает точность анализа. Проблема баланса между этими подходами при их комбинировании остаётся открытой.

Статья организована следующим образом. В разделе 2 приводится краткий обзор работ по тематике восстановления формата и проводится их сравнительный анализ. В разделе 3 описываются возможные источники данных для анализа и ограничения накладываемые особенностями этих источников на применяемый подход. В разделе 4 приводится методика, используемая в системе, разрабатываемой в ИСП РАН, а также особенности её реализации. Раздел 5 содержит общее описание планируемых работ и направлений развития в рамках решения общей задачи восстановления протоколов.

2. Обзор работ

На данный момент существует большое количество работ по восстановлению форматов данных [1,2,3,4,5,6]. Обзор применяемых подходов и их анализ приведён в работе [7], которая описывает инструмент анализа формата, разрабатываемый в ИСП РАН. Используемое далее понятие "формат сообщения" подробно описывается в этой работе. Система восстановления формата реализована в виде модуля расширения к системе динамического анализа TrEx[8], так же разрабатываемой в ИСП РАН. Динамический подход, используемый в этой системе, основан на анализе «помеченных данных» (taint analysis)[9].

Подход	Входные данные	Точность	Получаемые данные
Анализ перехваченного сетевого трафика	Перехваченный сетевой трафик исследуемого протокола	Низкая. Зависит от точности анализа паттернов	Границы и возможные значения полей
Анализ статического представления программы	Программа (исполнимый файл), реализующая работу в рамках исследуемого протокола	Средняя. Сложность определения точных границ буферов сообщений	Границы (неточные) полей, все возможные варианты структур и последовательностей полей, ограничения на значения полей
Динамический анализ программы-клиента или сервера	Трасса исследуемой программы в процессе получения/отправки и обработки/генерации сетевого пакета	Высокая. Границы буферов точно известны.	Границы полей, структуры и последовательности полей для конкретного сообщения, ограничения на значения полей. Семантика полей.

Табл. 1. Существующие подходы к задаче восстановления форматов и их основные характеристики

Как видно по представленному в [7] обзору, существует три основных подхода к решению данной задачи, различающиеся по входным данным, точности и количество возможной информации о формате, которую можно получить при использовании данного подхода. Существующие подходы и их характеристики приведены в табл. 1.

Анализ перехваченного сетевого трафика, без анализа кода сетевого приложения, основан на выделение повторяющихся паттернов. Основным ограничением является невозможность восстановления семантики полей, так как полностью отсутствует информация о том, как именно используются значения указанных полей. Этот подход применяется в работе[6].

Анализ статического представления программы, реализующей работу в рамках исследуемого протокола (или работу с файлом исследуемого формата), основан на анализе возможных последовательностей вызовов функций чтения/записи в файл (получения/отправки пакета по сети). Для точного анализа необходимы знания о параметрах, содержащих собственно читаемый/записываемый буфер. Ограничение подхода связано с тем, что для восстановления формата необходимо уметь достаточно точно восстанавливать фактические значения этих параметров (размеры буферов), чтобы верно определять границы отдельных полей и структур. При чистом статическом анализе это представляет определённые сложности. В качестве варианта решения этой проблемы может быть использован алгоритм VSA (Value Set Analysis) описанный в работе[10]. Отдельную проблему представляет собой возможность того, что исследуемая программа может работать с несколькими файлами различных форматов, так как для каждой функции чтения/записи требуется уметь определять, с каким именно файлом идёт работа, что также непросто при статическом подходе. Статический подход рассматривается в работе [1].

Различные вариации динамического анализа программы-клиента или сервера в процессе получения/отправки и обработки/генерации сетевого пакета описаны в работах [2,3,4,5,7]. Причём работы [5, 7] основаны на развитии и обобщении идей, изложенных в работах [2,3,4]. При использовании этого подхода основным преимуществом является его точность - использование анализа "помеченных данных" [9], позволяет точно определять, с данными каких участков исходного буфера, содержащего сообщение, оперирует каждая конкретная инструкция. Основной недостаток подхода вытекает из его динамической природы - восстанавливается только та часть возможной структуры сообщения, которая была обработана при выполнении на фиксированных входных данных (конкретном сетевом сообщении или файле). Для частичной компенсации этого недостатка предлагается несколько различных методов по уточнению формата, за счёт объединения данных восстановленных по разным сообщениям (файлам) одного типа. Эти методы описаны в работах [4,5]. Методы отличаются областью применимости и точностью. Объединение информации о форматах подразумевает решение

двух задач: выравнивание и обобщение. Схема, поясняющая эти задачи, приведена на рис.1.

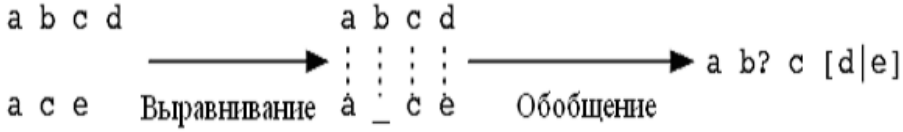


Рис.1. Объединение форматов двух сообщений. Буквами обозначены поля разных типов.

Подход, описанный в [4] для типизации полей (необходимой для решения задачи выравнивания) использует данные об адресах инструкций, что подразумевает, что при анализе формата использовалась одна и та же программа (одна её версия). Дополнительное требование состоит в том, что программа должна быть загружена в память по одним и тем же адресам (может не соответствовать действительности для ОС Windows, начиная с версии 7). При этом данный подход обеспечивает достаточно высокую точность. Подход работы [5] основан на генетическом алгоритме Нидлмана-Вунша[11], который используется для сравнения двух последовательностей нуклеотидов. Этот метод имеет более эвристическую природу и сравнительно меньшую точность, однако позволяет объединять информацию, полученную при анализе различных реализаций одного протокола и никак не привязан к особенностям используемой ОС. Вариант этого подхода также реализован в системе [7].

Основной тенденцией в области восстановления форматов сообщений является использование динамического анализа. Это можно пояснить тем, что основным требованием, в рамках данной задачи, является точность определения границ полей, так как именно на результатах этого этапа базируется дальнейший анализ структуры сообщения. Требуемую точность крайне трудно обеспечить в рамках статического подхода. Что касается анализа сетевого трафика, то, как уже было сказано, он неспособен предоставить информацию о семантике значений отдельных полей, что также сильно ограничивает область его применимости.

3. Источники данных для анализа

Источниками бинарных данных могут служить два типа объектов – сетевые соединения и локальные файлы. С точки зрения анализа структуры файлы являются более простым объектом, так как, во-первых, в них содержится только та информация, структура которой интересна исследователю – в отличие от сетевых соединений, где в сетевых пакетах передаётся большое количество вспомогательной информации, обеспечивающей собственно:

- возможность её передачи (например, координаты отправителя и адресата),
- надёжности доставки (например, номера пакетов для проверки полноты доставки),
- отсутствия возникновения искажения при передаче (например, контрольные суммы).

Другой особенностью файла является то, что его данные доступны целиком и однородны, в отличие от сетевых соединений, в которых данные передаются отдельными, как правило, небольшими порциями (пакетами). Пакеты, в свою очередь могут иметь различную структуру, описываемую в рамках протокола, по которому происходит конкретная коммуникационная сессия в рамках данного сетевого соединения. В зависимости от сложности и многообразия задач, для решения которых разрабатывался протокол, количество типов и сложность структуры сетевых сообщений, входящих в него, могут сильно варьироваться. Под сообщением здесь понимаются данные передаваемые сети на уровне приложения модели OSI. Вообще говоря, восстанавливать структуру данных можно на любом, произвольно выбранном уровне этой модели, однако более низкие уровни, как правило, хорошо стандартизированы, проверены и имеют известную структуру, а также, что немаловажно, относительно редко меняются. В то же время, количество протоколов уровня приложения растёт очень быстро, они в меньшей степени стандартизированы, многие из них являются проприетарными и закрытыми. Описанную тенденцию легко видеть на примере наиболее известной open-source системы разбора сетевых пакетов Wireshark[12]. Эта система имеет развитое API и SDK, что позволяет с минимальными трудозатратами добавлять в неё поддержку новых форматов сетевых пакетов в виде модулей расширений на языке C. Количество доступных на данный момент расширений исчисляется сотнями, и большинство из них относятся к протоколам пользовательского уровня.

3.1. Уточнение понятия "сообщение"

Следует отметить, что понятие "сообщение", в том виде как оно определено выше, не соответствует понятию "часть сетевого пакета". Между этими понятиями нет эквивалентности. Более того, ни одно из этих понятий не включает другое. Сказанное можно пояснить на следующих примерах. С одной стороны, протокол может позволять размещение в одном сетевом пакете более одного сообщения, что приводит к необходимости перед выполнением собственно анализа структуры выделять отдельные сообщения в рамках одного пакета. С другой стороны, большие сообщения могут быть разбиты на большое число сетевых пакетов, что соответствует парадигме сети с «коммутацией пакетов».

3.2. Особенности анализа пакетов разного уровня

Сложности анализа протоколов нижележащего уровня связана с одной стороны с используемым подходом, который подразумевает наличие точки в трассе, в которой существует единый буфер, содержащий всё "сообщение", а с другой техническими особенностями реализации подсистемы обработки сетевых сообщений ОС Windows NDIS (Network Driver Interface Specification)[13]. Эта подсистема сильно оптимизирована для максимально быстрой обработки нижних уровней стека протоколов. В результате этих оптимизаций единого буфера содержащего всё "сообщение" может не существовать. Таким образом, чтобы применять указанный подход требуется его предварительная модификация под существующие реализационные ограничения. В разделе 4.3 будет описан один из возможных подходов к решению данной проблемы.

3.3. Типы сообщений, с точки зрения их структуры

Обычно сообщения можно разделить на два класса – управляющие и информационные. Управляющие переводят программу-адресата (её автомат состояния протокола) в состояние, соответствующее приёму тех данных, которые будут переданы далее, либо в состояние соответствующе началу или концу сессии. Информационные сообщения содержат непосредственно данные, которые необходимо передать, информацию об их типе и структуре, данные для контроля целостности и т.д. Такая схема организации протокола соответствует протоколам, подразумевающим потенциально длительный обмен данными. Обычно их называют протоколами «с установлением соединения». Примером такого протокола может служить протокол TFTP, описанный в RFC1350[14]. Часто возникают другие задачи, в которых обмен данными организован по принципу "однократный запрос-ответ". В этом случае в рамках протокола может существовать всего один тип сообщения, но его структура может быть достаточно сложна, так как в него нужно инкапсулировать как управляющую, так и информационную составляющие. То есть оно должно содержать исчерпывающую информацию о типе запроса, который оно содержит и полное описание данных, которые в рамках него передаются. Если количество различных запросов относительно велико, а структура данных, передаваемых в рамках этих запросов, может сильно разниться, то сложность структуры сетевого сообщения может быть весьма высокой. Примером такого протокола (и сообщения) может служить протокол DNS, описанный в RFC1035[15] и RFC3425[16].

С точки зрения задачи восстановления формата первый тип протоколов гораздо проще для анализа, так как вся сложность, связанная с организацией взаимодействия удалённых узлов преобразуется в количество разных сообщений, в то время, как структура каждого отдельного сообщения довольно проста. В таких сообщениях, обычно, не встречаются последовательности структур и вложенные последовательности.

Управляющие сообщения, как правило, предельно просты и лаконичны и не содержат ни последовательностей, ни структур, а только относительно небольшой набор полей, задающих команду, содержащуюся в данном сообщении и необходимые параметры. Второй тип протоколов, напротив, представляет большую сложность и по тому, какой результат различные инструменты анализа показывают на сообщениях из этого класса, можно судить об уровне их развития. Сообщения из этого класса имеют высокую сложность организации структуры, содержат вложенные последовательности, поля указатели и т.д. Это является отражением того факта, что все возможные действия в рамках данного протокола должны быть переданы с помощью единственного сообщения. То, что разные команды протокола могут иметь сильно отличающиеся параметры, приводит к высокой вариативности структуры сообщения - для разных команд структура сообщения будет существенно различаться. В терминах грамматики восстанавливаемого формата это будет выглядеть, как большое количество дизъюнкций в правых частях правил вывода. В терминах статьи [7] это будет выглядеть, как большое количество полей-вариантов в дереве обобщённого формата. Для случаев таких сообщений, особенно актуальна задача объединения информации о формате, получаемой при анализе различных сообщений одного типа в рамках динамического подхода. Несколько вариантов решения этой задачи было рассмотрено выше.

3.4. Задача анализа протокола

Следует сказать, что задача анализа формата данных являясь с одной стороны самостоятельной и важной задачей обратной инженерии, в то же время, в контексте анализа сетевого трафика, является частью более общей задачи анализа протокола. Наряду с ней, частью задачи анализа протокола является задача восстановления автомата состояний протокола. Собственно наличие протокола, то есть списка правил обмена информацией также является свойством исключительно сетевых соединений (в отличие от файлов), что сильно усложняет их анализ. Автомат состояний протокола определяет, в каком состоянии находится разборщик протокола, то есть, фактически, какой набор сетевых сообщений он может принимать, а также возможные переходы между этими состояниями в зависимости от типов получаемых сообщений и их содержимого. Отдельный тип переходов соответствует переходам по таймеру, по истечению некоторого, обычно фиксированного в протоколе, времени, в случае, если за этот промежуток времени не произошло других событий (не было получено сообщений). Наиболее простой формой представления автомата состояний является граф, в котором вершинам соответствуют состояния, а рёбрам - переходы между ними, причём рёбра помечены условием соответствующего перехода (тип сообщения, его содержимое или истечение таймера). Пример графа автомата состояний протокола FTP (File Transfer Protocol) приведён на рис. 2.

Необходимым условием, для решения задачи восстановления автомата протокола является решение задачи восстановления формата - требуется восстановить форматы всех сообщений протокола и классифицировать их, то есть понять, какие сообщения относятся к разным типам. Это необходимо, чтобы понять, какой набор вершин (состояний) и рёбер (переходов) в принципе возможен в соответствующем графе автомата состояний.

До недавнего времени большинство усилий было сосредоточено на решение первоочередной задачи - автоматическом, либо максимально автоматизированном, восстановлении формата сетевых сообщений (и файлов). Однако в последнее время стали появляться статьи с новыми подходами к решению второй составляющей общей проблемы восстановления протоколов - задаче восстановления автомата состояния.

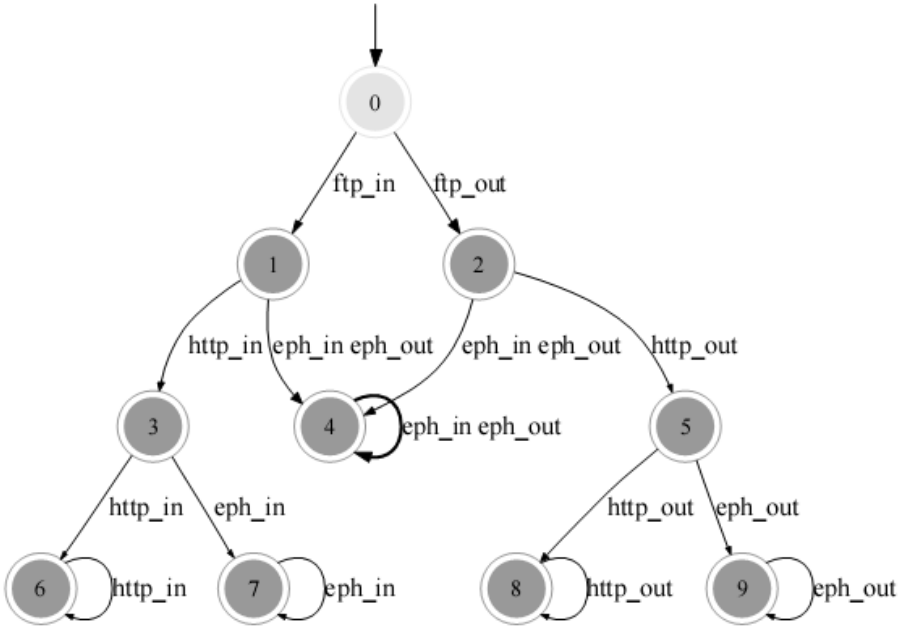


Рис.2. Представление автомата состояния протокола FTP в виде графа

4. Особенности реализованной системы

Работа [7] описывает систему восстановления форматов данных, разрабатываемую в ИСП РАН и реализованную в виде модуля расширения к среде динамического анализа TrEx. При разработке этой системы были обобщены и дополнены существующие методы. По сравнению с предыдущими работами можно выделить следующие нововведения.

- Использовалась битовая гранулярность представления памяти, что позволило анализировать поля-флаги.
- Реализована возможность анализировать совокупность вызовов функций получения пакетов (чтения из файлов), что позволило восстанавливать формат сообщений разбитых на несколько пакетов с одной стороны, и файлы с другой стороны, так как обычно файлы считываются именно по частям.
- Введено понятия "идентификатор источника", параметра, который идентифицирует объект, с которым работает конкретный вызов функции (например, сокет в случае сетевых пакетов). Это позволило автоматически разделять потоки сообщений, относящихся к разным соединениям.
- Усовершенствован алгоритм анализа циклов, что позволило восстанавливать последовательности с полями указателями. Пример таких последовательностей имеет место в DNS сообщениях, использующих сжатие строк.
- Для анализа циклов используется граф потока управления, восстановленный по трассе, что позволяет анализировать приложения, защищённые программами упаковщиками, для которых этот граф не может быть получен чисто статическими методами. Кроме того, использование такого графа потока управления и динамического подхода в целом, позволяет эффективно анализировать программы, защищённые от дизассемблирования, а также, содержащие обфускации вида "мёртвый код", "неисполняемый код" и т.п.
- Введено понятие "виртуальный буфер", соответствующее объекту, структура которого восстанавливается (всему сообщению или файлу). Необходимость этого понятия обусловлена рядом причин. В случае сетевых сообщений это вызвано тем, что может не существовать единого буфера, содержащего на некотором фиксированном шаге всё сообщения (например, в случае разбиения сообщения на отдельные пакеты). В случае файлов это обусловлено тем, что файл, как правило, считывается по частям и, возможно, непоследовательно, в результате чего простая "склейка" считанных данных может не соответствовать ни всему файлу, ни его части.

4.1. Особенности анализа файлов

Анализ файлов имеет ряд особенностей, являющихся следствием того, что данные файла сразу целиком доступны и доступ к ним может осуществлять в произвольном порядке. В случае сообщений это может быть не так - часто, для разбора некоторой части сообщения необходимо предварительно разобрать предшествующую ей часть. Результатом этого является ряд сложностей и ограничений возможности восстановления структуры файла с помощью динамического подхода. Одной из таких особенностей является

наличие функциональности позиционирования в данных файла (например, функция SetFilePointer API Windows), которая приводит к тому, что данные файла могут считываться и обрабатываться непоследовательно, не в прямом порядке и, более того, не полностью. То есть файл, с точки зрения модели доступа к его данным, представляет собой RAM (Random Access Memory). Это приводит, во-первых, к необходимости отслеживать, обработка какой части файла происходит в данный момент, во-вторых, следствием этого является то, что структура значительной части файла может быть не восстановлена в рамках динамического подхода, так как она не считывалась и не обрабатывалась. Эта особенность хорошо видна на примере файлов имеющих заголовок, в котором есть таблица смещений отдельных таблиц данных в этом файле. В качестве конкретного примера можно взять формат исполнимого файла PE32[17]. Основная часть файла состоит из отдельных секций (кода, данных и др.), смещения которых в файле описываются в таблице секций (section table), которая находится в заголовке файла. Другой пример будет приведён в разделе 4.6. Таким образом, возникает подзадача контроля позиции в файле, с одной стороны, за счёт суммирования размеров считанных (записанных) буферов, а с другой стороны, за счёт отслеживания вызовов функции (функций) позиционирования.

Одной из причин того, что файл читается не целиком, может быть избыточность содержащейся в нём информации (часть информации тем или иным способом дублируется), порождённая, с одной стороны, развитием формата и появлением в нём новых полей данных, а с другой необходимостью поддержки обратной совместимости. Примером подобной избыточности формата файла может служить формат для хранения растровой графики BMP[18], разработанный фирмой Microsoft. В начале файла идёт структура BITMAPFILEHEADER, которая в частности содержит поля, хранящие размер файла и смещение собственно данных изображения от начала файла.

Далее, в зависимости от версии файла, идёт одна из следующих структур:

- BITMAPINFOHEADER
- BITMAPV4HEADER
- BITMAPV5HEADER

Данные в начале этих структур совпадают и содержат информацию о:

- Размере самой структуры
- Ширине и длине изображения в пикселях
- Количестве битов, описывающих каждый пиксель

Очевидно, что на основе этих данных можно вычислить размер файла, и таким образом, поле структуры BITMAPFILEHEADER, хранящее размер файла становится избыточным. Это приводит к тому, что различные приложения могут использовать вычисления на основе значений различных полей или их комбинаций для достижения одного и того же результата – получения размера файла. Кроме того, одни приложения могут проверять, что скоррелированность значений в различных полях структур, а другие могут

просто игнорировать избыточные с их точки зрения поля. Кроме того, структура BITMAPFILEHEADER, содержит несколько зарезервированных полей, которые, согласно формату, должны быть заполнены нулями, что как показывает практика, проверяется далеко не каждым приложением. Пример восстановления формата BMP файла будет приведён ниже.

Следует отметить, что некоторые из особенностей, описанные выше для файла и связанные с возможностью позиционирования, могут быть ограниченно применимы и к сетевым сообщениям. Особенно это касается сложных сообщений типа DNS. В рамках сообщений таких протоколов считывание данных отдельных пакетов (но не пакетов в сетевом потоке) может происходить непоследовательно и не полностью. С точки зрения реализации это организуется с помощью полей-указателей – ячеек памяти пакета, содержащих смещение других полей относительно известных точек (например, начала сообщения). Неполнота обработки, как и в случае файлов, может быть связана с избыточностью сообщения, которая, в свою очередь, может быть порождена развитием протокола во времени и необходимостью поддержки обратной совместимости с программными продуктами, которые были разработаны для более ранних версий протоколов.

4.2. Особенности анализа входящих и исходящих данных

Следует отметить существенное различие подходов при восстановлении протокола по входящим сообщениям (считываемым файлам) и по сообщениям, которые формируются на выходе исследуемой программы (записываемые файлы). В случае входящих сообщений, проще определить семантику различных полей, таких как поля длины, контрольные суммы, разделители, так как они используются по назначению в коде программы. Например, поля длины используются как максимальные значения счётчиков итераций в цикле обработки последовательности. В случае анализа формата исходящего сообщения (выходного файла) часть этой семантики можно восстановить по способу формирования значения поля (контрольная сумма зависит от значений других полей), а часть семантики восстанавливается только в самом общем виде («магические константы» протокола тяжело отличить от разделителей). С другой стороны, границы полей и сообщений в выходном потоке, как правило, определять проще, так как они часто уже известны в момент вывода. Другой важный аспект - явное тяготение разработчиков к анализу форматов входящих данных в рамках динамического подхода. Этот вид анализа описывается в работах [2,3,4,5]. Это, по-видимому объясняется тем, что базовый структурный анализ сообщения во всех перечисленных работах, в том или ином виде, основывается на хорошо разработанном методе анализа "помеченных данных"[9]. В тоже время, до недавнего времени не было известно базового подхода, позволяющего эффективно анализировать структуру формирующегося пакета. Только в работе [19] был предложен первый оригинальная методика "деконструкции буфера" (buffer deconstruction) к восстановлению структуры входящих данных

на основе динамического анализа. Идея методики заключается в том, что программа сохраняет поля, структуры и последовательности в отдельных буферах памяти и создаёт сообщения путём комбинации этих буферов и копировании их в буфер сообщения. Соответственно, алгоритм осуществляет итеративный поиск соответствующих буферов назад по трассе от точки отправки сообщения (в этот момент его буфер полностью сформирован). Другой важной особенностью работы является чёткое разделение алгоритмов восстановления как структуры входящих и исходящих данных, так и подробный анализ того, какая именно семантика и как именно, может быть с достаточной точностью восстановлена для входящих и исходящих данных. Разница в восстановлении структуры хорошо просматривается на примере базового алгоритма, лежащего в основе большинства подходов к восстановлению на основе динамического анализа: анализ "помеченных" данных в случае входящего потока данных и "деконструкция буфера" в случае исходящего потока. Анализ семантики состоит из двух несвязанных подходов - структурный анализ графа потока управления и выводы о типах на основе его особенностей (так восстанавливают поля длины и поля-разделители) и, так называемый, алгоритм "выведения типов" (type inference), который позволяет делать выводы о семантике содержимого полей на основе того, как это содержимое используется в точках где семантика известна (например, в качестве параметров функций известного вида, типа обработки строк). В частности, в работе приведена табл.2, показывающая, какая именно семантика и для какого типа данных (входящих или исходящих) может быть эффективно и с достаточной точностью восстановлена.

Также в этой работе предлагался метод выделения из общей бинарной трассы сетевого диалога отдельных подтрасс, соответствующих получению и обработке (или формированию и отправке) отдельных сообщений. В рамках метода была предложена качественная эвристика, позволяющая определять, какие пакеты соответствуют одному сообщению, в случае, если сообщение было разбито на несколько пакетов. Эвристика сводится к определению наличия или отсутствия зависимости по данным между максимальным размером текущего получаемого пакета (параметр функции получения) и данными какого либо из предшествовавших пакетов.

Связь между подходами к анализу структуры входящих и исходящих бинарных данных во многом аналогична связи между прямым и обратным слайсингом, хотя в случае слайсинга, первым появился именно обратный анализ[20], эффективно применявшийся для целей отладки, и, лишь впоследствии, идеи были обобщены и разработан прямой анализ[21]

Field Semantics	Received	Sent
Cookies	yes	yes
IP addresses	yes	yes
Error codes	no	yes
File data	no	yes
File information	no	yes
Filenames	yes	yes
Hash / Checksum	yes	yes
Hostnames	yes	yes
Host information	no	yes
Keyboard input	no	yes
Keywords	yes	yes
Length	yes	yes
Padding	yes	no
Ports	yes	yes
Registry data	no	yes
Sleep timers	yes	no
Stored data	yes	no
Timestamps	no	yes

Табл. 2. Виды семантики полей, которые могут быть восстановлены для входящего и исходящего потока данных

4.3. Система описания моделей функций

Во многих подзадачах, возникающих в процессе решения задачи восстановления форматов и более общей задачи восстановления протоколов, требуется информация о функциях и их параметрах. Как правило, требуется иметь возможность задавать набор описаний некоторой группы функций и их параметров. Ярким примером может служить процесс автоматизации выделения отдельных потоков сообщений в полной трассе, содержащей сетевой диалог, а также отдельных алгоритмов получения и обработки отдельных сообщений в каждом потоке. Один из вариантов такого алгоритма был реализован в системе TrEx. Для работы алгоритма необходимо задать функции получения и отправки сообщений. В случае работы с файлом -

функции чтения/записи и позиционирования. После этого нужно описать параметры этих функций:

- Идентификатор объекта, с которым идёт работа: сокет в случае сетевого соединения, идентификатор файла, в случае файла
- Буфер, содержащий отправляемые/получаемые данные
- Смещение в файле для функции позиционирования

Собственно семантика параметров описывается с помощью системы описания семантики, которая будет рассмотрена в разделе 4.4. В данном разделе речь идёт не о задании конкретной семантики параметра, а об определении его места в памяти, то есть способе получения ячейки памяти (или регистра), содержащей значение фактического параметра для каждого конкретного вызова.

Приведём формальное определение того, что мы называем моделью функции. Модель представляет собой набор следующих атрибутов:

- Имя модели (функции может соответствовать несколько моделей)
- Границы модели
  - Адрес входа в функцию
  - Набор адресов возврата
- Список описание параметров (входных и выходных)
- Список зависимостей между входными и выходными параметрами

Приведём несколько характерных примеров, когда функции может соответствовать несколько моделей. Во-первых, многие функции ведут себя существенно различным образом (генерируют различные выходные параметры), в зависимости от того, произошла в ходе их выполнения ошибка или нет. Например, если функция получения сетевого пакета recv завершилась успешно, она возвращает размер полученных данных. В случае если произошла ошибка, она возвращает код ошибки. Другим примером являются функции-обработчики прерываний, которые в качестве одного из параметров принимают код ситуации, которая имела место и в зависимости от этого осуществляют различные виды обработки. Подобное поведение также характерно для функций-диспетчеров.

Описание параметра содержит имя параметра, его тип и описание способа получения ячейки памяти, содержащей значение фактического параметра для каждого конкретного вызова. По типу параметры делятся на стековые, регистровые и параметры-выражения.

Стековые параметры передаются вызову через стек, соответственно описание получения содержит смещение и размер параметра в стеке. Регистровые параметры передаются (или возвращаются) через регистры. Описание параметра содержит имя регистра. Параметры-выражения соответствуют данным, передаваемым через буферы памяти, адрес и размер которых определяются другими параметрами. Описание параметра выражения содержит следующие части:

- Опциональный сегментный регистр (для платформы x86)
- Арифметическое выражение, результат вычисления которого даёт адрес буфера. В качестве переменных в выражении могут фигурировать имена ранее описанных параметров.
- Арифметическое выражение (которое может быть константой), результат вычисления которого даёт размер буфера (или его верхнюю границу - см. ниже)
- Опциональный тип содержимого буфера - ASCII или Unicode строка

Указание сегментного регистра наиболее актуально для 16-разрядного кода. Если сегментный регистр не указан, то арифметическое выражение задаёт линейный адрес буфера. В противном случае, оно задаёт логический адрес, который транслируется в линейный с учётом значения соответствующего сегментного дескриптора.

В случае если задан тип содержимого буфера, значение выражения для размера интерпретируется как верхняя граница размера буфера, а его фактический размер определяется на основе его содержимого. Например, для ASCII строки её фактический размер это позиция первого символа «0» в этой строке.

Список зависимостей между параметрами состоит из пар входной-выходной параметр, которые задают зависимости по данным между этими параметрами. Наличие зависимости означает, что значение заданного входного параметра влияет на значение заданного выходного параметра и, наоборот, значение выходного параметра формируется на основе значения заданного входного параметра. Нужно отметить, что в данном случае не рассматривается вопрос о конкретном функциональном виде зависимости между параметрами.

Значения фактических входных параметров некоторого вызова определяются, как значение соответствующей параметру ячейки памяти или регистра в точке вызова функции, а значения выходных параметров - как значения соответствующих ячеек в точке выхода из функции.

4.4. Система описания семантики функций и их параметров

После того, как некоторая функция описана, часто требуется задать некоторые её свойства и свойства её параметров, для того чтобы соответствующие алгоритмы могли выделить эту функцию из общего набора и определить важные для них параметры. Один из примеров, когда такая информация о функциях может быть необходима, был приведён выше. Другой пример, о котором пойдёт речь в разделе 5.1 - анализ шифрованных сообщений, для проведения которого необходимы знания о функциях шифрации/дешифрации. Существует несколько подходов, позволяющих выделять такие функции автоматически (ни будут рассмотрены в соответствующем разделе). Однако в некоторых случаях нужно предоставить пользователю возможность задать такие функции явно. Реализованная система конструктивно состоит из двух подсистем:

- Подсистема описания семантики функций и параметров, без привязки к конкретным функциям и их параметрам
- Подсистема сопоставления описанной семантики конкретным моделям функций и их параметров

Обе подсистемы хранят свои данные в БД MySQL. Формально семантика функции описывается набором следующих атрибутов:

*Имя семантики
*Уникальный идентификатор семантики
*Текстовое описание семантики
*Набор связанных семантик параметров

Семантика параметра это:

*Имя семантики параметра
*Глобально уникальный (среди идентификаторов семантик параметров всех функций) идентификатор
*Текстовое описание семантики параметра

Эти атрибуты задаются при описании новой семантики в рамках первой подсистемы. Имена и текстовое описание необходимы для организации взаимодействия с пользователем в рамках графического интерфейса. За это взаимодействие отвечает вторая подсистема.

4.5. Общая схема подхода

Исходной точкой анализа формата бинарных данных является получение трассы – этот процесс подробно описан в статье [7]. Однако дальнейшая последовательность действий, по сравнению с описанной в этой статье, претерпела значительные изменения, связанные, в первую очередь, с повышением уровня автоматизации и расширением области применения подхода. После того, как трасса получена, последовательность действий аналитика следующая:

1. Описать модели функций получения/отправки пакета, либо чтения/записи/позиционирования файла – в зависимости от вида исследуемого потока бинарных данных с помощью системы описания моделей. Эти данные могут быть получены на основе:
   - Проведённого исследования бинарного кода (в случае анализа неизвестной программной системы)
   - Подключённой символьной информации
   - Знаний, полученных на основании анализа этой же программной системы в других проектах
2. Описать соответствующие виды семантики функций и их параметров в подсистеме описания семантики системы работы с семантикой функций
3. Задать семантику каждой из описанных функций и семантику её параметров с помощью подсистемы привязки семантики функций и параметров к описанным ранее моделям

4. В случае, если в трассе присутствует несколько потоков бинарных данных – задать значение идентификатора исследуемого потока
5. Указать идентификатор исследуемого бинарного потока, в случае, если таких потоков несколько

Получив всю необходимую информацию, система автоматически найдёт все вызовы интересующих функций, построит «виртуальный буфер», определит, к какой его части относится каждый вызов функции чтения или записи, восстановит содержимое соответствующих частей, а затем и структуру всего буфера.

4.6. Проверка реализованной системы

Для проведения экспериментов использовалась рабочая станция со следующими характеристиками: 2 ЦПУ Intel Xeon E5540 2.53 GHz, 48 GB RAM, 64-разрядная ОС Windows 7 SP1. Трассировка выполнялась в симуляторе AMDSymNow v4.5.1, в виртуальной машине была установлена ОС Windows XP SP2. Из библиотек ОС виртуальной машины были предварительно извлечены экспортируемые символы, которые затем использовались для разметки трассы.

Требовалось восстановить формат неизвестного файла, который подавался на вход сторонней программы, доступной в виде бинарного файла. Впоследствии было выяснено, что исследуемый файл содержит исполняемый код, который интерпретируется данной программой. Исследуемая программа принимает на вход бинарный файл testfile.bin размером 2100 KB и выводит некоторые данные на консоль. Так как вывод был достаточно объёмен, а обновление экрана в виртуальной машине при снятии трассы сильно замедлено - вывод был перенаправлен в файл.

Для применения алгоритма восстановления требовалось найти точку (или точки) ввода данных из файла. Первоначальный поиск функции ReadFile не дал результатов. Вторым вариантом получения данных из файла является его прямое отображение в память и чтение этой памяти без использования функции ReadFile. Изучение символьной информации позволило установить, что для отображения использовалась функция ZwMapViewOfSection. Восстановление её параметров позволило выделить область памяти, в которую был отображён файл. Восстановление формата осуществлялось от точки возврата данной функции до конца трассы. По результатам восстановления буфера, содержащего указанный файл, был сделан вывод о том, что фактически читалась незначительная часть файла, причём файл читался непоследовательно - с разрывами. Содержимое считанных областей было восстановлено. В результате автоматического анализа формат считанной части файла был восстановлен. Результаты восстановления выявили следующие аспекты:

- файл содержал несколько полей-указателей, хранящих смещения данных в файле (что соответствует формату исполняемого PE-файла)

- восстановлено некоторое количество ключевых полей со значением 0x66 (по-видимому, части префиксов машинных команд)
- восстановлены последовательности, содержащие двухбайтные префиксы
- восстановлены флаговые поля, содержащие от двух до трёх групп флагов.

Основная масса данных интерпретировалась как отдельные байты, что также соответствует исполняемому файлу.

5. Дальнейшие работы

5.1. Анализ шифрованного трафика

Следующим шагом в области восстановления форматов сообщений явилась разработка подходов для расширения области применимости существующих методов на шифрованный трафик. Необходимость развития в этом направлении возникла прежде всего вследствие распространения вирусов, организовывающих ботнеты из заражённых компьютеров. Для контроля ботнетов и их обезвреживания потребовались инструменты для проведения в краткие сроки анализа протоколов обмена сообщениями в рамках этих сетей.

Подробное описание модели функционирования ботнетов, а также методы их контроля и обезвреживания содержатся в работе [19]. Так как цель злоумышленников - максимально затруднить анализ как кода, так и командного протокола и, таким образом, продлить существование ботнета, то все сообщения передаются в зашифрованном виде. Одним из примеров такого рода вирусов является agobot, осуществляющий шифрацию своих сообщений с помощью собственной реализации SSL. Одной из первых работ в этом направлении является описание системы ReFormat[22], тестирование которой производилось как раз на примере анализа протокола agobot.

Идея подхода заключается в том, чтобы автоматически найти в трассе обработки сообщения точки, где сообщение уже полностью прошло через стадию дешифрации (или, в случае отправки сообщения, уже полностью сформировано, но ещё не зашифровано) и определить буферы, в которых в этот момент содержатся данные сообщения. Фактически, эти точки соответствуют вызовам функций шифрации и возвратам из функций дешифрации. Их поиск, таким образом, сводится к выделению функций шифрации/дешифрации. В качестве основного признака была выбрана повышенная концентрация арифметико-логический операций в этих функциях. Эмпирически была установлен порог процента арифметических операций, позволяющий с высокой долей уверенности идентифицировать функции шифрации/дешифрации.

Проведённые эксперименты над рядом как открытых, так и закрытых шифрованных протоколов (IRC, HTTPS, MIME, agobot) показали, что этот

порог может быть достаточно произвольно взят в пределах [0.25, 0.8]. На практике был взято значение 0.5. Ограничением описанного подхода является, во-первых, то, что в нём рассматриваются трассы содержащие получение и обработку одного сетевого сообщения и не описывается инструмента выделения таких трасс из трассы содержащей полноценный сетевой диалог. Кроме того, явно предполагается наличие ровно двух отдельных фаз расшифровки и обработки сообщения, причём вторая должна строго следовать за первой.

Эта модель может быть неприменима в случае, если только часть сообщения шифруется, причём шифрованная и незашифрованная части чередуются. В этом случае количество фаз обработки и дешифрации увеличивается, причём эти фазы чередуются в трассе. Это ведёт к тому, что отсутствует единая граница между этими двумя фазами (поиск которой, собственно, осуществляет алгоритм системы ReFormat). После того, как граница между фазами найдена, требуется определить буфер (или буферы) содержащий дешифрованные данные сообщения в граничной точке. Для этого применяется следующий критерий: буферы содержащие сообщение должны обладать рядом свойств. А именно, их данные должны:

- быть "помеченными" (зависеть от данных полученного сообщения)
- записываться на фазе дешифрации и читаться на фазе разбора

То есть, эти буферы можно получить в три этапа:

1. Провести анализ "помеченных данных"
2. Восстановить записываемое множество (write set) фазы дешифрации и читаемое множество (read set) фазы разбора
3. Пересечь эти три множества

Отдельная проблема возникает, если таких буферов оказывается несколько - необходимо понять в каком порядке и с какими смещениями содержат эти буферы расшифрованные данные сообщения. В работе эта проблема решается следующим эвристическим методом:

1. Буферы выстраиваются в том порядке, в котором осуществляется их чтение на фазе разбора
2. Их содержимое рассматривается, как единый "виртуальный буфер", содержащий всё расшифрованное сообщение

Следует отметить, что такой метод применим не всегда - если некоторая часть сообщения была дешифрована дважды (или была сделана копия дешифрованных данных), то применение данного метода сильно исказит структуру сообщения, так как в "виртуальном буфере" будет содержаться несколько одинаковых частей сообщения.

Реализация этого метода выполнена в виде отдельного независимого модуля, которые может применяться в комплексе с произвольной системой восстановления форматов, расширяя её возможности. Для тестирования использовалась система восстановления [4].

Развитие и обобщение этого подхода является предметом уже упоминавшейся работы [19]. В этой работе содержится описание инструмента Dispatcher, проверка которого осуществляется на примере анализа командного протокола (C&C protocol) достаточно широко известного вируса MegaD. Сообщения командного протокола этого вируса обладают указанными выше свойствами (делающими невозможным прямое применение подхода из работы [22]) : они содержат незашифрованную часть - размер пакета (Msg_len), который обрабатывается вначале, за которым идёт шифрованная часть, содержащая основные данные пакета (Payload). Структура пакета MegaD приведена на рис. 3.

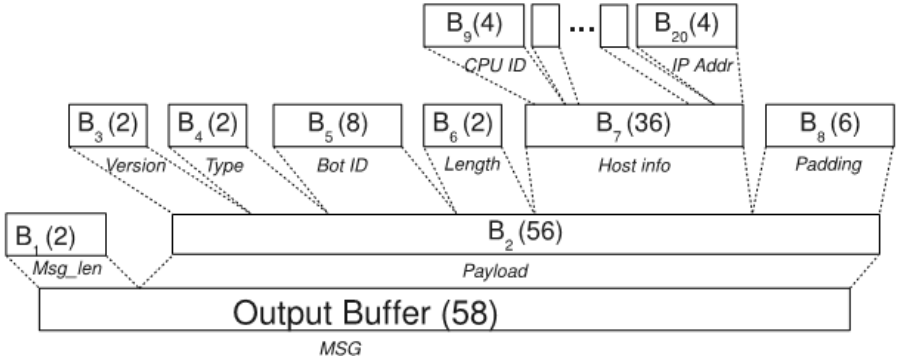


Рис. 3. Структура пакета управляющего пакета вируса MegaD.

Поход из работы [22] был развит в двух направлениях. Во-первых, было снято ограничение на то, что должна существовать единственная граница между фазами дешифрации и разбора. Вместо этого на основании тех же эвристик производилась идентификация вызовов (и, соответственно, функций) шифрации/дешифрации, после чего все вызовы этих функций, параметры которых оказывались "помечены" (зависели от потока входных данных) считаются границами между этими фазами. Во-вторых, методика была обобщена на случай шифрации исходящих пакетов (ReFormat обрабатывал только входящий трафик). Для идентификации буферов, содержащих данные сообщения непосредственно перед вызовом функции шифрации, применяется следующий алгоритм:

1. Для всех вызовов функции дешифрации вычисляется читаемое множество (read set)
2. Восстанавливаются значения всех этих буферов (фактические параметры вызовов)
3. Для каждого вызова выбираются те буферы, содержимое которых меняется от вызова к вызову. Это обусловлено тем, что читаемые данные содержат два типа информации - собственно данные для

шифрации и таблицы шифрования, которые не должны менять своё содержимое.

На данный момент, в рамках среды TrEx разрабатывается алгоритм автоматического поиска функций шифрации/дешифрации, на основе комбинации подходов ReFormat и Dispatcher. Результаты работы этого алгоритма представляют собой два вида информации:

- Модели функций шифрации/дешифрации, добавляемые в систему описания моделей
- Семантика функций и их параметров-буферов, добавляемая в систему описания семантики

5.2. Другие приложения результатов восстановления формата

При рассмотрении формата BMP приводились примеры возможных различий при реализации разборщика этого формата. Это может служить примером отличия формата, описанного в документе, от формата, поддержка которого реализуется в приложении. Так, например, приложения не проверяющие, что зарезервированные поля заголовка BITMAPFILEHEADER содержат нули, по сути, поддерживают более широкий класс сообщений, чем тот, что описан в документе. Такие особенности реализации могут использоваться для идентификации конкретных приложений обработчиков (или их классов). Подход, занимающийся изучением таких приложений, называется deviation detection (поиск отклонений). Его результаты, в свою очередь могут использоваться для поиска ошибок (error detection) и создания «отпечатков» (fingerprint generation) конкретных приложений, использующихся для их однозначной идентификации. Анализ отклонений при поиске ошибок обусловлен тем, что указанные особенности реализации часто могут приводить к ошибкам. Указанный подход подробно рассмотрен в работе [25].

6. Заключение

Большое количество работ по теме восстановления протоколов и форматов файлов от различных коллективов по всему миру, говорит о том, что это направление - бурно развивающаяся область. Получено множество важных результатов, в том числе и для зашифрованных протоколов, которые позволяют говорить о том, что в ближайшие годы, по-видимому, процесс решения данной задачи будет автоматизирован настолько, что трудозатраты на анализ каждого нового протокола будут сведены до разумных пределов. Это позволит анализировать протоколы гораздо менее подготовленным специалистам. В свою очередь, это повлечёт за собой бурный рост количества анализируемых протоколов (качество анализа также возрастёт), что позволит, с одной стороны, в сжатые строки устранять ошибки во вновь разрабатываемых протоколах, а с другой, обнаружить и ликвидировать "дыры" в тех, что широко используются на сегодняшний день.

Кроме того, уже сейчас в связи с высоким уровнем разработанности задачи восстановления форматов появляется большое количество потенциальных приложений результатов её решения:

- разработка open-source ПО на основе закрытых протоколов. Примером в области форматов файлов может служить пакет OpenOffice, поддерживающий закрытый формат doc фирмы Microsoft, а в области сетевых протоколов - мультипротокольный интернет-пейджер qutIM, поддерживающий, в частности, закрытый протокол ICQ OSCAR[26] (спецификация этого протокола была позже открыта и опубликована)
- использование форматов в антивирусных средствах и ПО для обнаружения и предотвращения вторжений (IDS и IPS соответственно), развитие подходов на основе сигнатурной фильтрации (signature-based filtering)
- автоматическая идентификация приложения по генерируемому и принимаемому им сетевому трафику (application fingerprinting). В частности это может использоваться в системах автоматического анализа и фильтрации трафика (DPI системы)
- автоматический анализ протоколов зловредного ПО (malware), что особенно актуально для программ, создающих ботнеты. Такой анализ позволяет реагировать на появление нового ботнета в режиме реального времени, контролировать его размер и уничтожать, не допуская чрезмерного разрастания
- автоматическая генерация сообщений протокола с целью его тестирования, например, по технологии фаззинга (fuzzing)
- улучшенный анализ сетевого трафика на ключевых узлах обмена с целью получить точную статистику и принять необходимые административные и технические решения для обеспечения качества связи

Список литературы

[1] J.Lim, T. Reps B. Liblit. Extracting Output Formats from Executables. // Proceedings of the 13th Working Conference on Reverse Engineering, 2006, pp. 167—178.

[2] J. Caballero , H. Yin ,Z. Liang ,D. Song . Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. // In Proceedings of the 14th ACM Conference on Computer and and Communications Security, 2007, pp. 317—329.

[3] Z. Lin, X. Jiang, D. Xu, X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. // In Network and Distributed System Security, 2008.

[4] G. Wondracek, P. Milani Comparetti, C. Kruegel, E. Kirda. Automatic Network Protocol Analysis. // In 15th Symposium on Network and Distributed System Security, 2008.

[5] W. Cui, M. Peinado, K. Chen, H. J. Wang, L. Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. // Proceedings of the 15th ACM conference on Computer and communications security, 2008, pp. 391—402.

[6] W.Cui, J. Kannan, H. J. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. // Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, 2007, pp. 14:1—14:14
[7] А.И. Гетьман, Ю.В. Маркин, В.А. Падарян, Е.И. Щетинин. Восстановление формата данных. // Труды Института системного программирования РАН, том 19, 2010, стр. 195-214
[8] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды Института системного программирования РАН, том 16, 2009, стр. 51-72
[9] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. // In Proceedings of the Network and Distributed System Security Symposium (NDSS 2005), 2005
[10] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in X86 Executables. // In 13th International Conference on Compiler Construction, 2004. pp. 5—23
[11] S. Needleman, C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. // Journal of molecular biology. 1970 Mar;48(3), pp. 443-453.
[12] Wireshark. http://www.wireshark.org/ дата обращения 22 апреля 2012
[13] Network Driver Interface Specification 5.1 http://msdn.microsoft.com/en-us/library/ff556916.aspx дата обращения 22 апреля 2012
[14] Trivial File Transfer Protocol (revision 2) http://www.ietf.org/rfc/rfc1350.txt дата обращения 22 апреля 2012
[15] DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION http://www.ietf.org/rfc/rfc1035.txt November 1987
[16] Domain Name System. Obsoleting IQUERY http://www.ietf.org/rfc/rfc3425.txt November 2002
[17] Portable Executable and Object File Format Specification http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/pecoff.doc дата обращения 22 апреля 2012
[18] Bitmap Storage http://msdn.microsoft.com/en-us/library/dd183391(VS.85).aspx дата обращения 22 апреля 2012
[19] J. Caballero, P. Poosankam, C. Kreibich, D. Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. // Proceedings of the 16th ACM conference on Computer and communications security (CCS '09), 2009. pp. 621—634
[20] Weiser M. Program slicing // Proceedings of the 5th International Conference on Software Engineering. — IEEE Computer Society Press, 1981. pp. 439—449
[21] G. A. Venkatesh: The Semantic Approach to Program Slicing. // Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI '91). pp. 26— 28.
[22] Z. Wang, X. Jiang , W. Cui, X. Wang, M. Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. // Proceedings of the 14th European conference on Research in computer security (ESORICS'09). Springer-Verlag Berlin, Heidelberg, 2009. pp. 200—215
[23] J. Caballero. Grammar and Model Extraction for Security Applications using Dynamic Program Binary Analysis. / PhD thesis in Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, September 2010
[24] OSCAR "Open System for Communication in Realtime" http://iserverd1.khstu.ru/oscar/ Обновлено 07.02.2005

Recovery of binary data structures from program traces

A.I. Avetisyan, A.I. Getman
ISP RAS, Moscow, Russia
{arut, thorin}@ispras.ru

Abstract. In this paper we consider the problem of recovery of binary data formats and describe the format recovery system implemented in ISP RAS. First, we enumerate general approaches to this problem, their advantages and constraints: static, dynamic and network trace analysis. Here we also describe the fundamental dynamic analysis constraint (incomplete code coverage) and several possible methods to partly compensate it in this particular problem. Second, we discuss data sources and features of analysis of such objects as files, network packets of different levels and different kinds of protocols (stateful and stateless), incoming and outgoing messages. We also discuss the problem of protocol analysis and specifically the problem of recovering the protocol state machine. Third, we describe our function specification facility that allows us to define models of functions and their parameters and brings additional accuracy to our format recovery approach through taking into consideration user's knowledge about the features of a specific software environment. In this paper we also present the general scheme of our approach and test results of the implemented system. Finally, we discuss future research directions: encrypted traffic analysis and several possible applications for recovery results.

Keywords: dynamic analysis, binary trace, format recovery, protocol analysis

References

[1]. Lim J., Reps T., Liblit B. Extracting Output Formats from Executables. Proceedings of the 13th Working Conference on Reverse Engineering, 2006. pp. 167—178.

[2]. Caballero J., Yin H., Liang Z., Song D. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. Proceedings of the 14th ACM Conference on Computer and and Communications Security, 2007. pp. 317—329.

[3]. Lin Z., Jiang X., Xu D., Zhang X. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. Proceedings of the 15th Symposium on Network and Distributed System Security, 2008.

[4]. Wondracek G., Kruegel C., Kirda E., Milani P. Automatic Network Protocol Analysis. Proceedings of the 15th Symposium on Network and Distributed System Security, 2008.

[5]. Cui W. , Peinado M., Chen K., Wang H. J., Irun-Briz L. Tupni: Automatic Reverse Engineering of Input Formats. Proceedings of the 15th ACM conference on Computer and communications security, 2008. pp. 391—402.

[6]. Cui W., Kannan J., Wang H. J. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, 2007. pp. 14:1—14:14

[7]. Getman A.I., Markin YU.V., Padaryan V.A., Shhetinin E.I. Vosstanovlenie formata dannykh [Data format recovery]. Trudy ISP RAN [The Proceedings of ISP RAS], 2010, vol. 19, pp. 195-214 (in Russian)
[8]. Padaryan V.A., Getman A.I., Solov'ev M. A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software framework for dynamic binary code analysis]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009. pp. 51-72 (in Russian).
[9]. Newsome J., Song D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Proceedings of the Network and Distributed System Security Symposium (NDSS 2005), 2005.
[10]. Balakrishnan G., Reps T. Analyzing Memory Accesses in X86 Executables. Proceedings of the 13th International Conference on Compiler Construction, 2004. pp. 5-23
[11]. Needleman S., Wunsch C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology. 1970 Mar;48(3), pp. 443-453.
[12]. Wireshark. http://www.wireshark.org April, 2012
[13]. Network Driver Interface Specification 5.1. http://msdn.microsoft.com/en-us/library/ff556916.aspx April, 2012
[14]. Trivial File Transfer Protocol (revision 2). http://www.ietf.org/rfc/rfc1350.txt April, 2012
[15]. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION http://www.ietf.org/rfc/rfc1035.txt November 1987
[16]. Domain Name System. Obsoleting IQUERY http://www.ietf.org/rfc/rfc3425.txt November 2002
[17]. Portable Executable and Object File Format Specification http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/pecoff.doc April, 2012
[18]. Bitmap Storage http://msdn.microsoft.com/en-us/library/dd183391(VS.85).aspx April, 2012
[19]. Caballero J. , Poosankam P., Kreibich C., Song D. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. Proceedings of the 16th ACM conference on Computer and communications security (CCS '09), 2009. pp. 621-634
[20]. Weiser M. Program slicing. Proceedings of the 5th International Conference on Software Engineering. IEEE Computer Society Press, 1981. pp. 439-449
[21]. Venkatesh G. A. The Semantic Approach to Program Slicing. Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI '91). pp. 26-28.
[22]. Wang Z., Jiang X. , Cui W., Wang X., Grace M. ReFormat: Automatic Reverse Engineering of Encrypted Messages. Proceedings of the 14th European conference on Research in computer security (ESORICS'09). Springer-Verlag Berlin, Heidelberg, 2009. pp. 200-215
[23]. Caballero J. Grammar and Model Extraction for Security Applications using Dynamic Program Binary Analysis. PhD thesis in Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, September 2010
[24]. OSCAR "Open System for Communication in Realtime" http://iserverd1.khstu.ru/oscar/ February, 2005

Инкрементальный анализ исходного кода на языках C/C++

В.О. Савицкий ssavitsky@ispras.ru, Д.В. Сидоров sidorov@ispras.ru [1]

Аннотация. В статье описан метод построения статического анализатора кода, позволяющий существенно сократить время повторного поиска дефектов для языков C\C++. Для этого используется свойство малого отношения количества лексем из исходного файла к количеству лексем из заголовочных файлов. Метод реализован в программном продукте для среды разработки MS Visual Studio.

Ключевые слова: статический анализа кода, повторный поиск дефектов, C\C++

1. Введение.

Одной из наиболее распространенных проблем, возникающих в процессе разработки программного обеспечения, является проблема недостаточной надежности. Поэтому неудивительно, что существует большое количество современных средств, позволяющих проводить анализ программ с целью выявления тех или иных дефектов. Анализ, выполняемый по исходному коду до выполнения программы, называется статическим. Такой анализ подходит для обнаружения ошибок реализации программы. Если программист заметит предупреждение о реальной ошибке при компиляции программы, то исправить ее не составит труда. Сложнее предоставить решение, если ошибку обнаружил пользователь, который приобретает готовый продукт. Еще труднее диагностировать и исправить ошибку, если программа установлена во встраиваемой системе. Исследования на конкретных продуктах показывают, что рост времени, затраченного на устранение дефекта, экспоненциален. То есть чем раньше обнаружен дефект, тем проще его исправить.

Статический анализ довольно сложен и обычно выполняется на мощной серверной машине для всего проекта раз в сутки. Программист просматривает результаты через какой-либо интерфейс, затем вносит исправления и на следующий день снова проверяет результаты. В таком режиме дефекты могут

[1] Работа проводится в рамках реализации ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы», ГК № 07.514.11.4040 от 29.09.2011. Работа проводится при поддержке программы «Университетский кластер», http://www.unicluster.ru.

попасть, например, в ночную сборку, которая будет использоваться тестировщикам. Кроме того, на следующий день программист может уже переключиться на другую задачу, и ему придется потратить время на то, чтобы переключаться между вчерашней и сегодняшней задачами. Очевидно, большой интерес представляет возможность сокращения времени этого цикла. Так, если дефекты будут отображаться по мере ввода и редактирования текста программы, то время, затраченное на исправление, будет минимально. Рассмотрим метод инкрементального анализа исходного файла, который позволяет существенно сократить это время.

2. Обзор существующих решений

Рассмотрим общую схему работы статического анализатора. На начальной стадии она совпадает со схемой обычного компилятора.

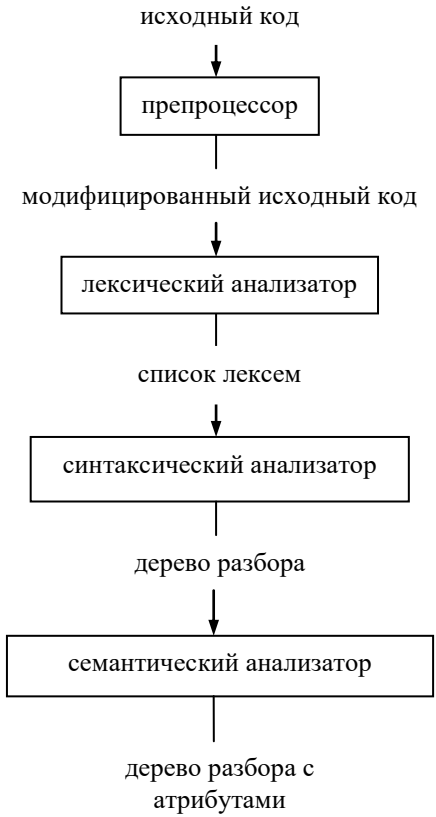


Рис. 1.

Для языков C\C++ вначале исходный текст обрабатывается препроцессором, который заменяет макросы и раскрывает директивы подключения заголовочных файлов. Затем на стадии лексического анализа полученный текст разбивается на распознаваемые единицы текста - лексемы, которые передаются синтаксическому анализу (парсеру). Синтаксический анализ определяет, может ли такая последовательность лексем соответствовать грамматике анализируемого языка, и строит дерево разбора, отражающее синтаксическую структуру исходной программы. После синтаксического анализа или одновременно с ним выполняется семантический анализ. Семантический анализ осуществляет проверку типов, вычисляет значения выражений времени компиляции, разрешает вызовы функций и снабжает узлы дерева разбора атрибутами – семантическими элементами. Простой поиск дефектов может осуществляться уже на этой стадии путем обхода дерева разбора и анализом атрибутов. К примеру, можно обнаружить сравнение переменной целочисленного беззнакового типа с нулем. Для этого в узле дерева, соответствующего бинарному выражению с оператором сравнения, необходимо проверить типы и значения операндов.

```
void foo(unsigned int n_boxes) {
 …
  if (n_boxes < 0)  { // always false
    …
  }
}
```

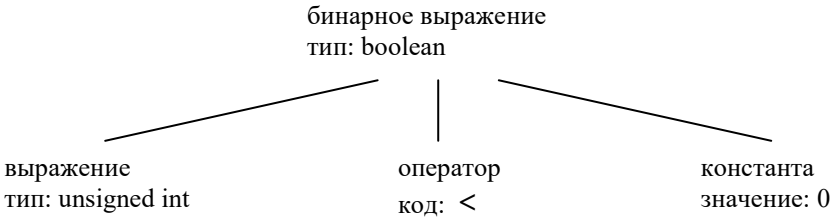


Рис.2.

Например, в продукте Klocwork для описания таких дефектов используется язык KAST [2]. Этот язык позволяет описывать **чекеры** - шаблоны поиска поддеревьев с условиями. Если при выполнении обхода дерева его часть соответствует указанному шаблону и выполнены заданные условия, то поддерево помечается как содержащее дефект. Например, для приведенного дефекта шаблон может выглядеть так:

```
// BinaryExpr
   [ OpCode = OP_LESS | OpCode = OP_GREATER_OR_EQUAL ]
```

```
      [ Left.isUnsigned() ]
      [ Right.getValue() = 0]
```

Поиск более сложных дефектов, таких как разыменование нулевого указателя или переполнение буфера, требует более сложного анализа.

```
void foo(user_type *element) {
  *element = getNext(element); // not checked for NULL
}
int main() {
  …
  element = NULL;
  …
  foo(element);
  …
}
```

Для того, чтобы определить, не является ли аргумент при вызове функции foo нулевым указателем, необходим анализ значений объектов в каждой точке программы. Такой анализ называется анализом потоков данных. Для этого дерево разбора преобразуется в промежуточное представление [1], содержащее информацию о базовых блоках (последовательность инструкций без ветвления) и вызовах функций, которые изменяют значения переменных. Анализ потоков данных существенно сложнее поиска дефектов по дереву разбора и, как правило, занимает больше времени.

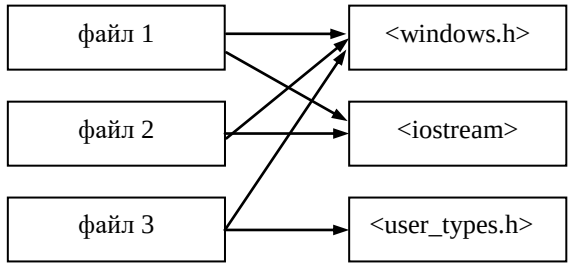
Когда программист изменяет исходный код, необходимо заново построить синтаксическое дерево, выполнить на нем анализ и сравнить новый список дефектов со старым. Для того, чтобы быстро построить новое дерево разбора, можно строить его инкрементально. История исследования возможности обновления только изменившихся частей дерева ведется еще с 70х годов. В работах [7-11] рассматриваются вопросы построения парсеров LALR(1) и GLR грамматик для работы в инкрементальном режиме. От такого парсера требуется сохранение множества состояний в каждой точке входной последовательности и поиск цепочки узлов дерева для замены при построении узлов для новых лексем. Так как возможных состояний парсера очень много, то рассматриваются оптимизации хранения этих данных в памяти. Не менее сложной является задача поиска минимального множества зависимых узлов для замены, поэтому рассматриваются возможности сокращения времени этого поиска. Любой метод построения анализатора, позволяющего инкрементальное построение дерева разбора, предполагает внесение серьезных изменений в существующие генераторы парсеров или же реализацию новых сложных систем. Покажем, что небольшой доработкой существующего анализатора для языков C и C++ можно значительно ускорить построение дерева разбора, если изменения не касаются подключаемых заголовочных файлов.

Разработчики компиляторов всегда искали пути оптимизации времени обработки кода. Для проектов, написанных на языках C и C++, важную роль сыграло следующее наблюдение. Обычно структура каждого файла такова: в начале записаны директивы препроцессора для подключения заголовочных файлов, затем следуют определения, относящиеся непосредственно к этому файлу.

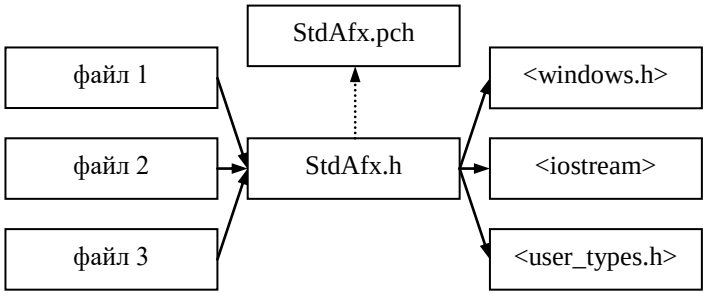
```
#include <windows.h>
#include <iostream>
#include <user_types.h>

typedef unsigned int boxes;
```

В подключаемые файлы обычно выносятся декларации и определения, которые могут использоваться во многих файлах. Каждый файл может использовать лишь небольшое количество определений из заголовочного файла, но вынужден подключать его целиком. Кроме того, одни и те же заголовки анализируются много раз. В результате время компиляции файла и всего проекта неоправданно увеличивается. Естественное решение в таком случае обрабатывать общие заголовочные файлы лишь однажды. Такая технология получила название "предварительно откомпилированные заголовки". Например, в варианте, реализованном в компиляторе MS Visual Studio [3], для всего проекта или его части создается один общий заголовочный файл, в который подключаются все заголовочные файлы, которые могут использоваться в разных исходных файлах проекта. Затем директивы подключения заголовков в каждом файле заменяются одной директивой подключения этого общего заголовочного файла.



без предварительно
откомпилированного заголовка



с предварительно
откомпилированным заголовком

Компилятору сначала передается команда создания предварительно откомпилированного заголовка. В результате компилятор создает файл, в который записывает таблицу макросов препроцессора, таблицу найденных идентификаторов, и другие данные, необходимые для последующего разбора файлов. При дальнейшей сборке проекта, компилятор тратит время только на загрузку сохраненной информации. Это существенно сокращает время анализа каждого отдельного файла. Несколько иной подход используется в компиляторе gcc [4]. gcc позволяет скомпилировать заголовочный файл как обычный файл проекта, при этом будет создан файл с расширением .gch в директории, в которой находится заголовок. Затем во время обработки директив подключения заголовков компилятор сначала ищет соответствующий .gch файл и использует его, если это возможно. Возможен и другой способ - часто используемые заголовки собираются в один. Этот заголовок компилируется и подключается к файлам проекта при помощи опции **–include**. Если в заголовочных файлах присутствует защита подключения (include guard), то заголовки из общего файла не будут обрабатываться повторно. Метод компилятора gcc предпочтительнее метода, используемого компилятором MS Visual Studio, поскольку не требует вносить изменения в исходный код проекта.

3. Анализ «на лету».

Существенное отличие выполнения статического анализа в среде разработки от анализа полной сборки проекта состоит в том, что пользователь одновременно работает лишь с небольшим количеством исходных файлов. Однако по мере ввода и редактирования файла, один и тот же исходный код с небольшими изменениями должен анализироваться вновь и вновь. Поэтому

исключительное значение приобретает скорость выполнения анализа. Будем пользоваться следующим наблюдением. Анализ нескольких проектов с открытым кодом показывает, что более 90% символов, обрабатываемых компилятором, приходят из заголовочных файлов. В таблице приведены результаты сборки проектов firefox-9.0 и boost 1.47 на платформе Linux с раздельным подсчетом лексем из заголовочных и исходного файлов.

проект	Firefox-9.0	Httpd-2.2.22	Boost 1.47
количество файлов в проекте	3,523	391	569
А) количество лексем из всех исходных файлов	13,932,951	1,037,941	214,114
Б) количество лексем из всех заголовочных файлов	477,861,983	9,294,281	195,378,599
Отношение количества А к Б	0.029	0.112	0.001
Максимальное количество лексем в одном исходном файле	414,422	37,690	6,201
Максимальное количество лексем в заголовочных файлах, подключенных в один исходный	832,767	44,920	977,346

Как показывает практика, на платформе Windows объем системных заголовочных файлов еще больше. Приведем статистику сборки тех же проектов на этой платформе:

проект	Firefox-9.0	Httpd-2.2.22	Boost 1.47
количество файлов в проекте	3,753	425	304
А) количество лексем из всех исходных файлов	14,037,713	2,601,184	107,671
Б) количество лексем из всех заголовочных файлов	667,775,667	30,542,791	103,001,359
Отношение количества А к Б	0.021	0.085	0.001
Максимальное количество лексем в одном исходном файле	406,213	143,556	5,984
Максимальное количество лексем в заголовочных файлах, подключенных в один исходный	877,804	180,374	1,406,616

Программист в процессе работы обычно дополняет или изменяет уже имеющийся код, редко изменяя подключенные заголовочные файлы. Поэтому если сохранить результаты анализа заголовочных файлов, то можно существенно сократить время, затрачиваемое на повторный запуск анализа. Рассмотрим алгоритм подробнее

4. Описание алгоритма.

Для каждого открываемого в среде разработки файла анализатор будет создавать кэш следующего вида. Анализатор обрабатывает входную последовательность символов, отслеживая момент появления первой лексемы непосредственно из текста файла. На рисунке это место отмечено пунктирной линией.

```
#include <stdio.h>
#include <stdlib.h>
-------------------------------------------------
typedef char number;

#include <inlined.h>
```

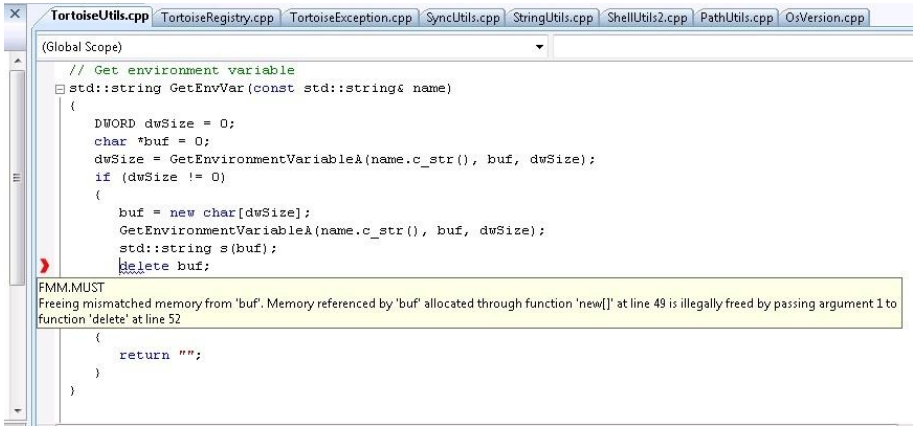
Заметим, что в данном примере последний заголовок не попадет в кэш, так как директива его подключения находится после первой лексемы из исходного файла, но это редко встречающаяся ситуация. В момент обнаружения такой лексемы в памяти сохраняется "снимок" информации, обработанной к этому моменту. В него включены таблица символов препроцессора, таблица символов парсера, необходимые семантические элементы, а также состояние области видимости идентификаторов. Также сохраняется состояние работы чекеров, которым для обнаружения дефекта нужна информация о файле целиком. Затем анализируется оставшаяся часть файла. После того как пользователь изменяет исходный файл, с некоторой задержкой запускается поиск дефектов.

Вначале необходимо перевести анализатор в состояние, которое он имел на момент создания снимка. Например, для семантического анализатора это означает удаление из областей видимости заголовков, тех элементов, которые были внесены при анализе самого исходного файла. Также необходимо освободить память, выделенную динамически для структур из основной части текста исходного файла. Для того, чтобы упростить переход к состоянию до анализа исходного файла, можно создать пулы памяти - один для всех заголовочных файлов, и один для основной части файла. Тогда память будет освобождаться в одно действие, при условии, что из пула, соответствующего заголовочным файлам, будут удалены все ссылки на элементы во втором пуле. Затем синтаксический анализатор начинает анализ с первой лексемы после заголовочных файлов. Построенное по файлу дерево синтаксиса инкрементально дополняет сохраненное дерево для заголовков. В результате

полное дерево синтаксиса не будет отличаться от построенного при полной сборке.

Поиск дефектов по дереву синтаксиса происходит довольно быстро, поэтому не имеет смысла ограничивать область поиска только областью файла, в которой произошли изменения - например, классом или функцией. Но эта оптимизация имеет большое значение для ускорения построения промежуточного представления и анализа потоков данных. Для этого при обходе дерева для каждой функции сохраняется контрольная сумма, посчитанная по лексемам. При последующих итерациях анализ ограничивается функциями с изменившимися контрольными суммами, и зависящими от них объектами.

После обнаружения дефектов они отображаются в окне с исходным кодом в виде колонки с маркерами либо подчеркиваются прямо в коде. На рисунке показано, как отображаются дефекты в плагине Klocwork для Visual Studio.



Имеет смысл запускать анализ с небольшой задержкой (например, полсекунды) после того, как пользователь прекратит набор текста. Зачастую, исходный код программы в этот момент синтаксически некорректен. Поэтому большое значение приобретает обработка ошибок. Синтаксический анализатор должен правильно восстановиться после обнаруженной ошибки, а также запретить дальнейший поиск дефектов, иначе велика вероятность ложных срабатываний и пропуска настоящих дефектов. Необходимо также проверять, не изменился ли контекст с момента сохранения "снимка" для заголовочных файлов. В данном случае под контекстом понимается набор опций компилятора, содержание заголовочных файлов и набор дефектов для поиска. Если один из этих компонентов меняется, следует запустить анализ для заголовочных файлов заново.

Постоянное хранение кэша в памяти требует аккуратного управления ресурсами. Когда пользователь открывает все новые и новые файлы в среде разработки, необходимо поддерживать объем доступной памяти не ниже

определенного порога. Он может определяться, например, так: все запущенные процессы анализа должны занимать не более 75% свободной памяти. Конечно, выгружать лишние процессы необходимо так, чтобы пользователь имел возможность быстро получать результаты анализа для активно редактируемых файлов. Если файлы проекта при сборке явно используют один и тот же предварительно откомпилированный заголовок, то все эти файлы можно обрабатывать одним процессом с общим кэшем, что облегчает задачу освобождения памяти. Так часто обстоит дело с проектами Microsoft Visual Studio. Подобная техника оптимизации времени построения синтаксического дерева может применяться не только для статического анализа, но и для задач автоматического рефакторинга кода [5]. Большинство правил рефакторинга – изменения структуры кода, не затрагивающего поведения программы – можно реализовать, используя синтаксическое дерево с атрибутами. Если пользователь будет проводить много операций по изменению кода на одном файле, то быстрое обновление синтаксического дерева будет очень полезно.

5. Результаты.

Результаты работы описанного метода оценивались следующим образом. Была написана программа, которая запускает статический анализ на файле один раз и измеряет время первого запуска. Затем следуют еще 9 итераций, на которых в файл вносятся произвольные изменения и измеряется время повторного анализа.

проект	firefox-9.0	httpd-2.2.22	boost 1.47
среднее время первого анализа, с	1.6	0.8	3
среднее время последующих стадий анализа, с	0.16	0.01	0.1
среднее количество обнаруженных дефектов	77	31	37

Как видно из таблицы, поставленная цель достигнута: за счет инкрементального анализа файла время повторных стадий анализа более чем в 10 раз меньше времени первого анализа, что позволяет перезапускать анализ практически на каждое нажатие клавиши.

6. Заключение.

Описанная техника существенно сокращает время поиска дефектов на исходном файле. Это позволяет программисту своевременно исправлять

возможные дефекты. Технология была реализована нами в коммерческом продукте для статического анализа Klocwork 9.5. Дальнейшим развитием оптимизации времени анализа может стать дополнение инкрементального анализа «ленивым». При таком подходе анализ областей видимости блоков кода откладывается до момента непосредственного обращения к элементам из этой области.

Литература.

[1] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[2] С.В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST.
[3] MS Visual Studio precompiled headers http://msdn.microsoft.com/en-us/library/szfdksca(v=vs.71).aspx.
[4] GCC Using precompiled headers http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.
[5] Martin Fowler. Refactoring: Improving the Design of Existing Code.
[6] Andrew Hunt and David Thomas The Pragmatic Programmer: From Journeyman to Master Addison-Wesley, Oct 1999.
[7] J.-M. Larcheveque *Optimal Incremental Parsing* ACM Transactions on Programming Languages and Systems, Vol. 17, No 1, January 1995.
[8] Tim A. Wagner and Susan L. Graham *Incremental Analysis of Real Programming Languages* 1997 ACM.
[9] Dashing Yeh, Uwe Kastens *Automatic Construction of Incremental LR(1) Parsers*
[10] Carlo Ghezzi, Dino Mandrioli *Incremental Parsing* ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July 1979.
[11] Mats Wirdn *Minimal Change and Bounded Incremental Parsing* .

Incremental source code analysis for C/C++ languages.

V. O. Savitsky, D. V. Sidorov
ssavitsky@ispras.ru, sidorov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. The article describes a way of implementing static code analysis for C/C++ languages that allows to reduce dramatically the time of the second and following analysis iterations. Main idea behind algorithm based on fact that programs on C/C++ languages widely use common parts of code placed inside header files with declarations and such header files included inside of several compilation units of projects. The ratio of the number of tokens in the source file to the number of tokens in the included headers is small. It is possible to reduce time of analysis using cache of compilation results for rarely changed part of compilation unit and use it in following iterations of analysis during work inside modern IDEs. The method is implemented in a commercial product for MS Visual Studio. The result of implementation described method is reduced more that in ten times time needed to analyze changed source code. Such performance improvement allows to re-analyze code on every changed symbol inside IDE and issue defects found in source code on-the-fly while developer creates program. This method is implemented inside of product Klocwork inSight 9.5

Keywords. static code analysis, second and following analysis iterations, C/C++

References

[1]. Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[2]. S.V. Syromyatnikov. Deklarativnyj interfejs poiska defektov po sintaksicheskim derev'yam: yazyk KAST **[**Declarative Interface of Detecting Defects on Syntax Trees: KAST Language**]**. *Trudy ISP RAN* [*The Proceedings of ISP RAS*]. 2011, vol. 20, pp. 51-68 (in Russian).
[3]. MS Visual Studio precompiled headers http://msdn.microsoft.com/en-us/library/szfdksca(v=vs.71).aspx.
[4]. GCC Using precompiled headers http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.
[5]. Martin Fowler. Refactoring: Improving the Design of Existing Code.
[6]. Andrew Hunt and David Thomas The Pragmatic Programmer: From Journeyman to Master Addison-Wesley, Oct 1999.
[7]. J.-M. Larcheveque Optimal Incremental Parsing ACM Transactions on Programming Languages and Systems, Vol. 17, No 1, January 1995.
[8]. Tim A. Wagner and Susan L. Graham Incremental Analysis of Real Programming Languages 1997 ACM.
[9]. Dashing Yeh, Uwe Kastens Automatic Construction of Incremental LR(1) Parsers
[10]. Carlo Ghezzi, Dino Mandrioli Incremental Parsing ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July 1979.
[11]. Mats Wirdn Minimal Change and Bounded Incremental Parsing .

Комбинированный (статический и динамический) анализ бинарного кода

А.Ю.Тихонов, А.И. Аветисян
{fireboo@ispras.ru}, {arut@ispras.ru}

Аннотация. Рассматриваются проблемы анализа программы в бинарном коде для распознавания алгоритмов составляющих ее функций, выяснения особенностей реализации алгоритмов, обнаружения недокументированных возможностей. Традиционно для этих целей используются дизассемблеры и средства статического анализа потоков данных. Однако в случаях, когда составители программы приняли меры для защиты своей программы от анализа (например, использовали запакованный код, который распаковывается во время выполнения программы), статический анализ может не дать результатов. В данной статье предлагается использовать в таких случаях динамический анализ (анализ трасс программы) в дополнение к статическому. Рассматриваются проблемы, возникающие при анализе бинарных программ, и предлагаются способы автоматизации их решения. Описывается среда динамического анализа бинарного кода TrEx, разработанная и реализованная авторами статьи, и ее интеграция с известной средой статического анализа Ida Pro. Приведены примеры анализа конкретных бинарных программ.

Ключевые слова: восстановление алгоритмов по бинарному коду, поиск недокументированных возможностей в бинарном коде, методы получения трасс бинарных программ.

1. Введение

Основной задачей, в которой появляется необходимость в анализе бинарного кода, является ***задача понимания программ***, возникающая в целом ряде практических ситуаций: анализ вредоносного кода, когда исходный код отсутствует, либо недоступен; поиск недекларированных возможностей (далее – НДВ) в коммерческом ПО, которое поставляется без исходных кодов, в том числе в рамках сертификационных испытаний; поддержка унаследованных приложений, когда бинарный код приложения в результате многочисленных исправлений перестаёт соответствовать исходному коду. Целью таких исследований является получение информации об особенностях реализации алгоритмов, распознавание алгоритмов, восстановление протоколов обмена данными и форматов данных.

Для анализа бинарного используются методы статического и динамического анализа. При статическом анализе код исследуется без выполнения самой

программы, при динамическом анализе, программа исследуется на основании информации, полученной во время ее выполнения. Следует отметить, что при анализе бинарного кода могут применяться те же методы статического анализа, что и для исходного кода. Однако их применение требует предварительного получения графа вызовов функций программы и набора графов потока управления для каждой функции. Некоторые виды анализа потока данных требуют кроме того построения графа зависимостей (по данным и управлению). При различных анализах потока данных отслеживаются значения атрибутов, которые сопоставляются переменным, что приводит к необходимости наличия информации о переменных, использующихся в программе, их типах и других атрибутах.

В случае анализа исходного кода программа представлена на языке высокого уровня, что позволяет достаточно просто получить представление программы в виде набора графов: вызовы функций осуществляются по именам, передачи управления по меткам, все использующиеся переменные и их типы описаны явно. Единственным исключением является вызов функций по указателю, при котором не всегда можно сказать, какая функция (или их набор) вызывается в данной точке программы. Но и в этом случае доступен прототип вызываемой функции, что часто позволяет существенно сузить набор потенциально вызываемых функций. В бинарном коде отсутствуют переменные и типы данных в явном виде – вместо этого инструкции оперируют регистрами и ячейками памяти, отсутствуют и явные описания прототипов функций и их границ в коде.

Следует отметить, что бинарный код программы, как правило, бывает представлен в виде исполняемого файла некоторого известного формата (например, PE32 [1] для ОС Windows или ELF [2] для Linux). В данной работе предполагается, что анализируемый бинарный код представлен в виде *стандартной модели компиляции*, т.е. удовлетворяет следующим условиям:

- исполняемый файл содержит функции, глобальную область данных, кучу;
- могут использоваться виртуальные функции и динамически загружаемые библиотеки;
- поддерживается стек времени выполнения;
- каждая статическая (глобальная) переменная находится по фиксированному смещению в адресном пространстве программы;
- локальные переменные хранятся в стековом фрейме функции по фиксированным смещениям;
- фактические параметры функций передаются через стек вызывающей функцией, так что формальным параметрам функции соответствуют фиксированные смещения в её фрейме.

Кроме того, предполагается, что компоненты бинарного кода программы занимают фиксированные области памяти, и что код не является

самомодифицирующимся. Следует отметить, что этим предположениям удовлетворяет широкий класс программ, получаемых компиляцией с языков C/C++.

Исполняемый файл содержит области кода и статических данных программы, таблицы функций, импортируемых программой из сторонних, динамически загружаемых библиотек, а также список функций, экспортируемых программой, в случае, если программа сама является библиотекой. Область кода содержит последовательность функций программы, которая не является непрерывной, так как между функциями могут возникать неиспользуемые участки памяти вследствие выравнивания. Кроме того, многие компиляторы вставляют часть данных функции рядом с её кодом (статические данные). Библиотечные функции, которые при сборке программы были статически связаны с ней, содержатся в её коде и неотличимы от функций программы.

Одним из наиболее распространённых и развитых средств статического анализа бинарного кода является система интерактивного дизассемблирования IDA Pro [3], поддерживающая широкий ряд бинарных форматов и процессорных архитектур. При анализе исполняемых файлов IDA Pro позволяет проводить их автоматический анализ. При этом выполняется:

- выделение таблиц импортируемых и экспортируемых функций;
- дизассемблирование кода функций;
- выделение границ статических переменных в области данных;
- выделение параметров функций в стеке и на регистрах, а также локальных переменных функции и возвращаемого значения;
- построение графа вызовов и графа потока управления за счёт анализа адресов переходов;
- автоматическое распознавание библиотечных функций и их параметров с помощью технологии FLIRT [4].

При статическом анализе, выделение областей кода и данных, а также таблиц импорта и экспорта выполняется в процессе разбора исполняемого файла – данные о смещениях и размерах этих областей содержатся в заголовке. Для того чтобы выделить в области данных границы отдельных переменных требуется проанализировать команды обращения (чтения и записи) к этим данным из области кода. Аналогично, для того чтобы выделить границы функций требуется определить их начало, то есть адрес, на которой управление передаётся с помощью специальной инструкции вывода, и конец – адрес (точнее максимальный адрес, в случае если их несколько) специальной команды возврата. Перед анализом выполняется дизассемблирование кода, то есть представление его в виде листинга на языке ассемблера целевой архитектуры. Для дизассемблирования требуется отделять машинные инструкции от данных (дизассемблированию должны подвергаться только машинные инструкции). При этом, вследствие неразрешимости задачи разделения команд и данных на компьютерах с архитектурой Фон-Неймана,

проблемы могут возникать даже в случае файлов, удовлетворяющих стандартной модели компиляции. Кроме того, при дизассемблировании требуется восстанавливать корректную семантику операндов инструкции (задача различения констант и адресов). В общем случае эта задача также неразрешима.

Дизассемблирование начинается с точки входа в программу, которая прописана в заголовке исполняемого файла и идёт последовательно сверху вниз. Существует два основных алгоритма дизассемблирования – линейный алгоритм и алгоритм рекурсивного спуска.

Линейный алгоритм не анализирует поток управления программы, предполагая, что всё содержимое области кода является кодом. Алгоритм осуществляет линейный проход от начала до конца области кода, разбирая инструкции по порядку. При разборе очередной инструкции, определяется её размер, прибавляется к её смещению и получается адрес следующей инструкции. Преимуществом линейного алгоритма является высокая скорость дизассемблирования, основным недостатком – ошибки в дизассемблировании в случаях, когда код располагаться не непрерывно вследствие выравнивания, либо, когда в области кода содержатся статические данные программы.

Алгоритм рекурсивного спуска состоит в анализе потока управления, который начинается с точки (точек) входа в программу. При анализе инструкций, не осуществляющих передачу управления, алгоритм ведёт себя так же, как линейный. При анализе инструкций передачи управления вычисляется адрес перехода и добавляется в список адресов для дальнейшего анализа. После условной передачи управления или инструкции вызова функции анализ продолжается линейно, после безусловной передачи управления или инструкции возврата из функции последовательный анализ прерывается и осуществляется переход на следующий адрес в списке ранее добавленных, так как сразу за инструкцией может отсутствовать код (выравнивание, статические данные). Таким образом, алгоритм рекурсивного спуска позволяет автоматически определять области кода, соответствующие отдельным функциям.

Дизассемблирование в IDA осуществляется алгоритмом рекурсивного спуска, что приводит к необходимости вычислений адресов переходов и вызовов. В случае косвенной адресации вычисление адресов может оказаться невозможным, поэтому в IDA Pro встроен интерактивный отладчик, позволяющий получать значения динамически вычисляемых адресов. Однако его запуск и процесс экспорта динамической информации не автоматизирован, что усложняет применимость IDA Pro для больших программ.

Однако в некоторых случаях статический анализ может оказаться неприменимым. В качестве примера рассмотрим программу, снабженную защитой от анализа: код программы в исполняемом файле находится в зашифрованном виде и расшифровывается в процессе выполнения. В этом

случае можно попытаться снять дамп в тот момент, когда весь код будет расшифрован. Однако, в случае «ленивой» расшифровки (то есть выполняемой, только в момент передачи управления на соответствующий код) такого момента может и не существовать, если на текущих входных данных исполняется не весь код программы.

Статический анализ может быть осложнен и из-за отсутствия кода динамически загружаемых библиотек в исполняемом файле. В частности, могут отсутствовать компоненты ОС, с которыми взаимодействует программа. Это может помешать обнаружению зловредного кода, одним из способов внедрения которого является перезапись начала одной или нескольких системных функций с целью вставки инструкции перехода на зловредный код. Один из методов поиска и выделения механизмов внедрения зловредного кода выделения на основе динамического подхода описан в [5].

Рассмотренные примеры указывают на недостаточность статического подхода для анализа защищенного бинарного кода и целесообразность использования в подобных случаях методов динамического анализа.

Наиболее эффективным видится подход на основе комбинации динамического и статического анализа, при котором статическое представление дополняется данными, полученными в ходе динамического анализа. Интерактивный отладчик, в составе IDA Pro является одним из примеров реализации подобного подхода. Желательно, чтобы средство комбинированного анализа бинарного кода удовлетворяло следующим требованиям:

- возможность получения информации об адресах расположения кода и данных, а также вычисляемых адресах переходов и вызовов, необходимых для дополнения статического представления;
- полносистемный анализ, включающий код не только анализируемой программы, но и других модулей, в частности компонентов операционной системы;
- возможность анализа многопоточного ПО;
- поддержка различных процессорных архитектур.

В данной статье описывается система анализа бинарного кода, позволяющая комбинировать статические и динамические анализы, обеспечивая преодоление ограничений каждого подхода. Система является автоматизированной, и в ключевые моменты работы может потребовать от пользователя некоторых действий. В дальнейшем такого пользователя, обладающего дополнительными знаниями о работе программы, будем называть *аналитиком*. Примером ситуации, когда системе требуется помощь аналитика, является выбор функций, с которых следует начать анализ программы для его ускорения. Система позволяет аналитику решать такие задачи как поиск и выделение кода алгоритма, интересующего аналитика, восстановление входных и выходных данных найденного алгоритма,

восстановление интерфейсов между отдельными частями найденного алгоритма (в частности, параметров функций, используемых в алгоритме). Система использует промежуточное представление анализируемой бинарной программы [6], в котором программа представляется как последовательность инструкций абстрактной RISC-машины. Отметим, что в качестве такого промежуточного представления можно использовать и бит-код *LLVM* [7].

Дальнейшее изложение строится следующим образом. В разделе 2 описывается общая схема комбинированного подхода, основой которого является совместное использование статических и динамических методов, с использованием графа вызовов и графов потока управления функций, составляющих анализируемую программу (статический анализ) и бинарных трасс выполнения программы (динамический анализ) для выделения из них информации, уточняющей статическое представление. В разделе 3 описывается интерфейс взаимодействия компонент системы, реализующих статический и динамический анализ. В качестве компоненты статического анализа была выбрана среда *IDA Pro* [3], так как она реализует максимальный набор статических средств анализа. Раздел 4 содержит описание компоненты динамического анализа – среды *TrEx*, разработанной при участии авторов статьи. В разделе 5 рассматриваются примеры применения среды *TrEx* для решения различных задач обратной инженерии.

2. Общая схема комбинированного анализа

Общую схему комбинированного анализа, предлагаемую в данной статье, можно кратко описать следующим образом. Сначала с помощью среды *TrEx* выполняется динамический анализ программы в бинарном коде: генерируется трасса исполнения программы на уровне машинных инструкций, полученная на входных данных, выбранных таким образом, чтобы трасса включала выполнение исследуемой функциональности. Затем результаты исследования (граф вызовов и графы потока управления функций) передаются для уточнения системе *IDA Pro*.

Таким образом, бинарная программа анализируется в следующем порядке:

- Получение трассы выполнения программы, в которой выполняется интересующий аналитика сценарий работы программы.
- Структурирование трассы – выделение процессов и потоков, функций и их вызовов, построение графов потока управления выделенных функций по трассе.
- Поиск точки трассы, относящейся к исследуемой функциональности (выполняется аналитиком с использованием инструментальных средств среды *TrEx*).

- Выделение кода функций, выполняющих исследуемую функциональность (выполняется аналитиком с использованием инструментальных средств среды *TrEx*).
- Анализ выделенной функциональности – выделение входных и выходных данных функций, выполняющих исследуемую функциональность, восстановление внутренних интерфейсов между функциями (параметров функций) и связей между ними (зависимости по данным между параметрами).
- Восстановление структуры входных и выходных данных, а также параметров составных типов.
- Представление выделенной функциональности в виде ассемблерной программы (построение контрольного примера).
- Исследование свойств выделенной функциональности с применением статического подхода.

Генерация трассы, выполняется на специально оборудованном стенде с помощью одного из средств трассировки. Набор средств для решения этой задачи достаточно богат и включает как аппаратные (основанные, например, на интерфейсе JTAG), так и программные решения (например, на основе потактовых симуляторов AMD SimNow [8], Virtutech Simics [9]). Средства трассировки различаются по скорости работы и уровню изоляции процесса снятия трассы от анализируемой программы. Кроме того, класс исследуемой программы ограничивает набор средств, способных осуществить её трассировку: ограничения могут зависеть как от уровня защиты программы (например, программа может пытаться обнаружить факт собственной отладки или выполнения в симуляторе), так и от наличия связи программы с внешним миром по сети. В последнем случае критической становится скорость трассировки, так как при сильном замедлении сетевые соединения будут разрываться из-за превышения интервалов ожидания сетевых пакетов.

В трассе фиксируется не только последовательность выполнявшихся на процессоре инструкций, но и аппаратные прерывания, пользовательский ввод-вывод, приходящие извне сетевые пакеты. Таким образом, трасса содержит массив данных, описывающий все аспекты функционирования исследуемой программы.

Несомненным преимуществом автоматизированного динамического анализа (трассировки) является устойчивость ко многим методам защиты исполняемого кода от анализа, потому что эти методы защищают только форму представления защищаемого кода, не меняя его функциональности. Например, в случае применения для защиты программы *распаковщика*, когда исполняемый код хранится в зашифрованном виде и распаковывается только в момент исполнения, в трассе будет отражён как сам алгоритм распаковки, так и процесс работы распакованного кода [10]. Таким образом, исследование трассы позволяет автоматически получать статическое представление программы, путём проецирования трассы на адресное пространство, даже в

тех случаях, когда непосредственное извлечение и дизассемблирование кода из исполняемого файла невозможно. Кроме того, получение трассы с помощью симуляторов позволяет обойти большинство методов защиты кода от отладчиков.

После получения трассы требуется произвести первичное структурирование трассы: выделить код, относящийся к различным процессам и потокам. Для этого требуется либо фиксация трассировщиком точек переключения контекстов, либо наличие информации о ячейках памяти (регистрах), содержащих значения идентификаторов текущего процесса и потока.

На следующем шаге строится трасса более высокого уровня: последовательность инструкций, составляющих трассу, заменяется последовательностью вызовов функций. Это позволяет получить данные об областях в адресного пространства, соответствующих отдельным функциям, составляющим программу.

Представление трассы в виде последовательности вызовов функций позволяет построить *граф вызовов*: для каждого вызова определяется соответствующая функция и список точек вызова (возможно, пустой), лежащих внутри трассы рассматриваемого вызова. Если список точек вызова не пуст, для каждой новой точки вызова определяется соответствующая функция и в граф вызовов добавляется ребро от вызывающей функции, к вызываемой. Преимуществом такого графа вызовов является наличие в нём рёбер, соответствующих вычисляемым вызовам, которые невозможно получить, используя статический подход. Недостаток заключается в неполноте полученного графа – он содержит только вершины (функции), код которых исполнялся на заданных входных данных, и рёбра, соответствующие вызовам, которые попали в трассу.

После выделения границ кода функций для каждой функции строится *граф потока управления*, имеющий такие же преимущества и недостатки, что граф вызовов: неполнота, за счёт неполного покрытия кода функции трассой и большая точность за счёт возможности получения адресов переходов вычисляемых в процессе выполнения.

Если в состав анализируемой программы входит большое количество функций, то полученная структурированная трасса будет наряду с полезными данными содержать коды функций, не относящиеся к анализируемой функциональности. Для *отсеивания* из трассы ненужных данных необходимо найти в трассе точку, заведомо относящуюся к выполнению исследуемой функциональности: эта часть исследования выполняется аналитиком, который вручную отмечает в трассе искомую точку (*точку зацепления*).

Наиболее очевидным примером точки зацепления является точка вызова библиотечной функции, реализующей известную аналитику функциональность, например, функции ввода-вывода. Можно привести и другие подобные примеры. В частности, при исследовании алгоритмов шифрации часто бывают известны характерные константы, использующиеся

исследуемым алгоритмом. Если программа представлена в виде трассы, точки обращения к таким константам могут быть найдены с помощью алгоритма поиска состояния, осуществляющего последовательный просмотр состояний регистров на каждом шаге трассы.

В случае статического подхода для обнаружения таких точек используется механизм перекрёстных ссылок (IDA Pro). Для того чтобы найти вызов библиотечной функции нужно вначале идентифицировать эту функцию. В случае статического анализа для этого применяется сигнатурный анализ с использованием технологии сигнатурного распознавания функций FLIRT. Для успешного применения этого подхода необходимо пополнять базы данных сигнатур с выходом новых версий библиотек. Получение символьной информации возможно напрямую из исполняемых файлов известных форматов, путём её извлечения из таблицы экспорта, содержащейся в одном из заголовков файла.

При подключении символьной информации к трассе возникает дополнительная сложность, связанная с тем, что адреса символов указаны в виде смещения относительно начала модуля, и для вычисления абсолютных адресов требуются данные об адресах загрузки соответствующих модулей. Эти данные могут быть получены, например, из карты загрузки модулей перед началом снятия трассы или в некоторый момент в процессе снятия. При наличии символьной информации точки вызова известной функции могут быть найдены автоматически, однако в статическом представлении не сразу будут обнаружены вызовы по вычисляемым адресам.

Далее требуется определить место ввода начальных данных алгоритма и место вывода результатов его работы. Алгоритм реализуется некоторой функцией, или их набором, поэтому требуется определить эти функции и восстановить их входные и выходные параметры. В статическом подходе для этого может быть использован граф вызовов и анализ потоков данных. В случае анализа трассы для поиска функций можно воспользоваться динамическим стеком вызовов, что позволит определить ограниченный набор функций для анализа в графе вызовов. Восстановление параметров может осуществляться с использованием как статических, так и динамических методов, однако параметры составных типов, например, массивов, обычно передаются по указателю, что в случае статического подхода приводит к необходимости анализа алиасов [11]. Использование трассы упрощает эту проблему, так как позволяет точно вычислить адрес памяти, к которому обращается инструкция в заданном шаге трассы.

Знание входных и выходных данных позволяет выделить код алгоритма с помощью процедур статического слайсинга [12]. Их применение как правило сокращает объём анализируемого кода лишь незначительно, поэтому имеет смысл использовать динамический слайсинг [13]. При анализе трассы можно воспользоваться процедурой, аналогичной алгоритму слайсинга: из трассы выделяются только те инструкции, входных операндов которых достигли

входные данные, а также те инструкции, чьи выходные операнды повлияли на выходные данные алгоритма. В дальнейшем такой способ фильтрации шагов трассы будем называть слайсингом трассы. Полученный слайс программы содержит значительно меньшее количество инструкций и, как правило, является обозримым. Пример, показывающий эффективность этого метода приведён в разделе 5. Общая методика извлечения кода алгоритма на основе динамического подхода описана в [14].

Слайсинг трассы может использоваться и с целью дополнения входных и выходных данных алгоритма, в том случае, когда на начальном этапе известна только их часть. Пусть для определённости известна часть выходных данных алгоритма. Запустив обратный слайсинг на этом наборе данных с точки выхода алгоритма, в точке входа можно получить список ячеек, значение которых повлияло на заданный набор выходных данных. Этот набор ячеек добавляется к уже известным входным данным. Если при выполнении этой операции набор входных данных был расширен, требуется запустить прямой слайсинг по новому набору входных данных от точки входа в алгоритм и, аналогично, получить набор ячеек в точке выхода. Эта процедура применяется итерационно, пока не будет достигнуто состояние, когда оба множества (входных и выходных данных) перестанут расширяться.

По полученному слайсу может быть получен список функций, участвующих в обработке входных данных алгоритма и, следовательно, являющихся его частью. В случае если для некоторой их части доступна символьная информация – общую схему алгоритма можно понять, построив по трассе дерево вызовов, которое отражает последовательность вызовов функций, с учётом их вложенности.

Одним из удобных вариантов представления алгоритма является граф зависимостей, вершинами которого могут быть как отдельные инструкции – для изучения локальных частей алгоритма, так и функции (для получения глобальной схемы работы алгоритма). В первом случае, рёбра графа зависимостей отражают зависимости по данным между операндами инструкций, во втором – между входными и выходными параметрами функций. Для построения графа зависимостей на уровне функций используется уже упоминавшийся алгоритм слайсинга – для вычисления зависимостей по данным между входными и выходными параметрами одной функции, а также для получения зависимостей по данным между выходными параметрами одной функции и входными параметрами другой.

При работе с трассой важным этапом является проверка точности выделения алгоритма. Она осуществляется путём построения работоспособного ассемблерного листинга. Построение осуществляется по следующей схеме. Из элементов трассы извлекаются инструкции, упорядочиваются по их расположению в памяти, для константных переходов и адресов памяти генерируются метки, строятся пролог и эпилог, обеспечивающие размещение и выдачу начальных и результирующих данных.

Полученная ассемблерная программа рассматривается как контрольный пример, выполнение которого способно подтвердить корректность всех проведенных аналитиком операций, если конечно выполнение построенной программы выдает те же результаты, что и исходная программа. По требованию аналитика контрольный пример может быть декомпилирован в язык высокого уровня, либо в другое удобное для аналитика представление (например, псевдокод).

Следует отметить, что трасса позволяет частично восстановить дамп памяти, отразив в нем фактически выполнявшиеся инструкции. Этот дамп, дополненный разметкой и восстановленными символами, может быть передан подсистеме статического анализа. Такой подход обычно используется в случае, когда код в исполняемом файле был запакован (зашифрован) и, следовательно, недоступен для непосредственного извлечения. В реализованной системе в качестве компонента анализа трасс используется специально разработанная среда TrEx, а в качестве компонента статического анализа – среда IDA Pro. Их взаимодействие подробно описано в разделе 3.

Следует отметить, что описанные выше действия, выполняющиеся в процессе анализа, не зависят от типа процессорной архитектуры, к которой относится бинарный код. Это позволило реализовать единые алгоритмы анализа, избежав их дублирования для каждой архитектуры, тем самым значительно ускорив разработку, снизив трудозатраты и облегчив поддержку разработанного ПО.

3. Интеграция компонент статического и динамического анализа

Потребность в совместном использовании статического и динамического подхода обуславливается рядом причин. Основой для большинства исследований, проводимых в рамках анализа кода, таких как поиск дефектов и НДВ, является статическое представление программы, так как дефекты и НДВ обычно бывают привязаны к редко употребляемым наборам входных данных, что существенно снижает вероятность их обнаружения в ходе динамического анализа. Ошибки, соответствующие часто реализуемым путям обычно устраняются на этапе тестирования. Таким образом, основой для проведения исследований является построение полного и возможно более точного статического представления программы в виде графа вызовов и набора графов потока управления для каждой функции.

При разработке системы комбинированного анализа, в качестве компонента динамического анализа использовалась разработанная среда анализа бинарных трасс TrEx, а в качестве статического компонента – среда IDA Pro.

3.1. Цели и направления интеграции.

В рамках интеграции решались следующие основные задачи:

- дополнить статическое представление программы, информацией о динамически вычисляемых адресах, чтобы увеличить автоматически-дизассемблируемую часть программы, а также уточнить графы вызовов и потока управления, получаемые в рамках статического анализа;
- обеспечить генерацию статического представления программы по трассе выполнения, что позволит проводить анализ программ, код которых зашифрован и не может быть непосредственно извлечён из исполняемого файла;
- обеспечить возможность синхронизации между статическим и динамическим представлением программы для повышения эффективности анализа.

Для того чтобы воспользоваться возможностями среды IDA Pro требуется загрузить в неё всё статическое представление или его часть. Статическое представление программы включает в себя дамп кода и данных программы и разметку этого дампа, позволяющую, во-первых, отделить код программы от данных, а во-вторых, правильно дизассемблировать код. Дамп и разметка могут быть получены несколькими способами, применяемыми в зависимости от особенностей анализируемой программы.

3.2. Способы получения дампа.

(1) Дамп получается в результате разбора исполняемого файла или, в случае использования упаковщика, как снимок адресного пространства программы перед началом её трассировки. В обоих случаях в дамп попадёт весь код программы и её данные. Полнота информации составляет основное преимущество данного подхода. Среди ограничений возможности применения подхода отметим:

a) В ряде ассемблеров определены инструкции, позволяющие выполнять переходы по вычисляемым адресам (например, инструкция CALL mem32 для IA32). Статический анализ в этом случае не позволяет определять значения исполнительных адресов, что затрудняет автоматическое восстановление графа потока управления программы.

b) В дамп попадёт код только тех динамически загружаемых библиотек, загрузка которых будет выполнена до момента снятия дампа.

c) В случае, когда загружаемый код зашифрован (запакован), а также в случае самомодифицирующегося кода дамп не будет соответствовать исполнявшейся программе.

(2) Генерация дампа по трассе (точнее по графу потока управления, построенному по трассе), на основании анализа исполнявшегося кода. Такой механизм реализован в среде TrEx и будет подробно описан ниже. Его преимуществом является точность: дамп точно соответствует коду, который

фактически исполнялся. Это решает проблемы динамически загружаемых библиотек и защищенного кода. Однако в случае самомодифицирующегося кода для генерации однозначного дампа необходима модификация исходного кода. Ограничения применимости данного подхода:

a) Неполнота дампа – в дамп попадает только код, исполнявшийся на текущих входных данных.
b) Восстановление части дампа, содержащей значения данных в момент начала снятия трассы требует больших затрат по времени. В текущей версии среды TrEx эта часть дампа не восстанавливается.

3.3. Способы разметки дампа.

Если дамп был сгенерирован по трассе, то в качестве разметки ему можно передать адреса всех функций встречающихся в трассе – при этом IDA Pro может автоматически дизассемблировать код этих функций. В случае получения дампа из исполняемого файла или снятия снимка памяти часть кода может быть дизассемблирована в IDA автоматически, начиная от точки входа в программу и для дополнения достаточно передать адреса динамически вычисляемых переходов и вызовов. Однако в случае применения приёмов, препятствующих отладке, дизассемблирование будет неверным. Примером приёма, препятствующего отладке, может служить применение «мусорных байтов»: инструкция безусловного перехода заменяется инструкцией условного перехода с непрозрачным предикатом, который всегда выполняется; это невозможно выяснить при статическом анализе; по адресу непосредственно следующему за условным переходом размещается «мусорный байт», приводящий к ошибочному дизассемблированию последующего кода. Для преодоления таких приёмов можно передать в IDA Pro адреса начал всех исполняемых инструкций. Если при этом передавать также и коды операций, отмечая их несовпадения с данными дампа, загруженными в IDA Pro, можно автоматически выделить участки самомодифицирующегося кода.

3.4. Символьная информация.

В IDA Pro содержится достаточно богатая база данных символов, сопоставленных с адресами, для различных стандартных библиотек, кроме того есть встроенный механизм FLIRT, позволяющий «узнавать» функции на основе анализа их кода. Тем не менее, библиотеки постоянно обновляются, информация устаревает. В связи с этим реализована возможность экспорта дополнительной символьной информации из трассы в IDA Pro. Эта информация позволяет сопоставить функциям, обнаруженным в трассе их символьные имена, что значительно облегчает анализ.

3.5. Совмещение динамического и статического анализа.

При анализе программы бывает полезно одновременно наблюдать её статическое представление и её поведение на некоторых входных данных. Для реализации этой возможности в TrEx был встроен компонент синхронизации с IDA Pro: при каждом изменении шага в трассе, загруженной в TrEx, отображение в IDA Pro автоматически позиционируется на статическое представление этого шага. Таким образом, TrEx выступает в роли интерактивного отладчика, сопровождающего просмотр статического кода. Значения регистров, доступные в TrEx, а также адреса, с которых была выполнена загрузка выполнявшихся инструкций, позволяют вычислять динамически задаваемые адреса. Кроме того, совмещение статического и динамического представления удобно в случае анализа условных переходов: можно просматривать как ветвь кода, попавшую в трассу, так и альтернативную ветвь.

3.6. Реализация системы TrEx-IDA Pro.

Для обеспечения взаимодействия среды TrEx с дизассемблером IDA PRO были реализованы: модуль-расширение IDALuaPlugin, обеспечивающий доступ к интерфейсу SDK IDA Pro из Lua-скриптов; загрузчик VadsIDALoader, позволяющий загружать в IDA дамп и карту памяти в формате TrEx; клиентский модуль системы TrEx, обеспечивающий доступ к IDA Pro; генератор файла дампа и карты памяти на основе трассы.

Модуль-расширение IDALuaPlugin приспособлен для работы с версией SDK IDA PRO Advanced 5.2. Он представляет собой сервер, обеспечивающий доступ клиента к Lua-обёртке над SDK IDA Pro. В процессе запуска сервер ищет в командной строке IDA Pro адрес и порт для запуска. Для их указания нужно запустить IDA Pro с параметром: «-OIdaLua:127.0.0.1:44», подставив требуемый адрес и порт. Если параметры запуска не найдены – сервер запрашивает адрес и порт через диалог в процессе запуска. Обёртка представляет из себя (в терминах языка Lua) глобальную таблицу IdaInt функции которой соответствуют функциям SDK. Дополнительная локальная функциональность модуля-расширения – возможность вызова Lua-скриптов из файла в рамках среды IDA Pro. Для этого в меню Edit/Plugins/Lua plugin нужно указать имя Lua-скрипта. Следует отметить, что запуск модуля-расширения происходит не в момент запуска IDA Pro, а сразу после открытия выбранного файла для дизассемблирования. Старт модуля-расширения сопровождается выводом в консоль IDA Pro сообщения «IdaLua: server started <IP> : <Port>».

Загрузчик VadsIDALoader приспособлен для работы с версией SDK IDA PRO Advanced 5.2. Результат сборки проекта – динамическая библиотека VadsIDALoader.ldw. Чтобы подключить загрузчик к IDA Pro нужно положить эту библиотеку в директорию <IDA PATH>/loaders. Для использования загрузчика нужно в диалоге выбора файла для дизассемблирования указать

файл дампа, предварительно сгенерированного в среде TrEx (например memory.bin). После этого пользователю будет выдан диалог для выбора загрузчика из списка подходящих – нужно выбрать Vads dump files (вехний в списке). При этом для каждого диапазона памяти в карте памяти будут созданы соответствующие сегменты и в них загружена информация из дампа.

Внутри системы TrEx настройка взаимодействия с IDA Pro осуществляется через меню IDA Pro. Следует отметить, что только в окне базовой трассы в меню представлены все пункты – в окнах производных трасс присутствует только пункт Attach, назначение которого описано ниже.

4. Среда динамического анализа TrEx

Подробная организация среды TrEx представлена на рис. 1.

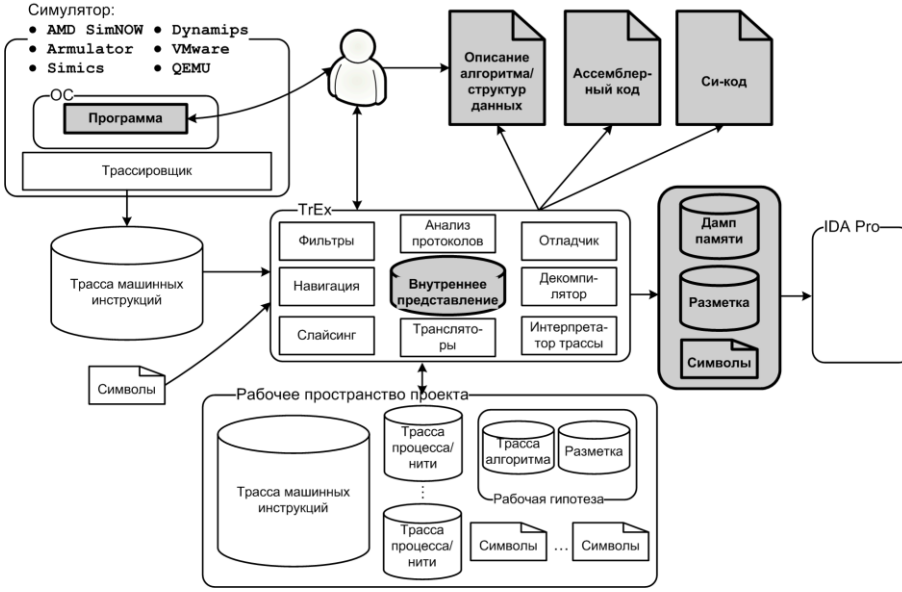


Рис. 1. Компоненты среды TrEx.

Среда предлагает следующие возможности:

- поддержка средств трассировки, обеспечивающих получение трасс для различных архитектур и способов защиты;
- возможность комбинированного анализа совместно со средой IDA Pro;
- набор генераторов различных представлений анализируемой программы;

- подсистема получения и обработки символьной информации;
- подсистема описания моделей функции; позволяет автоматически получать разметку трассы в виде комментариев к вызовам функций, содержащих значения фактических параметров вызова.

Большая часть компонент среды реализована в виде модулей расширения базовой функциональности, включающей средства работы с внутренним представлением и символьной информацией. Для обеспечения возможности встраивания подключаемых модулей TrEx в графическую среду разработан набор интерфейсных вызовов, с помощью которых модули могут указывать, каким образом следует дополнить графический интерфейс пользователя.

Центральным объектом взаимодействия между средой и модулями является интерфейс TrEx, обеспечивающий доступ к различным аспектам управления графической оболочкой. Объект этого интерфейса передаётся модулю при его инициализации. Объект TrEx позволяет получить доступ к информации об открытом проекте (трассе и подтрассах, а также вспомогательных данных) в виде объекта Workspace. Модули могут добавлять записи в журнал системы через объект Logger, который может быть получен из объектов TrEx или Workspace. Для каждой записи сохраняется дата и время её добавления, а также характер: информационная запись, предупреждение или ошибка.

Главное окно и окна трасс являются инструментируемыми, поддерживая следующие возможности: добавление (удаление) пункта меню; добавление (удаление) кнопки на панель инструментов; добавление (удаление) док-панели или панели в строке статуса. Средой производится минимальная проверка корректности работы добавлений. В частности, требуется, чтобы все добавленные в окна инструменты были удалены на момент завершения работы.

Менеджер окон трасс обеспечивает: получение упорядоченного списка всех открытых окон трасс, либо открытых окон конкретной трассы; создание нового или активацию уже открытого окна трассы; добавление плавающего окна. Кроме того, менеджер окон трасс описывает набор сигналов («окно создано», «закрытие окна», «изменение состояния окна»), на которые могут подписываться модули, чтобы иметь к ним доступ.

Для обеспечения гибкой настройки распространения изменений выбранного шага в трассе между открытыми окнами реализован следующий подход. Вся конфигурация открытых окон воспринимается в виде ориентированного графа, вершины которого соответствуют окнам, а дуги указывают на распространения изменений по выбранному шагу. Обеспечено визуальное управление описанным графом.

Все компоненты среды TrEx реализованы как задачи: по определению задача - это процедура, выполняющаяся в отдельном потоке, возвращающая по запросу другой задачи текстовое описание текущего состояния, либо значение

прогресса выполнения в виде числа от 0 до 1000, а также предоставляющая возможность прерывания пользователем с корректным освобождением ресурсов. Реализация компонентов TrEx как задач позволяет, использовать единое графическое отображение прогресса выполнения, а также вызывать одни задачи из других в качестве подзадач. Каждый вызов задачи фиксируется в журнале: сохраняется время начала и конца обработки, идентификатор и параметры задачи.

В среде реализован механизм визуальных свёрток, позволяющий при отображении пропускать части трассы, сворачивая их в древовидную структуру. Стандартный вид трассы отображает свёртки в левой части окна в виде полосок (в развёрнутом виде) или квадратов (в свёрнутом виде) различного цвета.

Одним из важных источников информации о функционировании программы является набор символов (имён) функций, которые вызываются программой. В связи с этим в среде TrEx реализована специальная подсистема получения такой информации из динамически загружаемых библиотек. Одним из важных компонентов подсистемы является разборщик библиотек наиболее распространённых форматов исполняемых файлов PE32/PE32+,COFF, ELF. Он позволяет извлекать имена экспортируемых этими библиотеками функций.

При выделении и анализе алгоритма важную роль играет знание функций и их параметров, реализующих отдельные части алгоритма. Если описание параметров некоторой функции известно, например, получено в ходе анализа или, если функция является частью известной библиотеки, то оно может быть использовано для поиска всех вызовов данной функции и получения значений фактических параметров этих вызовов. Эта информация, может быть чрезвычайно полезна для понимания работы алгоритма. В системе TrEx реализована подсистема описания моделей функций.

При создании модели функции сначала выбирается процесс, в котором должна выполняться функция, затем указываются: имя функции, начальный адрес ее кода, набор конечных адресов, связанных с функцией (может быть получен автоматически по начальному адресу), а также описания ее входных и выходных параметров. Виды параметров: параметр, передаваемый через стек с заданием смещения относительно указателя стека и размера ячейки (две константы, либо арифметических выражения); параметр-регистр (имя регистра).

5. Примеры применения среды TrEx.

Разработанный метод комбинированного анализа позволяет в полуавтоматическом режиме выделять алгоритмы, реализованные исследуемым кодом, отслеживать работу алгоритмов, распределенных по адресным пространствам нескольких процессов или использующих

особенности работы кода операционной системы. Рассмотрим несколько примеров применения комбинированного анализа.

5.1. Выделение исследуемого кода

Первый пример состоит из двух модельных программ, реализующих одну и ту же функцию вычисления суммы натурального числового ряда из N чисел.

```
Function test_func(ByVal n As Integer) As Integer
    If n = 0      Then test_func = 0
Else test_func = n + test_func(n - 1)
End test_func
```

Первая программа – `vb6test` – реализована на *VisualBasic* 6.0 и скомпилирована в виде p-кода (кода виртуальной машины). Вторая программа – vbnettest – реализована на *VisualBasic.NET* и скомпилирована в виде управляемого кода.

Трасса снималась для n=10. Размер трассы, полученной для каждой программы, составляет около 700Мб. Результаты обработки трассы представлены в таблице 1. После того как в трассе была обнаружена печать результатов функции `test_func`, был выполнен обратный слайсинг, в качестве критерия которого выступало результирующее значение. Время выполнения слайсинга около 50 секунд.

Из таблицы видно, что в случае ручного анализа рассматриваемых программ аналитик имел бы дело либо с миллионами шагов при динамическом анализе, либо с десятками тысяч шагов при статическом, использование слайсинга за несколько минут сводит задачу анализа к нескольким сотням инструкций, т.е. принципиально упрощает задачу аналитика.

	Трасса, содержащая код работы анализируемой программы в пользовательском режиме			Число инструкций после обработки
	Размер, МБ	Число шагов	Число инструкций в листинге	
VB v6.0	42	575 392	26 620	356
VB .NET	35	484 248	62 726	143

Таб. 1. Применение инструментов системы TrEx для сокращения размеров анализируемого кода.

5.2. Восстановление алгоритма генерации лицензионного ключа

В рамках данной задачи требовалось восстановить алгоритм генерации ключа системы SPLM (SmartPlant License Manager 2010). Трассировалась работа файла licgen.exe, который получал на вход файл machine-id.txt, содержащий идентификатор машины, а на выходе генерировал файл, содержащий лицензию в бинарном виде. Файл licgen.exe был получен в результате компиляции в отладочном режиме с прикреплённой отладочной информацией в формате pdb. В частности это привело к тому, что для всех вызовов были сгенерированы заглушки, т.е. первой инструкцией всех функций программы была инструкция JMP.

В полученной трассе были восстановлены модули и подключена символьная информация, содержащая изначально только символы системных библиотек, но не символы самой программы, т.к. pdb является закрытым форматом и напрямую получить из него символы затруднительно. В трассе был найден вызов функции WriteFile, соответствующий выводу лицензии в файл. Путём восстановления значений его параметров были восстановлены данные лицензии (предварительно были восстановлены параметры буфера, который содержал саму лицензию). Далее была получена символьная информация из файла pdb, путём загрузки исходного файла licgen.exe в дизассемблер IDA Pro, который автоматически способен извлекать символьную информацию из pdb файлов. Для генерации символьной информации в формате TextSym был применён разработан IDC-скрипт make_sym. По полученным символам была выяснена точка в трассе, где начал работать код licgen – вызов функции main. Для извлечения алгоритма генерации ключа был применён обратный слайсинг с учётом адресных зависимостей для буфера лицензии, от точки вызова функции WriteFile до точки вызова функции main. В процессе было выяснено, что запись в файл проводилась после завершения функции main, во время сброса буферов на диск, т.е. файл не был закрыт пользователем, а закрывался неявно при завершении процесса. Полученный слайс был далее отфильтрован по процессу, с удалением кода ядра. На результирующий слайс было наложено дерево вызовов функций (CallTree), что позволило эффективно представить структуру выделенного алгоритма.

5.3. Исследование зловредного кода

В рамках этой задачи требовалось исследовать поведение зловреда Trojan.Win32.Tdss.ajfl и выделить в виде ассемблерного листинга код этого зловреда, после его разворачивания в памяти зараженной машины. Виртуальная машина AMD SimNow была заражена указанным зловредом, путём запуска его исполняемого файла Trojan.Win32.Tdss.ajfl.exe. Данный зловред перехватывает любое обращение к диску на уровне файловой системы. Чтобы убедится, что система заражена был использован инспектор ядра GMER. После этого была включена трассировка и задействована команда

оболочки cmd.exe type для вывода в консоль содержимого файла Windows_default.pif. Выполнение этой команды гарантированно должно было привести к передаче управления зловреду.

В первую очередь в полученной трассе была найдена функция CreateFileW, для которой было восстановлено значение первого параметра, содержащего имя открываемого файла (_default.pif), с целью убедится, что это нужный вызов. После чего был найден вход в ядро, реализованный инструкцией SYSENTER. Затем был найден переход в модуль файловой системы ntfs.sys. Этот переход соответствовал вызову функции IofCallDriver. Первая инструкция этой функции была перезаписана инструкцией JMP, передающей управление зловредному коду, который затем осуществляет вызов одной из своих функций, используя таблицу переходов. В результате был получен один из диапазонов адресов, по которым был загружен зловредный код. По данному диапазону была проведена фильтрация трассы и построен ассемблерный листинг, который был сохранён в файл в текстовом виде.

Список литературы

[1] Tool Interface Standards (TIS). Portable Executable Formats (PE), http://www.x86.org/intel.doc/tools.htm.

[2] Tool Interface Standards (TIS). Executable and Linkable Format (ELF), http://www.x86.org/intel.doc/tools.htm

[3] IDA Pro Disassembler, http://www.hex-rays.com/idapro/.

[4] Fast Library Identification and Recognition Technology (FLIRT), http://www.idapro.ru/description/flirt/.

[5] H. Yin, Z. Liang, D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. // Proceeding of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), Feb. 2008

[6] В. А. Падарян, М. А. Соловьев, А. И. Кононов. Моделирование операционной семантики машинных инструкций. // Программирование №3, 2011. стр. 50–64

[7] LLVM Language Reference Manual. // Llvm.org/docs/LangRef.html

[8] AMD SimNow Simulator, http://developer.amd.com/cpu/simnow/Pages/default.aspx.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. // IEEE Computer, 35(2):50–58, Feb. 2002.

[10] S. Debray, J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. // Proceedings of the 17th. IEEE Working Conference on Reverse Engineering, Oct. 2010, pp 131–140.

[11] Wang C., Hill J., Knight J,. Davidson J. Software tamper resistance: obstructing static analysis of programs // Tech. Rep., N 12, Dep. of Comp. Sci., Univ. of Virginia, 2000.

[12] Weiser M. Program slicing // Proceedings of the 5th International Conference on Software Engineering. – IEEE Computer Society Press, 1981.– pp. 439–449.

[13] Korel B., Laski J. Dynamic program slicing // Information Processing Letters, Vol. 29, Issue 3.– 1988.– p. 155–163

[14] А. Тихонов, А. Аветисян, В. Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. 2008, №3. с. 66-71

[15] В. А. Падарян, А. И. Гетьман, М. А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды Института системного программирования, том 16, 2009, с. 51-72. –.
[16] M. Venable, M.R. Chouchane, M.E. Karim, and A. Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In DIMVA'05, pages 1-18, 2005.

Combined (static and dynamic) analysis of binary code

A.YU. Tikhonov, A.I. Avetisyan
{fireboo@ispras.ru}, {arut@ispras.ru}
ISP RAS, Moscow, Russia

Abstract. This paper investigates the process of binary code analysis. To achieve typical goals (such as extracting algorithm and data formats, exploiting vulnerabilities, revealing backdoors and undocumented features) a security analyst needs to explore control and data flow, reconstruct functions and variables, identify input and output data. Traditionally for this purposes disassemblers and other static data flow analysis tools have been used. However, since developers have been taking steps to protect their programs from analysis (for example, code being unpacked or decrypted at runtime), static analysis may not yield results.

In such cases we propose to use dynamic analysis (analysis of execution traces of the program) to complement static. The problems that arise in the analysis of binary programs are discussed, and corresponding ways to automate solving them are suggested. The core of proposed method consists of whole system tracing and consecutive representation lifting: OS-aware events, process/thread identification, fully automated control and data flow reconstruction. The only manual step is searching for anchor instructions in the trace, e.g. I/O operations, which are used as input criteria for another automated step: precise algorithm extraction by trace slicing. The final step of the method constructs static test case code suitable for further analysis in tools such as IDA Pro.

We implemented the proposed approach in an environment for dynamic analysis of binary code and evaluated it against a model example and two real-world examples: a program license manager and a malware program. Our results show that approach successfully explores algorithms and extracts them from whole system traces. The required efforts and amount of time are significantly reduced as compared with traditional disassembler and interactive debugger.

Keywords: algorithm recovery, binary code, backdoors, instruction level tracing

References

[1]. Tool Interface Standards (TIS). Portable Executable Formats (PE), http://www.x86.org/intel.doc/tools.htm.
[2]. Tool Interface Standards (TIS). Executable and Linkable Format (ELF), http://www.x86.org/intel.doc/tools.htm
[3]. IDA Pro Disassembler, http://www.hex-rays.com/idapro/.
[4]. Fast Library Identification and Recognition Technology (FLIRT), http://www.idapro.ru/description/flirt/.
[5]. H. Yin, Z. Liang, D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. Proceeding of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), Feb. 2008

[6]. V.A. Padaryan, M.A. Solov'ev, A.I. Kononov. Simulation of operational semantics of machine instructions. Programming and Computer Software, May 2011, Volume 37, Issue 3, pp 161-170, DOI 10.1134/S0361768811030030
[7]. LLVM Language Reference Manual http://llvm.org/docs/LangRef.html
[8]. AMD SimNow Simulator http://developer.amd.com/cpu/simnow/Pages/default.aspx
[9]. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. IEEE Computer, 35(2):50–58, Feb. 2002. DOI 10.1109/2.982916
[10]. S. Debray, J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. Proceedings of the 17th. IEEE Working Conference on Reverse Engineering, Oct. 2010, pp 131–140.
[11]. Wang C., Hill J., Knight J,. Davidson J. Software tamper resistance: obstructing static analysis of programs. Tech. Rep., N 12, Dep. of Comp. Sci., Univ. of Virginia, 2000.
[12]. Weiser M. Program slicing. Proceedings of the 5th International Conference on Software Engineering. IEEE Computer Society Press, 1981. pp. 439–449.
[13]. Korel B., Laski J. Dynamic program slicing. Information Processing Letters, Vol. 29, Issue 3. 1988. p. 155–163.
[14]. Tikhonov A.YU., Avetisyan A.I., Padaryan V.A., Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of exploring of an algorithm from binary code by dynamic analysis]. Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy. 2008, №3. pp. 66-71 (in Russian)
[15]. Padaryan V.A., Getman A.I., Solov'ev M.A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 16, 2009, pp. 51-72 (in Russian).
[16]. M. Venable, M.R. Chouchane, M.E. Karim, and A. Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In DIMVA'05, pages 1-18, 2005. DOI 10.1007/11506881_1

Подход к восстановлению потока управления запутанной программы

И.Н.Ледовских, М.Г.Бакулин
{il, bakulinm}@ispras.ru

Аннотация. Запутывание потока управления является одним из наиболее распространенных способов защиты бинарного кода приложений от анализа. Запутывающие преобразования резко увеличивают трудоёмкость выделения и распознавания алгоритмов и структур данных. В статье описывается подход к восстановлению потока управления программы, запутанного комбинацией различных преобразований – от вставки недостижимого кода с помощью непрозрачных предикатов до виртуализации. Приводятся результаты тестирования прототипной реализации данного метода на модельном примере, запутанном различными инструментальными средствами защиты.

Ключевые слова: запутывание потока управления, восстановление потока управления программы

1. Введение

Восстановление алгоритмов при анализе приложений в бинарных кодах осуществляется посредством достаточно большого числа различных методов анализа. Часть этих методов применяется в виде последовательно выполняемых этапов, выполняющих выделение и различные преобразования интересующих аналитика фрагментов приложения. В частности, это относится к методам преодоления механизмов защиты кода от анализа.

В настоящее время применяется множество способов защиты бинарного кода программ как от статических, так и от динамических методов анализа. Примерами таких защитных механизмов могут служить полиморфизм кода, превентивные преобразования, нарушающие работоспособность средств анализа, или средства обнаружения отладчиков и виртуальных сред. Однако наиболее распространенным механизмом защиты кода от анализа остаются запутывающие преобразования. Известные разновидности запутывающих преобразований, а также оценки их эффективности и устойчивости, приводятся в работе [1]. Применительно к анализу программ в бинарных кодах, из всех перечисленных в данной классификации видов запутывания, наиболее актуальными являются преобразования, запутывающие поток управления программы. В этой категории можно выделить преобразования переупорядочивания (переупорядочивание выражений, операторов и циклов), преобразования

агрегации (например, вставки либо выделение функций, клонирование кода, раскрутка циклов) и преобразования вычислений (например, преобразование графа потока управления к несводимому виду, расширение условий циклов, виртуализация кода). В данной работе рассматривается преодоление некоторых запутывающих преобразований, относящихся к группам агрегации и преобразования вычислений, в системе TrEx.

Значительное число преобразований, запутывающих поток управления, основано на использовании непрозрачных переменных и непрозрачных предикатов. Эти понятия вводятся в работах [1] и [2]. Достаточно будет сказать, что это переменные и предикаты, свойства которых априори известны при выполнении запутывающего преобразования, но являются достаточно сложными для распознавания при распутывании. Таким свойством для непрозрачной переменной может являться, например, диапазон ее значений, а для предиката - значение (истина или ложь). Внесение в код непрозрачного предиката $\boldsymbol{P^F}$ (со значением «ложь») или $\boldsymbol{P^T}$ (со значением «истина») приводит к тому, что ветки, соответствующие условию «истина» в первом случае и условию «ложь» во втором, никогда не выполнятся, и соответствующий код становится *недостижимым*. Непрозрачные предикаты используются для запутывания потока управления как сами по себе, так и в комбинации с другими методами запутывания. В частности, *расширение условий циклов* делает более сложным условие завершения цикла за счёт добавления непрозрачного предиката, не влияющего на число итераций цикла. С помощью предикатов вида $\boldsymbol{P^?}$ реализуется клонирование кода, то есть копирование некоторого фрагмента кода, который выполняется наравне с исходным фрагментом в порядке, зависящем от значения непрозрачного предиката. При этом оба фрагмента-клона, как правило, дополнительно запутываются различными способами (например, переупорядочиванием инструкций и вставкой мёртвого или избыточного кода) для затруднения распознавания клонирования.

Широкое применение непрозрачных предикатов обусловлено в значительной степени невысокими накладными расходами в плане влияния данного запутывающего преобразования на быстродействие, и достаточно большим временем их взлома известными методами [4].

Другое рассматриваемое в настоящей статье запутывающее преобразование – значительно более затратное, но при этом более устойчивое к распутыванию – виртуализация кода [5,6]. Защищаемый участок кода приложения транслируется в байт-код некоторой виртуальной машины (ВМ), после чего выполняется замена этого фрагмента приложения кодом соответствующего интерпретатора (рис.1). Исходный фрагмент удаляется безвозвратно, и для восстановления его семантики необходимо провести достаточно сложный анализ использованной виртуальной машины, а затем выполнить обратную трансляцию ее байт-кода в систему команд исходной целевой архитектуры. Задача осложняется тем, что как код самой виртуальной машины, так и ее данные подвергаются дополнительному запутыванию различного уровня

сложности – от вставки непрозрачных предикатов, недостижимого и мёртвого кода до мутации кодов операций виртуальной машины и ее ре-виртуализации.

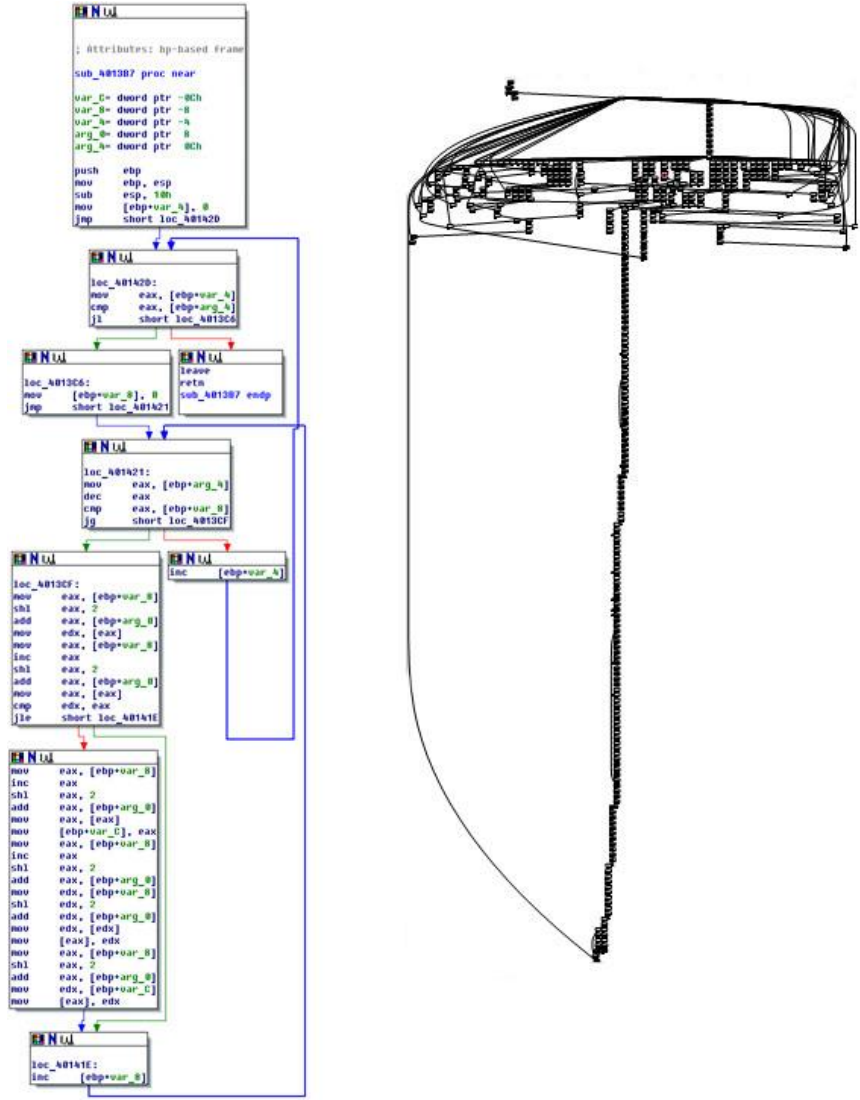
В настоящей статье мы рассмотрим подход к восстановлению потока управления программы, запутанной посредством комбинации преобразований различной сложности. Работа организована следующим образом. Во втором разделе рассматриваются примеры распространенных в настоящее время инструментальных средств защиты программ, использующих виртуализацию совместно с другими запутывающими преобразованиями. В третьем разделе дается краткий обзор известных методов распутывания потока управления, в частности, методов преодоления виртуализации. Далее, в разделах 4 и 5 мы рассматриваем подход, реализуемый системой TrEx, и полученные при апробировании этого подхода результаты. В заключительном шестом разделе полученные результаты оцениваются с точки зрения их практической применимости, а также рассматриваются направления дальнейших исследований.

2. Инструменты запутывания программ

В ходе выполнения данной работы мы провели сравнительный анализ и тестирование шести инструментов запутывания программ, использующих технологию виртуализации. Это три российские разработки – VMProtect [12], Enigma Protector [14], Private exe Protector [16], и три зарубежных – CodeVirtualizer [13], Safengine [15] и Obsidium [17]. Две системы были исключены из дальнейшего анализа – Obsidium, демонстрационная версия которого не поддерживает технологии виртуализации кода, и Private exe Protector, реализующий защиту упаковкой кода; виртуализацией в этой системе защищен только распаковщик, а код защищаемого приложения после распаковки доступен в памяти и может быть легко восстановлен динамическими методами.

Оставшиеся четыре обфускатора реализуют виртуальную машину с интерпретатором кода типа Decode-Dispatch, поток управления которой имеет характерный вид (рис.1Б) с главным циклом выборки и декодирования инструкций байт-кода и широким ветвлением, реализующим обработку декодированных инструкций. Различия рассматриваемых систем защиты заключаются в основном в модели виртуальной машины (например, стековая RISC-машина в [12] и регистровые RISC и CISC-машины в [13]), в конкретных реализациях байт-кода и в средствах дополнительной защиты. Так, CodeVirtualizer и Safengine предоставляют возможность дополнительного запутывания кода с помощью непрозрачных предикатов (Multi-Branch Technology); первая из этих систем реализует также запутывание данных (обфускация кодов операций и константных операндов), а также ряд дополнительных методов запутывания, недоступных в демо-версии (мутация кодов операций, ре-виртуализация, эмуляция стека). Различаются у рассматриваемых виртуальных машин также реализации обработчиков

инструкций – это отдельные функции для обработки различных инструкций в [15] и блоки обработки в операторе ветвления внутри одной функции в [12,13,14].



А Б

Рис.1.Граф потока управления исходной программы (А) и после запутывания посредством виртуализации (Б)

Все рассматриваемые виртуализаторы работают с программами для архитектуры Intel 64 и способны защищать приложения как для среды Win32, так и для 64-разрядного окружения.

Выше приводится пример запутывания виртуализацией – граф потока управления простой тестовой программы (рис. 1А) и ее защищенной версии (рис.1Б), которая получена с помощью системы CodeVirtualizer с дополнительным запутыванием потока управления непрозрачными предикатами.

3. Подходы к распутыванию потока управления

В настоящее время основой большинства известных подходов к распутывания программ в бинарных кодах являются различные комбинации динамических и статических методов. Это применимо как к преодолению достаточно простых видов запутывания потока управления (как описано в работах [3,4]), так и к распутыванию кода после виртуализации.

Статический анализ и восстановление кода, защищенного с помощью виртуальных машин, представляет собой сложный многоэтапный процесс [5]. В процессе анализа необходим полный анализ виртуальной машины «вручную». Опытный инженер-аналитик должен восстановить систему команд машины, разработать внутреннее представление и реализовать транслятор, отображающий байт-код виртуальной машины в это внутреннее представление. Хотя для каждой отдельной реализации ВМ эта задача имеет вполне приемлемую трудоемкость, ограничения при таком подходе велики. Необходимо учитывать шаблонную структуру инструкций байт-кода и изменчивость кодов операций в рамках шаблонов; далее, появление модифицированных версий существующих ВМ может потребовать повторного анализа.

Вторым этапом распознается точка входа в виртуальную машину; эта задача, сложная для статического анализа, не представляет собой проблемы при динамическом подходе. Затем распознается структура ВМ, и локализуется ее байт-код. Далее, для конкретной реализации байт-кода и конкретной ВМ должен автоматически генерироваться дизассемблер, распознающий шаблоны операций; задача осложняется применением к защищенному виртуализацией коду дополнительных запутывающих преобразований, препятствующих распознаванию и автоматическому анализу блоков обработки байт-кода. Кроме того, рассматриваемые обфускаторы применяют различные виды запутывания данных, затрудняющие распознавание шаблонов инструкций байт-кода – в первую очередь, это запутывание константных операндов инструкций с помощью арифметико-логических преобразований, различающихся для различных экземпляров ВМ.

При успешном построении дизассемблера байт-кода конкретной ВМ, выполняется генерация внутреннего представления для последовательности

инструкций этой машины. Затем к полученному внутреннему представлению применяется ряд компиляторных оптимизаций. В первую очередь, делается замена обращений к стеку простыми присваиваниями, затем – распространение констант и копий. Такие оптимизации способны уменьшить объем внутреннего представления до 20% от изначально сгенерированного [5]. Далее, для виртуальных RISC-машин выполняется обратная замена последовательностей арифметических и логических инструкций – во внутреннем представлении восстанавливаются операции, отсутствующие в системе команд ВМ и реализованные через другие примитивы (например, VMProtect реализует логическую операцию AND посредством трех NOR, а арифметическую SUB – через операции NEG и ADD). Наконец, по оптимизированному внутреннему представлению генерируется код целевой архитектуры – в рассматриваемых случаях, Intel 64.

Как можно видеть, такой статический подход сложен, трудоемок и не обладает свойством общности, хотя и способен в случае успеха максимально точно восстановить исходный код запутанной программы. Динамический подход на основе трассировки защищенного приложения является более универсальным и гибким.

Например, в работе [7] описывается подход, основанный на анализе обращений к памяти при выполнении запутанного кода в защищенной среде программного эмулятора QEMU. Описываемый подход основан на распознавании в трассе программы переменных, выделении из их числа подмножества переменных, которые могли служить виртуальными счетчиками команд (VPC), и анализе кластеров памяти, адресуемых посредством этих переменных; в результате такого полностью автоматического анализа удается локализовать интерпретируемый байт-код. Далее, для анализа байт-кода используется знание структуры ВМ в самых общих чертах – достаточно распознавать главный цикл выборки и интерпретации, а также отдельные блоки интерпретации инструкций виртуальной машины (фазы эмуляции). Дальнейшее распознавание синтаксиса и семантики инструкций байт-кода основано на анализе способов доступа фаз эмуляции к кластеру памяти, хранящему интерпретируемую программу, и на восстановлении семантики отработавших в процессе трассировки блоков интерпретации байт-кода. Такой подход позволяет, в частности, относительно легко восстанавливать семантику инструкций передачи управления путем анализа изменений значения VPC. Восстановление вычислений остается достаточно сложной задачей.

В работе [8] предлагается следующий шаг по обобщению подхода к восстановлению виртуализованного кода – восстановление семантики исходной запутанной программы с помощью анализа зависимостей по данным и динамического слайсинга [11]. В трассе запутанной программы выделяется рад событий (например, системные вызовы или просто инструкции, оставленные виртуализатором без изменений); эти события анализируются и

используются как опорные точки для построения динамических обратных слайсов на основе только зависимостей по данным. Инструкции, попавшие в этот набор слайсов, затем объединяются в подтрассу, которая является аппроксимацией семантики запутанного посредством виртуализации кода. Соответствующий поток управления восстанавливается отдельно на основе анализа обращений к памяти при выборке инструкций байт-кода интерпретатором виртуальной машины.

4. Восстановление потока управления в системе TrEx

В системе TrEx в настоящее время исследуется подход, сочетающий достоинства перечисленных в разделе 3 работ. Вначале мы стараемся упростить трассу запутанного приложения посредством обнаружения и удаления непрозрачных предикатов, а также мертвого и избыточного кода. Затем происходит локализация интерпретатора виртуальной машины, его анализ и восстановление программы комбинацией описанных выше динамических методов. В частности, поток управления запутанной программы восстанавливается на основе алгоритма, описанного в работе [7].

4.1. Поиск и распознавание непрозрачных предикатов.

При динамическом анализе бинарного кода по трассе можно легко найти все предикаты, которые всегда передавали управление по одной и той же ветке, то есть собрать и проанализировать статистику переходов. Однако при таком чисто статистическом подходе будет наблюдаться большое число ошибочных распознаваний непрозрачных предикатов. Так, известно, что в среднем даже в незапутанной программе около 60% предикатов, наблюдаемых в трассе, всегда передают управление по одной и той же ветке[3]. Ситуация может быть улучшена посредством анализа большего количества трасс, однако полное покрытие кода всё равно не может быть достигнуто, а затраты времени возрастают. Поэтому в системе TrEx распознавание непрозрачных предикатов выполняется следующим образом.

1. По нескольким трассам одного приложения составляется список всех предикатов, которые передавали управление по одной ветке.
2. Этот список фильтруется, и в нём остаются только те предикаты, которые потенциально являются непрозрачными. При этом удаляются как полезные предикаты, которые могут передавать управление по другой ветке при других входных параметрах, так и предикаты, слишком сложные для автоматического анализа, хотя и являющиеся потенциально непрозрачными.

Для отсеивания ищется первая позиция исследуемого предиката в трассе, после чего выполняется динамический обратный слайсинг. Полученный слайс предиката проверяется на соответствие различным критериям (например, количество инструкций, содержащихся в слайсе; количество базовых блоков, в которые попадают инструкции слайса; модуль, к которому принадлежит

инструкция перехода, и т.д.). Следующий этап фильтрации отсеивает предикаты, которые на самом деле являются проверками значений параметров и результатов функций. Затем при наличии других трасс приложения проверяется, по одним и тем же путям выполняются переходы в предикатах из сокращенного списка, или нет. В случае, если в каком-либо месте трассы слайс предиката получается другим (то есть условие перехода по тому же адресу вычисляется иначе), дальнейшая обработка этого предиката прекращается, и он отвергается.

Далее происходит анализ предикатов на использование непрозрачных переменных. Строится список адресов используемых в предикатах переменных, и по имеющимся трассам находятся все обращения к этим адресам. В результате получаются списки:

- адресов, к которым обращались из каких-либо иных мест программы, кроме предикатов из сокращенного списка;
- адресов, в которые что-либо записывалось;
- адресов, в которых содержались исполненные инструкции.

Эти списки далее используются как для отсева сомнительных предикатов, так и для автоматического распознавания непрозрачных предикатов, основанных на использовании непрозрачных переменных, а также аналитиком для распознавания таких предикатов в автоматизированном режиме.

Наконец, к сокращенному списку потенциально непрозрачных предикатов применяется поиск по шаблонам – слайсы сравниваются с имеющимися шаблонами ранее распознанных непрозрачных предикатов. Не распознанные и не отвергнутые ранее потенциально непрозрачные предикаты поочередно выносятся на рассмотрение аналитику, который в итоге выносит вердикт о том, принимаются они, или отвергаются.

В результате такого отсеивания остаётся сокращённый список предикатов, на основе которого система упрощает трассу приложения, удаляя распознанные предикаты $\boldsymbol{P^F}$. Предикаты $\boldsymbol{P^T}$ в настоящее время из трассы не удаляются.

4.2. Поиск простого комбинированного запутывания потока управления

Часто применяется не один метод запутывания, а их комбинация. Например, непрозрачный предикат $\boldsymbol{P^F}$ может задавать переход на инструкцию внутри другого непрозрачного предиката, что усложняет его анализ; также непрозрачные предикаты маскируют участки клонированного кода, когда в разные ветки добавляются разные непрозрачные предикаты. Мёртвый или избыточный код также зачастую вставляется в программу вместе с непрозрачными предикатами. В системе TrEx факт присутствия мёртвого кода в ряде случаев выявляется одновременно с анализом предикатов. При фильтрации списка предикатов ищутся участки мёртвого кода, перемешанного со слайсом предиката. Примером такого перемешивания

может быть добавление операций с регистром, значение которого потом не используется или сразу затирается другим значением. В результате простой предикат может выглядеть, например, так:

```
MOV AX, 1
MOV BX, 10
SUB BX, 6
CMP AX, 0
JZ LABEL
MOV BX, …
{…}
LABEL: {…}
```

Дополнительный анализ основан на проверке наличия между шагами трассы, составляющими слайс предиката, не попавших в слайс инструкций. Если такая инструкция есть, то строится и анализируется последовательность зависящих от нее по данным шагов трассы – динамический прямой слайс.

Избыточный код, комбинированный с непрозрачными предикатами, обнаруживается с помощью шаблонов. Например, вместо этой последовательности команд:

```
PUSH EAX
MOV EAX, ECX
PUSH EDX
MOV EDX, EBX
NOT EAX
NOT EDX
OR EAX, EDX
NOT EAX
MOV ECX, EAX
POP EDX
POP EAX
```

можно использовать инструкцию AND ECX, EBX (если мы не учитываем, что в стеке остались значения регистров EAX и EDX). Поэтому на участках трассы, где локализованы непрозрачные предикаты, производится поиск фрагментов, совпадающих с известными шаблонами запутывания.

4.3. Восстановление потока управления виртуализованного кода

Большая часть эмуляторов виртуальных машин действует по следующей схеме:

1. Считывание кода операции из памяти.
2. Передача управления на соответствующую ветку.
3. Исполнение кода, эмулирующего данный код операции. Переход к 1).

У каждого такого эмулятора есть счетчик команд виртуальной машины (VPC), который определяет, из какой области памяти необходимо считывать следующий код операции.

Основываясь на этих общих свойствах, можно автоматизировать анализ того, какая переменная или регистр машины, на которой запускается приложение, используется как счетчик команд виртуальной машины, а также поиск участка памяти, содержащего интерпретируемый байт-код. Это делается в два этапа:

1. Кластеризация всех считываний по неконстантным адресам из памяти в регистр.
2. Поиск кластеров, считанные из которого данные влияют на передачу управления.

Кластеризация

Будем говорить, что на шаге трассы регистр привязан к участку памяти, если значение регистра на этом шаге является копией или модификацией значения, хранящегося в этом диапазоне памяти. Пусть в трассе имеется считывание из памяти, причем адрес вычисляется динамически на основании значения одного или нескольких регистров. Тогда такое чтение можно связать со всеми участками памяти, к которым привязаны эти регистры. После этого все чтения из памяти можно разбить на кластеры в зависимости от того, к каким участкам памяти привязаны эти считывания.

Необходим алгоритм, который бы позволил автоматически связать считывания из памяти с регистрами, а регистры с участками памяти. В статье [7] приводится такой алгоритм, модифицированная версия которого описывается ниже.

Обозначения:

`B(R,i)`	участки памяти, к которым привязан регистр `R` на шаге `i`;
`R<--Val`	в `R` заносится значение, зависящее от `Val`;
`R<<-Val`	в R заносится значение, зависящее от `Val` и значения регистра R;
`[M,S]`	участок памяти с адресом `M` и длиной `S`;
`C`	константа;
`OutDep(i)`	множество регистров, значение которых изменилось после выполнения инструкции на шаге `i`.
`InDep(i)`	множество регистров, значения которых повлияли на выходные параметры инструкции.

Связывание происходит в два этапа. На первом этапе мы привязываем считывания из памяти в регистр, и распространяем его вперёд. Алгоритм в виде псевдокода:

```
for i = start to end do
```

```
      foreach R in OutDep(i)
            if R <-- [M,S]
            then
                  B(R,i) = {[M,S]}
            end if
            if R <<- [M,S]
            then
                  B(R,i) = B(R,i-1) UNITE {[M,S]}
            end if
            if R1 <-- [f(R2,R3,...),S] OR R1 <<-
[f(R2,R3,...),S]
                  VarsF(i) = B(R2,i-1) UNITE B(R3,i-1)
UNITE ...
            end if
      end foreach
end for
```

На втором этапе мы привязываем регистр к участку памяти в момент записи его значения в памяти, и распространяем связывание назад. В виде псевдокода это выгляди так:

```
for i = end to start do
      foreach R in InDep(i)
            if [M,S] <-- R
            then
                  B(R,i) = {[M,S]}
            end if
            if R1 <-- R
            then
                  B(R,i) = B(R1,i+1)
            end if
            if R1 <-- [f(R2,R3,...),S] OR R1 <<-
[f(R2,R3,...),S]
                  VarsB(i) = B(R2,i-1) UNITE B(R3,i-1)
UNITE ...
            end if
      end foreach
end for
```

Следует заметить, что привязки B(R,i) обнуляются перед запуском каждого этапа.

После этого для каждого шага трассы вычисляется множество привязанных переменных Vars(i) = VarsF(i) UNITE VarsB(i). Далее происходит разбиение считываний на кластеры, в один кластер попадают все считывания из памяти, для которых пересечение их множества привязанных переменных не пусто.

Предположим, что нам известны все позиции операций считывания кода операции из памяти в трассе, а также известен регистр, использующийся как

счетчик команд виртуальной машины. Тогда можно восстановить граф потока управления исходной программы. Для этого для каждого значения счетчика команд запоминаются все значения, которые счетчик команд принимает при следующем считывании кода операции из памяти. Полученные данные можно представить в виде ориентированного графа, где каждая вершина обозначает возможное значение счетчика команд, а ребро a->b обозначает, что счетчик команд после значения a может принимать значение b. После этого полученный граф упрощается: все вершины, у которых ровно один предшественник и один последователь, и у их предшественника только один последователь, удаляются, и добавляется ребро из их предшественника в их последователя.

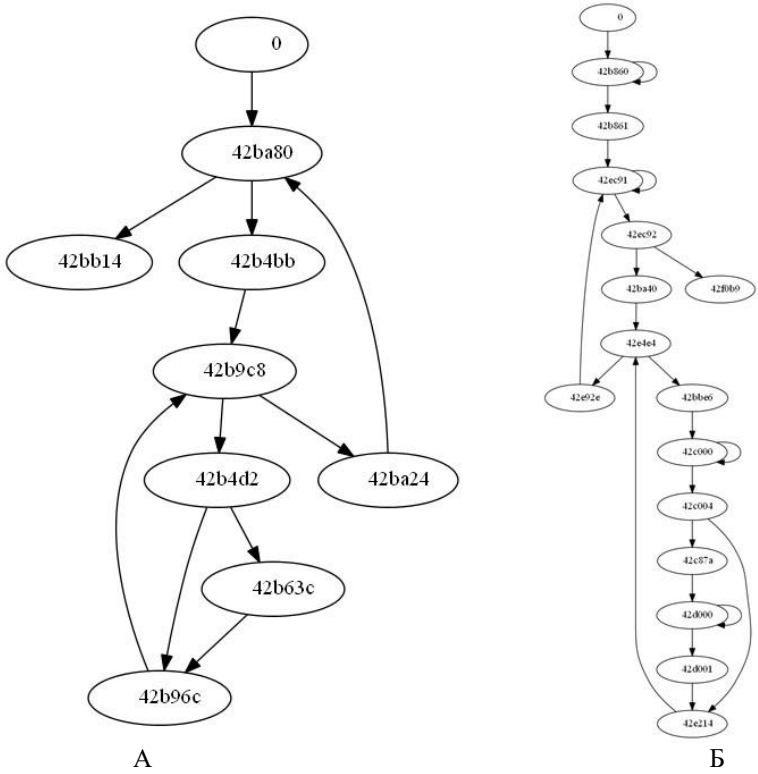


Рис.2 Восстановленный граф потока управления после запутывания с помощью CodeVirtualizer (А) и CodeVirtualizer с Multi Branch Technology (Б)

5. Результаты экспериментов

Описанный подход к восстановлению потока управления запутанной программы был опробован на модельных примерах, код которых был запутан демонстрационными версиями обфускаторов VMProtect, CodeVirtualizer, Enigma Protector и Safengine. В качестве основного примера, используемого в данной статье, использовалась программа, реализующая алгоритм пузырьковой сортировки. Исходный граф потока управления, полученный с помощью IDA Pro, приведен выше на рисунке 1А.

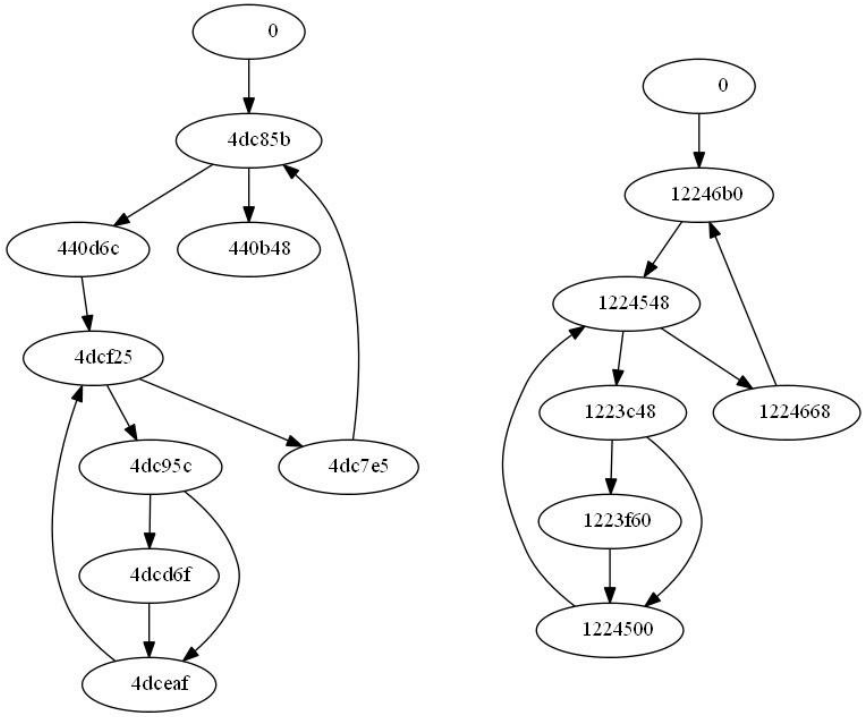


Рис. 3.Восстановленный граф потока управления после запутывания посредством VMProtect (А) и Enigma Protector (Б)

Запутывание посредством CodeVirtualizer и Safengine выполнялось в двух вариантах: 1)только с использованием виртуальной машины и 2)виртуализацией с добавлением Multi Branch Technology. На рисунке 1 приводятся результаты восстановления потока управления программы, запутанной с помощью CodeVirtualizer. Можно видеть, что для программы без дополнительного запутывания (рис. 1А) поток управления восстановлен

точно. Восстановленный граф потока управления после виртуализации с добавлением Multi Branch Technology (рис. 2Б) содержит больше базовых блоков и переходов, то есть непрозрачные предикаты не были удалены из анализируемой трассы. Это связано с тем, что для экспериментов использовалась версия системы TrEx 2.0, в которой формат трассы не поддерживает перемещения фрагментов трассы и, соответственно, удаления непрозрачных предикатов вида $\boldsymbol{P^T}$. Однако эти предикаты распознаются соответствующим модулем расширения системы и помечаются в трассе для удаления в будущих версиях. Что касается восстановления потока управления исходной программы, то и в этом варианте оно выполнено полностью.

Восстановление потока управления программы, запутанной с помощью VMProtect и Enigma Protector, также было выполнено успешно (рис. 3 А,Б).

В то же время, демонстрационная версия обфускатора Safengine при первых попытках восстановления потока управления вызвала ряд сложностей как в базовом варианте (только виртуализация), так и с использованием технологий дополнительного запутывания. Восстановленный граф потока управления в обоих случаях содержит избыточные передачи управления, анализ которых нами пока не завершен. Поэтому в данной статье мы воздерживаемся от оценки результатов, полученных применительно к данной системе защиты приложений.

6. Заключение

Описанный в разделе 4 подход к восстановлению потока управления был реализован прототипом комплекса программных инструментов в рамках среды динамического анализа защищенного бинарного кода TrEx. Реализация представляет собой набор модулей расширения системы, для корректной работы которых требуется построение графа потока управления для каждой функции анализируемого приложения. Для ускорения анализа с использованием алгоритмов слайсинга желательно построение графа зависимостей.

При тестировании прототипа были выявлены ограничения и неточности реализованного подхода, в частности, невозможность удаления из трассы части распознанных непрозрачных предикатов, а также неточность восстановления потока управления программы, запутанной с помощью обфускатора Safengine. Ограничения, связанные с форматом трассы и невозможность удаления непрозрачных предикатов $\boldsymbol{P^T}$, планируется преодолеть в одной из будущих версий. Восстановление потока управления после запутывания с помощью Safengine требует дальнейшего исследования; эти работы продолжаются в настоящее время наряду с работами по исследованию задачи восстановлению потока данных запутанной посредством виртуализации программы. В целом мы считаем, что первые эксперименты демонстрируют возможность практического применения предложенного

подхода для преодоления комплексного запутывания потока управления приложений в бинарных кодах.

В дальнейшем предполагается продолжить работы по улучшению и расширению применимости предложенной методики и ее программной реализации. Также планируется реализовать восстановление потока данных виртуализованного кода и объединение подтрасс, полученных в результате раздельного восстановления потока управления и потока данных. Также за рамками данной статьи осталось описание методики, с помощью которой в трассе локализуются участки с различными видами запутывания, что позволяет существенно ускорить работу описанных в статье алгоритмов за счет их применения к предварительно выбранным частям трасс приложения.

Литература

[1] Christian Collberg, Clark Thomborson, Douglas Low. A Taxonomy of Obfuscating Transformations. / Technical Report #148. Department of Computer Science, The University of Auckland. July, 1997

[2] А.В. Чернов. Анализ запутывающих преобразований программ. // Труды Института системного программирования РАН, том 3, 2002 г. Стр. 7-38.

[3] S.K. Udupta, S.K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In Proc. 12th IEEE Working Conference on Reverse Engineering, pp.45-54, November 2005

[4] Mila Dalla Preda, Matias Madou, Koen De Bosschere, Roberto Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. // In Proc. Intern. Conf on Algebraic Methodology and Software Technology. 2006 pp. 35-50.

[5] R. Rolles. Unpacking virtualization obfuscators. In Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT'09), August 2009

[6] B. Lau. Dealing with virtualization packer. In Second CARO Workshop on Packers, Decryptors, and Obfuscators, May 2008.

[7] M. Sharif, A. Lanzi, J. Griffin, and W. Lee. Automatic reverse engineering of malware emulators. In Proc. 2009 IEEE Symposium on Security and Privacy, May 2009.

[8] Kevin Coogan, Gen Lu, Saumya Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantic-Based Approach. In Proc. 18-th ACM Conference on Computer and Communication Security, October 2011.

[9] Тихонов А.Ю., Аветисян А.И., Падарян В.А. Методика извлечения алгоритма из бинарного кода на основе динамического анализа // Проблемы информационной безопасности. Компьютерные системы. — 2008. — Т. №3. — С. 66–71.

[10] Падарян В.А., Гетьман А.И., Соловьев М.А. Программная среда для динамического анализа бинарного кода // Труды Института Системного Программирования. — 2009. — Т. 16. — С. 51–72.

[11] B. Korel, J. Laski. Dynamic program slicing. // Information Processing Letters, vol. 29, issue 3, pp. 155-163. — 1988.

[12] VMPSoft. VMProtect. http://www.vmpsoft.com

[13] Oreans Technologies. CodeVirtualizer. http://www.oreans.com

[14] The Enigma Protector. http://enigmaprotector.com

[15] Safengine. Safengine Protector. http://www.safengine.com

[16] Setisoft Technology. Private Exe Protector. http://www.setisoft.com

Obsidium Software. Obsidium. http://www.obsidium.de

An Approach to Reconstruction of Control Flow of an Obfuscated Program

Ilya N.Ledovskikh, Maxim G.Bakulin
ISP RAS, Moscow, Russia
{il, bakulinm}@ispras.ru

Abstract. Control flow obfuscation is one of widespread methods used to protect application binary code from analysis. The obfuscation transformations dramatically increase the complexity of separation and recognition of the algorithm and data structures. This paper provides a short survey of obfuscating transformations and tools, essentially the control flow obfuscations like opaque predicates, dispatcher and code virtualization. Our research was concentrated in area of de-obfuscation based on dynamic binary code analysis methods. An approach to locating of potentially opaque predicates and accompanying obfuscations (like dead and redundant code insertion) in program trace is given. The main idea underlying the described method of opaque predicate detection is extraction of predicate code via dynamic program slicing. Next, we apply some heuristics to automatically reduce predicate list, and then our tool provides the analyst with information about potentially opaque predicates found in application binary code.

Another control flow obfuscation eliminated by described tool was code virtualization. An algorithm for control flow graph recovery of virtualized code is based on virtual program counter analysis. The corresponding tool was implemented and successfully tested on several applications that were protected via various virtualization obfuscators (VMProtect, CodeVirtualizer, Safengine, etc.). The experimental results described in this paper demonstrate the practical adaptability of implemented method to de-obfuscate virtualized applications.

Keywords: obfuscating transformations, opaque predicates detection, virtualization, control flow graph recovery.

References

[1]. Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Technical report #148, Department of Computer Sciences, The University of Auckland, 1997. doi: 10.1.1.38.9852

[2]. Chernov A.V. Analiz zaputyvayushhikh preobrazovanij programm [Analysis of Program Obfuscating Transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol. 3, no. 1, pp. 7-38 (in Russian).

[3]. Udupa S. K., Debray S. K., Madou M. Deobfuscation: Reverse Engineering Obfuscated Code. Proc. 12th IEEE Working Conference on Reverse Engineering, 2005, pp. 45-54. doi: 10.1.1.302.7805

[4]. Dalla Preda M., Madou M., Bosschere K. D., Giacobazzi R., Opaque Predicates Detection by Abstract Interpretation., Proc. Intern. Conf on Algebraic Methodology and Software Technology, 2006, pp. 81-95. doi: 10.1007/11784180_9

[5]. Rolles R. (2009), Unpacking virtualization obfuscators. Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT'09), 2009, pp. 1-1. doi: 10.1.1.159.7552

[6]. Lau B. Dealing with virtualization packer. Second CARO Workshop on Packers, Decryptors, and Obfuscators, 2008, pp.275-284. doi: 10.1145/2046707.2046739
[7]. Sharif M., Lanzi A., Griffin J., Lee W. Automatic reverse engineering of malware emulators. Proc. 2009 IEEE Symposium on Security and Privacy, 2009, pp. 94-109. doi: 10.1109/SP.2009.27
[8]. Coogan K., Lu G., Debray S. Deobfuscation of Virtualization-Obfuscated Software: A Semantic-Based Approach. Proc. 18-th ACM Conference on Computer and Communication Security, 2011, pp. 275-284. doi: 10.1145/2046707.2046739
[9]. Tikhonov A.Y., Avetisyan A.I., Padaryan V.A. Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of Exploring of an Algorithm from Binary Code by Dynamic Analysys]. Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy [Problems of Computer Security. Computer Systems], 2008, vol. 3, pp. 66–71 (in Russian).
[10]. Padaryan V.A., Get'man A.I., Solov'ev M.A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software Framework for Dynamic Binary Code Analysis]. Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, no. 1, pp. 51-72 (in Russian).
[11]. Korel B., Laski J. Dynamic program slicing. Information Processing Letters, 1988, vol. 29, issue 3, pp. 155-163. doi: 10.1016/0020-0190(88)90054-3
[12]. VMPSoft. VMProtect. http://www.vmpsoft.com
[13]. Oreans Technologies. CodeVirtualizer. http://www.oreans.com
[14]. The Enigma Protector. http://enigmaprotector.com
[15]. Safengine. Safengine Protector. http://www.safengine.com
[16]. Setisoft Technology. Private Exe Protector. http://www.setisoft.com
[17]. Obsidium Software. Obsidium. http://www.obsidium.de

Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования

В.Н. Игнатьев
valery.ignatyev@ispras.ru

Аннотация. Для достижения переносимости, надёжности и безопасности программ на C и C++ может применяться введение дополнительных ограничений на язык и стиль программирования. В работе предложен новый метод формализации и классификация таких ограничений, описана система их автоматической проверки, основанная на применении быстрого и нетребовательного к ресурсам статического анализа, использующего средства компилятора, а также способ её интеграции с распространёнными системами сборки проектов.

Ключевые слова: стандарты написания исходного текста программ; статический анализ; LLVM; CLANG.

1. Введение

Языки программирования C и C++ часто используются в крупных проектах. Поиск и исправление ошибок в большом проекте является сложным и дорогим. Статические анализаторы могут применяться на ранних стадиях разработки и позволяют обнаруживать ошибки до тестирования, однако большинство из них требует продолжительного времени для анализа. Поэтому для раннего обнаружения ошибок в программе может быть использован легковесный анализатор, работающий перед компиляцией.

При разработке крупных проектов часто используется введение дополнительных ограничений на возможности языка разработки. Их можно разделить на стилистические, синтаксические и ситуационные. К стилистическим относятся правила форматирования и именования объектов в исходном тексте программ, которые повышают удобочитаемость и облегчают понимание. Синтаксические ограничения позволяют избегать потенциально небезопасных конструкций языка программирования, допускаемых стандартом. Ситуационные ограничения определяют условия, при которых возникают или могут возникать ошибки.

В программах на языках C и C++ неправильное использование некоторых возможностей, как например, неявное приведение типов, работа с памятью и указателями, конструкции, результат работы которых не определён, часто приводит к серьёзным ошибкам. Данная проблема имеет два пути решения:

1) Создание близкого по синтаксису языка, избавленного от основных недостатков. Например, Java и C# имеют строгую типизацию и поддерживают автоматическую работу с памятью.

2) Введение ограничений языка, позволяющих избегать ситуаций, приводящих к ошибкам.

Например, распространённые ошибки, возникающие при разработке на C и C++ изучены и составлены рекомендации для программистов, такие как MISRA C 2004[1], MISRA C++ 2008[2], JSF[3], HICPP[4].

Компилятор проверяет соответствие программы стандарту языка. Используя перечисленные в [1-4] рекомендации, можно составить набор ограничений для использования в конкретном проекте. В результате добавления ограничений к языку C или C++ получится новый язык, сокращающий возможности исходного. Важно, чтобы полученный язык был достаточным для написания программы и набор ограничений был непротиворечивым. Полученный язык не соответствует стандарту исходного (C или C++), поэтому проверяется компилятором не в полном объёме. Его проверка может осуществляться с помощью статического анализатора, описание которого предлагается в работе.

В связи с тем, что описание рекомендаций в сборниках[1-4] производится на естественном языке, они могут по разному интерпретироваться разработчиками и неоднозначно формализоваться. Кроме того, не существует стандартизированной или общепризнанной нотации для формализации ограничений. Различные записи одного и того же правила могут требовать принципиально отличающихся по времени работы, сложности, наличию ложных срабатываний и полученным результатам алгоритмов их проверки. В работе рассмотрено несколько возможностей задания правил и предложен формальный способ их определения. Также описана инфраструктура, на основе которой можно задавать и проверять рассматриваемые ограничения.

Данная работа имеет следующую структуру: в разделе 2 рассматриваются возможные ограничения, существующие методы их формальной записи и классификации, в разделе 3 предлагается метод формального задания ограничений, в 4 разделе рассмотрена их классификация на основе формальной записи. Далее в разделе 5 описана разработанная инфраструктура проверки правил. В заключительном разделе 6 представлены результаты работы анализатора, сформулированы выводы и предложены идеи для дальнейших исследований.

2. Обзор существующих ограничений языка, методик их формализации и классификации

Рекомендации для языков C и C++, на основе которых задаются правила, перечислены в сборниках MISRA C 2004[1], MISRA C++ 2008[2], JSF[3], HICPP[4], а также в книгах, посвящённых программированию на C и C++[6,7]. Они представлены в виде списка с описанием преимуществ использования, примеров нарушений и исправлений ошибок. Обычно этот список является одноранговым, т.е. не имеет структуры, поскольку не найдено однозначных критериев, согласно которым можно было бы систематизировать правила. Производители промышленных анализаторов группируют их по природе обнаруживаемых ошибок: форматирование текста программы; правила, связанные с обработкой исключений; конвенции именования; ошибки работы с памятью. Группировка по терминам языка, к которым правило имеет наибольшее отношение, также используется на практике. Основным недостатком описанных попыток классификации является их неоднозначность, поскольку каждое правило не имеет определённого места в предлагаемой структуре, и может быть отнесено к разным группам.

Задача классификации состоит в выборе признака, согласно которому производится разделение на группы. В статье [8] предлагается структуризация *по целям* правил, которые должны достигаться их соблюдением. Предложено 7 групп:

1. Ясность, чёткость, прозрачность: для избежания непонимания разработчиками.
2. Совместимость: для минимизации стоимости переноса.
3. Защита, безопасность.
4. Производительность.
5. Предсказуемость: для избежания неоднозначной интерпретации.
6. Простота: для сохранения простоты программы, минимизации стоимости последующего тестирования.
7. Правила, регламентирующие процесс разработки.

Систематизация правил, предложенная в [9], использует в качестве признака введённый автором термин «сложность» правила. В статье предложено 6 групп правил:

1. Тривиальные. Поиск простых шаблонов в исходном тексте.

Пример: «не использовать malloc()» (MISRA-C 20.4).

2. Синтаксические. Уточнения грамматики языка.

Пример: «нельзя использовать inline для методов класса» (HICPP 3.1.7).

3. Ограничения на типы. Требуется использование информации о типах.

Пример: «логические выражения могут использоваться в качестве операндов только для операторов &&, ||, !» (MISRA-C 12.6).

4. Структурные. Правило связано со статическими отношениями (наследование, член класса) между объектами в коде. Информация не содержится в дереве разбора, но полностью статична, как например, иерархия наследования.

Пример: «если виртуальная функция базового класса не переопределена в производных, то она не должна быть виртуальной» (HICPP 3.3.6).

5. Динамические. Правило определяет последовательность событий при выполнении. Использование графа потока управления недостаточно, требуется модель памяти.

Пример: «все автоматические переменные должны быть инициализированы» (MISRA-C 9.1).

6. Неавтоматизируемые. Правила, которые сложно или невозможно формализовать, а также использующие невычислимые свойства.

Пример: «реализация функциональности в классе не должна дублироваться» (HICPP 3.1.9).

Программные ошибки, с которыми связано правило, являются другим признаком для классификации. В основе классификации правил в этом случае лежит классификация ошибок, которая на данный момент изучена гораздо лучше. Ошибки можно группировать по их значимости, причинам, источникам. Описание методов классификации ошибок рассмотрено в статье [11].

С точки зрения [10] невозможно создать классификацию, в которой каждое правило будет принадлежать только одной категории. В связи с этим предлагается группировка небольшого количества правил по большому количеству категорий и построение дерева категорий.

Рассмотренные систематизации правил не позволяют решить задачу классификации, поскольку не задают формального критерия, согласно которому правило может быть отнесено к определённому классу. Кроме того, некоторые правила, описанные в [1-4] не могут быть отнесены ни к одной группе предлагаемых систематизаций. Для создания классификации требуется формальный язык определения правил, на основе которого могут быть заданы критерии классификации.

На данный момент не существует общепринятого языка описания правил, поэтому они задаются на естественном языке. В работах, исследующих методы их статической проверки, предложено несколько способов формального описания. Так в [9, 10] предложены декларативные языки на основе Пролога, позволяющие определять и проверять некоторое множество правил, а в [10] ещё и задавать базу знаний о программе. Например, правило «Переменные, являющиеся членами класса, не должны быть публичными (public)», записанное на языке из [10] выглядит следующим образом:

```
faulty_cldata :-
      get_classdefinition(Class),
```

```
      get_data_member_in_classdefinition(Class,
Var_decl),
      get_protection_flag_of_variable_decl(Var_decl,
Prot_flag),
      is_protection_flag_public(Prot_flag),
      writef(«Rule Violated: Public data member in
class»),
      print_name_and_location_of_classdefinition(Class).
```

Предложенный авторами язык используется не только для определения правил, но и для описания базы знаний, которая представляет собой набор фактов о программе, полученных во время её анализа. Для проверки соответствия программы заданному набору ограничений используется интерпретатор Пролога, исходными данными которого являются формализация правил, база знаний и библиотека описания интерфейсных функций, а в результате получается список предупреждений.

В большинстве крупных проектов существуют *неявные правила* [12]. К ним относятся ограничения, имеющие отношение только к конкретному проекту. Для их описания в статье [12] используется расширенный авторами язык шаблонов из статьи [13]. Правило «перед вызовом функции ChangeView необходимо установить глобальную переменную CtrlNo», записанное на предлагаемом языке выглядит следующим образом:

```
      $vg_1 = 'CtrlNo' // переменная CtrlNo – глобальная
      $f_1 = 'ChangeView' // функция ChangeView %%
      ^@$vg_1 = #; // отсутствие выражения
      @$f_1(#*);
```

Использованная нотация создавалась для рефакторинга программных систем, поэтому её применение для формализации ограничено. Она позволяет задавать лишь небольшое множество правил в связи с тем, что возможен доступ к небольшому числу объектов языка программирования.

Как видно из приведённых примеров, для записи формализации правил авторы предлагают различные синтаксические конструкции: декларативный проблемно-ориентированный язык, язык программирования общего назначения (Пролог).

3. Формализация ограничений

Ограничение языка программирования (правило) – это предикат, на вход которому подается модель программы или её части, а результатом является одно значение из двухэлементного множества {И, Л}, соответствующее истинности и ложности. Пусть **p** — корректная согласно стандарту программа на языке C или C++. Исходный текст программы является строкой символов. Пусть $\mathbf{m_s}$ – модель программы **p**, задаваемая в виде исходного текста программы и представляемая в виде строки. В процессе компиляции программа имеет несколько различных представлений. На вход лексическому

анализу компилятора подается модель $\mathbf{m_s}$ программы, а полученную в результате его работы модель назовем $\mathbf{m_{lex}}$. Она состоит из последовательности лексем и таблицы символов. Поскольку рассматриваемая программа корректна, то лексический анализ всегда успешно завершается и множество диагностических сообщений пусто. Объектом модели $\mathbf{m_{lex}}$ является лексема (токен), для которого заданы атрибуты: тип лексемы, положение в модели $\mathbf{m_s}$, ссылка на таблицу символов для некоторых типов лексем. Кроме перечисленных атрибутов, вычисляемых компилятором, анализатор создает дополнительные необходимые для анализа атрибуты. Например, компилятор не различает табуляцию, количество символов «пробел» и «перевод строки», которые используются для проверки стилистических правил.

Синтаксический анализ компилятора по модели $\mathbf{m_{lex}}$ строит модель программы $\mathbf{m_{synt}}$, состоящую из абстрактного синтаксического дерева, узлы которого делятся на два основных класса: узлы, соответствующие объявлениям функций, переменных, типов, классов и т.д. в программе, и узлы, соответствующие выражениям в программе. Объектом модели $\mathbf{m_{synt}}$ является узел дерева. Для каждого узла задан его тип, определяющий какому объекту в программе он соответствует, например, «объявление функции» или «бинарный оператор». В зависимости от типа, узел дерева имеет различные наборы атрибутов. После семантического анализа из модели $\mathbf{m_{synt}}$ строится модель $\mathbf{m_{sem}}$, отличающаяся наличием дополнительных атрибутов.

В процессе компиляции для выполнения оптимизаций обычно используется одно или несколько внутренних представлений программы. Одним из таких представлений является биткод LLVM, модель которого назовем $\mathbf{m_{llvm}}$. Объектом в данной модели является инструкция виртуальной машины LLVM с соответствующим её типу множеством атрибутов.

Программа может состоять из нескольких файлов. За один запуск компилятор обычно обрабатывает один модуль компиляции, которые затем объединяются в исполняемый файл редактором связей. Для каждого модуля компиляции, подающегося на вход редактору связей, существует представление в виде модели $\mathbf{m_{llvm}}$. Модель программы, состоящую из множества моделей $\mathbf{m_{llvm}}$ модулей компиляции, назовём $\mathbf{m_{im}}$.

Введённые модели программы задают область определения ограничений. В определении правила указано, что оно может применяться не только ко всей модели, но и к её части. Для каждой модели определён объект, являющийся минимальной неделимой частью. При разделении модели минимальная часть соответствует объекту данной модели и правило может применяться к каждому объекту или к их последовательности, определяемой моделью. Таким образом, алфавит формальной модели правил состоит из объектов модели программы. Для задания формальной модели правил необходимо введение функциональных символов и предикатов на множестве правил. Пусть r, s, q – правила, m – модель программы, тогда модель системы ограничений задается следующим образом:

Алфавит — множество объектов модели m**;**

Предикаты:

- Двуместный предикат *равенство*:
  $\forall r, s : =(r,s) \Leftrightarrow r = s \Leftrightarrow \forall m : r(m) = s(m);$

Функциональный символы (ФС):

- Одноместный ФС *отрицание:*
  $\forall r \exists s : s = \neg(r) \Leftrightarrow s = r \Leftrightarrow \forall m : r(m) = s(m);$
- Двуместный ФС *конъюнкция*:
  $\forall r, s \exists q : q = \wedge(r,s) \Leftrightarrow q = r \wedge s \Leftrightarrow \forall m : q(m) = r(m) \wedge s(m);$

Полученная модель изоморфна модели логики первого порядка и позволяет задавать одни правила с помощью других. Множество правил R является *непротиворечивым*, если $\exists m : \wedge_{r \in R} r = И$.

В качестве примера формального определения ограничения рассмотрим правило: «нельзя использовать операции сравнения на равенство и неравенство с операндами нецелочисленного типа». Для этого будем использовать модель программы $\mathbf{m}_{\mathbf{sem}}$. Операция сравнения в указанной модели представлена в виде узла дерева *BinaryOperation* с атрибутом *BinaryOperation::kind* равным *BinaryOperation::EQ* или *BinaryOperation::NEQ*. Для данного узла заданы два операнда, также являющиеся узлами дерева разбора, и доступные с помощью атрибутов *LHS* и *RHS*. Независимо от представляемого ими объекта программы, для них определён атрибут *Type* соответствующий типу рассматриваемого узла. Введем специальное правило, принимающие на вход узел объявления типа, и проверяющее, является ли он типом с плавающей запятой.

```
isFloatingPoint (Type t) ⇔
      t.CanonicalType = BuiltinType::Float ∨
      t.CanonicalType = BuiltinType::Double ∨
      t.CanonicalType = BuiltinType::LongDouble
```

Тогда рассматриваемое правило будет иметь вид:

```
hasFpEqNeqCmp (BinaryOperator bop) ⇔
      (bop.kind == BinaryOperation::EQ ∨ bop.kind ==
BinaryOperation::NEQ) ⇒
      (isFloatingPoint(bop.LHS.Type) ∧
isFloatingPoint(bop.RHS.Type))
```

Для записи ограничений не всегда удобно использовать язык логики предикатов, потому что сформулированные подобным образом ограничения получаются громоздкими и трудны для понимания. В разработанном анализаторе для задания правил используется язык общего назначения C++. Для каждой модели программы в анализаторе присутствуют соответствующие объекты с методами доступа к их атрибутам, например, лексема представлена классом Token с атрибутами TokenKind, SourceLocation . Правило

представляет собой класс, реализующий необходимые методы базового класса. На вход методам обычно подается объект модели (лексема, узел дерева разбора требуемого типа). Разработанная библиотека вычисляет ряд дополнительных атрибутов, упрощающих задание правил.

4. Классификация правил

В разделе о формализации описаны модели программы. Ограничения задаются на определённой модели, в которой доступны все необходимые для формализации атрибуты. Для каждой модели определено подмножество правил и их пересечение является пустым множеством. Таким образом, правила однозначно разделяются на классы в соответствии с моделью программы, на которой они задаются. В компиляторе одна модель обычно строится на основе другой. Например, для построения $\mathbf{m_{llvm}}$ используется $\mathbf{m_{sem}}$, которая в свою очередь использует $\mathbf{m_{synt}}$, строящуюся на основе $\mathbf{m_{lex}}$. Предложенная классификация является однозначной и характеризует сложность правила, т.е. относительное количество ресурсов, требуемых для его проверки. Модель программы задаёт множество объектов и атрибутов. Использованное в формальной записи правила множество объектов и атрибутов определяет класс правила.

В качестве примера рассмотрим правило «не использовать malloc()»(MISRA-C 20.4), формально записанное следующим образом на основе модели $\mathbf{m_{synt}}$:

isMallocUsed(CallStmt s) $\Leftrightarrow$ *s.CalledFunc.Name = «malloc»*

Узел дерева типа CallStmt имеет атрибут, указывающий на объявление вызываемой функции, который в свою очередь имеет атрибут — имя функции. Таким образом, правило задается в модели $\mathbf{m_{synt}}$ и принадлежит классу синтаксических правил.

С помощью предложенного критерия все правила можно распределить на 4 основных класса, строго определяющих положение каждого правила. При этом сохраняется возможность группировки правил внутри основных пунктов классификации по тем или иным нестрогим критериям.

4.1. Лексические ограничения

В обычном режиме работы компилятор не сохраняет данные лексического анализа (информацию о лексемах), а также пропускает лексемы, соответствующие комментариям, не различает пробелы, табуляцию, перевод строки и их количество, поскольку данная информация не влияет на семантику программы. Однако для определения некоторых правил необходима данная информация. В модели $\mathbf{m_{lex}}$ вся информация присутствует и доступна в анализаторе.

К классу лексических ограничений относятся правила, формализуемые на основе объектов, описывающих лексемы. В терминах неформального языка описания правил к данному классу относятся ограничения, содержащие слова:

ключевое слово языка и их полный список, строка, символ, токен, лексема, знаки операций («&&», «?:», «+» и т.д.), комментарий, директивы препроцессора и их список, идентификатор.

Большую часть ограничений данного класса составляют правила форматирования исходного текста программы, правила использования комментариев, директив препроцессора и макроопределений. В качестве примеров лексических правил можно привести следующие:

«каждый модуль компиляции должен содержать в начале комментарий в формате doxygen[14]» или «каждый заголовочный файл должен быть заключён в директивы #ifndef FILE_NAME_H #define FILE_NAME_H ... #endif».

4.2. Синтаксические ограничения

Синтаксический анализ или *разбор(parsing)* являются второй фазой компиляции. Анализатор использует информацию о токенах, полученных после лексического анализа для создания древовидного промежуточного представления, узлы которого являются объектами модели $\mathbf{m}_{\mathbf{synt}}$. Модель $\mathbf{m}_{\mathbf{synt}}$ определяет объекты, соответствующие каждому типу выражений, например, унарный и бинарный оператор, выражение if-then-else и каждому типу объявлений, например, функции, типа, переменной, класса. Для этих объектов определены атрибуты, такие как имя, аргументы, квалификаторы в соответствии с типом объекта.

Правила, формализованные с помощью объектов модели $\mathbf{m}_{\mathbf{synt}}$, принадлежат классу синтаксических. Связь между $\mathbf{m}_{\mathbf{synt}}$ и $\mathbf{m}_{\mathbf{lex}}$ обеспечивается в контрольных точках, соответствующих началу объекта в $\mathbf{m}_{\mathbf{synt}}$, т.е. для выражения *if* задан атрибут StartToken, указывающий на лексему - ключевое слово *if* . Таким образом, например, обеспечивается проверка правила «перед каждой функцией должен быть комментарий в формате doxygen».

Большую часть синтаксических правил составляют конвенции именования переменных, функций, типов; конвенции использования функций стандартных библиотек; проверка форматной строки; проверка наличия комментариев для функций, переменных, блоков и т.д.; уточнение грамматики языка, не требующее информации о типах.

4.3. Контекстные ограничения

К классу контекстных правил относятся ситуационные ограничения, зависящее от состояний нескольких объектов модели программы. Например, правило «недопустимо отсутствие оператора return в не void функции» зависит от состояния объекта, соответствующего объявлению функции, и от наличия в теле функции объекта ReturnStmt на всех путях выполнения. При этом проверка второго условия производится в контексте наличия первого.

Для определения правил данного класса используются две модели программы: $\mathbf{m}_{\mathbf{sem}}$, отличающаяся от $\mathbf{m}_{\mathbf{synt}}$ наличием дополнительных атрибутов, и $\mathbf{m}_{\mathbf{llvm}}$ –

промежуточное языково-независимое внутреннее представление, объектом которой является инструкция виртуальной машины LLVM. В связи с этим, в данном классе можно выделить подклассы языково-зависимых и языково-независимых ограничений, заданных на $\mathbf{m}_{\mathbf{sem}}$ и $\mathbf{m}_{\mathbf{llvm}}$ соответственно.

Статический семантический анализ осуществляет проверку исходной программы на семантическую согласованность со стандартом языка на основе синтаксического дерева и информации из таблицы символов. Во время анализа осуществляется распространение информации о типах. На данном этапе компилятор решает похожую на поставленную задачу: осуществляет проверку правил, только их источником является стандарт языка.

Большинство правил описывается в терминах языка программирования, поэтому для них лучше подходит языково-зависимое представление программы. Проверка формализованных с помощью модели $\mathbf{m}_{\mathbf{sem}}$ правил осуществляется после семантического анализа. Все доступные для формализации атрибуты делятся на вычисляемые компилятором и анализатором. Для вычисления некоторых атрибутов на модели $\mathbf{m}_{\mathbf{sem}}$ анализатор строит дополнительные структуры данных, такие как, например, граф потока управления, граф вызовов, информация о границах значений переменных.

Для правил, формализованных на модели $\mathbf{m}_{\mathbf{llvm}}$ доступны атрибуты, вычисляемые в процессе анализа и оптимизаций компилятором. К ним относятся атрибуты для доступа к информации о потоках данных и управления, анализа псевдонимов, циклов.

В качестве примера приведено несколько групп правил, принадлежащие классу семантических:

1. Структурные правила внутри одного модуля компиляции, связанные со статическими отношениями в программе, не представленными в дереве разбора (наследование, член класса), проверка которых полностью осуществляется внутри одного модуля компиляции. Подробнее о проверке таких правил во всей программе можно прочитать в [9].

2. Ограничения на типы.
1) Нельзя использовать оператор сравнения на равенство для операндов не целочисленного (с плавающей запятой) типа .
2) Нельзя использовать переменные, квалифицированные как volatile в выражениях более одного раза.

3. Уточнения грамматики языка.
1) Отсутствие оператора return в функции, определённой не void .
2) Ограничения преобразования типов. (Неявное преобразование большего к меньшему, преобразование указателя на функцию к типу не «void *»).

В качестве примера правил, реализуемых на языково-независимом внутреннем представлении, можно привести следующие группы, использующие

1. анализ циклов: счётчик цикла должен изменяться только в одном месте, условие цикла не должно изменяться внутри цикла;
2. продвижение констант и анализ границ значений переменных: разыменование нулевого указателя, выход за границы массива.

4.4. Межмодульные статические правила.

Следующим после компиляции обычно выделяется этап связывания (linking), во время которого результирующая программа собирается из объектных файлов и устанавливаются связи между функциями и переменными, определёнными в разных модулях и библиотеках. На этом этапе современные компиляторы могут также осуществлять ряд оптимизаций, называемых оптимизациями времени связывания, а также он является подходящим для проверки межмодульных статических правил. Исходными данными для проверки этой группы правил являются файлы с промежуточным представлением (например, LLVM), а также специально экспортированная и сохранённая информация, собранная на предыдущих этапах. Для стадии связывания можно выделить следующие группы правил: анализ исключений, поиск взаимных блокировок, и race condition; проверка ряда структурных правил, требующих информацию обо всей программе; проверка tainted данных с использованием всей программы.

Предлагаемая классификация является однозначной и позволяет определять оптимальную последовательность анализа правил, а также сравнительную сложность (количество ресурсов), необходимые для осуществления их проверки. Таким образом можно ограничивать количество сообщений, не проводя ресурсоёмких вычислений.

Предложенная классификация включает в себя описанные в [12] неявные правила, которые специфичны для конкретной системы, редко описаны в спецификациях и содержатся в опыте разработчиков программы. В качестве примера такого правила можно привести следующее: «перед вызовом функции A необходимо установить глобальную переменную B». В зависимости от необходимых для их проверки данных о программе они распределены по нескольким классам, например, приведённое правило принадлежит классу контекстных правил.

5. Подсистема проверки правил

5.1. Особенности основных компонентов системы

Для реализации анализатора используются открытые компиляторные инфраструктуры LLVM[15] и CLANG[16]. Разработанная система состоит из

трёх основных компонентов: подсистема определения правил, сбора информации и планировщик правил.

Для определения ограничений применяется язык C++, используемый в реализации LLVM и CLANG. Правила в анализаторе задаются в виде классов, унаследованных от базового класса Pass, определяющего набор интерфейсных методов. Список необходимых правилу объектов модели программы передается с помощью параметров шаблона базового класса. Для обработки каждого объекта правило должно определить соответствующий метод. Например, правилу проверки исключений требуются объекты CXXThrowExpr, CXXCatchStmt, CallExpr для сбора информации и необходим запуск в конце обработки модуля компиляции для анализа. Тогда его объявление будет выглядеть следующим образом:

```
class ExceptionObjectCheckRule :
   public Pass<process::ASTStmt<CXXThrowExpr>,
               process::ASTStmt<CXXCatchStmt>,
               process::ASTStmt<CallExpr>,
               process::TranslationUnitFinalize> {
  public:
    void processStmt(const CXXThrowExpr *, RCContext *)
const;
    void processStmt(const CXXCatchStmt *, RCContext *)
const;
    void processStmt(const CallExpr *, RCContext *)
const;
    void finalizeTU(const TranslationUnitDecl *,
RCContext *) const;
};
```

Подсистема определения правил предоставляет интерфейс для доступа к объектам модели программы и их атрибутам. Использование C++ позволяет создать библиотеку, в которой объекты модели программы представлены в виде классов, а доступ к их атрибутам осуществляется с помощью методов. Например, термин лексема имеет следующий интерфейс:

```
class Token {
public:
      TokenKind getKind();
      std::string getContent();
      SourceLocation getSourceLocation();
      bool hasLeadingSpace();
}
```

Подобным образом определены остальные объекты. Использование методов позволяет получать доступ к атрибутам терминов по мере их вычисления компилятором или анализатором. В качестве примера рассмотрим правило «не использовать пробелы вокруг '.', '->' ». Для его проверки недостаточно

информации об одной лексеме, поэтому на вход поступает список лексем, и реализация выглядит следующим образом:

```
class SpaceAroundMemberAccessRule : public Pass <
process::TokenList > {
public:
  void processTokenList(const TokenList lst) const {
    for (auto i = lst.begin(), e = lst.end(); i != e;
++i) {
      Token *tok = *i;
      if ((tok->getKind() == TokenKind::arrow) ||
         (tok->getKind() == TokenKind::period)) {
        Token *next = *(i + 1);
        if (tok->hasLeadingSpace() || next-
>hasLeadingSpace()))

ReportRuleCheckerMessage(rcdiag::space_around_member_acc
ess, Token->getSourceLocation());
      }
    }
  }
};
```

Класс, задающий правило, получает доступ только на чтение к общим структурам данных анализатора, что обеспечивает их неизменность. Данные, необходимые для проверки ограничения в пределах одного модуля компиляции, хранятся в полях класса. Это создает возможность параллельной обработки объектов модели разными правилами. Описанный интерфейс позволяет переписать формальную запись правил на языке C++.

Подсистема сбора информации также использует интерфейс определения правил для доступа к объектам модели программы. Результат её работы необходим при вызове методов доступа к вычисляемым анализатором атрибутам объектов. Часть из них вычисляется непосредственно в момент вызова. Это позволяет исключить избыточный анализ в случае, если атрибут объекта не потребуется ни для одного правила. Для часто используемых атрибутов анализ осуществляется перед запуском проверки ограничений. Собранная информация сохраняется в специальное хранилище – базу данных SQLite[19] и доступна несколькими независимыми правилами при анализе разных модулей компиляции, т. е. в нескольких запусках анализатора. Таким образом функционирует проверка межмодульных правил. Они состоят из классов собирающих и сохраняющих в базу необходимую информацию, обрабатываемую на этапе связывания. Хранилище используется, например, при анализе исключений, которые могут возникать и перехватываться внутри функций, определённых в разных модулях компиляции. Подсистема работы с базой данных используется менеджером правил и подсистемой сбора информации. Возможность сохранения результата работы анализатора в БД

позволяет выполнять SQL запросы и, например, собирать статистику. Планировщик сохраняет в БД информацию о просмотренных заголовочных файлах, что исключает их повторный анализ и сокращает время работы анализатора.

В процессе анализа планировщик правил обходит промежуточное представление программы (список лексем, дерево разбора, граф потока управления, граф вызовов) и вызывает необходимые интерфейсные методы правил. Поскольку правила не меняют внутреннее представление, то они могут запускаться параллельно, что сокращает время анализа. Упрощённая схема представлена на рисунке 1.

Использование достаточно низкоуровневого языка C++ совместно с планировщиком проверки ограничений сокращает время анализа и обеспечивает возможность написания новых и параметризации существующих проверок.

5.2. Использование анализатора в процессе разработки

Разработанный анализатор поддерживает различные системы сборки проектов для автоматического запуска и анализа только той части исходных текстов, которая компилируется в результирующую программу. Для каждого запущенного при сборке процесса с помощью переменной окружения LD_PRELOAD принудительно загружается разработанная динамическая библиотека, которая размещается в памяти и производится динамическое связывание. В процессе компиляции проекта функции группы exec (execve, execv, execl, execvp, execlp) заменяются описанными в библиотеке. Новые функции анализируют командную строку и запускают анализатор в случае обнаружения там пути к компилятору или редактору связей. Такая система позволяет использовать практически любой компилятор и любую систему сборки. Для удобства используется программа-оболочка, обрабатывающая входные параметры и запускающая команды сборки. Предложенный подход был также адаптирован для системы кросс-компиляции scratchbox[20].

5.3. Результаты работы анализатора

Реализованная система способна обнаруживать более 50 различных дефектов программ. В результате запуска анализатора на примере LLVM с CLANG (~1250 тыс. строк кода) были получены следующие результаты:

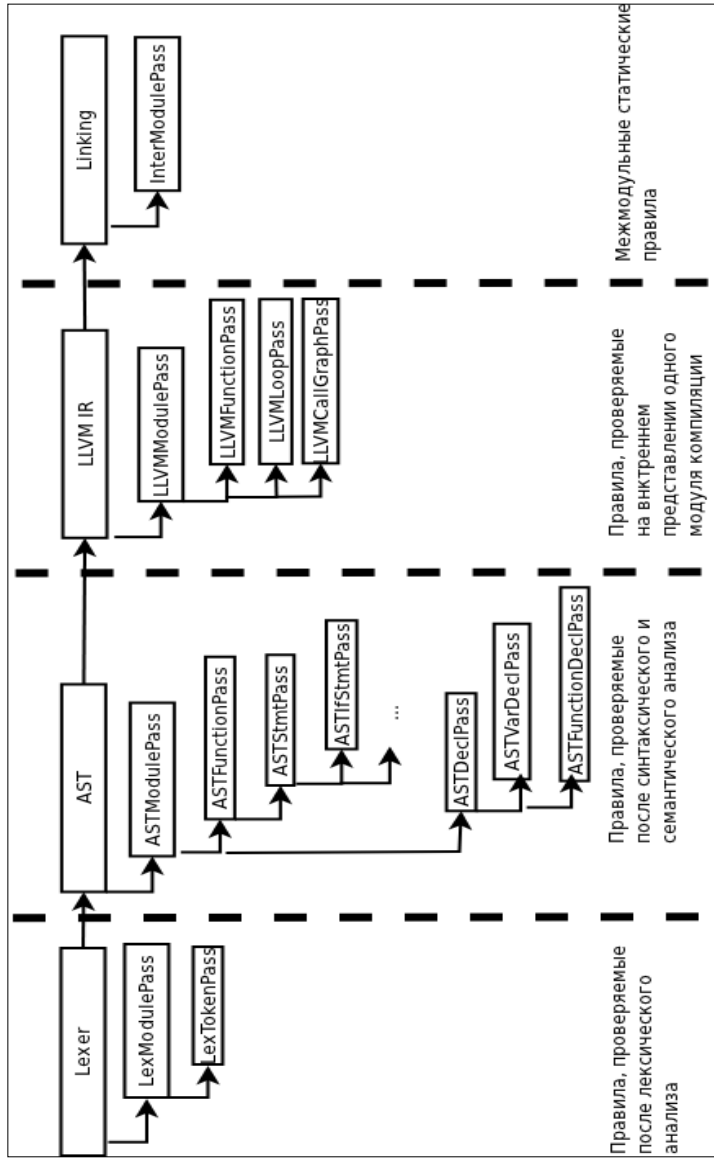


Рис. 1. Структура проверки правил

Коли-чество ошибок	Название правила	Комментарий
1	SIZE_CHECK	Тип аргумента sizeof не соответствует типу результирующего указателя
3	EXCEPT-15_DMC	Исключение, имеющие тип класс должно перехватываться по ссылке
7	MISRA2004-11_1_DMC	Недопустимое преобразование между указателем на функцию и типом не void*
11	OOP-24	Деструктор в базовом классе должен быть объявлен virtual
13	MISRA2004-12_2_d	Результат зависит от порядка вычисления аргументов
25	PB-17	Присваиваемое значение больше размера типа unsigned char
30	SECURITY-12_DMC	Использование небезопасной функции работы со строками
31	MRM-43	Переопределённый оператор присваивания должен копировать все члены класса, включая базовые
41	MISRA2004-12_8_a_DMC	Аргумент оператора сдвига должен быть больше 0 и меньше битового размера типа сдвигаемого значения
76	MISRA2004-13_3_DMC	Недопустимо сравнение на равенство неравенство не целочисленных операндов
116	PB-20	Недопустим slicing параметров функции / возвращаемого значения
118	MISRA2008-15_3_4	Исключение не имеет обработчика
124	MISRA2004-10_1_b-3	Неявное преобразование между целочисленным и не целочисленным значением
129	MISRA2004-17_2	Вычитание указателей допустимо только для указателей на один и тот же массив
150	MISRA2004-17_3	Сравнение (<, <=, >, >=) указателей допустимо только для указателей на один и тот же массив

156	MISRA2004-12_3_DMC	Аргумент sizeof имеет sideeffect
218	MISRA2004-13_1_DMC	Оператор присваивания использован в выражении логического типа
275	MISRA2004-12_2_b_DMC	Результат зависит от порядка вычисления аргументов функции
796	INIT-06	Все переменные должны быть инициализированы в конструкторе

Табл. 1. Результаты работы анализатора на LLVM и CLANG(~1250 тыс. строк кода)

В случае использования программы-оболочки, которая наряду с обычной сборкой приложения осуществляет запуск CLANG и нескольких проходов анализа представления LLVM, были получены следующие результаты:

Пакет	GCC, сек.	GCC + Анализатор LLVM, сек	Замедление, %
binutils	73.88	90.52	22.52%
glib	21.14	25.96	22.80%
sqlite	38.58	46.80	21.31%

Большая часть сообщений анализатора была проанализирована вручную и подтверждена их истинность.

Таким образом, использование анализатора замедляет сборку приблизительно на 22%. Приведённые результаты показывают, что применение встроенного в систему сборки анализатора допустимо при каждой компиляции и позволяет обнаруживать ошибки на самой ранней стадии разработки.

6. Заключение

Для решения задачи автоматической проверки настраиваемых ограничений и поиска распространённых ошибок в исходном коде программы система должна удовлетворять следующим критериям:

1. Малые дополнительные ресурсы, требуемые для проверки программы.

Промышленные статические анализаторы реализуют большое количество правил, а также способны обнаруживать сложные дефекты, требующие долговременного анализа. В разработанной системе существует планировщик правил, обеспечивающий оптимальную последовательность их проверки, а

также не используются эвристические и ресурсоёмкие алгоритмы. В результате этого время работы анализа сравнительно невелико.

2. Интеграция в процесс разработки

Описанная в разделе 5.3 подсистема интеграции обеспечивает возможность удобного использования анализатора с различными компиляторами и системами сборки проекта.

3. Использование результатов статического анализа на самых ранних этапах разработки.

Первым этапом тестирования написанного текста программы является компиляция. Разработанный анализатор может работать перед компиляцией.

4. Возможность отключения и конфигурирования существующих проверок, а также добавления новых.

В большинстве проектов конвенции написания исходных текстов уже выбраны, поэтому система имеет возможность гибко подстраиваться под существующие, а также позволяет не только параметризовать правила, но и задавать новые. Для этого применяется подсистема определения правил и планировщик, который в зависимости от принадлежности правил тому или иному пункту классификации, определяет порядок их проверки.

Дальнейшие работы будут направлены на исследование применимости различных видов анализа программ, которые позволят осуществлять проверку более сложных ограничений, реализацию новых ограничений и построение системы ограничений для каждого из языков C и C++.

Список литературы

[1]. «Guidelines for the use of the C language in critical systems», MISRA-C:2004, 2004

[2]. «Guidelines for the Use of the C++ Language in Critical Systems», ISBN 978-906400-03-3 (paperback), June 2008.

[3]. «Joint Strike Fighter air vehicle C++ coding standards for the system development and demonstration program», Lockheed Martin Corporation, 2005

[4]. «High Integrity C++ Coding Standard Manual - Version 2.4», The programming research group, 2007

[5]. The annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990

[6]. «Code Complete», S. McConnel, Microsoft Press; 1 edition (January 1, 1993)

[7]. «The Practice of Programming», Brian W. Kernighan, Rob Pike, Addison-Wesley Professional; 1 edition (February 14, 1999)

[8]. «Coding standards for high-confidence embedded systems», Paul Anderson, MILCOM2008

[9]. «A Coding Rule Conformance Checker Integrated into GCC», Marpons, Marino, Carro, Herranz, Fredlund, Moreno-Navarro, and Polo, Electronic Notes in Theoretical Computer Science, 2007

[10]. «A tool for checking coding standards for C++», S. Mohammed Saleem (Master of technology thesis), 1999

[11]. «Standard error classification to support software reliability assessment», John B. Bowen, afips, pp.697, 1980 Proceedings of the National Computer Conference, 1980

[12]. «The Detection of Faulty Code Violating Implicit Coding Rules», Tomoko Matsumura, Akito Monden, Ken-ichi Matsumoto, isese, pp.173, 2002 International Symposium on Empirical Software Engineering (ISESE'02), 2002
[13]. «A Framework for Source Code Search Using Program Patterns», S. Paul, A. Prakash, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 463-475, June 1994
[14]. Doxygen. Средство автоматической генерации документации. http://www.doxygen.org
[15]. «LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation», Chris Lattner, Vikram Adve, cgo, pp.75, International Symposium on Code Generation and Optimization (CGO'04), 2004
[16]. CLANG: C, Objective C, C++ frontend for LLVM, Apple, Inc, http://clang.llvm.org/
[17]. «Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ.» А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Труды Института системного программирования РАН, том 21, 2011 г, стр. 23-38
[18]. «Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок.» А. Аветисян, А. Бородин. Труды Института системного программирования РАН, том 21, 2011 г, Стр. 39-54
[19]. «The Definitive Guide to SQLite», Allen, Grant; Owens, Mike (November 5, 2010). (2nd ed.). Apress. p. 368. ISBN 1430232250.
[20]. Инфраструктура для кросс-компиляции. http://www.scratchbox.org/
[21]. Standard – the C++ language. Report ISO/IEC 14882: 1998, Information Technology Council (NCTIS).
[22]. «О некоторых задачах анализа и трансформации программ.» С. Гайсарян, А. Чернов, А. Белеванцев, О. Маликов, Д. Мельник, А. Меньшикова. Труды Института системного программирования РАН, том 5, 2004 г. Стр. 7-40.

Static analysis usage for customizable checks of programming languages semantic constraints

Valery Ignatyev <valery.ignatyev@ispras.ru>
ISP RAS, Moscow, Russia

Abstract. We describe the usage of programming language constraints to achieve program security and portability, which are especially important for large projects. The existing collections of such constraints (like MISRA C++, JSF, or HICPP) contain only descriptive natural language rule definitions, which could be possibly ambiguous or incomplete.

We propose the formal model for constraints definition, which allows specifying stylistic, syntax and contextual rules. We also give the constraints classification, which splits all constraints into 4 disjoint groups determining the constraint complexity and optimal checking order. To solve the problem of automatic rules checking, we have developed the analyzer based on the C/C++ Clang compiler maintained as a part of the LLVM project. We also describe some specific details of analyzer implementation: basic components overview, the set of specially developed and well-known static analysis algorithms used for constraint violation discovering, rule ordering approach, permanent external storage (SQLite database) usage for intermodule analysis and work with error messages (sorting, history). We also provide analyzer integration with popular build systems so that the source files used in the build process are analyzed automatically.

The implemented system is able to check approximately 50 different C and C++ constraints and requires only 20% more time than the regular optimized build.

Keywords: static analysis, coding standards, security vulnerability, Clang, LLVM.

References

[1]. «Guidelines for the use of the C language in critical systems», MISRA-C:2004, 2004
[2]. «Guidelines for the Use of the C++ Language in Critical Systems», ISBN 978-906400-03-3 (paperback), June 2008.
[3]. «Joint Strike Fighter air vehicle C++ coding standards for the system development and demonstration program», Lockheed Martin Corporation, 2005
[4]. «High Integrity C++ Coding Standard Manual - Version 2.4», The programming research group, 2007
[5]. The annotated ANSI C Standard American National Standard for Programming Languages. ANSI/ISO 9899-1990
[6]. «Code Complete», S. McConnel, Microsoft Press; 1 edition (January 1, 1993)
[7]. «The Practice of Programming», Brian W. Kernighan, Rob Pike, Addison-Wesley Professional, 1999
[8]. «Coding standards for high-confidence embedded systems», Paul Anderson, MILCOM2008
[9]. «A Coding Rule Conformance Checker Integrated into GCC», Marpons, Marino, Carro, Herranz, Fredlund, Moreno-Navarro, and Polo, Electronic Notes in Theoretical Computer Science, 2007

[10]. «A tool for checking coding standards for C++», S. Mohammed Saleem (Master of technology thesis), 1999
[11]. «Standard error classification to support software reliability assessment», John B. Bowen, afips, pp.697, 1980 Proceedings of the National Computer Conference, 1980
[12]. «The Detection of Faulty Code Violating Implicit Coding Rules», Tomoko Matsumura, Akito Monden, Ken-ichi Matsumoto, isese, pp.173, 2002 (ISESE'02), 2002
[13]. «A Framework for Source Code Search Using Program Patterns», S. Paul, A. Prakash, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 463-475, June 1994
[14]. Doxygen. Documentation generator. http://www.doxygen.org
[15]. «LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation», Chris Lattner, Vikram Adve, pp.75, International Symposium on Code Generation and Optimization (CGO'04), 2004
[16]. CLANG: C, Objective C, C++ frontend for LLVM, Apple, Inc, http://clang.llvm.org/
[17]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm. [Using static analysis for finding security vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 23-38 (in Russian).
[18]. A. Avetisyan, A. Borodin. Mekhanizmy rasshireniya sistemy staticheskogo analiza Svace detektorami novykh vidov uyazvimostej i kriticheskikh oshibok. [Svace static analysis system extension with new types of vulnerabilities and critical errors detectors] Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 39-54 (in Russian).
[19]. «The Definitive Guide to SQLite», Allen, Grant; Owens, Mike (November 5, 2010). (2nd ed.). Apress. p. 368. ISBN 1430232250.
[20]. Scratchbox: cross-compiler infrastructure http://www.scratchbox.org/
[21]. Standard – the C++ language. Report ISO/IEC 14882: 1998, Information Technology Council (NCTIS).
[22]. S.Gaissaryan, A.Chernov, A.Belevantsev, O.Malikov, D.Melnik, A.Menshikova. O nekotorykh zadachakh analiza i transformatsii programm.[Some problems in program analysis and transformation] Trudy ISP RAN [The Proceedings of ISP RAS], 204, vol. 5, pp. 7-40 (in Russian).

Перспективы виртуализации высокопроизводительных систем архитектуры x64

А. О. Кудрявцев, alexk@ispras.ru, В. К. Кошелев, vedun@ispras.ru
А. И. Аветисян, arut@ispras.ru

Аннотация. В данной работе изучаются перспективы применения технологий виртуализации в области высокопроизводительных вычислений на платформе x64. Рассматриваются основные причины падения производительности при запуске параллельных программ в виртуальной среде. Подробно рассматриваются системы виртуализации KVM/QEMU и Palacios, в качестве тестовых пакетов используются HPC Challenge и NAS Parallel Benchmarks. Тестирование выполняется на современном вычислительном кластере, построенном на базе высокоскоростной сети Infiniband. Результаты проведенного исследования в целом показывают целесообразность применения виртуализации для большого класса высокопроизводительных приложений. Доводка рассматриваемых систем виртуализации позволила снизить накладные расходы с 10-60% до 1-5% на большинстве тестов пакетов HPC Challenge и NAS Parallel Benchmarks. Основными “узкими местами” систем виртуализации являются уменьшенная производительность системы памяти (критично только для узкого класса задач), расходы при виртуализации устройств, а также повышенный уровень “шума”, источником которого становятся основная ОС и гипервизор. Шум может оказывать негативное влияние на производительность и масштабируемость “мелкозернистых” приложений (приложений с частыми коммуникациями небольшого объема). При увеличении числа узлов в системе, влияние шума существенно усиливается.

Ключевые слова: высокопроизводительные системы; виртуализация; мониторы виртуальных машин; высокопроизводительные вычисления; параллельные вычисления

1. Введение

Возможности технологий виртуализации платформы x86_64 в последние годы растут высокими темпами. Технологии аппаратной и контейнерной виртуализации уже позволяют обеспечить производительность отдельных классов приложений в виртуальных средах, практически не отличимую от производительности на реальном оборудовании. В результате исследователи по всему миру начали изучать возможности и ограничения виртуализации высокопроизводительных вычислений (HPC, High Performance Computing).

Достоинства применения виртуализации в области высокопроизводительных вычислений широко обсуждаются [1], [2]. Среди них устойчивость к сбоям, совместимость и гибкость при работе с виртуальными машинами (ВМ). Также весьма интересна идея применения концепции облачных вычислений для создания высокопроизводительных облачных систем, предоставляющих масштабируемость, эффективность использования ресурсов и удобство доступа. Последние исследования показывают, что виртуализация HPC имеет смысл по крайней мере для отдельных классов приложений [1]. Несмотря на это, в настоящее время существует серьезная нехватка экспериментальных данных. Необходимо исследовать поведение различных приложений при запуске на различном оборудовании с использованием ряда систем виртуализации, для того, чтобы достоверно оценить ограничения и возможности существующих технологий. Современные многопроцессорные, многоядерные системы предъявляют новые требования к системам виртуализации, включая корректную эмуляцию архитектуры NUMA (Non-Uniform Memory Access, архитектура с неравномерным доступом к памяти) в гостевой системе.

Одной из целей данной работы было достичь максимально возможной производительности задач, запущенных в виртуальной среде, по сравнению с запуском на реальном оборудовании. Основной целью работы является оценка накладных расходов виртуализации в целом, а также выявление причин и минимизация этих расходов. Исследования проводятся с использованием системы виртуализации KVM (Kernel-based Virtual Machine) [3] и гипервизора Palacios [4], который изначально разрабатывался специально для высокопроизводительных систем. Необходимо отметить, что в ходе данной работы использовалась специально модифицированная версия гипервизора Palacios, поскольку оригинальная версия недоработана и не может быть запущена на используемом тестовом оборудовании. Сделанные изменения будут кратко описаны в тексте статьи.

Для достижения наилучшей производительности (относительно случая запуска на реальном оборудовании), виртуальным машинам выделяется максимально возможное количество ресурсов, включая все процессорные ядра. Помимо этого, виртуальным машинам предоставляется реальное коммуникационное устройство, с использованием технологий виртуализации ввода-вывода Intel VT-d в случае KVM и паравиртуализации в случае Palacios. Также в ВМ эмулируется архитектура NUMA в соответствии с реальной конфигурацией узлов кластера.

В качестве тестовых пакетов используются пакеты HPC Challenge [5] и NAS Parallel Benchmarks [6]. Данные бенчмарки достаточно широко распространены в области высокопроизводительных вычислений, а пакет HPCC используется для построения рейтинга суперкомпьютеров Top500. Тестирование производительности выполнялось на современном кластере, с использованием до 8 узлов, построенных на базе процессоров Intel Xeon (в

сумме до 96 процессорных ядер). Узлы кластера связаны высокоскоростной сетью Infiniband 40Гбит/сек. Стоит отметить, что многие исследования в области виртуализации HPC, представленные ранее, проводились на системах из одного узла, что не позволяло полноценно оценить накладные расходы, вызываемые виртуализацией.

В ходе выполнения данной работы достигнуты следующие результаты:

- Описаны базовые, но эффективные методы настройки KVM/QEMU для запуска HPC-приложений в виртуальной среде. Показана необходимость корректной эмуляции архитектуры NUMA в виртуальной среде в соответствии с реальной топологией системы.
- Проведена оценка накладных расходов виртуализации для современного HPC кластера, построенного на базе высокоскоростной сети Infiniband. Узлы кластера – двухпроцессорные 12-ядерные серверы архитектуры NUMA, используется до 8 узлов.
- Выполнено сравнение гипервизора Palacios, встроенного в основную ОС Kitten, и системы виртуализации KVM/QEMU, с основной ОС Linux. Полученные данные показывают, что KVM/QEMU дает более стабильные и предсказуемые результаты, в то время как Palacios имеет меньшие накладные расходы на “мелкозернистых” тестах, особенно при использовании большого числа вычислительных ядер.
- Проведено исследование полученных тестовых данных и накладных расходов, вызванных системой виртуализации. В частности, исследовано влияние гранулярности коммуникаций HPC-приложения и частоты прерываний на величину накладных расходов.

Текст данной работы организован следующим образом. В следующем разделе приводится обзор работ. В разделе 3 описаны исследуемые системы виртуализации и некоторые базовые методы по улучшению производительности ВМ. В разделе 4 описывается методика оценки производительности, приводятся результаты тестов и их анализ. Раздел 5 содержит заключение.

2. Обзор работ

Развитие технологий виртуализации открывает широкие перспективы в различных областях исследований. В частности, аппаратная виртуализация достаточно часто рассматривается в контексте информационной безопасности [7], а системы динамической бинарной трансляции нашли применение в области отладки. На базе симулятора QEMU, в ИСП РАН была реализована система детерминированного воспроизведения процесса выполнения программ в ВМ [8], позволяющая отлаживать трудновоспроизводимые ошибки как в приложениях, так и в коде ОС с использованием методов реверсивной отладки. В последнее время исследователи все чаще рассматривают перспективы применения виртуализации в области высокопроизводительных вычислений.

Подробный анализ существующих работ в области виртуализации HPC проведен в работе [1]. Авторы отмечают, что результаты экспериментов, доступные из статей, зачастую противоречат друг другу, и ставят задачу объективного сравнения существующих систем виртуализации, включая KVM, Xen [9], VirtualBox [10]. Для сравнения используются такие критерии как доступная функциональность, удобство использования и производительность. Авторы использовали тестовые пакеты HPCC и SPEC OpenMP; для теста HPL пакета HPCC накладные расходы виртуализации составили около 30% на 8-ядерной системе. Падение производительности в остальных тестах составило до 50% по сравнению с запуском на реальном оборудовании. Как показывают результаты наших исследований, такие значительные накладные расходы можно существенно сократить. Авторы статьи [1] отмечают, что производительность системы Xen нестабильна – различные запуски одного и того же приложения в одинаковых условиях дают результаты с существенным разбросом по производительности – что сильно снижает масштабируемость параллельных приложений. Наши предварительные эксперименты также показали, что гипервизор Xen не приспособлен для высокопроизводительных задач, по крайней мере, без специальной настройки. Необходимо также отметить, что авторы статьи выполняли все тесты на одной машине, что не позволило оценить падение производительности виртуальной среды при масштабировании числа узлов. Масштабируемость некоторых HPC приложений может также ухудшаться из-за дополнительного шума, вызываемого системой виртуализации. Шумом в контексте выполнения конкретного приложения называют влияние некоторых действий системы, не относящихся к приложению, на его работу. Среди таких действий – работа программ-демонов, обработчиков прерываний, других программ в системе.

Авторы работы [1] приходят к следующему выводу. Виртуализация применима, по крайней мере, к некоторым HPC-приложениям, а наиболее приспособленной системой виртуализации для таких задач является KVM/QEMU. С нашей точки зрения, система KVM становится одной из наиболее развитых систем виртуализации с открытым исходным кодом. По сравнению с Xen, KVM намного проще для установки и понимания, поскольку является частью исходного кода ядра Linux, а гипервизор Xen представлен в виде патчей. В целом, гипервизор KVM кажется более перспективным, в связи с чем в наших исследованиях было решено использовать именно его. При этом, нельзя утверждать, что производительность системы Xen хуже, чем KVM. Проблема в том, что каждая система виртуализации вместе с основной ОС должны быть тщательно настроены для запуска высокопроизводительных задач, однако для Xen такой настройки не проводилось. Также одной из причин исследования именно системы KVM/QEMU является наличие опыта оптимизации симулятора QEMU в ИСП РАН [11].

В работе [12] проводится сравнительный анализ производительности ввода-вывода систем виртуализации KVM, Xen, OpenVZ [13] и облачного сервиса фирмы Amazon “EC2 Cluster Compute Node”. Авторы отмечают, что вопросы виртуализации процессора в настоящее время тщательно изучены, и различные подходы к реализации гипервизоров позволяют достичь производительности, близкой к производительности реального процессора при запуске приложения, требовательного к вычислительной мощности. С другой стороны, накладные расходы виртуализации ввода-вывода изучены недостаточно подробно, и в работе авторы оценивают производительность дискового и сетевого ввода-вывода. Впервые оценивается эффективность работы адаптера Infiniband при его пробросе внутрь ВМ с использованием технологии Intel VT-d. Результаты пакета NAS Parallel Benchmarks полученные при запуске на системе из 4-х узлов на базе процессоров Intel Xeon E5520, до 32 процессорных ядер в сумме, показывают ухудшение производительности в худшем случае до 30 раз для KVM и до 50% для Xen, по сравнению с запуском на реальном оборудовании. Возможно, авторы использовали устаревшую версию KVM/QEMU, или была допущена ошибка в методике тестирования, поскольку результаты, полученные в нашей работе, намного более оптимистичны.

Производительность системы OpenVZ не тестировалась в рассматриваемой работе, поскольку в данной системе отсутствует поддержка технологии Infiniband в контейнерах, однако авторы утверждают, что OpenVZ дает производительность ввода-вывода, идентичную случаю без виртуализации. В целом, это утверждение правдоподобно, поскольку виртуализация уровня ОС (контейнерная виртуализация) требует намного меньше дополнительных ресурсов, чем полная виртуализация. С другой стороны, технологии аппаратной виртуализации продолжают развиваться, и с нашей точки зрения в ближайшее время производительность подсистемы ввода-вывода также улучшится.

Как утверждают авторы работы [4], основной причиной падения производительности устройства ввода-вывода, проброшенного внутрь ВМ, являются накладные расходы на обработку прерываний устройства. При каждом прерывании процессор должен передать управление от виртуальной машины гипервизору. Далее, гипервизор передает управление основной ОС, которая, в свою очередь, определяет, что прерывание принадлежит гипервизору. После этого, гипервизор производит “вброс” прерывания в гостевую систему. После окончания обработки прерывания гостевой системой, сигнал окончания обработки, подаваемый контроллеру прерываний, также обрабатывается гипервизором. Эта цепочка вызовов может замедлить доставку прерываний и серьезно увеличить задержку коммуникаций. В работе [14] данное предположение подтверждается. Авторы предложили систему ELI (ExitLess Interrupts), которая позволяет обрабатывать часть прерываний в гостевой системе без необходимости выходов в гипервизор и основную ОС. Результаты тестов показали, что система ELI может уменьшить накладные

расходы KVM/QEMU с 40% до 1-2% при работе ВМ на одном вычислительном ядре и использовании Netperf, Apache и Memcached в качестве тестовых приложений. В данном случае ухудшение производительности было связано с большим числом прерываний, генерируемых проброшенной внутрь ВМ сетевой картой Ethernet 10Гбит/сек. С использованием ELI, число выходов из ВМ в гипервизор сократилось примерно в 100 раз.

В работе [15] показано, что при запуске тестов NAS Parallel Benchmarks в виртуальных машинах, выполняющихся на системах архитектуры NUMA, производительность снижается в среднем на 50% по сравнению с запуском на реальном оборудовании. Авторы отмечают, что поддержка NUMA в существующих системах виртуализации не полностью реализована, вследствие чего не удается достигнуть локальности данных в гостевой системе. Более того, авторы считают, что даже реализация полной поддержки NUMA в ВМ не сильно поможет при работе нескольких ВМ на одном сервере. Для решения проблемы предлагается использовать паравиртуализацию путем добавления нового бит-транспортного протокола “bypass” в реализацию библиотеки OpenMPI. Этот протокол позволяет осуществлять передачу данных между ВМ с использованием общей памяти, доступной всем участвующим в обмене ВМ. При запуске по одной ВМ на процессорном сокете, авторы добились снижения накладных расходов до 5-7%. В нашей работе тесты выполнялись на 2-х процессорных 12-ядерных серверах архитектуры NUMA, однако, накладные расходы в целом меньше 50%. В случае виртуализации HPC-систем, гипервизор может эмулировать топологию NUMA в соответствии с реальной топологией сервера, что позволит избежать описанных в статье проблем, поскольку гостевая ОС будет осуществлять управление памятью с учетом архитектуры NUMA.

Исследования по виртуализации суперкомпьютеров выполнены в работе [4]. Использовался специально разработанный гипервизор Palacios, встроенный в основную ОС Kitten [16], предназначенную для работы на узлах суперкомпьютера. Гипервизор Palacios разрабатывается в рамках проекта V3VEE [17]. Авторы получили результаты для ряда HPC-приложений и тестовых пакетов, накладные расходы виртуализации сохраняются на уровне не более 5% с увеличением числа узлов до 4096. Однако необходимо отметить, что данные результаты были получены при запуске по одной 1-ядерной ВМ на каждом 4-х ядерном узле суперкомпьютера. Как показывает наше исследование, современные многопроцессорные системы требуют соответствующей поддержки со стороны гипервизора, включая корректную эмуляцию архитектуры NUMA. Авторы рассматриваемой статьи также исследовали различные методы страничной адресации, включая теневую страничную адресацию (shadow paging) и вложенную страничную адресацию (nested paging). Вложенная адресация дает лучшие результаты для гостевых систем с частым переключением таблиц страниц, например для ОС Linux. В

нашей работе мы также использовали вложенную адресацию, поскольку в качестве гостевой ОС используется Linux.

Авторы гипервизора Palacios также отмечают проблему влияния шума ОС на работу параллельных приложений. Влияние шума широко изучается [18], [19]. В целом, влияние шума сильно зависит от ряда факторов, в том числе от соотношения объема вычислений и коммуникаций приложения, гранулярности коммуникаций (размер и частота посылок), характеристик сетки процессов. Для ряда приложений, даже минимальный уровень шума может привести к значительному ухудшению производительности и масштабируемости. Таким образом, при виртуализации высокопроизводительной системы, особенно с большим числом узлов, необходимо учитывать дополнительный шум от гипервизора и основной ОС. С этой точки зрения, ОС Kitten имеет преимущество перед Linux, как основная ОС, так как данная ОС специально разработана для запуска HPC-приложений на системах с большим числом узлов.

3. Рассматриваемые системы виртуализации

Наиболее распространенные системы виртуализации в целом обладают одинаковыми возможностями. Такие системы используют доступные на сервере технологии аппаратной виртуализации. Среди систем с открытым исходным кодом, наиболее популярны KVM и Xen. Как уже отмечалось ранее, в данной работе исследуется гипервизор KVM, поскольку, на наш взгляд, KVM более перспективен и весьма удобен в использовании. Для установки и исследования Xen требуется больше времени, так как существует несколько различных версий данного гипервизора. Также рассматривается гипервизор Palacios, который специально разработан для исследования архитектуры компьютеров и использования в области высокопроизводительных вычислений.

3.1. Система виртуализации KVM/QEMU

Гипервизор KVM состоит из нескольких модулей ядра Linux и управляется посредством файла устройства, доступ к которому осуществляется с использованием системного вызова ioctl. KVM реализует виртуальный процессор и систему памяти, остальные компоненты виртуальной машины реализованы эмулятором QEMU. QEMU управляет ресурсами и предоставляет виртуальные устройства. С помощью QEMU можно запускать ВМ с использованием технологий аппаратной виртуализации (для этого используется KVM) либо бинарной трансляции (генератор кода встроен в QEMU).

KVM/QEMU вместе с основной ОС Linux позволяют назначать реальные устройства отдельным ВМ (пробрасывать устройства внутрь ВМ). Для данной возможности необходимо наличие специальной аппаратуры – устройства управления памятью ввода-вывода (IOMMU, IO Memory Management Unit). В

случае платформы производства Intel технология называется Intel VT-d, у фирмы AMD – AMD IOMMU. Основной задачей IOMMU является отображение адресного пространства пробрасываемого устройства в адресное пространство физической памяти гостевой системы. При отсутствии такого устройства, гостевая система должна быть осведомлена о наличии гипервизора и положении гостевой физической памяти в реальной физической памяти, что позволит выполнять запросы на прямой доступ к памяти (DMA, Direct Memory Access) корректно.

Для возможности использования высокоскоростной сети в виртуальной машине, коммуникационный адаптер Infiniband был проброшен внутрь ВМ с использованием Intel VT-d. Необходимо отметить, что назначенные ВМ устройства могут иметь более низкую производительность, чем если бы они работали в основной ОС. Как было описано выше, это может быть вызвано накладными расходами по обработке прерываний. Более подробно данная проблема рассматривается в разделе 4.

3.2. Использование больших страниц

Система виртуальной памяти архитектуры x86_64 поддерживает несколько размеров страниц при страничной адресации, среди них 4Кб, 2Мб и 1Гб страницы. Ядро Linux по умолчанию использует 4Кб страницы, страницы большего размера могут быть задействованы с использованием системы HugeTLB. С точки зрения виртуализации памяти, большие страницы могут быть использованы для уменьшения числа доступов к памяти, необходимого для трансляции гостевого физического адреса в реальный физический адрес, а также для уменьшения числа промахов TLB (Translation Lookaside Buffer, буфер трансляции страниц).

Ядро Linux предоставляет два способа использования HugeTLB: Transparent Hugepages и HugeTLBfs. Механизм Transparent Hugepages позволяет ядру выделять память большими страницами автоматически, если запрошенный объем памяти кратен размеру большой страницы, при этом приложение не требуется модифицировать. В отличие от Transparent Hugepages, механизм HugeTLBfs использует заранее зарезервированные группы больших страниц. HugeTLBfs представляет собой специальную файловую систему, для хранения файлов в которой используются большие страницы. Программа может отобразить такие файлы в память (используя системный вызов mmap) и для этого отображения ядро Linux будет использовать большие страницы.

Используемая нами версия QEMU (1.0) выделяет память для гостевой системы посредством вызова posix_memalign с 2Мб выравниванием, поэтому механизм Transparent Hugepages должен работать по умолчанию.

3.3. Предоставление ВМ реальной топологии NUMA

Как показывают результаты тестов, запуск виртуальной SMP системы на реальной системе архитектуры NUMA ведет к существенным накладным

расходам. Каждый узел в топологии NUMA имеет свой набор процессорных ядер и участков физической памяти, и доступ процессора одного узла к памяти другого узла требует существенно больше тактов, чем доступ к локальной памяти узла. Эмулятор QEMU имеет поддержку архитектуры NUMA в ВМ, однако эмулируемая топология никак не связана с реальной топологией системы. Для решения этой проблемы, были внесены изменения в исходный код QEMU. Прежде всего, была осуществлена "привязка" виртуальных процессоров (ядер) к отдельным физическим ядрам процессора, что позволило предотвратить перемещение нитей QEMU между ядрами процессоров. Далее, был использован системный вызов mbind для выделения отрезков памяти гостевой системы на соответствующих узлах реального сервера. Было также выявлено, что вызов mbind препятствует работе механизма Transparent Hugepages, поэтому память выделяется посредством HugeTLBfs с использованием ключа командной строки QEMU -mempath. Механизм HugeTLBfs имеет некоторые проблемы с безопасностью и масштабируемостью [20], однако в данном случае этим можно пренебречь.

3.4. Гипервизор Palacios

Гипервизор Palacios разрабатывался с целью эффективной виртуализации высокопроизводительных систем. Версия 1.0 вышла в 2008 году, последняя версия 1.3 вышла в декабре 2011 года. Отличительной особенностью Palacios является его встраиваемость, реализованная путем выделения набора унифицированных интерфейсов с основной ОС. Код Palacios может быть встроен в различные системы путем реализации этих интерфейсов. Изначально гипервизор Palacios был встроен в специализированную ОС Kitten, в последней версии 1.3 также реализован модуль для встраивания в Linux. Другой отличительной особенностью Palacios является его конфигурируемость, то есть возможность на этапе компиляции включить или исключить различные компоненты гипервизора. Эта особенность позволяет легко создавать версии гипервизора, специально "подогнанные" под конкретные нужды. На наш взгляд, система Palacios является весьма перспективной. Основная ОС Kitten дает важное преимущество – эта ОС генерирует намного меньше шума по сравнению с Linux, и это преимущество особенно интересно с точки зрения проводимых исследований. Часть кода ОС Kitten заимствована из ядра Linux, однако критические с точки зрения производительности подсистемы разработаны с учетом особенностей высокопроизводительных систем.

Palacios поддерживает вложенную страничную адресацию и несколько методов теневой адресации. Для экспериментов использовалась вложенная адресация, так как данный метод лучше подходит для гостевой ОС Linux [4].

3.5. Метод проброса устройств

Авторы Palacios отмечают необходимость проброса реального коммуникационного устройства внутрь ВМ для достижения сопоставимого с

реальным оборудованием уровня производительности. Для проброса устройств используется специальный паравиртуальный интерфейс (все взаимодействие между гипервизором и гостевой системой в рамках Palacios называется “Симбиотическая виртуализация”). Память для гостевой системы выделяется единым непрерывным отрезком, после чего гостевая система запрашивает у гипервизора смещение памяти ВМ в реальной физической памяти. С использованием этого смещения, гостевая ОС может правильно формировать DMA-запросы для проброшенных в ВМ устройств путем прибавления смещения к целевому адресу. Изменения гостевой ОС, необходимые для поддержки данного механизма, минимальны. Был написан патч для ядра Linux версии 2.6.18, который позже был портирован на более новые версии ядра.

Поскольку ОС Kitten практически не содержит драйверов устройств, помимо адаптера Infiniband внутрь ВМ были также проброшены SATA-контроллер и сетевая карта Ethernet.

3.6. Возникшие трудности с гипервизором Palacios

Во время экспериментов с гипервизором Palacios был обнаружен ряд трудностей, вызванных “незрелостью” системы. Трудности включали в себя неполную поддержку проброса устройств (элемент Capabilities конфигурационного пространства PCI был недоступен в ВМ, из-за чего были недоступны прерывания типов MSI и MSI-X), неполная реализация поддержки технологии Intel VT-x, максимальный объем памяти ВМ примерно 3.5Гб, отсутствие поддержки архитектуры NUMA в ВМ и ряд ошибок в исходном коде. Была создана локальная ветка репозитория Palacios, в которой была реализована недостающая функциональность. Часть созданного кода была передана разработчикам Palacios в виде патчей, некоторые из которых доступны на официальном сайте проекта V3VEE. Часть патчей включены в основную ветку разработки Palacios.

Одна серьезная проблема совсем не была решена –проблема с реализацией системных таймеров в ВМ. Время в ВМ гипервизора Palacios идет неточно, за сутки достигается сдвиг часов примерно на 30-40 минут. Для того чтобы контролировать этот эффект во время запуска тестов, использовалась синхронизация по протоколу NTP и проверялось значение смещения времени.

4. Оценка и анализ производительности

За счет использования двух популярных тестовых пакетов HPC Challenge и NAS Parallel Benchmarks, оценивается достаточно широкий класс HPC-приложений. В следующих подразделах описываются экспериментальная среда, методология тестирования и результаты. В конце раздела анализируются причины накладных расходов в различных случаях.

4.1. Экспериментальная среда

В качестве стенда использовался кластер фирмы HP. Кластер состоит из 8 блейдов HP ProLiant BL2x220c G7, каждый блейд комбинирует в себе по два вычислительных узла. Для тестов использовалось до 8 узлов одновременно. Каждый узел содержит по 2 процессора Intel Xeon X5670 с частотой 2.93 Ггц (по 6 ядер на процессор), а также 24Гб ОЗУ. Технология Hyperthreading деактивирована на всех узлах. Узлы взаимодействуют посредством сервисной сети Ethernet 1Гбит/сек. и вычислительной сети Infiniband 40Гбит/сек.

Для тестов на реальном оборудовании и в качестве гостевой ОС использовалась система Linux CentOS 6.0 с ядром версии 2.6.32-71.29.1.el6.x86_64. Ядро ОС модифицировано для поддержки проброса устройств Palacios. Версия библиотеки MPI – OpenMPI 1.4.3, компилятора GCC – 4.4.4. ПО Infiniband – Mellanox OFED 1.5.3-1.0.0-rhel6-x86_64.

Основная ОС для KVM/QEMU – Linux CentOS 6.2, версия ядра 2.6.32-220.2.1.el6.x86_64. Версия QEMU – qemu-kvm-1.0 с модификациями, описанными ранее. Версия Palacios – 1.2 с изменениями, описанными выше. Версия ОС Kitten – 1.2.0 с изменениями, необходимыми для корректного проброса устройств в ВМ.

Гостевая система сконфигурирована с 16 Гб ОЗУ, 12-ю процессорными ядрами, архитектуры SMP или NUMA. Контроллер Infiniband проброшен в ВМ, как описано ранее. Для Palacios также пробрасывается сетевая карта и SATA-контроллер.

4.1.1.Бенчмарки

Для оценки производительности используются тестовые пакеты HPC Challenge и NAS Parallel Benchmarks. Каждый тестовый пакет выполнялся на 2, 4 и 8 узлах.

Версия NPB – 3.3.1 MPI. Из данного пакета используются тесты IS, EP, CG, MG, FT, BT и LU, класс задачи C. Для тестов IS, CG, FT, MG запускается по 8 процессов на узел (число процессов должно быть степенью двойки). Для EP и LU – по 12 процессов на узел, для BT – по 8, 9 и 8 процессов на узел соответственно (суммарное число процессов должно быть квадратом). Также тесты NPB выполнялись на одном узле, по 9 процессов для BT, по 12 для EP и LU, по 8 для остальных тестов. Результаты тестов NPB приведены к унифицированному виду и выражаются в секундах.

В тесте IS (Integer Sorting) выполняется цифровая сортировка, большой объем коммуникаций и случайных доступов к памяти. EP (Embarrassingly Parallel) – генератор значений Гауссовых случайных величин. CG – метод сопряженных градиентов нахождения локального минимума, с нерегулярным характером доступов к памяти и коммуникаций. В тесте MG аппроксимируется решение дискретного уравнения Пуассона, тест характеризуется интенсивной работой с памятью и коммуникациями. FT – дискретное преобразование Фурье, использует коммуникации типа “all-to-all”. Тесты BT и LU – решатели систем

дифференциальных уравнений в частных производных с использованием различных методов.

Используемая версия пакета HPCC – 1.4.1, библиотеки ATLAS – 3.8.4. Тесты пакета HPCC выполняются с запуском по 12 процессов на узел. Оцениваются результаты тестов STREAM, RandomAccess, HPL. Тест PTRANS выполняется слишком быстро (0.1-0.5 секунды) и принадлежит к тому же классу задач, что и STREAM. Также из рассмотрения исключены тесты DGEMM, Bandwidth/Latency, FFT из-за ограничений по объему материала. Исследуемые тесты пакета HPCC имеют следующие характеристики:

- STREAM. Оценка устойчивой пропускной способности памяти для векторной операции (копирование, умножение на константу, сложение, комбинация), в данной работе исследовались результаты комбинированной операции (в Гб/сек.). Оцениваются результаты для версий Single (выполняется ровно один процесс) и EP (Embarrassingly Parallel) – все доступные ядра выполняют одну и ту же задачу независимо. Запуск версий Single и EP осуществляется только на одном узле.
- RandomAccess. Оценка скорости обновлений памяти по случайным адресам в миллиардах обновлений в секунду (GUP/s). Известно, что данный тест работает в виртуальной среде достаточно плохо из-за высокой нагрузки на буфер трансляции (TLB). Используются версии Single, EP и MPI.
- HPL (High Performance Linpack). Решение системы линейных уравнений, результат – скорость решения в GFlops. Параметры теста: размер матрицы – 30000, размер блока 150, вычислительная сетка в виде квадрата, когда это возможно.

Известно, что время в виртуальных машинах идет неточно. Гипервизор и основная ОС являются дополнительными источниками шума и вызывают неточности в таймерах гостевой ОС. В результате, время выполнения одного теста при многократных запусках в ВМ имеет больший разброс по сравнению с запусками на реальном оборудовании. Для того чтобы результаты проводимых экспериментов были более точными, большая часть запусков NPB выполнялась по 50 раз, HPCC – по 20 раз. В качестве результата серии запусков, вычисляется оценочное математическое ожидание (среднее арифметическое) и доверительный интервал с вероятностью 97%, вычисленный с использованием распределения Стьюдента. Доверительный интервал позволяет оценивать достоверность полученных данных. Хотя последовательные запуски тестов не могут считаться независимыми экспериментами, доверительный интервал может помочь оценить влияние различных источников ошибки в измерении времени и удостовериться, что выполнено достаточное число запусков.

4.2. Результаты тестов

Первоначальные тесты проводились с конфигурацией QEMU “по умолчанию”, по 12 виртуальных ядер на ВМ. Первичные результаты для пакета NPB представлены на рис. 1. “KVM” обозначает конфигурацию “по умолчанию”, тонкие черные столбцы – доверительный интервал для соответствующих средних значений. Данные изображены относительно случая Native (запуск на реальном оборудовании), которому соответствует среднее значение 0 в каждой группе столбцов. Подпись под каждой группой столбцов состоит из названия теста и числа процессов, на котором выполнялся тест. Как показывает рисунок, в конфигурации “KVM” накладные расходы изменяются от 2-4% на тесте EP до 32% на тесте MG, 64 процесса.

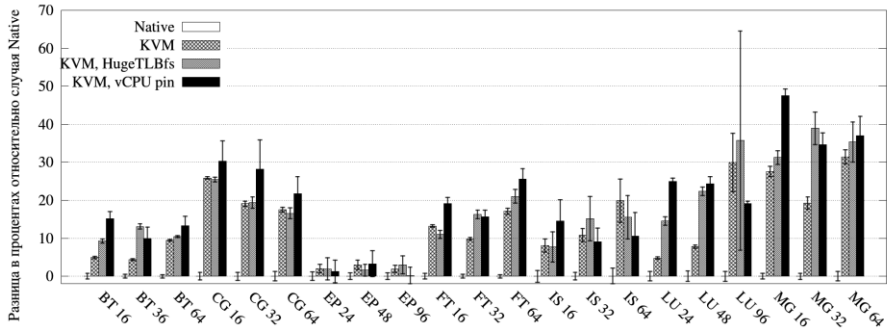


Рис. 1. Первичные результаты KVM/QEMU на тестах пакета NPB.

Далее было предположено, что миграция нитей QEMU между ядрами процессоров может негативно сказываться на производительности. Чтобы предотвратить миграцию нитей, была выполнена привязка этих нитей к соответствующим ядрам процессоров, по одному виртуальному ядру на реальное. Также было проверено, что механизм HugeTLBfs дает такие же накладные расходы, как и Transparent Hugepages. Результаты для описанных модификаций представлены также на рис. 1, столбцы “KVM, vCPU pin”, “KVM, HugeTLBfs”. Данные две группы тестов были выполнены только по 10 раз из-за нехватки времени. Как показывает рисунок, после изменений накладные расходы остались такими же и даже увеличились в ряде случаев. Случай привязки нитей дает более нестабильные результаты для тестов BT, CG, FT. Это может быть вызвано нелокальными доступами к памяти. Также интересен случай LU, 96 процессов, где виден весьма нестабильный результат. Предположительно, влияние оказал неизвестный неучтенный фактор.

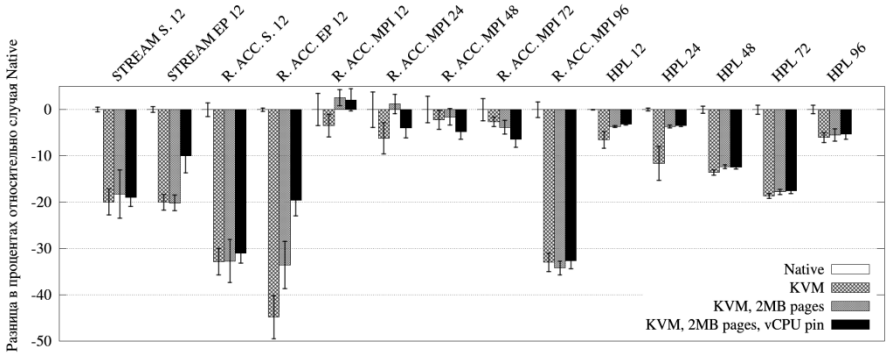


Рис. 2. Первичные результаты KVM/QEMU на тестах пакета HPCC.

Первичные результаты тестов пакета HPCC представлены на рис. 2. В данном случае, оценивались конфигурации с использованием HugeTLBfs и с привязкой нитей вместе с HugeTLBfs. Сокращение “R. ACC.” означает тест RandomAccess, “S.” – Single-версия теста. Разница в процентах в данном случае отрицательна, поскольку измеряемые величины пропорциональны производительности. Для тестов Stream версий Single, EP накладные расходы около 20% кроме случая привязки нитей, где для версии EP только 10% расходов. Тест RandomAccess показывает заметное падение производительности для версий Single и EP – от 20 до 45%, опять же, случай привязки нитей дает лучший результат для версии EP, 20% расходов. Результаты RandomAccess MPI нестабильны и расходы менее 7% при запуске до 72 процессов, однако для 96 процессов ситуация кардинально меняется – накладные расходы устойчивы на уровне 34%. Возможно, на 96 процессах узким местом становятся коммуникации, однако подробно данный вопрос не исследовался.

Наконец, было выдвинуто предположение о том, что основной причиной падения производительности является несоответствие виртуальной SMP архитектуры реальной архитектуре NUMA. По умолчанию, QEMU эмулирует SMP систему, что ведет к неучтенным гостевой ОС накладным расходам при нелокальном доступе. Следующим шагом было предоставление гостевой системе архитектуры NUMA так, чтобы топология соответствовала реальной структуре системы. Такая возможность была реализована на основе поддержки эмуляции NUMA в эмуляторе QEMU, как это описано в предыдущем разделе.

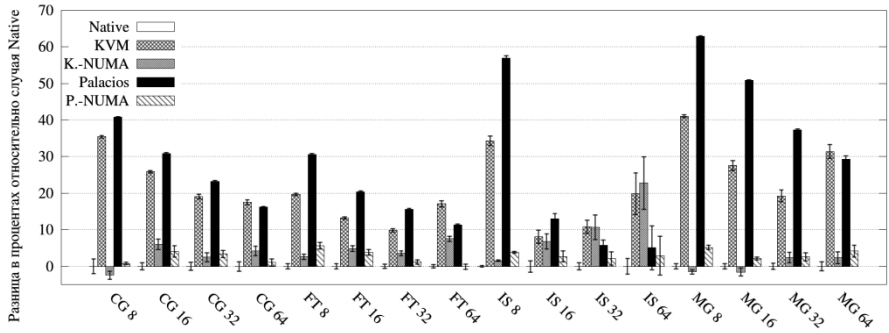


Рис. 3. Результаты KVM/QEMU и Palacios для тестов CG, FT, MG, IS пакета NPB.

Для системы Kitten + Palacios тестировались две конфигурации: ВМ архитектуры SMP и NUMA. В обоих случаях использовалась вложенная страничная адресация с 2Мб страницами. Результаты сравнения NUMA и SMP на тестах NPB для систем KVM/QEMU и Palacios представлены на рис. 3, 4, результаты тестов HPCC – на рис. 5.

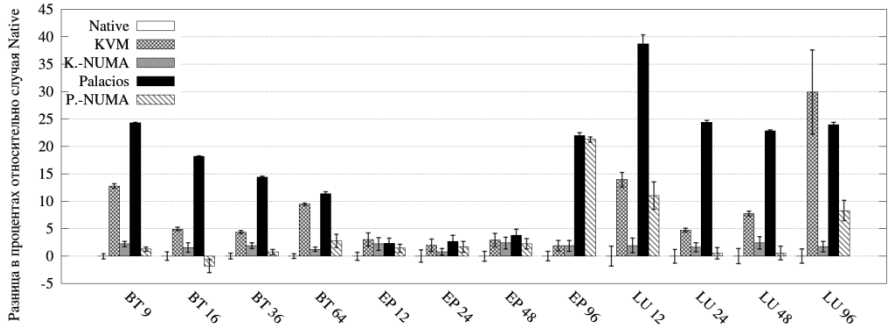


Рис. 4. Результаты KVM/QEMU и Palacios для тестов BT, EP, LU пакета NPB.

Первый достаточно очевидный вывод, который можно сделать из полученных данных – влияние корректной эмуляции NUMA на производительность весьма значительно. При запуске ВМ архитектуры SMP, Palacios и KVM показывают падение производительности более чем 60% и 40% соответственно на тесте MG 8, при эмуляции NUMA данные расходы практически исчезают. То же самое верно для большинства других тестов. Интересен также тот факт, что для многих результатов виртуальной архитектуры SMP, накладные расходы уменьшаются при увеличении числа процессов, причем разница для одного

теста может быть значительной – 30% и более, см. тесты IS и MG. В то же время, для результатов эмуляции архитектуры NUMA на многих тестах накладные расходы колеблются незначительно.

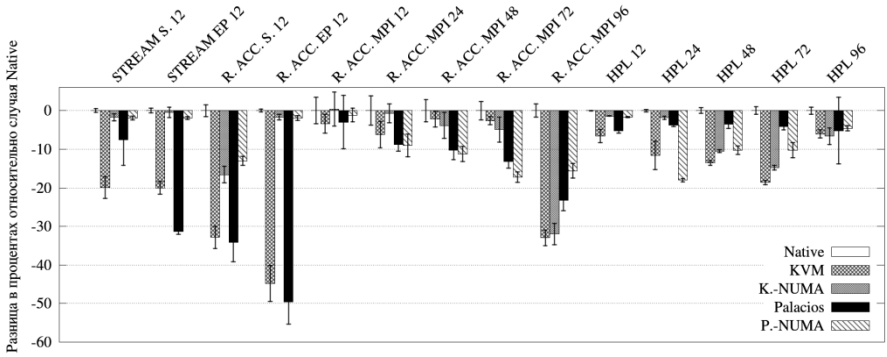


Рис. 5. Результаты KVM/QEMU и Palacios для тестов пакета HPCC.

Для тестов HPCC, эмуляция NUMA улучшает производительность на тестах STREAM, RandomAccess версий Single и EP. Для теста RandomAccess EP, эмуляция NUMA позволяет практически полностью устранить накладные расходы 45-50% случая без NUMA. Для теста RandomAccess MPI, ситуация не столь однозначна. KVM показывает лучшие результаты при запуске на 24, 48, 72 процессах, Palacios – при запуске на 96 процессах.

Можно сделать следующий вывод: реализация Palacios имеет некоторые недостатки, что показывает сравнение с результатами KVM без поддержки NUMA для практически всех тестов пакета NPB и теста RandomAccess для 24, 48, 72 процессов. Наибольшая разница между результатами двух гипервизоров около 24% на тесте LU, 12 процессов. Также Palacios дает “аномальные” результаты для теста EP на 96 процессах, учитывая тот факт, что данный тест оценивает производительность процессора. Результаты теста HPL также вызывают вопросы – на 24, 48 и 72 процессах Palacios без эмуляции NUMA показывает наилучший результат, однако с эмуляцией накладные расходы резко увеличиваются. Основная гипотеза, объясняющая разницу в производительности между KVM и Palacios в случае эмуляции архитектуры SMP, заключается в следующем. Ядро Linux имеет преимущество, автоматически управляя памятью и процессами в случае использования конфигурации QEMU “по умолчанию”, в то время как для гипервизора Palacios отображение виртуальных ядер в реальные изначально фиксировано. Как принудительное назначение ядер нитям, так и использование механизма HugeTLBfs ведут к ухудшению производительности, поскольку механизмы управления памятью и процессами ОС Linux теряют часть своих функций.

В нескольких случаях результаты в виртуальной среде оказываются лучше результатов на реальном оборудовании. Например, тесты CG 8, MG 8, MG 16, где KVM с эмуляцией NUMA дает лучшие результаты, чем в случае Native. Аналогичная ситуация с гипервизором Palacios на тесте BT 16. Вероятно, причиной такого поведения является наличие неких неучтенных устойчивых факторов, повлиявших на результаты данных тестов, однако требуется более тщательное исследование этого феномена.

Наконец, результаты тестов CG, FT, IS на 64 процессах и RandomAccess MPI на 96 процессах показывают преимущество связки Kitten + Palacios над Linux + KVM/QEMU в случае эмуляции архитектуры NUMA. Преимущество варьируется от 3.5 до 21%. Вероятно, такое поведение связано с уменьшенным шумом ОС Kitten по сравнению с Linux, особенно для теста IS. Также исследовалась гранулярность коммуникаций в тестах NPB путем оценки частоты прерываний, см. следующий подраздел.

Если сравнить результаты с архитектурой NUMA систем KVM/QEMU и Palacios и использовать порог в 3%, KVM будет лучше в 11 случаях, Palacios в 9 случаях, результаты одинаковы в пределах порога в 22 случаях. При использовании порога в 5%, KVM лучше в 8 случаях, Palacios – только в 4. В целом, KVM предоставляет более стабильные и предсказуемые результаты, в то время как Palacios выигрывает на “мелкозернистых” тестах типа IS (особенно при большом числе процессов). В то же время, Palacios имеет аномалии в производительности на некоторых тестах.

4.2.1 Влияние частоты прерываний

Согласно статье [14], виртуализация прерываний может привести к существенному падению производительности реального устройства, назначенного ВМ. Для проверки данной теории, были проведены замеры частоты прерываний коммуникационного устройства Infiniband во время выполнения тестов NPB. Результаты представлены на рис. 6. На данном рисунке сгруппированы два набора данных: красным цветом отмечено падение производительности в процентах, зеленым – среднее число прерываний в секунду (IPS, Interrupts per second) для устройства Infiniband.

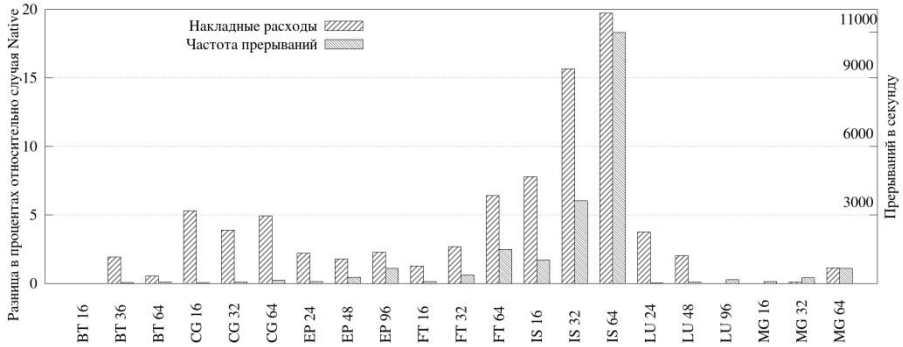


Рис. 6. Сравнение накладных расходов на тестах пакета NPB и частоты прерываний для случая эмуляции архитектуры NUMA, система виртуализации KVM/QEMU.

Максимальная частота прерываний равна около 11000 IPS, что означает по крайней мере 11000 выходов в гипервизор в секунду, что должно привести к заметному снижению производительности. Хотя в оригинальной работе [14] частота прерываний была выше (40000-60000 прерываний в секунду), вероятно, в случае теста IS, по крайней мере часть расходов связана с виртуализацией прерываний. Чтобы доказать это предположение, необходимо провести тесты с аналогом системы ExitLess Interrupts.

Высокая частота прерываний для теста IS на 32, 64 процессах также отражает гранулярность коммуникаций в данном тесте. В данном случае, коммуникации имеют “мелкозернистый” характер, что объясняет большое число прерываний. На рис. 3 видно, что накладные расходы гипервизора Palacios меньше на 20% чем расходы KVM, что может быть связано с пониженным уровнем шума основной ОС Kitten.

5. Заключение

Основным вкладом данной работы является демонстрация необходимости корректной эмуляции архитектуры NUMA в соответствии с реальной конфигурацией при запуске HPC-приложений в виртуальных машинах на серверах архитектуры NUMA. Исследования проведены с использованием систем виртуализации KVM/QEMU и Palacios. KVM/QEMU широко используется в индустрии, в то время как Palacios – гипервизор, специально разрабатываемый для виртуализации HPC. В исходный код QEMU и Palacios были внесены необходимые изменения для предоставления гостевой системе реальной топологии NUMA. С использованием эмуляции NUMA, накладные расходы виртуализации были снижены с 10-60% до 1-5% на многих тестах из пакетов HPCC и NAS Parallel Benchmarks. Единственные тесты, на которых накладные расходы выше – RandomAccess и HPL из пакета HPCC, с потерями производительности до 30% и 15% соответственно.

Все тесты выполнялись на современном высокопроизводительном кластере, объединенном сетью Infiniband. Для вычислений использовалось до 96 процессорных ядер. Полученные результаты позволили оценить накладные расходы, вызванные виртуализацией как процессора, так и коммуникационной среды. Также было проведено исследование связи частоты прерываний коммуникационного устройства и накладных расходов виртуализации; для адаптера Infiniband, измерена максимальная частота до 11000 прерываний в секунду на тесте IS пакета NPB при запуске на 64 процессах, накладные расходы при этом достигали 20% для системы виртуализации KVM/QEMU. Нельзя утверждать, что данные расходы вызваны только высокой частотой

прерываний, однако, такая частота может свидетельствовать о “мелкозернистом” характере коммуникаций теста IS.

Сравнивая системы виртуализации KVM/QEMU и Palacios, можно сделать вывод, что в среднем результаты тестов с эмуляцией NUMA одинаковы для обеих систем, причем KVM дает более стабильные и предсказуемые результаты, в то время как Palacios показывает лучшие результаты на “мелкозернистых” тестах при большом числе задействованных процессов. В то же время, Palacios имеет аномалии в производительности на некоторых тестах. Шум основной ОС имеет большое значение для масштабируемости “мелкозернистых” тестов, и в этом аспекте основная ОС Kitten имеет преимущество по сравнению с ОС Linux, что дает лучшую проихводительность и масштабируемость для тестов, выполняющихся в ВМ гипервизора Palacios. С нашей точки зрения, объем шума, генерируемого основной ОС и системой виртуализации, является критически важным параметром для успешной виртуализации многопроцессорных, многоядерных высокопроизводительных систем с достаточно большим числом вычислительных узлов.

В целом, полученные результаты свидетельствуют о целесообразности применения виртуализации для большого класса высокопроизводительных приложений. Основными “узкими местами” систем виртуализации являются уменьшенная производительность системы памяти (критично только для узкого класса задач, в том числе RandomAccess), расходы при виртуализации устройств, а также повышенный уровень “шума”, источником которого становятся основная ОС и гипервизор.

Список литературы

[1] A. J. Younge, R. Henschel, J. Brown, G. von Laszewski, J. Qiu, и G. C. Fox. Analysis of Virtualization Technologies for High Performance Computing Environments. In The 4th International Conference on Cloud Computing (IEEE CLOUD 2011), июль 2011.

[2] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, и P. Saraiya. Abstract High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt), in conjunction with EuroSys 2007, 2007.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, и A. Liguori. KVM: the Linux virtual machine monitor. In OLS ’07: The 2007 Ottawa Linux Symposium, с. 225–230, июль 2007.

[4] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, и A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE ’11, с. 169–180, 2011.

[5] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, и D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC ’06, 2006.

[6] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, и M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center. Декабрь 1995.

[7] Д. В. Силаков. Использование аппаратной виртуализации в контексте информационной безопасности. Труды Института системного программирования РАН, том 20, с. 25-36, 2011.
[8] П. Довгалюк. Детерминированное воспроизведение процесса выполнения программ в виртуальной машине. Труды Института системного программирования РАН, том 21, с. 123-132, 2011.
[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, и A. Warfield. Xen and the art of virtualization. SIGOPS Oper. Syst. Rev., 37, с. 164–177, октябрь 2003.
[10] J. Watson. VirtualBox: bits and bytes masquerading as machines. Linux Journal, февраль 2008.
[11] К. Батузов, А. Меркулов. Оптимизация динамической двоичной трансляции. Труды Института системного программирования РАН, том 20, с. 37-50, 2011.
[12] N. Regola и J.-C. Ducom. Recommendations for virtualization technologies in high performance computing. In Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM '10, с. 409–416, 2010.
[13] OpenVZ: container-based virtualization for Linux, http://openvz.org/.
[14] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, и D. Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. In Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012), 2012.
[15] K. Z. Ibrahim, S. Hofmeyr, и C. Iancu. Characterizing the performance of parallel applications on multi-socket virtual machines. In Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '11, с. 1–12, 2011.
[16] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, и R. Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), с. 1 –12, апрель 2010.
[17] V3VEE: An Open Source Virtual Machine Monitor Framework For Modern Architectures, http://v3vee.org/.
[18] K. Ferreira, P. Bridges, и R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, с. 1–12, ноябрь 2008.
[19] F. Petrini, D. Kerbyson, и S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In 2003 ACM/IEEE Conference on Supercomputing, с. 55, ноябрь 2003.
[20] A. Arcangeli. Transparent Hugepage Support, http://www.linux-kvm.org/wiki/images/9/9e/2010-forum-thp.pdf. KVM Forum 2010.

The prospects for virtualization of high performance x64 systems

Alexander Kudryavtsev alexk@ispras.ru, ISP RAS, Moscow, Russia
Vladimir Koshelev vedun@ispras.ru, ISP RAS, Moscow, Russia
Arutyun Avetisyan arut@ispras.ru, ISP RAS, Moscow, Russia

Abstract. In this paper we explore the prospects of virtualization technologies being applied to x64-based high performance systems. The main reasons for performance overhead when running parallel programs inside multiple virtual machines are discussed. We consider KVM/QEMU and Palacios systems and use HPC Challenge and NASA Advanced Supercomputing Parallel Benchmarks packages as a test suite. All tests are performed using high performance cluster with high-speed Infiniband interconnect.

Gathered results show feasibility of applying virtualization to a big class of high performance applications. We use huge memory pages to decrease number of TLB cache misses and NUMA emulation mechanism to provide the description of memory topology of real system to virtual machine. Such optimizations decrease virtualization overhead from 10-60% to 1-5% for most tests from HPC Challenge and NASA Advanced Supercomputing Parallel Benchmarks suites. The main bottlenecks of virtualization systems are decreased memory system performance (critical only for a narrow class of applications), device virtualization overhead, and increased noise level caused by the host operating system and hypervisor. Noise may affect performance and scalability of fine-grained applications (those with frequent communications of small size). While the number of nodes in the system grows, the noise influence substantially increases.

Keywords: high performance systems; virtualization; virtual machine monitors; high performance computing; parallel computing

Reference

[1]. A. J. Younge, R. Henschel, J. Brown, G. von Laszewski, J. Qiu, and G. C. Fox. Analysis of Virtualization Technologies for High Performance Computing Environments. In The 4th International Conference on Cloud Computing (IEEE CLOUD 2011), July 2011.

[2]. A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. Abstract High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt), in conjunction with EuroSys 2007, 2007. A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In OLS ’07: The 2007 Ottawa Linux Symposium, c. 225–230, July 2007.

[3]. J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE ’11, p. 169–180, 2011.

[4]. P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC ’06, 2006.

[5]. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center. December 1995.
[6]. D. V. Silakov. Ispol'zovanie apparatnoj virtualizatsii v kontekste informatsionnoj bezopasnosti.[Usage of hardware virtualization for enhancing security] Trudy ISP RAN [The Proceedings of ISP RAS], volume 20, p. 25-36, 2011. (In Russian)
[7]. P. Dovgalyuk. Determinirovannoe vosproizvedenie protsessa vypolneniya programm v virtual'noj mashine [**Deterministic program rerun within a virtual machine**]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, p. 123-132, 2011. (In Russian)
[8]. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. SIGOPS Oper. Syst. Rev., 37, с. 164–177, октябрь 2003.
[9]. J. Watson. VirtualBox: bits and bytes masquerading as machines. Linux Journal, февраль 2008.
[10]. K. Batuzov, A. Merkulov. Optimizatsiya dinamicheskoj dvoichnoj translyatsii [**Optimizations in Dynamic Binary Translation**]. (In Russian) Trudy ISP RAN [The Proceedings of ISP RAS], volume 20, p. 37-50, 2011.
[11]. N. Regola and J.-C. Ducom. Recommendations for virtualization technologies in high performance computing. In Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM ’10, с. 409–416, 2010.
[12]. OpenVZ: container-based virtualization for Linux, http://openvz.org/.
[13]. A. Gordon, N. Amit, N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. In Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012), 2012.
[14]. K. Z. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the performance of parallel applications on multi-socket virtual machines. In Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID ’11, с. 1–12, 2011.
[15]. J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), p. 1 – 12, April 2010.
[16]. V3VEE: An Open Source Virtual Machine Monitor Framework For Modern
[17]. Architectures, http://v3vee.org/.
[18]. K. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, p. 1–12, November 2008.
[19]. F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In 2003 ACM/IEEE Conference on Supercomputing, p. 55, November 2003.
[20]. A. Arcangeli. Transparent Hugepage Support, http://www.linux- kvm.org/wiki/images/9/9e/2010-forum-thp.pdf. KVM Forum 2010.

Использование стандарта OpenCL для программирования ПЛИС

Андрей Белеванцев <abel@ispras.ru>
Алексей Меркулов <steelart@ispras.ru>
Владимир Платонов <soh@ispras.ru>

Аннотация. В статье предлагается использование стандарта OpenCL для облегчения программирования логических интегральных схем (ПЛИС), использующихся в качестве акселератора в гетерогенной вычислительной системе. Описывается схема реализации подмножества стандарта, обеспечивающая обмен памятью и управление выполнением задач на ПЛИС в предположении, что ПЛИС связан с центральным процессором через шину PCI-express.Код, выполняющийся на ПЛИС, может быть написан на языке описания аппаратуры в соответствии с предлагаемым интерфейсом взаимодействия либо автоматически сгенерирован из функций языка Си с помощью известных трансляторов типа C-to-Verilog.

Ключевые слова: ПЛИС, OpenCL, Verilog, PCI-express.

1. Введение

В настоящий момент затруднено программирование гетерогенных систем, состоящих из основного многоядерного компьютера и акселератора на базе платы с программируемой логикой (ПЛИС). Не предложено программной модели (как высокоуровневой, так и низкоуровневой) с соответствующей системной поддержкой времени компиляции и выполнения, которая позволила и облегчила бы эффективное использование вычислительных ресурсов таких гибридных систем. Исследования последних десятилетий показывают, что автоматическое получение параллельной программы из последовательной даже для однородной архитектуры не представляется возможным в общем случае.

В случае ПЛИС реализация с их помощью необходимых компонент задачи на аппаратном уровне позволяет достигать большей производительности по сравнению с обычными процессорами, затрачивая при этом меньше энергии в процессе работы. Тем не менее, существенным недостатком ПЛИСов является сложность как самого процесса разработки логики устройства, так и организации взаимодействия между полученным акселератором и хост-машиной. Логика устройства должна быть описана на языке описания аппаратуры, а все механизмы взаимодействия с хостом должны быть так или

иначе реализованы программистом. При этом отсутствует какой-либо единый стандарт на организацию данного взаимодействия.

Целью данной работы является разработка системы программирования ПЛИС с использованием стандарта OpenCL. Эта система будет опираться на низкоуровневый интерфейс, использующий для реализации OpenCL драйвер интерфейса PCI-Express для обмена данными и командами между компьютером общего назначения и ПЛИС, а также планировщик команд на ПЛИС.

В настоящей статье в разделе 2 предлагаются краткие сведения о стандарте OpenCL и работе с ПЛИС-устройствами. Раздел 3 содержит разработанную схему работы с ПЛИС как с OpenCL-устройством, а раздел 4 более подробно описывает подмножество OpenCL, необходимое для реализации предложенной схемы. Раздел 5 заключает статью.

2. Обзор стандарта OpenCL и ПЛИС-устройств

2.1. Программная модель OpenCL

Программная модель OpenCL [1,2] позволяет программисту описывать функции, которые будут параллельно выполнены на некотором акселераторе или наборе акселераторов, доступных на данной машине. Для задания конкретного акселератора служит понятие контекста – структуры данных, задающей класс требуемого акселератора (многоядерный процессор, графический процессор, ПЛИС) – и понятие очереди команд – структуры данных, через которую осуществляется выполнение операций на конкретном акселераторе.

В основе программной модели OpenCL лежит понятия ядра (kernel). Ядро – это функция, которая будет выполнена параллельно на акселераторе определенным количеством потоков. Ядра определяются программистом в виде функций на некотором расширении языка Си. Создание ядра разделено на несколько этапов, каждому из которых отвечает вызов соответствующей функции OpenCL на центральном процессоре. В начале требуется преобразовать исходный код одного или нескольких ядер, заданный в виде массива строк, в OpenCL-программу, далее полученная программа должна быть скомпилирована для нужного устройства. После компиляции OpenCL-программы из нее могут быть получены ядра, отвечающие функциям в ее исходном коде.

Для запуска ядра на акселераторе необходимо указать его аргументы. Аргументами ядра могут быть как скалярные значения, так и адреса буферов в памяти акселератора. Так как в общем случае акселератор не имеет прямого доступа к памяти центрального процессора (например, когда акселератором является ПЛИС или графический ускоритель), ядра работают с адресами в памяти акселератора. OpenCL предоставляет механизм выделения буферов

памяти на акселераторе и обмена данными между памятью центрального процессора и акселератора.

Запуск нужного ядра осуществляется с помощью вызова функции библиотеки OpenCL на центральном процессоре. Акселератор, на котором будет выполняться ядро, задается очередью команд, которой передается запрос на выполнение ядра.

2.2. ПЛИС-устройства

Программируемая логическая интегральная схема (ПЛИС) – полупроводниковое устройство, логика работы которого может быть сконфигурирована пользователем. ПЛИС используются для построения различных цифровых устройств, для которых необходимо большое количество портов ввода-вывода. Такие устройства могут использоваться для высокоскоростной передача данных, криптографии, обработки сигналов, в качестве мостов (коммутаторов) между различными системами. Возможность получения устройства, которое решает требуемую задачу на уровне железа, сразу после разработки схемы этого устройства позволяет использовать ПЛИСы для проектирования и прототипирования интегральных схем, ориентированных на решение одной конкретной задачи. При этом преимущества, даваемые возможностью аппаратной реализации алгоритма, решающего некоторую задачу, позволяют использоваться ПЛИС в качестве ускорителя, настраиваемого на решение определенной задачи или класса задач. Более низкое электропотребление по сравнению с обычными процессорами позволяет использовать ПЛИСы во встраиваемых системах.

Далее под ПЛИС мы будем подразумевать самый мощный класс программируемых схем, а именно программируемые пользователем вентильные матрицы (FPGA). Такие схемы состоят из логических блоков, реализующих базовые операции, блоков ввода-вывода и внутренних связей. Логические блоки имеют некоторое количество входов и выходов, реализуя произвольную функцию алгебры логики от соответствующего числа аргументов на любом своем выходе. Блоки ввода-вывода обеспечивают связь между контактами корпуса и внутренними связями схемы, позволяя схеме взаимодействовать с внешними устройствами. Наконец, внутренние связи организованы в виде вертикальных и горизонтальных каналов, состоящих из нескольких связей. На пересечении каналов находятся блоки переключения, позволяющие подключать связи из канала, входящего в блок, к соответствующим связям из оставшихся каналов.

Одним из ключевых параметров, используемых для оценки ПЛИС, является число логических блоков, определяющее максимальную допустимую сложность конфигурации. Состав и количество блоков различаются как для различных семейств кристаллов ПЛИС, так и в рамках одного семейства. В качестве примера рассмотрим схемы семейства Virtex-6 (Xilinx) [[3]]. Устройства, входящие в данное семейство, могут содержать от ~6000 до

~60000 блоков, каждый из которых содержит: 4 логических элемента, реализующие булевы функции от 6 аргументов в виде таблицы истинности (look-up table, LUT), причем некоторые элементы могут использоваться для хранения 64-битных слов; запоминающие элементы, мультиплексоры, поддержка для учёта переноса разрядов при сложении; элементы блочной памяти (Block RAM, по 36 Kb, до 720 блоков); DSP-элементы, позволяющие более эффективно реализовывать некоторые функции (умножение, сложение, сдвиг и др.). Генерация специализированных блоков из программы на языке описания аппаратуры возможна как автоматически во время синтеза, так и явно за счёт использования предлагаемых производителем специальных модулей (hard macro) [6].

Конфигурация ПЛИС осуществляется путем задания соединений в блоках переключения и реализуемых функций в логических блоках. С точки зрения пользователя этот процесс может быть разбит на несколько частей: описание схемы устройства на уровне логических вентилей с помощью языков описания аппаратуры (Verilog или VHDL); генерация списка соединений, т.е. функций, реализуемых в LUT, и связей между ними; генерация битового потока для прошивки ПЛИС, т.е. собственно разводка созданных функций по физическим LUT-блокам, каналам соединения и блокам переключения; загрузка битового потока на устройство. Для последних шагов по генерации битовой прошивки из описания схемы производителем ПЛИС, как правило, предоставляются интегрированные среды разработки.

3. Схема работы с ПЛИС с использованием OpenCL

В предлагаемой схеме работы с ПЛИС как с ускорителем нами рассматривается гетерогенный узел вычислительной системы, состоящий из центрального (многоядерного) процессора общего назначения, который по шине PCI-express связывается с одним или несколькими ПЛИСами (см. рисунок 1). Управление задачами на ПЛИС и обмен данными с ПЛИС осуществляется со стороны центрального процессора – через OpenCL-библиотеку поддержки времени выполнения, которая, в свою очередь, выдает команды драйверу ПЛИС-устройств (ОС-специфичному, в нашем случае – драйверу ОС Linux); со стороны самого устройства – с помощью системной части прошивки ПЛИС, которая отвечает за запуск пользовательских задач и организацию DMA-обменов с хостовой частью системы через драйвер шины PCI-express. Максимальная системная функциональность по органихации очереди запросов на обмен данными и запуск задач возлагается на драйвер, чтобы освободить ПЛИС от системной нагрузки и отдать максимум прошивки ПЛИС для реализации пользовательской функциональности.

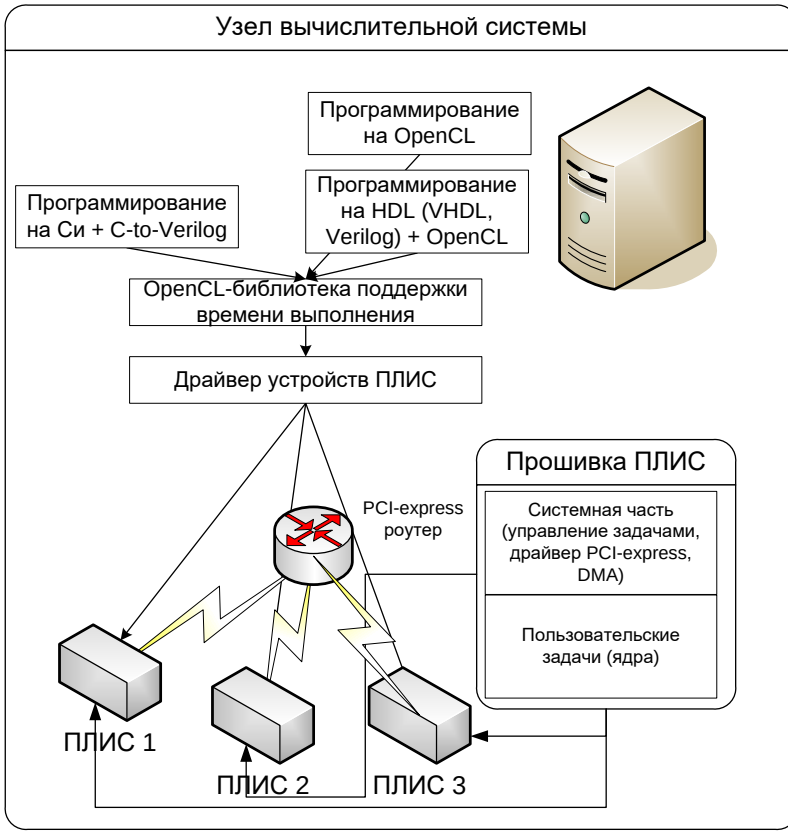


Рис. 1. Схема программирования ПЛИС как акселератора.

Библиотека поддержки времени выполнения со стороны центрального процессора должна поддерживать следующие сценарии использования:

- Программирование на языке Си или на его расширении (OpenCL) с автоматической генерацией необходимых OpenCL-вызовов системной библиотеки [4] (например, с помощью привлечения транслятора C-to-Verilog [5]) и/или с автоматической генерацией кода на языке описания аппаратуры и ручной вставкой необходимых OpenCL-вызовов;
- Программирование непосредственно на языке описания аппаратуры (VHDL/Verilog) и автоматическое встраивание получившегося кода в прошивку ПЛИС, а также ручная вставка необходимых OpenCL-вызовов системной библиотеки для управления получившимся кодом на ПЛИС.

4. Реализация подмножества OpenCL в библиотеке поддержки

Для базовой работы с ПЛИС с помощью стандарта OpenCL с учетом описанной в разделе 3 схемы необходимо реализовать набор перечисленных ниже функций OpenCL. При описании каждой функции мы даем комментарии о ее реализации в библиотеке:

- `clGetDeviceIDs` – получение списка всех устройств заданного типа. Библиотека поддержки в реализации этой функции опирается на драйвер, создающий файлы в собственном подкаталоге дерева /sys/bus/pci/drivers.
- `clCreateContextFromType` – создание контекста для работы с устройством ПЛИС. Контекст представляет из себя структуру данных, хранящую информацию о типе устройства (или устройств), с которым будет работать пользовательская программа. В ходе этого вызова библиотека также инициализирует необходимые внутренние структуры данных для всех устройств, найденных через вызов `clGetDeviceIDs`.
- `clCreateCommandQueue` – создание очереди команд для данного устройства. Выполнение команд организуется библиотекой в отдельном потоке для достижения асинхронности большинства вызовов.
- `clCreateBuffer` – выделение буфера памяти в памяти устройства. Библиотека делегирует эту функцию драйверу устройства, который без обращения к устройству поддерживает у себя карту памяти устройства, на которой отмечены выданные и свободные области.
- `clEnqueueWriteBuffer/clEnqueueReadBuffer` – запись данных из памяти центрального процессора в память устройства и из памяти устройства в память центрального процессора. Эти асинхронные вызовы, как уже упоминалось, выполняются библиотекой с помощью обработки очереди соответствующих запросов в отдельном потоке. Так как функции требуют некоторой работы со стороны устройства, то соответствующим запросам присваивается идентификатор, через который можно отследить окончание их выполнения.
- `clEnqueueNDRangeKernel` – запуск ядра на выполнение. Для этой функции верны все те же замечания, что и для функций обмена данными. Предварительно перед запуском ядру передаются необходимые параметры, установленные ранее через вызовы `clSetKernelArg`. Пользовательские ядра, содержащиеся в прошивке, должны удовлетворять заданному

бинарному интерфейсу на аргументы для успешного запуска. В текущей версии библиотеки и драйвера аргументы могут быть целыми числами либо указателями на массивы в памяти ПЛИС.

- `clCreateProgramWithBinary` – создание OpenCL-программы и инициализация устройства. Программа создается из заранее сгенерированного бинарного образа, который представляет собой битовый поток для загрузки на устройство с дополнительной информацией о том, какие операции в нем реализованы, чтобы программист мог ссылаться на конкретные операции для данной прошивки. Выполнение этой функции библиотекой влечет за собой автоматическую прошивку устройства через JTAG.
- `clCreateKernel` – получение ядра из созданной ранее OpenCL-программы. Для библиотеки эта операция требует лишь инициализации внутренних структур данных после проверки того, что данная операция присутствует в текущей прошивке на устройстве (см. выше).
- `clSetKernelArg` – установка аргументов ядра. Аналогично, функция реализуется через соответствующие изменения внутренних структур данных после проверки корректности установки аргументов.
- `clWaitForEvents` – ожидание выполнения запущенных ранее асинхронных команд. Библиотека опрашивает драйвер устройства на предмет окончания работ, соответствующих параметрам-идентификаторам. Драйвер поддерживает актуальное состояние работ через общение с устройством – окончание работы влечет генерирование устройством прерывания, обрабатываемого драйвером.
- `clReleaseMemObject`, `clReleaseKernel`, `clReleaseProgram`, `clReleaseCommandQueue`, `clReleaseContext` – удаление объектов (буфера памяти, OpenCL-ядра, OpenCL-программы, очереди команд, контекста соответственно), созданных парными вызовами `clCreateXXX`.

Данный набор операций позволяет создавать и загружать на ПЛИС битовый поток с нужным набором пользовательских операций, работать с буферами памяти на устройстве и контролировать выполнение операций, поддерживаемых загруженным битовым потоком.

Весь набор вызовов, предоставляемый предлагаемой библиотекой, можно разделить на два класса – синхронные вызовы, возврат из которых означает, что требуемая операция уже завершена, и асинхронные, возврат из которых означает, что требуемая операция будет выполнена в дальнейшем. К асинхронным вызовам относятся вызовы функций, управляющих обменом данными между памятью центрального процессора и ПЛИС

(clEnqueueReadBuffer и clEnqueueWriteBuffer), а также запуск пользовательского ядра на устройстве (clEnqueueNDRangeKernel). Все прочие вызовы являются синхронными.

Для синхронизации с асинхронными вызовами, запущенными ранее, служит понятие события. При каждом вызове асинхронной операции создается событие, ассоциированное с данным вызовом, позволяющее процессу дождаться окончания работы соотвествующей операции (clWaitForEvents). Пример работы с асинхронными вызовыми приведен ниже (рис. 2).

```
clEnqueueWriteBuffer (queue, buff_a, CL_TRUE, 0, sizeof
(host_buff_a), host_buff_a, 0, NULL, &event_a);
clEnqueueWriteBuffer (queue, buff_b, CL_TRUE, 0, sizeof
(host_buff_b), host_buff_b, 0, NULL, &event_b);
clWaitForEvents(1, &event_a);
clWaitForEvents(1, &event_b);
```

Рис. 2. Пример работы с асинхронными вызовами OpenCL.

В данном примере инициируется запись двух буферов памяти на устройстве из соотвествующих областей памяти центрального процессора (строки 1-2). После того, как запись была инициирована, пользовательское приложение может либо продолжать свою работу параллельно с записью данных на устройство, либо заблокироваться до окончания выполнения записи (строки 3-4).

Рассмотрим более полный пример работы с ПЛИС с помощью функций OpenCL библиотеки поддержки (рисунок 3). Как указано ранее, данный пример может быть написан программистом вручную либо частично сгенерирован автоматически из описания функции на языке Си.

```
int A [100][100];
int B [100][100];
int C [100][100];

cl_device_id dev;
cl_uint num_of_devs;
cl_uint m_size = 100 * 100;
size_t binary_size;
cl_event main_event, event_a, event_b, event_c;
int arg_value = 100;
const unsigned char * binary = "/path/to/kernels.bin";
int binary_status;
size_t size = 1;
cl_context context = clCreateContextFromType(NULL,
CL_DEVICE_TYPE_FPGA, NULL, NULL, &error);
```

```
clGetDeviceIDs(0, CL_DEVICE_TYPE_FPGA, 1, &dev,
&num_of_devs);
cl_command_queue queue = clCreateCommandQueue(context,
dev, 0, &error);
cl_mem buff_a = clCreateBuffer(context,
CL_MEM_READ_WRITE, sizeof (A), NULL, &error);
cl_mem buff_b = clCreateBuffer(context,
CL_MEM_READ_WRITE, sizeof (B), NULL, &error);
cl_mem buff_c = clCreateBuffer(context,
CL_MEM_READ_WRITE, sizeof (C), NULL, &error);
clEnqueueWriteBuffer(queue, buff_a, CL_TRUE, 0, sizeof
(A), A, 0, NULL, &event_a);
clEnqueueWriteBuffer(queue, buff_b, CL_TRUE, 0, sizeof
(B), B, 0, NULL, &event_b);
cl_program program = clCreateProgramWithBinary(context,
1, &dev, &binary_size, &binary, &binary_status, &error);
cl_kernel kernel = clCreateKernel(program, "mmm",
&error);
clSetKernelArg(kernel, 0, sizeof (buff_a), &buff_a);
clSetKernelArg(kernel, 1, sizeof (buff_b), &buff_b);
clSetKernelArg(kernel, 2, sizeof (buff_c), &buff_c);
clSetKernelArg(kernel, 3, sizeof (m_size), &m_size);
clWaitForEvents(1, &event_a);
clWaitForEvents(1, &event_b);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &size,
NULL, 0, NULL, &main_event);
clWaitForEvents(1, &main_event);
clEnqueueReadBuffer (queue, buff_c, CL_FALSE, 0, sizeof
(C), C, 0, NULL, &main_event);
clWaitForEvents(1, &main_event);
clReleaseMemObject(buff_a);
clReleaseMemObject(buff_b);
clReleaseMemObject(buff_c);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

Рис. 3. Пример использования OpenCL для запуска задачи и передачи данных.

В первую очередь необходимо создать контекст для работы с некоторым классом устройств (в данном случае – ПЛИС). После создания контекста можно получить список устройств данного типа, доступных библиотеке (вызов clGetDeviceIDs). Для работы с устройством для него необходимо создать очередь команд, через которую на него будут передаваться

управляющие команды. Также необходимо выделить буферы памяти для обмена данными между памятью устройства и памятью центрального процессора. В данном примере производится выделение трех буферов памяти и запуск копирования данных из памяти центрального процессора в выделенные буферы на устройстве.

Далее производится создание OpenCL-программы из заранее созданной бинарной программы. При вызове clCreateProgramWithBinary устройство автоматически программируется битовым потоком, содержащимся в указанном бинарном образе. Далее из полученной OpenCL-программы выделяется ядро и указываются аргументы для его запуска.

Для корректной работы ядра требуется, чтобы все необходимая для его работы информация была записана в память устройства. Вызовы clWaitForEvents блокируют процесс до тех пор, пока операции передачи данных на устройство не будут завершены. После завершения записи данных на устройство требуемая операция может быть запущена на выполнение.

Далее производится ожидание окончания выполнения операции и копирование результатов в память центрального процессора. В конце работы программы производится освобождение выделенной памяти на устройстве и удаление объектов, созданных в процессе работы с библиотекой.

Данный пример иллюстрирует полный цикл работы с библиотекой с точки зрения пользователя, работающего с ПЛИС как с OpenCL-устройством. При этом с целью сокращения размера примера, были опущены все проверки возвращаемых значений.

5. Заключение

В данной статье были рассмотрена схема работы с ПЛИС-устройством как с ускорителем, доступным в системе через PCI-express-шину, с помощью стандарта OpenCL. Кратко описаны особенности реализации необходимой библиотеки поддержки. Основной задачей библиотеки является поддержка информации о текущих устройствах в системе, организация асинхронных обменов данными и запусков задач через драйвер устройств. Таким образом, вся специфичная для данного ПЛИС-устройства и данной системной прошивки работа ложится на драйвер. Реализация самой библиотеки не представляет большой трудности.

По нашему опыту, основная сложность поддержки OpenCL заключается в создании корректной системной прошивки ПЛИС, позволяющей выполнять быстрые обмены данными с хостом через PCI-express. Разработка прошивки трудна как из-за общей сложности языков описания аппаратуры, так и из-за проблем с отладкой в связке с драйвером ОС Linux со стороны хоста. Так, исправление ошибки драйвера или прошивки может потребовать трех перезагрузок инструментальной машины. В настоящий момент эти работы подходят к концу, и для образцов кода на Си из тестовой программы уже

получено первоначальное ускорение с использованием ПЛИС. По окончании этой работы основной задачей станет производительность обменов данными и конвейеризация вычислений (пользовательских ядер) на ПЛИС, также с использованием стандарта OpenCL.

Список литературы

[1]. Khronos OpenCL Working Group. The OpenCL 1.1 Specification, September 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[2]. NVIDIA OpenCL JumpStart Guide, April 2009. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

[3]. Xilinx Virtex-6 Family Overview. Version 2.3, March 2011. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[4]. Андрей Белеванцев Алексей Кравец, Александр Монаков. Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели. Труды ИСП РАН, том 21, 2011.

[5]. Nadav Rotem and Yosi Ben Asher. C to Verilog. Automating circuit design. http://c-to-verilog.com/.

[6]. C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using Hard Macros to Reduce FPGA Compilation Time. International Conference on Field Programmable Logic and Applications, IEEE, 2010, pp. 438–44.

Optimizations in Dynamic Binary Translation

Andrey Belevantsev <abel@ispras.ru> ISP RAS, Moscow, Russia
Alexey Merkulov <steelart@ispras.ru> ISP RAS, Moscow, Russia
Vladimir Platonov <soh@ispras.ru> ISP RAS, Moscow, Russia

Abstract. We suggest using OpenCL standard for programming FPGA devices that are used as accelerators in a heterogeneous system. We describe the implementation of a subset of OpenCL that is required for organizing data exchange and task management for FPGAs given that CPU and FPGA are connected via PCI-express bus. Basically, the first part of the required functions is the simple device manipulation and FPGA program loading; the latter requires flashing the FPGA via the JTAG interface. The second part is the memory buffer transfer to and from the FPGA. Its implementation in the runtime library is straightforward given that the FPGA supports PCI-express exchanges; the main load falls onto the FPGA driver and the FPGA system-level firmware organizing these exchanges. The final part is the FPGA task management that is achieved via the simple task scheduler implemented within the FPGA driver. The code running on FPGA can be created with a hardware description language or generated automatically using one of the known translators, e.g. C-to-Verilog, but it should adhere to the ABI described by the FPGA driver and firmware implementations.

Keywords: ПЛИС, OpenCL, Verilog, PCI-express.

References

[1]. Khronos OpenCL Working Group. The OpenCL 1.1 Specification, September 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[2]. NVIDIA OpenCL JumpStart Guide, April 2009. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

[3]. Xilinx Virtex-6 Family Overview. Version 2.3, March 2011. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[4]. A. Belevantsev, A. Kravets, A. Monakov. Avtomaticheskaya generatsiya OpenCL-koda iz gnezd tsiklov s pomoshh'yu poliehdral'noj modeli. [Automatically generating OpenCL code from loop nests via a polyhedral model] Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, p. 5-22, 2011. (In Russian)

[5]. Nadav Rotem and Yosi Ben Asher. C to Verilog. Automating circuit design. http://c-to-verilog.com/.

[6]. C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using Hard Macros to Reduce FPGA Compilation Time. International Conference on Field Programmable Logic and Applications, IEEE, 2010, pp. 438–44.

Оптимизация расчётов в пакете OpenFOAM на GPU[1]

А. В. Монаков <amonakov@ispras.ru>

Аннотация. В статье рассматривается задача повышения скорости расчётов в пакете OpenFOAM за счёт переноса части вычислений на графические акселераторы (GPU). Приводится краткий обзор пакета и анализ переноса на GPU метода сопряжённых градиентов. Описаны несколько оптимизаций, часть из которых специфична для рассматриваемой реализации, а часть применима не только для GPU. Приводятся первичные результаты тестирования производительности.

Ключевые слова. OpenFOAM, CUDA, GPGPU, метод сопряжённых градиентов, предобуславливание на GPU, оптимизация для GPU

1. Введение

OpenFOAM — пакет библиотек и программ для организации научных расчётов преимущественно в области вычислительной гидродинамики и механики сплошных сред [1]. Пакет содержит поддержку запуска расчётов на кластере через MPI. Кроме приложений, выполняющих непосредственные расчёты, OpenFOAM содержит вспомогательные программы для подготовки сеток и постобработки результатов.

OpenFOAM содержит несколько десятков программ, выполняющих расчёты в соответствии с различными схемами. Выбор схемы зависит от особенностей решаемой задачи, таких как сжимаемость жидкости, ламинарность потока, способ учёта турбулентности и другие. Примерами этих программ в OpenFOAM являются icoFoam, pisoFoam, interFoam и другие. Обычно запуск этих программ требует наибольшего компьютерного времени в цикле использования OpenFOAM (подготовка сетки – решение – постпроцессинг и визуализация результатов) и поэтому требует тщательной оптимизации.

[1] Работа проводится в рамках реализации ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы», ГК № 07.514.11.4119 от 02 ноября 2011 года.

Работа проводится при поддержке программы «Университетский кластер», http://www.unicluster.ru

Большинство этих солверов в процессе своей работы решают множество дифференциальных уравнений на сетке, сводя их к обыкновенным системам линейных алгебраических уравнений над вещественными числами. OpenFOAM содержит лишь небольшое число методов решения СЛАУ. В то же время, для больших задач время решения этих систем может покрывать существенную долю общего времени расчёта. Таким образом, оптимизация методов решения СЛАУ является первоочередным способом повышения производительности OpenFOAM.

OpenFOAM содержит реализации следующих методов решения СЛАУ:

- Метод сопряжённых градиентов, с предобуславливанием (PCG). Этот алгоритм применим для систем с симметричной матрицей (в контексте OpenFOAM это обычно системы для давления).
- Метод бисопряжённых градиентов (PBiCG). Вариант предыдущего метода, работающий для систем с несимметричной матрицей (например, уравнения для компонент скоростей). В развиваемом сообществе проекте OpenFOAM-extend [2] также реализован более надёжный стабилизированный метод бисопряжённых градиентов (BiCGStab). Как и PCG, эти реализации используют предобуславливание для увеличения скорости сходимости.
- Геометро-алгебраический многосеточный метод (GAMG). Этот метод применим вне зависимости от симметричности матрицы системы.

Все реализации поддерживают распределённое решение на кластере, используя коммуникации через MPI.

Все приведённые методы решения являются не прямыми (как, например, метод Гаусса для решения плотных систем), а итерационными: при решении системы $Ax = b$ они начинают с некоторого начального приближения x_0 и циклически строят новые решения x_1, x_2, ..., которые постепенно приближаются к точному решению. На практике достаточно близкие к точному решения получаются за сравнительно небольшое по сравнению с размером системы количество итераций.

Большинство несимметричных систем в OpenFOAM решается за сравнительно малое (обычно не более 10) количество итераций. Это означает, что использование акселератора с отдельной памятью не может существенно ускорить решение этих систем, так как возникнут накладные расходы, связанные с копированием матрицы системы и векторов в память акселератора. Таким образом, основное внимание требуют метод сопряжённых градиентов и многосеточный метод для случая симметричных систем.

Матрицы решаемых систем являются весьма разреженными. Поскольку в большинстве задач используется метод конечных объёмов с дискретизацией первого порядка, количество строк и столбцов в матрицах систем равняется количеству ячеек сетки, а количество внедиагональных ненулевых элементов

в каждой строке — количеству соседей соответствующей ячейки. В частности, для задач, решаемых на ортогональных трёхмерных сетках, большинство строк системы содержат семь ненулевых элементов.

2. Предобуславливание

Пусть для матрицы A известен оператор M, в каком-то смысле приближающий обратную к A матрицу. Тогда при решении системы

$$Ax = b$$

можно перейти к решению эквивалентной системы

$$AMx' = b,\ x = Mx'$$

В силу выбора M, матрица новой системы AM близка к единичной, и для неё итерационные методы будут сходиться быстрее. M называется предобуславливающим оператором. Известно множество подходов к построению предобуславливателей, некоторые из которых будут рассмотрены далее [3].

3. Постановка задачи

Нашей задачей является доработка OpenFOAM с целью повышения производительности решателей за счёт использования GPU. В соответствии с обзором, это включает решение следующих подзадач:

1. Портирование итерационных методов решения на GPU для работы на одном GPU:
    1. Метода сопряжённых градиентов (PCG)
    2. Метода бисопряжённых градиентов (PBiCG, PBiCGStab)
    3. Многосеточного метода
2. Поддержка запуска на кластере, содержащим несколько узлов с GPU:
    1. Поддержка MPI-коммуникаций в GPU-варианте солвера
    2. Выбор метода поддержки использования нескольких GPU на одном узле

Описываемая в статье работа была посвящена портированию и оптимизации метода сопряжённых градиентов на GPU. Несимметричные солверы (1.2) не были портированы, так как на всех тестовых задач они сходятся достаточно быстро, и порт на GPU не даст заметного прироста производительности.

4. Реализация метода сопряжённых градиентов на GPU

Простой вариант метода сопряжённых градиентов выглядит в псевдокоде следующим образом [4]:

```
r = b - Ax
p = z = Mr
v' = (r, z)
do {
  t = Ap
  a = v' / (p, t)
  x = x + ap
  r = r - at
  if (small (r))
        break;
  z = Mr
  v = (r, z)
  w = v / v'
  v' = v
  p = r + wp
}
```

(где A — матрица системы, M — выбранный предобуславливатель, b — правая часть, x — начальное приближение и искомое решение, r, z, p, t — вспомогательные векторы, v, v', a, w — скалярные величины, (,) - взятие скалярного произведения).

Таким образом, на GPU необходима реализация следующих операций:

1. Простые операции уровня BLAS-1: взятие линейной комбинации и скалярного произведения.
2. Умножение разреженной матрицы на вектор.
3. Применение предобуславливающего оператора.

Все эти операции будут ограничены на GPU пропускной способностью памяти. Соответственно, минимизация трафика в глобальной памяти GPU важна для повышения скорости GPU-реализации, а достигнутая пропускная способность может использоваться для оценки эффективности.

Первая группа операций несложно распараллеливается на GPU и доступна в библиотеке CUBLAS. Умножение матрицы на вектор исследовано в множестве работ. Известно, что эффективность ядра на GPU существенно зависит от формата, в котором представлена разреженная матрица. В нашей работе используется разработанный ранее специализированный формат и ядро умножения, которые позволяют получить более высокую по сравнению с опубликованными ранее подходами производительность.

Отметим, что в случае однопоточной работы копирование данных через интерфейс PCI-Express внутри цикла итерационного метода необходимо только в одном месте: при использовании нормы вектора невязки для выхода из цикла. В остальных случаях можно хранить все промежуточные значения исключительно на GPU, что требует реализации тривиального ядра взятия

отношения двух чисел (и противоположного ему, для операции r = r - at). В нашей реализации ненужные копирования исключены, а копирование нормы невязки осуществляется асинхронно.

В случае распараллеленного запуска возникнут дополнительные копирования через сеть после операций взятия скалярного произведения и перед домножением на матрицу системы.

5. Предобуславливание на GPU

5.1. Вычисление предобуславливателя

Задача эффективного предобуславливания на GPU исследована сравнительно неглубоко. Известны две основные категории предобуславливателей:

1. Основанные на неполном разложении на треугольные множители (ILU). Эти предобуславливатели могут быть вычислены сравнительно дёшево, но их применение требует решения двух треугольных систем уравнений, что обладает низким параллелизмом и сложно реализуемо для GPU.
2. Основанные на явном вычислении приближённой обратной матрицы (FSAI, AINV). Применение этих предобуславливателей является простым умножением матрицы на вектор, однако их вычисление сравнительно дорого.

Также следует отметить, что существуют блочные варианты ILU, разработанные для многопроцессорных реализаций, и простой диагональный предобуславливатель, который хорошо подходит для GPU и не требует долгих предвычислений, но слабо снижает скорость сходимости итерационного метода.

Для реализации в рамках первого этапа был выбран предобуславливатель AINV [5]. Он хорошо подходит для ранних исследований, так как параметризован drop tolerance — минимальным абсолютным значением своих ненулевых элементов, варьирование которого позволяет тестировать различные компромиссы между скоростью применения предобуславливателя и общей скоростью сходимости.

Параметр drop tolerance (droptol) может быть выбран в диапазоне 0.0 .. 1.0. Для droptol = 0 AINV построит точное разложение исходной матрицы на треугольные множители. В этом случае метод сопряжённых градиентов сойдётся за одну итерацию (если пренебречь потерями точности в вещественной арифметике), но полученные множители будут плотно заполненными треугольными матрицами, т.е. умножение на предобуславливатель будет слишком дорогой операцией, а количество памяти, необходимой для его представления, будет пропорционально квадрату размера системы.

Для droptol = 1.0 полученные треугольные множители, как правило, не будут содержать ненулевых элементов вне диагонали. Таким образом, полученный предобуславливатель будет совпадать с диагональным: его применение будет сравнительно дешёвой операцией, но скорость сходимости итерационного метода может оказаться слишком низкой.

Идеальное значение drop tolerance зависит от решаемой системы, и динамический подбор оптимального значения этого параметра является нетривиальной задачей.

Отметим, что существует возможность полностью скрыть расходы, связанные с вычислением предобуславливателя. Она основана на том, что последовательно решаемые для одного и того же дифференциального уравнения системы имеют близкие коэффициенты; как следствие, получаемые для них предобуславливатели также будут мало отличаться. Поскольку точное знание предобуславливателя не требуется, возникает возможность повторно использовать предобуславливатели, полученные для предшествующих матриц. Это соображение позволяет вычислять предобуславливатель не каждый раз, а только для какой-то доли решаемых систем.

Развивая эту идею дальше, можно заметить, что вычисление предобуславливателя можно организовать асинхронно. Таким образом, каждый запуск реализации метода сопряжённых градиентов на GPU может немедленно начинать итерации на GPU, используя последний из полученных ранее предобуславливателей, и в то же время запускать вычисление предобуславливателя для текущей системы в отдельной нити на CPU (если такая нить не активна).

5.2. Применение предобуславливателя

Применение предобуславливателя является самой дорогой операцией в цикле метода сопряжённых градиентов (если не принимать во внимание случаи близких к единице значений drop tolerance). Были найдены два способа оптимизации:

1. Использование формата с одинарной точностью (float) для хранения предобуславливателя. Это возможно сделать, так как для предобуславливателя потеря точности не важна.
2. Переупорядочивание строк и столбцов в треугольных множителях, которыми оперирует AINV.

Последний пункт требует отдельного пояснения. Дело в том, что полученный по алгоритму AINV предобуславливатель представляет собой произведение верхнетреугольной и нижнетреугольной матрицы (для симметричной системы эти матрицы получаются друг из друга транспозицией). Таким образом, применение предобуславливателя выполняется как два домножения вектора невязки на матрицу: сначала на нижнетреугольную, затем на верхнетреугольную. Было замечено, что домножение на нижнетреугольный

множитель требует в два-три раза больше времени на GPU, хотя количество ненулевых элементов в них одинаково. Проще всего такой эффект объясняется тем, что блоки CUDA-нитей на GPU запускаются в порядке возрастания их индексов; таким образом, для нижнетреугольного множителя сначала будут запущены блоки, обрабатывающие слабо заполненные верхние строки матрицы. Это может приводить к тому, что более плотно заполненные нижние строки обрабатываются блоками, запущенными в последнюю очередь, что приводит к неполной загрузке GPU.

Поскольку промежуточный вектор, полученный от домножения на нижнетреугольный множитель, используется исключительно для последующего домножения на второй множитель предобуславливателя, можно заметить, что можно обратить порядок строк нижнетреугольного множителя и получать таким образом "перевёрнутый" промежуточный вектор. Чтобы сохранить результат применения предобуславливателя, далее достаточно обратить порядок столбцов второго, верхнетреугольного множителя. После этого оба множителя становятся лево-верхнетреугольными, и не приводят к эффекту плохой балансировки нагрузки на GPU.

6. Переупорядочивание расчётной сетки

Известно, что для фиксированного drop tolerance количество ненулевых элементов в множителях AINV существенно зависит от порядка нумерации элементов расчётной сетки. Это объясняется тем, что порядок нумерации влияет на структуру расположения ненулевых элементов ("портрет" разреженной матрицы), а от неё, в свою очередь, зависит количество добавленных ненулевых элементов в процессе биортогонализации, с помощью которого строится предобуславливатель.

Аналогично, для основанных на неполном треугольном разложении (ILU) предобуславливателей перестановка ячеек также имеет значение. Для ILU0 (ILU-разложение без добавления ненулевых элементов) она влияет на локальность по кешу при решении треугольных систем.

Порядок, который соответствует исходной нумерации ячеек полученной от какого-либо генератора сетки, обычно не оптимален с точки зрения минимизации заполнения предобуславливателя. Так, для ортогональной сетки в кубической расчётной области порядок будет соответствовать лексикографической нумерации ячеек. Этот порядок породит семидиагональную матрицу, а AINV добавит плотное облако элементов вокруг каждой диагонали.

В OpenFOAM реализовано переупорядочивание в соответствии с порядком поиска в ширину (упрощённый вариант алгоритма Cuthill-McKee [6, гл. 4.5]). Такое переупорядочивание хорошо подходит для ILU0-предобуславливателя, аналогом которого является используемый в OpenFOAM DILU, так как

группирует ненулевые элементы плотнее вокруг главной диагонали. Воспользоваться этой реализацией позволяет утилита renumberMesh.

В то же время, для AINV более предпочтительно использовать так называемые "снижающие заполнение" порядки (fill-reducing ordering). Нами была реализована библиотека для OpenFOAM, которая позволяет переупорядочивать сетки по алгоритму Nested Dissection (используется реализация из библиотеки METIS [7]).

7. Вычисления со смешанной точностью на GPU

Структура итерационных методов решения систем уравнений позволяет успешно использовать вычисления со смешанной точностью, когда для решения системы, заданной в двойной точности, большая часть вычислений производится с одинарной точностью, а небольшая доля вычислений с двойной точностью используется для того, чтобы обеспечить такую точность, как если бы использовалась реализация, использующая исключительно двойную точность.

Обычно в статьях, обсуждающих вычисления со смешанной точностью, отправной точкой для обоснования их использования является ограниченность ресурсов ALU двойной точности (ранние GPU или FPGA, в которых снижение количества ALU двойной точности позволяет разместить больше ALU одинарной точности на кристалле [8]). Однако даже для устройств, не испытывающих существенного недостатка в ALU двойной точности (GPU серии Tesla на технологии Fermi), использование этого подхода приносит существенное ускорение, так как вычисления ограничены пропускной способностью памяти.

Нами была реализована схема итеративного уточнения (iterative refinement) для метода сопряжённых градиентов с сохранением вектора направления спуска (p) и обновлениями вектора невязки, когда его норма снижается в 16 раз по сравнению с максимальной после последнего обновления решения в двойной точности (иными словами, когда получены 4 старших двоичных разряда вектора поправки, он прибавляется к вектору решения в двойной точности, и невязка пересчитывается также в двойной точности; вектор направления остаётся неизменным).

8. Экспериментальные результаты

Далее приведены результаты тестирования описанной реализации на узле с CPU Intel Xeon X5650 (2.67 GHz) и GPU Nvidia Tesla M2090. Распределённый запуск OpenFOAM не использовался, т.е. задействовано только одно ядро процессора.

8.1. Исследование ускорения в зависимости от размера сетки

В таблице 1 приведены результаты запусков для тестовой задачи Cavity3D. Задача получена из стандартного примера icoFoam/cavity путём преобразования геометрии области в трёхмерный куб. При уменьшении шага сетки в два раза шаг по времени также уменьшался в два раза, чтобы сохранить сходимость схемы; при этом конечное время симуляции уменьшалось в 16 раз, чтобы приблизительно сохранить общее количество вычислений.

Сетка	Шаг	End time	Время на CPU, с	Время на GPU, с	Ускорение
20x20x20	0.001000	2.00000	22	42	0.52
40x40x40	0.000500	0.12500	71	56	1.27
80x80x80	0.000250	0.00775	284	91	3.12
100x100x100	0.000125	0.00050	105	48	2.18

Таб. 1. Результаты экспериментальных запусков OpenFOAM.

При решении на CPU использовался метод PCG в двойной точности со стандартным для него предобуславливателем DILU. При решении на GPU использовалась фиксированный drop tolerance, равный 0.1; перенумерация сетки и вычисления со смешанной точностью не использовались. В столбцах «Время на CPU/GPU» указано общее время счёта в icoFoam.

Можно отметить, что из-за накладных расходов GPU реализация медленнее в два раза для сетки с 8000 ячеек, но уже для 64000 ячеек наблюдается прирост производительности.

9. Основные результаты

В качестве основных результатов работы хотелось бы отметить реализацию следующих оптимизаций:

1. Асинхронное вычисление предобуславливателя.
2. Использование лево-верхнетреугольных множителей в AINV.
3. Использование переупорядочивания расчётной сетки для повышения эффективности предобуславливания.
4. Использование вычислений со смешанной точностью.

Насколько известно автору, первые два улучшения предложены впервые.

10. Заключение

В дальнейшем планируется добавить поддержку распределённого запуска в разработанную реализацию метода сопряжённых градиентов. Также планируется разработать и протестировать GPU версии многосеточного метода и BiCGStab. Наконец, необходимо найти эффективный метод автоматического подбора drop tolerance для предобуславливателя.

Кроме того, интересно будет исследовать возможность более эффективного по сравнению с распараллеливанием через MPI использования нескольких акселераторов на одном узле, возможность использования метода IDR(s), который может обладать более высокой скоростью сходимости, оптимизацию вычисления предобуславливателя и перенумерации сетки под особенности GPU.

Список литературы

[1] SGI, The OpenFOAM Foundation, http://openfoam.org/
[2] The OpenFOAM Extend Project, http://www.extend-project.de/
[3] M. Benzi, Preconditioning techniques for large linear systems: a survey. J. Comput. Phys., 128 (2002), 418–477
[4] Y. Saad, Iterative methods for sparse linear systems, SIAM, Philadelphia, 2003, 567
[5] R. Bridson, W.-P. Tang, **Ошибка! Недопустимый объект гиперссылки.**, Journal on Computational and Applied Math, 123 (2000), Numerical Analysis 2000 vol. III: Linear Algebra, pp. 293-306.
[6] S. Pissanetzky, Sparse Matrix Technology, Academic Press, Waltham, 1984, 312
[7] G. Karypis, V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system, version 4.0, http://www.cs.umn.edu/~metis, 2009
[8] D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, International Journal of Parallel, Emergent and Distributed Systems (IJPEDS), Special issue: Applied parallel computing, 22 (2007), 221–256

Optimizing OpenFOAM GPU Solvers

Alexander Monakov amonakov@ispras.ru, ISP RAS, Moscow, Russia

Abstract.The paper presents preliminary research on improving performance of CFD simulations in OpenFOAM via offloading parts of computations (specifically, solution of linear systems) to a graphics accelerator (GPU). We present a short review of OpenFOAM package and describe porting conjugate gradient method to the GPU architecture using CUDA programming model. Porting the basic algorithm is straightforward, however care should be taken to avoid unnecessary copying over PCI-Express bus. Efficient preconditioning on the GPU is then discussed. We use approximate inverse preconditioning, which can be implemented with good parallelism on the GPU. To amortize the cost of preparing the preconditioner, we allow reuse of preconditioners on the GPU and compute them on the CPU in a helper thread asynchronously. We mention several optimization opportunities: reordering the preconditioner to upper-left triangular form so that CUDA blocks multiplying by denser parts of preconditiner factors are scheduled first; using single-precision storage for the preconditioner to save memory bandwidth; reordering the mesh with nested dissection method from Metis library and using mixed-precision iteration for the conjugate gradient method. Preliminary performance testing results show performance improvement starting from 64000-cell meshes and reaching 2x for a 1-million cell mesh for a non-parallel run. As future work we mention support for parallel runs with MPI, research of other solvers such as multigrid, BiCGStab and IDR, and choosing drop tolerance automatically for the AINV preconditioner.

References

[1]. SGI, The OpenFOAM Foundation, http://openfoam.org/
[2]. The OpenFOAM Extend Project, http://www.extend-project.de/
[3]. M. Benzi, Preconditioning techniques for large linear systems: a survey. J. Comput. Phys., 128 (2002), 418–477
[4]. Y. Saad, Iterative methods for sparse linear systems, SIAM, Philadelphia, 2003, 567
[5]. R. Bridson, W.-P. Tang, Refining an approximate inverse, Journal on Computational and Applied Math, 123 (2000), Numerical Analysis 2000 vol. III: Linear Algebra, pp. 293-306.
[6]. S. Pissanetzky, Sparse Matrix Technology, Academic Press, Waltham, 1984, 312
[7]. G. Karypis, V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system, version 4.0, http://www.cs.umn.edu/~metis, 2009
[8]. D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, International Journal of Parallel, Emergent and Distributed Systems (IJPEDS), Special issue: Applied parallel computing, 22 (2007), 221–256

Финальные модели спецификации

Игорь Бурдонов <igor@ispras.ru>,
Александр Косачев kos@ispras.ru

Аннотация. Работа посвящена исследованию формальных методов тестирования соответствия (конформности) исследуемой системы требованиям, заданным в форме спецификации. Такое тестирование основано на семантике взаимодействия, которая определяет тестовые возможности по управлению (заданный набор тестовых воздействий) и наблюдению действий и отказов (отсутствие действий). Допускаются также ненаблюдаемые (внутренние) действия. Семантика параметризуется семействами наблюдаемых и ненаблюдаемых отказов. Вводится разрушение – запрещённое действие, которого следует избегать при взаимодействии. Определяется понятие безопасного тестирования, при котором не возникают ненаблюдаемые отказы и разрушение, и тестовые воздействия не подаются при дивергенции (бесконечной последовательности ненаблюдаемых действий). На этой основе определяются реализационная гипотеза о безопасности и безопасная конформность, а также генерация полного набора тестов по спецификации.

В работе исследуются различные модели для описания спецификационных требований. Наиболее распространенной моделью является система помеченных переходов – LTS (Labelled Transition System). В то же время для рассматриваемой семантики взаимодействия существенны только трассы (последовательности наблюдений), но не состояния LTS. Поэтому естественной оказывается трассовая модель как множество трасс LTS. Такая семантика позволяет определять только конформности типа редукции, в отличие от конформностей типа симуляций, для проверки которых требуется дополнительная тестовая возможность – опрос состояния реализации [10],[11],[12],[19],[22]. При безопасном тестировании тесты генерируются по безопасным трассам спецификации, для прохождения которых используются только безопасные тестовые воздействия.

Целью данной работы является выделение подмножества трасс спецификации, достаточного для генерации полного набора тестов. Такое подмножество мы назвали финальной трассовой моделью спецификации. С другой стороны, LTS-модель удобна тем, что является способом конечного представления регулярных множеств трасс. Для представления финальной трассовой модели в работе предлагается разновидность LTS, названная финальной RTS (Refusal Transition System). Переходы по наблюдаемым отказам задаются явно (эти отказы входят в алфавит RTS). Такая модель обладает целым рядом полезных для генерации тестов свойств: 1) она детерминирована, 2) трасса наблюдений безопасна тогда и только тогда, когда она заканчивается в нетерминальном состоянии, где нет разрушения, 3) тестовое воздействие безопасно после трассы тогда и только тогда, когда оно безопасно в конечном состоянии трассы, то есть в этом состоянии нет дивергенции, тестовое воздействие не вызывает разрушения и ненаблюдаемых отказов .

В работе предложены алгоритмы преобразования LTS-модели в финальную RTS-

модель и определены достаточные условия построения конечной RTS за конечное время.

Ключевые слова: Семантика взаимодействия, отказы, разрушение, дивергенция, конформность, безопасное тестирование, трассы, LTS, генерация тестов.

1. Теория конформности

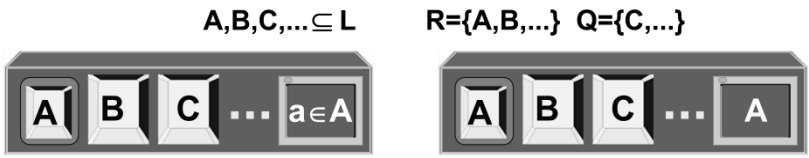
1.1. Семантика взаимодействия и безопасное тестирование

Верификация конформности понимается как проверка соответствия исследуемой системы заданным требованиям. В модельном мире система отображается в реализационную модель, требования – в спецификационную модель, а их соответствие – в бинарное отношение конформности. Если требования выражены в терминах взаимодействия системы с окружающим миром, возможно тестирование как проверка конформности в процессе тестовых экспериментов, когда тест подменяет собой окружение системы. В этом случае само отношение конформности и его тестирование основаны на той или иной модели взаимодействия.

Мы рассматриваем семантики взаимодействия, которые определяются только внешним, наблюдаемым поведением системы и не учитывают её внутреннее устройство, которое на уровне модели отображается понятием *состояния*. Мы можем наблюдать только такое поведение реализации, которое, во-первых, «спровоцировано» тестом (управление) и, во-вторых, наблюдаемо во внешнем взаимодействии. Такое взаимодействие может моделироваться с помощью, так называемой, машины тестирования [2],[4],[5],[20],[21],[23]. Она представляет собой «чёрный ящик», внутри которого находится реализация (рис. 1). Управление сводится к тому, что оператор машины, выполняя тест (понимаемый как инструкция оператору), нажимает кнопки на клавиатуре машины, «разрешая» реализации выполнять те или иные действия, которые могут им наблюдаться. Наблюдения (на «дисплее» машины) бывают двух типов: наблюдение некоторого *внешнего (наблюдаемого) действия*, разрешённого оператором и выполняемого реализацией, и наблюдение *отказа* как отсутствия каких бы то ни было наблюдаемых действий из числа тех, что разрешены нажатой кнопкой.

Подчеркнём, что при управлении оператор разрешает реализации выполнять именно множество действий, а не обязательно одно действие. Будем считать, что каждой кнопке соответствует своё множество разрешаемых действий, и оператор нажимает только одну кнопку. После наблюдения (действия или отказа) кнопка отжимается, и все внешние действия запрещаются. Далее оператор может нажать другую (или ту же самую) кнопку.

Кроме внешних действий реализация может совершать *внутренние* (*ненаблюдаемые*) действия, обозначаемые символом τ, которые считаются всегда разрешенными.



Машина тестирования

Тестовые возможности определяются тем, какие «кнопочные» множества есть в машине, а также для каких кнопок возможно наблюдение отказа. Тем самым, семантика взаимодействия определяется алфавитом внешних действий **L** и двумя наборами кнопок (соответствующих им множеств действий) машины тестирования: с наблюдением соответствующих отказов – семейство $\mathbf{R}\subseteq P(\mathbf{L})$ и без наблюдения отказа – семейство $\mathbf{Q}\subseteq P(\mathbf{L})$. Предполагается, что $\mathbf{L}\cap P(\mathbf{L})=\varnothing$ (действия и отказы «не путаются» между собой), $\mathbf{R}\cap\mathbf{Q}=\varnothing$ (отказ либо наблюдаемый, либо ненаблюдаемый), $\cup\mathbf{R}\cup\cup\mathbf{Q}=\mathbf{L}$ (каждое внешнее действие разрешается хотя бы одной кнопкой и каждая кнопка разрешает действия только из алфавита внешних действий). Такую семантику мы называем **R**/**Q**-семантикой [2],[5],[7],[9],[13],[14],[15].

Множество наблюдений (действий и отказов), разрешаемых кнопкой P, обозначим $\boldsymbol{obs}(P) \triangleq P \cup \{P|P\in\mathbf{R}\}$. Наоборот, множество кнопок, разрешающих наблюдение u (действие и отказ), обозначим $\boldsymbol{but}(u) \triangleq \{P\in\mathbf{R}\cup\mathbf{Q}|u\in\boldsymbol{obs}(P)\}$.

В качестве примера можно привести хорошо известную семантику отношения ***ioco*** [24],[25], в которой действия делятся на стимулы и реакции. Каждый стимул x разрешается одной **Q**-кнопкой {x}, а все реакции разрешаются одной **R**-кнопкой, обозначаемой δ.

Для выполнимости любого действия необходимо, чтобы оно было определено в реализации и разрешено оператором. Если этого условия также и достаточно, то есть на выполнение выбирается любое из таких действий недетерминированным образом, то говорят, что в системе нет приоритетов [21]. Здесь мы ограничимся системами без приоритетов. Тестирование систем с приоритетами рассмотрено в наших работах [6],[8].

Предполагается, что любая конечная последовательность любых действий (как внешних, так и внутренних) совершается за конечное время, а бесконечная – за бесконечное время. Также предполагается, что «передача» тестового воздействия (нажатие кнопки) от машины тестирования в реализацию и наблюдения от реализации на дисплей машины выполняются за конечное время.

Бесконечная последовательность τ-действий называется *дивергенцией* и обозначается символом Δ. Дивергенция сама по себе не опасна, но при попытке выхода из неё, когда оператор нажимает любую кнопку, получение наблюдения не гарантируется, поскольку реализация может бесконечно долго

выполнять только внутренние (ненаблюдаемые) действия. Поэтому оператор не может ни продолжать тестирование, ни закончить его.

При отсутствии дивергенции указанные выше предположения гарантируют, что если после нажатия кнопки реализация выполняет некоторое внешнее действие, разрешаемое этой кнопкой, то через конечное время это действие будет наблюдаться на дисплее машины тестирования.

Эти же предположения часто используются для реализации наблюдения **R**-отказа при отсутствии дивергенции, но в усиленном варианте: время выполнения каждого действия, разрешаемого кнопкой, вместе с возможными предшествующими ему внутренними действиями не только конечно, но и ограничено. В этом случае вводится тайм-аут, истечение которого без наблюдения действия трактуется как отказ (при условии отсутствия дивергенции). Важно отметить, что это не единственный возможный способ реализации наблюдения отказа.

В любом случае предполагается, что при отсутствии дивергенции после нажатия **R**-кнопки через конечное время оператор наблюдает или разрешенное этой кнопкой внешнее действие или соответствующий отказ. Однако при нажатии **Q**-кнопки, если в реализации возможен отказ, то, поскольку этот отказ не наблюдаем, оператор не знает, нужно ли ему ждать наблюдения внешнего действия или такого действия не будет, поскольку возник отказ. Поэтому оператор не может ни продолжать тестирование, ни закончить его.

Кроме этого мы вводим [1],[2],[3],[4],[5] специальное, всегда разрешенное действие, которое называем *разрушением* и обозначаем символом γ. Оно моделирует любое поведение реализации, которое не должно допускаться во время тестирования.

Тестирование, при котором не возникает попыток выхода из дивергенции, ненаблюдаемых отказов и разрушения, называется *безопасным* [2],[5][1].

2. LTS-модель

В качестве основной модели мы используем *систему помеченных переходов* (LTS – Labelled Transition System) – ориентированный граф с выделенной начальной вершиной, дуги которого помечены некоторыми символами. Формально, LTS – это совокупность $\mathbf{S}=LTS(V_S,\mathbf{L},E_S,s_0)$, где V_S – непустое множество состояний (вершин графа), $\mathbf{L}$ – алфавит внешних действий, $E_S \subseteq V_S \times (\mathbf{L} \cup \{\tau,\gamma\}) \times V_S$ – множество переходов (помеченных дуг графа), $s_0 \in V_S$ – начальное состояние (начальная вершина графа).

Пример LTS (в ***ioco***-семантике) приведен на рис. 2 слева.

[1] В наших более ранних работах [1][2][3][4] опасной считалась сама дивергенция, а не попытка выхода из нее.

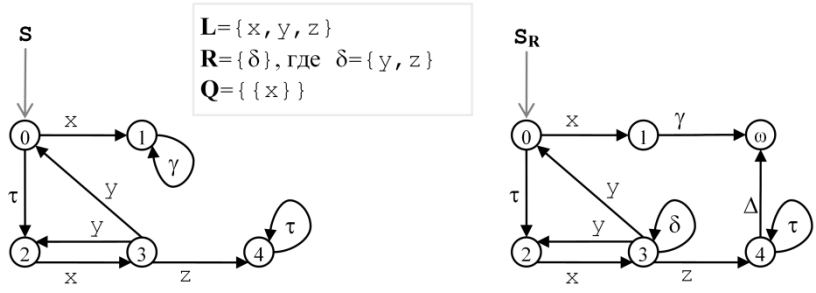


Рис. 2. Пример LTS и ее $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$*-трасс*[2]

Множество всех LTS в алфавите **L** обозначим ***LTS*(L)**. Также введем обозначения для наличия/отсутствия переходов:

$s\text{—}z\rightarrow s' \triangleq (s,z,s')\in E_S$, $\quad s\text{—}z\not\rightarrow s' \triangleq \neg(s\text{—}z\rightarrow s')$,

$s\text{—}z\rightarrow \triangleq \exists s'\ s\text{—}z\rightarrow s'$, $\quad s\text{—}z\not\rightarrow \triangleq \neg(s\text{—}z\rightarrow)$.

Там, где это не приводит к недоразумениям, будем использовать запись $s\text{—}z\rightarrow s'$ для обозначения самого перехода (s,z,s'), а не как предикат $(s,z,s')\in E_S$.

Маршрутом LTS называется последовательность смежных переходов: конец каждого перехода, кроме последнего, совпадает с началом следующего перехода. В машине тестирования выполнение LTS сводится к прохождению маршрута, каждый переход которого разрешается нажатой кнопкой. После выполнения перехода по внешнему действию, разрешаемому нажатой кнопкой, или после возникновения **R**-отказа кнопка автоматически отжимается. При этом τ- и γ-переходы всегда разрешены.

Состояние s` *достижимо из состояния* s, если в s` заканчивается маршрут, начинающийся в состоянии s. Просто *достижимое состояние* – это состояние, достижимое из начального состояния s_0. Множество состояний,

[2] При записи регулярных выражений мы для краткости будем опускать знаки конкатенации «·» для последовательностей и множеств последовательностей, запятые «,», разделяющие элементы последовательности, угловые скобки «⟨» и «⟩», отмечающие начало и конец последовательности, а также фигурные скобки «{» и «}» для множеств. Например, регулярное выражение на данном рисунке, записанное в краткой форме как $(\delta^*xy)^*(\delta^*[x[z[\Delta]]] \mid x\gamma)$, в полной форме записывается так:

$(\{\langle\delta\rangle\}^*\cdot\{\langle x,y\rangle\})^*\cdot(\{\langle\delta\rangle\}^*\cdot\{[\langle x\rangle\cdot[\langle z\rangle\cdot[\langle\Delta\rangle]]]\} \mid \{\langle x,\gamma\rangle\})$.

достижимых из состояния s, будем обозначать ***der***(s). Определим ***der***(**S**)≜***der***(s_0).

Состояние *терминально*, если из него не выходят никакие переходы, и *стабильно*, если из него не выходят τ- и γ-переходы. Отказ P порождается стабильным состоянием, из которого не выходят переходы по действиям из P. Состояние *дивергентно*, если в нем начинается бесконечный τ-маршрут (маршрут из τ-переходов); в противном случае состояние *конвергентно*.

3. Тестовые истории и трассы LTS

3.1. Тестовые истории

Для взаимодействия, основанного на наблюдениях, единственным результатом тестового эксперимента является чередующаяся последовательность кнопок (тестовых воздействий) и наблюдений, которую будем называть (тестовой) *историей*. Дивергенция и разрушение считаются условно-наблюдаемыми действиями: хотя они не должны возникать при безопасном тестировании, но для полноты моделирования должны присутствовать в историях, отмечая те их них, после которых возможны дивергенция или разрушение. Так как нас не интересует поведение системы после дивергенции или разрушения, символы Δ и γ могут быть только последними элементами историй. Любое другое наблюдение u (внешнее действие или **R**-отказ) должно разрешаться непосредственно предшествующей ему в истории кнопкой P∈***but***(u). Заметим, что по этому определению любой префикс истории является историей.

Подпоследовательность истории, состоящая только из наблюдений (включая Δ и γ), называется *трассой* (этой истории). В общем случае будем называть ***A***-трассой последовательность в алфавите ***A***. Трасса истории – это **L**∪**R**∪{Δ,γ}-трасса.

Для систем без приоритетов важны только трассы, поскольку возможность или невозможность появления данного наблюдения после некоторой трассы определяется только тем, что нажимаемая кнопка разрешает данное наблюдение, и не зависит от того, какие еще наблюдения она разрешает или запрещает. Для данной тестируемой системы без приоритетов множество ее историй однозначно восстанавливается по множеству ее трасс.

Как было сказано выше, в машине тестирования выполнение LTS сводится к прохождению маршрута, каждый переход которого разрешается нажатой кнопкой. При этом на дисплее машины мы наблюдаем последовательность внешних действий, которыми помечены выполняемые переходы, и возникающие **R**-отказы в стабильных состояниях. Если LTS находится в конвергентном состоянии s, когда нажимается кнопка P, то через конечное время либо выполняется переход по действию z∈***obs***(P), то есть z∈P, либо происходит отказ P. Этот отказ P наблюдается, то есть P∈***obs***(P), если P∈**R**. Поскольку τ-переходы всегда разрешены, может оказаться, что переход по

действию z выполняется не из состояния s, а из другого состояния s`, достижимого из s по цепочке τ-переходов. Аналогично, поскольку при возникновении **R**-отказа P LTS должна находиться в стабильном состоянии, это состояние совпадает с состоянием s только в том случае, когда состояние s стабильно, а в противном случае отказ происходит в стабильном состоянии s`, достижимом из s по цепочке τ-переходов. Если состояние s дивергентно, то после нажатия кнопки P никаких внешних действий и отказов может не быть, если бесконечно долго будут выполняться только τ-переходы.

Простые трассы LTS

Простой трассой LTS **S** будем называть последовательность σ пометок на переходах маршрута с пропуском символов τ. Простая трасса LTS – это **L**∪{γ}-трасса. Множество простых трасс, начинающихся (то есть маршруты которых начинаются) в состоянии s будем обозначать ***tr***(s,**S**).

Через s ***after*** σ обозначим множество состояний, в которых заканчивается простая трасса σ, начинающаяся в состоянии s, то есть заканчиваются все маршруты с простой трассой σ, начинающиеся в состоянии s. Распространим оператор ***after*** на множество A состояний обычным образом: A ***after*** σ ≜ {s ***after*** σ|s∈A}, результатом является множество множеств состояний LTS. По умолчанию, будем считать, что простая трасса начинается в начальном состоянии LTS, и обозначать **S** ***after*** σ ≜ s_0 ***after*** σ, ***tr***(**S**) ≜ ***tr***(s_0,**S**).

Введем следующие обозначения для состояний s и s` и трассы σ:

s=σ⇒s` ≜ s`∈(s ***after*** σ), s=σ⇏s` ≜ ¬(s=σ⇒s`),

s⇒s` ≜ s=$\epsilon$⇒s`, где ϵ пустая трасса, s⇏s` ≜ ¬(s⇒s`),

s=σ⇒ ≜ ∃s` s=σ⇒s`, s=σ⇏ ≜ ¬(s=σ⇒).

Распространим эти обозначения на множества состояний A и A`:

A=σ⇒A` ≜ A`=(A ***after*** σ)≠∅, A=σ⇏A` ≜ ¬(A=σ⇒A`),

A⇒A` ≜ A=$\epsilon$⇒A`, A⇏A` ≜ ¬(A⇒A`),

A=σ⇒ ≜ ∃A` A=σ⇒A`, A=σ ⇏ ≜ ¬(A=σ⇒).

LTS *детерминирована*, если каждая ее простая трасса заканчивается ровно в одном состоянии. Очевидно, что в детерминированной LTS нет τ-переходов из достижимых состояний.

L∪**R**∪{Δ,γ}***-трассы LTS***

Последовательность наблюдений, которая может быть получена при взаимодействии с LTS в **R**/**Q**-семантике, то есть последовательность не только действий, но и **R**-отказов, является **L**∪**R**∪{Δ,γ}-трассой. Для определения таких трасс LTS **S** добавим в каждом ее стабильном состоянии виртуальные петли s—R→s, помеченные **R**-отказами, порождаемыми в этом состоянии, добавим новое терминальное состояние ω, перенаправим в это состояние все γ-переходы, а также проведем в него Δ-переходы из дивергентных состояний.

Формально для любого заданного семейства множеств $\mathbf{R}\subseteq P(\mathbf{L})$ это преобразование $\mathbf{S}\to\mathbf{S_R}$ дает LTS $\mathbf{S_R}=LTS(V_S\cup\{\omega\},\mathbf{L}\cup\mathbf{R}\cup\{\Delta\},E_{\mathbf{R}},s_0)$, где $\omega\notin V_S$, а множество переходов $E_{\mathbf{R}}$ определяется как минимальное множество, порождаемое следующими правилами вывода: $\forall s,s`\in V_S\ \forall z\in\mathbf{L}\cup\{\tau\}\ \forall R\in\mathbf{R}$

$s\text{—}z\to s` \quad\vdash\quad s\text{—}z\to s`,$

$\forall z\in R\cup\{\tau,\gamma\}\ s\text{—}z\not\to \quad\vdash\quad s\text{—}R\to s,$

$s\text{—}\gamma\to \quad\vdash\quad s\text{—}\gamma\to\omega,$

s *дивергентно* $\quad\vdash\quad s\text{—}\Delta\to\omega.$

Пример LTS **S**, LTS $\mathbf{S_R}$ и множества $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс LTS **S** приведен на рис. 2.

Формально $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трассой LTS **S** будем называть простую трассу LTS $\mathbf{S_R}$. Множество $\boldsymbol{tr}(\mathbf{S_R})$ всех $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс LTS **S** называется *(трассовой)* **R**-*моделью*, когда по умолчанию предполагается заданным алфавит **L**, или просто *трассовой моделью*, когда по умолчанию предполагаются заданными как алфавит **L**, так и семейство $\mathbf{R}\subseteq P(\mathbf{L})$). Для $\mathbf{R}=P(\mathbf{L})$ получаем *полную трассовую модель* – она содержит все отказы, порождаемые стабильными состояниями. Полная трассовая модель (как и LTS) описывает как устроена система «на самом деле», а подмножество ее $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс, то есть трассовая **R**-модель, – это «взгляд» на систему, определяемый тестовыми возможностями по управлению и наблюдению, которые описываются **R**/**Q**-семантикой, и безопасностью тестирования (отсутствия в трассах ненаблюдаемых отказов).

Распространим LTS-обозначения, использующие трассы, на $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трассы. Такими обозначениями являются: $s=\sigma\Rightarrow s`$, а также производные от этого обозначения $s=\sigma\Rightarrow$, $s=\sigma\not\Rightarrow s`$, $s=\sigma\not\Rightarrow$, s ***after*** σ и **S** ***after*** σ. Обозначение $s=\sigma\Rightarrow s`$ имеет один и тот же смысл для LTS **S** и $\mathbf{S_R}$, если трасса σ содержит только внешние действия. Если σ содержит отказы или заканчивается Δ-символом, двойная стрелка, очевидно, применяется к LTS $\mathbf{S_R}$, поскольку в LTS **S** нет переходов по Δ-символу и отказам. Двусмысленность возможна лишь в том случае, когда σ не содержит отказов и символа Δ и заканчивается разрушением γ, что происходит из-за того, что все γ-переходы в LTS $\mathbf{S_R}$ перенаправляются в состояние ω. Мы будем считать, что в этом случае имеется в виду LTS $\mathbf{S_R}$. Тем самым, мы переопределяем двойную стрелку $s=\sigma\Rightarrow s`$ и производные обозначения для LTS **S** и трасс без отказов и Δ-символа, заканчивающихся разрушением.

Состояния s и s` называют *эквивалентными*, если в них начинаются одни и те же простые трассы: $\boldsymbol{tr}(s,\mathbf{S})=\boldsymbol{tr}(s`,\mathbf{S})$. Множества состояний A и A` будем

называть *эквивалентными* и обозначать $A \sim_T A`$, если в их состояниях начинаются одни и те же простые трассы: $\cup\{\boldsymbol{tr}(s,\mathbf{S}) | s \in A\} = \cup\{\boldsymbol{tr}(s`,\mathbf{S}) | s` \in A`\}$.

4. Трассовая модель

4.1. Сравнение LTS-модели и трассовой модели

Сравним LTS-модель и трассовую модель, определенную выше как множество $\mathbf{L} \cup \mathbf{R} \cup \{\Delta,\gamma\}$-трасс LTS. LTS-модель более «наглядна», чем трассовая модель. Формально это означает, что любой граф с выделенной начальной вершиной и дугами, помеченными символами из алфавита $\mathbf{L} \cup \{\tau,\gamma\}$, является LTS-моделью в алфавите **L**. В то же время далеко не любое множество $\mathbf{L} \cup \mathbf{R} \cup \{\Delta,\gamma\}$-трасс является трассовой моделью: как будет показано ниже, оно должно обладать определенным набором нетривиальных свойств. Кроме того, LTS-модель является способом конечного представления регулярных трассовых моделей, в том числе бесконечных, то есть регулярных множеств последовательностей, являющихся трассовыми моделями.

Но с другой стороны трассовая модель имеет то преимущество перед LTS-моделью, что не содержит ничего «лишнего», поскольку представляет собой множество трасс, которые можно наблюдать при работе с системой в **R**/**Q**-семантике. Поэтому трассовая модель является наиболее естественной моделью такой системы. В частности, одной и той же трассовой модели может соответствовать несколько LTS с одним и тем же множеством $\mathbf{L} \cup \mathbf{R} \cup \{\Delta,\gamma\}$-трасс, и эти LTS неразличимы при взаимодействии с ними в **R**/**Q**-семантике.

В этом подразделе мы дадим независимое от LTS определение трассовой модели как множества трасс, удовлетворяющего некоторому набору свойств. Иными словами, мы покажем, что множество трасс является трассовой моделью тогда и только тогда, когда оно удовлетворяет этому набору свойств. Сначала дадим несколько определений и обозначений.

1. Свойства трасс и операции над трассами

Пусть $\mathbf{R} \subseteq P(\mathbf{L})$ произвольное семейство подмножеств алфавита.

Трассу будем называть *допустимой*, если все ее элементы, кроме, быть может, последнего, отличны от дивергенции и разрушения.

Трассу будем называть *согласованной*, если после любой непустой последовательности отказов в трассе не следует ни дивергенция, ни разрушение, ни какое-либо внешнее действие, принадлежащее какому-либо отказу, входящему в эту последовательность.

Трассу будем называть **R**-*конвергентной* во множестве трасс $\boldsymbol{\Sigma}$, если она содержит или продолжается в $\boldsymbol{\Sigma}$ разрушением или дивергенцией, а в противном случае для каждого **R**-отказа продолжается в $\boldsymbol{\Sigma}$ этим отказом или каким-либо внешним действием, принадлежащим этому отказу.

Определим три операции над трассами.

Операция $\boldsymbol{p_{re}}$ – взятие префикса трассы: $\mu\cdot\lambda\text{—}\boldsymbol{p_{re}}\!\rightarrow\!\mu$. Введем отношение «является префиксом»: $\mu\le\sigma \triangleq \exists\lambda\ \sigma=\mu\cdot\lambda$. Множество трасс будем называть *префикс-замкнутым*, если оно вместе с каждой трассой содержит все ее префиксы, то есть все трассы, получаемые из данной по $\boldsymbol{p_{re}}$-операциям. Через $\boldsymbol{p_{re}}(\sigma)$ обозначим префикс-замыкание трассы σ как множество всех ее префиксов, то есть замыкание по операции $\boldsymbol{p_{re}}$ ($\boldsymbol{p_{re}}$-замыкание) как множество трасс, получаемых из трассы σ всеми возможными применениями операции:

$\boldsymbol{p_{re}}(\sigma) \triangleq \{\mu|\mu\le\sigma\} = \{\mu|\sigma\text{—}\boldsymbol{p_{re}}\!\rightarrow\!\mu\}$.

Распространим операцию $\boldsymbol{p_{re}}$ на множество $\boldsymbol{\Sigma}$ трасс обычным образом: $\boldsymbol{p_{re}}(\boldsymbol{\Sigma})\triangleq\{\boldsymbol{p_{re}}(\sigma)|\sigma\in\boldsymbol{\Sigma}\}$, результатом такой операции является семейство множеств трасс. Множество трасс $\boldsymbol{\Sigma}$ $\boldsymbol{p_{re}}$-замкнуто тогда и только тогда, когда $\cup\boldsymbol{p_{re}}(\boldsymbol{\Sigma})=\boldsymbol{\Sigma}$. Более того, для любого множества трасс $\boldsymbol{\Sigma}$ множество $\cup\boldsymbol{p_{re}}(\boldsymbol{\Sigma})$ $\boldsymbol{p_{re}}$-замкнуто и является наименьшим (по вложенности) $\boldsymbol{p_{re}}$-замкнутым множеством, содержащим $\boldsymbol{\Sigma}$.

Операция $\boldsymbol{d}$ – удаление $\mathbf{R}$-отказа P из трассы $\mu\cdot\langle P\rangle\cdot\lambda$: $\mu\cdot\langle P\rangle\cdot\lambda\text{—}\boldsymbol{d}\!\rightarrow\!\mu\cdot\lambda$. Множество трасс будем называть $\boldsymbol{d}$-*замкнутым*, если оно вместе с каждой трассой содержит все трассы, получаемые из данной по $\boldsymbol{d}$-операциям. Через $\boldsymbol{d}(\sigma)$ обозначим замыкание трассы σ по операции $\boldsymbol{d}$ ($\boldsymbol{d}$-замыкание) как множество трасс, получаемых из трассы σ всеми возможными конечными цепочками применения операции $\boldsymbol{d}$:

$\boldsymbol{d}(\sigma) \triangleq \{\sigma`|\exists n\ \exists\sigma_1,\sigma_2,\ldots,\sigma_n\ \sigma_n=\sigma` \ \&\ \sigma_1\text{—}\boldsymbol{d}\!\rightarrow\!\sigma_2\text{—}\boldsymbol{d}\!\rightarrow\!\ldots\text{—}\boldsymbol{d}\!\rightarrow\!\sigma_n\}$.

Распространим операцию $\boldsymbol{d}$ на множество $\boldsymbol{\Sigma}$ трасс обычным образом: $\boldsymbol{d}(\boldsymbol{\Sigma})\triangleq\{\boldsymbol{d}(\sigma)|\sigma\in\boldsymbol{\Sigma}\}$, результатом такой операции является семейство множеств трасс. Объединение множество этого семейства обозначим $\boldsymbol{D}(\boldsymbol{\Sigma})=\cup\boldsymbol{d}(\boldsymbol{\Sigma})$. Множество трасс $\boldsymbol{\Sigma}$ $\boldsymbol{d}$-замкнуто тогда и только тогда, когда $\boldsymbol{D}(\boldsymbol{\Sigma})=\boldsymbol{\Sigma}$. Более того, для любого множества трасс $\boldsymbol{\Sigma}$ множество $\boldsymbol{D}(\boldsymbol{\Sigma})$ $\boldsymbol{d}$-замкнуто и является наименьшим (по вложенности) $\boldsymbol{d}$-замкнутым множеством, содержащим $\boldsymbol{\Sigma}$.

Операция $\boldsymbol{i_R}$ – вставка $\mathbf{R}$-отказа P в трассу $\mu\cdot\lambda$ после трассы μ при условии, что трасса μ заканчивается каким-нибудь отказом и не продолжается во множестве трасс $\boldsymbol{\Sigma}$ дивергенцией, разрушением и действиями из P: $\mu\cdot\lambda\text{—}\boldsymbol{i_R}\!\rightarrow\!\mu\cdot\langle P\rangle\cdot\lambda$. Множество $\boldsymbol{\Sigma}$ трасс будем называть $\boldsymbol{i_R}$-*замкнутым*, если оно вместе с каждой трассой содержит все трассы, получаемые из данной по $\boldsymbol{i_R}$-операциям. Через $\boldsymbol{i_R}(\sigma)$ обозначим замыкание трассы σ по операции $\boldsymbol{i_R}$ как множество трасс, получаемых из трассы σ с помощью всех возможных одновременных вставок любого (в том числе нулевого) числа отказов:

$\boldsymbol{i_R}(\sigma)\triangleq \{\sigma`|\exists n\ \exists\mu_0,\ldots,\mu_n\ \exists R_1,\ldots,R_n\ \sigma=\mu_0\cdot\ldots\cdot\mu_n$

$\&\ \sigma`=\mu_0\cdot\langle R_1\rangle\cdot\mu_1\cdot\ldots\cdot\langle R_n\rangle\cdot\mu_n$

$\&\ \forall i\in[1..n]\ \sigma\text{—}\boldsymbol{i_R}\!\rightarrow\!\mu_0\cdot\ldots\cdot\mu_{i-1}\cdot\langle R_i\rangle\cdot\mu_i\cdot\ldots\cdot\mu_n\}$.

Распространим операцию $\boldsymbol{i}_{\mathbf{R}}$ на множество $\mathbf{\Sigma}$ трасс обычным образом: $\boldsymbol{i}_{\mathbf{R}}(\mathbf{\Sigma})\triangleq\{\boldsymbol{i}_{\mathbf{R}}(\sigma)|\sigma\in\mathbf{\Sigma}\}$, результатом такой операции является семейство множеств трасс.

Заметим, что $\boldsymbol{i}_{\mathbf{R}}$-операция над трассой, в отличие от $\boldsymbol{p}_{re}$- и $\boldsymbol{d}$-операций, зависит не только от этой трассы, но и от множества трасс $\mathbf{\Sigma}$, которому она принадлежит. Так же, как для $\boldsymbol{p}_{re}$- и $\boldsymbol{d}$-операций, множество трасс $\mathbf{\Sigma}$ $\boldsymbol{i}_{\mathbf{R}}$-замкнуто тогда и только тогда, когда $\cup\boldsymbol{i}_{\mathbf{R}}(\mathbf{\Sigma})=\mathbf{\Sigma}$. Также для любого множества трасс $\mathbf{\Sigma}$ множество $\cup\boldsymbol{i}_{\mathbf{R}}(\mathbf{\Sigma})$ $\boldsymbol{i}_{\mathbf{R}}$-замкнуто, однако, вообще говоря, оно не является не только наименьшим, но и минимальным $\boldsymbol{i}_{\mathbf{R}}$-замкнутым множеством, содержащим $\mathbf{\Sigma}$. В качестве примера рассмотрим множество $\mathbf{A}=\cup\boldsymbol{p}_{re}(\{\sigma_1,\sigma_2\})$, где $\sigma_1=\langle A,x,A,b,\Delta\rangle$, $\sigma_2=\langle A,B,x,A\rangle$, $A=\{a\}$, $B=\{b\}$ и $\mathbf{R}=\{A,B\}$, $\mathbf{Q}=\{\{x\}\}$. Тогда $\boldsymbol{i}_{\mathbf{R}}(\sigma_1)$ и $\boldsymbol{i}_{\mathbf{R}}(\sigma_2)$ это множества трасс, которые могут быть записаны регулярными выражениями $A(AB)^*xAA^*b$ и $A(AB)^*B(AB)^*xA(AB)^*$, соответственно. Множество $\mathbf{A}_1=\cup\boldsymbol{p}_{re}(\boldsymbol{i}_{\mathbf{R}}(\sigma_1))=[A(AB)^*[x[AA^*[b[\Delta]]]]]$ содержит множество $\mathbf{A}$ и $\boldsymbol{i}_{\mathbf{R}}$-замкнуто, но не содержит трассы ABxAB, принадлежащей $\boldsymbol{i}_{\mathbf{R}}(\sigma_2)$ и, следовательно, принадлежащей $\cup\,\boldsymbol{i}_{\mathbf{R}}(\mathbf{A})$. Тем самым $\cup\boldsymbol{i}_{\mathbf{R}}(\mathbf{A})\not\subseteq\mathbf{A}_1$.

Можно было бы выбрать другой, второй, способ определения $\boldsymbol{i}_{\mathbf{R}}$-замыкания трассы σ как множество трасс, получаемых из трассы σ с помощью всех возможных одиночных вставок отказов. Однако в этом случае множество $I_1=\cup\boldsymbol{i}_{\mathbf{R}}(\mathbf{\Sigma})$, хотя и содержит множество $\mathbf{\Sigma}$, но может быть не $\boldsymbol{i}_{\mathbf{R}}$-замкнуто. Например, если $\mathbf{R}=\{\{a\}\}$ и $\mathbf{\Sigma}=\{\langle\{a\}\rangle\}$, то одиночные вставки дадут только одну новую трассу $\langle\{a\},\{a\}\rangle$. Полученное множество, состоящее из двух трасс $\langle\{a\}\rangle$ и $\langle\{a\},\{a\}\rangle$, не $\boldsymbol{i}_{\mathbf{R}}$-замкнуто, поскольку не содержит трассы $\langle\{a\},\{a\},\{a\}\rangle$, хотя одновременная вставка нескольких отказов дает такую трассу (как и трассы с любым ненулевым числом отказов $\{a\}$).

Поэтому при одиночной вставке отказов нужно снова взять $\boldsymbol{i}_{\mathbf{R}}$-замыкание $I_2=\cup\boldsymbol{i}_{\mathbf{R}}((\cup\boldsymbol{i}_{\mathbf{R}}(\mathbf{\Sigma})))$, и так далее. При этом мы получим, вообще говоря, бесконечную последовательность расширяющихся множеств $I_1\subseteq I_2\subseteq\ldots$. Для того, чтобы окончательно получить $\boldsymbol{i}_{\mathbf{R}}$-замкнутое множество, содержащее исходное множество $\mathbf{\Sigma}$, нужно взять предел этой последовательности, то есть объединение $\cup\{I_n|n=1,2\ldots\}$. Результат будет тот же, что при первом способе, то есть при однократном $\boldsymbol{i}_{\mathbf{R}}$-замыкании множества $\mathbf{\Sigma}$ с одновременной вставкой нескольких отказов. Понятно, что первый способ предпочтительнее, поэтому мы его и выбрали.

Множество всех отказов в конце трассы σ (то есть отказов, после которых в σ следуют только отказы) будем обозначать $\boldsymbol{Ip}(\sigma)$.

2. Характеристические свойства трассовой модели

Теорема 1: Множество $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс $\mathbf{\Sigma}$ является $\mathbf{R}$-моделью (то есть множеством $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс некоторой LTS в алфавите $\mathbf{L}$) тогда и только тогда, когда оно не пусто, префикс-замкнуто $\cup\boldsymbol{p_{re}}(\mathbf{\Sigma})=\mathbf{\Sigma}$ и удовлетворяет следующим требованиям:

- допустимость: все трассы $\mathbf{\Sigma}$ допустимы,
- согласованность: все трассы $\mathbf{\Sigma}$ согласованы,
- $\mathbf{R}$-конвергентность: все трассы $\mathbf{\Sigma}$ $\mathbf{R}$-конвергентны в $\mathbf{\Sigma}$,
- замкнутость: $\mathbf{\Sigma}$ $\boldsymbol{d}$-замкнуто: $\boldsymbol{D}(\mathbf{\Sigma})=\mathbf{\Sigma}$,
- $\mathbf{R}$-полнота: $\mathbf{\Sigma}$ $\boldsymbol{i}_{\mathbf{R}}$-замкнуто: $\cup\boldsymbol{i}_{\mathbf{R}}(\mathbf{\Sigma})=\mathbf{\Sigma}$.

□Доказательство этого утверждения см. в [5], теорема 14.

В дальнейшем для заданной $\mathbf{R}/\mathbf{Q}$-семантики в качестве трассовой модели спецификации мы будем использовать $\mathbf{R}$-модель, а в качестве трассовой модели реализации мы будем использовать $\mathbf{R}\cup\mathbf{Q}$-модель.

5. Безопасное тестирование и безопасная конформность

При безопасном тестировании будут проходиться только $\mathbf{L}\cup\mathbf{R}$-трассы, причем нажиматься будут только кнопки, которые считаются безопасными после них. Это предполагает формальное определение на уровне модели отношения безопасности «кнопка P безопасна во множестве трасс $\mathbf{N}$ после $\mathbf{L}\cup\mathbf{R}$-трассы σ».

5.1. Произвольное отношение безопасности

Пусть заданы $\mathbf{R}/\mathbf{Q}$-семантика, непустое префикс-замкнутое множество трасс $\mathbf{N}$ и произвольное отношение $\boldsymbol{rel}\subseteq(\mathbf{R}\cup\mathbf{Q})\times(\mathbf{N}\cap(\mathbf{L}\cup\mathbf{R})^{*})$, связывающее кнопки с $\mathbf{L}\cup\mathbf{R}$-трассами из $\mathbf{N}$, которое будем называть отношением безопасности. Запись P $\boldsymbol{rel}$ $\mathbf{N}$ $\boldsymbol{after}$ σ читается как «кнопка $P\in\mathbf{R}\cup\mathbf{Q}$ *безопасна по отношению* $\boldsymbol{rel}$ в $\mathbf{N}$ после трассы σ.

Будем говорить, что отказ $P\in\mathbf{R}$ *безопасен по отношению* $\boldsymbol{rel}$ в $\mathbf{N}$ после трассы σ и писать P $\boldsymbol{rel}$ $\mathbf{N}$ $\boldsymbol{after}$ σ, если кнопка P безопасна по отношению $\boldsymbol{rel}$ в $\mathbf{N}$ после σ.

Будем говорить, что действие $z\in\mathbf{L}$ *безопасно по отношению* $\boldsymbol{rel}$ в $\mathbf{N}$ после трассы σ и писать z $\boldsymbol{rel}$ $\mathbf{N}$ $\boldsymbol{after}$ σ, если оно разрешается такой кнопкой $P\in\boldsymbol{but}(z)$, что P $\boldsymbol{rel}$ $\mathbf{N}$ $\boldsymbol{after}$ σ.

Соответственно, будем говорить, что кнопка или наблюдение $u\in\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}$ опасны по отношению $\boldsymbol{rel}$ в $\mathbf{N}$ после $\mathbf{L}\cup\mathbf{R}$-трассы $\sigma\in\mathbf{N}$ и писать u $\boldsymbol{rel}$ $\mathbf{N}$ $\boldsymbol{after}$ σ, если они не являются безопасными по отношению $\boldsymbol{rel}$ в $\mathbf{N}$ после σ.

Будем считать, что, кроме отношения ***rel***, указано, является пустая трасса безопасной или опасной. Тогда множество всех $\mathbf{L}\cup\mathbf{R}$-трасс по отношению ***rel*** разбивается на три множества трасс: безопасных, опасных и остальных.

$\mathbf{L}\cup\mathbf{R}$-трассу $\sigma\in\mathbf{N}$ будем называть *безопасной по отношению* ***rel***, если пустая трасса безопасна и для каждого префикса $\mu\cdot\langle u\rangle\le\sigma$ наблюдение u ***rel*** $\mathbf{N}$ ***after*** μ. Множество безопасных по отношению ***rel*** трасс, как правило, будем обозначать ***Rel***$(\mathbf{N})$. Это множество трасс, очевидно, префикс-замкнуто и пусто, если пустая трасса опасна.

$\mathbf{L}\cup\mathbf{R}$-трассу σ будем называть *опасной по отношению* ***rel***, если пустая трасса опасна или трасса σ имеет вид $\sigma=\mu\cdot\langle u\rangle\cdot\lambda$, где $\mu\in$***Rel***$(\mathbf{N})$, а u ***rel*** $\mathbf{N}$ ***after*** μ. Заметим, что по этому определению опасная трасса может как принадлежать, так и не принадлежать множеству $\mathbf{N}$. Множество опасных по отношению ***rel*** трасс, как правило, будем обозначать ***UnRel***$(\mathbf{N})$.

Если пустая трасса опасна, то все трассы опасны. В противном случае могут быть еще трассы, которые не безопасные и не опасные, такие трассы имеют вид $\mu\cdot\langle u\rangle\cdot\lambda$, где трасса $\mu\in$***Rel***$(\mathbf{N})$, u ***rel*** $\mathbf{N}$ ***after*** μ, но $\mu\cdot\langle u\rangle\notin\mathbf{N}$.

5.2. Отношение неразрушаемости

При тестировании безопасная кнопка – это, прежде всего, *неразрушающая* кнопка, под которой мы понимаем кнопку, нажатие которой после $\mathbf{L}\cup\mathbf{R}$-трассы не может привести к попытке выхода из дивергенции или к разрушению после выполнения разрешаемого этой кнопкой действия. Это отношение безопасности обозначим через $\boldsymbol{safe}_{\gamma\Delta}$ и определим его формально для любого непустого префикс-замкнутого множества трасс $\mathbf{N}$:

$\forall P\in\mathbf{R}\cup\mathbf{Q}\ \forall\sigma\in\mathbf{N}\cap(\mathbf{L}\cup\mathbf{R})^*$

P $\boldsymbol{safe}_{\gamma\Delta}$ $\mathbf{N}$ ***after*** $\sigma\triangleq\sigma\cdot\langle\Delta\rangle\notin\mathbf{N}\ \&\ \forall z\in P\ \sigma\cdot\langle z,\gamma\rangle\notin\mathbf{N}$.

Пустую трассу будем считать неразрушающей (безопасной по $\boldsymbol{safe}_{\gamma\Delta}$), если разрушение невозможно с самого начала, то есть $\langle\gamma\rangle\notin\mathbf{N}$. Кнопку, наблюдение или трассу будем называть *неразрушающими* (*разрушающими*), если они безопасны (опасны) по отношению $\boldsymbol{safe}_{\gamma\Delta}$. Множества неразрушающих и разрушающих трасс обозначим $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{N})$ и $\boldsymbol{UnSafe}_{\gamma\Delta}(\mathbf{N})$, соответственно.

5.3. Отношение безопасности в реализации

Теперь определим отношение безопасности кнопок для реализации. Такая кнопка, во-первых, должна быть неразрушающей и, во-вторых, ее нажатие не должно приводить к возникновению ненаблюдаемого отказа. Для любого непустого префикс-замкнутого множества трасс $\mathbf{I}$ мы определим безопасность кнопок только после неразрушающих трасс этого множества, которые по определению являются $\mathbf{L}\cup\mathbf{R}$-трассами:

$\forall P\in\mathbf{R}\cup\mathbf{Q}\ \forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{I})$

P ***safe in*** $\mathbf{I}$ ***after*** σ $\quad\triangleq P$ ***safe***$_{\gamma\Delta}$ $\mathbf{I}$ ***after*** σ & $(P\in\mathbf{Q} \Rightarrow \sigma\cdot\langle P\rangle\notin\mathbf{I})$.

Пустую трассу будем считать безопасной по ***safe in***, если она неразрушающая (безопасна по ***safe***$_{\gamma\Delta}$), то есть $\langle\gamma\rangle\notin\mathbf{I}$. Множества безопасных и опасных по ***safe in*** трасс обозначим ***SafeIn***($\mathbf{I}$) и ***UnSafeIn***($\mathbf{I}$), соответственно. Очевидно, ***SafeIn***($\mathbf{I}$)$\subseteq$***Safe***$_{\gamma\Delta}$($\mathbf{I}$), но, вообще говоря, эти множества не равны. Это объясняется тем, что действие может быть неразрушающим, но опасным по ***safe in***, если оно разрешается только опасными по ***safe in*** **Q**-кнопками и хотя бы одна из них неразрушающая, то есть опасная по ***safe in*** из-за **Q**-отказа, продолжающего трассу.

Для заданной **R**/**Q**-семантики в качестве модели реализации будем рассматривать трассовую **R**$\cup$**Q**-модель, которая по определению непуста и префикс-замкнута, то есть на ней можно рассматривать отношение ***safe in***.

5.4. Отношение безопасности в спецификации

Теперь определим отношение безопасности кнопок для спецификации. Для произвольного непустого префикс-замкнутого множества трасс $\mathbf{\Sigma}$ мы будем определять безопасность кнопок только после неразрушающих трасс этого множества, которые по определению являются **L**$\cup$**R**-трассами. Такое отношение безопасности обозначается ***safe by*** и может быть определено, вообще говоря, различным образом. Мы сформулируем только основное условие, которому оно должно удовлетворять: A) ***safe by*** совпадает с ***safe***$_{\gamma\Delta}$ для **R**-кнопок, то есть ***safe by*** отличается от ***safe in*** только для **Q**-кнопок, B) множество безопасных по ***safe by*** трасс, принадлежащих $\mathbf{\Sigma}$, должно совпадать с множеством неразрушающих трасс, принадлежащих $\mathbf{\Sigma}$,[3] C) если кнопка безопасна по ***safe by*** после трассы, то она должна разрешать некоторое наблюдение, имеющееся в $\mathbf{\Sigma}$ после этой трассы.

Подусловие A объясняется тем, что, поскольку **R**-отказы наблюдаемы, при нажатии **R**-кнопки опасность могут представлять собой только дивергенция и разрушение после действия, разрешаемого кнопкой. Подусловие B объясняется тем, что мы хотим использовать спецификацию «по максимуму»: если ее трасса разрушающая, то, конечно, она не может быть пройдена при безопасном тестировании; но любая неразрушающая трасса, принадлежащая спецификации, может быть пройдена при безопасном тестировании. Множества безопасных и опасных по ***safe by*** трасс обозначим ***SafeBy***($\mathbf{\Sigma}$) и ***UnSafeBy***($\mathbf{\Sigma}$), соответственно. Тогда ***SafeBy***($\mathbf{\Sigma}$)=***Safe***$_{\gamma\Delta}$($\mathbf{\Sigma}$), но, вообще говоря, ***UnSafeBy***($\mathbf{\Sigma}$)$\neq$***UnSafe***$_{\gamma\Delta}$($\mathbf{\Sigma}$), поскольку наблюдение, неразрушающее после трассы, но отсутствующее в $\mathbf{\Sigma}$, может быть опасным по ***safe by***. Подусловие C нужно для того, чтобы безопасность по ***safe by*** кнопки гарантировала возможность получения хотя бы какого-нибудь конформного наблюдения, а

[3] В наших предыдущих работах [8][15][17] этот факт не отмечался, хотя формальное определение требований к отношению ***safe by*** было тем же.

такими наблюдениями, как будет определено ниже, являются только те наблюдения, которые имеются в спецификации.

Для выполнения этого основного условия отношения ***safe by*** необходимо и достаточно потребовать выполнения следующих свойств отношения. 1) **R**-кнопка безопасна по ***safe by*** тогда и только тогда, когда она неразрушающая, что совпадает с отношениями ***safe in*** и $\boldsymbol{safe}_{\gamma\Delta}$ для **R**-кнопок. 2) Если действие разрешается хотя бы одной неразрушающей кнопкой, то оно должно разрешаться какой-нибудь безопасной по ***safe by*** кнопкой. Если это неразрушающая **R**-кнопка, то она же и безопасна по ***safe by***. Но если все неразрушающие кнопки, разрешающие действие, являются **Q**-кнопками, то хотя бы одна из них должна быть объявлена безопасной по ***safe by***. 3) Если кнопка безопасна по ***safe by*** после трассы, то трасса продолжается в спецификации хотя бы одним наблюдением, разрешаемым этой кнопкой. Кроме того, пустая трасса считается безопасной по ***safe by***, если она неразрушающая (безопасная по $\boldsymbol{safe}_{\gamma\Delta}$), то есть $\langle\gamma\rangle \notin \boldsymbol{\Sigma}$:

$\epsilon \in \boldsymbol{SafeBy}(\boldsymbol{\Sigma}) \Leftrightarrow \epsilon \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Запишем эти требования формально: $\forall R \in \mathbf{R}\ \forall z \in \mathbf{L}\ \forall P \in \mathbf{R} \cup \mathbf{Q}\ \forall \sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$

1) R ***safe by*** $\boldsymbol{\Sigma}$ ***after*** $\sigma \Leftrightarrow R$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** σ,

2) $\sigma\cdot\langle z\rangle \in \boldsymbol{\Sigma}\ \&\ \exists A \in \boldsymbol{but}(z)\ A$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** $\sigma \Rightarrow \exists B \in \boldsymbol{but}(z)\ B$ ***safe by*** $\boldsymbol{\Sigma}$ ***after*** σ,

3) P ***safe by*** $\boldsymbol{\Sigma}$ ***after*** $\sigma \Rightarrow P$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** $\sigma\ \&\ \exists v \in \boldsymbol{obs}(P)\ \sigma\cdot\langle v\rangle \in \boldsymbol{\Sigma}$.

Такое отношение ***safe by*** существует тогда и только тогда, когда все неразрушающие трассы $\boldsymbol{\Sigma}$ **R**-конвергентны. Действительно, если есть неразрушающая не-**R**-конвергентная трасса, то для некоторой **R**-кнопки P трасса не продолжается наблюдениями из $\boldsymbol{obs}(P)$, но P неразрушающая и, следовательно, безопасная по ***safe by*** после трассы. Тем самым, не может быть выполнено третье правило отношения ***safe by***. Обратно, если все трассы $\boldsymbol{\Sigma}$ **R**-конвергентны, то неразрушающая трасса продолжается каким-нибудь наблюдением из каждой **R**-кнопки. Поэтому достаточно объявить безопасной по ***safe by*** каждую неразрушающую **Q**-кнопку, разрешающую действие, продолжающее эту трассу. Однако в целом указанные требования неоднозначно определяют отношение ***safe by***, и при задании спецификации указывается конкретное отношение.

Для заданной **R**/**Q**-семантики в качестве модели спецификации будем рассматривать трассовую **R**-модель. Эта модель по определению непуста, префикс-замкнута и все ее трассы **R**-конвергентны, поэтому на ней может быть задано отношение ***safe by***. Кроме того, в силу **R**-конвергентности **R**-модели третье правило ***safe by*** достаточно сформулировать только для **Q**-кнопок: $\forall Q \in \mathbf{Q}\ \forall \sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$

4) Q ***safe by*** $\boldsymbol{\Sigma}$ ***after*** $\sigma \Rightarrow Q$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** $\sigma\ \&\ \exists v \in Q\ \sigma\cdot\langle v\rangle \in \boldsymbol{\Sigma}$.

Для заданной **R**/**Q**-семантики в алфавите $\mathbf{L}=\cup\mathbf{R}\cup\cup\mathbf{Q}$ спецификация полностью задается трассовой **R**-моделью и отношением безопасности кнопок ***safe by***.

3. Замечание о пустых кнопках

Пустая **Q**-кнопка опасна как после любой безопасной по отношению ***safe in*** трассы любой реализации, так и после любой безопасной по любому отношению ***safe by*** трассы любой спецификации. Такую кнопку никогда нельзя нажимать при безопасном тестировании: поскольку **Q**-отказы ненаблюдаемы, а пустая **Q**-кнопка не разрешает никаких внешних действий, нажатие **Q**-кнопки либо означает попытку выхода из дивергенции, либо приводит к возникновению ненаблюдаемого отказа. Поэтому в дальнейшем будем считать, что такой **Q**-кнопки в семантике нет: $\varnothing\notin\mathbf{Q}$.

В то же время пустая **R**-кнопка имеет смысл: она безопасна как по ***safe in***, так и по ***safe by***, после любой неразрушающей трассы, не продолжающейся дивергенцией, а наблюдение пустого **R**-отказа означает, что реализация оказалась в стабильном состоянии[4].

4. Гипотеза о безопасности и безопасная конформность

Требование безопасности тестирования выделяет класс *безопасно-тестируемых* реализаций, то есть таких, которые могут быть безопасно протестированы для проверки их конформности заданной спецификации. Этот класс определяется следующей *гипотезой о безопасности*: для спецификационной **R**-модели $\mathbf{\Sigma}$ с заданным отношением ***safe by*** реализационная $\mathbf{R}\cup\mathbf{Q}$-модель $\mathbf{I}$ *безопасно-тестируема*, если выполнены следующие условия:

1) если пустая трасса безопасна по ***safe by*** в спецификации, то она безопасна по ***safe in*** в реализации, что эквивалентно: в реализации нет разрушения с самого начала, если этого нет в спецификации;
2) после общей безопасной трассы реализации и спецификации любая кнопка, безопасная по ***safe by*** в спецификации, безопасна по ***safe in*** после этой трассы в реализации:

$$\mathbf{I}\ \textbf{\textit{safe for}}\ \mathbf{\Sigma} \triangleq (\langle\gamma\rangle\notin\mathbf{\Sigma} \Rightarrow \langle\gamma\rangle\notin\mathbf{I})\ \&\ \forall\sigma\in\textbf{\textit{SafeBy}}(\mathbf{\Sigma})\cap\mathbf{I}\ \forall P\in\mathbf{R}\cup\mathbf{Q}$$

$$(P\ \textbf{\textit{safe by}}\ \mathbf{\Sigma}\ \textbf{\textit{after}}\ \sigma \Rightarrow P\ \textbf{\textit{safe in}}\ \mathbf{I}\ \textbf{\textit{after}}\ \sigma).$$

Обозначим класс безопасно-тестируемых реализаций для данной спецификации $\mathbf{\Sigma}$ и данного отношения ***safe by***:

$$\textbf{\textit{SafeImp}}(\mathbf{\Sigma},\textbf{\textit{safe by}}) = \{\mathbf{I}|\mathbf{I}\ \textbf{\textit{safe for}}\ \mathbf{\Sigma}\}.$$

Теперь можно определить отношение (безопасной) *конформности*: для спецификации $\mathbf{\Sigma}$ с заданным отношением ***safe by*** реализация $\mathbf{I}$ *безопасно конформна* (или просто *конформна*), если она безопасно-тестируема и

[4] В [21] такое наблюдение обозначено символом S.

выполнено *тестируемое условие*: любое наблюдение, возможное в реализации в ответ на нажатие безопасной в спецификации по ***safe by*** кнопки, разрешается спецификацией:

$$\mathbf{I}\ \textit{saco}\ \boldsymbol{\Sigma} \triangleq \mathbf{I}\ \textit{safe for}\ \boldsymbol{\Sigma}$$
$$\& \quad \forall \sigma \in \textit{SafeBy}(\boldsymbol{\Sigma}) \cap \mathbf{I}\ \forall P\ \textit{safe by}\ \boldsymbol{\Sigma}\ \textit{after}\ \sigma$$
$$\textit{obs}(\sigma,P,\mathbf{I}) \subseteq \textit{obs}(\sigma,P,\boldsymbol{\Sigma}),$$

где $\textit{obs}(\sigma,P,\mathbf{M}) \triangleq \{u \mid \sigma\cdot\langle u\rangle \in \mathbf{M}\} \cap \textit{obs}(P)$ множество наблюдений, которые можно получить над множеством трасс $\mathbf{M}$ при нажатии кнопки P после трассы σ.

Обозначим класс конформных реализаций для данной спецификации $\boldsymbol{\Sigma}$ и данного отношения ***safe by***:

$\textit{ConfImp}(\boldsymbol{\Sigma},\textit{safe by}) = \{\mathbf{I} \mid \mathbf{I}\ \textit{saco}\ \boldsymbol{\Sigma}\}$.

Заметим, что, если $\mathbf{I}\ \textit{safe for}\ \boldsymbol{\Sigma}$, $\sigma \in \textit{SafeBy}(\boldsymbol{\Sigma}) \cap \mathbf{I}$, P $\textit{safe by}\ \boldsymbol{\Sigma}\ \textit{after}\ \sigma$, то $\textit{obs}(\sigma,P,\mathbf{I}) \cap \mathbf{Q} = \varnothing$. Поэтому, если $u \in \textit{obs}(\sigma,P,\mathbf{I})$ и $u=P$, то $P \in \mathbf{R}$.

Следует отметить, что гипотеза о безопасности не проверяема при тестировании и является его предусловием; тестирование проверяет тестируемое условие конформности.

Определенные выше гипотеза о безопасности и конформность называются трассовыми, поскольку они основаны только на трассах наблюдений.[5] Если модели реализации и спецификации заданы в виде LTS $\mathbf{I}$ и $\mathbf{S}$, то используются множества $\textit{tr}(\mathbf{I}_{\mathbf{R}\cup\mathbf{Q}})$ и $\textit{tr}(\mathbf{S}_{\mathbf{R}})$ трасс этих LTS.

6. Стандартная генерация тестов

Тестирование основано на, так называемой, *тестовой гипотезе* [18], согласно которой у реализации есть модель, адекватно описывающая поведение реализации в машине тестирования. При этом сама модель реализации может быть неизвестна, предполагается лишь ее существование. Для безопасного тестирования мы дополнительно предполагаем выполнение гипотезы о безопасности.

Для спецификации $\boldsymbol{\Sigma}$ с заданным отношением ***safe by*** *тестовой* трассой будем называть безопасную трассу, продолженную безопасным после нее наблюдением, то есть трассу $\sigma\cdot\langle u\rangle$, где $\sigma \in \textit{SafeBy}(\boldsymbol{\Sigma})$ & u $\textit{safe by}\ \boldsymbol{\Sigma}\ \textit{after}\ \sigma$. При безопасном тестировании проходятся только тестовые трассы.

Безопасные трассы тестовые, но могут быть и другие тестовые трассы, если $\sigma\cdot\langle u\rangle \notin \boldsymbol{\Sigma}$. Такие тестовые трассы, отсутствующие в спецификации, будем называть *ошибками* (*ошибочными трассами*). Будем говорить, что *в*

[5] Другим видом конформности является симуляция, основанная, кроме того, на соответствии состояний реализации и спецификации. Вопросы безопасной симуляции и ее тестирования рассмотрены в наших работах [10][11][12].

реализации есть ошибка $\sigma\cdot\langle u\rangle$, если спецификация определяет ошибку $\sigma\cdot\langle u\rangle$, а в реализации есть трасса $\sigma\cdot\langle u\rangle$. Очевидно, безопасно-тестируемая реализация конформна тогда и только тогда, когда в ней нет ошибок.

В терминах машины тестирования тест – это инструкция оператору машины. В каждом пункте инструкции указывается кнопка, которую оператор должен нажимать, и для каждого наблюдения – пункт инструкции, который должен выполняться следующим, или вердикт (***pass*** или ***fail***), если тестирование нужно закончить. Тест можно понимать как префикс-замкнутое множество конечных историй с назначенными вердиктами, в котором:

1) каждая максимальная история заканчивается наблюдением, и ей приписан вердикт, немаксимальным историям вердикты не приписаны;
2) каждая немаксимальная история, заканчивающаяся кнопкой, продолжается во множестве только теми наблюдениями, которые разрешаются этой кнопкой (это следует из определения истории в п.3);
3) каждая немаксимальная история, заканчивающаяся кнопкой, обязательно продолжается во множестве всеми наблюдениями, которые могут встречаться в безопасно-тестируемых реализациях после трассы этой истории.

Тест безопасен тогда и только тогда, когда трассы всех его историй являются тестовыми. Реализация *проходит* тест, если её тестирование с помощью этого теста всегда заканчивается с вердиктом ***pass***. Реализация проходит набор тестов, если она проходит каждый тест из набора. Набор тестов *значимый*, если каждая конформная реализация его проходит; *исчерпывающий*, если каждая неконформная реализация его не проходит; *полный*, если он значимый и исчерпывающий. Для определения конформности или неконформности любой безопасно-тестируемой реализации ставится задача генерации полного набора тестов по спецификации.

Строгим тестом будем называть такой тест, в котором вердикт ***pass*** назначается максимальной истории, если ее трасса не является ошибкой (имеется в спецификации), а вердикт ***fail*** – если ее трасса является ошибкой (отсутствует в спецификации). Такие тесты, во-первых, значимые (не фиксируют ложных ошибок) и, во-вторых, не пропускают обнаруженных ошибок. Мы будем рассматривать только строгие тесты.

Полный набор тестов всегда существует, в частности, им является набор всех *примитивных* тестов [5]. Примитивный тест строится по одной немаксимальной безопасной трассе спецификации $\sigma=\langle u_1,u_2,\ldots,u_n\rangle$. Для этого в трассу вставляются кнопки, которые оператор должен нажимать: перед каждым наблюдением u_i вставляется какая-нибудь безопасная после префикса $\langle u_1,u_2,\ldots,u_{i-1}\rangle$ кнопка $P_i\in$***but***(u_i): если $u_i\in\mathbf{R}$, то $P_i=u_i$, иначе $u_i\in P_i$. После всей трассы σ вставляется любая безопасная после нее кнопка P_{n+1}. В результате получается тестовая история $T=\langle P_1,u_1,P_2,u_2,\ldots,P_{n+1}\rangle$. Любая другая история T`

теста – это строгий префикс истории T или имеет вид $T`=\langle P_1,u_1,P_2,u_2,\ldots,P_i,u_i`\rangle$, где $u_i`$ наблюдение, продолжающее трассу σ при i=n+1, или ответвляющееся от трассы при $i\le n$ и $u_i`\neq u_i$. Безопасность трассы σ для каждого отказа $u_i\in\mathbf{R}$ гарантирует безопасность кнопки $P_i=u_i$, и для каждого действия $u_i\in\mathbf{L}$ гарантирует наличие разрешающей его безопасной кнопки $P_i\in\boldsymbol{but}(u_i)$, а немаксимальность безопасной трассы σ гарантирует наличие последней кнопки P_{n+1}. Выбор кнопок P_i для $u_i\in\mathbf{L}$ и кнопки P_{n+1} может быть неоднозначным: по одной безопасной трассе спецификации можно сгенерировать, вообще говоря, несколько разных примитивных тестов.

Любой строгий тест как множество историй равен объединению некоторого множества примитивных тестов (в объединении максимальные истории сохраняют те вердикты, которые были им приписаны в объединяемых примитивных тестах). Поэтому такой строгий тест обнаруживает те же самые ошибки, что и соответствующее множество примитивных тестов. Поэтому обычно рассматриваются только примитивные тесты.

7. Финальные модели спецификации

Из определения отношений ***safe by***, ***safe for*** и ***saco*** видно, что не все трассы спецификации нужны для определения безопасно-тестируемости и конформности. Кроме того, отношение ***safe by*** приходится задавать отдельно от спецификационной модели.

В этом разделе мы определим такое множество трасс для спецификации, которого достаточно для определения этих отношений и которое в некотором смысле «минимально». Это множество будем называть финальной трассовой моделью спецификации. Разным отношениям ***safe by*** для одной и той же **R**-модели спецификации будут соответствовать разные финальные модели спецификации, на которых эти отношения ***safe by*** совпадают с отношением ***safe in***. Иными словами, поскольку ***safe in*** определяется однозначно, финальная модель спецификации одновременно задает как нужное множество трасс спецификации, так и отношение ***safe by***.

Мы определим преобразования **R**-модели спецификации в финальную трассовую модель спецификации. Также определим характеристические свойства финальной модели, то есть набор свойств, которых необходимо и достаточно для того, чтобы множество трасс было результатом такого преобразования.

В этом подразделе будем считать, что заданы **R**/**Q**-семантика в алфавите **L** и **R**-модель спецификации **Σ**. Финальная модель будет множеством трасс $\boldsymbol{\Omega}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$. Будем обозначать подмножество его трасс без **Q**-отказов через $\boldsymbol{\Omega}_\mathbf{R}=\boldsymbol{\Omega}\cap(\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\})^*$.

Мы определим два вида трассовых моделей спецификации: предфинальную (трассовую) модель, которую будем называть p-моделью, и финальную (трассовую модель), которую будем называть f-моделью. В последнем разделе

рассмотрим представление этих моделей в виде детерминированных LTS в алфавите $\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta\}$.

8. Финальные трассовые модели

8.1. Финальные суффиксы и продолжения трасс

Пусть $\mathbf{\Psi}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$ непустое префикс-замкнутое множество трасс. *Финальным суффиксом* трассы $\mu\in\mathbf{\Psi}$ во множестве $\mathbf{\Psi}$ будем называть такую трассу λ, что $\mu\cdot\lambda\in\mathbf{\Psi}$ и 1) $\lambda=\epsilon$, 2) $\lambda=\langle\Delta\rangle$, 3) $\lambda=\langle z,\gamma\rangle$, 4) $\lambda=\langle z\rangle$ и $\mu\cdot\langle z,\gamma\rangle\in\mathbf{\Psi}$, или 5) $\lambda=\langle Q\rangle$, где $Q\in\mathbf{Q}$ & $Q\ \boldsymbol{safe}_{\gamma\Delta}\ \mathbf{\Psi}\ \boldsymbol{after}\ \mu$. В случае 5 будем называть λ **Q**-финальным суффиксом. Множество финальных суффиксов трассы μ во множестве $\mathbf{\Psi}$ обозначим $\boldsymbol{fs}(\mathbf{\Psi},\mu)$.

Финальным продолжением трассы $\mu\in\mathbf{\Psi}$ во множестве $\mathbf{\Psi}$ будем называть продолжение трассы каким-либо ее финальным суффиксом, то есть такую трассу $\mu\cdot\lambda$, что $\lambda\in\boldsymbol{fs}(\mathbf{\Psi},\mu)$. Продолжение трассы ее **Q**-финальным суффиксом будем называть **Q**-финальным продолжением или **Q**-финальной трассой. Множество финальных продолжений трассы μ обозначим $\boldsymbol{fx}(\mathbf{\Psi},\mu)$. Финальные суффиксы и продолжения трассы, очевидно, связаны следующим образом:

$\boldsymbol{fx}(\mathbf{\Psi},\mu)=\{\mu\cdot\lambda|\lambda\in\boldsymbol{fs}(\mathbf{\Psi},\mu)\}$ и $\boldsymbol{fs}(\mathbf{\Psi},\mu)=\{\lambda|\mu\cdot\lambda\in\boldsymbol{fx}(\mathbf{\Psi},\mu)\}$.

5. Предфинальные трассы и предфинальные множества

Трассу из непустого префикс-замкнутого множества $\mathbf{\Psi}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\})^*$ будем называть *предфинальной*, если это пустая трасса ϵ, трасса $\langle\gamma\rangle$ или финальное продолжение неразрушающей трассы. Заметим, что в таком множестве $\mathbf{\Psi}$ нет **Q**-отказов, поэтому у его трасс нет **Q**-финальных суффиксов. Формально множество предфинальных трасс определяется так:

$\boldsymbol{ptr}(\mathbf{\Psi})\triangleq\{\epsilon,\langle\gamma\rangle\}\cap\mathbf{\Psi}\cup\cup\{\boldsymbol{fx}(\mathbf{\Psi},\sigma)|\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})\}$.

Теорема 2: Пусть множество $\mathbf{\Psi}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\})^*$ непусто и префикс-замкнуто. Тогда:

$\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\mathbf{\Psi}))$,

$\forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})\ \forall P\in\mathbf{R}\cup\mathbf{Q}\ (P\ \boldsymbol{safe}_{\gamma\Delta}\ \mathbf{\Psi}\ \boldsymbol{after}\ \sigma\Leftrightarrow P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{ptr}(\mathbf{\Psi})\ \boldsymbol{after}\ \sigma)$.

□264

Очевидно, что $\boldsymbol{ptr}(\mathbf{\Psi})\subseteq\mathbf{\Psi}$. Будем говорить, что множество $\mathbf{\Psi}$ *предфинально*, если оно совпадает с подмножеством его предфинальных трасс: $\boldsymbol{ptr}(\mathbf{\Psi})=\mathbf{\Psi}$.

Теорема 3: Если множество $\mathbf{\Psi}$ *предфинально*, то подмножество его неразрушающих трасс совпадает с подмножеством его трасс, не содержащих дивергенции и разрушения и не продолжающихся разрушением:

$\boldsymbol{ptr}(\mathbf{\Psi})=\mathbf{\Psi}\Rightarrow\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=\{\sigma\in\mathbf{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\mathbf{\Psi}\}$.

Если трассы из $\boldsymbol{\Psi}$ допустимы и после отказа в них нет разрушения (часть условия согласованности), то:

$$\boldsymbol{ptr}(\boldsymbol{\Psi})=\boldsymbol{\Psi} \Leftarrow \langle\gamma\rangle\notin\boldsymbol{\Psi} \;\&\; \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})=\{\sigma\in\boldsymbol{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^{*}|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Psi}\} \vee \boldsymbol{\Psi}=\{\epsilon,\langle\gamma\rangle\}.$$

□265

Из этой теоремы следует, что, если трассы из $\boldsymbol{\Psi}$ допустимы и после отказа в них нет разрушения (часть условия согласованности), то:

$$\boldsymbol{ptr}(\boldsymbol{\Psi})=\boldsymbol{\Psi} \Leftrightarrow \langle\gamma\rangle\notin\boldsymbol{\Psi} \;\&\; \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})=\{\sigma\in\boldsymbol{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^{*}|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Psi}\} \vee \boldsymbol{\Psi}=\{\epsilon,\langle\gamma\rangle\}.$$

6. Незамкнутость и финально-замкнутость

Для заданных алфавита $\mathbf{L}$ и семейства $\mathbf{R}\subseteq P(\mathbf{L})$ множество, обладающее всеми свойствами $\mathbf{R}$-модели, кроме замкнутости, будем называть *незамкнутой (трассовой)* $\mathbf{R}$*-моделью*.

Теорема 4: Если $\boldsymbol{\Sigma}$ незамкнутая $\mathbf{R}$-модель, то множество $\boldsymbol{D}(\boldsymbol{\Sigma})$ является (замкнутой) $\mathbf{R}$-моделью.

□267

Множество $\boldsymbol{\Psi}$ будем называть *финально-замкнутым*, если после удаления отказов из неразрушающей трассы получается либо разрушающая трасса, либо неразрушающая трасса с сохранением финальных суффиксов:

$\forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}})\ \forall\mu\in\boldsymbol{d}(\sigma)$
$(\mu\in\boldsymbol{UnSafe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}}) \vee \mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}}) \;\&\; \boldsymbol{fs}(\boldsymbol{\Omega}_{\mathbf{R}},\sigma)\subseteq\boldsymbol{fs}(\boldsymbol{\Omega}_{\mathbf{R}},\mu))$.
Из финально-замкнутости следует, что, если $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}})$, $\mu\in\boldsymbol{d}(\sigma)$ и
$\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}})$, то для каждой кнопки P
P $\boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \sigma \Rightarrow$ P $\boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \mu$.

***P*-модель**

Незамкнутую $\mathbf{R}$-модель $\boldsymbol{\Psi}$ будем называть *p-моделью*, если она предфинальна и финально-замкнута. Иными словами, p-модель – это непустое, префикс-замкнутое множество $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс, обладающее свойствами допустимости, согласованности, $\mathbf{R}$-конвергентности, финально-замкнутости, $\mathbf{R}$-полноты и предфинальности.

Теорема 5: Множество $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ предфинальных трасс $\mathbf{R}$-модели $\boldsymbol{\Sigma}$ является p-моделью.

□268

Теорема 6: $\boldsymbol{d}$-замыкание p-модели $\boldsymbol{\Psi}$ является $\mathbf{R}$-моделью, множество предфинальных трасс которой совпадает с исходной p-моделью: $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))=\boldsymbol{\Psi}$.

□271

7. F-модель

Из теоремы 2 следует, что множество $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ предфинальных трасс $\mathbf{R}$-модели спецификации $\boldsymbol{\Sigma}$ однозначно определяет отношение $\boldsymbol{safe}_{\gamma\Delta}$ и множество неразрушающих трасс $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$. Множества $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ также

достаточно для определения отношения ***safe by***. По теореме 5 множество ***ptr***(**Σ**) является р-моделью.

Мы будем рассматривать отношение ***safe by*** на произвольной р-модели **Ψ**. Оно должно удовлетворять правилам отношения ***safe by***:

$\forall R\in\mathbf{R}\ \forall z\in\mathbf{L}\ \forall P\in\mathbf{R}\cup\mathbf{Q}\ \forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$

1) R ***safe by*** $\boldsymbol{\Sigma}$ ***after*** $\sigma \Leftrightarrow R$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** σ,

2) $\sigma\cdot\langle z\rangle\in\boldsymbol{\Psi}\ \&\ \exists A\in\boldsymbol{but}(z)\ A$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Psi}$ ***after*** $\sigma \Rightarrow \exists B\in\boldsymbol{but}(z)\ B$ ***safe by*** $\boldsymbol{\Psi}$ ***after*** σ,

3) P ***safe by*** $\boldsymbol{\Psi}$ ***after*** $\sigma \Rightarrow P$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Psi}$ ***after*** $\sigma\ \&\ \exists v\in\boldsymbol{obs}(P)\ \sigma\cdot\langle v\rangle\in\boldsymbol{\Psi}$.

Поскольку это отношение однозначно определяется для **R**-кнопок и разрушающих **Q**-кнопок, достаточно доопределить это отношение только для неразрушающих **Q**-кнопок после неразрушающих трасс. Оно должно удовлетворять второму и третьему правилам отношения ***safe by***, которые можно записать в следующем виде: $\forall z\in\mathbf{L}\ \forall Q\in\mathbf{Q}\ \forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$

2a) $\sigma\cdot\langle z\rangle\in\boldsymbol{\Psi}\ \&\ \exists A\in\boldsymbol{but}(z)\ A$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Psi}$ ***after*** $\sigma \Rightarrow \exists B\in\boldsymbol{but}(z)\ B$ ***safe by*** $\boldsymbol{\Psi}$ ***after*** σ,

3a) Q $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Psi}$ ***after*** $\sigma\ \&\ Q$ ***safe by*** $\boldsymbol{\Psi}$ ***after*** $\sigma \Rightarrow \exists v\in Q\ \sigma\cdot\langle v\rangle\in\boldsymbol{\Psi}$.

Для того чтобы изобразить теперь выбранное отношение ***safe by*** в модели добавим в р-модель продолжение каждой неразрушающей трассы σ каждым неразрушающим, но опасным после нее по ***safe by*** **Q**-отказом Q, то есть добавим трассу $\sigma\cdot\langle Q\rangle$. Добавленная трасса $\sigma\cdot\langle Q\rangle$ – это **Q**-финальная трасса, то есть неразрушающая трасса σ, продолженная **Q**-финальным суффиксом $\langle Q\rangle$.

Далее *финальными трассами* будем называть предфинальные и **Q**-финальные трассы. Для спецификации, заданной в виде **R**-модели **Σ** или р-модели **Ψ**, и отношения ***safe by***, множество финальных трасс обозначим ***ftr***(**Σ**,***safe by***) и ***ftr***(**Ψ**,***safe by***), соответственно.

Такое множество финальных трасс определяется так:

$\boldsymbol{ftr}(\boldsymbol{\Psi},\boldsymbol{safe\ by}) \triangleq \boldsymbol{\Psi}$

$\cup\ \{\sigma\cdot\langle Q\rangle | \sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})\ \&\ Q\in\mathbf{Q}\ \&\ Q$ ***safe by*** $\boldsymbol{\Psi}$ ***after*** $\sigma\ \&\ Q$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Psi}$ ***after*** $\sigma\}$,

$\boldsymbol{ftr}(\boldsymbol{\Sigma},\boldsymbol{safe\ by}) \triangleq \boldsymbol{ftr}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\boldsymbol{safe\ by})$.

Для р-модели **Ψ** обозначим $\boldsymbol{\Omega}=\boldsymbol{ftr}(\boldsymbol{\Psi},\boldsymbol{safe\ by})$. Очевидно, $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Второе и третье правила отношения ***safe by*** теперь можно понимать как требования к добавлению **Q**-отказов после неразрушающих трасс: $\forall z\in\mathbf{L}\ \forall Q\in\mathbf{Q}\ \forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})$

2b) $\sigma\cdot\langle z\rangle\in\boldsymbol{\Omega}\ \&\ \sigma\cdot\langle z,\gamma\rangle\notin\boldsymbol{\Omega}$

$\Rightarrow \exists P\in\boldsymbol{but}(z)\ P$ $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Omega}$ ***after*** $\sigma\ \&\ (P\in\mathbf{Q} \Rightarrow \sigma\cdot\langle P\rangle\notin\boldsymbol{\Omega})$,

3b) Q $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Omega}$ ***after*** $\sigma\ \&\ \sigma\cdot\langle Q\rangle\notin\boldsymbol{\Omega} \Rightarrow \exists v\in Q\ \sigma\cdot\langle v\rangle\in\boldsymbol{\Omega}$.

Далее запишем формально правило, согласно которому мы добавляем только неразрушающие **Q**-отказы только после неразрушающих трасс и после **Q**-отказов ничего не добавляем:

4b) $\forall P\in\mathbf{Q}\ \forall\mu\cdot\langle P\rangle\cdot\lambda\in\mathbf{\Omega}\ \mu\in\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})$ & P ***safe***$_{\gamma\Delta}$ **Ω *after*** μ & $\lambda=\epsilon$.

Наконец, запишем условие того, что все добавляемые трассы содержат **Q**-отказы: $\mathbf{\Psi}=\mathbf{\Omega}_\mathbf{R}$, где $\mathbf{\Omega}_\mathbf{R}=\mathbf{\Omega}\cap(\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\})^*$.

Итак, если $\mathbf{\Omega}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$ и $\mathbf{\Psi}=\mathbf{\Omega}_\mathbf{R}$ р-модель, то условия 2b), 3b), 4b) эквивалентны $\mathbf{\Omega}=\boldsymbol{ftr}(\mathbf{\Psi},\boldsymbol{safe\ by})$.

Пусть множество трасс $\mathbf{\Omega}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$ непусто и префикс-замкнуто. Будем говорить, что **Ω** *финально*, если

$\langle\gamma\rangle\notin\mathbf{\Omega}$ & $\boldsymbol{SafeIn}(\mathbf{\Omega})=\{\sigma\in\mathbf{\Omega}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\mathbf{\Omega}\}\ \vee\ \mathbf{\Omega}=\{\epsilon,\langle\gamma\rangle\}$.

Будем говорить, что **Ω** **Q**-*допустимо*, если

$\forall P\in\mathbf{Q}\ \forall\mu\cdot\langle P\rangle\cdot\lambda\in\mathbf{\Omega}\ \mu\in\boldsymbol{SafeIn}(\mathbf{\Omega})$ & P ***safe***$_{\gamma\Delta}$ **Ω *after*** μ & $\lambda=\epsilon$.

Теорема 7: Пусть $\mathbf{\Omega}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$ и $\mathbf{\Psi}=\mathbf{\Omega}_\mathbf{R}$ р-модель. Тогда условия 2b), 3b), 4b) эквивалентны следующим трем условиям:

2c) **Ω** финально,
3c) трассы из $\mathbf{\Omega}_\mathbf{R}$ **Q**-конвергентны в **Ω**,
4c) **Ω** **Q**-допустимо.

□274

Очевидно, $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega}_\mathbf{R})$. Очевидно также, что, если **Ψ** р-модель и $\mathbf{\Omega}=\boldsymbol{ftr}(\mathbf{\Psi},\boldsymbol{safe\ by})$, то $\mathbf{\Psi}=\mathbf{\Omega}_\mathbf{R}$.

Множество трасс $\mathbf{\Omega}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$ будем называть *f-моделью*, если это непустое, префикс-замкнутое множество трасс, обладающее свойствами допустимости, **Q**-допустимости, согласованности, **R**∪**Q**-конвергентности трасс из $\mathbf{\Omega}_\mathbf{R}$ в **Ω**, финально-замкнутости $\mathbf{\Omega}_\mathbf{R}$, **R**-полноты $\mathbf{\Omega}_\mathbf{R}$ и финальности.

Теорема 8: Пусть на р-модели **Ψ** задано отношение ***safe by***. Тогда множество $\mathbf{\Omega}=\boldsymbol{ftr}(\mathbf{\Psi},\boldsymbol{safe\ by})$ является f-моделью. При этом

$\boldsymbol{SafeIn}(\mathbf{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\boldsymbol{SafeBy}(\mathbf{\Omega})=\boldsymbol{SafeBy}(\mathbf{\Psi})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=\boldsymbol{SafeIn}(\mathbf{\Psi})$.

□275

Теорема 9: Пусть **Ω** f-модель. Тогда $\mathbf{\Omega}_\mathbf{R}$ р-модель, отношение ***safe in*** на **Ω**, взятое для трасс из $\mathbf{\Omega}_\mathbf{R}$, удовлетворяет правилам отношения ***safe by***, и $\mathbf{\Omega}=\boldsymbol{ftr}(\mathbf{\Omega}_\mathbf{R},\boldsymbol{safe\ in})$.

□276

Обозначим класс безопасно-тестируемых реализаций для f-модели **Ω**:

$\boldsymbol{SafeImp}(\mathbf{\Omega})=\boldsymbol{SafeImp}(\mathbf{\Sigma},\boldsymbol{safe\ by})$, где $\mathbf{\Omega}=\boldsymbol{ftr}(\mathbf{\Sigma},\boldsymbol{safe\ by})$.

Обозначим класс конформных реализаций для f-модели **Ω**:

$\boldsymbol{ConfImp}(\mathbf{\Omega})=\boldsymbol{ConfImp}(\mathbf{\Sigma},\boldsymbol{safe\ by})$, где $\mathbf{\Omega}=\boldsymbol{ftr}(\mathbf{\Sigma},\boldsymbol{safe\ by})$.

f-модель и ∇-пополнение

f-модель в общем случае может содержать безопасные трассы, которые, тем не менее, не нужны для генерации тестов, поскольку они неконформны, то есть не встречаются в конформных реализациях. В [17] определяется преобразование трассовой спецификации, целью которого является удаление из спецификации неконформных трасс. Результат такого преобразования называется ∇-пополнением (то, что мы здесь называем ∇-пополнением исходной **R**-модели **Σ** в [17] обозначается как $\mathbf{\Sigma}^{\mathbf{01}\nabla}$). В общем случае ∇-пополнение в исходной **R**/**Q**-семантике может не существовать. Поэтому ∇-пополнение строится не в исходной **R**/**Q**-семантике, а в расширенной $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-семантике, которая получается добавлением в каждую кнопку P фиктивного действия, которое называется не-отказом и обозначается ⏥ . Тестирование в такой семантике эквивалентно тестированию в исходной семантике для реализаций в исходном алфавите **L**, то есть реализаций, в которых не-отказы не встречаются. В трассах ∇-пополнения не-отказы (и только они) непосредственно предшествуют дивергенции или разрушению, то есть не-отказы предназначены только для индикации безопасности кнопок после трасс. Кнопка P безопасна после трассы, если трасса не продолжается не-отказом ⏥ , за которым следует разрушение (либо не продолжается ⏥, либо после ⏥ дивергенция).

Заметим, что ∇-пополнение может быть пустым, что означает отсутствие конформных реализаций. Далее будем считать, что такие реализации есть и ∇-пополнение не пусто.

∇-пополнение не является $\mathbf{R}^{\#}$-моделью, но его ***d***-замыкание является $\mathbf{R}^{\#}$-моделью, причем множество безопасных трасс этой $\mathbf{R}^{\#}$-модели совпадает с множеством всех трасс без не-отказов ∇-пополнения.

∇-пополнение строится в два этапа. На первом этапе в спецификацию добавляются все трассы, по которым можно вести безопасное тестирование всех реализаций из класса безопасно-тестируемых реализаций исходной спецификации ***SafeImp*(Σ,*safe by***). Это значит, что примитивный тест, сгенерированный по такой трассе, будет безопасным для любой реализации из этого класса. Заметим, что такие трассы могут и не принадлежать исходной спецификации. На втором этапе из спецификации удаляются неконформные трассы. Поэтому, хотя все безопасные трассы без не-отказов ∇-пополнения конформны, однако некоторые из них отсутствуют в исходной спецификации, поскольку были добавлены на первом этапе пополнения и оказались конформными.

Такие добавленные трассы можно было бы удалить из ∇-пополнения, оставив только те безопасные трассы без не-отказов, которые есть в f-модели исходной спецификации. Это первая задача. Можно поставить и вторую задачу: удалить из f-модели неконформные трассы, используя ∇-пополнение, то есть удалить те безопасные трассы, которых нет в ∇-пополнении.

При решении обеих этих задач тесты для реализации в алфавите **L** генерируются по одним и тем же трассам: безопасным конформным трассам исходной спецификации (или, что то же самое, ее f-модели). Отличие только в том, как определяется безопасность кнопок после этих трасс: по ∇-пополнению или по f-модели.

Опишем решения этих задач формально.

Пусть **Ω** f-модель, а **Ξ** ∇-пополнение.

Поскольку **Ω** и **Ξ** заданы в разных семантиках, сначала определим перевод трасс из одной семантики в другую.

Преобразование из **R**/**Q**-семантики в $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-семантику. Если мы берем трассу без не-отказов $\sigma\in\mathbf{\Omega}$, то ей соответствует трасса в $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-семантике, которая строится следующим образом: в каждый **R**-отказ в трассе σ добавляется соответствующий ему не-отказ. В результате будет получена трасса, которую будем обозначать как $\sigma^{\#}$. Формально: $\forall P\in\mathbf{R}\cup\mathbf{Q}\ \forall z\in\mathbf{L}\ \forall\sigma\in(\mathbf{L}\cup\mathbf{R})^{*}$

$P^{\#} \triangleq P\cup\{\pitchfork\}$, $z^{\#} \triangleq z$, $\sigma^{\#} \triangleq \langle\sigma(i)^{\#}|i=1..|\sigma|\rangle$.

Преобразование из $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-семантики в **R**/**Q**-семантику. Если мы берем трассу без не-отказов $\sigma\in\mathbf{\Xi}$, то ей соответствует трасса в **R**/**Q**-семантике, которая строится следующим образом: из каждого $\mathbf{R}^{\#}$-отказа в трассе удаляется не-отказ, то есть оставляются только действия из **L**. В результате будет получена трасса, которую будем обозначать как $\sigma_{\mathbf{L}}$. Формально: $\forall P\in\mathbf{R}^{\#}\ \forall z\in\mathbf{L}\ \forall\sigma\in(\mathbf{L}\cup\mathbf{R}^{\#})^{*}$

$P_{\mathbf{L}} \triangleq P\cap\mathbf{L}$, $z_{\mathbf{L}} \triangleq z$, $\sigma_{\mathbf{L}} \triangleq \langle\sigma(i)_{\mathbf{L}}|i=1..|\sigma|\rangle$.

Первая задача. Для решения первой задачи мы должны, во-первых, взять все трассы из **Ξ**, которые встречаются в $\mathbf{\Omega}^{\#}$. Это будут безопасные и конформные трассы, которые есть и в **Ξ**, и в $\mathbf{\Omega}^{\#}$. Во-вторых, мы должны взять все их продолжения не-отказами и далее разрушением или дивергенцией, которые есть в **Ξ**. В результате получится следующее множество:

$$\mathbf{\Xi}①\mathbf{\Omega} \triangleq \mathbf{\Xi}\cap\mathbf{\Omega}^{\#} \cup \{\mu\cdot\langle P\rangle\cdot\lambda|\mu\in\mathbf{\Xi}\cap\mathbf{\Omega}^{\#}\ \&\ P\in\mathbf{R}\cup\mathbf{Q}\ \&\ \mu\cdot\langle\pitchfork\rangle\cdot\lambda\in\mathbf{\Xi}\}.$$

Проблема здесь в том, что тесты, сгенерированные по $\mathbf{\Xi}①\mathbf{\Omega}$ могут ловить «ложные» ошибки. Это объясняется тем, что оставшаяся в $\mathbf{\Xi}①\mathbf{\Omega}$ трасса может не продолжаться наблюдениями, которые были в **Ξ**, но удалены из $\mathbf{\Xi}①\mathbf{\Omega}$. Если удалены все такие наблюдения из некоторой **Q**-кнопки, которая была опасной в f-модели, но безопасна в ∇-пополнении, то после нажатия такой кнопки будет ловиться «ложная» ошибка.

Пример приведен на рис. 3. Здесь все трассы LTS-спецификации **S** безопасны, поэтому все они будут в ее f-модели **Ω**. Также все эти трассы конформны, поэтому все они будут в ∇-пополнении **Ξ**. Но в **Ξ** добавляется переход, изображенный пунктиром. В этом примере имеем $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})\subset(\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Xi}))_{\mathbf{L}}$. Кроме того, добавляются не изображенные переходы по не-отказам, ведущие в состояние, где есть дивергенция (и нет разрушения). В состояниях, после

трасс, заканчивающихся отказом δ (эти состояния помечены символом δ), проводятся переходы только по не-отказу ~~{x}~~, а в остальных состояниях – по обоим не-отказам ~~{x}~~ и ~~δ~~. Все добавленные трассы $\delta\delta^*x\delta\delta^*x(xa)^*$ конформны. После удаления этих трасс в $\Xi①\boldsymbol{\Omega}$ кнопка {x} остаётся безопасной после трассы δxδ, поскольку по-прежнему нет трассы δxδ~~{x}~~γ. В любой безопасно-тестируемой реализации после трассы δxδ должно быть действие x, однако теперь оно отсутствует в спецификации $\Xi①\boldsymbol{\Omega}$ и, следовательно, считается ошибкой. В исходной спецификации **S** (и в ее f-модели **Ω**) кнопка {x} опасна после трассы δxδ, поэтому действие x после этой трассы не считается ошибкой. Из-за этого реализация может быть конформна исходной спецификации **S** (и ее f-модели **Ω**) и неконформна $\Xi①\boldsymbol{\Omega}$.

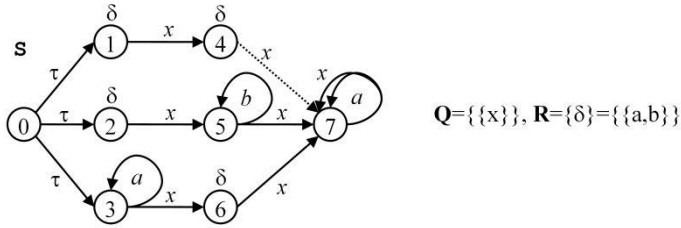


Рис. 3. $\Xi①\boldsymbol{\Omega}$

Вторая задача. Для решения второй задачи мы должны, во-первых, взять все трассы без не-отказов из $\boldsymbol{\Xi}_{\mathbf{L}}$, которые встречаются в **Ω**. Это будут безопасные конформные трассы, которые есть и в $\boldsymbol{\Xi}_{\mathbf{L}}$, и в **Ω**. Во-вторых, мы должны взять все их продолжения финальными суффиксами, которые есть в **Ω**. Поскольку пустая трасса тоже является финальным суффиксом, достаточно только «во-вторых». В результате получится множество:

$\boldsymbol{\Omega}②\boldsymbol{\Xi} \triangleq \{\mu{\cdot}\lambda | \mu \in \boldsymbol{\Omega} \cap \boldsymbol{\Xi}_{\mathbf{L}} \;\&\; \lambda \in \boldsymbol{fs}(\boldsymbol{\Omega},\mu)\}$.

Множество безопасных трасс множества $\boldsymbol{\Omega}②\boldsymbol{\Xi}$ совпадает с множеством безопасных трасс без не-отказов множества $\Xi①\boldsymbol{\Omega}$. В обоих случаях это безопасные конформные трассы исходной спецификации. Однако теперь безопасность кнопок после этих трасс определяется по множеству $\boldsymbol{\Omega}②\boldsymbol{\Xi}$, то есть по их финальным суффиксам в $\boldsymbol{\Omega}②\boldsymbol{\Xi}$, которые такие же, как в f-модели **Ω**. Поэтому набор тестов, сгенерированный по всем безопасным трассам $\boldsymbol{\Omega}②\boldsymbol{\Xi}$ является полным для исходной спецификации (и ее f-модели **Ω**).

В качестве примера можно рассмотреть спецификацию на рис. 4. Здесь пунктиром отмечены переходы и состояния, удаляемые в ∇-пополнении **Ξ**: трасса δxδ неконформна. Неконформные трассы удаляются и из f-модели **Ω** при построении $\boldsymbol{\Omega}②\boldsymbol{\Xi}$. В этом примере имеем $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}) \supset (\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Xi}))_{\mathbf{L}}$.

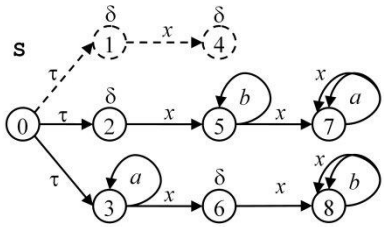


Рис. 4. **Ω②Ξ**

Проблема лишь в том, какие классы безопасно-тестируемых ***SafeImp*(Ω②Ξ)** и конформных ***ConfImp*(Ω②Ξ)** реализаций определяет само множество **Ω②Ξ**. Класс безопасно-тестируемых реализаций не сужается: ***SafeImp*(Ω)=*SafeImp*(Σ,*safe by*)⊆*SafeImp*(Ω②Ξ)**. Это объясняется тем, что, удаляя трассы из **Ω**, мы оставляем все финальные суффиксы остающихся трасс. Тем самым, мы, быть может, только ослабляем требования к реализации по безопасности. Также очевидно, что не меняется подмножество конформных реализаций множества безопасно-тестируемых реализация для исходной спецификации: ***ConfImp*(Ω②Ξ)∩*SafeImp*(Ω)=*ConfImp*(Ω)**. Остается открытым вопрос о том, не появляются в расширенном классе безопасно-тестируемых реализаций новые конформные реализации:

(*SafeImp*(Ω②Ξ)*SafeImp*(Ω))∩*ConfImp*(Ω②Ξ
= *ConfImp*(Ω②Ξ)*ConfImp*(Ω) = ∅?

Здесь следует отметить, что множество **Ω②Ξ** в общем случае не является f-моделью. Это следствие того, что ∇-пополнение в исходной **R**/**Q**-семантике может не существовать. В частности, может быть нарушена **R**-полнота множества $(\Omega②\Xi)_{\mathbf{R}}$ в **Ω②Ξ**: могут не сохраняться финальные суффиксы при вставке **R**-отказа $i_{\mathbf{R}}$-операцией (пример в [17] на рис.15). Это является еще одной проблемой.

9. RTS-модели спецификации

Как было отмечено в п.4.1, LTS-модель является наиболее «наглядной» моделью. Кроме того, LTS-модель является способом конечного представления регулярных трассовых моделей. Этот способ, однако, обладает одним существенным недостатком: LTS-модель, вообще говоря, недетерминирована: трасса может заканчиваться не в одном, а в нескольких состояниях. Работать с такими трассами на LTS неудобно. В то же время этот недетерминизм вовсе не является неизбежным следствием недетерминизма моделируемой системы. Причина недетерминизма LTS-модели в том, что наблюдения делятся на два вида: действия и отказы, которые существенно различным образом отображаются в LTS-модели. Если трасса продолжается как отказом R, так и действием z∈R, то эти два продолжения не могут быть определены в одном и том же состоянии LTS-модели.

9.1. Порождающий граф

С другой стороны, для любого множества последовательностей (в том числе для трассовой модели) всегда существует порождающий его граф, в котором выделены множества начальных и конечных вершин. Такой граф всегда можно построить с одной начальной вершиной[6]. Поэтому его можно рассматривать как LTS, если вершины графа назвать состояниями LTS, дуги – переходами, а переходы, соответствующие непомеченным дугам, пометить символом τ. Будем считать, что порождающий граф задан парой: LTS и множество ее конечных состояний.

Множество последовательностей префикс-замкнуто тогда и только тогда, когда существует порождающий ее граф, в котором все вершины конечные. Такой граф будем задавать только LTS, порождаемое множество – это множество простых трасс этой LTS.

Любой порождающий граф можно детерминизировать, то есть преобразовать в детерминированную LTS с выделенным множеством ее конечных состояний. При этом сохраняется множество порождающих трасс. Процедуру детерминизации мы опишем ниже.

Поскольку каждая рассматриваемая нами выше трассовая модель префикс-замкнута, порождающий ее граф можно задать одной LTS, но не в исходном алфавите **L**, а (для **R**/**Q**-семантики) в алфавите $\mathbf{L}\cup\mathbf{R}\cup\{\Delta\}$.

Детерминированную LTS в алфавите $\mathbf{L}\cup\mathbf{R}\cup\{\Delta\}$ будем называть RTS (Refusal Transition System).

Нужно отметить, что за детерминизм модели приходится чем-то «жертвовать»: не любая LTS в алфавите $\mathbf{L}\cup\mathbf{R}\cup\{\Delta\}$ является графом, порождающим трассовую модель в **R**/**Q**-семантике. Характеристические свойства RTS зависят от типа трассовой модели, множество трасс которой она представляет.

Для спецификации мы рассматривали три вида таких моделей: **R**-модель, p-модель и f-модель, причем для первых двух моделей отдельно задается отношение ***safe by***.

8. Отношение safe by

Отношение ***safe by*** однозначно определяется для **R**-кнопок и разрушающих **Q**-кнопок, то есть вычисляется по модели (**R**- или p-модели). Поэтому достаточно определить только те **Q**-кнопки, которые неразрушающие, но опасные по ***safe by*** после каждой неразрушающей трассы σ. Это задается множеством трасс вида $\sigma\cdot\langle Q\rangle$, где σ неразрушающая трасса спецификации, а $Q\in\mathbf{Q}$ кнопка, неразрушающая, но опасная по ***safe by*** после σ. Для такого

[6] Достаточно добавить новую начальную вершину и провести из нее непомеченные дуги во все старые начальные вершины.

множества тоже существует порождающий граф, но только в нем могут быть уже не конечные вершины. Поскольку неразрушающие трассы не содержат **Q**-отказов, все конечные вершины этого графа терминальные.

Будем говорить, что отношение ***safe by*** регулярно, если это множество трасс регулярно, то есть существует конечный порождающий граф. В [17] было введено понятие ограниченного отношения ***safe by***, когда безопасность кнопок одинакова после трасс, заканчивающихся в одном множестве состояний LTS-спецификации (эти трассы заканчиваются в одном состоянии RTS, которую мы ниже построим по LTS-спецификации). Для конечной LTS спецификации ограниченный ***safe by*** регулярный, хотя обратное, вообще говоря, не верно.

9. Детерминизация порождающего графа

Сначала определим формально процедуру детерминизации порождающего графа, заданного LTS $\mathbf{T}=LTS(V_\mathbf{T},\mathbf{L}_\mathbf{T},E_\mathbf{T},t_0)$ и множеством конечных состояний $F\subseteq V_\mathbf{T}$. Эта процедура ***determ***(**T**,F) строит детерминированную LTS с тем же множеством простых трасс: $LTS(P(V_\mathbf{T}),\mathbf{L}_\mathbf{T},E,t_0$ ***after*** $\epsilon)$, где множество переходов E определяется как наименьшее множество, порождаемое следующим правилом вывода: $\forall A,B\in P(V_\mathbf{T})\ \forall u\in\mathbf{L}_\mathbf{T}$

$B=\cup(A$ ***after*** $\langle u\rangle)\neq\varnothing \quad \vdash \quad A\text{—}u\rightarrow B.$

Состояние A этой LTS объявляется конечным, если $A\cap F\neq\varnothing$. Если $F=V_\mathbf{T}$, то на этом процедура детерминизации заканчивается. Иначе, нам нужно удалить из построенной LTS все состояния, из которых недостижимо хотя бы одно конечное состояние; естественно, вместе с удаляемым состоянием удаляются и все входящие в него и выходящие из него переходы.

Мы построим детерминированный порождающий граф для **R**-модели, p-модели и f-модели. Поскольку все эти модели префикс-замкнуты, будет строиться только RTS, все ее состояния считаются конечными.

10. RTS для R-модели

Построим RTS для **R**-модели **Σ**, заданной как множество $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс LTS **S** в алфавите **L**. Эту RTS обозначим ***rts***(**S**). Напомним, что для определения $\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\}$-трасс LTS **S** строится LTS $\mathbf{S_R}=LTS(V_{\mathbf{SR}},\mathbf{L}\cup\mathbf{R}\cup\{\Delta\},E_{\mathbf{SR}},s_0)$. Для этого добавляются в каждом стабильном состоянии LTS **S** петли s—R→s, помеченные **R**-отказами, порождаемыми в этом состоянии, добавляется новое терминальное состояние ω, перенаправляются в это состояние все γ-переходы, а также проводятся в него Δ-переходы из дивергентных состояний. По определению $\mathbf{\Sigma}=\mathbf{tr}(\mathbf{S_R})$. Для построения ***rts***(**S**) выполняется преобразование детерминизации LTS $\mathbf{S_R}$: ***rts***(**S**)= ***determ***($\mathbf{S_R}$).

RTS ***rts***(**S**) будет конечной тогда и только тогда, когда конечна LTS $\mathbf{S_R}$. А для этого достаточно конечности исходной LTS **S** и семейства **R** (поскольку в $\mathbf{S_R}$ добавляются петли по **R**-отказам).

11. RTS для p-модели

Поскольку любая RTS **U** – это детерминированная LTS, каждая ее трасса σ заканчивается только в одном состоянии s, и для всех трасс, заканчивающихся в s, безопасность кнопок после них по отношению $\boldsymbol{safe}_{\gamma\Delta}$ зависит только от этого состояния. Определим: $\forall s\in V_{\mathbf{U}}\ \forall P\in\mathbf{R}\cup\mathbf{Q}\ \forall u\in\mathbf{L}\cup\mathbf{R}$

$P\ \boldsymbol{safe}_{\gamma\Delta}\ s \triangleq s\text{—}\Delta\not\rightarrow\ \&\ \forall z\in P\ s=\langle z,\gamma\rangle\not\Rightarrow,$

$u\ \boldsymbol{safe}_{\gamma\Delta}\ s \triangleq \exists P\in\boldsymbol{but}(u)\ P\ \boldsymbol{safe}_{\gamma\Delta}\ s.$

Очевидно, если **U** ***after*** σ = {s}, то:

$P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{tr}(\mathbf{U})\ \boldsymbol{after}\ \sigma \Leftrightarrow P\ \boldsymbol{safe}_{\gamma\Delta}\ s,\ u\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{tr}(\mathbf{U})\ \boldsymbol{after}\ \sigma \Leftrightarrow u\ \boldsymbol{safe}_{\gamma\Delta}\ s.$

Определим преобразование любой RTS, то есть детерминированной LTS $\mathbf{T}=LTS(V_{\mathbf{T}},\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\},E_{\mathbf{T}},t_0)$, которое оставляет только переходы по разрушению, по дивергенции при отсутствии разрушения, по действию с последующим разрушением, а также по неразрушающим наблюдениям: $\boldsymbol{delete}(\mathbf{T})=LTS(V_{\mathbf{T}},\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\},E,t_0)$, где множество переходов E определяется как наименьшее множество, порождаемое следующими правилами вывода: $\forall a,b\in V_{\mathbf{T}}\ \forall u\in\mathbf{L}\cup\mathbf{R}$

$a\text{—}\gamma\rightarrow b$	$\vdash$	$a\text{—}\gamma\rightarrow b,$
$a\text{—}\gamma\not\rightarrow\ \&\ a\text{—}\Delta\rightarrow b$	$\vdash$	$a\text{—}\Delta\rightarrow b,$
$a\text{—}\gamma\not\rightarrow\ \&\ a\text{—}\Delta\not\rightarrow\ \&\ a\text{—}u\rightarrow b\ \&\ b\text{—}\gamma\rightarrow$	$\vdash$	$a\text{—}u\rightarrow b,$
$a\text{—}\gamma\not\rightarrow\ \&\ a\text{—}\Delta\not\rightarrow\ \&\ a\text{—}u\rightarrow b\ \&\ u\ \boldsymbol{safe}_{\gamma\Delta}\ a$	$\vdash$	$a\text{—}u\rightarrow b.$

С помощью этого преобразования RTS для p-модели $\boldsymbol{ptr}(\boldsymbol{\Sigma})$, где $\boldsymbol{\Sigma}=\boldsymbol{tr}(\mathbf{S_R})$, строится так: $\boldsymbol{prts}(\mathbf{S})=\boldsymbol{delete}(\boldsymbol{rts}(\mathbf{S}))=\boldsymbol{delete}(\boldsymbol{determ}(\mathbf{S_R}))$.

Если $\boldsymbol{rts}(\mathbf{S})$ конечна, то $\boldsymbol{prts}(\mathbf{S})$ конечная. Обратное, вообще говоря, не верно, поскольку в $\boldsymbol{rts}(\mathbf{S})$ может быть бесконечное число переходов, но только конечное число их лежит на маршрутах с предфинальными трассами.

12. RTS для f-модели

Теперь построим RTS для f-модели $\boldsymbol{ftr}(\boldsymbol{\Sigma},\boldsymbol{safe\ by})$, где $\boldsymbol{\Sigma}=\boldsymbol{tr}(\mathbf{S_R})$. Эту RTS обозначим $\boldsymbol{frts}(\mathbf{S},\boldsymbol{safe\ by})$ и будем называть финальной RTS. По определению $\boldsymbol{ftr}(\boldsymbol{\Sigma},\boldsymbol{safe\ by}) = \boldsymbol{ftr}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\boldsymbol{safe\ by})$. Будем считать, что у нас уже построена RTS $\mathbf{P}=\boldsymbol{prts}(\mathbf{S})$ для p-модели $\boldsymbol{ptr}(\boldsymbol{\Sigma})$. Также будем считать, что отношение ***safe by*** задано парой (**G**,F), где **G** RTS, а F множество ее конечных состояний. Определим преобразование $\boldsymbol{final}(\mathbf{P},\mathbf{G},F)$, которое строит новую RTS **F** в алфавите $\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta\}$. Состояниями RTS будут все пары состояний **P** и **G**, а также два новых выделенных состояния: терминальное состояние ω и состояние γ с единственным выходящим переходом $\gamma\text{—}\gamma\rightarrow\omega$. Начальное состояние – пара начальных состояний.

Формально $\mathbf{F}$ = LTS($(V_{\mathbf{P}}\times V_{\mathbf{G}})\cup\{\omega,\gamma\}$, $\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta\}$, $E_{\mathbf{F}}$, (p_0,g_0)), где множество переходов $E_{\mathbf{F}}$ – наименьшее множество, порождаемое следующими правилами вывода: $\forall p\in V_{\mathbf{P}}\ \forall g\in V_{\mathbf{G}}\ \forall u\in\mathbf{L}\cup\mathbf{R}\ \forall Q\in\mathbf{Q}$

1. $p\text{—}u\rightarrow p` \ \&\ g\text{—}u\rightarrow g` \ \vdash\ (p,g)\text{—}u\rightarrow(p`,g`)$,
2. $p\text{—}\Delta\rightarrow p` \ \vdash\ (p,g)\text{—}\Delta\rightarrow\omega$,
3. $p\text{—}u\rightarrow p` \ \&\ p`\text{—}\gamma\rightarrow \ \vdash\ (p,g)\text{—}u\rightarrow\gamma$,
4. $\vdash\ \gamma\text{—}\gamma\rightarrow\omega$,
5. $g\text{—}Q\rightarrow g` \ \vdash\ (p,g)\text{—}Q\rightarrow\omega$.

Правило вывода 1 гарантирует наличие в RTS всех безопасных трасс. Правила вывода 2,3 и 4 сохраняют все финальные суффиксы безопасных трасс, которые были в p-модели. Тем самым, первые четыре правила сохраняют все трассы p-модели. Правило вывода 5 добавляет Q-финальные суффиксы трасс, задаваемые отношением ***safe by***. Заметим, что в правиле 2 $p`=\{\omega\}$, а в правиле 5 состояние g` конечное. Состояния, не достижимые по финальным трассам, будут недостижимы, то есть мы получим RTS, множество простых трасс которой является f-моделью, а это и есть финальная RTS.

Все состояния RTS **P** будут входить в левые части пар, представляющих достижимые состояния RTS **F** (кроме состояния $\{\omega\}$, вместо которого будет состояние ω, и начал γ-переходов, вместо которых будет состояние γ). Также все состояния RTS **G** будут входить в правые части пар, представляющих достижимые состояния RTS **F** (кроме конечных состояний, вместо которых будет состояние ω).

Поскольку RTS – это детерминированная LTS, каждая ее трасса σ заканчивается только в одном состоянии s, и для всех трасс, заканчивающихся в s, безопасность кнопок после них по отношению $\boldsymbol{safe}_{\gamma\Delta}$ зависит только от этого состояния. В финальной RTS отношение ***safe by*** определяется равным отношению ***safe in***, и безопасность кнопки по ***safe in*** после трассы зависит только от состояния в конце трассы. Определим: $\forall s\in V_{\mathbf{U}}\ \forall P\in\mathbf{R}\cup\mathbf{Q}$: P ***safe in*** s $\triangleq$ P $\boldsymbol{safe}_{\gamma\Delta}$ s & $(P\in\mathbf{Q}\Rightarrow s\text{—}P\not\rightarrow)$.

Очевидно, если **F** ***after*** σ = {s}, то P ***safe in tr***(**F**) ***after*** $\sigma\Leftrightarrow$ P ***safe in*** s.

Исследуем вопрос о конечности финальной RTS ***frts***(**S**,***safe by***). Достаточным условием является конечность каждого шага построения:

$\mathbf{S}\text{—}_{\mathbf{R}}\rightarrow\mathbf{S}_{\mathbf{R}}\text{—}\boldsymbol{determ}\rightarrow\boldsymbol{rts}(\mathbf{S})\text{—}\boldsymbol{delete}\rightarrow\boldsymbol{prts}(\mathbf{S})\text{—}\boldsymbol{final}\rightarrow\boldsymbol{frts}(\mathbf{S},\boldsymbol{safe\ by})$.

Для конечности ***prts***(**S**) достаточно конечности исходной LTS **S** (конечность множества достижимых переходов) и семейства **R**. При этих условиях для конечности ***frts***(**S**) достаточно конечности регулярности отношения ***safe by***. Заметим, что мы не требует конечности алфавита **L** и конечности семейства **Q**.

10. Заключение

В работе исследованы различные модели спецификации. В качестве результата предложены финальные трассовые и RTS-модели, последние дают возможность конечного представления регулярных финальных трассовых моделей.

Финальная RTS как модель спецификации обладает целым рядом полезных для тестирования свойств, выгодно отличающих ее от LTS-модели, **R**- и p-моделей, а также от RTS для **R**- и p-моделей:

1. **Детерминизм**. RTS детерминирована, следовательно, каждая трасса, по которой нужно генерировать тесты, заканчивается в одном состоянии.
2. **Безопасные трассы**. Безопасные трассы спецификации – это все ее простые трассы, заканчивающиеся в состояниях, отличных от ω и γ.
3. **Безопасные кнопки**. Кнопка P безопасна по ***safe by*** в финальной RTS **F** после неразрушающей трассы σ тогда и только тогда, когда она безопасна по ***safe in*** в конечном состоянии трассы s: P ***safe by tr***(**F**) ***after*** σ ⇔ P ***safe in*** s, где **F** ***after*** σ = {s}.

Также предложены алгоритмы преобразования LTS-модели с заданным отношением ***safe by*** в финальную RTS-модель. Если конечна исходная LTS **S** и семейство **R**, а отношение ***safe by*** регулярно, то финальная RTS конечна и строится за конечное время.

В то же время множество финальных трасс, хотя и достаточно, но, вообще говоря, не необходимо для генерации тестов. Это объясняется тем, что среди безопасных трасс спецификации могут встречаться неконформные трассы, то есть трассы, которых не может быть ни в одной конформной реализации. Понятно, что по таким трассам не нужно генерировать тесты. Такие трассы можно удалить из множества трасс, по которым генерируются тесты с помощью ∇-пополнения. Исследованию неконформных трасс спецификации и построению алгоритмов, удаляющих такие трассы из спецификации, посвящены наши работы [16],[17].

Доказательства утверждений

13. Доказательство Теорема 2:

1. Пусть $\langle\gamma\rangle \in \boldsymbol{\Psi}$.

Тогда $\langle\gamma\rangle \in \boldsymbol{ptr}(\boldsymbol{\Psi})$. Имеем $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) = \varnothing = \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$.

Утверждение доказано.

2. Пусть $\langle\gamma\rangle \notin \boldsymbol{\Psi}$.

Будем вести доказательство индукцией по трассе σ.

2.1. База индукции.

Из $\langle\gamma\rangle\notin\boldsymbol{\Psi}$ следует $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Из $\langle\gamma\rangle\notin\boldsymbol{\Psi}$ следует $\langle\gamma\rangle\notin\boldsymbol{ptr}(\boldsymbol{\Psi})$, что влечет $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$.

2.2. Шаг индукции. Пусть $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})\cap\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$ и $P\in\mathbf{R}\cup\mathbf{Q}$.

2.2.1. Докажем, что $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \sigma \Rightarrow P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{ptr}(\boldsymbol{\Psi})\ \boldsymbol{after}\ \sigma$).

Допустим обратное. Тогда либо $\sigma\cdot\langle\Delta\rangle\in\boldsymbol{ptr}(\boldsymbol{\Psi})$, либо $\sigma\cdot\langle z,\gamma\rangle\in\boldsymbol{ptr}(\boldsymbol{\Psi})$ для некоторого $z\in P$. Поскольку $\boldsymbol{ptr}(\boldsymbol{\Psi})\subseteq\boldsymbol{\Psi}$, имеем $\sigma\cdot\langle\Delta\rangle\in\boldsymbol{\Psi}$, либо $\sigma\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$ для некоторого $z\in P$. Но это противоречит условию $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \sigma$. Следовательно, наше допущение не верно, и импликация доказана.

2.2.2. Докажем, что $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \sigma \Leftarrow P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{ptr}(\boldsymbol{\Psi})\ \boldsymbol{after}\ \sigma$).

Допустим обратное. Тогда либо $\sigma\cdot\langle\Delta\rangle\in\boldsymbol{\Psi}$, либо $\sigma\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$ для некоторого $z\in P$. Поскольку $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, имеем либо $\langle\Delta\rangle\in\boldsymbol{fs}(\boldsymbol{\Psi},\sigma)$, либо $\langle z,\gamma\rangle\in\boldsymbol{fs}(\boldsymbol{\Psi},\sigma)$ для некоторого $z\in P$. Но тогда либо $\sigma\cdot\langle\Delta\rangle\in\boldsymbol{ptr}(\boldsymbol{\Psi})$, либо $\sigma\cdot\langle z,\gamma\rangle\in\boldsymbol{ptr}(\boldsymbol{\Psi})$ для некоторого $z\in P$. Но это противоречит условию $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{ptr}(\boldsymbol{\Psi})\ \boldsymbol{after}\ \sigma$. Следовательно, наше допущение не верно, и обратная импликация доказана.

2.2.3. Для $u\in\mathbf{L}\cup\mathbf{R}$ докажем $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) \Rightarrow \sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$.

Поскольку $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, найдется такое $P\in\boldsymbol{but}(u)$, что $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \sigma$. По доказанному в п.2.2.1 $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{ptr}(\boldsymbol{\Psi})\ \boldsymbol{after}\ \sigma$). Поскольку $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, имеем $\sigma\cdot\langle u\rangle\in\boldsymbol{ptr}(\boldsymbol{\Psi})$. Следовательно, $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$.

2.2.4. Для $u\in\mathbf{L}\cup\mathbf{R}$ докажем $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) \Leftarrow \sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$.

Поскольку $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$, найдется такое $P\in\boldsymbol{but}(u)$, что $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{ptr}(\boldsymbol{\Psi})\ \boldsymbol{after}\ \sigma$. По доказанному в п.2.2.2 $P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \sigma$). Поскольку $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Psi}))$, имеем $\sigma\cdot\langle u\rangle\in\boldsymbol{ptr}(\boldsymbol{\Psi})$. Поскольку $\boldsymbol{ptr}(\boldsymbol{\Psi})\subseteq\boldsymbol{\Psi}$, имеем $\sigma\cdot\langle u\rangle\in\boldsymbol{\Psi}$. Следовательно, $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

14. Доказательство Теорема 3:

1. Прямая импликация. Пусть $\boldsymbol{ptr}(\boldsymbol{\Psi})=\boldsymbol{\Psi}$.

Докажем, что $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) = \{\sigma\in\boldsymbol{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Psi}\}$.

1.1. Если $\langle\gamma\rangle\in\boldsymbol{\Psi}$, то $\boldsymbol{ptr}(\boldsymbol{\Psi})=\{\epsilon,\langle\gamma\rangle\}$. Тогда $\boldsymbol{\Psi}=\{\epsilon,\langle\gamma\rangle\}$ и утверждение очевидно.

1.2. Пусть $\langle\gamma\rangle\notin\boldsymbol{\Psi}$. По определению $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$ влечет $\sigma\in\boldsymbol{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^*$.

Поскольку $\langle\gamma\rangle\notin\boldsymbol{fs}(\boldsymbol{\Psi},\sigma)$, а $\boldsymbol{ptr}(\boldsymbol{\Psi})=\boldsymbol{\Psi}$, имеем $\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Psi}$.

Следовательно, $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) \subseteq \{\sigma\in\boldsymbol{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Psi}\}$.

Покажем обратную вложенность. Допустим противное: существует трасса $\sigma\in\Psi\cap(\mathbf{L}\cup\mathbf{R})^*$ такая, что $\sigma\cdot\langle\gamma\rangle\notin\Psi$, но $\sigma\notin\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$.

Поскольку $\langle\gamma\rangle\notin\Psi$, имеем $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$. Следовательно, у трассы σ есть префикс $\mu\cdot\langle u\rangle$ такой, что $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$, но $\mu\cdot\langle u\rangle\notin\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$.

Следовательно, либо $u=\Delta$, либо $u=\gamma$, либо $u\in\mathbf{L}\cup\mathbf{R}$ и u $\boldsymbol{safe}_{\gamma\Delta}$ Ψ ***after*** μ. Случаев $u=\Delta$ и $u=\gamma$ быть не может, так как $\mu\cdot\langle u\rangle\le\sigma\in(\mathbf{L}\cup\mathbf{R})^*$.

Следовательно, $u\in\mathbf{L}\cup\mathbf{R}$ и u $\boldsymbol{safe}_{\gamma\Delta}$ Ψ ***after*** μ.

Если $\mu\cdot\langle u,\gamma\rangle\notin\Psi$, то $\langle u\rangle\notin\boldsymbol{fs}(\Psi,\mu)$, поэтому $\mu\cdot\langle u\rangle\notin\boldsymbol{ptr}(\Psi)=\Psi$, что противоречит префикс-замкнутости Ψ, поскольку $\mu\cdot\langle u\rangle\le\sigma\in\Psi$.

Если $\mu\cdot\langle u,\gamma\rangle\in\Psi$, то возможны два варианта: 1) $\mu\cdot\langle u\rangle=\sigma$ и 2) $\mu\cdot\langle u\rangle<\sigma$. Случай 1 противоречит $\sigma\cdot\langle\gamma\rangle\notin\Psi$. В случае 2 найдется такое $v\in\mathbf{L}\cup\mathbf{R}$, что $\mu\cdot\langle u,v\rangle\le\sigma$. Но $\langle u,v\rangle\notin\boldsymbol{fs}(\Psi,\mu)$, поэтому $\mu\cdot\langle u,v\rangle\notin\boldsymbol{ptr}(\Psi)=\Psi$, что противоречит префикс-замкнутости Ψ, поскольку $\mu\cdot\langle u,v\rangle\le\sigma\in\Psi$.

Мы пришли к противоречию, следовательно, наше допущение не верно и обратная вложенность доказана.

2. Обратная импликация. Пусть выполнены условия обратной импликации. Докажем, что $\boldsymbol{ptr}(\Psi)=\Psi$.

2.1. Пусть $\langle\gamma\rangle\in\Psi$.

Тогда $\Psi=\{\epsilon,\langle\gamma\rangle\}$ и $\boldsymbol{ptr}(\Psi)=\{\epsilon,\langle\gamma\rangle\}$.

2.2. Пусть $\langle\gamma\rangle\notin\Psi$.

Тогда $\boldsymbol{Safe}_{\gamma\Delta}(\Psi)=\{\sigma\in\Psi\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\Psi\}$, трассы из Ψ допустимы и после отказа в них нет разрушения.

Поскольку по определению $\boldsymbol{ptr}(\Psi)\subseteq\Psi$, нам нужно доказать обратную вложенность. Допустим, это не так: существует трасса $\sigma\in\Psi$ такая, что $\sigma\notin\boldsymbol{ptr}(\Psi)$. Поскольку по определению предфинальности $\boldsymbol{Safe}_{\gamma\Delta}(\Psi)\subseteq\boldsymbol{ptr}(\Psi)$, имеем $\sigma\notin\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$. Поскольку $\langle\gamma\rangle\notin\Psi$, имеем $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$. Следовательно, трассу σ можно представить в виде $\sigma=\mu\cdot\lambda$ таком, что $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\Psi)$ максимальный безопасный префикс σ, но $\lambda\notin\boldsymbol{fs}(\Psi,\mu)$.

Поскольку все трассы из Ψ допустимы, возможны только следующие случаи: 1) $\lambda=\lambda_1\cdot\langle\Delta\rangle$, либо 2) $\lambda=\lambda_1\cdot\langle\gamma\rangle$, либо 3) $\lambda=\lambda_1$, где $\lambda_1\in(\mathbf{L}\cup\mathbf{R})^*$.

Поскольку $\boldsymbol{Safe}_{\gamma\Delta}(\Psi)=\{\sigma\in\Psi\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\Psi\}$, в этих случаях имеет место: 1) $\lambda_1=\epsilon$ и $\lambda=\langle\Delta\rangle$, 2) $\lambda_1=\epsilon$ и $\lambda=\langle\gamma\rangle$, 3) $\lambda=\langle u\rangle$, где $u\in\mathbf{L}\cup\mathbf{R}$ и $\mu\cdot\langle u,\gamma\rangle\in\Psi$.

Случай 1 противоречит $\lambda=\langle\Delta\rangle\notin\boldsymbol{fs}(\Psi,\mu)$.

В случае 2 трасса μ не может быть пустой, так как $\langle\gamma\rangle \notin \boldsymbol{\Psi}$, и не может заканчиваться отказом, так как в трассах из $\boldsymbol{\Psi}$ после отказа нет разрушения. Следовательно, трасса μ заканчивается действием, за которым в $\boldsymbol{\Psi}$ следует разрушение, что противоречит $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Рассмотрим случай 3. Поскольку в трассах из $\boldsymbol{\Psi}$ после отказа нет разрушения, должно быть $u \in \mathbf{L}$. Но тогда $\lambda = \langle u\rangle \in \boldsymbol{fs}(\boldsymbol{\Psi},\mu)$, что неверно.

Мы пришли к противоречию, следовательно, наше допущение не верно и обратная вложенность доказана.

3. Необходимость условий теоремы для обратной импликации показывается следующими примерами. В каждом из этих примеров нарушено только одно условие обратной импликации.

3.1. Выполнены все условия, кроме условия на множество неразрушающих трасс при $\langle\gamma\rangle \in \boldsymbol{\Psi}$.

Пример $\boldsymbol{\Psi} = \{\epsilon, \langle P\rangle, \langle z\rangle, \langle z,\gamma\rangle\}$, где $P \in \mathbf{R}$ и $z \in P$.

$\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) = \{\epsilon\} \neq \{\epsilon, \langle P\rangle\} = \{\sigma \in \boldsymbol{\Psi} \cap (\mathbf{L} \cup \mathbf{R})^* \mid \sigma \cdot \langle\gamma\rangle \notin \boldsymbol{\Psi}\}$,

$\boldsymbol{ptr}(\boldsymbol{\Psi}) = \{\epsilon, \langle z\rangle, \langle z,\gamma\rangle\} \neq \boldsymbol{\Psi}$.

3.2. Выполнены все условия, кроме условия $\boldsymbol{\Psi} = \{\epsilon, \langle\gamma\rangle\}$ при $\langle\gamma\rangle \in \boldsymbol{\Psi}$.

Пример $\boldsymbol{\Psi} = \{\epsilon, \langle\gamma\rangle, \langle u\rangle\}$, где $u \in \mathbf{L} \cup \mathbf{R} \cup \{\Delta,\gamma\}$. $\boldsymbol{ptr}(\boldsymbol{\Psi}) = \{\epsilon, \langle\gamma\rangle\} \neq \boldsymbol{\Psi}$.

3.3. Выполнены все условия, кроме допустимости трасс.

Пример 1 $\boldsymbol{\Psi} = \{\epsilon, \langle\Delta\rangle, \langle\Delta,u\rangle\}$, где $u \in \mathbf{L} \cup \mathbf{R} \cup \{\Delta,\gamma\}$.

$\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) = \{\sigma \in \boldsymbol{\Psi} \cap (\mathbf{L} \cup \mathbf{R})^* \mid \sigma \cdot \langle\gamma\rangle \notin \boldsymbol{\Psi}\} = \{\epsilon\}$, $\boldsymbol{ptr}(\boldsymbol{\Psi}) = \{\epsilon, \langle\Delta\rangle\}$.

Пример 2 $\boldsymbol{\Psi} = \{\epsilon, \langle z\rangle, \langle z,\gamma\rangle, \langle z,\gamma,u\rangle\}$, где $z \in \mathbf{L}$ и $u \in \mathbf{L} \cup \mathbf{R} \cup \{\Delta,\gamma\}$.

$\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) = \{\sigma \in \boldsymbol{\Psi} \cap (\mathbf{L} \cup \mathbf{R})^* \mid \sigma \cdot \langle\gamma\rangle \notin \boldsymbol{\Psi}\} = \{\epsilon\}$,

$\boldsymbol{ptr}(\boldsymbol{\Psi}) = \{\epsilon, \langle z\rangle, \langle z,\gamma\rangle\} \neq \boldsymbol{\Psi}$.

3.4. Выполнены все условия, кроме отсутствия разрушения после отказа.

Пример $\boldsymbol{\Psi} = \{\epsilon, \langle P\rangle, \langle P,\gamma\rangle, \langle z\rangle, \langle z,\gamma\rangle\}$, где $P \in \mathbf{R}$ и $z \in P$.

$\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi}) = \{\sigma \in \boldsymbol{\Psi} \cap (\mathbf{L} \cup \mathbf{R})^* \mid \sigma \cdot \langle\gamma\rangle \notin \boldsymbol{\Psi}\} = \{\epsilon\}$,

$\boldsymbol{ptr}(\boldsymbol{\Psi}) = \{\epsilon, \langle z\rangle, \langle z,\gamma\rangle\}$.

15. Доказательство Теорема 4:

1. Непустота множества. Поскольку ***d***-замыкание только добавляет трассы, ***d***-замыкание непустого множества не пусто.
2. Префикс-замкнутость. Поскольку при ***d***-замыкании отказ P удаляется после трассы μ перед каждой продолжающей ее трассой λ, ***d***-замыкание префикс-замкнутого множества префикс-замкнуто.

3. Допустимость. Поскольку при ***d***-замыкании только удаляются отказы из трасс, ***d***-замыкание множества допустимых трасс добавляет только допустимые трассы.
4. Согласованность. Поскольку при ***d***-замыкании только удаляются отказы из трасс, ***d***-замыкание множества согласованных трасс добавляет только согласованные трассы.
5. **R**-конвергентность. Поскольку при ***d***-замыкании отказ P удаляется после трассы μ перед каждой продолжающей ее трассой λ, ***d***-замыкание **R**-конвергентного множества **R**-конвергентно.
6. Замкнутость. По определению ***d***-замыкание любого множества трасс ***d***-замкнуто.
7. **R**-полнота. Поскольку при ***d***-замыкании отказ P удаляется после трассы μ перед каждой продолжающей ее трассой λ, ***d***-замыкание полного множества полно.

16. Доказательство Теорема 5:

Если $\langle\gamma\rangle \in \mathbf{\Sigma}$, то $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Sigma})=\varnothing$, поэтому $\boldsymbol{ptr}(\mathbf{\Sigma})=\{\epsilon,\langle\gamma\rangle\}$ и выполнение всех свойств p-модели очевидно. Далее будем считать, что $\langle\gamma\rangle \notin \mathbf{\Sigma}$.

1. Непустота множества. Поскольку трассовая модель не пуста и префикс-замкнута $\epsilon \in \mathbf{\Sigma}$. Поэтому по определению предфинальных трасс $\epsilon \in \boldsymbol{ptr}(\mathbf{\Sigma})$, следовательно, множество $\boldsymbol{ptr}(\mathbf{\Sigma})$ не пусто.
2. Префикс-замкнутость. Множество $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Sigma})$ префикс-замкнуто по определению. По определению префикс финального продолжения неразрушающей трассы является либо префиксом этой трассы, либо ее финальным продолжением. Тем самым множество $\boldsymbol{ptr}(\mathbf{\Sigma})$ префикс-замкнуто.
3. Допустимость. Все трассы трассовой модели допустимы, в том числе и ее предфинальные трассы.
4. Согласованность. Все трассы трассовой модели согласованы, в том числе и ее предфинальные трассы.
5. **R**-конвергентность. Пусть σ предфинальная трасса. Рассмотрим два случая.

5.1. $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Sigma})$.

5.1.1. Если **R**-кнопка P разрушающая в $\mathbf{\Sigma}$ после σ, то $\sigma\cdot\langle\Delta\rangle \in \mathbf{\Sigma}$ или $\sigma\cdot\langle z,\gamma\rangle \in \mathbf{\Sigma}$ для некоторого $z \in P$. А тогда $\sigma\cdot\langle\Delta\rangle \in \boldsymbol{ptr}(\mathbf{\Sigma})$ или $\sigma\cdot\langle z,\gamma\rangle \in \boldsymbol{ptr}(\mathbf{\Sigma})$ для некоторого $z \in P$.

5.1.2. Если **R**-кнопка P неразрушающая в $\mathbf{\Sigma}$ после σ, то по **R**-конвергентности трассовой модели в $\mathbf{\Sigma}$ имеется продолжение трассы σ неразрушающим наблюдением $u \in \boldsymbol{obs}(P)$, и трасса $\sigma\cdot\langle u\rangle$ неразрушающая и, следовательно, предфинальная.

В обоих случаях трасса σ **R**-конвергентна в $\boldsymbol{ptr}(\mathbf{\Sigma})$.

5.2. $\sigma \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Тогда по определению финального продолжения трасса σ либо заканчивается дивергенцией или разрушением, либо продолжается разрушением и, следовательно, **R**-конвергентна в $\boldsymbol{ptr}(\boldsymbol{\Sigma})$.

6. **R**-полнота. Пусть трасса $\mu\cdot\lambda \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$, трасса μ заканчивается некоторым отказом и не продолжается в $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ действиями из отказа P. Нужно доказать, что $\mu\cdot\langle P\rangle\cdot\lambda \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$.

6.1. Покажем, что трасса $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Трасса μ как префикс предфинальной трассы $\mu\cdot\lambda$ по доказанной префикс-замкнутости также предфинальна. А тогда, поскольку трасса μ заканчивается на отказ, по определению предфинальных трасс $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

6.2. Покажем, что P $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** μ. Действительно, в противном случае в $\boldsymbol{\Sigma}$ имелась бы либо 1) предфинальная трасса $\mu\cdot\langle\Delta\rangle$, либо 2) предфинальная трасса $\mu\cdot\langle z,\gamma\rangle$ для некоторого $z \in P$. Случая 1 не может быть, так как трасса μ заканчивается отказом и по согласованности **R**-модели не может продолжаться в $\boldsymbol{\Sigma}$ дивергенцией. В случае 2 $\mu\cdot\langle z,\gamma\rangle \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$, чего быть не может, так как трасса μ не продолжается в $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ действиями из отказа P.

6.3. Покажем, что трасса μ не продолжается в $\boldsymbol{\Sigma}$ действиями из отказа P.

Действительно, поскольку P $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** μ, такие продолжения были бы неразрушающими в $\boldsymbol{\Sigma}$ и, следовательно, финальными, что противоречит тому, что трасса μ не продолжается в $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ действиями из отказа P.

6.4. Покажем, что $\mu\cdot\langle P\rangle \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$. Поскольку $\boldsymbol{\Sigma}$ **R**-конвергентна и трасса μ не продолжается в $\boldsymbol{\Sigma}$ действиями из отказа P, должно быть $\mu\cdot\langle P\rangle \in \boldsymbol{\Sigma}$. Поскольку P $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** μ, $\mu\cdot\langle P\rangle \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Поскольку $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$, у предфинальной трассы $\mu\cdot\lambda$ имеется максимальный неразрушающий префикс $\mu\cdot\lambda_1$.

6.5. Покажем, что $\mu\cdot\langle P\rangle\cdot\lambda_1 \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Поскольку $\mu\cdot\langle P\rangle\cdot\lambda \in \boldsymbol{\Sigma}$, по префикс-замкнутости **R**-модели $\mu\cdot\langle P\rangle\cdot\lambda_1 \in \boldsymbol{\Sigma}$. Допустим $\mu\cdot\langle P\rangle\cdot\lambda_1 \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$. Тогда, поскольку $\mu\cdot\langle P\rangle \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$, у трассы λ_1 имеется такой префикс $\lambda_2\cdot\langle u\rangle$, что $\mu\cdot\langle P\rangle\cdot\lambda_2 \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$, а $\mu\cdot\langle P\rangle\cdot\lambda_2\cdot\langle u\rangle \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$. Отсюда, поскольку $\mu\cdot\langle P\rangle\cdot\lambda \in \boldsymbol{\Sigma}$ и, следовательно, $\mu\cdot\langle P\rangle\cdot\lambda_2\cdot\langle u\rangle \in \boldsymbol{\Sigma}$, имеем, либо $u=\Delta$, либо $u \in \mathbf{L}\cup\mathbf{R}$ и u $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Sigma}$ ***after*** $\mu\cdot\langle P\rangle\cdot\lambda_2$. Поэтому в $\boldsymbol{\Sigma}$ имеется трасса $\mu\cdot\langle P\rangle\cdot\lambda_2\cdot\langle\Delta\rangle$ или для каждой кнопки $R \in \boldsymbol{but}(u)$ имеется трасса $\mu\cdot\langle P\rangle\cdot\lambda_2\cdot\langle z,\gamma\rangle$ для некоторого $z \in R$. Но тогда по замкнутости **R**-модели в $\boldsymbol{\Sigma}$ имеется трасса $\mu\cdot\lambda_2\cdot\langle\Delta\rangle$ или для каждой кнопки $R \in \boldsymbol{but}(u)$ имеется трасса $\mu\cdot\lambda_2\cdot\langle z,\gamma\rangle$ для

некоторого $z \in R$, что, поскольку $\mu\cdot\lambda_2\cdot\langle u\rangle \le \mu\cdot\lambda_1$, противоречит тому, что $\mu\cdot\lambda_1 \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

6.6. Поскольку $\mu\cdot\lambda_1$ максимальный неразрушающий префикс предфинальной трассы $\mu\cdot\lambda$, трасса $\mu\cdot\lambda$ является финальным продолжением трассы $\mu\cdot\lambda_1$: 1) $\mu\cdot\lambda=\mu\cdot\lambda_1$, или 2) $\mu\cdot\lambda=\mu\cdot\lambda_1\cdot\langle\Delta\rangle$, или 3) $\mu\cdot\lambda=\mu\cdot\lambda_1\cdot\langle z,\gamma\rangle$, или 4) $\mu\cdot\lambda=\mu\cdot\lambda_1\cdot\langle z\rangle$ и $\mu\cdot\lambda_1\cdot\langle z,\gamma\rangle \in \boldsymbol{\Sigma}$, или 5) $\mu\cdot\lambda=\mu\cdot\lambda_1\cdot\langle Q\rangle$, где $Q \in \mathbf{Q}$. Случая 5 быть не может, так как **R**-модель $\boldsymbol{\Sigma}$ не содержит **Q**-отказов.

Поскольку $\mu\cdot\lambda \in \boldsymbol{\Sigma}$ и трасса μ заканчивается отказом и не продолжается в $\boldsymbol{\Sigma}$ действиями из отказа P, по **R**-полноте трассовой модели: 1) $\mu\cdot\langle P\rangle\cdot\lambda=\mu\cdot\langle P\rangle\cdot\lambda_1 \in \boldsymbol{\Sigma}$, или 2) $\mu\cdot\langle P\rangle\cdot\lambda=\mu\cdot\langle P\rangle\cdot\lambda_1\cdot\langle\Delta\rangle \in \boldsymbol{\Sigma}$, или 3) $\mu\cdot\langle P\rangle\cdot\lambda=\mu\cdot\langle P\rangle\cdot\lambda_1\cdot\langle z,\gamma\rangle \in \boldsymbol{\Sigma}$, или 4) $\mu\cdot\langle P\rangle\cdot\lambda=\mu\cdot\langle P\rangle\cdot\lambda_1\cdot\langle z\rangle \in \boldsymbol{\Sigma}$ и $\mu\cdot\langle P\rangle\cdot\lambda_1\cdot\langle z,\gamma\rangle \in \boldsymbol{\Sigma}$. Поэтому, поскольку $\mu\cdot\langle P\rangle\cdot\lambda_1 \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$, во всех случаях трасса $\mu\cdot\langle P\rangle\cdot\lambda$ является финальным продолжением трассы $\mu\cdot\langle P\rangle\cdot\lambda_1$. Следовательно, $\mu\cdot\langle P\rangle\cdot\lambda \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$, что и требовалось доказать.

7. предфинальность: $\boldsymbol{ptr}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))=\boldsymbol{ptr}(\boldsymbol{\Sigma})$.

По определению предфинальных трасс $\boldsymbol{ptr}(\boldsymbol{ptr}(\boldsymbol{\Sigma})) \subseteq \boldsymbol{ptr}(\boldsymbol{\Sigma})$. Нам достаточно показать, что $\boldsymbol{ptr}(\boldsymbol{ptr}(\boldsymbol{\Sigma})) \supseteq \boldsymbol{ptr}(\boldsymbol{\Sigma})$. Рассмотрим трассу $\sigma \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$. По определению предфинальных трасс $\sigma \in \boldsymbol{\Sigma}$ и возможны два варианта.

7.1. $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$. Покажем, что $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, из чего будет следовать, что $\sigma \in \boldsymbol{ptr}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$.

Допустим противное. Тогда, поскольку $\langle\gamma\rangle \notin \boldsymbol{\Sigma}$, у трассы σ имеется такой префикс $\mu\cdot\langle u\rangle$, что $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, а $\mu\cdot\langle u\rangle \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$. Поскольку $\sigma \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$, $\mu\cdot\langle u\rangle \in \boldsymbol{ptr}(\boldsymbol{\Sigma})$. Следовательно, $u=\Delta$ или $u \in \mathbf{L}\cup\mathbf{R}$ и u ***safe***$_{\gamma\Delta}$ $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ ***after*** μ. Тогда в $\boldsymbol{ptr}(\boldsymbol{\Sigma})$ либо имеется трасса $\mu\cdot\langle\Delta\rangle$, либо для каждой кнопки $R \in \boldsymbol{but}(u)$ имеется трасса $\mu\cdot\langle z,\gamma\rangle$ для некоторого $z \in R$. Но тогда такие трассы есть и в $\boldsymbol{\Sigma}$, что противоречит тому, что $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$. Следовательно, $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$.

7.2. $\sigma \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Тогда, поскольку $\langle\gamma\rangle \notin \boldsymbol{\Sigma}$, σ является финальным продолжением некоторой неразрушающей трассы μ. По доказанному $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, что влечет $\sigma \in \boldsymbol{ptr}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$.

8. финально-замкнутость: Пусть $\mu\cdot\langle P\rangle\cdot\lambda \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, где $P \in \mathbf{R}$. Нужно доказать, что $\mu\cdot\lambda \in \boldsymbol{UnSafe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$ или $\mu\cdot\lambda \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$ и $\boldsymbol{fs}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\mu\cdot\langle P\rangle\cdot\lambda) \subseteq \boldsymbol{fs}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\mu\cdot\lambda)$.

Поскольку $\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, имеем $\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{ptr}(\boldsymbol{\Sigma})$ и $\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{\Sigma}$. По замкнутости **R**-модели $\mu\cdot\lambda\in\boldsymbol{\Sigma}$. Поскольку $\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{ptr}(\boldsymbol{\Sigma})$, по доказанной префикс-замкнутости $\mu\cdot\langle P\rangle\in\boldsymbol{ptr}(\boldsymbol{\Sigma})$. А тогда, поскольку $P\in\mathbf{R}$, должно быть $\mu\cdot\langle P\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$, поскольку трасса $\mu\cdot\langle P\rangle$ заканчивается **R**-отказом. По префикс-замкнутости неразрушающих трасс имеем $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$. Далее рассмотрим два случая.

8.1. $\mu\cdot\lambda\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Тогда по доказанному в п.7.1 $\mu\cdot\lambda\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$. Пусть $\kappa\in\boldsymbol{fs}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\mu\cdot\langle P\rangle\cdot\lambda)$. Тогда 1) $\kappa=\epsilon$, или 2) $\kappa=\langle\Delta\rangle$, или 3) $\kappa=\langle z,\gamma\rangle$, 4) $\kappa=\langle z\rangle$ и $\mu\cdot\langle P\rangle\cdot\lambda\cdot\langle z,\gamma\rangle\in\boldsymbol{ptr}(\boldsymbol{\Sigma})$, или 5) $\mu\cdot\lambda=\mu\cdot\lambda_1\cdot\langle Q\rangle$, где $Q\in\mathbf{Q}$. Случая 5 быть не может, так как **R**-модель $\boldsymbol{\Sigma}$ не содержит **Q**-отказов. Поскольку $\boldsymbol{ptr}(\boldsymbol{\Sigma})\subseteq\boldsymbol{\Sigma}$, $\boldsymbol{\Sigma}$ ***d***-замкнуто, имеем $\kappa\in\boldsymbol{fs}(\boldsymbol{\Sigma},\mu\cdot\lambda)$. Поскольку $\mu\cdot\lambda\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$, имеем $\kappa\in\boldsymbol{fs}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\mu\cdot\lambda)$. Следовательно, $\boldsymbol{fs}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\mu\cdot\langle P\rangle\cdot\lambda)\subseteq\boldsymbol{fs}(\boldsymbol{ptr}(\boldsymbol{\Sigma}),\mu\cdot\lambda)$.

8.2. $\mu\cdot\lambda\notin\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Sigma})$.

Тогда, поскольку $\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, имеем $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$. Поэтому у трассы λ найдется такой префикс $\lambda_1\cdot\langle u\rangle$, что $\mu\cdot\lambda_1\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$, но $\mu\cdot\lambda_1\cdot\langle u\rangle\notin\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$.

Следовательно, $u=\Delta$ или $u\in\mathbf{L}\cup\mathbf{R}$ и $u\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Sigma}\ \boldsymbol{after}\ \mu\cdot\lambda_1$. Поскольку, $\mu\cdot\langle P\rangle\cdot\lambda_1\cdot\langle u\rangle\le\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{ftr}(\boldsymbol{\Sigma}))$, трасса $\mu\cdot\lambda_1\cdot\langle u\rangle$ является $\mathbf{L}\cup\mathbf{R}$-трассой, следовательно, случая $u=\Delta$ быть не может. Поэтому $u\in\mathbf{L}\cup\mathbf{R}$ и $u\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Sigma}\ \boldsymbol{after}\ \mu\cdot\lambda_1$. Тогда либо $\mu\cdot\lambda_1\cdot\langle\Delta\rangle\in\boldsymbol{\Sigma}$, либо для каждой кнопки $R\in\boldsymbol{but}(u)$ имеется трасса $\mu\cdot\lambda_1\cdot\langle z,\gamma\rangle\in\boldsymbol{\Sigma}$ для некоторого $z\in R$. А тогда по определению предфинальных трасс либо $\mu\cdot\lambda_1\cdot\langle\Delta\rangle\in\boldsymbol{ptr}(\boldsymbol{\Sigma})$, либо для каждой кнопки $R\in\boldsymbol{but}(u)$ имеется трасса $\mu\cdot\lambda_1\cdot\langle z,\gamma\rangle\in\boldsymbol{ptr}(\boldsymbol{\Sigma})$ для некоторого $z\in R$. Следовательно, в обоих случаях трасса $\mu\cdot\lambda\in\boldsymbol{UnSafe}_{\gamma\Delta}(\boldsymbol{ptr}(\boldsymbol{\Sigma}))$.

17. Доказательство Теорема 6:

1. Сначала докажем, что $\boldsymbol{D}(\boldsymbol{\Psi})$ является **R**-моделью.

По определению p-модели $\boldsymbol{\Psi}$ она является незамкнутой **R**-моделью. Поэтому по теореме 4 $\boldsymbol{D}(\boldsymbol{\Psi})$ является **R**-моделью.

2. Теперь покажем, что $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))=\boldsymbol{\Psi}$.

2.1. Пусть $\langle\gamma\rangle\in\boldsymbol{\Psi}$. Тогда по определению предфинальных трасс $\boldsymbol{ptr}(\boldsymbol{\Psi})=\{\epsilon,\langle\gamma\rangle\}$. По определению p-модели, $\boldsymbol{\Psi}=\boldsymbol{ptr}(\boldsymbol{\Psi})=\{\epsilon,\langle\gamma\rangle\}$. По

определению $\boldsymbol{d}$-замыкания $\boldsymbol{D}(\boldsymbol{\Psi})=\boldsymbol{\Psi}=\{\epsilon,\langle\gamma\rangle\}$. По определению предфинальных трасс $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))=\{\epsilon,\langle\gamma\rangle\}$. Следовательно, $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))=\{\epsilon,\langle\gamma\rangle\}=\boldsymbol{\Psi}$.

2.2. Пусть $\langle\gamma\rangle\notin\boldsymbol{\Psi}$. Тогда по согласованности р-модели $\langle\gamma\rangle\notin\boldsymbol{D}(\boldsymbol{\Psi})$. А тогда $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$ и $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$.

2.2.1. Сначала покажем, что $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))\subseteq\boldsymbol{\Psi}$.

2.2.1.1. Покажем, что $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))\subseteq\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Пусть трасса $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$.

Допустим противное: $\sigma\notin\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Поскольку $\epsilon\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, у трассы σ найдется префикс $\mu\cdot\langle u\rangle$ такой, что $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, а $\mu\cdot\langle u\rangle\notin\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Поскольку $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$, а $\mu\cdot\langle u\rangle\le\sigma$, имеем $u\ne\Delta$ и $u\ne\gamma$, то есть $u\in\mathbf{L}\cup\mathbf{R}$ и $u\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Psi}\ \boldsymbol{after}\ \mu$.

А тогда либо трасса $\mu\cdot\langle\Delta\rangle\in\boldsymbol{\Psi}$, либо для каждой кнопки $R\in\boldsymbol{but}(u)$ имеется трасса $\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$ для некоторого $z\in R$. А тогда $\mu\cdot\langle\Delta\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$, либо для каждой кнопки $R\in\boldsymbol{but}(u)$ имеется трасса $\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$ для некоторого $z\in R$. Следовательно, $u\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{D}(\boldsymbol{\Psi})\ \boldsymbol{after}\ \mu$. Но это противоречит тому, что $\mu\cdot\langle u\rangle\le\sigma$ и $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$. Мы пришли к противоречию, следовательно, $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, что и требовалось доказать.

2.2.1.2. Покажем, что $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))\setminus\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))\subseteq\boldsymbol{\Psi}\setminus\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Пусть трасса $\sigma\in\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi}))\setminus\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$. Тогда трасса σ является финальным продолжением в $\boldsymbol{D}(\boldsymbol{\Psi})$ некоторой трассы $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$ и $\sigma\ne\mu$. Тогда по доказанному $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Нам надо показать, что σ является финальным продолжением μ в $\boldsymbol{\Psi}$ и $\sigma\notin\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Поскольку $\sigma\ne\mu$, по определению финального продолжения и отсутствию в р-модели $\mathbf{Q}$-отказов возможны три варианта: 1) $\sigma=\mu\cdot\langle\Delta\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$, 2) $\sigma=\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$, 3) $\sigma=\mu\cdot\langle z\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$ и $\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$.

По определению $\boldsymbol{d}$-замыкания найдется такая трасса $\mu`\in\boldsymbol{\Psi}$, что $\mu\in\boldsymbol{d}(\mu`)$ и 1) $\mu`\cdot\langle\Delta\rangle\in\boldsymbol{\Psi}$, 2) $\mu`\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$, 3) $\mu`\cdot\langle z\rangle\in\boldsymbol{\Psi}$ и $\mu`\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$. Отсюда по свойству предфинальности $\boldsymbol{\Psi}$ трасса $\mu`\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

А тогда по финально-замкнутости $\boldsymbol{\Psi}$ либо 1) $\mu\in\boldsymbol{UnSafe}_{\gamma\Delta}(\boldsymbol{\Psi})$, либо 2) $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$ и $\boldsymbol{fs}(\boldsymbol{\Psi},\mu`)\subseteq\boldsymbol{fs}(\boldsymbol{\Psi},\mu)$.

Случай 1 противоречит $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Следовательно, 1) $\sigma=\mu\cdot\langle\Delta\rangle\in\boldsymbol{\Psi}$, 2) $\sigma=\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$, 3) $\sigma=\mu\cdot\langle z\rangle\in\boldsymbol{\Psi}$ и $\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{\Psi}$.

Следовательно, σ является финальным продолжением μ в $\boldsymbol{\Psi}$.

Во всех этих случаях также $\sigma \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

2.2.2. Теперь покажем, что $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi})) \supseteq \boldsymbol{\Psi}$.

2.2.2.1. Покажем, что $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi})) \supseteq \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Пусть трасса $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Тогда $\sigma \in \boldsymbol{\Psi}$ и, следовательно, $\sigma \in \boldsymbol{D}(\boldsymbol{\Psi})$. Допустим, утверждение не верно: $\sigma \in \boldsymbol{D}(\boldsymbol{\Psi}) \setminus \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$.

Поскольку $\epsilon \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, у трассы σ найдется префикс $\mu \cdot \langle u \rangle$ такой, что $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$, а $\mu \cdot \langle u \rangle \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$.

Поскольку $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, а $\mu \cdot \langle u \rangle \le \sigma$, имеем $u \ne \Delta$ и $u \ne \gamma$, то есть $u \in \mathbf{L} \cup \mathbf{R}$ и тогда u $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{D}(\boldsymbol{\Psi})$ ***after*** μ.

Следовательно, либо 1) $\mu \cdot \langle \Delta \rangle \in \boldsymbol{D}(\boldsymbol{\Psi})$, либо 2) для каждой кнопки $R \in \boldsymbol{but}(u)$ имеется трасса $\mu \cdot \langle z_R, \gamma \rangle \in \boldsymbol{D}(\boldsymbol{\Psi})$ для некоторого $z_R \in R$.

Тогда по определению ***d***-замыкания либо 1) найдется такая трасса $\mu` \in \boldsymbol{\Psi}$, что $\mu \in \boldsymbol{d}(\mu`)$ и $\mu` \cdot \langle \Delta \rangle \in \boldsymbol{\Psi}$, либо 2) для каждой кнопки $R \in \boldsymbol{but}(u)$ найдется такая трасса $\mu_R` \in \boldsymbol{\Psi}$, что $\mu \in \boldsymbol{d}(\mu_R`)$ и $\mu_R` \cdot \langle z_R, \gamma \rangle \in \boldsymbol{\Psi}$.

По свойству предфинальности $\boldsymbol{\Psi}$ 1) трасса $\mu` \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$ или 2) все трассы $\mu_R` \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Поскольку $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$, по п.2.2.2.1.1 $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

А тогда по финально-замкнутости $\boldsymbol{\Psi}$ имеем 1) $\langle \Delta \rangle \in \boldsymbol{fs}(\boldsymbol{\Psi}, \mu`) \subseteq \boldsymbol{fs}(\boldsymbol{\Psi}, \mu)$ либо 2) $\langle z_R, \gamma \rangle \in \boldsymbol{fs}(\boldsymbol{\Psi}, \mu_R`) \subseteq \boldsymbol{fs}(\boldsymbol{\Psi}, \mu)$ для каждой кнопки R.

Следовательно, 1) трасса $\mu \cdot \langle \Delta \rangle \in \boldsymbol{\Psi}$ либо 2) для каждой кнопки $R \in \boldsymbol{but}(u)$ трасса $\mu \cdot \langle z_R, \gamma \rangle \in \boldsymbol{\Psi}$.

Тем самым, u $\boldsymbol{safe}_{\gamma\Delta}$ $\boldsymbol{\Psi}$ ***after*** μ, что, поскольку $\mu \cdot \langle u \rangle \le \sigma$, противоречит $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Итак, мы пришли к противоречию, следовательно, наше допущение не верно и $\sigma \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$, что и требовалось доказать.

2.2.2.2. Покажем, что $\boldsymbol{ptr}(\boldsymbol{D}(\boldsymbol{\Psi})) \setminus \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi})) \supseteq \boldsymbol{\Psi} \setminus \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$.

Пусть трасса $\sigma \in \boldsymbol{\Psi} \setminus \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Тогда трасса σ является финальным продолжением в $\boldsymbol{\Psi}$ некоторой трассы $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$ и $\sigma \ne \mu$. Тогда по п.2.2.2.1 $\mu \in \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$. Нам надо показать, что σ является финальным продолжением μ в $\boldsymbol{D}(\boldsymbol{\Psi})$ и $\sigma \notin \boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$.

Поскольку $\sigma \ne \mu$, по определению финального продолжения и отсутствию в p-модели **Q**-отказов возможны три варианта: 1) $\sigma = \mu \cdot \langle \Delta \rangle \in \boldsymbol{\Psi}$, 2) $\sigma = \mu \cdot \langle z, \gamma \rangle \in \boldsymbol{\Psi}$, 3) $\sigma = \mu \cdot \langle z \rangle \in \boldsymbol{\Psi}$ и $\mu \cdot \langle z, \gamma \rangle \in \boldsymbol{\Psi}$.

Тогда по определению ***d***-замыкания 1) $\sigma=\mu\cdot\langle\Delta\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$, 2) $\sigma=\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$, 3) $\sigma=\mu\cdot\langle z\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$ и $\mu\cdot\langle z,\gamma\rangle\in\boldsymbol{D}(\boldsymbol{\Psi})$.

Тогда, поскольку $\mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$, трасса σ является финальным продолжением μ в $\boldsymbol{D}(\boldsymbol{\Psi})$, во всех этих случаях также $\sigma\notin\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{D}(\boldsymbol{\Psi}))$, что и требовалось доказать.

18. Доказательство Теорема 7:

Пусть $\boldsymbol{\Omega}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\Delta,\gamma\})^*$ и $\boldsymbol{\Psi}=\boldsymbol{\Omega}_{\mathbf{R}}$ p-модель.

Обозначим $A=\{\sigma\in\boldsymbol{\Omega}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Omega}\}$.

Поскольку $\boldsymbol{\Psi}=\boldsymbol{\Omega}_{\mathbf{R}}$, имеем $A=\{\sigma\in\boldsymbol{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Psi}\}$.

Условия 4b и 4c совпадают, если $\boldsymbol{SafeIn}(\boldsymbol{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})$. Поскольку $\boldsymbol{\Psi}=\boldsymbol{\Omega}_{\mathbf{R}}$, имеем $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$. Поскольку $\boldsymbol{\Psi}$ p-модель, она предфинальна, поэтому по теореме 3 $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})=A$. Следовательно, для эквивалентности условий 4b и 4c достаточно $\boldsymbol{SafeIn}(\boldsymbol{\Omega})=A$. Очевидно, $\boldsymbol{SafeIn}(\boldsymbol{\Omega})\subseteq A$, поэтому достаточно условия $\boldsymbol{SafeIn}(\boldsymbol{\Omega})\supseteq A$.

1. Пусть выполнены условия: $\forall z\in\mathbf{L}\ \forall Q\in\mathbf{Q}\ \forall\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})$

2b) $\sigma\cdot\langle z\rangle\in\boldsymbol{\Omega}\ \&\ \sigma\cdot\langle z,\gamma\rangle\notin\boldsymbol{\Omega}$

$\Rightarrow\exists P\in\boldsymbol{but}(z)\ P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Omega}\ \boldsymbol{after}\ \sigma\ \&\ (P\in\mathbf{Q}\Rightarrow\sigma\cdot\langle P\rangle\notin\boldsymbol{\Omega})$,

3b) $Q\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Omega}\ \boldsymbol{after}\ \sigma\ \&\ \sigma\cdot\langle Q\rangle\notin\boldsymbol{\Omega}\Rightarrow\exists v\in Q\ \sigma\cdot\langle v\rangle\in\boldsymbol{\Omega}$.

4b) $\forall P\in\mathbf{Q}\ \forall\mu\cdot\langle P\rangle\cdot\lambda\in\boldsymbol{\Omega}\ \mu\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})\ \&\ P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{\Omega}\ \boldsymbol{after}\ \mu\ \&\ \lambda=\epsilon$.

Докажем, что $\boldsymbol{SafeIn}(\boldsymbol{\Omega})\supseteq A$ и выполнены условия 2c и 3c.

1.1. Пусть $\langle\gamma\rangle\in\boldsymbol{\Psi}$.

Тогда по определению p-модели $\boldsymbol{\Psi}=\{\epsilon,\langle\gamma\rangle\}$ и $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})=\varnothing$. Поскольку $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})$, по правилу 4b ни одна трасса не содержит $\mathbf{Q}$-отказов. Следовательно, $\boldsymbol{\Omega}=\boldsymbol{\Omega}_{\mathbf{R}}=\boldsymbol{\Psi}=\{\epsilon,\langle\gamma\rangle\}$, поэтому выполнены условия 2c и 3c. Также $A=\varnothing$, следовательно, $\boldsymbol{SafeIn}(\boldsymbol{\Omega})\supseteq A$.

1.2. Пусть $\langle\gamma\rangle\notin\boldsymbol{\Psi}$.

Будем вести доказательство по индукции. Из $\langle\gamma\rangle\notin\boldsymbol{\Psi}$ следует, что $\langle\gamma\rangle\notin\boldsymbol{\Omega}$, поэтому $\epsilon\in\boldsymbol{SafeIn}(\boldsymbol{\Omega})$. Также $\epsilon\in A$. Пусть для $\sigma\in\boldsymbol{SafeIn}(\boldsymbol{\Omega})\cap A$ утверждение верно и пусть $\sigma\cdot\langle u\rangle\in A$. Тогда $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})$. Поскольку $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Psi})=A$, имеем $\sigma\cdot\langle u\rangle\in\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega})$. Поэтому, если $u\in\mathbf{R}$, то $\sigma\cdot\langle u\rangle\in\boldsymbol{SafeIn}(\boldsymbol{\Omega})$. Если $u\in\mathbf{L}$, то $\sigma\cdot\langle u,\gamma\rangle\notin\boldsymbol{\Omega}$. Поэтому по правилу 2b $u\ \boldsymbol{safe\ in}\ \boldsymbol{\Omega}\ \boldsymbol{after}\ \sigma$. Следовательно, $\sigma\cdot\langle u\rangle\in\boldsymbol{SafeIn}(\boldsymbol{\Omega})$. Итак мы доказали, что $\boldsymbol{SafeIn}(\boldsymbol{\Omega})\supseteq A$. И, поскольку $\boldsymbol{SafeIn}(\boldsymbol{\Omega})\subseteq A$, имеем $\boldsymbol{SafeIn}(\boldsymbol{\Omega})=A$, то есть условие 2c доказано. Также по

правилу 3b, если трасса σ не продолжается **Q**-отказом, то либо этот **Q**-отказ разрушающий после трассы, либо она продолжается действием из этого **Q**-отказа. Если **Q**-отказ разрушающий после трассы, то трасса продолжается либо дивергенцией, либо действием из **Q**-отказа (и далее разрушением). Во всех случаях выполнено свойство 3c **Q**-конвергентности трасс из $\mathbf{\Omega_R}$ в **Ω**.

2. Пусть выполнены условия:

2c) $\langle\gamma\rangle \notin \mathbf{\Omega}$ & $\boldsymbol{SafeIn}(\mathbf{\Omega})=\{\sigma\in\mathbf{\Omega}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\mathbf{\Omega}\} \vee \mathbf{\Omega}=\{\epsilon,\langle\gamma\rangle\}$,

3c) трассы из $\mathbf{\Omega_R}$ **Q**-конвергентны в **Ω**,

4c) $\forall P\in\mathbf{Q}\ \forall\mu\cdot\langle P\rangle\cdot\lambda\in\mathbf{\Omega}\ \mu\in\boldsymbol{SafeIn}(\mathbf{\Omega})$ & $P\ \boldsymbol{safe}_{\gamma\Delta}\ \mathbf{\Omega}\ \boldsymbol{after}\ \mu$ & $\lambda=\epsilon$.

Докажем, что $\boldsymbol{SafeIn}(\mathbf{\Omega})\supseteq A$ и выполнены условия 2b и 3b.

2.1. Пусть $\langle\gamma\rangle\in\mathbf{\Psi}$.

Тогда $\mathbf{\Omega}=\{\epsilon,\langle\gamma\rangle\}$, что влечет $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\varnothing$ и, следовательно, выполнены правила 2b и 3b. Также будет $A=\varnothing$, следовательно, $\boldsymbol{SafeIn}(\mathbf{\Omega})\supseteq A$.

2.2. Пусть $\langle\gamma\rangle\notin\mathbf{\Psi}$.

Тогда $\boldsymbol{SafeIn}(\mathbf{\Omega})=A$, следовательно, $\boldsymbol{SafeIn}(\mathbf{\Omega})\supseteq A$. Также $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=A$. Пусть выполнено условие левой части импликации в правиле 2b: $z\in\mathbf{L}$, $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})$, $\sigma\cdot\langle z\rangle\in\mathbf{\Omega}$, $\sigma\cdot\langle z,\gamma\rangle\notin\mathbf{\Omega}$. Тогда $\sigma\cdot\langle z\rangle\in A=\boldsymbol{SafeIn}(\mathbf{\Omega})$. Следовательно, выполнено условие в правой части импликации правила 2b, то есть правило 2b доказано. Пусть выполнено условие левой части импликации в правиле 3b: $\sigma\in\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})$, $Q\ \boldsymbol{safe}_{\gamma\Delta}\ \mathbf{\Omega}\ \boldsymbol{after}\ \sigma$, $\sigma\cdot\langle Q\rangle\notin\mathbf{\Omega}$. Тогда по **Q**-конвергентности выполнено условие в правой части импликации правила 3b, то есть правило 3b доказано.

19. Доказательство Теорема 8:

1. Пусть $\langle\gamma\rangle\in\mathbf{\Psi}$. Тогда $\mathbf{\Omega}=\mathbf{\Psi}=\{\epsilon,\langle\gamma\rangle\}$. Доказательство утверждения в этом случае тривиально.
2. Пусть $\langle\gamma\rangle\notin\mathbf{\Psi}$.

2.1. Сначала докажем часть утверждения, связанную с множествами безопасных (по разным отношениям) трасс.

По определению отношения ***safe by*** имеет место: $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\boldsymbol{SafeBy}(\mathbf{\Omega})$ и $\boldsymbol{SafeBy}(\mathbf{\Psi})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})$. Поскольку трассы из р-модели **Ψ** не содержат **Q**-отказов, имеем $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=\boldsymbol{SafeIn}(\mathbf{\Psi})$.

По теореме 3 $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=\{\sigma\in\mathbf{\Psi}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\mathbf{\Psi}\}$, а по правилу 2c) $\boldsymbol{SafeIn}(\mathbf{\Omega})=\{\sigma\in\mathbf{\Omega}\cap(\mathbf{L}\cup\mathbf{R})^*|\sigma\cdot\langle\gamma\rangle\notin\mathbf{\Omega}\}$. Следовательно, $\boldsymbol{SafeIn}(\mathbf{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})$.

Поскольку $\mathbf{\Psi}=\mathbf{\Omega_R}$, имеем $\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})$. Из этих равенств следует доказываемое утверждение:

$\boldsymbol{SafeIn}(\mathbf{\Omega})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Omega})=\boldsymbol{SafeBy}(\mathbf{\Omega})=\boldsymbol{SafeBy}(\mathbf{\Psi})=\boldsymbol{Safe}_{\gamma\Delta}(\mathbf{\Psi})=\boldsymbol{SafeIn}(\mathbf{\Psi})$.

2.2. Теперь докажем выполнение свойств f-модели для $\boldsymbol{\Omega}=\boldsymbol{ftr}(\boldsymbol{\Psi},\boldsymbol{safe\ by})$.

2.2.1. Непустота. Поскольку $\boldsymbol{\Psi}$ р-модель, $\boldsymbol{\Psi}\neq\varnothing$. Следовательно, поскольку $\boldsymbol{\Psi}\subseteq\boldsymbol{\Omega}$, $\boldsymbol{\Omega}\neq\varnothing$.

2.2.2. Префикс-замкнутость. Поскольку $\boldsymbol{\Psi}$ р-модель, она префикс-замкнута. По теореме 7 и правилу 4c) трассы из $\boldsymbol{\Omega}\backslash\boldsymbol{\Psi}$ – это $\mathbf{Q}$-финальные продолжения трасс из $\boldsymbol{\Psi}$. Поэтому префикс-замкнутость сохраняется.

2.2.3. Допустимость. Поскольку $\boldsymbol{\Psi}$ р-модель, все ее трассы допустимы. По теореме 7 и правилу 4c) трассы из $\boldsymbol{\Omega}\backslash\boldsymbol{\Psi}$ – это $\mathbf{Q}$-финальные продолжения трасс из $\boldsymbol{\Psi}$, то есть трассы, заканчивающиеся $\mathbf{Q}$-отказами. Поэтому допустимость сохраняется.

2.2.4. $\mathbf{Q}$-допустимость. Это следует из теоремы 7 и свойства 4c).

2.2.5. Согласованность. Поскольку $\boldsymbol{\Psi}$ р-модель, все ее трассы согласованы. По теореме 7 и правилу 4c) трассы из $\boldsymbol{\Omega}\backslash\boldsymbol{\Psi}$ – $\mathbf{Q}$-финальные продолжения трасс из $\boldsymbol{\Psi}$. Поэтому согласованность сохраняется.

2.2.6. $\mathbf{R}\cup\mathbf{Q}$-конвергентность трасс из $\boldsymbol{\Omega}_\mathbf{R}$ в $\boldsymbol{\Omega}$. Поскольку $\boldsymbol{\Psi}$ р-модель, все трассы из $\boldsymbol{\Omega}_\mathbf{R}$ $\mathbf{R}$-конвергентны $\boldsymbol{\Omega}_\mathbf{R}$, следовательно, они $\mathbf{R}$-конвергентны в $\boldsymbol{\Omega}\supseteq\boldsymbol{\Omega}_\mathbf{R}$. По теореме 7 и правилу 3c) трассы из $\boldsymbol{\Omega}_\mathbf{R}$ $\mathbf{Q}$-конвергентны в $\boldsymbol{\Omega}$.

2.2.7. финально-замкнутость $\boldsymbol{\Omega}_\mathbf{R}$. Поскольку $\boldsymbol{\Psi}$ р-модель, она финально-замкнута. Поскольку $\boldsymbol{\Psi}=\boldsymbol{\Omega}_\mathbf{R}$, имеем финально-замкнутость $\boldsymbol{\Omega}_\mathbf{R}$.

2.2.8. $\mathbf{R}$-полнота $\boldsymbol{\Omega}_\mathbf{R}$. Поскольку $\boldsymbol{\Psi}$ р-модель, она $\mathbf{R}$-полная. Поскольку $\boldsymbol{\Psi}=\boldsymbol{\Omega}_\mathbf{R}$, имеем $\mathbf{R}$-полноту $\boldsymbol{\Omega}_\mathbf{R}$.

2.2.9. финальность. Это следует из теоремы 7 и свойства 2c).

20. Доказательство Теорема 9:

1. Пусть $\langle\gamma\rangle\in\boldsymbol{\Omega}$. Тогда $\boldsymbol{\Omega}=\boldsymbol{\Omega}_\mathbf{R}=\{\epsilon,\langle\gamma\rangle\}$. Доказательство утверждения в этом случае тривиально.
2. Пусть $\langle\gamma\rangle\notin\boldsymbol{\Omega}$.

2.1. Сначала покажем, что $\boldsymbol{\Omega}_\mathbf{R}$ р-модель.

2.1.1. Алфавит. По определению $\boldsymbol{\Omega}_\mathbf{R}\subseteq(\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\})^*$.

2.1.2. Непустота. Так как $\boldsymbol{\Omega}$ не пусто и префикс-замкнуто, $\epsilon\in\boldsymbol{\Omega}$. Следовательно, поскольку $\langle\gamma\rangle\notin\boldsymbol{\Omega}$, $\epsilon\in\boldsymbol{\Omega}_\mathbf{R}$, то есть $\boldsymbol{\Omega}_\mathbf{R}$ не пусто.

2.1.3. Префикс-замкнутость. Поскольку $\boldsymbol{\Omega}$ f-модель, она префикс-замкнута. Также префикс-замкнуто множество $(\mathbf{L}\cup\mathbf{R}\cup\{\Delta,\gamma\})^*$. Отсюда $\boldsymbol{\Omega}_\mathbf{R}$ префикс-замкнуто как пересечение префикс-замкнутых множеств.

2.1.4. Допустимость. Поскольку $\boldsymbol{\Omega}$ f-модель, все ее трассы допустимы. Поскольку $\boldsymbol{\Omega}_\mathbf{R}\subseteq\boldsymbol{\Omega}$, допустимость сохраняется.

2.1.5. Согласованность. Поскольку $\boldsymbol{\Omega}$ f-модель, все ее трассы согласованы. Поскольку $\boldsymbol{\Omega}_\mathbf{R}\subseteq\boldsymbol{\Omega}$, согласованность сохраняется.

2.1.6. $\mathbf{R}$-конвергентность $\boldsymbol{\Omega}_\mathbf{R}$. Поскольку $\boldsymbol{\Omega}$ f-модель, все трассы из $\boldsymbol{\Omega}_\mathbf{R}$ $\mathbf{R}$-конвергентны $\boldsymbol{\Omega}$, следовательно, они $\mathbf{R}$-конвергентны в $\boldsymbol{\Omega}_\mathbf{R}$.

2.1.7. финально-замкнутость $\underline{\boldsymbol{\Omega}_{\mathbf{R}}}$. Поскольку $\boldsymbol{\Omega}$ f-модель, $\boldsymbol{\Omega}_{\mathbf{R}}$ финально-замкнута.

2.1.8. $\mathbf{R}$-полнота $\underline{\boldsymbol{\Omega}_{\mathbf{R}}}$. Поскольку $\boldsymbol{\Omega}$ f-модель, $\boldsymbol{\Omega}_{\mathbf{R}}$ $\mathbf{R}$-полная.

2.1.9. предфинальность. По определению $\boldsymbol{SafeIn}(\boldsymbol{\Omega}_{\mathbf{R}})\subseteq\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}})$. Также очевидно $\boldsymbol{SafeIn}(\boldsymbol{\Omega}_{\mathbf{R}})=\boldsymbol{SafeIn}(\boldsymbol{\Omega})$. Поскольку $\boldsymbol{\Omega}$ f-модель, $\boldsymbol{\Omega}$ финально, что означает выполнение свойства 2c для случая $\langle\gamma\rangle\notin\boldsymbol{\Omega}$:

$\boldsymbol{SafeIn}(\boldsymbol{\Omega})=\{\sigma\in\boldsymbol{\Omega}\cap(\mathbf{L}\cup\mathbf{R})^{*}|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Omega}\}$. По определению

$\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}})\subseteq\{\sigma\in\boldsymbol{\Omega}_{\mathbf{R}}\cap(\mathbf{L}\cup\mathbf{R})^{*}|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Omega}_{\mathbf{R}}\}$

$=\{\sigma\in\boldsymbol{\Omega}\cap(\mathbf{L}\cup\mathbf{R})^{*}|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Omega}\}$.

Следовательно, $\boldsymbol{Safe}_{\gamma\Delta}(\boldsymbol{\Omega}_{\mathbf{R}})=\{\sigma\in\boldsymbol{\Omega}_{\mathbf{R}}\cap(\mathbf{L}\cup\mathbf{R})^{*}|\sigma\cdot\langle\gamma\rangle\notin\boldsymbol{\Omega}_{\mathbf{R}}\}$. Поэтому, учитывая, что по доказанному все трассы из $\boldsymbol{\Omega}_{\mathbf{R}}$ допустимы и согласованы, а $\langle\gamma\rangle\notin\boldsymbol{\Omega}$, по теореме 3 $\boldsymbol{\Omega}_{\mathbf{R}}$ предфинально.

2.2. Докажем, что отношение ***safe in*** на $\boldsymbol{\Omega}$, взятое для трасс из $\boldsymbol{\Omega}_{\mathbf{R}}$, удовлетворяет правилам отношения ***safe by***. Поскольку $\boldsymbol{\Omega}$ f-модель, она $\mathbf{Q}$-допустима и финальна, а все трассы из $\boldsymbol{\Omega}_{\mathbf{R}}$ $\mathbf{Q}$-конвергентны в $\boldsymbol{\Omega}$. Поэтому по теореме 7 выполнены условия 2b), 3b), 4b) . Условия 2b), 3b) эквивалентны условиям на ***safe by***, если $\mathbf{Q}$-кнопка безопасна по ***safe by*** после неразрушающей трассы тогда и только тогда, когда она неразрушающая и нет $\mathbf{Q}$-отказа после трассы. Такое отношение ***safe by*** совпадает с отношением ***safe in*** и $\boldsymbol{\Omega}=\boldsymbol{ftr}(\boldsymbol{\Omega}_{\mathbf{R}},\boldsymbol{safe\ in})$.

Список литературы

[1] Bourdonov I., Kossatchev A., Kuliamin V. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proc. of MBT 2006, Vienna, Austria, March 2006.

[2] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, No. 5.

[3] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Безопасность, верификация и теория конформности. Материалы Второй международной научной конференции по проблемам безопасности и противодействия терроризму, Москва, МНЦМО, 2007.

[4] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Теория соответствия для систем с блокировками и разрушением. «Наука», 2008.

[5] Игорь Бурдонов. Теория конформности (функциональное тестирование программных систем на основе формальных моделей). LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9, 428 стр. (содержание книги доступно по адресу: http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf)

[6] Бурдонов И.Б., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. Труды Института системного программирования РАН, N 14.1, 2008.

[7] Бурдонов И.Б., Косачев А.С. Эквивалентные семантики взаимодействия. Труды Института системного программирования РАН, N 14.1, 2008.

[8] Бурдонов И.Б., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. «Программирование», 2009, No. 4.
[9] Бурдонов И.Б., Косачев А.С. Аналитическая верификация конформности. Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность: Труды Всероссийской суперкомпьютерной конференции (21-26 сентября 2009 г., г. Новороссийск). - М.: Изд-во МГУ, 2009.
[10] Бурдонов И.Б., Косачев А.С. Тестирование конформности на основе соответствия состояний. Труды Института системного программирования РАН, N 18, 2010.
[11] Бурдонов И.Б., Косачев А.С. Симуляция систем с отказами и разрушением. 5-ый Международный симпозиум по компьютерным наукам в России. Семинар «Семантика, спецификация и верификация программ: теория и приложения».Казань 2010.
[12] Бурдонов И.Б., Косачев А.С. Тестирование безопасной симуляции. 5-ый Международный симпозиум по компьютерным наукам в России. Семинар «Семантика, спецификация и верификация программ: теория и приложения».Казань 2010.
[13] Бурдонов И.Б., Косачев А.С. Семантики взаимодействия с отказами, дивергенцией и разрушением. Часть 1. Гипотеза о безопасности и безопасная конформность. «Вестник Томского государственного университета. Управление, вычислительная техника и информатика», №4, 2010.
[14] I.Burdonov, A.Kosachev. Formal conformance verification. Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.
[15] Бурдонов И.Б., Косачев А.С. Семантики взаимодействия с отказами, дивергенцией и разрушением. «Программирование», 2010, No. 5.
[16] Бурдонов И.Б., Косачев А.С. Пополнение спецификации для ***ioco***. «Программирование», 2011, No. 1.
[17] Бурдонов И.Б., Косачев А.С. Удаление из спецификации неконформных трасс. Препринт № 23, ИСП РАН, 2011.
[18] Bernot G. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, TAPSOFT'91, Volume 2, pp. 99-119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.
[19] Edmonds J. Johnson E.L. Matching. Euler Tours, and the Chinese Postman. Math. Programming 5, 88-124 (1973).
[20] van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.
[21] van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.
[22] Lee D., Yannakakis M. Principles and Methods of Testing Finite State Machines – A Survey. Proceedings of the IEEE 84, No. 8, 1090–1123, 1996.
[23] Milner R. Modal characterization of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.
[24] Tretmans J. Conformance testing with labelled transition systems: implementation relations and test generation. Computer Networks and ISDN Systems, v.29 n.1, p.49-79, Dec. 1996.
[25] Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.

The final models of specification

Igor Bourdonov <igor@ispras.ru>, Alexander Kossatchev kos@ispras.ru
ISP RAS, Moscow, Russia

Abstract. The paper describes the research in formal methods of conformance testing of the target system against requirements given in specifications. Such testing is based on interaction semantics defining test stimuli and observations of actions and refusals (absence of actions). Unobservable actions and refusals are also possible. Destruction is introduced as a forbidden action that should be avoided during interaction. A notion of safe testing is also introduced, when no unobservable refusals and destruction occur and no test stimuli applied in divergence. On this basis, the implementation hypothesis of safety and safe conformance are defined, as well as the generation of complete test suite from specification.

The most common model of specification is LTS (Labelled Transition System). However, for the described interaction semantics, only traces (sequences of observations) are important, not the LTS states. Therefore, the most natural is the trace model defined as a set of LTS traces.

The goal of this paper is to define the subset of specification traces sufficient for generation of the complete test suite. We called such subset the final trace model of specification. On the other hand, LTS model is convenient as a way of finite representation of regular trace sets. To represent the final trace model, the paper proposes a variation of LTS called final RTS (Refusal Transition System). The transitions on observable refusals are defined explicitly. Such model is very convenient for test generation: 1) it is deterministic, 2) trace of observations is safe iff it ends in non-terminal state with no destruction, 3) test stimulus is safe after the trace iff it is safe in the final state of the trace.

The paper proposes algorithms for transformation of LTS model into final RTS model. Sufficient conditions for creation of the final RTS in finite time are also defined.

Keywords: interaction semantics, refusals, destruction, divergence, conformance, safe testing, traces, LTS, test generation.

References

[1]. Bourdonov I., Kossatchev A., Kuliamin V. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proc. of MBT 2006, Vienna, Austria, March 2006.

[2]. Bourdonov I., Kossatchev A., Kuliamin V. Formalization of Test Experiments. Programming and Computer Software, Vol. 33, No. 5, 2007, pp. 239-260

[3]. Bourdonov I., Kossatchev A., Kuliamin V. Bezopasnost', verifikatsiya i teoriya konformnosti [Safety, Verification and Conformance Theory]. Materialy Vtoroj mezhdunarodnoj nauchnoj konferentsii po problemam bezopasnosti i protivodejstviya terrorizmu [The proceeding of the Second international conference on the problems of safety and counteraction against terrorizm], Moscow, MNCMO, 2007. pp. 135-158. (in Russian).

[4]. Bourdonov I., Kossatchev A., Kuliamin V. Teoriya sootvetstviya dlya system s blokirovkami I razrusheniem [Conformance theory of the systems with Refused Inputs and Forbidden Actions]. Moscow, «Nauka», 2008, 412 p. (in Russian)

[5]. Bourdonov I. Teoriya konformnosti (funkcional'noe testirovanie prorammny'kh system na osnove formal'ny'kh modelej [Conformance theory (functional testing on formal

model base)]. LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9, 428 pp.
http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf) (in Russian)
[6]. Bourdonov I., Kossatchev A. Sistemy s prioritetami: konformnost', testirovanie, kompozitsiya [Systems with priority: conformance, testing, composition]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 14.1, 2008, pp.23-54. (in Russian)
[7]. Bourdonov I., Kossatchev A. Ehkvivalentnye semantiki vzaimodejstviya [Equivalent interaction semantics]. Trudy ISP RAN [The proceeding of ISP RAS], v. 14.1, 2008, pp.55-72. (in Russian)
[8]. Bourdonov I., Kossatchev A. Systems with Priorities: Conformance, Testing, and Composition. Programming and Computer Software, Vol. 35, No. 4, 2009, pp.198-211.
[9]. Bourdonov I., Kossatchev A. Analiticheskaya verifikatsiya konformnosti [Analitical conformance verification]. Nauchnyj servis v seti Internet: masshtabiruemost', parallel'nost', ehffektivnost': Trudy Vserossijskoj superkomp'yuternoj konferentsii (21-26 sentyabrya 2009 g., g. Novorossijsk) [Scientific service of Internet: The proceeding of Russian Supercomputer conference (21-26 sept. 2009, Novorossijsk)] – Moscow, MSU publ., 2009. (in Russian)
[10]. Bourdonov I., Kossatchev A. Testirovanie konformnosti na osnove sootvetstviya sostoyanij [Conformance testing based on a state relation]. Trudy ISP RAN [The proceeding of ISP RAS, Vol. 18, 2010, pp. 183-320. (in Russian)
[11]. Bourdonov I., Kossatchev A. Simulyatsiya sistem s otkazami i razrusheniem [Simulation of the systems with Refused Inputs and Forbidden Actions]. 5-yj Mezhdunarodnyj simpozium po komp'yuternym naukam v Rossii. Seminar «Semantika, spetsifikatsiya i verifikatsiya programm: teoriya i prilozheniya» [5-th International symposium on computer science in Russia. Workshop “Semanticsand program specification and verification: Theory and applications”]. Kazan' 2010. (in Russian)
[12]. Bourdonov I., Kossatchev A. Testirovanie bezopasnoj simulyatsii [Safe simulation testing]. 5-yj Mezhdunarodnyj simpozium po komp'yuternym naukam v Rossii. Seminar «Semantika, spetsifikatsiya i verifikatsiya programm: teoriya i prilozheniya» [5-th International symposium on computer science in Russia. Workshop “Semanticsand program specification and verification: Theory and applications”]. Kazan' 2010. (in Russian)
[13]. Bourdonov I., Kossatchev A. Semantiki vzaimodejstviya s otkazami, divergentsiej i razrusheniem. CHast' 1. Gipoteza o bezopasnosti i bezopasnaya konformnost'. [Semantics of Interaction with Refused Inputs, Divergence and Forbidden Actions. Part 1. Hypothesis of safety and safe conformance]. «Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika» [Tomsk State University. Journal of Control and Computer Science]., №4, 2010. (in Russian)
[14]. I.Burdonov, A.Kosachev. Formal conformance verification. Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.
[15]. Bourdonov I., Kossatchev A. Interaction Semantics with Refusals, Divergence, and Destruction. Programming and Computer Software, Vol. 36, No. 5, 2010, pp. 247-263.
[16]. Bourdonov I., Kossatchev A. Specification Completion for IOCO. Programming and Computer Software, Vol. 37, No. 1, 2011, pp. 1-14.
[17]. Bourdonov I., Kossatchev A. Udalenie iz spetsifikatsii nekonformnykh trass [Nonconforming traces elimination from specification]. Preprint № 23, ISP RAN [Preprints of the Institute for System Programming of RAS, Preprint 23], 2011. (in Russian)

[18]. Bernot G. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, TAPSOFT'91, Volume 2, pp. 99-119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.
[19]. Edmonds J. Johnson E.L. Matching. Euler Tours, and the Chinese Postman. Math. Programming 5, 88-124 (1973).
[20]. van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.
[21]. van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.
[22]. Lee D., Yannakakis M. Principles and Methods of Testing Finite State Machines – A Survey. Proceedings of the IEEE 84, No. 8, 1090–1123, 1996.
[23]. Milner R. Modal characterization of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.
[24]. Tretmans J. Conformance testing with labelled transition systems: implementation relations and test generation. Computer Networks and ISDN Systems, v.29 n.1, p.49-79, Dec. 1996.
[25]. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.

Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов

С. П. Вартанов, Д.В. Сидоров[1]
svartanov@ispras.ru, sidorov@ispras.ru

Аннотация. В статье предложена оптимизация алгоритма проверки выполнимости булевых формул DPLL (Davis — Putnam — Logemann — Loveland) с помощью кэширования промежуточных результатов при решении задачи нахождения входных данных для неинтерактивных программ. Дополнительная информация для оптимизации алгоритма запоминается на одном из предыдущих запусков алгоритма. Возможность подобного рода модификации алгоритма основана на особенности последовательностей проверяемых формул.
В результате модифицированный алгоритм показал ускорение по сравнению с использованием алгоритма DPLL без оптимизаций. Для проверки использовался тестовый солвер, последовательности формул генерировались инструментом Avalanche.
Ключевые слова: проверка выполнимости булевых формул, алгоритм DPLL, оптимизация

1. Введение

1.1. Задача проверки выполнимости булевых формул

Задача проверки выполнимости булевых формул (SAT, или задача ВЫП) формулируется следующим образом:

> Задано множество булевых переменных и определено некоторое выражение над ними. Необходимо определить, существует ли такой набор значений этих переменных, при которых выражение выполнимо, т. е. принимает значение «истина».

[1] Работа проводится в рамках реализации ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы», ГК № 07.514.11.4040 от 29.09.2011. Работа проводится при поддержке программы «Университетский кластер», http://www.unicluster.ru

Согласно теореме Кука, доказанной им в 1971-м году, задача SAT NP-полна. Она также остаётся NP-полной, если выражение имеет вид конъюнктивной нормальной формы от своих переменных. [4]

Тема сведения задач из различных областей прикладной математики и информатики к задаче SAT хорошо развивается. Одна из причин этого — возможность сведения к ней задач из класса NP за полиномиальное время и тем самым возможность решать достаточно трудные задачи, возникающие в самых различных областях, единым алгоритмом решения задачи SAT.

1.2. Алгоритм DPLL

Одним из наиболее эффективных алгоритмов проверки выполнимости булевых формул является алгортим DPLL (Davis — Putnam — Logemann — Loveland), основанный на бэк-трекинге. Суть алгоритма заключается в том, что для исходной булевой формулы запускается рекуррентная функция, производящая над формулой следующие действия: [3]

Распространение «единичных» конъюнктов (unit propogation). Если в КНФ присутствует конъюнкт, состоящий из одного литерала (отрицания литерала), он удаляется из КНФ, а все вхождения литерала заменяются на true (false).

Удаление «чистых» литералов (pure literal elimination). Производится поиск в формуле «чистых» литералов — переменные в которых встречаются только без отрицаний или только с ними — и вместо них подставляются значения true или false соответственно. Несмотря на то, что этот шаг является частью оригинального алгоритма, на практике, обычно, его опускают, поскольку накладные расходы зачастую превосходят эффект от его применения.

Выбор очередного литерала и запуск рекуррентной функции от формулы, в которой выбранная литера заменена на true и запуск функции, в которой она заменена на false.

На вход алгоритм принимает булеву формулу в конъюнктивной нормальной форме.

Алгоритм прекращает работу, либо когда на одной из веток значение формулы стало равным true, либо когда полный обход дерева показал, что присвоить значения переменным таким образом, чтобы значение формулы стало равным истине, невозможно.

1.3. Постановка задачи

Существует проект Avalanche, в котором для поиска ошибок и уязвимостей в программах происходит подбор входных данных. Для неинтерактивных программ для этого необходимо проверить возможность истинности всех условных переходов, встретившихся в трассе программы. Построенные формулы могут быть конвертированы в конъюнктивные нормальные формы. В этом случае задача сводится к проверке выполнимости их конъюнкции, т. е. к проблеме SAT.

Последовательность ограничений, подаваемых на вход солверу STP, а также генерируемая им последовательность булевых формул, имеет некоторые особенности. В частности, две последовательно идущие формулы зачастую имеют одинаковые части.

Необходимо разработать алгоритм проверки выполнимости булевых формул, использующий информацию, полученную при проверке выполнимости одной формулы из последовательности для ускорения проверки выполнимости другой формулы.

1.4. Требования к решению

Алгоритм должен быть основан на использующимся в решателе MINISAT алгоритме DPLL. Также он должен опираться на особенности последовательностей формул, для которых решается задача проверки выполнимости.

2. Исследование и построение решения задачи

2.1. Проект Avalanche

Avalanche — инструмент обнаружения программных дефектов при помощи динамического анализа. Инструмент использует возможности динамической инструментации программы, представляемые Valgrind, для сбора и анализа трассы выполнения программы. [1]

На одном из этапов работы программы, инструменту необходимо проверить набор булевых ограничений. Для решения задачи выполнимости булевых ограничений в проекте используется солвер STP. Схема работы солвера представлена на рис. 1. [6]

Решение этой задачи занимает значительную часть работы. Так, на половине проектов, которые были проанализированы инструментом Avalahche, работа STP заняла больше 70 % всего времени, а на таких проектах как «sndfile-mix-to mono» и «cjpeg», солвер работал более 99 % от времени работы инструмента[1]. Следовательно, существует потребность в оптимизации алгоритма решения этой задачи.

STP выполняет некие преобразованя входных формул, в том числе преобразование выражений в КНФ, кототрые он подаёт на вход решателю MINISAT.

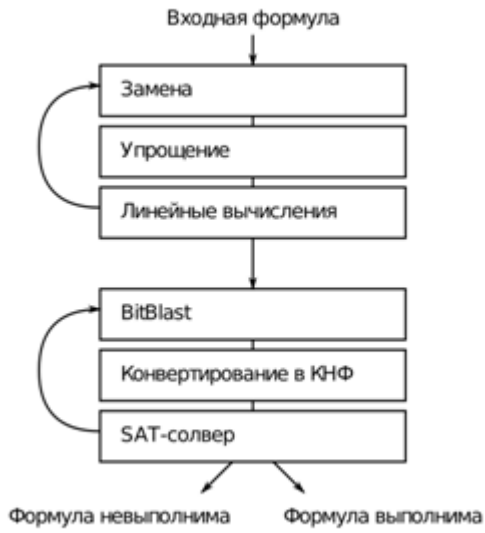


Рис. 1: *Работа солвера STP*

3. Модификация алгоритма DPLL

В решателе MINISAT используется рассмотренный ранее алгоритм DPLL. [5] Зачастую требуется проверить набор ограничений следующего вида:

$$\overline{A_1(x_1)};$$
$$A_1(x_1) \wedge \overline{A_2(x_2)};$$
$$\ldots$$
$$A_1(x_1) \wedge A_2(x_2) \wedge \ldots \wedge \overline{A_n(x_n)}.$$

Эта особенность формул основана на виде трасс, для которых проверяются ограничения. Рассмотрим простой пример (Рис. 2).

На схеме представлен граф потока управления некоторой программы. В ходе динамического анализа существует необходимость покрыть как можно больше операторов программы. Для этого необходимо выяснить, какие ограничения (например, значения переменных) должны выполняться, чтобы выполнения программы шло по одному из возможных путей.

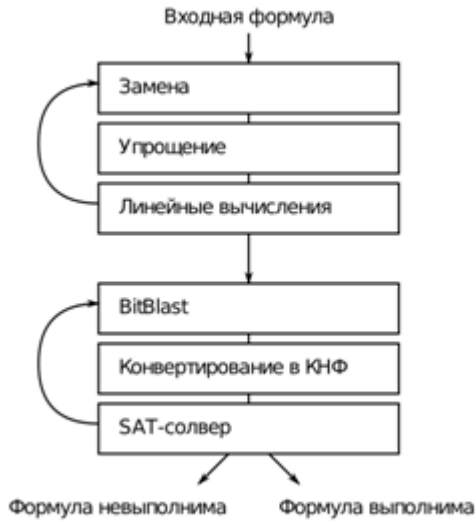


Рис. 2: *Пример графа потока управления*

На представленном примере возможны три выполнения программы. Им соответствуют следующие ограничения:

$$\overline{A_1(x_1)};$$
$$A_1(x_1) \wedge \overline{A_2(x_2)};$$
$$A_1(x_1) \wedge A_2(x_2) \wedge \overline{A_3(x_3)}$$

Продолжим рассмотрение общей формулы, и поскольку выполнимость формулы и выполнимость её отрицания — вещи, связанные слабо, преобразуем систему к более общему виду:

$$B_1;$$
$$A_1 \wedge B_2;$$
$$\ldots$$
$$A_1 \wedge A_2 \wedge \ldots \wedge B_n$$

Заметим, что формулы $m-1$ и m, соответственно, имеют вид:

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \wedge B_{m+1}$$
$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \wedge A_{m+1} \wedge B_{m+2}$$

При достаточно больших m одинаковая часть обеих формул значительно превышает части, в которых они различаются.

Попробуем использовать информацию, полученную при проверке выполнимости формулы A для проверки выполнимости $B = A \wedge C$

Преобразуем алгоритм следующим образом. Во время проверки выполнимости формулы A, запомним место в дереве, на котором мы остановились и состояние в нём, а также все состояния всех точек в которых остались непросмотренные ветки и в которые мы можем вернуться в случае неудачи на текущей ветке.

3.1. Пример работы алгоритмов

Рассмотрим предложенный алгоритм на двух следующих конкретных формулах:

$$A = (x_0 \vee x_1) \wedge (x_0 \vee \overline{x_1}) \wedge (\overline{x_0} \vee x_2) \wedge (\overline{x_3} \vee x_4),$$

$$B = (x_0 \vee x_1) \wedge (x_0 \vee \overline{x_1}) \wedge (\overline{x_0} \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (x_3 \vee x_5) \wedge (\overline{x_5}).$$

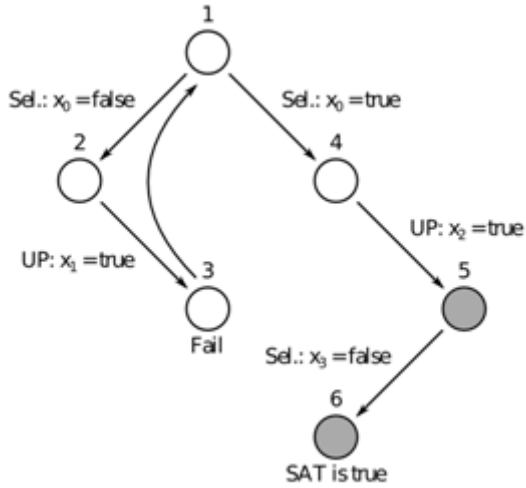


Рис. 3: *Пример работы алгоритма DPLL. Метками Sel. и UP обозначены присваивания, происходящие на шагах выбора переменной и распространения «единичных» конъюнктов, соответственно.*

Формула B есть конъюнкция формулы A и $C = (x_3 \vee x_5) \wedge (\overline{x_5})$.

Рассмотрим, сначала, как работает алгоритм DPLL без pure literal assign для формулы A.

Сплошными линиями на схеме показана работа алгоритма для формулы A. В вершине 3 формула принимает вид $false \wedge \left(\overline{x_3} \vee x_4\right)$ и тождественно равна false. После этого вычисление возвращается к последней нерассмотренной ветке, а именно, выбору true в качестве значения переменной x_0. В вершине 6 формула тождественно равна true, следовательно, исходная формула выполнима (значения переменных можно проследить, пройдя из этой вершины к корню). При этом осталась нерассмотренной ветка, выходящая из вершины 5, в которой переменной x_3 присваивается значение true.

Рассмотрим теперь работу алгоритма при проверке выполнимости формулы . Вычисление начнём не с корня дерева, а с точки, на которой мы закончили вычисление при проверке выполнимости формулы A. При этом мы сохранили значения формулы и переменных в вершине 6, а также в вершине 5 — единственной, оставшейся непросмотренной. На схеме вершины, в которых запомнены состояния закрашены.

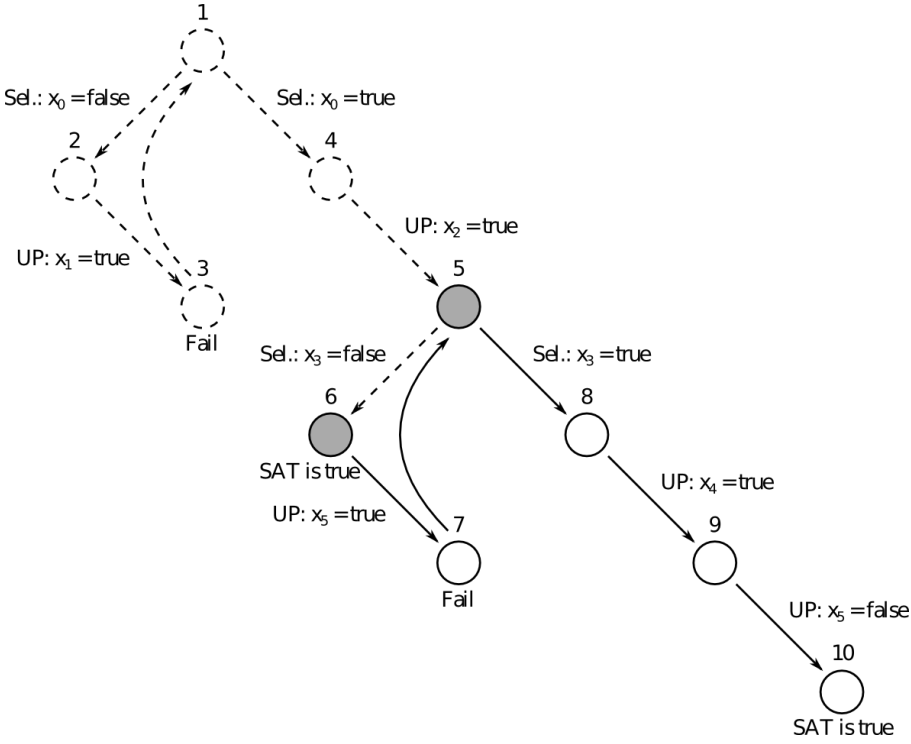


Рис. 4: *Пример работы модифицированного алгоритма DPLL*

Например, состояния в вершинах 5 и 6 следующие:

Таблица 1: Состояния

Состояние	x_0	x_1	x_2	x_3	x_4	x_5	Формула
5	true	—	true	—	—	—	$(x_2) \wedge (\overline{x_3} \vee x_4)$
6	true	—	true	false	—	—	true

Таблица 1: *Состояния*

Дальнейшее вычисление на схеме показано сплошными линиями. Добавим к формуле в вершине 6 конъюнкцию $(x_3 \vee x_5) \wedge (\overline{x_5})$ и присвоим переменным значения из состояния в этой вершине. Получим формулу $(x_5) \wedge (\overline{x_5})$. Последующее продвижение единичных конъюнктов в вершине 7 показывает, что формула равна false. Выполнимость формулы B в этой вершине не доказана. Вернёмся к последней нерассмотренной вершине (вершина 5) и продолжим вычисление с этой точки. В вершине 10 формула тождественно равна true и выполнимость формулы B доказана.

Таким образом, для формул

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \wedge B_{m+1}$$

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \wedge A_{m+1} \wedge B_{m+2}$$

можно сначала проверить выполнимость формулы $A_1 \wedge A_2 \wedge \ldots \wedge A_m$, т.е. общей части, а затем вышеуказанным методом проверить выполнимость формул вцелом.

Поскольку алгоритм выполняет некоторые дополнительные действия, необходимо проверить, оправданы ли они.

4. Тестовый солвер

Для проверки алгоритма на практике, был написан тестовый солвер на языке Java. Он реализует алгоритм DPLL без шага, на котором удаляются «свободные» литералы. Выбор очередного литерала выполняется только после того, как шаг продвижения единичных конъюнктов более неприменим. Такой подход используется в большинстве реализация алгоритма DPLL.

В солвере используется единственная оптимизация алгоритма DPLL — в памяти для каждой переменной хранится число вхождений этой переменной в формулу с отрицаниями и без.

Также солвер реализует описанную модификацию алгоритма. После каждой проверки формулы, если выполнимость её подтверждена, ветка дерева разбора на которой было найдено решение, а также все наборы присваиваний в вершинах, ответвления из которых остались непросмотренными, сохраняются в памяти.

Программа работает с формулами в конъюнктивной нормальной форме, записанными в текстовый файл. Переменные обозначаются символом «x» с индексом, отрицание знаком «!», логическое сложение — знаком «|», логическое умножение — знаком «&». Поскольку формула задаётся только в КНФ, скобки опускаются.

Например, формула

$$(x_0 \vee x_1) \wedge (x_0 \vee \overline{x_1}) \wedge (\overline{x_0} \vee x_2) \wedge (\overline{x_3} \vee x_4),$$

должна быть записана следующим образом: x0|x1&!x1|x2|!x3&x3|x4.

Пример вывода солвера:

```
(x0 | x1) & (x0 | !x1) & (!x0 | x2) & (!x3 | x4)
After UP — (x0 | x1) & (x0 | !x1) & (!x0 | x2) & (!x3 | x4)
Select 0
Assign (Select) x0 = false
(x1) & (!x1) & (!x3 | x4)
Assign (UP) x1 = true
After UP — () & (-x3 | x4)
SAT is false.

Assign (Select) x0 = true
(x2) & (!x3 | x4)
Assign (UP) x2 = true
After UP — (!x3 | x4)
Select 3
Assign (Select) x3 = false
SAT is true: 1210 - 0, -> 2, => 3, <-
```

В последней строке содержится информация о том, что для формулы установлена её выполнимость. Там же содержится вектор присваиваний — 1210, где каждая цифра соответствует значению переменной (true для переменных с номерами 0 и 2, false для переменной с номером 3 и переменная 1 осталась без присваивания). Строка 0, -> 2, => 3, <- — описание ветки, на которой было найдено решение. Все эти результаты сохраняются в памяти.

5. Проверка работы модифицированного алгоритма

5.1. Трассы инструмента Avalanche

Скорость работы предложенной модификации алгоритма была проверена на последовательностях трасс, передаваемых инструментоми Avalanche солверу STP при проверке им различных программ.

Для каждого теста в ходе работы инструмента извлекались две последовательно вычисляющиеся трассы. Сначала выполнимость обеих формул независимо проверялась алгоритмом DPLL. А затем выполнялась проверка первой формулы, в ходе которой запоминались необходимые состояния и на основе этой информации запускался модифицированный алгоритм на второй формуле.

При этом последовательности утверждений были конвертированы в конъюнктивные нормальные формы.

Для каждого алгоритма указано время его выполнения на тестовом солвере в секундах и количество выполненных присваиваний (в ходе шага выбора переменной или unit propogation) во второй формуле. Число конъюнктов также указано для второй формулы.

Тест	Число	DPLL			Модиф. DPLL			Ускорение (раз)	Итоговое ускорение
	конъюн.	1 ф.	2 ф.	пр.	1 ф.	2 ф.	пр.		
1	416	2.87	1.07	641	2.94	0.37	175	2.86	1.19
2	416	1.27	1.39	806	1.33	0.34	260	4	1.58
3	416	1.35	1.80	837	1.43	0.37	292	4.7	1.73
4	480	1.59	1.78	837	1.67	0.19	192	8.9	1.8
5	544	3.01	1.97	837	3.02	0.22	96	8.7	1.53
6	540	14.82	0.29	251	14.89	0.20	1	1.4	1.0007
7	362	8.15	6.8	407	8.16	0.74	71	9.17	1.67
8	1340	77.03	0.72	634	77.06	0.15	1	4.65	1.007
9	665	9.55	0.29	337	9.59	0.14	1	2	1.01
10	527	25.96	22.65	574	26.02	2.09	32	10.81	1.72
11	559	23.37	22.44	606	23.46	2.46	32	9.08	1.76
12	591	26.28	21.45	638	26.28	1.79	32	11.93	1.69

Таблица 2: *Сравнение скорости работы алгоритмов.*

5.2. Случайные КНФ

Также скорость работы алгоритма была проверена на случайным образом сгенерированных конъюнктивных нормальных формах.

В таблице приведены средние результаты запусков следующих формул:

50 конъюнктов с не более чем 50 литералами,

100 конъюнктов с не более чем 50 литералами,

50 конъюнктов с не более чем 100 литералами.

Тест	Число	DPLL			Модиф. DPLL			Ускорение (раз)	Итоговое ускорение
	конъюн.	1 ф.	2 ф.	пр.	1 ф.	2 ф.	пр.		
1	50	0.05	0.16	165	0.1	0.02	73	5.85	1.61
2	100	0.2	0.31	360	0.24	0.07	189	4.48	1.64
3	50	0.15	0.30	173	0.23	0.06	95	4.98	1.56

Таблица 3: *Сравнение скорости работы алгоритмов.*

Следует отметить, что в общем случае может быть выполнен полный обход ветки, которую использует алгоритм, прежде чем будет обнаружена переменная, предположение о значении которой становится неверным в «новой» формуле.

Однако, как видно из результатов тестов, на практике такой случай встречается редко.

Чтобы избежать падения эффективности в этих случаях, можно ограничивать глубину обхода ветки. При этом, если ограничение будет достигнуто, продолжать обход с корня, что эквивалентно запуску немодифицированного DPLL.

6. Заключение

Последовательности были проверены на тестовом солвере сначала алгоритмом DPLL, а затем его модифицированным вариантом.

После сравнения скорости работы алгоритмов, можно сделать вывод о том, что в среднем модифицированный алгоритм даёт ускорение.

Поскольку в среднем алгоритм даёт выигрыш во времени, он может быть применён для указанных выше целей.

В дальнейшие планы входит внедрение алгоритма в решатель MINISAT.

Список литературы

[1] Исаев, И. К. Сидоров Д. В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование [№ 4]. 2010. 16 с.

[2] С. В. Зеленов, С. А. Зеленова. Автоматическое определение выполнимости наборов формул для операций сравнения. Труды ИСП РАН, том 14. 109-118 с. Москва, 2008.

[3] Davis M., Logemann G., Loveland D. A machine program for theorem proving // Communication of the ACM. 1962. P. 394-397.

[4] Новикова Н. М. Основы оптимизации. Москва. 1998. 17-22 с.

[5] Eén N., Sörensson N. MiniSat solver [HTML] (http://minisat.se/)

[6] Katelman M., Soos M. STP Constraint Solver [HTML] (http://sites.google.com/site/stpfastprover/)

Optimization of Boolean satisfiability solver by caching intermediate results

S. Vartanov, D. Sidorov
svartanov@ispras.ru, sidorov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Recently, a number of dynamic analysis tools were developed that perform tainted data flow tracing and use algorithms for solving the CNF SAT problem for input data generation and path alteration. SAT problem is known to be NP-complete and it requires a lot of optimizations to partially solve the exponential growth problem. The problem appeared in Avalanche dynamic analysis project for automatic defect detection. In this paper we propose a modification of Davis—Putnam—Logemann—Loveland (DPLL) algorithm using intermediate result caching.

For path condition solving Avalanche uses STP which uses MiniSAT solver for Boolean formula satisfiability checking. MiniSAT main mechanism is based on variation of DPLL algorithm. We describe special properties of formula sets generated with dynamic analysis tools. Some formulas in generated set frequently have common parts. We introduce a DPLL optimization based on these peculiarity. Formula solving is performed on previously cached solver results and uses backtracking mechanism in case of failure. Described optimization is illustrated on simple examples.

Described mechanism was examined on random generated sets of Boolean formulas using implemented SAT solver prototype. Modified algorithm applications features a noteworthy increase in efficiency in comparison with standard DPLL approach.

Keywords: CNF-SAT problem, algorithm DPLL, optimization

References

[1]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1-16.

[2]. S. V. Zelenov, S. A. Zelenova. Avtomaticheskoe opredelenie vypolnimosti naborov formul dlya operatsij sravneniya [Automatic solving of set of formulas satisfiability problem for comparison operators]. Trudy ISP RAN, tom 14 [The Proceedings of ISP RAS, vol. 14]. 109-118 s. Moskva, 2008.

[3]. Davis M., Logemann G., Loveland D. A machine program for theorem proving. Communication of the ACM. 1962. P. 394-397.

[4]. Novikova N. M. Osnovy optimizatsii [Basics of Optimization]. M.: MGU [MSU], 1998. 17–22 p.

[5]. Eén N., Sörensson N. MiniSat solver [HTML] [1]. (http://minisat.se/)

[6]. Katelman M., Soos M. STP Constraint Solver [HTML] (http://sites.google.com/site/stpfastprover/)

Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux[1]

М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов,
mandrykin@ispras.ru, mutilin@ispras.ru, joker@ispras.ru, khoroshilov@ispras.ru

Аннотация. В работе рассмотрены методы и техники, используемые в современных инструментах статической верификации Си программ. Представлен обзор текущих возможностей инструментов, в том числе по таким направлениям, как поддержка конструкций языка Си, масштабируемость, виды проверяемых свойств и достоверность результатов анализа. Рассмотрены особенности применения инструментов статической верификации для анализа исходного кода драйверов устройств операционных систем и описаны существующие системы верификации драйверов, построенные на основе таких инструментов.

Ключевые слова: верификация, статический анализ, драйвер устройства, операционная система Linux.

1. Введение

Статическая верификация программ заключается в анализе кода программы без её реального выполнения, что в отличие от динамического анализа не требует настройки тестового окружения и предоставляет возможность проанализировать все возможные выполнения программы, даже те, для воспроизведения которых требуется одновременное выполнение сразу нескольких редко встречающихся условий.

В настоящее время наибольшее распространение получили две группы методов статической верификации: методы дедуктивного анализа программ и методы проверки моделей. Методы дедуктивного анализа используются для доказательства соответствия программы своей спецификации, обычно задаваемой в виде пред- и постусловий. На текущем уровне развития инструменты, реализующие эти методы, требуют ручной аннотации анализируемых функций и циклов в программе, что не позволяет их

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 07.514.11.4104)

применять для верификации больших программ. Тем не менее, эти методы являются эффективным средством для доказательства корректности наиболее важных компонентов программных систем.

Методы проверки моделей формируют по коду программы её модель в явном или неявном виде и проводят анализ этой модели на предмет выполнения заданных ограничений. Как правило, проверяемые ограничения не нацелены на полное описание требований к функциональности программы, а представляют собой проверку выполнения частных требований или проверку отсутствия в коде программы определённого класса ошибок.

Методы проверки моделей обычно работают с исходным текстом программы на языке программирования. Перед анализом происходит разбор текста программы во внутреннее представление, в ходе которого для каждой функции в программе строится граф потока управления (ГПУ). Вершины графа соответствуют местам в тексте программы. Рёбра содержат операторы из текста программы, возможно, преобразованные в последовательность инструкций. Инструкции обращаются к локальным и глобальным переменным программы, к значениям в динамической памяти или к окружению системы.

Провести детальный анализ всех возможных выполнений программы за разумное время не представляется возможным, поэтому при статической верификации выполнения программы моделируются лишь с некоторой точностью. От уровня точности зависит количество ложных сообщений об ошибках, количество упущенных ошибок, а также время, необходимое для выполнения анализа.

В зависимости от того, каким характеристикам отдается предпочтение выделяют легковесные и тяжеловесные подходы. В легковесных подходах упор делается на скорости анализа, при этом точность анализа может снижаться, что приводит к увеличению числа упущенных ошибок и ложных срабатываний. Как правило, высокая скорость достигается за счёт активного использования различных эвристик и игнорирования значительной части контекста точки выполнения программы, то есть пути попадания в эту точку. В тяжеловесных подходах предпочтение отдается точности, что при определённых условиях дает возможность доказать даже полное отсутствие искомых видов ошибок в программе. Для тяжеловесных подходов характерен аккуратный анализ контекста точки выполнения программы, а также более детальный анализ взаимодействия программы с окружением.

В настоящей работе рассмотрение ограничивается тяжеловесными подходами, которые демонстрируют значительный прогресс за последние годы. Одним из первых примеров успешного применения таких подходов в индустрии является использование инструмента Slam [1] для верификации драйверов устройств операционной системы (ОС) Microsoft Windows. Драйверы устройств являются подходящей площадкой для применения тяжеловесных инструментов статической верификации по целому ряду причин:

- массовость — драйверы разрабатываются в большом количестве по единым принципам, что позволяет проверять выполнимость одних и тех же свойств сразу у многих программ;
- размер кода — суммарный размер исходного кода драйверов, как правило, не превышает нескольких десятков тысяч строк кода;
- относительная простота кода — в драйверах обычно не используются операции с плавающей точкой, сложные структуры данных и т.п.

Поэтому в дальнейшем инструменты тяжеловесной верификации неоднократно пытались применить для верификации драйверов устройств, но уже ОС Linux, так как в отличии от разработчиков Slam из Microsoft Research, доступ других исследователей к исходному коду драйверов ОС Microsoft Windows ограничен. В рамках настоящего обзора представлено описание этих попыток, интересное в первую очередь с точки зрения применения тяжеловесных подходов на практике.

Основная часть работы построена следующим образом. В разделе 2 рассматриваются методы и техники, используемые в современных инструментах тяжеловесной статической верификации Си программ. Раздел 3 содержит описание характеристик инструментов верификации, важных для их практического применения. В разделе 4 представлен обзор инструментов тяжеловесной статической верификации Си программ. В разделе 5 обсуждаются особенности применения инструментов статической верификации для анализа исходного кода драйверов устройств операционных систем и описываются существующие системы верификации драйверов, построенные на основе таких инструментов. В заключение подводятся итоги работы и делаются выводы о перспективах развития инструментов тяжеловесной статической верификации в контексте их применения для верификации драйверов устройств.

2. Методы и техники статической верификации программ

2.1. CEGAR (Counter Example Guided Abstraction Refinement)

CEGAR – метод статического анализа кода, основанный на абстракции и уточнении на основе контрпримеров [2].

Свойства программы, которые проверяются при помощи CEGAR, формулируются в виде недостижимости заданного множества ошибочных состояний в программе. Идея CEGAR состоит в итеративном уточнении модели программы (абстракции) до тех пор, пока не будет доказана её корректность или не будет найден ошибочный путь, то есть путь, начинающийся в точке входа в программу и ведущий в ошибочное состояние. Для этого CEGAR предлагает строить такую модель программы, в которой сохраняются ошибочные пути, и осуществлять поиск ошибочных путей в этой модели. Если в модели ошибочных путей не найдено, то в программе нет

искомых ошибок. В противном случае, потребуется проверить реализуемость модельного ошибочного пути в программе, так как возможно, что ошибочный путь в модели появился из-за абстрагирования от существенных деталей в коде программы. Если проверка покажет, что путь реализуем, то это означает, что найдена ошибка в программе. Иначе этот путь в качестве контрпримера будет использован для уточнения модели на следующем шаге, так чтобы в уточнённой модели этого пути уже не было.

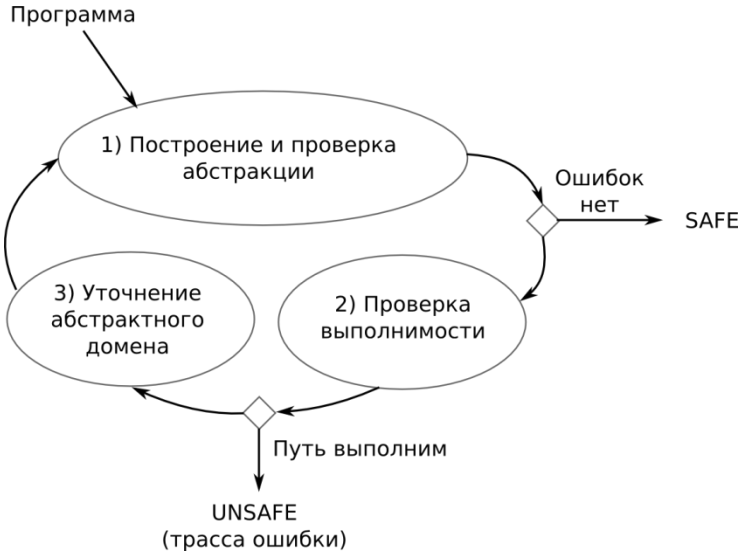


Рис. 1. Подход CEGAR (Counter Example Guided Abstraction Refinement)

В CEGAR выделяют следующие составляющие этапы:

1. Построение и проверка абстракции.
2. Проверка выполнимости.
3. Уточнение абстрактного домена.

На первом этапе строится абстракция программы, включающая в себя все ошибочные пути выполнения в программе. Абстракция строится по начальному абстрактному домену, задающему правила соответствия между абстрактными состояниями и переходами и состояниями и переходами в программе. Построение абстракции — это затратная операция, состоящая в вычислении абстрактных переходов между абстрактными состояниями. Точность абстракции зависит от абстрактного домена, определяющего точность приближения состояний программы абстрактными состояниями, а также от точности моделирования конструкций входной программы при вычислении абстрактных переходов.

Вторая часть первого этапа — проверка абстракции на наличие ошибки. Она может происходить как строго после построения абстракции, так и совместно. В первом случае построенная абстракция может передаваться в том числе и сторонним инструментам, например, классическим инструментам проверки моделей (model checker tools), принимающим программу на модельном входном языке.

Во втором случае проверка происходит по мере построения частей абстракции. Например, как только достигнуто абстрактное состояние, включающее ошибочное, построение абстракции приостанавливается, и начинается проверка выполнимости найденного ошибочного пути. Одной из распространённых структур данных в этих подходах является *абстрактное дерево достижимости*. Метод *ленивой абстракции* позволяет не перестраивать абстрактное дерево достижимости целиком при уточнении абстрактного домена, например, при нахождении новых предикатов для *предикатной абстракции*.

Метод адаптивного анализа (CPA) используется для того, чтобы при построении абстракции можно было комбинировать построение предикатной абстракции с анализом потоков данных, абстрактные элементы решётки которого хранятся в абстракции. Данный метод позволяет, например, одновременно с анализом на основе предикатной абстракции осуществлять *анализ с явными значениями*.

В результате первого этапа либо доказывается, что ошибок в абстрактной модели нет, либо находится ошибочный путь в абстракции, называемый контрпримером. Так как абстракция включает все ошибочные пути, то отсутствие ошибок в абстрактной модели говорит о том, что ошибки нет в исходной программе, инструмент выдает вердикт SAFE. Данный вердикт может оказаться неправильным, т. е. может быть упущена ошибка, например, если программа содержит не поддерживаемые конструкции языка, которые моделируются неточно, но инструмент не завершается аварийно при их наличии.

Если обнаруживается контрпример, то он передается на проверку его выполнимости в исходной программе. Для этого, как правило, строится формула пути, кодирующая выполнение программы по пути к ошибке. Эта формула затем передается решателю. Семантика выполнения кодируется в терминах теорий, поддерживаемых решателями. Таким образом, точность проверки выполнимости зависит как от точности кодирования конструкций языка, так и от возможностей решателя. Неточности проверки выполнимости ведут к ложным срабатываниям, когда инструмент выдает вердикт UNSAFE в случае отсутствия реальной ошибки.

Когда путь оказывается невыполнимым, это означает, что абстракция программы нуждается в уточнении. На этапе уточнения абстрактного домена, невыполнимый путь используется для конструирования нового домена, такого, чтобы можно было исключить данный путь к ошибке в уточнённой

абстракции, а вместе с ним, возможно, и другие невыполнимые пути. Например, в предикатной абстракции невыполнимый путь к ошибке используется для поиска новых предикатов как с помощью синтаксических методов, основанных на эвристиках, так и с помощью методов *интерполяции*.

2.2. Предикатная абстракция (Predicate Abstraction)

В качестве модели программы может выступать предикатная абстракция [3,4], состояниями которой являются наборы предикатов, описывающих множества конкретных состояний программы. Как правило, строится абстракция существования (existential abstraction), в которой при существовании перехода между двумя конкретными состояниями, всегда существует переход между любыми двумя абстрактными состояниями, включающими данные конкретные состояния. Абстракция существования гарантирует сохранение в абстрактной модели достижимых путей в ошибочное состояние. Таким образом, если абстрактная модель не содержит ошибку, то в программе её тоже нет. Предикатная абстракция может строиться явно или непосредственно при поиске ошибки, например, при построении дерева достижимости.

Точность приближения состояний программы зависит от используемых предикатов. Предикаты состояния могут быть выражены в логике высказываний (пропозициональной логике), с высказываниями относительно пропозициональных переменных, логике первого порядка, расширяющей логику высказываний фиксированными функциональными и предикатными символами, кванторами всеобщности и существования или логике более высоких порядков. Функциональные символы и предикаты могут иметь дополнительную интерпретацию в виде теорий. Примерами теорий являются теории целых и вещественных чисел, теории списков, массивов, битовых векторов и т. д. Использование тех или иных предикатов в состояниях накладывает ограничения на возможности решателей. Чем сложнее предикаты, тем больше возможностей требуется от решателей.

Вычисление переходов между состояниями требует представления конструкций языка программирования в виде логических формул, и здесь, также как и для состояний, могут использоваться различные логики и теории. Кроме того, может использоваться дополнительная аксиоматизация, например, при моделировании памяти в виде аксиом для функций получения размещения, взятия адреса и разыменования, которое было предложено в работе [5]. При вычислении переходов могут использоваться также дополнительные данные, полученные другими видами анализа, например могут использоваться данные анализа указателей для подстановки потенциально равных значений указателей в конструируемую формулу.

В случае использования подхода CEGAR предикаты в состояние добавляются на этапе уточнения предикатов посредством анализа формулы пути контрпримера, например, с помощью интерполирующих решателей или синтаксических методов. Извлекаемые на данном этапе предикаты

существенно влияют на возможность получения более точной абстракции состояний программы.

Наиболее распространенными вариантами предикатных абстракций являются декартова предикатная абстракция и булева предикатная абстракция. В декартовой абстракции предикаты могут быть соединены лишь конъюнкцией. Абстракцией является наиболее строгая конъюнкция предикатов, покрывающая состояния программы.

Для данной абстракции вычисление состояния в окончании перехода сводится к проверке истинности каждого предиката в отдельности. Результирующее состояние является конъюнкцией из истинных предикатов. Таким образом, количество вызовов решателя при вычислении декартовой предикатной абстракции равно количеству предикатов в состоянии.

В булевой предикатной абстракции допускаются произвольные комбинации предикатов в логике высказываний. Абстракцией является наиболее строгая формула, покрывающая состояния программы. Вычисление такой абстракции требует получения набора векторов истинных предикатов, которых может быть экспоненциально много. Поэтому вычисление булевой абстракции требует большей эффективности от решателей.

2.3. Абстрактное дерево достижимости (Abstract Reachability Tree)

Абстрактное дерево достижимости (ART) [6], содержит пути, достижимые из начального состояния. Вершинами дерева являются описания состояний программы. Например, содержащие вершину ГПУ и состояние предикатной абстракции. Ребра дерева достижимости помечены переходами, состоящими из одной или нескольких инструкций ГПУ. В случае, если используется кодирование последовательностей инструкций в ГПУ в один переход, так называемое представление программы в виде больших блоков (large block encoding) [7], одному ребру в ART может соответствовать последовательность инструкций, включающая в себя инструкции ветвления, точки слияния и вызовы функций.

Построение дерева состоит в вычислении конечного состояния по состоянию в текущей вершине и переходу, которым помечено ребро. Процесс вычисления зависит от используемого состояния. Например, в случае предикатной абстракции требуется вычисление следующего состояния по начальному абстрактному состоянию и набору инструкций, как правило, с использованием решателей.

Большие блоки увеличивают нагрузку на решатели, так как запросы к решателю при построении абстракции становятся сложнее. Например, если переход включает точки слияния, то конструируемая формула включает в себя дизъюнкцию для путей входящих в эту точку, что усложняет решение формулы. С другой стороны, становится меньше количество запросов к решателям. В связи с тем, что решатели постоянно совершенствуются и

способны решать все более сложные формулы за приемлемое время, уменьшение запросов в ряде случаев может ускорить анализ.

При применении адаптивного статического анализа каждый алгоритм CPA хранит часть своего состояния дерева и вычисляет следующее состояние для передаваемого перехода. Состояние алгоритма CPA может включать в себя состояние других CPA, вложенных в него. В этом случае при выполнении перехода этот CPA вызывает вложенные CPA для изменения своих элементов состояния. Неосуществимость перехода в одном из алгоритмов влечет неосуществимость перехода в целом.

Построение дерева достижимости происходит одновременно с проверкой заданного свойства программы. В случае проверки свойства достижимости, построение дерева прекращается при обнаружении ошибочного состояния, которое может задаваться как инструкция, помеченная специальной меткой. При использовании адаптивного анализа ошибочное состояние может определяться алгоритмом CPA.

Конечность дерева достижимости обеспечивается конечностью множества абстрактных состояний и тем, что пути из абстрактных состояний, покрываемых другими абстрактными состояниями, не перебираются. Покрытие состояний означает вложенность соответствующих множеств состояний программы. Для предикатной абстракции покрытие проверяется как импликация.

Дерево достижимости может строиться без вычисления следующего абстрактного состояния [8]. В этом случае вначале предполагается, что состояние может быть любым. На этапе уточнения предикатов, вместо выявления новых предикатов для уточнения всей абстракции, выделяются предикаты, которые уточняют состояния дерева, входящие в путь контрпримера. Таким образом, абстракция уточняется только для путей, ведущих к ошибке.

2.4. Ленивая абстракция (Lazy Abstraction)

В подходе CEGAR предикаты, для которых строится абстрактное дерево достижимости, постоянно изменяются, в результате дерево нужно перестраивать каждый раз заново. Для решения данной проблемы используется метод, описанный в [9]. Данный метод получил название «ленивая абстракция» (от англ. «Lazy Abstraction»). После осуществления анализа контрпримера (потенциального ошибочного пути) не осуществляется построение всего абстрактного дерева для всей программы «с нуля». Вместо этого, дерево пересматривается частично, только для тех путей, в которых его пересмотр мог бы повлиять на наличие ошибочных состояний.

При анализе пути контрпримера алгоритм ленивой абстракции находит в нем самую первую с конца пути вершину, из которой ошибка уже недостижима. Например, начиная с конца пути последовательно перебираются вершины, в которых ошибка достижима, до тех пор пока не дойдем до вершины, в

которой ошибка станет недостижима. Все пути в дереве, которые начинаются с этой вершины, необходимо пересмотреть. При пересмотре абстракции для таких путей используются предикаты, как предоставленные на вход, так и полученные при анализе ошибочного пути. Отметим, что остальная часть абстракции, с учетом новых предикатов, тоже может быть уточнена, и это будет сделано, если в ней находятся вершины, ранее покрытые другими пересматриваемыми вершинами. Не покрытые же ничем вершины в оставшейся части абстракции пересматривать не обязательно, поскольку, хотя пересмотр и приведет эту часть абстракции в более соответствующий реализуемым состояниям памяти вид, это не повлияет на решение о (не)достижимости ошибочного состояния, поскольку даже более грубая абстракция доказывала его недостижимость по путям в этой части абстракции.

В [9] авторами предложен расширяемый алгоритм осуществления ленивой абстракции, который при соблюдении определенных условий на расширения, всегда корректен, если он завершается.

Тем не менее, этот подход обладает тем недостатком, что если ранее рассмотренная вершина тем или иным образом попадала в очередь пересмотра абстракции, абстракцию в ней и в ее потомках необходимо пересматривать с учетом всех предикатов, когда-либо полученных при анализе контрпримеров или на вход.

В статье [10] предложено развитие подхода ленивой абстракции, преодолевающее эту трудность. В качестве решения предлагается сделать предикаты локальными для вершин дерева. На этапе уточнения предикатов новые предикаты приписываются только к вершинам на ошибочном пути. Уточнение абстракции в других вершинах не производится, что делает абстракцию еще более ленивой.

2.5. Адаптивный статический анализ (Configurable Program Analysis)

В теории известно, что анализ потоков данных широко используемый в легковесных анализаторах и методы проверки моделей, используемые в тяжеловесных анализаторах, могут быть сведены друг к другу [11-15]. Однако, данные теоретические работы мало влияют на практику верификации. Легковесные анализаторы проверяют некоторые простые свойства в больших программах. Тяжеловесные инструменты верификации нацелены на исключение ложных срабатываний для сравнительно небольших программ.

Будучи направленными на эффективность, легковесные статические анализаторы обычно нечувствительны к путям, так как даже с наиболее эффективными абстрактными множествами состояний анализаторы теряют точность в точках соединения. С другой стороны, тяжеловесные анализаторы, будучи направленными на точность, не соединяют пути выполнения вовсе, рассматривая каждый путь отдельно.

Главная идея адаптивного статического анализа состоит в том, чтобы объединить преимущества анализа потоков данных и методов проверки моделей и разработать подход, в котором можно конфигурировать их взаимосвязь. В анализе потоков данных конфигурируемость заключается в выборе соответствующего интерпретатора: абстрактного множества, функции перехода, оператора расширения. Недавние работы позволяют конфигурировать несколько интерпретаторов [13-15].

С одной стороны, в анализе потоков данных алгоритм работает с ГПУ, распространяя информацию вдоль ребер, с другой стороны, алгоритмы проверки моделей в тяжеловесном анализе разворачивают ГПУ в дерево достижимых состояний до тех пор, пока одно из состояний не будет покрыто другим состоянием.

В адаптивном статическом анализе, предложенном в [16], допускается конфигурировать алгоритм анализа CPA, выбирая оператор слияния (merge) и проверки завершения анализа пути (stop). Кроме того, в адаптивном статическом анализе можно конфигурировать сразу несколько алгоритмов CPA, осуществляющих разные виды анализа.

Оператор слияния (merge) определяет, когда две вершины дерева достижимости сливаются в одну, а когда они обходятся по отдельности. В анализе потоков данных слияние происходит всегда, когда они относятся к одной и той же точке программы. В классических методах проверки моделей вершины никогда не объединяются.

Проверка завершения (stop) определяет, когда рассмотрение данного пути останавливается в данной вершине и не рассматриваются дальнейшие продолжения выполнений. В анализе потоков данных останов происходит, когда не находится состояний включающих новые конкретные состояния, т. е. достигается неподвижная точка. В методах проверки моделей останов происходит, когда одно множество конкретных состояний, соответствующих абстрактному состоянию, является подмножеством множества конкретных состояний, соответствующих какому-то другому абстрактному состоянию.

Как было сказано выше теоретически возможно переопределить анализ, состоящий из операторов слияния и останова, как единый интерпретатор в анализе потоков данных. Однако цель адаптивного статического анализа — создать практически пригодный метод конфигурируемого анализа, на основе использования готового набора уже существующих реализаций алгоритмов анализа. Это особенно важно, когда мы хотим комбинировать несколько алгоритмов анализа одновременно. Для этого определяются составной оператор слияния и составной оператор останова, которые составлены из компонент, являющихся соответствующими операторами слияния и останова алгоритмов CPA [17,18].

Использование адаптивного статического анализа в ряде случаев позволяет увеличить скорость анализа. Например, алгоритм CPA, реализующий анализ с явными значениями совместно с предикатным анализом, позволяет

показывать недостижимость ветвей условных операторов с проверкой значений переменных, которым в программе присвоили константы, за счет отслеживания присваивания переменным константных значений. Выигрыш по скорости происходит за счет того, что не требуется тратить время на уточнение предикатной абстракции. С другой стороны, анализ с явными значениями требует дополнительную память для хранения состояний.

Адаптивный статический анализ может влиять на точность, например, подключение алгоритма CPA, реализующего анализ рекурсивных структур данных, позволяет учитывать семантику потенциально бесконечных структур данных, таких как списки, множества и т. д.

2.6. Анализ с явными значениями (Explicit Analysis)

В анализе с явными значениями для каждой переменной абстрактным состоянием является множество всех ее возможных значений. Если число хранимых возможных значений для переменной превышает некоторое установленное пороговое число, то с этого момента считается, что переменная может принимать любое значение. Подробная формализация этого вида анализа описана в статье [18] в разделе 3.1 «CPA+ for Explicit Analysis».

При конструировании переходов учитывается семантика инструкций, присваивающих переменным из абстрактного состояния явные значения, например, константы или выражения, которые могут быть вычислены в константы в данном абстрактном состоянии. Также учитывается семантика выражений в условных операторах, для которых можно вывести явное значение переменной. Анализ с явными значениями достаточно прост в реализации и не требует предварительной настройки точности (такой как выведение предикатов для абстракции).

Анализ с явными значениями достаточно эффективен в случае, когда число переменных и принимаемых ими возможных значений относительно невелико. Но при увеличении этого числа он либо теряет точность (при малом установленном пороговом значении), либо требует слишком много ресурсов (при большом). Предикатная абстракция гораздо лучше справляется с большим числом переменных, но требует предварительного и довольно сложного анализа контрпримера для вывода нужных предикатов. Адаптивный статический анализ позволяет совмещать анализ с явными значениями переменных и предикатную абстракцию.

2.7. SMT/SAT решатели

В широком смысле решающие процедуры (от англ. «decision procedures», или решатели, от англ. «solvers») – это алгоритмы, которые, приняв на вход задачу разрешимости, то есть вопрос, сформулированный в рамках какой-либо формальной системы (аксиоматической теории), и требующий ответа «да» или «нет», выдают на выходе соответствующий корректный результат. В узком смысле мы рассматриваем решатели для алгоритмически разрешимых задач, сформулированных в рамках теорий первого порядка, и используемых на

практике в областях верификации, доказательства корректности, оптимизации и др. Поскольку возможности многих технологий в этих областях ограничены возможностями используемых решателей, последние остаются объектом многих активных исследований во всем мире, как в академической среде, так и в промышленности.

Решающие процедуры можно разделять по поддерживаемым ими логикам и теориям. Решатели для логики высказываний, т.е. классической логики нулевого порядка, которые по сути являются инструментами для решения задачи выполнимости булевых формул (ВЫП), называются SAT-решателями. Решатели для формул классической логики первого порядка с равенством, заданных в рамках комбинации некоторых аксиоматических теорий, называются SMT-решателями. Решатели могут ограничиваться поддержкой только некоторых видов формул, например только формул без кванторов, в том числе решатель может поддерживать различные теории. Среди наиболее часто используемых на практике теорий – вещественная и целочисленная линейная арифметика, неинтерпретируемые функции, массивы и битовые векторы.

Практически все современные решатели для пропозициональной (логической) части формулы используют либо схему DPLL (Davis-Putnam-Logemann-Loveland) [19], основанную на поиске с возвратом, для решения задачи в виде КНФ, либо основаны на суперпозиционном исчислении (от англ. «superposition calculus»), расширении резолютивного вывода. Для взаимодействия с решающими процедурами для подформул, заданных в рамках какой-либо теории, используются в основном техники комбинирования Нельсона-Оппена (Nelson-Oppen Combination Procedure), пропозиционального кодирования (от англ. «propositional encodings») и отложенного комбинирования теорий (DTC, Delayed Theory Combination) [19]. Отложенное комбинирование теорий предложено в статье [20] и используется в решателе MathSAT.

2.8. Интерполяция

Интерполяция используется на этапе уточнения абстрактного домена для последующего уточнения абстракции, в первую очередь для уточнения предикатов в предикатной абстракции. Исходными данными для уточнения предикатов является невыполнимая формула пути. В результате должны быть найдены новые предикаты, таким образом, чтобы при повторном построении абстракции исключить из нее невыполнимый путь. Метод итерполяции состоит в том, чтобы использовать интерполянты Крейга (Craig), позволяющие находить неявные зависимости [10] (см. определение интерполянта в [21]). Интерполянт существует всегда, когда формулы заданы в логике предикатов первого порядка. В статье [22] показано, что интерполянт может быть построен за линейное время по размеру резолютивного

доказательства для формул в теориях равенства с неитерпретируемыми функциями и линейных неравенств.

Интерполяционные методы в дополнение к предикатам позволяют определять точки в программе, для которых эти предикаты полезны. Это позволяет рассматривать не глобальное множество предикатов, а разные множества предикатов для разных точек программы и соответственно перестраивать абстракцию только для частей программы, связанных с изменившимся набором предикатов.

2.9. Ограничиваемая проверка моделей (Bounded Model Checking)

Метод ограничиваемой проверки моделей (от англ. BMC, Bounded Model Checking) [23] — это одна из наиболее часто применяемых формальных техник при разработке полупроводниковых устройств. Метод обязан своим успехом большим возможностям SAT решателей. Он был предложен в 1999 как вспомогательный для неограниченного метода проверки моделей на основе BDD.

Этот метод основан на разворачивании графа состояний программы на конечное число шагов. На каждом этапе проверяется, может ли спецификация быть нарушена не более, чем за текущее выбранное число шагов. Если нарушение не обнаружено, выполняется проверка условий разворачивания (от англ. «unwinding assertions»), таких как условия в заголовках циклов, для того, чтобы исключить дальнейшее разворачивание графа по неосуществимым путям. Такая проверка может показать, что дальнейшее увеличение числа шагов бессмысленно, т.к. ни одно из условий разворачивания не выполнено. Это доказывает корректность исходной программы. В противном случае, число шагов увеличивают и проверку повторяют до тех пор, пока не обнаружат нарушение спецификации, либо не выйдут за пределы указанного максимального числа шагов. Проверка корректности развёрнутого фрагмента графа состояний обычно подразумевает кодирование условия нарушения спецификации в виде задачи для решателя.

В качестве оптимизаций применяются отдельное раскрытие циклов и вычисление порога полноты (от англ. «completeness threshold»).

В BMC, как правило, используются решатели, поддерживающие битовую арифметику.

BMC – это наилучший метод для нахождения поверхностных ошибок, предоставляющий полную трассу контрпримера в случае их обнаружения. Инструментам, реализующим этот метод, как правило, свойственна высокая точность анализа. Они поддерживают широчайший набор конструкций в программе, включая динамически выделяемые структуры в памяти, операции с массивами, использование арифметики указателей, битовые операции над числами, ограниченность размеров целочисленных типов данных (например, перечислимых) и другие, не требуя при этом встроенного знания о структурах

данных в программе. С другой стороны, доказательство отсутствия ошибок гарантируется только для простых программ, т.е. программ без глубоких циклов.

2.10. Анализ рекурсивных структур данных (Shape Analysis)

Анализ рекурсивных структур данных [24-26] — это анализ, который моделирует потенциально бесконечные рекурсивные структуры данных в куче, используя абстракции, основанные на графах связей. Примерами таких структур данных являются списки, множества, отображения, деревья, и т.д.. Для построения графов необходимо задание классов связей, описывающих связи между элементами структуры данных, которые необходимо отслеживать. Класс связей в том числе задает предикаты на данные, хранимые в элементах, например на значения данных, хранящиеся в списке. Граф может включать суммарные вершины, описывающие одно или несколько состояний, что дает возможность представлять бесконечные части структуры данных, неразличимые с точки зрения заданного класса связей.

Анализ рекурсивных структур данных позволяет использовать предикаты, которые сложно найти другими методами. Например, при использовании предикатной абстракции этап уточнения может выполняться бесконечное число раз, т. к. каждый раз будет находиться предикат для конечного экземпляра бесконечной структуры данных.

2.11. Модульность анализа

Под модульностью анализа понимается возможность анализировать программу по частям, используя результаты анализа части программы для анализа программы в целом. Наиболее быстрые подходы, применяемые в легковесных инструментах, анализируют каждую функцию программы отдельно, составляя для каждой проанализированной функции аннотацию, хранящую значимые результаты анализа. Для анализа потока данных аннотация может хранить некоторые элементы решетки. В случае использования предикатной абстракции аннотация может хранить абстрактное состояние на выходе из функции при условии её выполнения из начального абстрактного состояния [27]. Модульность может обеспечиваться не только на уровне функций. В подходе [28] запоминаются результаты анализа рёбер в абстрактном дереве достижимости, которые могут содержать как вызовы функций целиком, так и отдельные итерации циклов. Таким образом, в случае повторного прохода по аналогичному ребру, переиспользуются результаты предыдущего анализа. Модульность во многих случаях ускоряет анализ, а кроме того, делает инструмент масштабируемым.

2.12. Генерация тестов

Под генерацией тестов понимаются методы, которые позволяют генерировать входные параметры программы, так чтобы выполнение программы с

заданными параметрами обеспечивало покрытие того или иного участка программы. Нацеливание инструмента генерации тестов на покрытие ошибочного состояния программы, определяемого переходом на ошибочную метку, дает возможность использовать их для проверки свойств достижимости.

Инструмент статической верификации	**CEGAR**	**Предикатная абстракция**	**Абстрактное дерево достижимости**	**Ленивая абстракция**	**Адаптивный анализ**	**Анализ с явными значениями**	**Интерполяция**	**Решатель**	**Ограничиваемая проверка моделей**	**Анализ рекурсивных структур данных**	**Модульность анализа**	**Генерация тестов**
Blast	+	+	+	+	±	+	+	SMT				
CPAchecker	+	+	+	+	+	+	+	SMT			±	
HSF(C)	+	+	+				+	CUST				
Satabs	+	+						SAT				
Slam	+	+						SMT				
Wolverine	+		+	+			+	SAT				
Yogi	+	+						SMT			+	+
Cbmc								SAT	+			
Esbmc								SMT	+			
Llbmc								SMT	+			
FShell								SAT	+			+
Predator								—		+		

Табл. 1. Методы и техники инструментов статической верификации

3. Характеристики инструментов статической верификации

3.1. Поддержка конструкций анализируемой программы

Булевы выражения – это выражения логической конъюнкции, дизъюнкции, отрицания, равенства, неравенства. Не включает побитовые или/и/отрицания и т. д.

Линейные выражения – это выражения сложения, вычитания, умножения на константу, линейные неравенства, равенства.

Умножение/деление могут встречаться как для целых типов, так и для вещественных. Для последних требуется поддержка семантики вещественных чисел.

Битовое представление типов позволяет учитывать особенности представления целых типов в виде ограниченного набора бит, учитывать при этом переполнение для целых и округление для вещественных типов.

Битовые выражения включают битовые операции, такие как и/или/отрицание, исключающее или, сдвиги и т. д.

Выражения с указателями включают присваивание указателей и арифметику указателей, такую как сложение адреса с целым, разность адресов и т. д. Задача инструмента верификации – отслеживать изменения указателей так, что изменение памяти доступной по одному указателю, отражается при чтении этой же памяти по другому указателю на эту же память. Здесь можно выделить два подкласса: инструменты верификации, которые умеют работать только с присваиваниями указателей, и инструменты верификации, которые отслеживают арифметику указателей.

Поддержка рекурсивных структур данных означает возможность отслеживания взаимосвязей указателей в куче и нахождения общих шаблонов связей, дающих возможность распознавать потенциально бесконечные структуры данных и строить только те части из этих структур, которые необходимы для проверки свойства.

Поддержка указателей включает разыменование переменной указателя на функцию так, что будет осуществлен переход выполнения на тело функции, соответствующее значению указателя в данной переменной.

Поддержка рекурсии в наиболее полном случае означает нахождение соответствующих инвариантов для доказательства проверяемого свойства. Подходы на основе составления аннотации вызова функции осуществляют анализ с помощью использования аннотации для анализа самой себя. Также есть подходы с ограничением на глубину рекурсии.

Поддержка многопоточности подразумевает, что учитывается семантика выполнения в нескольких потоках, которая моделируется, например, с помощью чередования выполнения инструкций разных потоков. Имеет

значение, какие примитивы синхронизации поддерживает данный инструмент верификации, поддерживается ли динамическое создание потоков.

3.2. Масштабируемость

Под масштабируемостью понимается возможность применения инструмента верификации к большим и очень большим размерам исходного кода программы. Наиболее серьезных успехов в плане масштабируемости добиваются легковесные подходы к верификации, позволяя анализировать программы, состоящие из миллионов строк исходного кода, за счет использования модульных подходов к анализу. Тяжеловесные подходы способны анализировать десятки тысяч строк кода. Они могут использовать модульные подходы, а также подходы с абстракцией и уточнением.

3.3. Доказательство свойств

Инструменты, осуществляющие доказательство заданного пользователем свойства, как правило, предоставляют некоторые гарантии правдивости ответа, т. е. того, что свойство действительно не нарушено. Например, в CEGAR подходах абстракция программы может выбираться таким образом, что все выполнения, нарушающие заданное свойство в программе, присутствуют и в абстрактной модели, с точностью до поддерживаемых инструментом конструкций языка программирования при построении абстракции. Таким образом, отсутствие нарушения в абстрактной модели гарантирует ее отсутствие в программе. В подходах без доказательства гарантии отсутствия ошибок могут быть либо для очень простых программ, либо их может не быть вовсе. Например, в подходе ограничиваемой проверки моделей доказательство не осуществляется и утверждать об отсутствии нарушения свойства можно только вплоть до заданного ограничения на количество итераций цикла.

3.4. Проверяемые свойства

Свойства достижимости говорят о том, что программа не достигает состояний, в которых она делает что-то плохое. Например, программа не должна достигать состояния, в котором не выполнено выражение, передаваемое функции assert или состояния, помеченного специальной меткой ошибки.

Свойство завершаемости означает, что программа когда то завершится. Для проверки этого свойства необходимо задать точку входа в программу и точки выхода, которых должна достичь программа.

4. Инструменты статической верификации

4.1. Blast

Инструмент Blast [6,29] реализует подход CEGAR с использованием декартовой предикатной абстракции. В инструменте реализован метод ленивой абстракции на основе абстрактного дерева достижимости.

Реализован вариант адаптивного анализа, позволяющий параллельно с построением элементов предикатной абстракции строить элементы анализа потоков данных, основанного на задании решетки (lattice) в качестве алгоритма CPA, с различными вариантами функций слияния (merge) и функций покрытия (stop). В инструменте реализована решетка SymbolicStore, осуществляющая анализ с явными значениями. Для получения новых предикатов и уточнения абстракции инструмент использует интерполирующие решатели. Анализ контрпримера на выполнимость осуществляется решателями, работающими в формате SMT-LIB.

Инструмент Blast поддерживает программы, содержащие булевы и линейные выражения. Поддерживаются конструкции вызова функций по имени, с помощью подстановки тела функции. Поддерживает семантику присваивания указателей на базовые типы, обращения к полям структуры, присваивание полей и структуры целиком. Не поддерживаются операции сложения и вычитания с адресами, например, не поддерживаются смещения для элементов массива, получение внешней структуры с помощью вычитания смещения поля вложенной структуры относительно внешней. Поддерживаются указатели на функции. Анализ рекурсивных функций осуществляется с помощью ограничения на глубину раскрытия. От неподдерживаемых конструкций Blast абстрагируется до тех пор пока они не потребуются для доказательства свойства. В этом случае может быть выдан ложный вердикт.

Масштабируемость в Blast обеспечивается за счет применения абстракции. В ядре ОС Linux встречаются драйверы размером порядка 40 тыс. строк, на которых Blast успешно завершается [30]. Абстракция в Blast строится таким образом, чтобы включать все ошибочные пути программы, что позволяет доказывать выполнение заданного свойства. Blast поддерживает проверку свойств достижимости, задаваемых в виде метки, приписанной к точке в программе.

4.2. CPAchecker

Инструмент CPAchecker [31] во многом аналогичен инструменту Blast. В отличии от инструмента Blast, адаптивный статический анализ реализован в нем полностью, т. е. инструмент полностью компонуется из алгоритмов анализа CPA, задаваемых в конфигурации. Это позволяет добавлять новые виды анализа или изменять анализатор без его перекомпиляции и делает его удобным для проведения экспериментов с различными вариантами анализа.

В инструменте CPAchecker реализована поддержка длинных переходов в абстрактной модели (больших и адаптивных блоков), которые включают не только базовые блоки, содержащие выражения без вызова функций и ветвлений, но и слияния путей по выходу из цикла и вложенные вызовы функций [32].

Инструмент отчасти обеспечивает модульность анализа, реализуя технику сохранения абстрактных переходов, также называемых абстрактными блоками (Abstract Block Memorization) [28], для того чтобы не перевычислять их при последующем прохождении. Для этого из вычисленного абстрактного блока удаляется информация о зависимости с другими блоками, так что абстрактный блок можно использовать при выполнении в другом абстрактном состоянии. Например, для вложенного цикла из абстрактного состояния удаляется информация о переменных внешнего цикла. На следующей итерации внешнего цикла для анализа итерации внутреннего цикла используется сохраненный абстрактный блок.

В части поддержки конструкций языка программирования CPAchecker также аналогичен инструменту Blast. В настоящий момент CPAchecker поддерживает выражения с присваиваниями указателей на примитивные типы. Присваивание указателей на структуры и арифметические операции не поддерживаются.

По умолчанию CPAchecker считает, что все функции без тела являются неинтерпретируемыми, т. е. для одинаковых аргументов возвращают одинаковые значения, и что они не имеют побочных эффектов. Существует возможность задания неопределенных функций, которые каждый раз выдают произвольное значение вне зависимости от параметров.

4.3. HSF(C)

HSF(C) [33] — это инструмент статического анализа, основанный на использовании подхода CEGAR и булевой абстракции программ.

Инструмент поддерживает булевы и линейные выражения, умеет игнорировать неподдерживаемые конструкции. Не учитывает семантику выражений с указателями.

Отличительной особенностью инструмента является то, что он позволяет проверять не только свойства достижимости, но и завершаемости [34]. Доказательство завершаемости состоит в том, чтобы предложить функции ранжирования и проверить, что соответствующее им отношение фундировано. Долгое время в предлагаемых решениях данной задачи старались построить единственную функцию, обладающую свойством ранжирования. Однако, такой подход не давал возможности полностью автоматизировать доказательство и был непригоден для практически значимых программ.

Основным отличием от предыдущих инструментов является смещение основной вычислительной нагрузки от задачи нахождения кандидатов

ранжирующей функции к задаче проверки того, что предложенная функция является функцией ранжирования. В классическим методе основная нагрузка состоит в том, чтобы построить такую функцию, чтобы она сразу же обладала свойством ранжирования. Более того, в предлагаемом подходе строится не одна функция, а набор догадок о возможных функциях ранжирования, некоторые из которых могут быть неудачными догадками. Таким образом, множество догадок как объединение ранжирующих функций не обязано само быть функцией ранжирования, а может являться только надмножеством над ней. Это происходит в силу того, что объединение ранжирующих функций в общем случае не является ранжирующей функцией.

Инструмент также реализует методы многопоточного анализа, опирающиеся на автоматическое обнаружение переходов, выполняемых окружением, используя предикатную абстракцию переходов. Схема абстракции и уточнения основана на хорновских дизъюктах, т.е. ограничения на переходы записываются в виде хорновских дизъюнкций (англ. «Horn clause»), затем эти ограничения решаются с помощью общего алгоритма для хорновских дизъюнкций без рекурсии.

4.4. Satabs

Satabs [35] — это инструмент автоматической верификации, основанный на использовании подхода CEGAR и булевой абстракции программ. Отличительной особенностью инструмента является использование SAT решателей вместо SMT решателей, что позволяет работать с битовыми представлениями чисел и, таким образом, более точно моделировать семантику языка Си, включая семантику переполнения, арифметику указателей, массивы и объединения.

Для получения новых предикатов и уточнения абстракции Satabs использует синтаксические методы [36], основанные на вычислении слабейшего предусловия для трассы контрпримера. Эти методы не гарантирует получение предикатов, делающих заданный путь невыполнимым, поэтому несмотря на поддержку большинства конструкций языка при составлении абстракции, верификация может завершиться неудачно.

В своей работе инструмент опирается на сторонние инструменты проверки моделей, такие как Cadence SMV [37], Booро [38], Boom [39]. Инструмент строит предикатную абстракцию существования и передает ее стороннему инструменту проверки моделей, а затем использует полученный от него результат для дальнейшего уточнения абстракции или для выдачи итогового результата. Инструмент Satabs умеет строить абстракции для многопоточных программ [40], опираясь при этом на возможности по проверке параллельных программ сторонними инструментами проверки моделей.

4.5. Slam (Slam2)

Slam (Slam2) – это инструмент автоматической верификации, основанный на использовании подхода CEGAR и булевой абстракции программ. Во многом благодаря успехам этого инструмента предикатная абстракция в настоящий момент является доминирующей техникой статической верификации программного обеспечения [41].

В инструменте Slam используется построение абстракции существования. Абстрактная программа, соответствующая данному построению, является булевой программой, в которой используются переменные только булева типа и те же графы потоков управления, что и в исходной программе, включая вызовы процедур [42].

Спецификация для проверки в этом инструменте, так же, как и в Blast, может быть сформулирована в виде условия достижимости. При этом проблема проверка достижимости для булевых программ разрешима, несмотря на неограниченность стека вызовов функций. Интуитивно это основано на том, что последующее состояние определяется полностью по данным на вершине стека и значениям глобальных переменных, которые принимают конечное множество значений [43].

Для поиска набора новых предикатов для уточнения абстракции по контрпримеру в Slam вначале используется одно из простых решений – это поиск ядра недостижимости (от англ. «unsatisfiable core»), подмножества предикатов, встречающихся в формуле пути, конъюнкция которых невыполнима. Есть несколько способов нахождения данного множества. В инструменте Slam для этого используется жадный алгоритм [44]. Если предикатов, полученных в результате поиска ядра недостижимости, оказывается недостаточно для отсечения неосуществимого контрпримера, используется синтаксический метод поиска новых предикатов, описанный в [45]. Обоснование его полноты можно найти в [46].

Инструмент Slam используется в Microsoft Static Driver Verifier (SDV), подробно рассмотренном в соответствующем подразделе, для проверки корректности использования драйверами интерфейсов, предоставляемых ядром ОС Microsoft Windows. О некоторых результатах его применения с этой целью написано в статье [47].

4.6. Wolverine

Инструмент Wolverine [48] основан на подходе CEGAR. Он использует абстрактное дерево достижимости при построении переходов которого не используется вычисление постусловия. Абстрактные состояния в вершинах дерева уточняются посредством интерполяции по контрпримерам.

Встроенный интерполятор Wolverine версии 0.5c не предоставляет поддержки линейной арифметики, инвариантов с кванторами, арифметики с указателями.

4.7. Yogi

Инструмент Yogi [49] объединяет возможности статического и динамического анализа.

Динамический анализ позволяет получить аппроксимацию множества состояний снизу, т. е. состояния и пути, перебираемые в динамическом анализе, реально достижимы. Данный анализ традиционно хорошо подходит для того, чтобы показать, что ошибки есть. Тогда как сильная сторона статического анализа заключается в том, чтобы показать, что ошибок нет.

В работе [27] предлагается метод, позволяющий одновременно выполнять статический и динамический анализы. Статический анализ строит «может» (от англ. «may») абстракцию, включая в нее все осуществимые и, возможно, неосуществимые пути. Динамический анализ строит «должен» (от англ. «must») абстракцию, включая только осуществимые пути, но, возможно, не все.

Объединение статического и динамического анализа позволяет статическому анализу не доказывать неосуществимость путей, для которых динамический анализ показал осуществимость, а динамическому анализу позволяет не искать ошибку там, где статический анализ показал, что ее быть не может. Этот метод реализован в инструменте Yogi [49]. Инструмент встраивается в Microsoft SDV, используемый для верификации драйверов ОС Windows. По результатам, приведенным в статье [49], инструмент сравним с основным используемым в настоящее время инструментом Slam.

4.8. Cbmc

Инструмент Cbmc [50] — это инструмент ограничиваемой проверки моделей для верификации программ на ANSI-C и C++.

Поддерживает семантику большинства выражений, опираясь на решатели, поддерживающие решение с представлением целых чисел в виде битовых векторов. Для поддержки операций умножения/деления используется библиотека, вычисляющая результат через операции сложения и вычитания. Таким образом, Cbmc заменяет операции умножения/деления на вызов функций библиотеки, которые затем анализируются. Рекурсия анализируется с помощью ограничения на глубину раскрытия.

4.9. Esbmc

Инструмент Esbmc [51] во многом аналогичен Cbmc, и использует его компоненты в процессе верификации. В отличие от Cbmc он использует SMT решатели. Кроме того, в Esbmc реализована возможность проверки параллельных программ посредством чередования инструкций различных потоков с ограничением на количество переключений между потоками.

4.10. Llbmc

Инструмент Llbmc [52] реализует BMC подход, используя внутреннее представление LLVM. Он направлен на представление типов с битовой точностью и нахождение ошибок, связанных с неправильной работой с памятью. Инструмент использует SMT решатель Stp, который поддерживает битовые векторы и массивы.

4.11. FShell

FShell [53] — это инструмент генерации тестов для C программ. Пользователем задается критерий тестового покрытия, который необходимо достигнуть. Исходя из этого критерия методом BMC генерируются формулы, решение которых дает входные данные, необходимые для покрытия того или иного пути. В качестве критерия покрытия может также выступать достижимость метки в C программе. Инструмент использует SAT решатель MiniSAT.

4.12. Predator

Predator [54] — это анализатор программ, построенный на основе логики разделения (separation logic) и нацеленный на анализ программ, содержащих динамические списки. Формулы логики разделения, описывающие потенциально бесконечные структуры связей в куче, представляются в виде графов. Инструмент не использует решатели и реализован как встраиваемый модуль к GCC.

Характеристики инструментов статической верификации представлены в таблице 2.

Инструмент статической верификации		**Поддержка конструкций входной программы**											**Масштабируемость**	**Доказательство свойства**	**Свойства достижимости**	**Свойства завершаемости**
		Булевы выражения	**Линейные выражения**	**Умножение/деление**	**Вещественные числа**	**Битовое представление типов**	**Битовые выражения**	**Выражения с указателями**	**Рекурсивные структуры данных**	**Указатели на функции**	**Рекурсия**	**Многопоточность**				
CEGAR	Blast	+	+	−	−	−	−	±	−	+	±	−	±	+	+	−
	CPAchecker	+	+	−	−	−	−	±	−	+	±	−	±	+	+	−
	HSF(C)	+	+	−	−	−	−	−	−	−	−	+	±	+	+	+
	Satabs	+	+	+	+	±	±	±	−	+	±	+	±	+	+	−
	Slam	+	+	−	−	−	−	+	−	−	−	−	±	+	+	−
	Wolverine	+	±	−	−	−	−	±	−	−	−	−	±	+	+	−
	Yogi	+	+	−	−	−	−	+	−	−	±	−	+	+	+	−
BMC	Cbmc	+	+	+	+	+	+	+	−	+	±	−	−	−	+	−
	Esbmc	+	+	+	+	+	+	+	−	+	±	+	−	−	+	−
	Llbmc	+	+	+	−	+	+	+	−	−	±	−	−	−	+	−
	FShell	+	+	+	−	+	+	+	−	+	±	−	−	−	+	−
	Predator	+	±	−	−	−	−	+	+	+	±	−	+	+	+	−

Табл. 2. Характеристики инструментов статической верификации

4.13. Результаты соревнований

В таблице 3 представлены результаты международных соревнований по верификации SV-COMP-2012 [55] в категории DeviceDrivers64. Данная категория состоит из драйверов, подготовленных к верификации.

Категория DeviceDrivers64		**Корректные результаты (всего 41 файл)**	**Упущенные ошибки**	**Ложные предупреждения**	**Баллов (макс. 66)**	**Среднее время работы (сек.)**	**Среднее время для корректных результатов (сек.)**
CEGAR	Blast	33			55	76	42
	CPAchecker	33		2	49	117	15
	HSF(C)	–					
	Satabs	17			32	634	188
	Slam	–					
	Wolverine	12			16	585	108
	Yogi	–					
BMC	Cbmc	–					
	Esbmc	7	1		10	176	124
	Llbmc	3		2	1	293	37
	FShell	0			0	0,5	0
	Predator	0			0	44	0

Табл. 3. Результаты соревнований SV-COMP 2012

Каждый подготовленный драйвер включает необходимое окружение и проверки свойств достижимости, встроенные непосредственно в код. Таблица 3 показывает результаты запуска инструментов, участвовавших в соревнованиях, на наборе из 41 входного файла. Прочерком обозначены инструменты, не принимавшие участия в соревнованиях или в данной категории.

Инструмент выдает вердикт SAFE, если он не обнаружил ошибок в программе. Если он обнаружил ошибку, то выдает UNSAFE. Если же по каким то причинам инструмент не может определенно сказать имеется ли в программе ошибка, то выдается вердикт UNKNOWN.

Корректным считается результат, когда либо при вердикте инструмента UNSAFE в программе имеется ошибка, либо для программы без ошибок выдается результат SAFE. При подсчете очков в соревновании поощрялись правильные вердикты SAFE (+2 балла), правильные UNSAFE (+1 балл). За UNKNOWN баллы не начислялись. С другой стороны, за неправильные вердикты снимали баллы в двойном количестве. За каждый неправильный UNSAFE (ложное срабатывание) -2 балла, за неправильный SAFE (упущенная ошибка) -4.

Среднее время считалось от общего времени на всех задачах, в том числе и тех, на которых был получен вердикт UNKNOWN, например по причине превышения лимита по времени, который составлял 15 минут. Среднее время для корректных результатов включает только времена запусков на задачах, для которых был получен правильный результат. Отметим, что в соревнованиях учитывалось только время на корректных результатах, так как при некорректных результатах снимаются или не начисляются баллы. С точки зрения пользователя инструмента важно знать, сколько в среднем составит время ожидания ответа инструмента верификации, и в этом случае необходимо учитывать время всех запусков, в том числе неуспешных.

Из таблицы 3 можно видеть, что количество корректных результатов для CEGAR подходов значительно выше чем для BMC. Это объясняется лучшей масштабируемостью на драйверах, которые являются большими программами (несколько тысяч строк кода), относительно программ, представленных в других категориях. В тоже время видно, что инструменты, использующие точную семантику для целых чисел и битовых операций, например, Satabs и Wolverine, требуют больше времени. Ещё одной особенностью, выявленной в результате соревнований стало то, что некоторые инструменты существенно опираются на предположение о доступности определений всех функций, в том числе библиотечных, и что вся используемая память выделяется и освобождается в коде анализируемой программы. Для драйверов устройств, представленных в категории DeviceDrivers64, данное предположение принципиально не выполнено, так как драйвер использует различные структуры данных, инициализируемые в сердцевине ядра, а также использует программный интерфейс сердцевины ядра.

5. Системы тяжеловесной статической верификации драйверов устройств

Для широкомасштабного применения инструментов тяжеловесной статической верификации к драйверам устройств требуется разрешить целый ряд вопросов. При этом часть из них являются специфичными для верификации драйверов устройств. Например, инструменты верификации ожидают, что у анализируемой программы есть точка входа, иными словами, функции *main* или её аналог. У драйверов такая точка входа есть — функция инициализации драйвера. Но в функции инициализации обычно происходит только инициализация структур драйвера и регистрация в ядре ОС функций-обработчиков событий. А основная функциональность драйвера находится в функциях-обработчиках, которые вызываются ядром ОС для обработки событий, таких как обращение к устройству из прикладных приложений, обработка прерываний и т.д. В результате, если инструменты верификации начнут анализ с функции инициализации, то основная функциональность драйвера останется для них недостижимой, то есть непроверенной. Поэтому для проведения верификации драйверов требуется создание модельного окружения драйвера в виде искусственной точки входа, в которой сначала вызывается функция инициализации, а потом выполняются обращения к функциям-обработчикам аналогично тому, как это происходит при реальной работе драйвера. Для проведения единичных экспериментов такое окружение может быть создано вручную, но для широкомасштабного применения инструментов верификации модельное окружение должно генерироваться автоматически.

Вторая потребность заключается в необходимости автоматизировать извлечение информации о составе драйвера и настройках его компиляции из уже имеющихся данных, предназначенных для сборки драйверов. При этом важно учитывать, что с одной стороны у драйверов есть много различных зависимостей от функций из основной части ядра ОС, а с другой — что для эффективного применения статической верификации количество строк кода должно быть не очень большим.

Ещё один вопрос возникает с формулировкой проверяемых свойств, например, таких как корректное использование драйвером интерфейсов ядра ОС. Для постановки задачи инструменту верификации требуется описать, какая ситуация является ошибочной, причём это описание в случае широкомасштабного применения должно находиться вне кода драйверов. Некоторые инструменты верификации предоставляют встроенные возможности для формулировки проверяемых свойств, для других это должно быть реализовано отдельно.

Кроме того, для широкомасштабного применения требуется минимизации участия человека в настройке инструментов и максимально удобный интерфейс для их использования, что включает в себя удобство запуска верификации и удобство анализа её результатов.

Для разрешения всех этих вопросов фактически требуется разработка специальной системы верификации драйверов, автоматизирующей применение инструментов статической верификации. Такая система сама по себе является нетривиальным программным продуктом, поэтому желательно, чтобы система верификации поддерживала несколько инструментов верификации, в особенности с учетом динамичного развития в этой области.

Рассмотрим основные возможности систем верификации драйверов, разработанных в рамках существующих проектов по верификации драйверов с использованием тяжеловесных методов статического анализа кода.

Наиболее долгую историю имеет система верификации Microsoft SDV [56]. Данная система предоставляет широкие возможности по верификации с помощью статического анализа драйверов ОС Microsoft Windows, как входящих в состав самой ОС, так и поступающих от сторонних разработчиков в существующей программе сертификации драйверов. К особенностям системы относятся следующие:

- Для создания окружения от пользователя требуется вручную аннотировать исходный код драйверов, указав в нем роли каждой из функций обработчиков, после этого окружение генерируется автоматически.
- Проверяемые правила корректности формализуются с помощью языка SLIC [57], в котором связь проверок с исходным кодом драйвера задается с помощью аспектно-ориентированных конструкций, перехватывающих вызовы функций ядра. В настоящее время уже выделен набор из более чем 210 правил, а в исследовательской версии была реализована возможность добавления новых правил.
- Известно, что в собственных исследовательских целях разработчики Microsoft SDV могут подключать два инструмента статической верификации: Slam и Yogi [49]. Подключение инструментов верификации сторонними разработчиками не предусмотрено.
- Имеется возможность просмотреть сводную статистику по всем проверяемым правилам для анализируемого драйвера. Для найденных ошибок можно просмотреть представленный удобным и наглядным образом путь из точки входа в ошибочное состояние (трассу ошибки), который связан с исходным кодом драйверов и ядра ОС.

Существует также несколько систем, использующих тяжеловесные методы статического анализа кода для верификации драйверов ядра ОС Linux: DDVerify [58], разработка университета Карнеги-Меллон (США), Avinux [59], разработка университета города Тюбинген (Германия) и система верификации Linux Driver Verification [30], разработанная в Институте системного программирования РАН (Россия).

Особенности системы DDVerify таковы:

- Информацию о составе и настройках сборки драйверов DDVerify получает, используя собственные файлы сборки без учёта файлов сборки ядра. Поэтому система не учитывает специфику компиляции ядра в полной мере.
- Для создания окружения используется модель ядра для некоторых типов драйверов. Разработчиками были написаны модели ядра для трёх типов драйверов, а всего их несколько десятков.
- Правила корректности задаются как часть модели ядра. Код ограничений, накладываемых правилом, задается вместе с кодом, описывающим семантику функции. Поэтому для использования DDVerify требуется существенно изменять заголовочные файлы ядра, что осложняет поддержку применимости системы для различных версий ядра ОС Linux.
- Система позволяет подключать два инструмента статической верификации: Cbmc [50] и Satabs [35].
- Для анализа результатов анализа имеется специальный плагин для интегрированной среды разработки Eclipse. В частности, имеется возможность анализа трассы ошибки с одновременным просмотром соответствующего исходного кода драйверов и ядра ОС Linux, что существенно облегчает анализ трасс ошибок.

Система Avinux, которая также предназначена для верификации драйверов ядра ОС Linux, обладает следующими характерными особенностями:

- Получение исходного кода драйвера для последующей верификации происходит на основе встраивания в процесс сборки ядра путём модификации файлов, описывающих сборку. Однако, Avinux предоставляет возможность автоматической работы только с единичными препроцессированными файлами. Поэтому, например, верификация драйверов, состоящих из нескольких файлов возможна только вручную, так как информация о зависимостях теряется.
- Для создания окружения драйвера требуется вручную написать функцию *main*, моделирующую взаимодействие драйвера с ядром операционной системы и, кроме того, выбрать файлы, входящие в драйвер. На основе этого код инициализации параметров генерируется автоматически [60].
- Для задания правил используются аспектно-ориентированные конструкции, похожие на конструкции SLIC.
- Система интегрирована с единственным инструментом статической верификации Cbmc [50].
- Для упрощения запуска Avinux предоставляет плагин к среде Eclipse, но средства для визуализации трассы ошибок отсутствуют.

Система верификации Linux Driver Verification [30]:

- Система интегрирована с процессом сборки ядра, поэтому вся необходимая информация о составе и настройках сборки драйверов извлекается автоматически.
- Генерация окружения осуществляется полностью автоматически на основе иерархии шаблонов, покрывающей все типы драйверов.
- Система позволяет добавлять новые правила корректности с помощью аспектно-ориентированного расширения языка программирования Си.
- Система поддерживает встраивание внешних инструментов статической верификации с помощью написания адаптеров.
- Для удобного анализа трасс ошибок и сравнительного анализа результатов система предоставляет специальные компоненты с Веб-интерфейсом.

6. Заключение

В обзоре рассмотрены инструменты тяжеловесной статической верификации для программ на языке Си, которые позволяют проверять выполнимость определённых свойств программ или проверять отсутствие в коде программы определённого класса ошибок. Основными подходами, реализуемыми в современных инструментах, являются CEGAR и BMC. Подходы CEGAR обладают большей масштабируемостью по сравнению с BMC, что наглядно продемонстрировали результаты соревнований SV-COMP-2012 в категории DeviceDrivers64. Точность работы инструментов во многом определяется возможностями используемых решателей. Чем точнее происходит моделирование семантики языка, тем больше ресурсов требуется инструменту верификации для анализа, но тем меньше возможностей для появления ложных сообщений об ошибках. Инструменты верификации представляют различные гарантии отсутствия ошибок. Если инструменты на основе CEGAR показывают отсутствие ошибок с точностью до поддерживаемых конструкций языка Си, то инструменты на основе BMC лишь проверяют отсутствие ошибок вплоть до задаваемого в качестве параметра ограничения на количество итераций циклов.

Наибольшие проблемы в инструментах на основе CEGAR вызывает анализ указателей и поддержка операций с указателями. Потенциально перспективным направлением является сочетание различных методов, позволяющих увеличить точность и скорость верификации. Для обеспечения масштабируемости перспективным направлением является разработка модульных подходов к верификации.

Широкомасштабное применение инструментов тяжеловесной статической верификации для драйверов устройств требует разработки специализированных систем верификации драйверов. Рассмотренные системы верификации драйверов демонстрируют разнообразие подходов к построению таких систем, но в то же время демонстрируют разрешимость этой задачи.

Литература

[1] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In Integrated Formal Methods (IFM), volume 2999 of LNCS. Springer, 2004.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5), 752–794, 2003.

[3] S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In Proc. CAV, LNCS 1254, pages 72–83. Springer, 1997.

[4] T. Ball and S. K. Rajamani. The Slam project. Debugging system software via static analysis. In Proc. POPL, pages 1–3. ACM, 2002.

[5] T. Ball, E. Bounimova, V. Levin, and L. De Moura. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. 2010.

[6] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R. The Software Model Checker Blast. Int. J. Softw. Tools Technol. Transfer 9 (5-6), 505–525, 2007.

[7] D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, R. Sebastiani. Software Model Checking via Large-Block Encoding. In Proc. FMCAD, pp. 25–32. IEEE, 2009.

[8] K.L. McMillan. Lazy abstraction with interpolants. In CAV. Volume 4144 of LNCS. Springer, pp. 123–136, 2006.

[9] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL), ACM Press, pp. 58-70, 2002.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. POPL '04 Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM New York, NY, USA, 2004.

[11] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 293–308. Springer, Heidelberg, 1995.

[12] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In Proc. POPL, pp. 38–48. ACM Press, New York, 1998.

[13] B. Steffen. Data-flow analysis as model checking. In Proc. TACS, pp. 346–365, 1991.

[14] S. Gulwani and A. Tiwari. Combining abstract interpreters. In Proc. PLDI, pp. 376–386. ACM Press, New York, 2006.

[15] S. Lerner, D. Grove, and C. Chambers. Composing data-flow analyses and transformations. In Proc. POPL, pp. 270–282. ACM Press, New York, 2002.

[16] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007), pp. 504-518, 2007.

[17] J. Fischer, R. Jhala, and R. Majumdar. Joining data flow with predicates. In: Proc. ESEC/FSE, pp. 227–236. ACM Press, New York, 2005.

[18] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. ASE '08 Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society Washington, DC, USA, 2008.

[19] D. Kroening, O. Strichman. Decision Procedures. An Algorithmic Point of View. Springer, 2008.

[20] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. CAV'05, 2005.
[21] W. Craig. Linear reasoning. J. Symbolic Logic 22, 250–268, 1957.
[22] R. Jhala and R. Majumdar. Path Slicing. Proceedings of the 27th Annual ACM Conference on Programming Language Design and Implementation, 2005. (PLDI), 2005.
[23] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg, 1999.
[24] M. Sagiv, T. Reps, R. Wilhelm. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (TOPLAS), 24 (3), pp. 217–298, 2002.
[25] N.D. Jones, S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. POPL '82 Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1982.
[26] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy Shape Analysis. In T. Ball and R.B. Jones, editors, Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006), LNCS 4144, pages 532-546, Springer-Verlag, Berlin, 2006.
[27] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. POPL '10 Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM New York, NY, USA, 2010.
[28] D. Wonisch. Block Abstraction Memoization for CPAchecker. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 532–534. Springer, Heidelberg, 2012.
[29] P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with Blast 2.7. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg, 2012.
[30] В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, том 20, стр. 163-187, 2011.
[31] D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg, 2011.
[32] S. Löwe, P. Wendler. CPAchecker with Adjustable Predicate Analysis. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 528–530. Springer, Heidelberg, 2012.
[33] S. Grebenshchikov, A. Gupta, N.P. Lopes, C. Popeea, A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg, 2012.
[34] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. PLDI, 2006.
[35] G. Basler, A. Donaldson, A. Kaiser, D. Kröning, M. Tautschnig, T. Wahl. SATabs: A Bit-Precise Verifier for C Programs. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 552–555. Springer, Heidelberg, 2012.
[36] R. Jhala and R. Majumdar. Software model checking. ACM Computing Surveys, 2009.
[37] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[38] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In SPIN Workshop on Model Checking of Software, volume 3639 of LNCS. Springer, 2005.
[39] G. Basler, D. Kroening, and G. Weissenbacher. SAT-based summarisation for Boolean programs. In SPIN Workshop on Model Checking of Software, volume 4595 of LNCS, 2007.
[40] A. Donaldson, A. Kaiser, D. Kroening, T. Wahl. Symmetry-aware predicate abstraction for shared variable concurrent programs. Proceedings of the 23rd international conference on Computer aided verification, p.356-371, July 14-20, 2011.
[41] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In Model Checking and Software Verification (SPIN), volume 1885 of LNCS, pages 113–130. Springer, 2000.
[42] J. R. Buchi. Regular canonical systems. Archive for Mathematical Logic, 6(3-4):91, April 1964.
[43] T. Ball, and S. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09, Microsoft Research, 2002.
[44] T. Ball, B. Cook, S. Das, S. K. Rajamani. Refining Approximations in Software Predicate Abstraction. TACAS 2004, 2004.
[45] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In TACAS 02: Tools and Algorithms for Construction and Analysis of Systems. Lecture Notes in Computer Science 2280. Springer-Verlag, 158–172, 2002.
[46] T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. FMCAD, 2010.
[47] Weissenbacher, G., Kröning, D., Malik, S.: Wolverine: Battling Bugs with Interpolants. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 556–558. Springer, Heidelberg (2012)
[48] A. V. Nori, S. K. Rajamani, S. Tetali, A. V. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In TACAS '09: Tools and Algorithms for the Construction and Analysis of Systems, March, 2009.
[49] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. 2988/168-176, 2004.
[50] L. Cordeiro, J. Morse, D. Nicole, B. Fischer. Context-Bounded Model Checking with Esbmc. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 535–538. Springer, Heidelberg, 2012.
[51] C. Sinz, F. Merz, S. Falke. Llbmc: A Bounded Model Checker for Llvms Intermediate Representation. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 542–544. Springer, Heidelberg, 2012.
[52] A. Holzer, D. Kröning, C. Schallhart, M. Tautschnig, H. Veith. Proving Reachability Using FShell. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 538–541. Springer, Heidelberg, 2012.
[53] K. Dudka, P. Müller, P. Peringer, T. Vojnar. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg, 2012.
[54] D. Beyer. Competition on Software Verification. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg, 2012.
[55] T. Ball, E. Bounimova, V. Levin et al. The static driver verifier research platform. Computer Aided Verification, pp. 119–122, 2010.

[56] T. Ball, S.K. Rajamani. SLIC: A specification language for interface checking. Tech. rep. Microsoft Research, 2001.
[57] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent linux device drivers. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07), pages 501-504, 2007.
[58] H. Post, W. Küchlin. Integration of static analysis for linux device driver verification. The 6th Intl. Conf. on Integrated Formal Methods, IFM 2007, 2007.
[59] H. Post, W. Kuchlin. Automatic data environment construction for static device drivers analysis. Proceedings of the 2006 conference on Specification and verification of component-based systems (SAVCBS ’06), ACM, pp. 89–92, 2006.

Static Verification Tools for C Programs and Linux Device Drivers: A Survey

Khoroshilov A.V., Mandrykin M.U., Mutilin V.S., Novikov E.M.
khoroshilov@ispras.ru, mandrykin@ispras.ru, mutilin@ispras.ru,
novikov@ispras.ru
0ISP RAS, Moscow, Russia

Abstract. The survey considers methods and techniques used in modern static verification tools for C programs. It describes two main approaches Counter Example Guided Abstraction Refinement (CEGAR) and Bounded Model Checking (BMC) and techniques used to efficiently implement them such as Predicate Abstraction, Abstract Reachability Tree, Lazy Abstraction, Configurable Program Analysis, Explicit Analysis, Interpolation, and Shape Analysis. The paper also discusses current capabilities of the tools such as supported C programming language constructs, scalability, properties being checked, and trustworthiness of analysis results.

We provide description of such static verification tools, as BLAST, CPAchecker, HSF(C), SATABS, SLAM, WOLVERINE, YOGI, CBMC, ESBMC, LLBMC, FSHELL and PREDATOR. This description shows techniques implemented in these tools and their current capabilities. The paper presents results of the 1st International Competition on Software Verification in category DeviceDrivers64 which contains verification tasks based on device drivers from Linux kernel 3.0.

Specifics of device drivers verification are discussed and existing driver verification systems are described including Microsoft SDV for Windows operating system and DDVerify, Avinux and Linux Driver Verification for Linux.

The paper concludes that BMC-based tools work well for programs of limited size and control flow complexity. Regarding verification of device drivers that means these tools are able to quickly find violations of properties being checked if paths to these violations are quite short, but they mostly fail to prove correctness and to find complicated bugs. CEGAR-based tools demonstrate better scalability, while they have problems with handling address arithmetic and complex memory structures. For future improvements in static verification of C programs and Linux device drivers we propose composition of various techniques and modularization of analysis.

Keywords: verification, static analysis, device drivers, Linux operating system.

References

[1]. Ball T., Cook B., Levin V., Rajamani S. K. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In Proc. Integrated Formal Methods (IFM), LNCS, vol. 2999, pp. 1-20, 2004. doi: 10.1007/978-3-540-24756-2_1

[2]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement. In Proc. Computer Aided Verification (CAV), LNCS, vol. 1855, pp 154-169, 2000. doi: 10.1007/10722167_15

[3]. Graf S., Saidi H. Construction of Abstract State Graphs with PVS. In Proc. Computer Aided Verification (CAV), LNCS, vol. 1254, pp. 72-83, 1997. doi: 10.1007/3-540-63166-6_10

[4]. Ball T., Rajamani S. K. The Slam project. Debugging system software via static analysis. In Proc. Symposium on Principles of Programming Languages (POPL), pp. 1–3, 2002.
[5]. Ball T., Bounimova E., Levin V., De Moura L. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. Microsoft Research Technical Report MSR-TR-2010-24, 2010.
[6]. Beyer D., Henzinger T., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z
[7]. Beyer D., Cimatti A., Griggio A., Keremoglu M.E., Sebastiani R. Software Model Checking via Large-Block Encoding. In Proc. Formal Methods in Computer-Aided Design (FMCAD), pp. 25–32, 2009. doi: 10.1109/FMCAD.2009.5351147
[8]. McMillan K.L. Lazy abstraction with interpolants. In Proc. Computer Aided Verification (CAV), LNCS, vol. 4144, pp. 123–136, 2006. doi: 10.1007/11817963_14
[9]. Henzinger T.A., Jhala, R., Majumdar, R., Sutre, G. Lazy abstraction. In Proc. 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), Pages 58-70, 2002. doi: 10.1145/503272
[10]. Henzinger T.A., Jhala R., Majumdar R., McMillan K.L. Abstractions from proofs. In Proc. 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 232-244, 2004. doi: 10.1145/964001.964021
[11]. Cousot P., Cousot R. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In Proc. Computer Aided Verification (CAV), LNCS, vol. 939, pp. 293–308, 1995. doi: 10.1007/3-540-60045-0_58
[12]. Schmidt D.A. Data-flow analysis is model checking of abstract interpretations. In Proc. 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 38–48, 1998. doi: 10.1145/268946.268950
[13]. Steffen B. Data-flow analysis as model checking. In Proc. Theoretical Aspects of Computer Software (TACS), LNCS, vol. 526, pp. 346–365, 1991. doi: 10.1007/3-540-54415-1_54
[14]. Gulwani S., Tiwari A. Combining abstract interpreters. In Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 376–386, 2006. doi: 10.1145/1133255.1134026
[15]. Lerner S., Grove D., Chambers C. Composing data-flow analyses and transformations. In Proc. ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 270–282, 2002.
[16]. Beyer D., Henzinger T.A., Théoduloz G. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In Proc. International Conference on Computer Aided Verification (CAV), LNCS, vol. 4590, pp. 504-518, 2007. doi: 10.1007/978-3-540-73368-3_51
[17]. Fischer J., Jhala R., Majumdar R. Joining data flow with predicates. In Proc. 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE), pp. 227–236, 2005. doi: 10.1145/1081706.1081742
[18]. Beyer D., Henzinger T. A., Theoduloz G. Program Analysis with Dynamic Precision Adjustment. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 29-38, 2008. doi: 10.1109/ASE.2008.13

[19]. Kroening D., Strichman O. Decision Procedures: An Algorithmic Point of View. Journal of Automated Reasoning, vol. 51, issue 4, pp. 453-456, 2008. doi: 10.1007/s10817-013-9295-4
[20]. Bozzano M., Bruttomesso R., Cimatti A., Junttila T., Ranise S., Van Rossum P., Sebastiani R. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In Proc. Computer Aided Verification (CAV), LNCS, vol. 3576, pp. 335-349, 2005. doi: 10.1007/11513988_34
[21]. Craig W. Linear reasoning. Symbolic Logic, vol. 22, pp. 250–268, 1957.
[22]. Jhala R., Majumdar R. Path Slicing. In Proc. 27th Annual ACM Conference on Programming Language Design and Implementation (PLDI), pp. 38-47, 2005. doi: 10.1145/1065010.1065016
[23]. Biere A., Cimatti A., Clarke E., Zhu Y. Symbolic Model Checking without BDDs. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 1579, pp. 193–207, 1999. doi: 10.1007/3-540-49059-0_14
[24]. Sagiv M., Reps T., Wilhelm R. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 24, issue 3, pp. 217–298, 2002. doi: 10.1145/514188.514190
[25]. Jones N.D., Muchnick S.S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In Proc. 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 66-74, 1982. doi: 10.1145/582153.582161
[26]. Beyer D., Henzinger T. A., Théoduloz G. Lazy Shape Analysis. In Proc. Computer Aided Verification (CAV), LNCS, vol. 4144, pp. 532-546, 2005. doi: 10.1007/11817963_48
[27]. Godefroid P., Nori A.V., Rajamani S.K., Tetali S.D. Compositional may-must program analysis: unleashing the power of alternation. In Proc. 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 43-56, 2010. doi: 10.1145/1706299.1706307
[28]. Wonisch D. Block Abstraction Memoization for CPAchecker. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 532–534, 2012. doi: 10.1007/978-3-642-28756-5_41
[29]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39
[30]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
[31]. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16
[32]. Löwe S., Wendler P. CPAchecker with Adjustable Predicate Analysis. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 528–530, 2012. doi: 10.1007/978-3-642-28756-5_40
[33]. Grebenshchikov S., Gupta A., Lopes N.P., Popeea C., Rybalchenko A.. HSF(C): A Software Verifier Based on Horn Clauses. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 549–551, 2012. doi: 10.1007/978-3-642-28756-5_46

[34]. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 415–426, 2006. doi: 10.1145/1133981.1134029
[35]. Basler G., Donaldson A., Kaiser A., Kröning D., Tautschnig M., Wahl T.. SATabs: A Bit-Precise Verifier for C Programs. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 552–555, 2012. doi: 10.1007/978-3-642-28756-5_47
[36]. Jhala R., Majumdar R. Software model checking. ACM Computing Surveys (CSUR), vol. 41, issue 4, article 21, 2009. doi: 10.1145/1592434.1592438
[37]. McMillan K.L. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.
[38]. Cook B., Kroening D., Sharygina N. Symbolic model checking for asynchronous Boolean programs. In Proc. Model Checking Software, LNCS, vol. 3639, pp. 75-90, 2005. doi: 10.1007/11537328_9
[39]. Basler G., Kroening D., Weissenbacher G. SAT-based summarisation for Boolean programs. In Proc. Model Checking Software, LNCS, vol. 4597, pp. 131-148, 2007. doi: 10.1007/978-3-540-73370-6_10
[40]. Donaldson A., Kaiser A., Kroening D., Wahl T. Symmetry-aware predicate abstraction for shared variable concurrent programs. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 356-371, 2011. 10.1007/978-3-642-22110-1_28
[41]. Ball T., Rajamani S.K. Bebop: A symbolic model checker for Boolean programs. In Proc. Model Checking Software, LNCS, vol. 1885, pp. 113-130, 2000. doi: 10.1007/10722468_7
[42]. Buchi J. Regular canonical systems. Archive for Mathematical Logic, vol. 6, issue 3-4, pp. 91-111, 1964. doi: 10.1007/BF01969548
[43]. Ball T., Rajamani S. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, 2002.
[44]. Ball T., Cook B., Das S., Rajamani S.K. Refining Approximations in Software Predicate Abstraction. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2988, pp. 388-403, 2004. doi: 10.1007/978-3-540-24730-2_30
[45]. Ball T., Podelski A., Rajamani S.K. Relative completeness of abstraction refinement for software model checking. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2280, pp. 158-172, 2002. doi: 10.1007/3-540-46002-0_12
[46]. Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static Driver Verification with Under 4% False Alarms. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 35-42, 2010.
[47]. Weissenbacher G., Kröning D., Malik S. Wolverine: Battling Bugs with Interpolants. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 556–558, 2012. doi: 10.1007/978-3-642-28756-5_48
[48]. Nori A.V., Rajamani S.K., Tetali S., Thakur A.V. The Yogi Project: Software Property Checking via Static Analysis and Testing. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 5505, pp. 178-181, 2009. doi: 10.1007/978-3-642-00768-2_17
[49]. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2988, pp. 168-176, 2004. doi: 10.1007/978-3-540-24730-2_15

[50]. Cordeiro L., Morse J., Nicole D., Fischer B. Context-Bounded Model Checking with ESBMC 1.17. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 535–538, 2012. doi: 10.1007/978-3-642-28756-5_42
[51]. Sinz C., Merz F., Falke S. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 542–544, 2012. doi: 10.1007/978-3-642-28756-5_44
[52]. Holzer A., Kröning D., Schallhart C., Tautschnig M., Veith H. Proving Reachability Using FShell. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 538-541, 2012. doi: 10.1007/978-3-642-28756-5_43
[53]. Dudka K., Müller P., Peringer P., Vojnar T. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 545-548, 2012. doi: 10.1007/978-3-642-28756-5_45
[54]. Beyer D. Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[55]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The static driver verifier research platform. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6174, pp. 119-122, 2010. 10.1007/978-3-642-14295-6_11.
[56]. Ball T., Rajamani S.K. SLIC: A specification language for interface checking. Technical report MSR-TR-2001-21, Microsoft Research, 2001.
[57]. Witkowski T., Blanc N., Kroening D., Weissenbacher G. Model checking concurrent linux device drivers. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 501-504, 2007. doi: 10.1145/1321631.1321719.
[58]. Post H., Küchlin W. Integrated static analysis for Linux device driver verification. In Proc. Integrated Formal Methods (IFM), LNCS, vol. 4591, pp. 518-537, 2007. doi: 10.1007/978-3-540-73210-5_27
[59]. Post H., Kuchlin W. Automatic data environment construction for static device drivers analysis. In Proc. Conference on Specification and Verification of Component-Based Systems (SAVCBS), pp. 89–92, 2006. doi: 10.1145/1181195.1181215

Интерполяция формул с кванторами в CSIsat на основе инстанцирования[1]

В. С. Мутилин <mutilin@ispras.ru>,
М. У. Мандрыкин <mandrykin@ispras.ru>

Аннотация. В статье рассказывается о том, как на основе инструмента CSIsat реализована интерполяция Крейга для формул с кванторами, заданных в рамках комбинации теорий вещественной линейной арифметики и неинтерпретируемых функций. Предложен способ реализации интерполирования таких формул с помощью инстанцирования подкванторных выражений с использованием внешнего SMT-решателя CVC3. В статье описываются изменения, которые были внесены в интерполятор и решатель для реализации поддержки кванторной интерполяции. Также приводятся результаты тестирования модифицированного интерполятора как на задачах, полученных из набора SMTLIB, так и на специально сгенерированных для этих целей задачах.

Ключевые слова: интерполяция, интерполянт Крейга, кванторы, инстанцирование, решатель, аксиомы.

1. Введение

В настоящее время в области проверки моделей программ одним из наиболее распространённых и успешных подходов является предикатная абстракция. Это в значительной степени объясняется успехами инструмента статической верификации SLAM2 [1], использующего этот подход, и применяемого в рамках набора инструментов WDDK(Windows Driver Developer's Kit) для верификации драйверов ОС Windows. Предикатная абстракция программы строится на основе логических предикатов, набор возможных значений которых составляет абстрактный домен, разделяя всё множество возможных состояний программы на подмножества с одинаковыми значениями выбранных предикатов[2]. Отличительной чертой этого подхода является то, что выбор предикатов для абстракции обычно либо полностью определяется рассматриваемой программой, либо очень сильно зависит от неё. Поэтому основной проблемой при использовании предикатной абстракции является

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы (контракт N 11.519.11.4006).

автоматическое определение подходящего набора предикатов, подходящих для верификации данной конкретной программы. Наиболее часто с этой целью используют подход CEGAR [3] (Counter-Example Guided Abstraction Refinement – уточнение абстракции по контрпримеру). Он основывается на том, что абстракция программы строится итеративно, причём, если проверка модели с некоторой абстракцией даёт ложный (реально не существующий) контрпример (это значит, что абстракция оказалась недостаточно точной), то этот контрпример используется для получения новых предикатов и построения уточнённой абстракции программы.

Среди современных инструментов статической верификации программ, использующих предикатную абстракцию и реализующих подход CEGAR, на практике наиболее распространены инструменты SLAM2[1], BLAST[4,5] и CPAchecker[6]. В этих инструментах уточнение абстракции по контрпримеру осуществляется на основе двух различных и в то же время схожих подходов. Во всех этих инструментах, так или иначе, строится некоторая логическая формула, соответствующая рассматриваемому контрпримеру. В SLAM2 для этого используется наислабейшее предусловие для последовательности операторов, составляющих контрпример. В BLAST и CPAchecker используется построенная на основе SSA-представления формула пути. Так как контрпример является ложным, то достаточно точно построенная соответствующая ему логическая формула являться невыполнимой. Этот факт используется инструментами для получения из этой формулы новых предикатов с целью уточнения абстракции. При этом верификаторы прибегают к помощи специальных инструментов. В SLAM2 используется эвристика получения новых предикатов из минимальной невыполнимой подформулы («ядра невыполнимости» или unsatisfiable core), которую может выдавать инструмент проверки выполнимости логических формул (SMT-решатель) Z3[7], а также другие синтаксические методы получения новых предикатов[8]. BLAST и CPAchecker получают новые предикаты локально, то есть отдельно для нескольких выбранных точек программы, из интерполянтов Крейга для частей формулы пути до и после каждой выбранной такой точки. Для поиска этих интерполянтов оба инструмента используют специальные инструменты – интерполирующие решатели. BLAST может использовать FOCI[9] или CSIsat[10], а CPAchecker – MathSAT[11] или тот же CSIsat.

Одним из преимуществ текущей версии инструмента SLAM перед BLAST и CPAchecker является его хорошая точность (определяемая по числу ложных срабатываний) при анализе программ, существенно использующих указатели, в том числе на динамически выделенную память. При этом в инструменте не используется какой-либо специальный вид анализа структур в динамической памяти (вроде Shape-анализа[12]) или специальная логика для представления объектов кучи (например, separation logic [13]). Вместо этого используется некоторая упрощенная модель памяти на основе размещений. Эта модель предложена в статье [14], в которой она используется для построения предикатов с указателями, значения которых эффективно вычисляются

современным SMT-решателем (в данном случае Z3[7]). Для вычисления значений предикатов с указателями авторы этой статьи предлагают использовать специальный набор аксиом, который определяет логическую модель памяти. В статье также предлагается алгоритм представления предикатов с помощью неинтерпретируемых функций от целых чисел для передачи соответствующих задач SMT-решателю.

Возможность применения похожей логической модели памяти было предложено исследовать в верификаторах BLAST[4,5] и CPAchecker[6]. Как уже было сказано, получение новых предикатов для построения уточненной абстракции в них реализовано на основе интерполирования некоторых частей формулы невыполнимого пути [15].

Конъюнкция упорядоченной пары формул, соответствующих двум частям невыполнимого пути, невыполнима. Поэтому для этой пары формул существует интерполянт Крейга, то есть формула, логически следующая из первой формулы пары, невыполнимая в конъюнкции со второй формулой и содержащая лишь общие для обеих формул неинтерпретируемые символы. Предикаты, входящие в этот интерполянт, используются в BLAST и CPAchecker при построении уточненной абстракции программы. Для получения этого интерполянта используется специальный инструмент – интерполирующий решатель (также «интерполирующая процедура» или просто «интерполятор»). В BLAST и CPAchecker используются интерполирующие решатели (CSIsat[10] и MathSAT[11]) для бескванторных формул, заданных в рамках комбинации теорий линейной вещественной арифметики и неинтерпретируемых функций (LA+EUF). Для таких формул эти инструменты выдают интерполянт, также не содержащий кванторов. При применении модели памяти с набором аксиом возникает необходимость интерполировать также формулы, содержащие кванторы. При этом для таких формул становится допустимым получение интерполянта, содержащего кванторы.

Таким образом, если для реализации рассматриваемой модели памяти в инструменте SLAM2 оказалось достаточно SMT-решателя Z3, не поддерживающего интерполяцию, но поддерживающего наборы аксиом с кванторами, то для реализации подобной модели памяти в инструменте BLAST или CPAchecker обязательно требовался именно интерполирующий решатель, поддерживающий кванторы или наборы аксиом. Однако подходящего для этих целей инструмента найти не удалось.

Вместе с тем, в статье[16] предлагается расширение алгоритма интерполяции Макмиллана[9] для интерполирования формул с кванторами на основе инстанцирования. Эти формулы могут быть заданы в рамках произвольных теорий, для которых применим алгоритм Макмиллана. Расширенный алгоритм Макмиллана позволяет достаточно просто искать интерполянты, возможно, содержащие кванторы, для таких формул, при условии, что заранее известен набор экземпляров (инстанцирований) исходных подкванторных

выражений (для кванторов всеобщности), достаточный для доказательства невыполнимости конъюнкции, заданной в задаче поиска интерполянта.

Идея реализации расширенного алгоритма Макмиллана показалась интересной еще и потому, что при уточнении абстракции рассматриваемые инструменты (BLAST и CPAchecker) еще до применения интерполяции делают один или несколько соответствующих запросов к SMT-решателю. При этом большинство современных SMT-решателей реализуют поддержку формул с кванторами через инстанцирование подкванторных выражений на основе различных эвристик. Поэтому если в таком случае решателю удается доказать невыполнимость заданной формулы с кванторами, это означает, что перед запуском интерполятора у решателя уже имеется набор инстанцированных выражений, достаточный для доказательства невыполнимости конъюнкции в задаче интерполирования.

Поскольку реализация расширенного алгоритма Макмиллана с использованием инстанцирований, полученных из SMT-решателя, оказывается достаточно простой, такую реализацию можно было бы использовать для предварительной оценки эффективности различных возможных реализаций новой модели памяти в инструментах верификации, использующих интерполяцию для уточнения абстракции.

Поэтому было предложено реализовать расширенный алгоритм Макмиллана в таком виде на основе какого-либо существующего интерполятора и SMT-решателя и оценить эффективность работы полученного инструмента на специально сгенерированных задачах, которые бы мог подавать на вход модифицированному интерполятору верификатор при реализации в нем новой модели памяти.

2. Выбранный подход

Для применения расширенного алгоритма Макмиллана в том виде, в котором он описан в статье[17], нужно иметь дерево резолютивного вывода для доказательства невыполнимости исходной конъюнкции формул, в котором бы уже использовались необходимые подстановки подкванторных выражений и были заранее вычислены так называемые частичные интерполянты для листьев этого дерева. Такое дерево можно было получить различными способами. Одним из возможных способов является реализация некоторой эвристики для подстановки термов в подкванторные выражения (например, e-matching[17]) в каком-либо инструменте, реализующем алгоритм Макмиллана. Другой способ – использование для получения нужных подстановок внешнего инструмента, в котором уже хорошо реализованы соответствующие эвристики, например, SMT-решателя. В случае использования внешнего решателя набор подстановок можно извлекать из предъявляемого им доказательства невыполнимости.

Если из этого доказательства извлекать лишь набор необходимых постановок, то невыполнимость входной формулы будет доказываться дважды: сначала в SMT-решателе, поддерживающем кванторы (для получения подстановок), а затем – в интерполирующем решателе, с нужными подстановками и уже без кванторов, – для получения интерполянта. При этом напрямую использовать полученное SMT-решателем доказательство для получения интерполянта в интерполирующем решателе не получится, так как для интерполяции по алгоритму Макмиллана необходимо иметь доказательство невыполнимости в специальной системе правил вывода, которая значительно отличается от используемых современными SMT-решателями. Современные SMT-решатели по своей эффективности значительно превосходят известные нам реализации интерполяторов, работающие по алгоритму Макмиллана (FOCI, CSIsat и несколько экспериментальных реализаций), поэтому при использовании внешнего SMT-решателя накладные расходы на повторное доказательство невыполнимости формулы в интерполяторе оказываются невелики по сравнению со временем его работы после получения от SMT-решателя нужных подстановок. При такой реализации накладными расходами выглядит скорее повторное получение доказательства невыполнимости в самом интерполяторе.

В силу простоты реализации был выбран именно подход с использованием доказательства невыполнимости, выдаваемого внешним SMT-решателем. Для его реализации нужно было выбрать подходящий существующий интерполятор, который бы удовлетворял следующим требованиям:

- Реализовывал бы алгоритм индуктивного построения интерполянта по дереву резолютивного вывода пустого дизъюнкта на основе комбинирования частичных интерполянтов, описанный в статье Макмиллана (McMillan) [9]. Это необходимо, поскольку алгоритм, предложенный в [16] является расширением алгоритма Макмиллана.
- Поддерживал бы интерполяцию для формул в рамках комбинации теорий, которые используются в задачах поиска интерполянта инструментами BLAST и CPAchecker. Это теории линейной целочисленной (LIA) или вещественной (LRA) арифметики и равенства с неинтерпретируемыми функциями (EUF). Их комбинация часто обозначается как LA+EUF.
- Распространялся бы в виде исходного кода под лицензией, разрешающей его свободную модификацию.

Среди существующих интерполирующих решателей только CSIsat [10] удовлетворял всем этим требованиям. Он написан на языке OCaml, при этом части, отвечающие за чтение входной формулы, ее предобработку, и процесс комбинирования частичных интерполянтов в реализации соответствуют различным модулям. Поэтому оказалось достаточно реализовать

соответствующие модифицированные версии этих модулей и использовать их в случае, когда входная задача содержит кванторы.

Для реализации был выбран простой подход, использующий только набор подстановок, и не требующий глубокого анализа доказательства невыполнимости, выданного SMT-решателем. Поэтому основными критериями при выборе этого инструмента стали достаточно высокая эффективность при доказательстве невыполнимости для формул с кванторами и доступность его исходного кода для модификации – с целью упрощения интеграции. Исходя из этих критериев был выбран SMT-решатель CVC3[18]. При этом основным недостатком этого решателя является большая сложность предъявляемых им доказательств, в которых используется большое число правил вывода. На момент выбора SMT-решателя этот недостаток казался несущественным, поскольку для выделения нужных подстановок планировалось использовать достаточно простой анализ доказательства невыполнимости, однако впоследствии оказалось, что в некоторых случаях CVC3 вообще не включает в окончательное доказательство необходимых подстановок, что приводит к аварийному завершению модифицированного интерполятора.

3. Сравнение с другими подходами

Присутствие значительных накладных расходов на повторное доказательство невыполнимости формулы в интерполирующем решателе наталкивает на мысль о другом способе интерполяции. Преобразование дерева доказательства SMT-решателя к системе правил вывода, подходящей для индуктивного построения интерполянта, представляет собой этот другой, принципиально отличный от предложенного в данной статье, подход к интерполяции. Этот подход также кардинально отличается от используемого в CSIsat и описанного в статье [16]. Такой подход описан в статье[19] и, вполне возможно, превосходит выбранный для реализации в рамках данной работы по эффективности. Ведь при его использовании удается избежать существенных накладных расходов на повторное доказательство невыполнимости в интерполяторе CSIsat, на которое обычно уходит большая часть ресурсов, затрачиваемых этим инструментом. В статье[19] этот подход описан для решателя Z3[7]. Его применение для другого наиболее распространенного SMT-решателя, поддерживающего кванторы, CVC3[18] осложняется большим количеством правил вывода, используемым в этом инструменте. Следует отметить, что упомянутая статья была впервые представлена уже после того, как был реализован описываемый в данной статье инструмент.

Далее в статье приводятся детали реализации и результаты оценки эффективности только для модифицированного интерполятора CSIsat, реализующего расширенный алгоритм Макмиллана с использованием внешнего SMT-решателя CVC3.

4. Детали реализации

Модифицированный инструмент CSIsat поддерживает интерполяцию формул с кванторами, заданных в рамках комбинации теорий вещественной линейной арифметики и неинтерпретируемых функций (LA+EUF). Предложенная реализация основана на последней версии CSIsat 1.2 (июль 2008 года) от оригинальных разработчиков. Инструмент использует специально модифицированный инструмент CVC3, который позволяет ему получать необходимые для интерполяции экземпляры подкванторных выражений.

Перечислим основные изменения, внесенные в инструменты CSIsat и CVC3 при реализации выбранного подхода.

4.1. Изменения в интерполяторе CSIsat

- В модифицированной версии инструмента был расширен входной формат для задачи интерполяции, который совместим с использованным в интерполяторе FOCI (этот интерполятор реализует подход, описанный в [9]). В синтаксис описания входной формулы были добавлены обозначения для кванторов всеобщности и существования:

<квантор всеобщности> ::= ‘!’ <список переменных> ‘.’ <формула>
<квантор существования> ::= ‘?’ <список переменных> ‘.’ <формула>
<список переменных> ::= <идентификатор> <список переменных> | <пусто>

При этом список связываемых квантором переменных (<список переменных>) должен быть не пуст.

- Модифицированный инструмент поддерживает интерполяцию только для пар формул. При указании опции -useLemmata в исходной задаче задается три формулы *A, B* и *C*. Формула *C* при этом должна быть конъюнкцией выражений под кванторами всеобщности. При этом интерполянт ищется для пары формул (*A, B∧C)*, но считается, что кванторы из формулы *C* являются аксиомами теории и свободные символы в них – общие для формул *A* и *B*.
- В инструмент были добавлены некоторые предварительные преобразования исходной задачи перед приведением ее в конъюнктивную нормальную форму для дальнейшего решения. Каждая формула в исходной задаче вначале подвергается некоторым преобразованиям. В частности, в ней выделяются подформулы с кванторами. Эти подформулы переводятся в предваренную нормальную форму и сколемизуются[2]. Если после приведения в предваренную нормальную форму и сколемизации хотя бы в одной формуле все еще присутствуют кванторы

[2] Сколемизация – это удаление кванторов существования при помощи введения новых неинтерпретируемых функций и констант.

всеобщности, то применяется модифицированный алгоритм интерполяции (расширенный алгоритм Макмиллана). Для формул без кванторов используется не модифицированный алгоритм.

- Была добавлена поддержка вывода формул в формате SMTLIB[20] и вызов внешнего SMT-решателя для получения необходимых подстановок. В случае интерполирования задачи с кванторами преобразованная конъюнкция $A \wedge B$ передается модифицированному инструменту CVC3 с опцией +dump-insts. При успешном доказательстве невыполнимости CVC3 помимо вердикта unsat выдает также и набор подстановок, достаточный для доказательства невыполнимости конъюнкции. Если же конъюнкция оказывается выполнимой, CSIsat вместо интерполянта выдает вердикт Satisfiable. Так как решатель CVC3 не является полным, возможен также и вердикт Unknown.
- Был реализован алгоритм очищения (*purification*) полученных подстановок от смешанных термов (*mixed terms*) в соответствии с алгоритмом, описанным в[16]. Если конъюнкция оказывается невыполнимой, CSIsat очищает полученные от CVC3 подстановки, при этом строятся вспомогательные хеш-таблицы, содержащие информацию о новых общих символах.
- Был реализован расширенный алгоритм интерполяции Макмиллана для формул с кванторами. К полученной на предыдущих шагах работы конъюнкции вместе с очищенными подстановками применяется этот алгоритм, который в общем случае дает кванторный интерполянт.

4.2. Изменения в решателе CVC3

Рассматриваемая реализация требует от решателя список подстановок подкванторных выражений, достаточных для доказательства невыполнимости исходной формулы. Для того, чтобы получить такой набор подстановок, можно использовать генерируемое решателем доказательство невыполнимости, в котором в явном виде присутствуют правила вывода, соответствующие инстанцированию подкванторных выражений.

В формате доказательств CVC3 для этого используются четыре правила вывода:

$$\frac{T \vdash \forall \bar{x}.\, e(\bar{x})}{T \vdash e(\bar{t})}\, universal_elimination1$$

$$\frac{T \vdash \forall \bar{x}.\, e(\bar{x})}{T \vdash \psi \Rightarrow e(\bar{t})}\, universal_elimination2, universal_elimination3$$

$$\frac{T \vdash \forall \overline{x_1 x_2}.\, e(\overline{x_1}, \overline{x_2})}{T \vdash \psi \Rightarrow \forall \overline{x_2}.\, e(\bar{t}, \overline{x_2})}\, partial_universal_instantiation$$

Здесь $\bar{x} = (x_1, x_2, \ldots, x_n)$ $\bar{\mathrm{x}} = (\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_\mathrm{n})$— вектор связанных переменных, e – выражение, в которое переменные $x_1, \ldots, x_n \mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n}$ входят свободно, $\bar{t} = (t_1, \ldots, t_n) \bar{\mathrm{t}} = (\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n})$ – вектор термов для подстановки, все переменные в которых свободно входят в e. Для правила *universal_elimination*$1 \mathrm{x}_\mathrm{i}$ x_i и t_i t_i должны иметь одинаковые типы для $i = \overline{1, n} \mathrm{i} = \overline{1, \mathrm{n}}$, а для остальных правил – соответственно одинаковые базовые типы, в таких случаях ψ ψ – предикат, накладывающий ограничения подтипа на подставляемые термы.

Однако, как оказалось на практике, CVC3 не всегда включает данные правила вывода в доказательство невыполнимости для формул с кванторами. Иногда инстанцирование кванторов в нем заменяется упрощенным правилом вывода вида:

$$\frac{}{T \vdash e(\bar{t})} assump$$

Поэтому для извлечения необходимых подстановок была реализована эвристика, рассматривающая некоторые части еще не окончательно сформированного доказательства на предмет присутствия в них инстанцирований. Также была реализована эвристика упрощения выражений, получившихся в результате подстановки. Эти эвристики включаются опциями +extr-insts и +simpl-insts соответственно. Для включения выдачи экземпляров аксиом вместе с вердиктом unsat используется опция +dump-insts.

5. Результаты

5.1. Результаты на наборе из архива SMT-LIB

Реализованный инструмент был протестирован на двух различных наборах задач.

Первый набор задач был получен из архива SMT-LIB[21] путем случайного разделения невыполнимых формул, заданных в рамках логик AUFLIA и AUFLIRA на две части (AUFLIA – Linear Integer Arithmetic with Uninterpreted Functions and Arrays – линейная целочисленная арифметика с неинтерпретируемыми функциями и массивами, AUFLIRA – Arrays, Uninterpreted Functions, and Linear Arithmetic – массивы, неинтерпретируемые функции и линейная арифметика, в том числе целочисленная и вещественная). Полученные задачи интерполяции были переведены во входной формат нового инструмента. При этом целочисленные типы переменных заменялись вещественными, а операции с массивами представлялись неинтерпретируемыми функциями с соответствующим набором аксиом. Некоторые задачи при этом могли переставать быть корректными, а другие могли давать вырожденные интерполянты true и false.

Тесты запускались на бинарной оптимизированной версии инструмента с лимитом 5 секунд по времени и 1 Гб по памяти. В таблице 1 приведены результаты для всех формул из категорий AUFLIA и AUFLIRA.

	Результат	Число тестов	%
успешно	Интерполянт с кванторами	167	0,63%
	Интерполянт без кванторов	551	2,09%
	Вырожденный интерполянт true	8259	31,38%
	Вырожденный интерполянт false	8342	31,70%
	Входная конъюнкция выполнима	1	0,00%
не успешно	CVC3 выдал результат Unknown	536	2,04%
	Превышение лимита времени (5 с)	7846	29,81%
	Превышение лимита памяти	71	0,27%
	Выполнимая формула (неверный набор подстановок)	515	1,96%
	Другие ошибки	31	0,12%
	Всего	26319	100,00%

Таблица 1. Результаты тестирования набора задач из архива SMTLIB без эвристик

5.2. Результаты на специально сгенерированном наборе

Второй набор задач был специально сгенерирован как набор возможных запросов верификатора к интерполирующему решателю для получения новых предикатов при реализации в нем модели памяти, похожей на используемую в инструменте SLAM2. В этой модели с размещениями для построения предикатов использовались пять неинтерпретируемых функций:

- *A(l)* – возвращает адрес размещения *l*,
- *L(a)* – возвращает размещение, соответствующее адресу *a*,
- *S(x, f)* – возвращает размещение для элемента составного типа (например, массива или структуры), здесь *x* – размещение первого элемента составного типа, *f* – номер нужного элемента, считая с *0*,
- *B(l)* – возвращает размещение первого элемента составного типа по размещению другого его элемента,
- *O(l)* – возвращает номер элемента составного типа по его размещению

с соответствующим набором аксиом:

1. $\forall x.(x > 0 \Rightarrow A(x) > 0)$
2. $\forall l.L\big(A(l)\big) = l$
3. $\forall a.A\big(L(a)\big) = a$
4. $\forall x.\forall f.S(x,f) > N$

5. $\forall x.\forall f.B\big(S(x,f)\big) = x$
6. $\forall x.\forall f.O\big(S(x,f)\big) = f$

Для обозначения размещений в этой модели использовались строго положительные целые числа. Первая аксиома утверждает, что адрес любого корректного размещения строго положителен. Вторая и третья аксиомы вместе задают функции *A(l)* и *L(a)* как взаимно обратные для всех размещений и их адресов. Четвертая аксиома утверждает, что размещение для элемента составного типа не может совпадать ни с каким базовым размещением (то есть размещением простого типа или размещением составного типа целиком). Базовые размещения соответствуют явно размещаемым объектам в программе, таким как переменные, структуры и массивы. Им всем присваиваются размещения с номерами от 1 до некоторого N. Элементы же составных типов должны иметь размещения с номерами строго больше N, о чем и говорит четвертая аксиома. Пятая и шестая аксиомы задают функции *B(l)* и *O(l)* и одновременно утверждают, что элементам разных составных типов, а также различным элементам одного составного типа соответствуют разные размещения. В противном случае для двух элементов составных типов было бы выполнено:**Ошибка! Ожидалась цифра.**

$$S(x_1,f_1) = S(x_2f_2) \Longrightarrow B\big(S(x_1,f_1)\big) = B\big(S(x_2,f_2)\big), O\big(S(x_1,f_1)\big) = O\big(S(x_2,f_2)\big)$$
$$\Longrightarrow (5,6) \Longrightarrow x_1 = x_2,\ f_1 = f_2$$

Для получения значений размещений использовался специальный набор неинтерпретируемых функции $V_i(l)\mathrm{V_i(l)}$. Каждая такая функция соответствовала своей версии памяти между двумя последовательными ее обновлениями. При этом при генерации тестового набора не делалось никаких оптимизаций обновления памяти, каждому обновлению памяти было поставлено в соответствие выражение вида $\mathrm{V_{i+1}(l_{upd})} = \mathrm{v} \wedge \forall \mathrm{l}.\,(\mathrm{l} \neq \mathrm{l_{upd}} \Rightarrow \mathrm{V_{i+1}(l)} = \mathrm{V_i(l)})V_{i+1}\big(l_{upd}\big) = v \wedge\ \forall l.\,(l \neq l_{upd} \Rightarrow V_{i+1}(l) = V_i(l))$, где $l_{upd}\mathrm{l_{upd}}$ – размещение, значение которого обновляется на *v*.

В тестах с существенным использованием массивов и структур использовался измененный набор аксиом для учета свойства равенства адресов структур или массивов с адресами их первых элементов. В этом наборе аксиомы (4), (5) и (6) заменены следующими аксиомами:

7. $\forall l.A(l) = A(S(l,0))$
8. $\forall x.\forall f.(f \neq 0 \Rightarrow S(x,f) > N)$
9. $\forall x.\forall f.(f \neq 0 \Rightarrow B\big(S(x,f)\big) = x)$
10. $\forall x.\forall f.(f \neq 0\ \Rightarrow O\big(S(x,f)\big) = f)$

Здесь размещение для всего составного объекта совпадает с размещением для его первого элемента, а аксиомы (4), (5) и (6) верны только для всех элементов, кроме первого.

Приведем пример преобразования задачи построения новых предикатов в заданной точке неосуществимого пути выполнения программы в задачу поиска интерполянта для формулы с кванторами. Для примера используем следующий тест из подготовленного набора:

```
s_1->f_1 = 1;
s_1 = s_1->next;
s_1->f_1 = 2;
s_1 = s_1->next;
...
s_1->f_1 = n - 1;
s_1 = s_1->next;
s_1->f_1 = n;
s_2 = s_1;
s_1 = s_1->next;
-----------------------
check (s_2->f_1 != n);
ERROR:
```

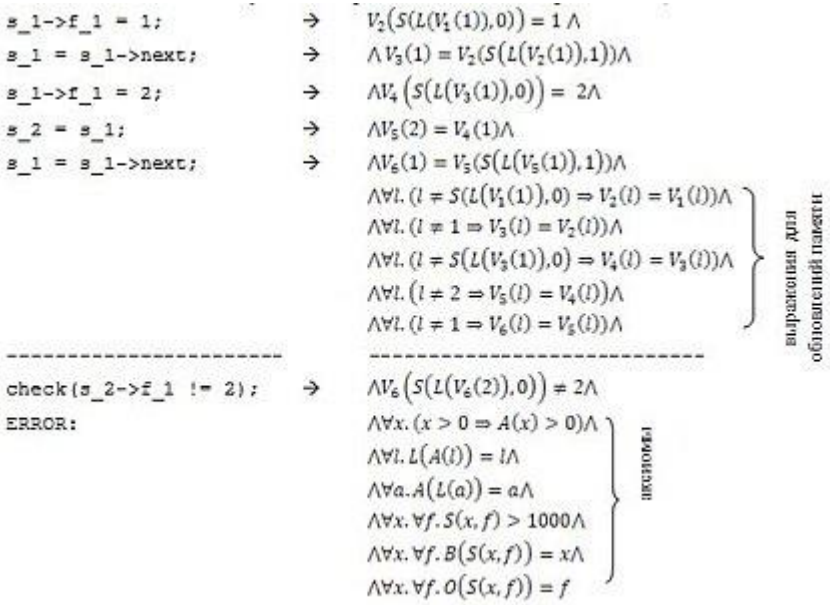
Здесь *n* – статический параметр, задающий размер генерируемого теста. При *n=2* этот тест будет выглядеть следующим образом:

```
s_1->f_1 = 1;
s_1 = s_1->next;
s_1->f_1 = 2;
s_2 = s_1;
s_1 = s_1->next;
----------------------
check(s_2->f_1 != 2);
ERROR:
```

В этом тесте указатель s_1 перемещается по связному списку длиной не менее двух элементов и последовательно присваивает его элементам значения 1 и 2 соответственно. Указателю s_2 присваивается адрес второго элемента списка. Оператор check условно обозначает переход по ветви, в которой выполнено заданное в нем условие. Таким образом, условие перехода на ошибочную метку ERROR – неравенство значения второго элемента списка двум. Оно не может быть выполнено, если верны предположения о корректности работы с памятью в исходной программе на данном пути выполнения. Пунктиром показана точка пути, для которой необходимо построить новый предикат, доказывающий его неосуществимость.

Как верхней (до пунктира), так и нижней части пути соответствует некоторая логическая формула. Для того чтобы верификатор мог исключить из абстракции программы данный невыполнимый путь, истинность построенного предиката должна следовать из формулы для верхней части этого пути, а конъюнкция этого предиката с его нижней частью должна быть невыполнимой. В предикате также должны использоваться только те значения

переменных программы, которые они принимают именно в заданной точке (не до и не после нее). Тогда такой предикат можно будет использовать для построения абстракции верхней части пути программы в указанной точке, в том числе, возможно, и при выполнении по какому-либо другому пути, проходящему через нее. Если в таком случае полученный предикат вновь окажется выполнен, он позволит вновь доказать недостижимость ошибочной метки уже вдоль другого пути. Такие требования к получаемому предикату как раз соответствуют ограничениям, накладываемым на интерполянт Крейга для пары логических формул[3]. Рассмотрим формулы и интерполянт, получаемый модифицированным инструментом CSIsat в данном примере (пусть переменной s_1 соответствует номер размещения 1, а s_2 – 2, в структуре два поля – f_1 и next с номерами 0 и 1 соответственно, константу N из четвёртой аксиомы возьмём равной 1000):



Интерполянт, выдаваемый CSIsat:

$V_6\left(S\left(L\left(V_6(2)\right),0\right)\right) = 2 \vee S\left(L\left(V_6(2)\right),0\right) \leq 1000$**Ошибка! Ожидалась цифра.** → (что соответствует предикату) V(S(L(V(2)), 0)) = 2 → (то есть) s_2->f_1 == 2. Это условие действительно выполнено в указанной точке программы и доказывает невыполнимость вдоль заданного пути. Оно же позволяет доказать

[3] Интерполянт Крейга должен логически следовать из первой формулы, быть невыполним в конъюнкции со второй и содержать лишь общие для обеих формул неинтерпретируемые символы.

невыполнимость, например, вдоль следующего пути, также проходящего через указанную точку:

```
s_1->f_1 = 1;
s_1 = s_1->next;
s_1->f_1 = 2;
s_2 = s_1;
s_1 = s_1->next;
----------------------
check(s_2->f_1 == 2); // Переход по другой ветви if, например
s_1->f_1 = s_2->f_1 – 2;
check(s_1->f_1 != 0);
ERROR:
```

На Рис. 1-7 приведены результаты замеров времени работы нового инструмента на сгенерированных тестовых наборах. В легенде для каждого теста условно показан фрагмент кода программы на языке C, примерно соответствующий формуле пути, использованной для получения интерполянтов в этом тесте. В приведенных фрагментах используется параметр генератора тестов *n*, задающий размер теста таким образом, что количество строк кода в нем линейно зависит от этого параметра примерно с коэффициентом 2. Место разреза, разделяющее получаемую формулу пути перед интерполяцией на две части, обозначено пунктиром. Проверки условий в ветвлениях на соответствующем пути выполнения указаны в виде вызовов функции check. В каждом тесте ошибочная метка ERROR недостижима, а соответствующая формула пути до нее невыполнима. Она генерируется с использованием описанных неинтерпретируемых функций и аксиом и передается модифицированному инструменту CSIsat с опциями +extrInsts и +simplInsts. Во всех тестах CSIsat запускался с лимитом по памяти 1 GB или лимитом по времени 600 секунд. Тестирование прекращалось после пяти повторных неуспешных завершений работы инструмента (в том числе превышений лимита по памяти или времени, а также аварийных завершений).

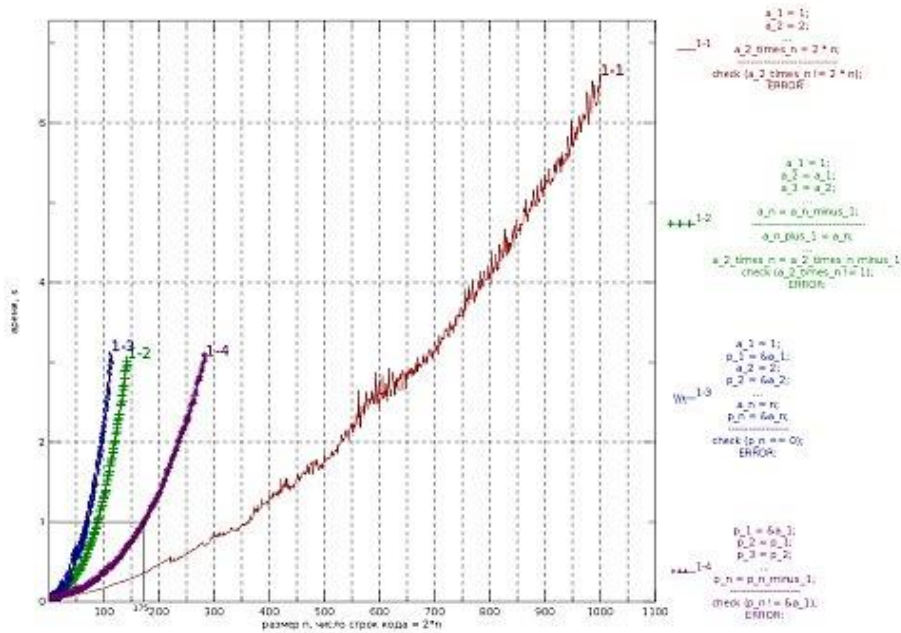


Рис. 1. Результаты тестирования на сгенерированном наборе.

На Рис. 1 приведены результаты замеров времени на интерполяцию в простых случаях, когда в пути отсутствуют операции с массивами или структурами. Кроме этого, для доказательства невыполнимости соответствующих формул не нужно инстанцировать выражения с кванторами для обновлений памяти (значение каждой переменной может использоваться только сразу после его обновления).

На Рис. 2 приведены результаты для аналогичных тестов с той разницей, что для доказательства невыполнимости в этих тестах требовалось линейное по количеству строк фрагмента программы число инстанцирований подкванторных выражений. Как видно из графиков, несмотря на то, что в обоих случаях с ростом размера теста время работы интерполятора возрастало примерно экспоненциально, в случае с линейным по размеру теста количеством инстанцирований время работы на некоторых тестах оказалось в 45 и более раз больше, чем в аналогичных случаях без инстанцирований (см., например, графики 1-4 и 2-4 для размера $n \approx 176$).

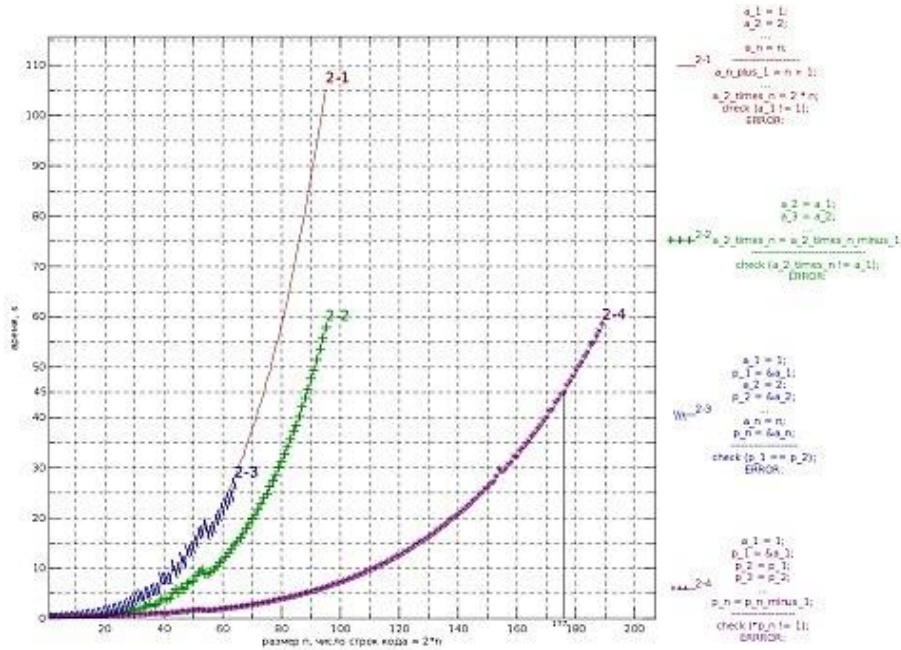


Рис.2. Результаты тестирования на сгенерированном наборе.

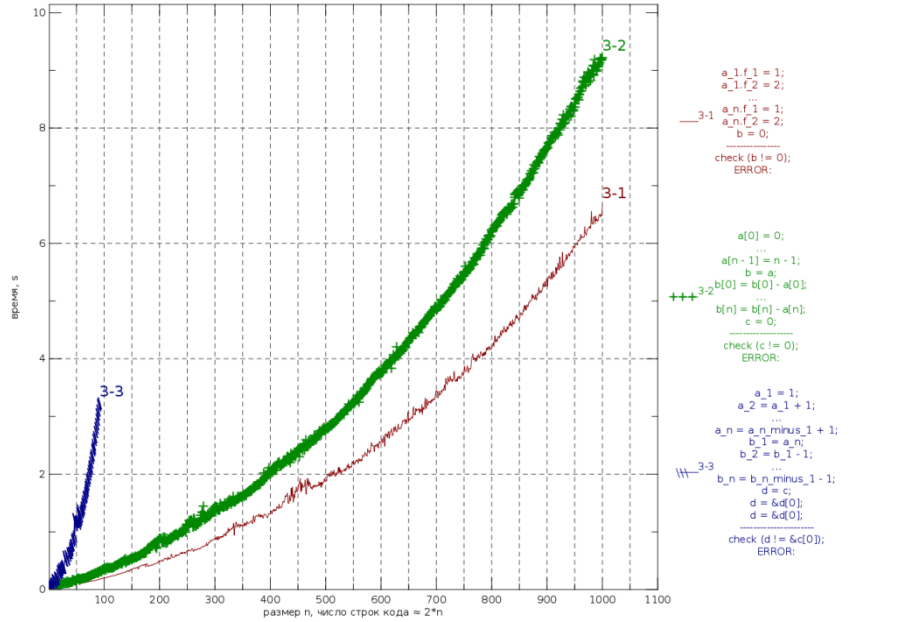


Рис. 3. Результаты тестирования на сгенерированном наборе.

На Рис. 3 приведены три графика: 3-1 и 3-2 для простых тестов с несущественным использованием массивов или структур (соответствующая часть формулы не требуется для доказательства невыполнимости), а также график 3-3 для теста, существенно использующего аксиому равенства адреса массива и его первого элемента.

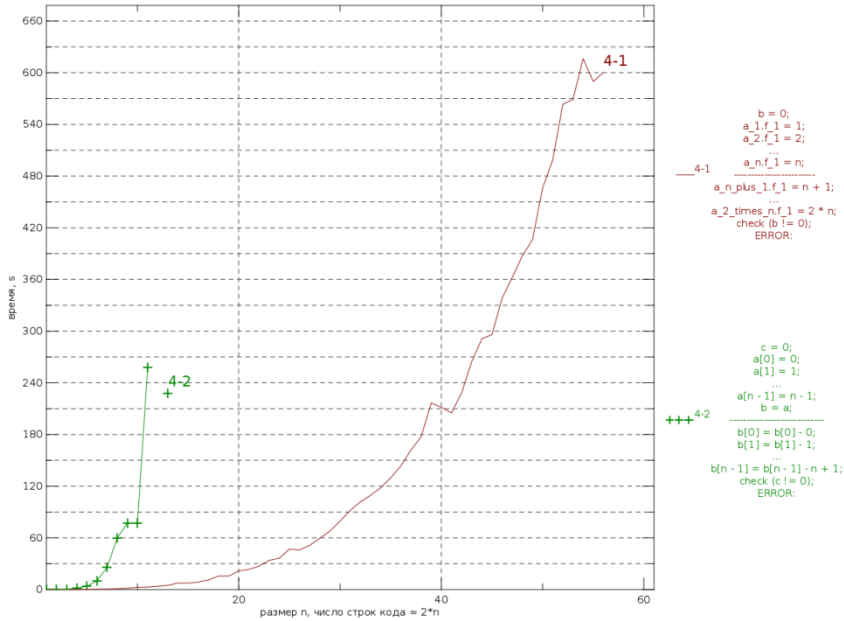


Рис. 4. Результаты тестирования на сгенерированном наборе.

На Рис. 4 приведены два графика для аналогичных тестов с линейным по размеру теста количеством инстанцирований. Разрыв графика 4-2 соответствуют отдельному аварийному завершению работы CSIsat (превышение лимита по памяти). Здесь разница в производительности интерполятора еще заметнее, чем на Рис. 1 и 2.

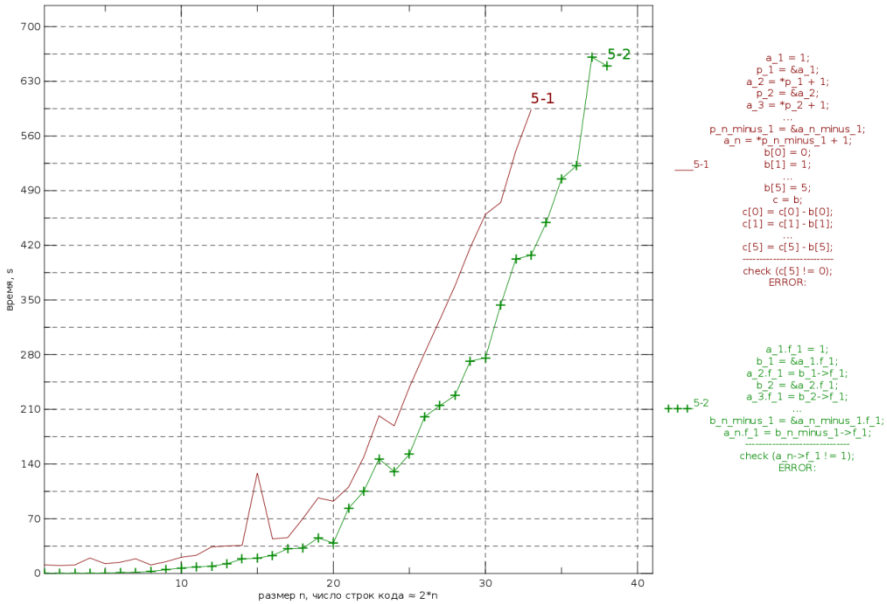


Рис. 5. Результаты тестирования на сгенерированном наборе.

На Рис. 5 в обоих тестах массивы или структуры использованы существенно. В тесте, соответствующем графику 5-1, для доказательства требовалось постоянное число инстанцирований, а в тесте для графика 5-2 – в два раза меньшее числа строк в пути. Однако на тестах небольших размеров эти величины оказались близкими (6 для первого теста и от 1 до 18 для второго).

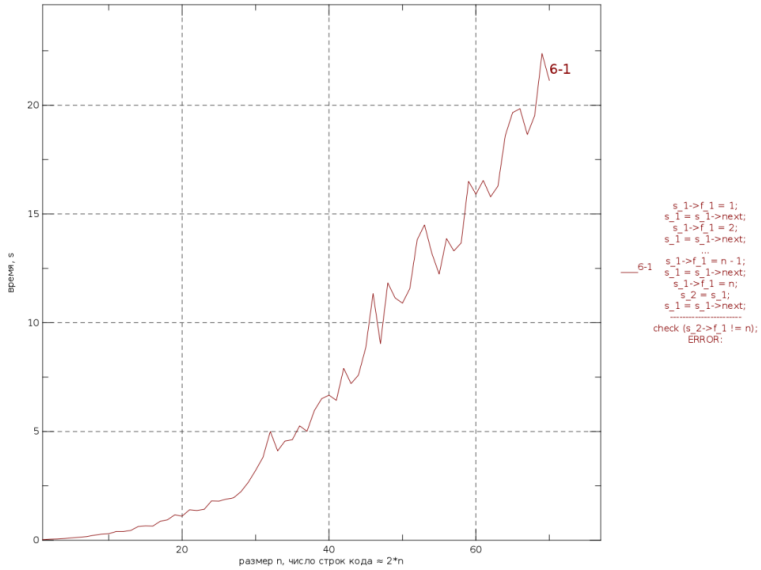


Рис. 6. Результаты тестирования на сгенерированном наборе.

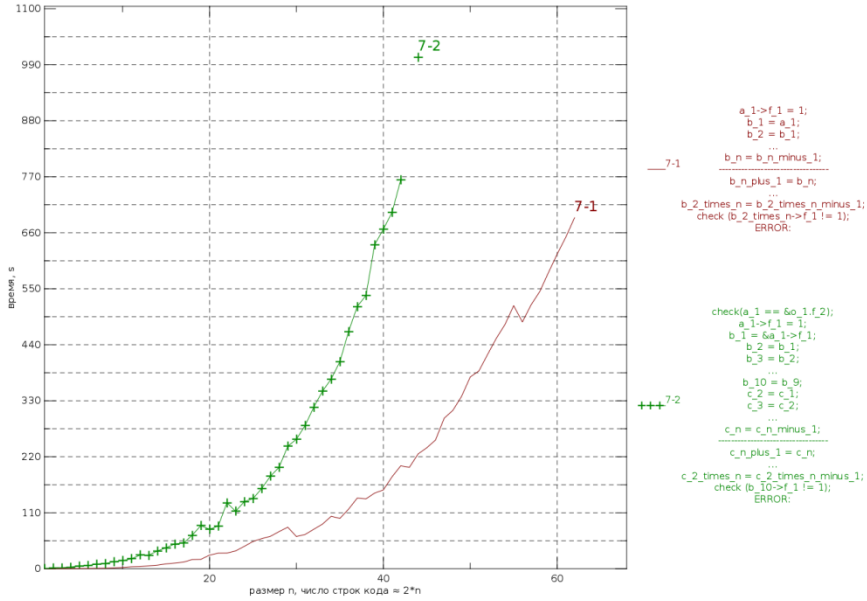


Рис. 7. Результаты тестирования на сгенерированном наборе.

На рисунках 6 и 7 приведены результаты для наиболее сложных тестов с существенным использованием структур. Разрыв графика 7-2 соответствуют отдельному аварийному завершению работы CSIsat (превышение лимита по памяти).

Результаты тестирования модифицированного инструмента показали, что он способен строить интерполянты для формул с кванторами лишь при условии, что для доказательства невыполнимости входной конъюнкции использовалось относительно небольшое число подстановок. При верификации, к примеру, драйверов ОС Linux средний размер трассы ошибки (то есть реальных контрпримеров) составляет порядка 1000-10000 операторов, а число вызовов интерполирующего решателя на один драйвер – примерно от 10 до 30[22]. Если считать разумным время верификации одного драйвера в пределах 15 минут, то для практического использования интерполятор должен в среднем справляться с формулами пути для 5000 строк за 45 секунд. Как видно, к примеру, из Рис. 2, даже для тестов без использования массивов или структур за 45 секунд модифицированный инструмент в лучшем случае может справиться с формулами для тестов с размером $n \approx 175$, то есть примерно с 350 строчками кода. Модифицированный инструмент, таким образом, оказался недостаточно эффективен для практического использования в качестве интерполятора при реализации в верификаторе рассмотренной логической модели памяти. Такая низкая производительность обусловлена как относительно низкой эффективностью решателя, используемого в CSIsat, так и использованием эвристик при извлечении необходимых подстановок из доказательства невыполнимости, полученного от CVC3. При использовании эвристик в получаемую задачу для интерполяции часто попадают несущественные для доказательства подстановки и, кроме этого, могут не попасть существенные, в результате чего инструмент завершит работу с ошибкой.

6. Выводы

Модифицированный инструмент CSIsat реализует интерполяцию Крейга для формул с кванторами, заданных в рамках комбинации теорий вещественной линейной арифметики и неинтерпретируемых функций. В нём используется предложенный в данной работе способ реализации интерполирования таких формул с помощью инстанцирования подкванторных выражений с использованием внешнего SMT-решателя CVC3.

Результаты тестирования модифицированного интерполятора как на задачах, полученных из набора SMTLIB, так и на специально сгенерированных для этих целей задачах показали общую работоспособность предложенного подхода и одновременно с эти его недостаточную эффективность.

Решатель, используемый при интерполяции для получения дерева резолютивного вывода в CSIsat, существенно уступает в эффективности CVC3. Поэтому одним из интересных направлений для дальнейшего

исследования является построение дерева резолютивного вывода по дереву доказательства невыполнимости, получаемому из CVC3. В таком случае удастся избежать существенных накладных расходов на повторное доказательство невыполнимости в самом CSIsat, которое, как оказалось по результатам проведенного тестирования, и являлось главным ограничением для инструмента.

Литература

[1] Ball, Thomas, Bounimova, Ella, Kumar, Rahul, and Levin, Vladimir. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Formal Methods in Computer Aided Design* (2010).

[2] Graf, Susanne and Saidi, Hassen. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification (CAV), 9th International Conference* (Haifa, Israel 1997), Springer, 72-83.

[3] Clarke, Edmund, Grumberg, Orna, Jha, Somesh, Lu, Yuan, and Veith, Helmut. Counterexample-Guided Abstraction Refinement. *Computer Aided Verification*, 1855 (2000), 154—169.

[4] Beyer, Dirk, Henzinger, Thomas A., Jhala, Ranjit, and Majumdar, Rupak. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9, 5 (2007), 505—525.

[5] Shved, Pavel, Mutilin, Vadim, and Mandrykin, Mikhail. Static Verfication `Under The Hood': Implementation Details and Improvements of BLAST. In *Proceedings of SYRCoSE* (Yekaterinburg 2011), 54-60.

[6] Beyer, Dirk and Keremoglu, M. Erkan. *CPAchecker: A Tool for Configurable Software Verification*. SFU-CS-2009-02, School of Computing Science, Simon Fraser University, 2009.

[7] De Moura, Leonardo and Bjorner, Nikolaj. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems* (Budapest, Hungary 2008), Springer-Verlag, 337—340. TACAS'08/ETAPS'08.

[8] Jhala, Ranjit and Majumdar, Rupak. Software model checking. *ACM Computing Surveys*, 41 (October 2009), 21:1—21:54.

[9] McMillan, Kenneth L. An Interpolating Theorem Prover. In *TACAS* (2004), 16-30.

[10] Beyer, Dirk, Zufferey, Damien, and Majumdar, Rupak. CSIsat: Interpolation for LA+EUF. In *CAV* (2008), 304-308.

[11] Cimatti, Alessandro, Griggio, Alberto, and Sebastiani, Roberto. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic (TOCL)*, 12, 1 (October 2010).

[12] Beyer, Dirk, Henzinger, Thomas A., and Théoduloz, Grégory. Lazy Shape Analysis. In *CAV'2006: Computer aided verification, LNCS 4144* (2005), Springer, 532—546.

[13] Reynolds, John. Separation Logic: A Logic for Shared Mutable Data Structures. (2002), IEEE Computer Society, 55—74.

[14] Ball, Thomas, Bounimova, Ella, Levin, Vladimir, and De Moura, Leonardo. *Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2*. 2010.

[15] Henzinger, Thomas A., Jhala, Ranjit, Majumdar, Rupak, and McMillan, Kenneth L. Abstractions from proofs. *SIGPLAN Not.*, 39, 1 (2004), 232—244.

[16] Christ, Juergen and Hoenicke, Jochen. Instantiation-Based Interpolation for Quantified Formulae. In *Decision Procedures in Software, Hardware and Bioware* (Dagstuhl, Germany 2010), Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. Dagstuhl Seminar Proceedings.

[17] Moura, Leonardo and Bjorner, Nikolaj. Efficient E-Matching for SMT Solvers. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction* (Bremen, Germany 2007), Springer-Verlag, 183—198. CADE-21.

[18] Barrett, Clark and Tinelli, Cesare. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)* (Berlin, Germany 2007), Springer-Verlag, 298—302. Lecture Notes in Computer Science.

[19] McMillan, Kenneth L. Interpolants from Z3 proofs. In *Formal Methods in Computer Aided Design (FMCAD 2011)* (Austin, TX, USA 2011).

[20] Barrett, Clark, Stump, Aaron, and Tinelli, Cesare. *The SMT-LIB Standard Version 2.0*. 2010.

[21] Library, SMT-LIB The Satisfiability Modulo Theories. *www.smtlib.org*. 2011.

[22] Khoroshilov, Alexey, Mutilin, Vadim, Novikov, Eugene, Shved, Pavel, and Strakh, Alexander. Towards An Open Framework for C Verification Tools Benchmarking. In *Proceedings of PSI* (Akademgorodok, Novosibirsk, Russia 2011), 82—91.

Instantiation-Based Interpolation for Quantified Formulae in CSIsat

Khoroshilov A.V., Mandrykin M.U., Mutilin V.S.
{khoroshilov, mandrykin, mutilin}@ispras.ru

Abstract. The paper describes an implementation of instantiation-based Craig interpolation for quantified formulae. The implementation is based on the CSIsat interpolating solver. The tool supports interpolation for formulae with linear real arithmetic, uninterpreted functions and quantifiers. The paper suggests usage of an external decision procedure (namely CVC3) for quantified formula instantiation. It describes how the CSIsat and CVC3 tools were modified in order to support quantified formulae interpolation. The paper also contains results for benchmarking the modified CSIsat tool on a set of tests obtained by randomly splitting tasks from the SMTLIB library as well as on a small collection of specific dedicated interpolation tasks generated by encoding several manually specified parameterized error trace patterns with quantified logical formulae. The approach to interpolation considered in the paper is based on the recently proposed extension of the McMillan's algorithm for resolution-based interpolant generation that was suggested earlier for implementation in the SMTInterpol interpolating solver by its developers. The extended algorithm additionally requires a set of quantified subformula instances sufficient for unsatisfiability proof of the initial formula and produces possibly quantified interpolants. A proper implementation of the algorithm could potentially be used in predicate abstraction-based verification tools for obtaining abstraction predicates from counterexamples by Craig interpolation. Though the evaluation presented in the paper showed that the considered implementation turned out to be too inefficient for this purpose due to significant repetitive proof overhead, which arose from combining a more efficient and advanced solver with a significantly less efficient one (in CVC3+CSIsat combination CSIsat is much less efficient than CVC3).

Keywords: interpolation, Craig interpolant, quantifiers, instantiation, solver, axioms.

References

[1]. Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static Driver Verification with Under 4% False Alarms. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 35-42, 2010.

[2]. Graf S., Saidi H. Construction of Abstract State Graphs with PVS. In Proc. Computer Aided Verification (CAV), LNCS, vol 1254, pp 72-83, 1997. doi: 10.1007/3-540-63166-6_10

[3]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement. In Proc. Computer Aided Verification (CAV), LNCS, vol. 1855, pp 154-169, 2000. doi: 10.1007/10722167_15

[4]. Beyer D., Henzinger T., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering, International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z

[5]. Shved P., Mutilin V., Mandrykin M. Static Verfication `Under The Hood': Implementation Details and Improvements of BLAST. In Proc. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), pp. 54-60, 2011.

[6]. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16
[7]. De Moura L., Bjorner N. Z3: an efficient SMT solver. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4963, pp. 337-340, 2008. doi: 10.1007/978-3-540-78800-3_24
[8]. Jhala R., Majumdar R. Software model checking. ACM Computing Surveys (CSUR), vol. 41, issue 4, article 21, 2009. doi: 10.1145/1592434.1592438
[9]. McMillan, Kenneth L. An Interpolating Theorem Prover. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 345, issue 1, pp. 101-121, 2005. doi: 10.1016/j.tcs.2005.07.003
[10]. Beyer D., Zufferey D., Majumdar R. CSIsat: Interpolation for LA+EUF. In Proc. Computer Aided Verification (CAV), LNCS, vol. 5123, pp. 304-308, 2008. doi: 10.1007/978-3-540-70545-1_29
[11]. Cimatti A., Griggio A., Sebastiani R. Efficient Interpolant Generation in Satisfiability Modulo Theories. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4963, pp. 397-412, 2008. doi: 10.1007/978-3-540-78800-3_30
[12]. Beyer D., Henzinger T. A., Théoduloz G. Lazy Shape Analysis. In Proc. Computer Aided Verification (CAV), LNCS, vol. 4144, pp. 532—546, 2005. doi: 10.1007/11817963_48
[13]. Reynolds J. Separation Logic: A Logic for Shared Mutable Data Structures. In Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 55-74, 2002.
[14]. Ball T., Bounimova E., Levin V., De Moura L. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. Microsoft Research Technical Report MSR-TR-2010-24, 2010.
[15]. Henzinger T. A., Jhala R., Majumdar R., McMillan K. L. Abstractions from proofs. In Proc. 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 232-244, 2004. doi: 10.1145/964001.964021
[16]. Christ J., Hoenicke J. Instantiation-Based Interpolation for Quantified Formulae. In Decision Procedures in Software, Hardware and Bioware, Dagstuhl Seminar Proceedings, no. 10161, 2010.
[17]. Moura L., Bjorner N. Efficient E-Matching for SMT Solvers. In Proc. 21st international conference on Automated Deduction (CADE), LNCS, vol. 4603, pp 183-198, 2007. doi: 10.1007/978-3-540-73595-3_13
[18]. Barrett C., Tinelli C. CVC3. In Proc. Computer Aided Verification (CAV), LNCS, vol. 4590, pp. 298-302, 2007. doi: 10.1007/978-3-540-73368-3_34
[19]. McMillan K. L. Interpolants from Z3 proofs. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 19-27, 2011.
[20]. Barrett C., Stump A., Tinelli C. The SMT-LIB Standard Version 2.0. In Proc. 8th International Workshop on Satisfiability Modulo Theories, 2010.
[21]. SMT-LIB The Satisfiability Modulo Theories. www.smtlib.org. 2011.
[22]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17

Анализ типовых ошибок в драйверах операционной системы Linux[1]

В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов
mutilin@ispras.ru, joker@ispras.ru, khoroshilov@ispras.ru

Аннотация. Быстрый темп развития ядра и драйверов операционной системы Linux, разрабатываемых большим распределенным сообществом программистов, привел к тому, что на сегодняшний день не существует единой базы правил, которые полностью описывают корректное взаимодействие драйверов и ядра. Это является препятствием, с одной стороны, для тех программистов, которые не обладают экспертными знаниями во всех особенностях данного взаимодействия; с другой стороны, для разработки и применения инструментов, которые могли бы находить соответствующие типовые ошибки автоматизированным образом. В данной статье предлагается методика выявления и классификации типовых ошибок и соответствующих им правил на основе изменений, вносимых в драйверы операционной системы Linux. В статье приводятся результаты применения данной методики, обсуждаются полученная классификация и распределение типовых ошибок по классам.

Ключевые слова: операционная система; ядро; драйвер; правило взаимодействия; классификация ошибок.

1. Введение

Ядро операционной системы (ОС) Linux является одним из самых динамично развивающихся и востребованных проектов в мире. Разработку ядра начал Линус Торвальдс в 1991 году. В настоящее время в подготовке каждого нового релиза ядра ОС Linux участвуют более 1000 человек, распределенных по всему миру [1]. Релизы выпускаются в среднем раз в 2-3 месяца. Начиная с ядра версии 2.6.24, выпущенного в начале 2008 года, каждый релиз включает порядка 9-12 тысяч изменений, что соответствует в среднем 5.4 изменениям в час. Размер исходного кода последней на сегодняшний день стабильной версии ядра 3.3 составляет более 15 млн. строк кода [2].

Официальное, так называемое оригинальное (от англ. mainline), ядро ОС Linux выпускает Линус Торвальдс [3]. Процесс разработки оригинального ядра устроен следующим образом. После того, как выпущена очередная версия

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 07.514.11.4104)

ядра, например, 2.6.36, начинается период активной разработки («merge window»). В ходе этого периода делается множество изменений, в основном, связанных со слиянием новой функциональности и исправлением ошибок, которые уже были обкатаны в ветках разработки соответствующих подсистем и в тестовом ядре (Рис. 1) [4].

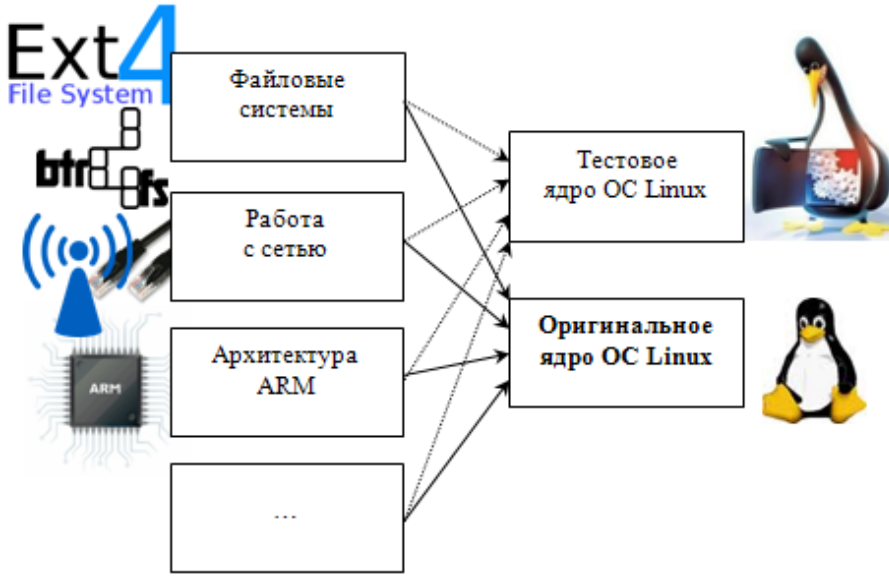


Рис. 1. Процесс формирования оригинального ядра ОС Linux

Через две-три недели период активной разработки завершается выпуском релиз-кандитата («release candidate») 2.6.37-rc1. По мере отладки выпускается серия релиз-кандитатов 2.6.37-rc2, 2.6.37-rc3 и т.д. После того, как все существенные проблемы исправлены, выпускается новая версия оригинального ядра 2.6.37 и начинается период активной разработки следующей версии 2.6.38. Одновременно с этим формируется специальная ветка для поддержки ядра версии 2.6.37, которая получает статус стабильной. В нее, как правило, попадают критические исправления из оригинальной ветки ядра. На основе этих изменений формируются стабильные версии 2.6.37.1, 2.6.37.2 и т.д. Помимо исправлений в стабильные версии попадает небольшое количество изменений, связанных с активацией поддержки новых устройств. Последнее означает, если в ядре уже реализована поддержка определенного семейства устройств, то при выпуске нового устройства из этого семейства в стабильную ветку ядра может попасть исправление, связывающее идентификатор устройства с драйвером семейства. По умолчанию, стабильная версия поддерживается до выпуска следующей версии оригинального ядра, но для отдельных версий продолжительность поддержки

может быть значительно продлена. Наглядно нумерация версий оригинального ядра ОС Linux представлена на **Ошибка! Источник ссылки не найден.**.

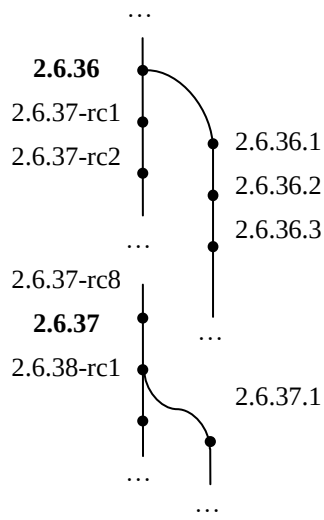


Рис. 2. Нумерация версий оригинального ядра ОС Linux

С мая 2011 года политика нумерации была изменена. Вместо ядра версии 2.6.40 было выпущено ядро версии 3.0. С этого момента для нумерации версий ядра вместо четырех цифр стали использовать три, причем вторая цифра играет роль бывшей третьей, третья – бывшей четвертой.

Существует большое количество других веток разработки ядра, которые основываются на оригинальном ядре ОС Linux (**Ошибка! Источник ссылки не найден.**). Как правило, разработчики различных дистрибутивов Linux поддерживают свои версии ядра. Об этом говорят, например, разработчики дистрибутивов Red Hat Enterprise Linux [5], openSUSE [6] и Debian [7]. Данные версии ядра отличаются от оригинальной тем, что в них поддерживается некоторая дополнительная функциональность и/или содержатся исправления ошибок. Есть ветки ядра, в которых основы операционной системы реализуются принципиально иначе по сравнению с оригинальной версией [8], [9]. С течением времени изменения, интересные широкому кругу лиц, из различных веток разработки попадают в оригинальное ядро [2], [4].

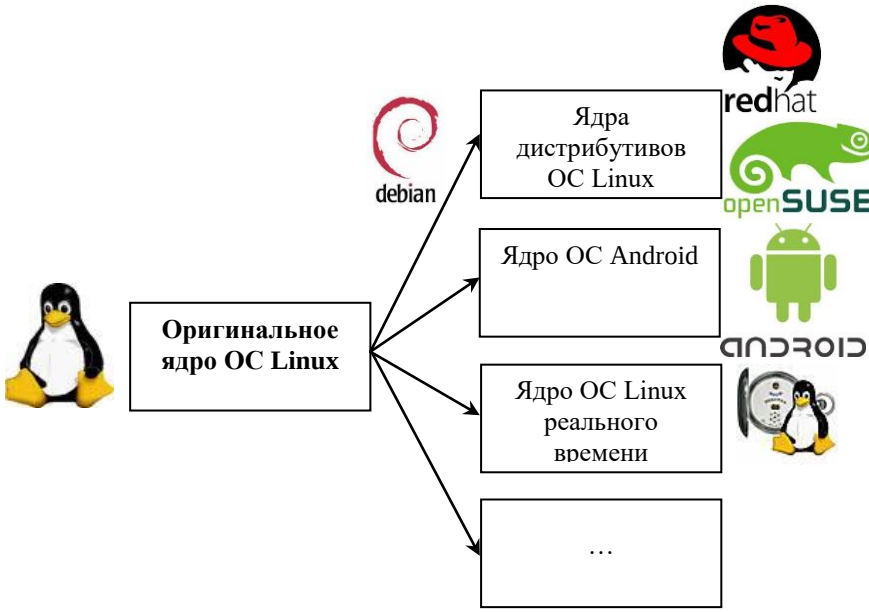


Рис. 3. Ядра, основывающиеся на оригинальном ядре ОС Linux

Схема устройства ядра ОС Linux представлена на Рис. 4. Ядро состоит из основной части («сердцевины») и драйверов устройств. Драйверы устройств участвуют во взаимодействиях:

- с сердцевиной ядра, обрабатывая события и запросы, связанные с соответствующим устройством, и вызывая «библиотечные» функции из сердцевины;
- с аппаратурой, которую они представляют в ядре;
- с пользовательскими приложениями посредством специальных файлов (procfs, sysfs или debugfs).

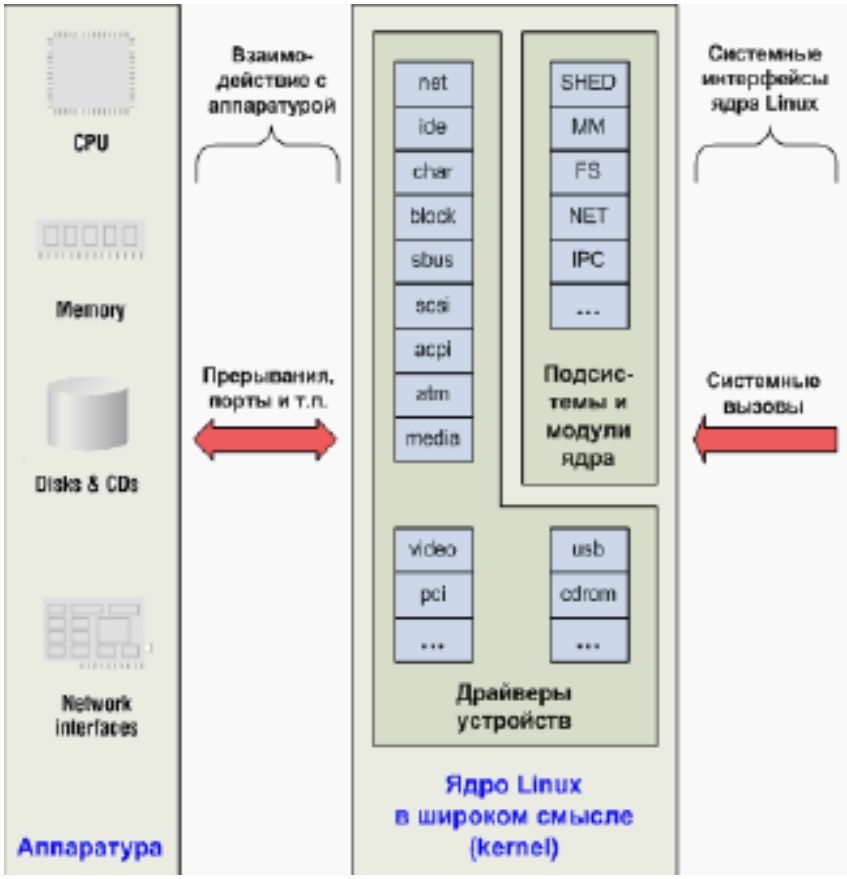


Рис. 4. Схема устройства ядра операционной системы Linux

Драйверы занимают наибольшую часть (до 70%) ядра ОС Linux. В исходном коде драйверов содержится достаточно большое количество различных ошибок, приводящих к некорректной работе всей ОС, зависаниям и падениям. Результаты исследований, которые были проделаны в работах [10] и [11] в начале 2000-х годов для ядер версий от 1.0 до 2.4.1, показали, что драйверы содержат до 85% всех ошибок, которые встречаются во всем ядре ОС Linux. Подобное исследование для ядра ОС Microsoft Windows XP в 2006 году также показало, что наибольшее количество ошибок в ядре данной ОС происходит в драйверах устройств [12]. Более поздние исследования, проведенные в 2011 году для ядер ОС Linux версий от 2.6.0 до 2.6.33, продемонстрировали, что хотя количество ошибок в драйверах стало меньше, чем в компонентах ядра, отвечающих за поддержку различных архитектур и файловых систем, их по-прежнему достаточно много [13].

Задача обеспечения надежности драйверов очень актуальна, так как драйверы работают с тем же уровнем привилегий, что и остальное ядро ОС Linux. Из-за этого, например, недоверенные пользовательские приложения могут использовать уязвимости в драйверах, чтобы исполнять произвольный код с привилегиями ядра и получать доступ к структурам ядра, что отмечено в работах [14] и [15]. В данных работах предлагаются автоматизированные решения, нацеленные на решение данной проблемы. Однако, с одной стороны, они не могут гарантировать полную защиту, требуют дополнительных настроек и т.п. С другой стороны, они могут достаточно сильно снизить производительность работы, что зачастую бывает недопустимо.

Ядро и большая часть драйверов ОС Linux, как входящих, так и не входящих в ядро, являются свободным программным обеспечением. Это, в частности, предоставляет широкие возможности по анализу и тестированию, в том числе на уровне исходного кода. В тестировании ядра и драйверов принимают участие огромное количество пользователей ОС Linux. То, что много людей имеют возможность просматривать исходный код ядра и драйверов ОС Linux с различных точек зрения создает эффект «многих глаз» [16]. Это позволяет обнаруживать и исправлять достаточно большое количество ошибок за короткое время. Тем не менее, данный подход не может гарантировать обнаружение всех ошибок в силу того, что только небольшое количество людей обладают достаточным уровнем знания во всех особенностях сложных взаимодействий между компонентами ядра, а также потому что на исходный код драйверов смотрят меньше, чем на исходный код сердцевины ядра [17].

Ошибки в драйверах можно условно разделить на типовые и нетиповые. К нетиповым относятся ошибки, которые связаны с некорректным взаимодействием с соответствующими устройствами и нарушениями контрактов. Нетиповые ошибки выделяются среди других ошибок тем, что они затрагивают характерные только для некоторого драйвера константы, ограничения, вычисления и т.д. Особенность типовых ошибок заключается в том, что для них можно выделить соответствующие правила, общие для всех драйверов либо для группы драйверов.

Среди типовых ошибок можно выделить три класса. В первый класс входят общие ошибки в драйверах, как в программах на языке программирования Си (ядро и драйверы ОС Linux разрабатываются на языке программирования Си). Например, разыменовывание нулевого указателя, превышение максимально возможных значений переменных с целым типом и т.п. Ко второму классу относятся специфичные ошибки, связанные с неправильным использованием интерфейса сердцевины ядра. К числу таких ошибок относятся, например, нарушения правил инициализации переменных, имеющих специфичные типы. Третий класс типовых ошибок включает в себя состояния гонок и взаимные блокировки, которые возникают при параллельном выполнении вследствие неиспользования или неправильного использования механизмов синхронизации.

Для типовых общих ошибок существуют классификации [18]. Правила, вследствие нарушения которых они происходят, практически не меняются со временем (вообще говоря, язык программирования Си и его расширения продолжают развиваться, что приводит к незначительным изменениям в общих правилах). Ошибки, связанные с параллельным выполнением, также достаточно стандартные и мало меняются с течением времени.

Типовые специфичные ошибки не являются стандартными. Правила, соответствующие им меняются, удаляются и появляются достаточно часто ввиду того, что разработка ядра ОС Linux, как уже отмечалось, ведется большими темпами, чтобы отвечать современным направлениям развития и поддерживать новые особенности архитектур и устройств, в том числе, меняется интерфейс сердцевины ядра. По заявлению Грега Кроа-Хартмана, одного из ведущих разработчиков ядра ОС Linux, предполагается, что интерфейс основной части ядра является нестабильным [19].

Целью данной работы является выявление и классификация типовых ошибок и соответствующих им правил в драйверах операционной системы Linux. В разделе 2 данной статьи предложена методика выявления и классификации типовых ошибок. Раздел 3 описывает результаты практического применения данной методики для драйверов ОС Linux. Аналогичные подходы рассмотрены в разделе 4. В заключении подводятся итоги проведенной работы и приводятся возможные направления дальнейшего развития.

2. Методика выявления и классификации типовых ошибок в драйверах ОС Linux

2.1. Выбор источника типовых ошибок

Выявлять правила, соответствующие типовым ошибкам в драйверах ОС Linux, в особенности, ошибкам некорректного использования интерфейса сердцевины ядра, можно на основе различных источников. Достаточно много информации можно найти в документации, в том числе, входящей в состав ядра [20] и в исходный код ядра и драйверов, а также в литературе по разработке драйверов и ядра ОС Linux (см., например, [21], [22]). Однако, в этих источниках документированы далеко не все особенности различных подсистем ядра и типов драйверов. Поскольку ядро развивается очень стремительно, данные источники не всегда поддерживаются в актуальном состоянии.

В качестве еще одного источника для выявления типовых ошибок можно рассмотреть список рассылки ядра Linux (от англ. Linux Kernel Mailing List, LKML) [23]. В данном списке рассылки обсуждаются различные актуальные вопросы по разработке ядра, в том числе, касательно исправления ошибок. На основании этих обсуждений можно выявить множество типовых общих и специфичных ошибок. Анализ сообщений в LKML достаточно трудоемок,

поскольку сообщений очень много и они содержат большое количество информации, причем не только технической. Кроме того, много ошибок обсуждается в других списках рассылок, на форумах, системах отслеживания ошибок и т.д.

Типовые ошибки и правила можно выявлять на основе анализа изменений, вносимых в драйверы ОС Linux. Данный источник является достаточно актуальным, поскольку раньше или позже среди изменений появляются все важные исправления ошибок, которые обсуждаются в различных списках рассылок, на конференциях, форумах и т.д. Изменения в драйверах ОС Linux проходят достаточно тщательную предварительную процедуру согласования и проверки, в том числе, у экспертов в соответствующих подсистемах ядра [4].

Каждое изменение содержит в себе достаточно подробную информацию, включающую, авторов и время, краткое название изменения, подробное описание (возможно, со ссылками на соответствующие обсуждения), изменение исходного кода драйверов и/или ядра ОС Linux и различную вспомогательную информацию (см. примеры далее в данном разделе). В рамках данной статьи изменения, вносимые в драйверы ОС Linux, были выбраны в качестве источника для анализа типовых ошибок в драйверах.

2.2. Методика выявления и классификации типовых ошибок

Первый шаг методики анализа изменения в драйвере ОС Linux заключается в том, чтобы определить, является ли оно исправлением ошибки или оно каким-либо образом расширяет функциональность драйвера (например, добавляет поддержку новых устройств). Для изменений, которые являются исправлением ошибок, необходимо определить типовые или нетиповые ошибки они исправляют. Для этого необходимо проанализировать описание исправления и изменение исходного кода. В том случае, если ошибка связана с неправильным использованием специфичных для драйвера или группы драйверов констант, условий и вычислений, ее необходимо отнести к нетиповым. Иначе – к типовым.

Основным шагом предлагаемой методики анализа изменений в драйверах ОС Linux является классификация типовых ошибок. Данный шаг требует осмысления причины, которая привела к типовой ошибке. Дело в том, что с первого взгляда может показаться, что ошибка проявилась вследствие нарушения некого общего правила. На самом деле, причина ошибки может крыться в особенностях параллельного выполнения либо в некорректном использовании интерфейса сердцевины ядра. Кроме того, не всегда легко определить, какую именно общую или специфичную ошибку исправляет рассматриваемое изменение. На данном шаге помимо анализа описания и изменения в исходном коде предполагается более тщательный анализ взаимодействий различных подсистем ядра и драйвера, для чего можно использовать, например, [24].

Важно отметить, что список классов не известен заранее и может быть определен только непосредственно по ходу самого анализа. Предполагается, что, в первую очередь, классы позволят различить общие и специфичные ошибки в драйверах, а также ошибки, связанные с параллельным выполнением. Общие ошибки и ошибки, связанные с параллельным выполнением, должны быть разделены посредством подклассов на некоторые хорошо устоявшиеся и общепринятые группы, например, наподобие [18]. Для специфичных ошибок подклассы должны показывать их характерные особенности, а также указывать на затрагиваемые подсистемы сердцевины ядра ОС Linux. Подобное разбиение позволит более точно классифицировать типовые ошибки и построить распределение типовых ошибок по различным классам и подклассам.

Заключительный шаг методики по анализу изменения в драйвере ОС Linux – это описание характерных особенностей изменения посредством вспомогательного комментария. В дальнейшем комментарии помогут быстрее понимать суть проанализированных изменений и вносить некоторые поправки в классификацию при необходимости.

2.3. Примеры применения предложенной методики

Рассмотрим применение предложенной методики на примерах. На основании описания и изменения в исходном коде для изменения в драйвере, представленного на **Ошибка! Источник ссылки не найден.** можно сделать вывод, что данное изменение в драйвере не является исправлением ошибки, а добавляет поддержку нового USB устройства. В качестве комментария к данному изменению можно написать «*Add WAGO 750-923 Service Cable device ID*».

```
commit 5fc8fe8e2ea30805bee5a13420817d6ad34ea9ce
Author: Anders Larsen <al@alarsen.net>
Date:   Wed Oct 6 23:46:25 2010 +0200

    USB: cp210x: Add WAGO 750-923 Service Cable device ID

    commit 93ad03d60b5b18897030038234aa2ebae8234748 upstream.

    The WAGO 750-923 USB Service Cable is used for configuration and
firmware
    updates of several industrial automation products from WAGO
Kontakttechnik GmbH.

    (skipped)

    Signed-off-by: Anders Larsen <al@alarsen.net>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/drivers/usb/serial/cp210x.c b/drivers/usb/serial/cp210x.c
index 3ad53bd..9927bca 100644
```

```
--- a/drivers/usb/serial/cp210x.c
+++ b/drivers/usb/serial/cp210x.c
@@ -132,6 +132,7 @@ static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x17F4, 0xAAAA) }, /* Wavesense Jazz blood
glucose meter */
        { USB_DEVICE(0x1843, 0x0200) }, /* Vaisala USB Instrument
Cable */
        { USB_DEVICE(0x18EF, 0xE00F) }, /* ELV USB-I2C-Interface */
+       { USB_DEVICE(0x1BE3, 0x07A6) }, /* WAGO 750-923 USB Service
Cable */
        { USB_DEVICE(0x413C, 0x9500) }, /* DW700 GPS USB interface */
        { } /* Terminating Entry */
 };
```

Рис. 5. Описание изменения 5fc8fe8, добавляющего поддержку нового устройства

Изменение в драйвере на Рис. демонстрирует исправление нетиповой ошибки, вызванной нарушением контракта драйвера трекпада. Изменение исходного кода задействует специфичные для данного драйвера константы и ограничения. Комментарий может быть следующий: «*Ignore the relative axes from the Magic Trackpad*».

```
commit 8e6b41c76c5b8a27b2abd7b9f6ed0877987fd11b
Author: Chase Douglas <chase.douglas@canonical.com>
Date:   Tue Jan 11 19:37:50 2011 +0100

    HID: magicmouse: Don't report REL_{X, Y} for Magic Trackpad

    [ Linus' tree commit 6a66bbd693c12f71697c61207aa18bc5a12da0ab ]

    With the recent switch to having the hid layer handle standard
axis
    initialization, the Magic Trackpad now reports relative axes. This
would
    be fine in the normal mode, but the driver puts the device in
multitouch
    mode where no relative events are generated. Also, userspace
software
    depends on accurate axis information for device type detection.
Thus,
    ignoring the relative axes from the Magic Trackpad is best.

    Signed-off-by: Chase Douglas <chase.douglas@canonical.com>
    Signed-off-by: Jiri Kosina <jkosina@suse.cz>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/drivers/hid/hid-magicmouse.c b/drivers/hid/hid-
magicmouse.c
index e6dc151..ed732b7 100644
--- a/drivers/hid/hid-magicmouse.c
+++ b/drivers/hid/hid-magicmouse.c
@@ -433,6 +433,11 @@ static int magicmouse_input_mapping(struct
hid_device *hdev,
        if (!msc->input)
```

```
                msc->input = hi->input;

+       /* Magic Trackpad does not give relative data after switching
to MT */
+       if (hi->input->id.product == USB_DEVICE_ID_APPLE_MAGICTRACKPAD
&&
+           field->flags & HID_MAIN_ITEM_RELATIVE)
+               return -1;
+
        return 0;
 }
```

Рис.6. Описание изменения 8e6b41c, исправляющего нетиповую ошибку

На **Ошибка! Источник ссылки не найден.** представлено исправление типовой общей ошибки, переполнения целых чисел. В качестве короткого комментария к данному изменению можно взять часть его описания, например: «*An integer overflow occurs in the calculation of RHlinear when the relative humidity is greater than around 30%*».

```
commit 4e3837bb22675cef56b921230f7604fe1a451a5f
Author: Vivien Didelot <vivien.didelot@savoirfairelinux.com>
Date:   Mon Mar 21 17:59:35 2011 +0100

    hwmon: (sht15) Fix integer overflow in humidity calculation

    commit ccd32e735de7a941906e093f8dca924bb05c5794 upstream.

    An integer overflow occurs in the calculation of RHlinear when the
    relative humidity is greater than around 30%. The consequence is a
subtle
    (but noticeable) error in the resulting humidity measurement.

    Signed-off-by:                    Vivien                    Didelot
<vivien.didelot@savoirfairelinux.com>
    Signed-off-by: Jean Delvare <khali@linux-fr.org>
    Cc: Jonathan Cameron <jic23@cam.ac.uk>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/drivers/hwmon/sht15.c b/drivers/hwmon/sht15.c
index a610e78..38a41d2 100644
--- a/drivers/hwmon/sht15.c
+++ b/drivers/hwmon/sht15.c
@@ -333,11 +333,11 @@ static inline int sht15_calc_humid(struct
sht15_data *data)

        const int c1 = -4;
        const int c2 = 40500; /* x 10 ^ -6 */
-       const int c3 = -2800; /* x10 ^ -9 */
+       const int c3 = -28; /* x 10 ^ -7 */

        RHlinear = c1*1000
```

```
                + c2 * data->val_humid/1000
-               + (data->val_humid * data->val_humid * c3)/1000000;
+               + (data->val_humid * data->val_humid * c3) / 10000;
        return (temp - 25000) * (10000 + 80 * data->val_humid)
                / 1000000 + RHlinear;
}
```

Рис.7. Описание изменения 4e3837b, исправляющего типовую общую ошибку (переполнение целых чисел)

На Рис. представлено изменение в драйвере, которое является исправлением типовой специфичной ошибки. Данная ошибка вызвана неправильным использованием интерфейса сердцевины ядра, а именно, в контексте блокировки вызывается функция выделения памяти, которая может заснуть [25]. Основываясь на описании изменения, возможно написать следующий комментарий: «*AppArmor may do a GFP_KERNEL memory allocation while holding lock*».

```
commit 83a9a8034ee98ac21804c376ec90af8e4997790e
Author: John Johansen <john.johansen@canonical.com>
Date:   Wed Jun 8 15:07:47 2011 -0700

    AppArmor: Fix sleep in invalid context from task_setrlimit

    commit 1780f2d3839a0d3eb85ee014a708f9e2c8f8ba0e upstream.

    Affected kernels 2.6.36 - 3.0

    AppArmor may do a GFP_KERNEL memory allocation with task_lock(tsk-
>group_leader);
    held when called from security_task_setrlimit.  This will only
occur when the
    task's current policy has been replaced, and the task's creds have
not been
    updated before entering the LSM security_task_setrlimit() hook.

    BUG: sleeping function called from invalid context at
mm/slub.c:847
     in_atomic(): 1, irqs_disabled(): 0, pid: 1583, name: cupsd

    (skipped)

    Signed-off-by: John Johansen <john.johansen@canonical.com>
    Reported-by: Miles Lane <miles.lane@gmail.com>
    Signed-off-by: James Morris <jmorris@namei.org>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/security/apparmor/lsm.c b/security/apparmor/lsm.c
index ec1bcec..3d2fd14 100644
--- a/security/apparmor/lsm.c
+++ b/security/apparmor/lsm.c
@@ -612,7 +612,7 @@ static int apparmor setprocattr(struct task struct
```

```
*task, char *name,
 static int apparmor_task_setrlimit(struct task_struct *task,
                unsigned int resource, struct rlimit *new_rlim)
 {
-       struct aa_profile *profile = aa_current_profile();
+       struct aa_profile *profile = __aa_current_profile();
        int error = 0;

        if (!unconfined(profile))
```

Рис.8. Описание изменения 83a9a80, исправляющего типовую специфичную ошибку (вызов функции, которая может заснуть, в контексте блокировки)

Рис. демонстрирует типовую ошибку с общим проявлением (использование памяти после ее освобождения) при том, что ее причина заключается в некорректном использовании спин блокировки при параллельном выполнении. Комментарий к данному изменению может быть следующий: «*Calls to ath9k_hw_startpcureceive should be performed under lock*».

```
commit 43d7d3dec4052bb98790a26f67abbb01b5e1b9f3
Author: Luis R. Rodriguez <lrodriguez@atheros.com>
Date:   Wed Oct 20 16:07:04 2010 -0700

    ath9k: add locking for starting the PCU on RX

    commit 7583c550c3e635dcc61ab127c36ecefd59fb8dc8 upstream.

    There was some locking for starting some parts of
    RX but not for starting the PCU. Include this otherwise
    we can content against stopping the PCU.

    This can potentially lead to races against different
    buffers on the PCU which can lead to to the DMA RX
    engine writing to buffers which are already freed.

    (skipped)

    Cc: Ben Greear <greearb@candelatech.com>
    Cc: Kyungwan Nam <kyungwan.nam@atheros.com>
    Signed-off-by: Luis R. Rodriguez <lrodriguez@atheros.com>
    Tested-by: Ben Greear <greearb@candelatech.com>
    Signed-off-by: John W. Linville <linville@tuxdriver.com>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

    index e581b1f..b3c9baf 100644

diff           --git           a/drivers/net/wireless/ath/ath9k/recv.c
b/drivers/net/wireless/ath/ath9k/recv.c
index 8b1b459..2a2b3f7 100644
--- a/drivers/net/wireless/ath/ath9k/recv.c
+++ b/drivers/net/wireless/ath/ath9k/recv.c
@@ -288,11 +288,11 @@ static void ath edma start recv(struct ath softc
```

```
*sc)
        ath_rx_addbuffer_edma(sc, ATH9K_RX_QUEUE_LP,
                              sc-
>rx.rx_edma[ATH9K_RX_QUEUE_LP].rx_fifo_hwsize);

-       spin_unlock_bh(&sc->rx.rxbuflock);
-
        ath_opmode_init(sc);

        ath9k_hw_startpcureceive(sc->sc_ah,       (sc->sc_flags       &
SC_OP_OFFCHANNEL));
+
+       spin_unlock_bh(&sc->rx.rxbuflock);
 }

 static void ath_edma_stop_recv(struct ath_softc *sc)
@@ -494,10 +494,11 @@ int ath_startrecv(struct ath_softc *sc)
        ath9k_hw_rxena(ah);

 start_recv:
-       spin_unlock_bh(&sc->rx.rxbuflock);
        ath_opmode_init(sc);
        ath9k_hw_startpcureceive(ah,           (sc->sc_flags           &
SC_OP_OFFCHANNEL));

+       spin_unlock_bh(&sc->rx.rxbuflock);
+
        return 0;
 }
```

Рис.9. Описание изменения 952ef78, исправляющего типовую ошибку, проявляющуюся при параллельном выполнении

3. Результаты анализа изменений в драйверах ОС Linux

3.1. Определение ветки ядра, наиболее удобной для анализа изменений

Для анализа изменений в драйверах в рамках данной работы был выбран официальный репозиторий ядра ОС Linux, в котором собраны все версии оригинального ядра за последние несколько лет [26]. На первом этапе был проанализирован ряд последовательных изменений на пути разработки между ядром версии 2.6.38 и версии 2.6.39-rc1. Сводные результаты представлены в Табл. 1. Всего было проанализировано 366 изменений, из которых 100 были исправлениями ошибок. 69 из них (около 19% от общего числа) составляли исправления типовых ошибок.

Изменения (366)		
Расширение функциональности (266 - 73%)	Исправление ошибок (100 – 27%)	
-	Нетиповые ошибки (31 – 8%)	Типовые ошибки (69 – 19%)

Табл. 1. Анализ ряда изменений между ядрами версий 2.6.38 и 2.6.39

Аналогичным образом были проанализированы последовательные изменения, сделанные для ядра 3.0.8 по сравнению с версией 3.0.7 (Табл. 2)[2]. В данном случае оказалось, что около 37% изменений являются исправлениями типовых ошибок, что существенно больше по сравнению с результатами предыдущего анализа (Табл. 1). Это наглядным образом демонстрирует утверждение о том, что при разработке следующей версии ядра большая часть изменений связана с реализацией поддержки новой функциональности, а не с исправлением ошибок. Также стоит отметить, что подряд идущие изменения в стабильной ветке оказались существенно разнообразнее в плане затрагиваемых подсистем по сравнению с основной веткой разработки ядра. В связи с этим в ходе основного анализа, выполненного в рамках данной работы, рассматривались только изменения в стабильные ветки ядра.

Изменения (43)		
Расширение функциональности (17 - 40%)	Исправление ошибок (26 – 60%)	
-	Нетиповые ошибки (10 – 23%)	Типовые ошибки (16 – 37%)

Табл. 2. Анализ изменений стабильной ветки ядра между версиями 3.0.7 и 3.0.8

3.2. Основной анализ изменений в драйверах ядра ОС Linux

В рамках основного анализа рассматривались изменения, которые были сделаны в стабильных ядрах с 2.6.35 по 3.0 в репозитории [26] за время, начиная с 26 октября 2010 года по 26 октября 2011 года. Всего таких изменений насчитывается 3101. Среди данных изменений некоторые являются дубликатами, поскольку одни и те же изменения попадают в различные стабильные ветки. Уникальных изменений за указанный период времени насчитывается 2623 (около 84.6% от общего числа изменений).

[2] Напомним, что нумерация для ядер версии больше 3 отличается, поэтому, например, версия 3.0.7 соответствует версии 2.6.38.7.

Для того чтобы отделить изменения в драйверах от остальных изменений в ядре, рассматривались только такие изменения, которые затрагивали файлы из папок: *crypto*, *drivers*, *sound*, *security*, *include/acpi*, *include/crypto*, *include/drm*, *include/media*, *include/mtd*, *include/pcmcia*, *include/rdma*, *include/rxrpc*, *include/scsi*, *include/sound* и *include/video* (в соответствии с, например, [27] и [28]). Из всех уникальных изменений, сделанных в стабильных ядрах с 26.10.10 по 26.10.11, в драйверах оказалось 1503, т.е. около 57.3% от общего числа уникальных изменений. Сводные результаты основного анализа изменений в драйверах представлены в Табл. 3. Детали анализа доступны публично [29].

<table>
<tr><td colspan="3">Изменения в драйверах (1503)</td></tr>
<tr><td>Расширение функциональности (321 - 21%)</td><td colspan="2">Исправление ошибок (1182 – 79%)</td></tr>
<tr><td>-</td><td>Нетиповые ошибки (786 – 52%)</td><td>Типовые ошибки (396 – 27%)</td></tr>
</table>

Табл. 3. Анализ изменений в драйверах стабильных версий ядра с 2.6.35 по 3.0 за время с 26.10.10 по 26.10.11

В соответствии с методикой, предложенной в данной работе, была произведена классификация выявленных типовых ошибок. При классификации не рассматривались изменения, которые являются исправлением типовых специфичных ошибок, связанных с некорректным использованием интерфейса группы драйверов (51 изменение). Изменениям, которые включали исправления сразу нескольких ошибок, приписывалось соответствующее количество классов.

Описание определенных подклассов первого уровня для типовых ошибок в драйверах ОС Linux представлено в Табл. 4. Подклассы ошибок первого уровня с небольшим количеством представителей (не более 3) были объединены в *misc*.

№	Класс	1-й подкласс	Краткое описание типовых ошибок
1	specific	resource	Утечки и использование после освобождения специфичных объектов
2		check_params	Нарушение ограничений на входные параметры
3		context	Вызовы функций в недопустимых контекстах
4		uninit	Некорректная инициализация специфичных объектов
5		lock	Некорректное использование механизмов синхронизации (не требуется параллельное выполнение)
6		style	Нарушение стиля оформления драйверов ядра ОС Linux
7		net	Сетевая подсистема
8		usb	USB подсистема
9		check_ret_val	Отсутствие проверок возвращаемых значений
10		dma	Подсистема прямого доступа к памяти
11		device	Общая модель драйвера
12		misc	Разное
13	generic	null_ptr_deref	Разыменование нулевого указателя
14		resource	Утечки памяти и ее использование после освобождения
15		syntax	Синтаксис
16		int_overflow	Переполнение целых чисел
17		buffer_overflow	Переполнение массива
18		uninit	Использование неинициализованных областей памяти
19		misc	Разное
20	sync	race	Состояние гонки
21		deadlock	Взаимная блокировка

Табл. 4. Описание подклассов первого уровня для типовых ошибок в драйверах ОС Linux

Максимальный уровень вложенности подклассов в рамках проведенной классификации составил 5 подклассов. В среднем насчитывалось 2-3 подкласса. В Табл. 5 приводится описание подклассов второго уровня для типовых специфичных ошибок в драйверах ОС Linux, связанных с

использованием особого контекста. Подклассы для других типовых ошибок составлялись аналогичным образом.

№	Класс и 1-й подкласс	2-й подкласс	Краткое описание типовых ошибок
1	specific:context	sleep	Вызов функций, которые могут заснуть, в контексте прерывания, блокировки и т.д.
2		interrupt	Некорректные операции в контексте прерывания
3		lock	Некорректные операции в контексте блокировки
4		timer	Некорректные операции в контексте таймера

Табл. 5. Описание подклассов второго уровня для типовых ошибок в драйверах ОС Linux, связанных с некорректным использованием механизмов синхронизации

В Табл. 6 и Табл. 7 представлены результаты классификации и распределение типовых ошибок в драйверах ОС Linux с точки зрения различных аспектов. В Табл. 6 приводится общее распределение типовых ошибок по классам и первым подклассам. Цветом выделяются границы, когда суммарное количество типовых ошибок достигают 50% и 80% от общего числа ошибок. В Табл. 7 приводится распределение типовых ошибок по классам и по первым подклассам для каждого класса. По аналогии для каждого класса цветом выделены границы 50% и 80%.

№	Класс и 1-й подкласс	Количество	% от общего	Суммарный % от общего
1	sync:race	60	17.2%	17.2%
2	specific:resource	32	9.2%	26.4%
3	generic:null_ptr_deref	31	8.9%	35.2%
4	specific:check_params	25	7.2%	42.4%
5	generic:resource	24	6.9%	49.3%
6	specific:context	19	5.4%	54.7%
7	specific:uninit	17	4.9%	59.6%
8	generic:syntax	14	4.0%	63.6%
9	specific:lock	12	3.4%	67.0%
10	sync:deadlock	11	3.2%	70.2%
11	specific:style	10	2.9%	73.1%
12	specific:net	10	2.9%	75.9%
13	specific:usb	9	2.6%	78.5%
14	generic:int_overflow	8	2.3%	80.8%

15	generic:buffer_overflow	8	2.3%	83.1%
16	specific:check_ret_val	7	2.0%	85.1%
17	generic:uninit	6	1.7%	86.8%
18	specific:dma	4	1.1%	88.0%
19	specific:device	4	1.1%	89.1%
20	specific:misc	27	7.7%	96.8%
21	generic:misc	11	3.2%	100.0%

Табл. 6. Классификация и распределение типовых ошибок в драйверах ОС Linux

№	Класс	1-й подкласс	Количество	% от общего	Суммарный % от общего
1	**specific** (176 – 50.4%)	resource	32	18.2%	18.2%
2		check_params	25	14.2%	32.4%
3		context	19	10.8%	43.2%
4		uninit	17	9.7%	52.8%
5		lock	12	6.8%	59.7%
6		style	10	5.7%	65.3%
7		net	10	5.7%	71.0%
8		usb	9	5.1%	76.1%
9		check_ret_val	7	4.0%	80.1%
10		dma	4	2.3%	82.4%
11		device	4	2.3%	84.7%
12		misc	27	15.3%	100.0%
13	**generic** (102 – 29.2%)	null_ptr_deref	31	30.4%	30.4%
14		resource	24	23.5%	53.9%
15		syntax	14	13.7%	67.6%
16		int_overflow	8	7.8%	75.5%
17		buffer_overflow	8	7.8%	83.3%
18		uninit	6	5.9%	89.2%
19		misc	11	10.8%	100.0%
20	**sync** (71 – 20.4%)	race	60	84.5%	84.5%
21		deadlock	11	15.5%	100.0%

Табл. 7. Классификация и распределение типовых ошибок в драйверах ОС Linux в зависимости от класса

3.3. Выводы по результатам анализа

На основании результатов, полученных в рамках данного исследования можно сделать следующие выводы:

- Порядка 27% от общего числа уникальных изменений в драйверах являются исправлениями типовых ошибок, 52% - исправления нетиповых ошибок и 21% - реализация поддержки новой функциональности.
- Типовые ошибки в драйверах ядра ОС Linux относятся к одному из 3 классов и 21 подклассов первого уровня. Классы разделяют ошибки общие, специфичные и связанные с параллельным выполнением. Их подклассы соответствуют наиболее выразительным особенностям типовых ошибок и достаточно четко определены. Подклассы следующих уровней в основном показывают, к каким подсистемам ядра относятся ошибки.
- Более 50% типовых ошибок соответствует 5 подклассам первого уровня. Самые часто встречающиеся ошибки связаны с состоянием гонок при параллельном выполнении программы (около 17%). На втором и третьем месте идут соответственно утечки специфичных объектов и разыменования нулевого указателя (по 9%). Более 80% типовых ошибок соответствует 14 подклассам первого уровня.
- Специфичные ошибки составляют примерно половину всех типовых ошибок и распределены по 12 подклассам первого уровня. Общие и связанные с параллельным выполнением программы – 30% и 7 подклассов, и 20% и 2 подкласса соответственно.
- В целом, результаты согласуются с ожиданиями. По заявлениям экспертов ядра ОС Linux именно типовые ошибки, связанные с состоянием гонки и взаимной блокировки (класс *sync*), на сегодняшний день являются наиболее критическими и сложными для обнаружения, вследствие чего их стремятся исправить как можно быстрее. Помимо них, к наиболее важным ошибкам относят разыменование нулевого указателя (класс *generic:null_ptr_deref*), выход за границу массива (класс *generic:buffer_overflow*) и работу с неинициализированными данными (классы *generic:uninit* и *specific:uninit*).
- Полученные результаты не являются полностью достоверными, поскольку часто определить причину ошибки бывает достаточно трудно. Например, некоторые ошибки *generic:null_ptr_deref* могут быть *generic:uninit*, так как разыменование нулевого указателя происходит вследствие того, что данные не были инициализированы, а в соответствующей области памяти оказалось нулевое значение. Также некоторые типовые общие ошибки на самом деле могут быть типовыми специфичными ошибками, так как общее проявление

может быть обусловлено некорректным использованием интерфейса основной части ядра.

- Следует сделать поправку на специфику формирования стабильных веток ядра ОС Linux. Изменение из оригинальной ветки ядра попадает в стабильную, если автор изменения, лицо, проводившее инспекцию кода, или ответственный за соответствующую подсистему посчитает его достаточно важным и пометит, как требующее рассмотрения для включения в стабильную ветку. При этом большинство авторов изменений считают, что предлагать изменение для включения в стабильную ветку не входит в область их ответственности. С другой стороны, известно, что ответственные за целый ряд подсистем ядра, например, wireless или scsi, также не уделяют внимание необходимости помечать изменения. Как следствие возникает неоднородность в составе изменений, попадающих в стабильные ветки ядра, что могло повлиять на результаты анализа. Также следует отметить, что изменения в драйверах интересны меньшему кругу лиц, и поэтому имеют меньшие шансы попасть в стабильные ветки ядра.

4. Аналогичные исследования

В большом количестве работ ошибки в исходном коде ядра ОС Linux, в том числе, в драйверах, были получены на основании применения различных подходов автоматизированного анализа. Например, в статьях [30] и [31] предложены подходы для автоматического выявления неявных правил, которым должен удовлетворять исходный код анализируемых программ, на основе статистических данных. В отличие от [30], где правила получались только для шаблонов из пар элементов (например, если вызвана некоторая функция, то должна быть вызвана и другая), [31] описывает более общий подход, который затрагивает помимо функций переменные и типы и может выявлять правила для большего количества элементов без каких-либо шаблонов. Применительно к ядру ОС Linux подходы позволили выявить тысячи правил и десятки ошибок. Однако, данные подходы не могут гарантировать, что найденные с их помощью ошибки не являются ложными срабатываниями. Они используют множество эвристик, в особенности, при различных статистических оценках и ранжировании выявленных правил. Поэтому требуется существенный ручной труд по анализу обнаруженных ошибок. Кроме того, подходы не могут выявить все возможные правила и соответствующие им ошибки (в [30] задается несколько фиксированных шаблонов; ни в [30], ни в [31] не сообщается о классах и распределении найденных ошибок), что в принципе не позволяет получить полную классификацию и распределение ошибок даже после их ручного анализа.

В статьях [10] и [13] распределение типовых ошибок в ядре ОС Linux было получено автоматизированным образом с помощью инструментов статического анализа на основе предопределенной классификации. Благодаря

данным исследованиям, в частности, было выявлено, что драйверы содержат до 85% всех ошибок, после чего много усилий было потрачено для повышения их надежности. Данным подходам, также как и подходам, описанным ранее, присуще наличие ложных срабатываний. По этой причине результаты, полученные для некоторых классов ошибок, вообще не принимались во внимание. Кроме того, подходы в значительной степени зависят от того, как инструменты статического анализа проверяют соответствующие классы. В [10] и [13] были использованы различные инструменты статического анализа. Очень вероятно, что именно это послужило причиной значительного расхождения полученных результатов.

Результаты, полученные в рамках данной работы, существенно отличаются от результатов данных исследований. Прежде всего, это связано с тем, что в предложенном подходе типовые ошибки анализируются вручную на основе изменений, вносимых в драйверы ОС Linux, в то время, как в аналогичных подходах ошибки определяются автоматизированным образом на основе некоторых версий исходного кода ядра. Автоматизированные решения позволяют проанализировать существенные объемы исходного кода различных версий ядра за сравнительно небольшое время. Поэтому на практике предпочтение отдается автоматизированным подходам. Кроме того, они являются объективными с точностью до реализации эвристик в используемых инструментах анализа. Ручной подход позволяет проанализировать типовые ошибки в драйверах более тщательно. Поэтому на основании его результатов можно получать достаточно полную классификацию и более точное распределение ошибок. Предложенная методика позволяет анализировать ошибки, проявляющиеся при различных условиях (при использовании ОС Linux, при целенаправленном ручном и автоматизированном поиске ошибок и т.д.) за счет анализа изменений в драйверах, сделанных по мере их разработки.

5. Заключение

В статье исследованы возможные пути по выявлению и классификации типовых ошибок в драйверах операционной системы Linux. Показывается, что анализ изменений, вносимых в драйверы ядра Linux, обладает рядом преимуществ по сравнению с другими источниками.

Для анализа изменений в драйверах ОС Linux предложена методика. Данная методика позволяет отделить исправления типовых ошибок от исправлений нетиповых ошибок и от реализации дополнительной функциональности. Также методика позволяет построить классификацию типовых ошибок в драйверах, причем основной упор делается на причину происхождения ошибки, а не на ее проявление.

В качестве основной области применения предложенной методики были выбраны изменения в стабильных версиях ядра ОС Linux, поскольку процент типовых ошибок среди них наиболее высокий. Были проанализированы

изменения, сделанные в стабильных ядрах с 2.6.35 по 3.0, начиная с 26 октября 2010 года по 26 октября 2011 года. Всего было проанализировано 1503 уникальных изменений в драйверах, из которых 396 оказались исправлениями типовых ошибок. Для них была составлена классификация и определено распределение ошибок по соответствующим классам. На основании полученной картины были сделаны соответствующие выводы.

Подводя итог, можно сказать, что выявлять типовые ошибки на основе изменений в драйверах операционной системы Linux эффективно, поэтому эту работу следует продолжать. Существенно облегчить и ускорить ручную работу по анализу изменений может помочь система, которая позволит анализировать изменения итеративным образом непосредственно по мере их появления в репозитории. Разработчики ядра и драйверов ОС Linux могут поспособствовать проведению классификации путем более четкого определения и описания истинной причины исправляемых ими ошибок.

Литература

[1] J. Corbet, G. Kroah-Hartman, A. McPherson. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.

[2] T. Leemhuis. What's new in Linux 3.3. http://www.h-online.com/open/features/What-s-new-in-Linux-3-3-1466872.html, 2012.

[3] Сайт оригинального ядра ОС Linux. http://www.kernel.org.

[4] J. Corbet. How to Participate in the Linux Community. A Guide To The Kernel Development Process. http://www.linuxfoundation.org/sites/main/files/How-Participate-Linux-Community_0.pdf, 2008.

[5] S. M. Kerner. The Red Hat Enterprise Linux 6 Kernel: What Is It? http://www.serverwatch.com/news/article.php/3880131/The-Red-Hat-Enterprise-Linux-6-Kernel-What-Is-It.htm, 2010.

[6] Ядро дистрибутива openSUSE. http://en.opensuse.org/Kernel.

[7] Ядро дистрибутива Debian. http://wiki.debian.org/DebianKernel.

[8] ОС Linux реального времени. https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html.

[9] ОС Android. http://developer.android.com/guide/basics/what-is-android.html.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and DR Engler. An Empirical Study of Operating System Errors. Proc. 18th ACM Symp. Operating System Principles, 2001.

[11] M. Swift, B. Bershad, H. Levy. Improving the reliability of commodity operating systems. In: SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[12] A. Ganapathi, V. Ganapathi, D. Patterson. Windows XP kernel crash analysis. Proceedings of the 2006 Large Installation System Administration Conference, 2006.

[13] N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, and Gilles Muller. Faults in linux: ten years later. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11), USA, 2011.

[14] S. Butt, V. Ganapathy, M.M. Swift, C.-C. Chang. Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. Computer Security Applications Conference (ACSAC '09), 2009.

[15] D. Tian, X. Xiong, C. Hu, P. Liu. Policy-centric protection of OS kernel from vulnerable loadable kernel modules. Proceedings of the 7th international conference on Information security practice and experience, 2011.
[16] E.S. Raymond. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA, USA, 2001.
[17] R.L. Glass. Facts and Fallacies of Software Engineering. AddisonWesley, Professional, Sebastopol, CA, USA, 2003.
[18] ISO/IEC TR 24772. Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use, 2010.
[19] G. Kroah-Hartman. The Linux kernel driver interface. http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.
[20] Документация ядра ОС Linux. http://kernel.org/doc/Documentation.
[21] A. Rubini. Linux Device Drivers (Nutshell Handbooks). O'Reilly Media, 1st edition, February 24, 1998.
[22] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner. Linux Kernel Internals. Addison-Wesley Professional, 2 edition, December 16, 1997.
[23] Список рассылки ядра ОС Linux. https://lkml.org.
[24] Проиндексированный исходный код ядра ОС Linux различных версий. http://lxr.linux.no.
[25] R. Russell. Unreliable Guide To Hacking The Linux Kernel. http://www.kernel.org/doc/htmldocs/kernel-hacking.html, 2005.
[26] Репозиторий стабильных версий ядра ОС Linux. http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=summary.
[27] M. K. Saad. Browsing Linux Kernel. Linux Day, May 6, 2007.
[28] G. Kroah-Hartman. Kernel development statistics for 2.6.35. http://lwn.net/Articles/395961, 2010.
[29] Результаты анализа изменений в драйверах ОС Linux. http://linuxtesting.org/downloads/ldv-commits-analysis-2012.zip.
[30] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. ACM SIGOPS Operating Systems Review, v.35 n.5, December, 2001.
[31] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. ACM SIGSOFT Software Engineering Notes, v.30 n.5, September, 2005.

Analysis of typical faults in Linux operating system drivers

Mutilin V.S., Novikov E.M., Khoroshilov A.V.
mutilin@ispras.ru, novikov@ispras.ru, khoroshilov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Fast evolution of the Linux operating system kernel and drivers, developed by a big programmers community distributed all over the world, led to nowadays there is not a common base of rules that completely describe a correct interaction between drivers and the kernel. This is an obstacle both for programmers that do not have expert knowledge in all peculiarities of the given interaction, and for development and application of tools that could find corresponding typical faults in the automatic way. The given paper presents a method to detect and to classify typical faults and corresponding rules. This method is based on analysis of changes, made to Linux operating system drivers.

The paper gives results of the method application to stable versions of the Linux kernel from 2.6.35 till 3.0 starting from October 26, 2010 till October 26, 2011. We analyzed in total 1503 unique commits to drivers, marked 396 (about 27%) of them as fixes of typical faults and provided a classification and a distribution by classes for these typical faults. We distinguished 3 classes of typical faults: generic (faults all C programs are subjected to), specific (faults related to misuses of the Linux kernel API) and synchronization (faults related to parallel execution). We found that specific faults constitute about 50% of all typical faults. Then each typical fault was ascribed to one of 21 first-level subclasses: 12, 7 and 2 for specific, generic and synchronization classes correspondingly. We found that more than 50% of typical faults correspond to 5 first-level subclasses. More than 80% of typical faults correspond to 14 first-level subclasses. The most frequent faults were race conditions during parallel execution – they account for about 17% of all typical faults. Second and third places with about 9% were occupied by leaks of specific resources and null pointer deference.

Keywords: operating system; kernel; driver; interaction rule; faults classification.

References

[1]. Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.
[2]. Leemhuis T. What's new in Linux 3.3. http://www.h-online.com/open/features/What-s-new-in-Linux-3-3-1466872.html, 2012.
[3]. Vanilla Linux kernel. http://www.kernel.org.
[4]. Corbet J. How to Participate in the Linux Community. A Guide To The Kernel Development Process. http://www.linuxfoundation.org/sites/main/files/How-Participate-Linux-Community_0.pdf, 2008.
[5]. Kerner S.M. The Red Hat Enterprise Linux 6 Kernel: What Is It? http://www.serverwatch.com/news/article.php/3880131/The-Red-Hat-Enterprise-Linux-6-Kernel-What-Is-It.htm, 2010.
[6]. openSUSE kernel. http://en.opensuse.org/Kernel.
[7]. Debian kernel. http://wiki.debian.org/DebianKernel.

[8]. Real-Time Linux. https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html.
[9]. Android operating system. http://developer.android.com/guide/basics/what-is-android.html.
[10]. Chou A., Yang J., Chelf B., Hallem S., Engler D. An Empirical Study of Operating System Errors. In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2001. doi: 10.1145/502034.502042
[11]. Swift M., Bershad B., Levy H. Improving the reliability of commodity operating systems. In Proc. 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2003. doi: 10.1145/502034.502042
[12]. Ganapathi A., Ganapathi V., Patterson D. Windows XP kernel crash analysis. In Proc. 20th Conference on Large Installation System Administration (LISA), pp. 149-159, 2006.
[13]. Palix N., Thomas G., Saha S., Calves C., Lawall J., Muller G. Faults in linux: ten years later. In Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 305-318, 2011. doi: 10.1145/1950365.1950401
[14]. Butt S., Ganapathy V., Swift M.M., Chang C.-C. Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In Proc. Computer Security Applications Conference (ACSAC), pp. 301-310, 2009. doi: 10.1109/ACSAC.2009.35
[15]. Tian D., Xiong X., Hu C., Liu P. Policy-centric protection of OS kernel from vulnerable loadable kernel modules. In Proc. Information Security Practice and Experience, LNCS, vol. 6672, pp. 317-332, 2011. doi: 10.1007/978-3-642-21031-0_24
[16]. Raymond E.S. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly Media, 1999.
[17]. Glass R.L. Facts and Fallacies of Software Engineering. Addison Wesley, 2002.
[18]. ISO/IEC TR 24772. Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use, 2010.
[19]. Kroah-Hartman G.. The Linux kernel driver interface. http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.
[20]. Linux kernel documentation. http://kernel.org/doc/Documentation.
[21]. Rubini A. Linux Device Drivers (Nutshell Handbooks). O'Reilly Media, 1998.
[22]. Beck M., Bohme H., Dziadzka M., Kunitz U., Magnus R., Verworner D. Linux Kernel Internals. Addison-Wesley, 1996.
[23]. Linux kernel mailing list. https://lkml.org.
[24]. Indexed source code of various versions of the Linux kernel. http://lxr.linux.no.
[25]. Russell R. Unreliable Guide To Hacking The Linux Kernel. http://www.kernel.org/doc/htmldocs/kernel-hacking.html, 2005.
[26]. Repository of stable versions of the Linux kernel. http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=summary.
[27]. Saad M.K. Browsing Linux Kernel. Linux Day, 2007.
[28]. Kroah-Hartman G. Kernel development statistics for 2.6.35. http://lwn.net/Articles/395961, 2010.
[29]. Results of analysis of commits to Linux drivers. http://linuxtesting.org/downloads/ldv-commits-analysis-2012.zip.
[30]. Engler D., Chen D.Y., Hallem S., Chou A., Chelf B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), vol. 35, issue 5, pp. 57-72, 2001. doi: 10.1145/502034.502041

[31]. Li Z., Zhou Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), vol. 30, issue 5, pp. 306-315, 2005. doi:10.1145/1081706.1081755

Автоматизация построения расписаний для периодических систем реального времени

Третьяков Андрей
andrromeda@ispras.ru

Аннотация. В статье рассматривается проблема построения расписаний для систем, имеющих жёсткие ограничения по времени работы. Это является одной из основных проблем при проектировании систем авионики. В работе проанализированы существующие алгоритмы, выделены их недостатки, и предложено собственное решение, удовлетворяющее необходимым требованиям. Кроме того, разработаны инструменты для визуализации, редактирования и валидации создаваемых расписаний.

1. Введение

Вычислительные системы в промышленности и технике используются для решения широкого спектра разнообразных задач. Чаще всего эти задачи связаны с важными производственными функциями, от которых зависит работоспособность системы в целом. Компьютерные системы обычно контролируют какое-нибудь устройство или физический процесс, собирают данные о нём, выполняют необходимые вычисления и отвечают на внешние физические события путём отправки соответствующих управляющих воздействий. Причём, поскольку процесс получения и обработки информации зачастую является непрерывным, система должна успевать реагировать на внешние события. То есть, задачи должны выполняться за такие промежутки времени, которые обеспечивают успешное решение всех поставленных перед системой задач. Такие системы, гарантирующие своевременную реакцию на внешние события, называются **системами реального времени** (СРВ). Существует несколько эквивалентных определений систем реального времени [7, 8, 9], в данной работе под этим термином будут подразумеваться системы, которые производят вычисления и решают свои задачи в заданные ограниченные промежутки времени. Если задача не успевает к требуемому моменту времени вычислить результат, последствия могут быть различной степени тяжести, вплоть до выведения системы из строя, что может привести к катастрофе. Поэтому системы реального времени проектируются особо тщательно, и задачи, которые они будут выполнять, просчитываются заранее, исходя из их временных свойств и заданных ограничений.

Если несколько событий происходят в близкие промежутки времени, системе необходимо распланировать выполнение своих вычислений так, чтобы ответ системы на каждое событие поступал в заданные временные рамки. Для этого обычно используется планировщик, который и определяет, как именно и в каком порядке должны решаться задачи. Во многих системах планировщик работает по заранее составленному **расписанию**, определяющему политику планирования.

Составление расписаний может оказаться очень большой проблемой, сложность которой зависит как от параметров системы и ограничений на целевые задачи, так и от количества задач. В работе показывается, как изменение одного, на первый взгляд несущественного, условия может превратить задачу построения расписания из задачи, для которой существуют классические полиномиальные алгоритмы, в NP-трудную задачу. При планировании задач для небольших систем составлять расписание можно любым способом, в том числе и вручную или с использованием каких-либо эвристических методов, – главное, чтобы оно в итоге оказалось корректным. Однако на практике для систем, насчитывающих сотни и тысячи задач, или для которых приходится часто переделывать расписания, требуется автоматизировать этот процесс.

2. Особенности целевой системы

Одной из сфер применения систем реального времени является авиастроение, причём особого внимания заслуживает бортовая система автоматического управления самолётом. Она выполняет ряд задач высокой степени важности по измерению микроклиматических условий окружающей среды (температура, давление), параметров полёта (высота, скорость) и по управлению некоторыми отдельными устройствами (например, подкрылками самолёта). Кроме того, она выполняет и другие функции (например, автомат тяги, автопилот, предупреждение экипажа, техническое обслуживание). Без сомнения, эти функции являются задачами «жёсткого» реального времени [1], так как если они не успеют отреагировать на внешние события в заданные промежутки времени, это может привести к нарушению работы бортовой системы и потенциально к крушению самолёта. Поэтому в качестве целевой системы реального времени выберем бортовую систему автоматического управления самолётом, которую и будем в дальнейшем рассматривать.

Прежде чем выяснить особенности выбранной системы, введём несколько необходимых определений. В зарубежной литературе срок, до которого задача должна выполниться, называется *deadline*, что может быть переведено как *критический срок обслуживания* или *предельный срок выполнения* [8]. Кроме этого, в большинстве систем задачи могут иметь также следующие свойства [1]:

- время запуска (или время готовности) – момент времени, в который задача приводится в состояние готовности и ожидает выполнения;
- минимальная (максимальная) задержка – минимальное (максимальное) количество времени, которое должно пройти между временем готовности и моментом времени фактического начала выполнения задачи;
- наибольшее время выполнения (или максимальное время отклика) – максимальное время, необходимое для завершения задачи;
- время выполнения (или длительность) – время, взятое без прерываний, необходимое для выполнения задачи;
- приоритет (или вес) – относительная важность задачи.

В связи с усложнением программного обеспечения в области авиастроения, в последнее время стало целесообразным возлагать на один процессорный модуль авиационные функции, выполняемые ранее несколькими модулями, при этом улучшив массогабаритные характеристики системы управления и снизив ее стоимость. Такая тенденция в авиационной технике привела к появлению концепции интегрированной модульной авионики – ИМА (Integrated Modular Avionics, IMA). Отсюда возникает проблема разделения ставших теперь общими ресурсов (процессорное время, память, каналы обмена) между различными приложениями. В авионике ключевую роль в решении данной проблемы играют операционные системы реального времени, работающие по стандарту ARINC 653 [10]. Поскольку от некоторых функций зависит безопасность полёта, для них ресурсы, в первую очередь, процессорное время, распределяются планировщиком между задачами особенно тщательно, по заранее определённой схеме. То есть, в этих системах, исходя из временных характеристик задач и других их свойств, предварительно составляется **расписание** вычислений системы, используя которое, планировщик распределяет во времени решаемые задачи так, что они гарантированно удовлетворяют всем временным ограничениям. При этом сам процесс составления расписания происходит за рамками системы реального времени, и потому является некритичным по времени. Кроме того, заранее становится известно, возможно ли в принципе составить расписание для данного набора задач, до того, как система реального времени будет введена в эксплуатацию.

Понятие «авиационной функции» близко к понятию «процесса» в обычной пользовательской операционной системе, принципы организации и реализации также схожи. Поэтому авиационные функции, периодически запускающиеся в бортовой системе, в данной работе будем называть процессами, а каждый конкретный запуск – итерацией процесса.

Итак, все процессы распределяются по нескольким процессорным модулям, на каждом модуле выполняется набор взаимодействующих процессов, для

которого и составляется расписание. Между процессами из различных модулей зависимости не учитываются. Поэтому можно считать, что мы имеем дело с однопроцессорной системой, так как для других процессоров будет составляться другое расписание аналогичным образом.

Планировщик задач в данной системе срабатывает через некоторые фиксированные промежутки времени (порядка 0,25 мс), называемые **тиками** процессора. Таким образом, тик – это минимальная распределяемая единица времени в расписании, и поэтому все числовые параметры процессов измеряются в тиках процессора.

Бортовая система самолёта работает с множеством авиационных функций, среди которых есть и периодические, и апериодические, и спорадические [1]. Апериодические процессы, важность которых не особо критична, выполняются на отдельном процессорном модуле, и в данной работе рассматриваться не будут. Для построения расписаний для спорадических процессов существуют эффективные классические алгоритмы, удовлетворяющие всем особенностям бортовой системы самолёта. Наибольший интерес представляют периодические процессы, размещаемые на одном процессоре.

Требование периодичности процессов следует рассмотреть подробнее, так как его можно определить по-разному. В некоторых работах [1, 2, 3] под периодом процесса понимается промежуток времени между двумя его последовательными моментами времени ***готовности***, в которые процесс переводится из состояния ожидания в состояние готовности и ожидает начала выполнения. То есть, процесс должен запуститься и завершиться в промежутке между очередными моментом времени готовности и предельным сроком выполнения (*deadline*). Причём момент фактического начала работы процесса может быть несколько позже времени готовности (см. рисунок 1).

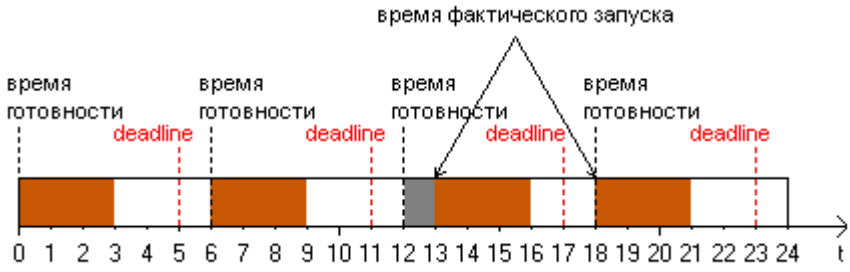


Рисунок 1

На рисунке 1 в качестве примера приведён процесс с периодом 6 и предельным сроком выполнения, равным 5 единицам времени. Причём 3-я

итерация процесса запустилась не в 12, а в 13 момент времени из-за другого процесса (выделен серым цветом).

Из авиационной практики известно, например, что если через определённые промежутки времени не срабатывают функции управления элеронами и закрылками самолёта, то крыло может отвалиться. Поэтому ряд экспертов в области авиастроения выдвигают более жёсткие требования по периодичности процессов. А именно, процесс должен запускаться в строго периодичные моменты времени и отрабатывать хотя бы одну единицу времени. То есть, время фактического запуска должно совпадать со временем готовности. Иными словами, максимальная задержка должна быть сведена к нулю.

Таким образом, расписание, представленное на рисунке 1, не является корректными с точки зрения периодичности процесса, а на рисунке 2 – является:



Рисунок 2

Требование строгой периодичности компенсируется следующим условием: процессы могут быть прерваны в любой момент (кроме начального, то есть времени запуска), любое число раз, другими процессами. После завершения прерывающего процесса прерываемый должен доработать оставшееся его время. При этом накладные расходы процессора на переключение между процессорами не учитываются, они уже включены в длительности процессов.

При прерывании процесса увеличивается его максимальное время отклика, и существует вероятность, что один или несколько последних фрагментов нарушат предельный срок выполнения. Многие авиационные функции работают с неразделяемыми ресурсами, то есть такими ресурсами, которые доступны только одной итерации конкретного процесса. Поэтому случай, когда какой-либо фрагмент k-ой итерации процесса выполняется после того, как началась (k+1)-ая итерация, является недопустимым. Во избежание этого относительный предельный срок выполнения устанавливается равным периоду процесса. Это условие рассматриваемую систему как раз и делает системой реального времени.

Следует подчеркнуть, что длительность каждого процесса в каждом периоде также должна строго соответствовать заданному значению.

Расписание, в котором для всех процессов выполняются вышеперечисленные условия, будем называть *корректным* (или *допустимым*). Остаётся добавить, что хотя накладные расходы на переключение между процессами не учитываются в явном виде, само число переключений требуется сократить. Корректное расписание, в котором число переключений процессора минимальное для данных входных параметров, будем называть *оптимальным*.

Резюмируя всё вышесказанное, сформулируем требования к целевой системе:

- Система является системой ***«жёсткого»*** реального времени.
- Рассматривается один процессор, на котором запускается *n* процессов.
- Процессы могут прерываться в любой момент и любое число раз.
- Процессы должны завершаться до запуска своей следующей итерации.
- Для каждого процесса должны **строго** соблюдаться заданные входные параметры:
    - фиксированный период фактического запуска (начала работы);
    - суммарная длительность процесса в каждом периоде.
- Зависимости процессов и накладные расходы на переключение не учитываются; все параметры измеряются в целых числах.
- Количество переключений между процессами должно быть минимальным.

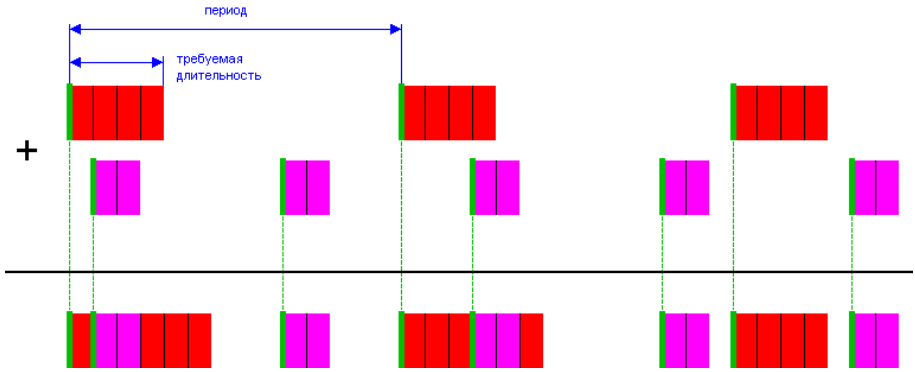


Рисунок 3

На рисунке 3 показаны в качестве примера два периодических процесса, как они запускались бы каждый на отдельном процессоре, и расписание их работы на одном общем процессоре. Каждый процесс запускается строго

периодически, даже если он в следующий момент времени будет прерван другим процессом.

3. Понятие расписания для периодической системы реального времени

Расписание для системы реального времени, на которую накладываются дополнительные требования, описанные в предыдущем разделе, построить весьма непросто, так как приходится учитывать множество ограничений. В различных авиационных компаниях поступают по-разному: в одних подобные расписания строят вручную, в других используют различные закрытые разработки. Однако на практике вручную строить расписания нерационально, так как число и характеристики процессов могут постоянно варьироваться, для больших значений параметров приходится делать много рутинной работы, причём очень легко можно допустить ошибку. Более того, для некоторых входных данных в принципе нельзя построить расписание, удовлетворяющее всем ограничениям. Такие же проблемы возникают при необходимости проверить, действительно ли построенное расписание удовлетворяет всем поставленным требованиям. Сложнее всего дело обстоит с поиском оптимального расписания, так как приходится гарантировать, что не может быть найдено расписания с меньшим количеством переключений процессора. Из всего вышесказанного следует, что оптимальным решением проблемы будет инструмент, позволяющий персоналу, ответственному за проектирование систем авионики (так называемым системным интеграторам), проектировать расписания в автоматическом или полуавтоматическом режиме.

Для анализа алгоритмов построения расписаний формализуем понятия периодической системы реального времени и расписания для неё. В общем случае, периодическую систему реального времени можно формально определить следующим образом. Периодическая система T, состоящая из n процессов, задаётся множеством своих процессов T_i, каждый из которых определяется четвёркой: $T_i = (e_i, d_i, p_i, s_i)$, $1 \le i \le n$, где:

- e_i – длительность i-ого процесса, то есть время работы, взятое без прерываний;
- d_i – относительный предельный срок выполнения i-ого процесса, то есть промежуток времени между очередными моментом времени готовности и предельным сроком выполнения;
- p_i – период i-ого процесса, то есть промежуток времени между двумя последовательными моментами времени готовности;
- s_i – начальное смещение i-ого процесса, то есть время начала первой итерации, первый момент времени готовности.

То есть, для k-го вызова i-ого процесса заданы следующие временные ограничения: должно существовать e_i моментов времени t, таких, что

$$s_i + (k-1)p_i \le t < s_i + (k-1)p_i + d_i \quad (1)$$

где k – натуральное число (натуральные числа, напомним, не включают 0) [3].

Для этих параметров процессов выполняются следующие очевидные соотношения:
$0 < e_i \le d_i \le p_i$, где $1 \le i \le n$. Для смещения чаще всего выполняется $s_i \le d_i - e_i$.

Для простоты все числовые параметры будем полагать целыми, поскольку, как показывается в работе [3], для периодических систем реального времени случай расписаний с действительными параметрами может быть сведён к случаю с дискретными параметрами.

Если задаются начальные смещения, то системы называются **полными**, в противном случае – **неполными**. То есть, для неполных систем процессы задаются тройкой:

$T_i = (e_i, d_i, p_i)$, где $1 \le i \le n$.

В рассматриваемой авиационной системе начальные смещения процессов не задаются, поэтому система является неполной. Одной из задач алгоритма составления расписания как раз и является определение начальных смещений.

Расписание для однопроцессорной системы, в свою очередь, можно формально определить как числовую функцию S от натурального переменного t:
S: $N \to \{0, 1, \ldots, n\}$, то есть, для каждого момента времени определяющая, какой из n процессов работает, или возвращающая 0, если ни один из процессов не запущен в данный момент. Таким образом, расписание S будем называть корректным, если для любого натурального k и $i \in \{1, \ldots, n\}$ существует e_i моментов времени t таких, что $S(t) = i$ и выполняется условие (1).

Собственно, проблему нахождения разбиения итераций процессов для исходного набора процессов и называют построением расписания (см. рис. 3).

Очевидно, что в общем случае корректное расписание для системы не единственно. При составлении расписаний обычно максимизируют или минимизируют некоторую целевую функцию, например, количество процессоров, выполняющих задачи, или размер расписания, или общее количество работ, или суммарное запаздывание, и так далее. В данном случае это число переключения процессора между процессами. Расписание, для которого целевая функция достигает своего экстремума, мы будем называть оптимальным. Однако, в зависимости от свойств системы, даже оптимальное расписание может быть не единственным. Или, даже если оно единственное,

получить его можно разными способами. Можно, например, составить его вручную, исходя из логических умозаключений. Или, если требования к системе не слишком жёсткие, попытаться сгенерировать расписание полуавтоматическим образом, а затем проверить, выполняются ли все требования системы. Во многих системах используются инструменты автоматического построения расписания, работающие по какому-либо алгоритму, подобранному для конкретной системы. На практике может применяться любой из описанных способов, и в каждом отдельном случае выбирается тот или иной подход.

4. Существующие решения

4.1. Циклические расписания

Для построения расписаний существуют различные алгоритмы.

Однако в большинстве печатных изданий рассматриваются либо непериодические системы, либо системы, не являющиеся системами реального времени. Например, в статьях [4, 5] авторов Peter Brucker и Thomas Kampmeyer даётся следующее определение: расписание для системы из n процессов называется периодическим с циклом α, если для каждого i-го процесса момент времени $t(i; k)$ начала k-ой его итерации определён формулой:

$t(i; k) = t(i; 0) + \alpha k$, где $1 \le i \le n$, k – целое.

То есть, периоды всех процессов одинаковы и равны α. Такую систему они называют *циклической*.

Тем не менее, периодическую систему (с разными периодами процессов), которая рассматривается в этой работе, можно свести к циклической (с одинаковыми периодами).

Но прежде мы определим, какова должна быть длина (размер) расписания, достаточная для функционирования системы, в которой бесконечно запускаются процессы с определённым периодом. Пусть мы имеем периодическую систему из n задач $T = \{1, \ldots, n\}$, с соответствующими периодами процессов $p_1, \ldots, p_n$ и их начальными смещениями $s_1, \ldots, s_n$. Тогда k-ый запуск процесса i происходит в момент времени $s_i + (k - 1)p_i$. Допустим, удалось построить расписание, в котором нет ни одной коллизии процессов. Возьмём длину этого расписания, равную наименьшему общему кратному всех периодов процессов, $L = \text{НОК}(p_1, \ldots, p_n)$, и назовём его первым циклом расписания. Пусть эти процессы за весь первый цикл расписания запускаются соответственно $k_1, \ldots, k_n$ раз, где $k_i = L / p_i$. Второй цикл расписания начинается непосредственно за первым. Тогда для каждого процесса его $(k_i + 1)$-ый запуск (т.е. уже во втором цикле расписания) будет в момент времени $s_i + ((k_i + 1) - 1)p_i = s_i + k_i p_i = s_i + L / p_i * p_i = s_i + L$. Таким образом, смещения каждого процесса относительно начала второго цикла расписания равно смещению относительно первого цикла. Далее процессы продолжают

запускаться с тем же периодом. То есть второй цикл расписания полностью идентичен первому. Промежуток времени между последним запуском i-го процесса в первом цикле и первым запуском во втором равен $(s_i + L) - (s_i + (k_i - 1)p_i) = (s_i + ((k_i + 1) - 1)p_i) - (s_i + (k_i - 1)p_i) = (s_i + k_ip_i) - (s_i + (k_i - 1)p_i) = k_ip_i - (k_i - 1)p_i = p_i$. Таким образом, мы доказали, что требуемая длина расписания может быть равна НОК($p_1, \ldots, p_n$). В [1, 2] показано, что меньше она быть не может. Рисунок 4 иллюстрирует цикличность расписания:

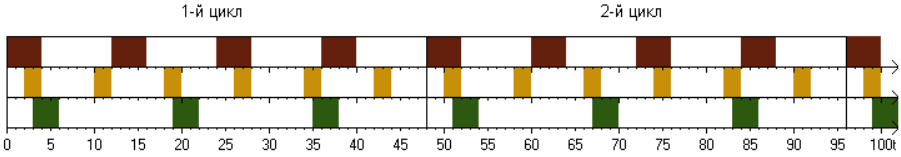


Рисунок 4

На рисунке 4 в качестве примера приведены 3 процесса с периодами 12, 8, 16 и начальными смещениями 0, 2, 3 соответственно. Длина расписания равна НОК(12, 8, 16)=48. Из рисунка видно, что второй цикл полностью повторяет первый, причём на их границе периодичность процессов не нарушается.

Сведём периодическую систему к циклической. Обозначим каждую итерацию каждого процесса – новым процессом. Новые процессы будут отличаться между собой только начальным смещением. Занумеруем все новые процессы за один цикл расписания сквозной нумерацией. Получим $k_1k_2\ldots k_n$ процессов, у каждого из которых уникальное начальное смещение. Во втором цикле расписания сделаем то же самое. Тогда получим, что каждый новый процесс (в старых обозначениях (i, j), где $1 \le j \le k_i$) будет запускаться периодически с периодом L. То есть, мы пришли к циклической системе.

Таким образом, построив один цикл расписания, мы строим расписание для всей шкалы по времени. Однако это не даёт ответа на вопрос, каким образом можно построить первый цикл расписания, то есть, как определить циклическую систему. А в работах [4, 5] она уже полагается заданной.

4.2. Алгоритмы Rate-Monotonic и Earliest Deadline First

Наиболее близкими алгоритмами к рассматриваемой сфере являются классические алгоритмы Rate-Monotonic (RM) и Earliest Deadline First (EDF) [1]. Оба они основываются на понятии *приоритета* процесса – целого положительного числа, определяющего относительную срочность; чем меньше число, тем выше считается приоритет процесса. Алгоритмы проходят временную шкалу от 0 до L и в каждый момент времени определяют, сколько процессов находятся в состоянии готовности. Если найден только один

процесс, то он работает согласно своей периодичности, либо завершается один из его фрагментов после того, как был завершён более приоритетный процесс. Если нет ни одного, то в расписании образуется свободное место (*idle time*). Если же несколько процессов находятся в состоянии готовности, выбирается тот из них, чей приоритет выше. Для полных систем алгоритмы учитывают начальные смещения процессов, а для неполных полагают их равными 0.

Различие между алгоритмами заключается, в основном, в том, что Rate-Monotonic подразумевает, что процессы имеют фиксированный приоритет (если приоритеты не задаются в условии, то алгоритм их задаёт сам: чем меньше относительный предельный срок выполнения процесса, тем его приоритет выше), а Earliest Deadline First – что динамический. То есть, приоритет разных фрагментов одной и той же итерации процесса может различаться в разные моменты времени. EDF полагает, что чем ближе предельный срок выполнения для данного процесса, тем его приоритет выше. Другими словами, на каждом шагу (т.е. в каждый момент времени), при анализе списка работающих в данный момент процессов, алгоритм RM выбирает тот из них, чей относительный *deadline* численно меньше, а EDF – тот, чей очередной абсолютный *deadline* ближе по времени.

И RM и EDF для периодических систем имеют алгоритмическую сложность $O(N^2)$ в худшем случае, где N – суммарное число итераций всех процессов за время одного цикла расписания [1], то есть, они являются полиномиальными.

Кроме того, алгоритм EDF является оптимальным в том смысле, что может нарушить предельный срок выполнения какого-либо процесса только в том случае, если никакой другой алгоритм построения расписания для системы реального времени не может выполнить задачу в срок [1].

И, наконец, алгоритмы RM и EDF довольно просты в реализации, различные их варианты и модификации описаны во многих работах [1, 2, 3, 11].

Однако они имеют очень существенный недостаток, который перечёркивает все их достоинства. Они не отвечают требованиям системы в том виде, в котором они поставлены в разделе 2, а именно: нарушается требование строго периодичного запуска процессов. То есть, в случае, когда в некоторый момент времени должен начать работать один процесс при запущенном втором процессе с более высоким приоритетом, предпочтение будет отдано второму. То есть, нередки ситуации, отражённые на рисунке 1, что является недопустимым.

Кроме того, RM и EDF вовсе не гарантируют, что число переключений процессора минимальное из всех возможных. Зачастую, задав начальные смещения, отличные от 0, можно получить лучше результат.

4.3. Приближённый алгоритм

В статье С. Зеленова [11] для решения этой задачи используется другой подход: предложен алгоритм, который строит расписание, близкое к оптимальному с точки зрения минимизации общего количества прерываний задач. Этот подход основан на выборе таких начальных смещений, для которых наиболее вероятно будет происходить непрерывное выполнение процессов. Такой алгоритм выполняется очень быстро, позволяет получить результат до 25 прерываний на 1000 тиков процессора, что во многих случаях вполне приемлемо. Однако часто можно получить и более оптимальный результат.

5. Система автоматического составления расписаний

5.1. Архитектура системы

В ИСП РАН разрабатывается система инструментов для автоматизации различных операций по составлению расписаний для системы реального времени, на которую накладываются дополнительные ограничения, изложенные в разделе 2. Параметры авиационных функций содержатся базе данных бортовой системы самолёта, копия которой локально сохранена на компьютере системного интегратора. Работая с ней с помощью этих инструментов, он создаёт расписания для каждого процессорного модуля, которое затем сохраняется в эту же базу данных и впоследствии используется в самом самолёте.

Разрабатываемая система получила название «Рабочее место системного интегратора ИМА» (*англ*. Modular Avionics System Integrator Workplace, MASIW) и основана на каркасе приложений (*framework*), предоставляемом популярной кроссплатформенной расширяемой средой разработки Eclipse. Схема работы с системой представлена на рисунке 5:

В рамках данной работы система работает с параметрами авиационных функций (процессов), сохранёнными в специальном проекте Eclipse (своего рода тоже локальной базе данных, только сильно упрощённой), которые загружаются из локальной базы данных бортовой системы самолёта. Эти параметры проверяются на согласованность и передаются на вход алгоритму (одному или нескольким), а на выходе получается расписание для этих процессов, которые сохраняются в этот же проект Eclipse. Позже система позволяет открыть полученные расписания и отобразить их в графическом или табличном виде, там же произвести исправления, если таковые окажутся необходимыми ответственному персоналу, и проверить на корректность модифицированную версию расписания. Затем окончательное расписание сохраняется в базу данных бортовой системы самолёта, но это уже выходит за рамки данной работы.

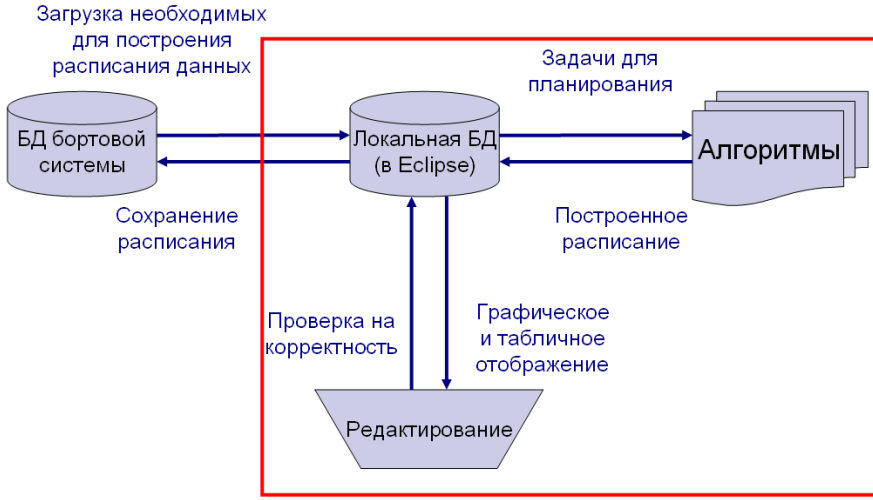


Рисунок 5

5.2. Задача построения расписаний

Дадим формальное определение рассматриваемой системы реального времени и корректного расписания для данного конкретного случая.

В разделе 3 было дано формальное определение более общей системы. Теперь учтём, что в данной системе относительный предельный срок выполнения равен периоду процесса (то есть $p_i = d_i$ для всех i, $1 \le i \le n$, где n – число процессов в системе), а также начальные смещения процессов не заданы (т.е. есть система неполная). Тогда система из n процессов задаётся как $T = \{T_1, \ldots, T_n\}$, где T_i $(1 \le i \le n)$ задаётся следующим образом: $T_i = (e_i, p_i)$, причём $0 < e_i \le p_i$ для всех $i \in \{1, \ldots, n\}$. Начальные смещения s_i алгоритм должен выбрать самостоятельно.

Формула (1) преобразуется к следующему виду:

$$s_i + (k-1)p_i \le t < s_i + kp_i \qquad (2)$$

где k – натуральное.

В разделе 4.1 было показано, что для периодической системы реального времени длина расписания равна $L = \text{НОК}(p_1, \ldots, p_n)$.

Расписание $S(t)$ как функция натурального переменного t $(S: N \to \{0, 1, \ldots, n\})$ называется корректным, если для любого натурального k и $i \in \{1, \ldots, n\}$ существует **ровно** e_i моментов времени t таких, что $S(t) = i$ и выполняется условие (2), причём $S(s_i + (k-1)p_i) = i$ (то есть, момент запуска каждой итерации **строго** периодичен).

5.2.1. Необходимые условия существования расписания

Как уже было сказано, не для любых наборов процессов в принципе возможно построить расписание.

1) Самое очевидное ограничение на входные данные следующее: для каждого процесса его период должен быть не меньше его длительности:

   $0 < e_i \le p_i$ (*)
   для всех $i \in \{1, \ldots, n\}$.

   Иначе предельные сроки выполнения заведомо нарушаются даже в случае одного процесса в системе.

2) Второе условие является своего рода обобщением первого, и смысл его заключается в том, чтобы суммарная длительность всех процессов с учётом числа их запусков была не больше длины расписания. То есть:

   $$\sum_{i=1}^{n} \sum_{j=1}^{k_i} e_i \le L,$$

   где k_i – число запусков i-го процесса за один цикл расписания. Упростим это выражение. Так как $L = \text{НОК}(p_1, \ldots, p_n)$, то $k_i = L / p_i = \text{НОК}(p_1, \ldots, p_n) / p_i$, а суммарная длительность i-го процесса за один цикл равна $e_i * \text{НОК}(p_1, \ldots, p_n) / p_i$. Таким образом, имеем:

   $$\sum_{i=1}^{n} \frac{НОК(p_1,..., p_n)}{p_i} e_i \le НОК(p_1,..., p_n)$$

   .

   Разделив обе части неравенства на $\text{НОК}(p_1, \ldots, p_n)$, имеем:

   $$\sum_{i=1}^{n} \frac{e_i}{p_i} \le 1$$ (**)

3) Последнее, самое неочевидное, условие вытекает из соотношения Безу для наибольшего общего делителя (НОД) двух чисел:

   Для любых двух натуральных чисел a и b существуют такие целые числа k_1 и k_2, что наибольший общий делитель a и b представляется в виде:

   $\text{НОД}(a, b) = k_1 a + k_2 b.$ (3)

Очевидно, что если $a > 0$ и $b > 0$, то в выражении (3) один из коэффициентов отрицательный. Пусть для определённости это будет k_2, тогда изменив знак перед ним, будем считать оба коэффициента положительными. Теперь возьмём в качестве a и b взаимно простые числа p_1 и p_2, являющихся периодами двух процессов, получим:

$$k_1 p_1 - k_2 p_2 = 1.$$

Умножим это соотношение на $(s_2 - s_1)$, где s_1 и s_2 – начальные смещения процессов:

$$(s_2 - s_1)k_1 p_1 - (s_2 - s_1)k_2 p_2 = (s_2 - s_1).$$

Обозначим $k'_1 = k_1(s_2 - s_1)$, $k'_2 = k_2(s_2 - s_1)$ и перенесём слагаемые:

$$k'_1 p_1 - k'_2 p_2 = (s_2 - s_1);$$
$$s_1 + k'_1 p_1 = s_2 + k'_2 p_2.$$

В обеих частях равенства стоит формула момента запуска процесса. Мы получили, что если периоды p_1 и p_2 являются взаимно простыми числами, то какие бы начальные смещения s_1 и s_2 мы ни выбрали, найдутся такие коэффициенты k'_1 и k'_2, что соответствующие итерации этих процессов будут начинаться в один и тот же момент, что недопустимо по причине единственности процессора и строгой периодичности процессов. Поэтому 3-е условие формулируется следующим образом:

Для каждой пары процессов периоды не должны являться взаимно простыми числами. (***)

Здесь следует отметить, что требуется отсутствие именно **попарной** взаимной простоты периодов. Если во всей системе нашлось хотя бы 2 процесса с взаимно простыми периодами, составить расписание для системы уже будет невозможно. Однако в **совокупности** периоды процессов могут быть взаимно простыми, то есть допустимо, если НОД(p_1, …, p_n) = 1. Пример успешно построенного расписания для периодов 6, 10 и 15 приведён на рисунке 6:

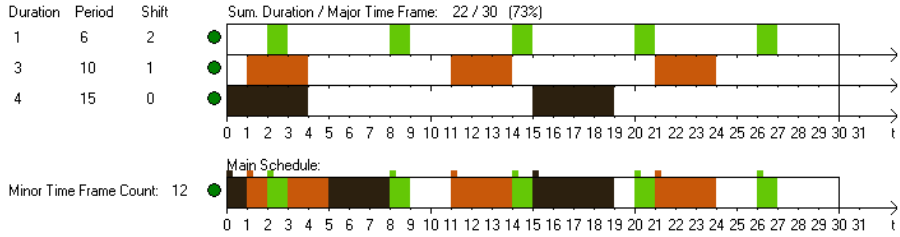


Рисунок 6

На рисунке 6 в верхней части показано взаимное расположение процессов, как они бы запускались на отдельных процессорах. В нижней части – получившееся результирующее расписание, соответствующее данной расстановке процессов.

Если хотя бы одно из трёх перечисленных условий нарушается, система выдаёт соответствующее диагностическое сообщение и расписание не строит.

Тем не менее, эти условия не являются достаточными. На рисунке 7 приведён пример, удовлетворяющий всем трём условиям, однако, какие бы расстановки процессов мы ни брали, расписание построить не удастся:

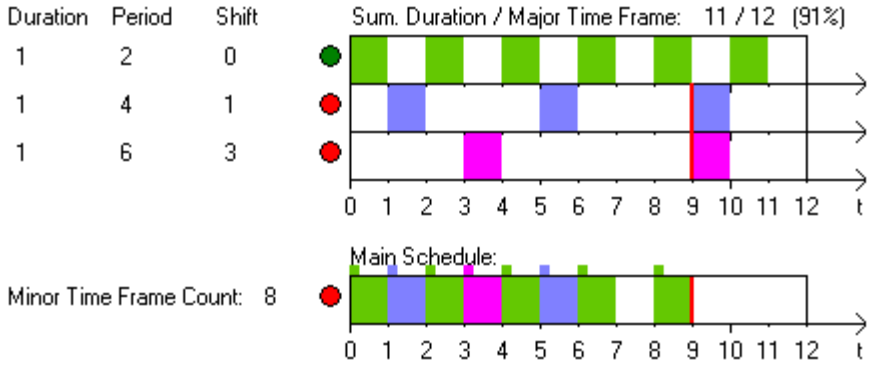


Рисунок 7

Однако формулировка и доказательство условий, отсекающих этот и подобные случаи, слишком сложны, чтобы приводить их здесь, поэтому в данной работе рассматриваться не будут. Если они всё же имеют место, алгоритм не может построить расписание, и в этом случае система выдаёт соответствующее сообщение.

5.2.2. Сложность задачи построения расписаний

В этом разделе мы покажем, что задача нахождения оптимального расписания, удовлетворяющего поставленным требованиям, как и многие другие задачи комбинаторной оптимизации, является NP-трудной.

Прежде всего, заметим, что для оптимального расписания число переключений процессора лежит между числом всех итераций всех процессов за один цикл (то есть, когда в данном расписании нет ни одной коллизии процессов) и числом, равным суммарной длительности всех итераций всех процессов (когда процессор переключается с одной задачи на другую каждый момент времени). Причём нижняя оценка для некоторых систем достигается.

В ряде работ [2, 3] исследуется задачи допустимости периодических систем реального времени, то есть возможности построить для них расписание. В этих работах рассматриваются системы в общем виде, в котором они были определены в разделе 3 данной работы. Авторы [2] доказывают теорему, что задача определения, является ли данная система недопустимой, принадлежит классу NP-трудных задач. К этой задаче они сводят NP-трудную задачу о «K одновременных конгруэнций». Однако в этих работах, как и во многих других, под периодом процесса понимается промежуток времени между его двумя последовательными моментами времени готовности, причём фактическое начало очередной итерации процесса может быть позже. То есть, такой подход ближе к спорадическим системам.

С другой стороны, в работе [6] рассматриваются системы реального времени, в которых задачи не могут прерываться. Авторы показывают, что задача определения того, возможно ли для системы с задачами без прерываний построить расписание, удовлетворяющее всем предельным срокам выполнения, – является NP-полной (а значит, и NP-трудной).

Теперь допустим, что в нашей рассматриваемой системе параметры длительностей и периодов подобраны так, что существует расписание, при котором не возникает коллизий процессов. Очевидно, оно будет оптимальным, так как при этом достигается нижняя оценка числа переключений процессора. Предположим, что рассматриваемый точный алгоритм строит это расписание за полиномиальное время. Но тогда получается, что за полиномиальное время была решена NP-полная задача, рассматриваемая в работе [6], чего не может быть в предположении $P \neq NP$. Если же такого расписания без прерываний для данной конфигурации системы не существует, то алгоритм строит какое-либо другое оптимальное расписание, в котором есть хотя бы одна коллизия процессов. Сравнив число переключений процессора с суммарным числом итераций процессов и обнаружив, что первое больше, алгоритм решает NP-трудную задачу из работы [6] с ответом «нет».

Таким образом, известная NP-трудная задача о построении расписания в системе без прерываний была сведена к данной задаче, что доказывает, что последняя также является NP-трудной.

5.2.3. Алгоритм построения расписаний

Поскольку задача построения расписаний для данной системы является NP-трудной, алгоритм не может найти оптимальное расписание за полиномиальное время. По сути, он является переборным и имеет экспоненциальную сложность.

Параметром перебора в алгоритме для данной неполной системы являются конфигурации полных систем, то есть векторы начальных смещений процессов $\{s_1, \ldots, s_n\}$, иначе говоря, всевозможные расстановки процессов относительно начала расписания. Для каждой такой расстановки алгоритм строит ровно одно корректное расписание и считает количество переключений между процессами. Из всех вариантов расстановок алгоритм запоминает и выбирает ту, в которой число переключений получилось меньше. Если данное число оказалось равным числу всех итераций всех процессов за один цикл расписания, то это значит, что найдено гарантированно оптимальное расписание, и алгоритм завершает свою работу досрочно. Если на каком-либо шаге при построении очередного варианта число переключений превысило минимальное число из уже обработанных вариантов, то алгоритм переходит к следующему шагу, не достроив неудачный вариант.

Составная часть рассматриваемого алгоритма – алгоритм построения отдельного варианта расписания – по сути, является эвристической, однако построенное расписание оказывается почти всегда оптимальным для данной расстановки или близким к оптимальному. Эта часть имеет две стратегии. Первая стратегия очень похожа на алгоритм EDF, с той лишь разницей, что она учитывает те моменты времени, в которых должен запуститься какой-либо процесс, и назначает на этот момент времени именно его. Однако в следующий момент времени может произойти переключение на другой процесс, чей относительный предельный срок выполнения ближе. Поэтому для такой стратегии нередки ситуации, когда последний фрагмент одного процесса лишний раз нарушает непрерывность другого. Следовательно, более предпочтительной является вторая стратегия, при которой запущенный процесс продолжает работать до своего завершения, либо до того, как его прервёт какой-нибудь другой процесс. Однако в этом случае алгоритм может нарушить *deadline* какого-либо процесса. Поэтому действия итогового алгоритма следующие: сначала он пытается применить вторую стратегию, а если нарушаются предельные сроки, то он возвращается к началу неудавшейся итерации процесса, применяет первую стратегию и проводит затем эвристическую оптимизацию. С применением только первой стратегии возможно построить расписание, кроме случаев, упомянутых в разделе 5.2.1.

Таким образом, на отдельном шаге может возникнуть потенциально неоптимальный вариант расписания (для данной расстановки), однако на следующем – оптимальный для данной расстановки. И если в новом варианте число переключений будет меньше, старый будет заменён. В конце концов, лучший из построенных вариантов почти всегда (в 97% случаев, проверявшихся на конфигурациях, в которых параметры выбирались случайным образом в небольших диапазонах) оказывается действительно оптимальным расписанием.

Без ограничения общности можно считать, что в момент времени 0 запускается один из процессов, иначе можно просто сместить рамки цикла расписания не необходимое число единиц времени. Поэтому далее будем полагать, что в момент времени 0 зафиксирован запуск какого-то одного процесса (например, с наибольшим периодом).

Рассмотрим случай, когда к моменту времени L, равному размеру цикла расписания, осталось нераспределённым несколько единиц времени некоторых процессов. Это не обязательно означает, что построить расписание не удалось. Момент L – это предельный срок выполнения только одного процесса, который начинается в момент времени 0, а у остальных *deadline* позже. Поэтому в этом случае алгоритм строит ещё и второй цикл расписания, учитывая процессы, не завершённые в первом цикле. Тогда если в конце второго цикла остаются незавершёнными ровно те же самые процессы, что и в конце первого (то, что это именно так, доказывается, например, в [2], хоть и без условия строгой периодичности запуска), расписание считается корректным и в качестве результата берётся второй цикл. На рисунке 8 приведён пример корректного расписания, при котором последние итерации жёлтого и зелёного процессов работают на границе первого и второго цикла расписания. Фрагменты, которые остались нераспределёнными в первом цикле, размещены во втором в моменты времени 4 и 5-6.

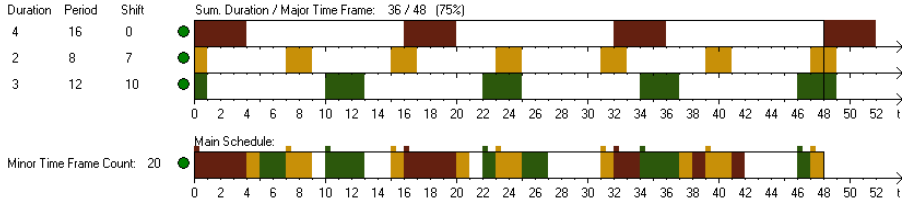


Рисунок 8

Перебор возможных вариантов расписания осуществляется рекурсивным образом. Процесс n проходит последовательно все свои допустимые значения s_n от 0 до $p_n - 1$, затем увеличивается значение начального смещения s_{n-1} процесса $(n - 1)$, и для n-ного процесса смещение снова принимает все свои

допустимые значения. Когда s_{n-1} проходит все свои значения, смещается процесс $(n-2)$, и так далее, до 1-го процесса.

Однако не любой вектор начальных смещений $\{s_1, \ldots, s_n\}$ является допустимым. Если для каких-либо двух процессов разность их начальных смещений окажется кратной наибольшему общему делителю их периодов, то в некоторый момент времени совпадут запуски итераций этих процессов. Это условие также вытекает из соотношения Безу. Докажем его. Возьмём два процесса с периодами p_1 и p_2 и начальными смещениями s_1 и s_2. Пусть НОД$(p_1, p_2) = d > 1$, тогда периоды представимы в виде $p_1 = p'_1 d$, $p_2 = p'_2 d$, где p'_1 и p'_2 – взаимно простые числа. Пусть также существует натуральное s' такое, что: $s_2 - s_1 = s'd$. Так как p'_1 и p'_2 – взаимно простые, то по соотношению Безу найдутся коэффициенты k'_1 и k'_2 такие, что:

$$k'_1 p'_1 - k'_2 p'_2 = 1$$

Умножим это равенство на s' и обозначим $k_1 = k'_1 s'$, $k_2 = k'_2 s'$ и получим:

$$s'k'_1 p'_1 - s'k'_2 p'_2 = s'$$
$$k_1 p'_1 - k_2 p'_2 = s'$$

Затем умножим получившееся равенство ещё на d, учтём равенства $p_1 = p'_1 d$, $p_2 = p'_2 d$, $s_2 - s_1 = s'd$ и перенесём слагаемые из одной части в другую:

$$k_1 p'_1 d - k_2 p'_2 d = s'd$$
$$k_1 p_1 - k_2 p_2 = s_2 - s_1$$
$$s_1 + k_1 p_1 = s_2 + k_2 p_2$$

Мы получили, что если

$$s_2 - s_1 \equiv \text{НОД}(p_1, p_2), \quad (4)$$

то найдутся такие k_1 и k_2, что соответствующие итерации процессов будут запускаться одновременно, что недопустимо. Условие доказано.

Алгоритм при каждом смещении i-го процесса проверяет, что ни для какого другого процесса не выполняется условие (4), и только тогда строит расписание, иначе сразу переходит к следующему шагу. На рисунке 9 в качестве примера представлены два последовательных шага алгоритма:

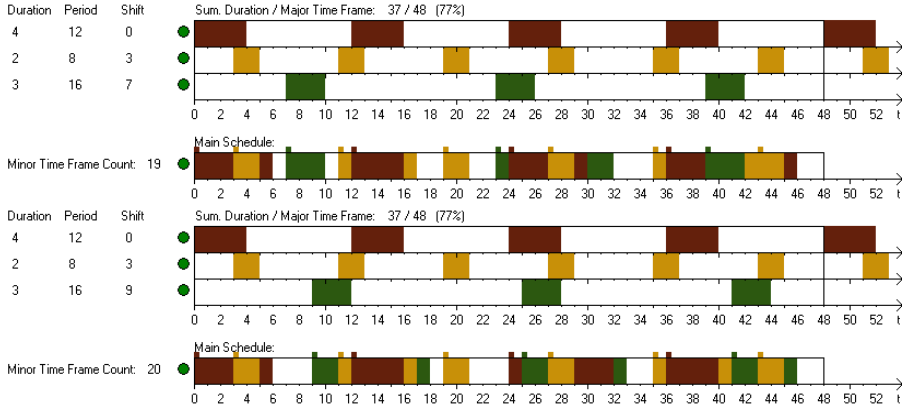


Рисунок 9

Между ними один шаг был пропущен, и алгоритм на него не тратит времени. Остаётся добавить, что существенно сокращают перебор случаи, когда пропускается сразу несколько шагов (например, когда *i*-й процесс пропускает какое-либо своё недопустимое смещение, весь перебор всевозможных смещений для процессов $j > i$ также пропускается).

Таким образом, точный алгоритм, выполняющий все требования, по своей сути является переборным, однако этот перебор можно во многом сократить. Тем не менее, для больших данных он малоприменим. Поэтому был предпринят **другой подход**: алгоритм в процессе перебора находит и отображает графически в качестве промежуточных результатов всё менее и менее фрагментированные расписания. И когда найдено расписание с достаточно малым числом переключений процессов, системный интегратор его может остановить и использовать промежуточный результат в качестве окончательного. При этом какое-либо расписание (неоптимальное) появляется почти сразу.

5.3. Графическое и табличное представления расписаний

Разработанная система отображает построенные окончательные (или промежуточные) расписания в графическом виде. Расписания могут быть представлены в двух видах: с детализацией (с разбивкой) по процессам или в одну строчку. Соответствующие примеры расписания приведены на рисунках 9 и 10.

Кроме описанного алгоритма, в системе могут работать и другие, решающие несколько модифицированные задачи составления расписаний. Например, для случая, когда строгая периодичность запуска процессов не так важна, могут применяться классические алгоритмы RM и EDF. Результаты их работы система также позволяет отобразить графически. На рисунке 10 представлены результаты работы трёх алгоритмов на одном примере. Красные штрихи

обозначают моменты запусков периодических процессов. Отсюда видно, что главный (точный) алгоритм выдерживает периодичность процессов, а классические алгоритмы её нарушают.

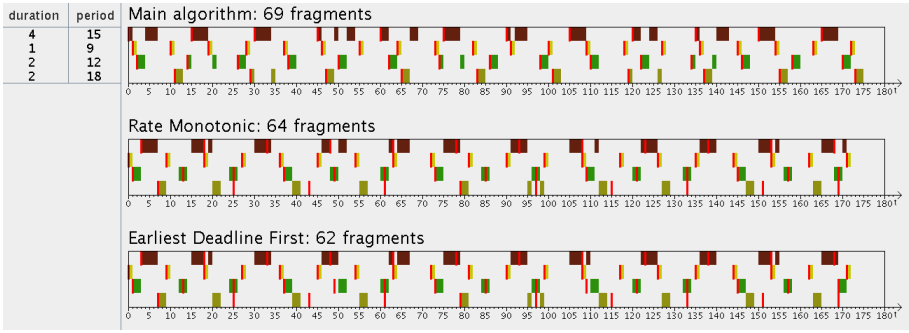


Рисунок 10

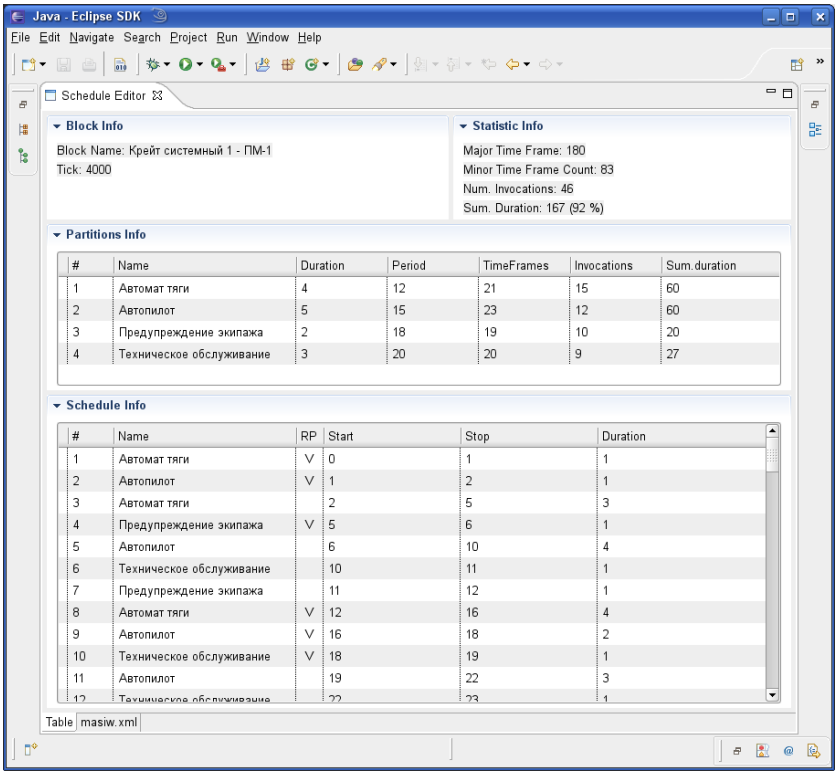


Рисунок 11

Разработанная система позволяет представить расписание и в табличном виде, в рабочем окружении интегрированной среды разработки Eclipse. На рисунке 11 приведён пример одной из таблиц, отображающих построенное расписание.

В верхней части таблицы приведена информация о процессоре, для которого было составлено расписание (его имя, размер тика), и краткая статистика (длина цикла расписания, число переключений процессора, число всех итераций всех процессов, суммарная длительность всех итераций всех процессов за один цикл и «плотность заполнения» расписания, выраженная в процентном соотношении). В середине таблицы перечисляются параметры процессов (партиций), такие как длительность и период и некоторая статистика по каждому процессу (число фрагментов, число итераций и суммарная длительность за один цикл расписания). Само расписание, занимающее всю оставшуюся часть таблицы, выражено не в виде функции S: $\{0, \ldots, L-1\} \rightarrow \{0, 1, \ldots, n\}$, а более компактно – в виде набора фрагментов (*англ.* Minor Time Frame) непрерывного выполнения конкретной задачи. Для каждого такого фрагмента указывается имя соответствующего ему процесса, содержит ли он момент времени периодического запуска процесса (флаг RP), а также начало, конец и длительность фрагмента.

Разработанная система позволяет изменять параметры процессов (длительность, период) путём занесения в соответствующую ячейку таблицы нового значения, а затем вызвать алгоритм построения расписания, чтобы получить его новую версию.

Кроме того, поскольку системный интегратор может остановить процесс поиска оптимального расписания, ему требуется возможность довести промежуточный результат до оптимального путём редактирования расписания. Для этого система позволяет изменить длительности существующих фрагментов, либо вообще удалить их, а также предоставляет меню, вызывающее мастер создания новых фрагментов.

5.4. Проверка корректности расписания

В случае если изменения нарушают соответствие расписания параметрам процессов, последние выделяются и для них отмечаются те параметры, которые оказались некорректными. Причём в самом расписании отмечаются те моменты времени, которые вызвали проблемы.

На рисунке 12 приведён пример, в котором после редактирования нарушилась периодичность запуска жёлтого процесса из-за того, что была увеличена длительность коричневого (добавлен момент времени 43 для коричневого процесса). Соответственно, для коричневого была помечена некорректная длительность, а для жёлтого – период.

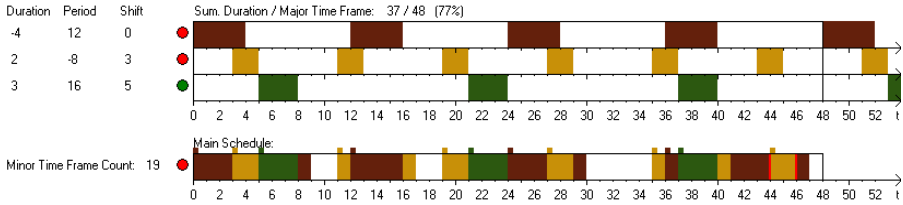


Рисунок 12

Алгоритм проверки сначала проходит по расписанию и определяет число процессов. Затем для каждого процесса вычисляется промежуток времени между первой и второй итерацией и длительность первой итерации и запоминает эти значения. Для всего оставшегося расписания он сравнивает длительности и периоды текущего процесса с сохранёнными значениями. Сумма начального смещения и промежуток между началом последней итерации и концом цикла расписания – также сравнивается с периодом. Случай, когда процесс работает на границе двух циклов расписания, тоже учитывается. Если на каком-либо этапе возникает несогласованность данных, алгоритм отмечает соответствующий процесс и его фрагмент в расписании.

Если все параметры оказались согласованными, то можно говорить о том, что алгоритм по имеющемуся расписанию восстановил список процессов и их исходные параметры.

Алгоритм не может определить, какой из нескольких фрагментов процесса привёл к нарушению согласованности параметров, поэтому он отмечает тот фрагмент, на котором расхождение стало очевидным.

Конечно, априори нельзя определить, каким алгоритмом (точным, RM или EDF) было построено расписание. Поэтому при проверке на корректность результатов работы классических алгоритмов всё расписание окажется помечено красными штрихами, а для большинства процессов будут отмечены (или даже неправильно определены) периоды. Длительности, тем не менее, будут указаны верно.

6. Заключение

В данной работе, при решении задачи составления расписаний для целевой системы, были получены следующие результаты:

- Сформулированы необходимые условия существования расписания для системы, удовлетворяющей заданным ограничениям.
- Доказана NP-трудность решаемой задачи построения расписаний.
- Разработан алгоритм, решающий задачу построения расписаний точно и находящий оптимальное расписание или

близкое к оптимальному. Использован метод ограниченного перебора.

- Разработана система получения промежуточных результатов работы алгоритма и управления перебором.
- Разработанная система отображает построенные расписания в графическом и табличном виде.
- Предоставлена возможность редактировать построенные расписания и проверять корректность изменений.
- Реализованы классические алгоритмы RM и EDF в качестве альтернативы для тех случаев, когда строгая периодичность запусков процессов не так важна.

В работе было показано, что для разных, несколько похожих задач построения расписаний существуют совершенно различные алгоритмы. Особенно хорошо это видно на примере контраста разработанного алгоритма и классических алгоритмов Rate Monotonic и Earliest Deadline First. Добавление всего лишь одного требования строгой периодичности запуска процессов делает из полиномиально разрешимой задачи NP-трудную.

Поэтому был предпринят другой подход: для целевой системы реального времени была разработана система инструментов, позволяющих системным интеграторам автоматизировать отдельные операции составления расписаний. Для систем с небольшим числом процессов и с их небольшими периодами алгоритм находит оптимальное расписание (т.е. решает задачу точно). А для больших систем, когда алгоритм работает очень долго, разработанный механизм время от времени выводит промежуточные корректные, но не оптимальные результаты в графическом виде. И в тот момент, когда ответственному персоналу покажется, что полученное расписание уже приемлемо или его легко довести до оптимального, он может остановить работу алгоритма, сохранить промежуточный результат, вручную довести его до оптимального, а затем проверить на корректность. Для тех случаев, когда скорость построения расписаний важнее требования строгой периодичности запуска процессов, система инструментов позволяет применять классические полиномиальные алгоритмы.

Литература

[1] Mohammadi A., G. Akl S.G. Scheduling Algorithms for Real-Time Systems // School of Computing, Queen's University . 2005. Technical Report N 2005–499.

[2] Leung J.Y.-T., Merrill M.L. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks // Information Processing Letters. 1980. Vol. 11. N 3. P.115–118.

[3] Baruah S.K., Rosier L.E., Howell R.R. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor // Real Time Systems. 1990. Vol. 2. P. 301–324.

[4] Brucker P., Kampmeyer T. Tabu search algorithms for cyclic Machine scheduling problems // Journal of Scheduling. 2005. N 8. P. 303–322.

[5] Brucker P., Kampmeyer T. A general model for cyclic machine scheduling problems // Discrete Applied Mathematics. 2008. Vol. 156. N 13. P. 2561–2572.

[6] Georges L., Muhlethaler P., Rivierre N. A Few Results on Non-Preemptive Real time Scheduling // Rapport de recherché. 200. N 3926.
[7] Сорокин С. Системы Реального Времени // СТА. 1997. №2. С. 22–29.
[8] Горошко Е. Операционные системы реального времени [PDF] (http://www.qnxclub.net/files/articles/rtos/rtos.html).
[9] Timmerman M., Beneden B.V., Uhres L. RTOS Evaluation Kick Off! // Real-Time Magazine. 1998. N3. P. 6–10.
[10] Операционные системы реального времени для авионики: обзор [HTML] (http://rnd.cnews.ru/reviews/index_science.shtml?2008/05/05/299461_1).
[11] Зеленов С.В. Планирование строго периодических задач в системах реального времени // Труды Института системного программирования РАН. 2011. Т. 20. С. 113–122.

Automation of scheduling for periodic real-time systems

A.V. Tretyakov.
andrromeda@ispras.ru
ISP RAS, Moscow, Russia

Abstract. In the paper, we consider the scheduling problem for strictly periodic real-time systems. Strict periodicity means that all release points of all instances of any task must produce an arithmetic progression. Classical scheduling algorithms, such as Earliest Deadline First (EDF) and Rate-Monotonic Scheduling (RMS), are not applicable under strict periodicity constraint. The main problem is to search release points for all tasks. In fact, this search is NP-hard problem.

First, we study some necessary schedulability conditions for both classical and strictly periodic schedulability problem. Next, we suggest scheduling algorithm that consists of two main parts: heuristic algorithm of release points search, and scheduling algorithm for given release points. The algorithm of release points search is based on some ideas in number theory that allows iterate not all possible release points but only such points that with a high probability yields correct schedule. The scheduling algorithm for given release points is based on ideas of EDF algorithm.

Finally, we present developed tool set that provides automated scheduling of given tasks, allows visualization of generated schedule, provides GUI to edit schedule, and supports validation of built schedules. This tool set is a part of workspace for design of modern avionics systems developed in ISP RAS.

Keywords: scheduling, real-time systems, hard periodic tasks, deadlines, NP-hard problem, Integrated Modular Avionics

References

[1]. Mohammadi A., G. Akl S.G. Scheduling Algorithms for Real-Time Systems. School of Computing, Queen's University . 2005. Technical Report N 2005–499.
[2]. Leung J.Y.-T., Merrill M.L. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. Information Processing Letters. 1980. Vol. 11. N 3. P.115–118.
[3]. Baruah S.K., Rosier L.E., Howell R.R. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. Real Time Systems. 1990. Vol. 2. P. 301–324.
[4]. Brucker P., Kampmeyer T. Tabu search algorithms for cyclic Machine scheduling problems. Journal of Scheduling. 2005. N 8. P. 303–322.
[5]. Brucker P., Kampmeyer T. A general model for cyclic machine scheduling problems. Discrete Applied Mathematics. 2008. Vol. 156. N 13. P. 2561–2572.
[6]. Georges L., Muhlethaler P., Rivierre N. A Few Results on Non-Preemptive Real time Scheduling. Rapport de recherché. 200. N 3926.
[7]. Sorokin S. Sistemy Real'nogo Vremeni [Real Time Systems]. Sovremennye tekhnologii avtomatizatsii [Contemporary Technologies in Automation]. 1997. No2. P. 22–29 (in Russian).

[8]. Goroshko E. Operatsionnye sistemy real'nogo vremeni [Real Time Operating Systems]. Khar'kov, 2003 (in Russian).
[9]. Timmerman M., Beneden B.V., Uhres L. RTOS Evaluation Kick Off! Real-Time Magazine. 1998. N3. P. 6–10.
[10]. Operatsionnye sistemy real'nogo vremeni dlya avioniki: obzor [Real Time Operating Systems for Avionics: Review]. R&D.CNews.ru, 2008 (in Russian). http://rnd.cnews.ru/reviews/index_science.shtml?2008/05/05/299461_1
[11]. Zelenov S.V. Planirovanie strogo periodicheskikh zadach v sistemakh real'nogo vremeni [Scheduling of Strictly Periodic Tasks in Real-Time Systems]. Trudy ISP RAN [Proceedings of ISP RAS], 2011, vol. 20, pp. 113–122 (in Russian).

Системы рекомендаций: обзор современных подходов

А.Г. Гомзин, А.В. Коршунов
gomzin@ispras.ru, korshunov@ispras.ru

Аннотация. Статья представляет собой обзор основных алгоритмов, использующихся в системах рекомендаций. Рассмотрены методы коллаборативной фильтрации, методы, анализирующие содержимое объектов и методы, использующие базы знаний. Все рассматриваемые методы имеют свои недостатки. В статье рассмотрено, какими способами можно комбинировать методы построения рекомендаций, чтобы избавиться от этих недостатков.

Ключевые слова: система рекомендаций, коллаборативная фильтрация, базы знаний, гибридная система рекомендаций

1. Введение

Очень часто человек выбирает, какие новые фильмы посмотреть, какую новую музыку послушать, какие товары купить в интернет-магазине. Это совсем не простая задача, потому что он имеет в распоряжении лишь названия, краткие описания доступных фильмов или товаров, и, может быть, советы друзей и знакомых. Имея столь малое количество информации, очень трудно определить, понравится ли конкретный фильм, песня или товар данному человеку. Задача усложняется еще и тем, что объектов, из которых приходится выбирать, очень много. Поэтому в сети Интернет широко распространены системы рекомендаций, помогающие пользователю решить задачу выбора песен, фильмов, товаров.

Системы рекомендаций позволяют пользователю выбрать среди всех доступных объектов именно те, которые будут ему интересны. Эти системы обрабатывают информацию о различных объектах, а также о том, какие пользователи какие объекты купили, посмотрели, послушали и т.д. Примерами таких сервисов являются last.fm, hunch.com, youtube.com и другие. Имея эти данные и алгоритмы, которые будут рассмотрены в данной работе, можно быстро и качественно отфильтровать самые подходящие конкретному пользователю объекты.

Кроме того, есть системы, которые рекомендуют других пользователей, имеющих схожие интересы, или потенциальных знакомых. Как правило, это

социальные сети. Facebook (facebook.com), ВКонтакте (vk.com), last-fm (last.fm).

В данной работе будут рассмотрены основные подходы и алгоритмы, используемые в различных системах рекомендаций. Будут выявлены сильные и слабые стороны каждого подхода.

2. Основные понятия

Перед тем, как рассматривать алгоритмы, введем некоторые понятия.

Объект – это песня, фильм, товар, пользователь (в случае рекомендации друзей). Т.е. то, что потребляют пользователи системы рекомендаций. Это то, что им нужно рекомендовать.

Пользователь – это человек, зарегистрированный в системе, он может покупать, слушать, смотреть, оценивать объекты и пользоваться сервисом рекомендации.

Рекомендация – это объект или несколько объектов, которые система рекомендации выдает пользователю.

Основная задача системы рекомендации – имея данные об объектах и пользователях, получить список объектов, наиболее интересных для конкретного пользователя. В зависимости от того, какие данные используются для расчета рекомендаций, системы делятся на три больших класса:

- Методы коллаборативной фильтрации [1-4]
- Методы, анализирующие содержимое объектов
- Методы, основанные на знаниях

Методы коллаборативной фильтрации предполагают, что интересы пользователей представлены оценками, которые они дают объектам после просмотра, покупки и т.д. Основная идея данных методов заключается в сравнении между собой интересов различных пользователей или объектов на основе этих оценок. При этом никакой дополнительной информации о самих пользователях и объектах не используется.

Методы второго класса, наоборот, используют содержимое объектов для получения рекомендаций. Эти методы работают в тех случаях, когда содержимое объектов представлено в виде текстов. Они хорошо подходят для рекомендации книг. Также их можно использовать для сравнения названий, описаний и другой текстовой информации, доступной у фильмов, песен, товаров и т.д. [5]

Методы, основанные на знаниях, требуют от пользователя описать свои требования к нужным ему объектам. А затем ищут с использованием своей базы знаний объекты, удовлетворяющие поставленным требованиям.

3. Коллаборативная фильтрация

Коллаборативная фильтрация (Collaborative filtering) – это метод рекомендации, при котором анализируется только реакция пользователей на объекты: оценки, которые выставляют пользователи объектам. Оценки могут быть как явными (пользователь явно указывает, на сколько «звездочек» он оценивает объект), так и неявными (например, количество просмотров одного ролика). Чем больше оценок собирается, тем точнее получатся рекомендации. Получается, что пользователи помогают друг другу в фильтрации объектов. Поэтому такой метод называется также совместной фильтрацией.

3.1. Описание алгоритма

Пусть в системе есть пользователи и объекты. Пусть некоторые пользователи оценили некоторые объекты. И пусть оценка – это натуральное число от 1 до 5. Тогда все оценки можно изобразить в виде матрицы (см. рис. 1).

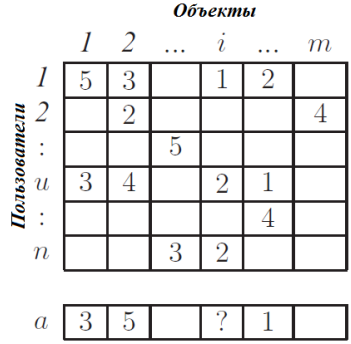


***Рис. 1** Матрица оценок.*

Пусть имеется пользователь a. Наша задача – предсказать, какую оценку поставил бы пользователь a объекту i. Будем рассматривать только пользователя a и тех пользователей, которые оценили объект i. Алгоритм включает в себя 3 шага:

1. Для каждого пользователя u вычислим, насколько его интересы совпадают с интересами пользователя a
2. После этого выберем множество пользователей, наиболее близких к a
3. Предскажем оценку на основе оценок объекта i "соседями" из предыдущего шага

Первый шаг. Каждому пользователю в матрице R соответствует одна строка. Поэтому будем вычислять близость векторов-строк пользователей.

Существует множество способов подсчета близости векторов [6,7]. Один из самых простых – посчитать косинус между этими векторами:

$$sim(u,a) = \frac{\sum_{i=1}^{m} r_{a,i} \cdot r_{u,i}}{\sqrt{\sum_{i=1}^{m} r_{a,i}} \cdot \sqrt{\sum_{i=1}^{m} r_{u,i}}} \quad (1)$$

Здесь *sim(u,a)* – мера близости (похожести) пользователей *a* и *u*. $r_{u,i}$ – значение матрицы *R*: *u* строка, *i* столбец. *sim(u,a)* принимает значения из отрезка *[0,1]*. Если пользователь не указал оценку для какого-то объекта, соответствующее значение матрицы равно 0.

Существует другая мера близости между векторами – коэффициент корреляции Пирсона:

$$sim(u,a) = \frac{\sum_{i \in I} (r_{a,i} - \overline{r_a})(r_{u,i} - \overline{r_u})}{\sqrt{\sum_{i \in I} (r_{a,i} - \overline{r_a})^2 \cdot \sum_{i \in I} (r_{u,i} - \overline{r_u})^2}} \quad (2)$$

Здесь *I* – множество объектов, которые оценил как пользователь *a*, так и пользователь *u*. $\overline{r_a}$ и $\overline{r_u}$ - это средние оценки пользователей *a* и *u*, соответственно.

В числителе подсчитывается произведение отклонений оценок двух пользователей от средних значений для одного объекта. Знаменатель необходим для того, чтобы данная величина принимала значения из отрезка *[-1,1]*. Чем сильнее совпадают интересы, тем ближе значение близости к *1*. Если коэффициент корреляции Пирсона отрицателен, то интересы пользователей противоположные.

(1) и (2) – самые распространенные меры близости, используемые в данном алгоритме. Коэффициент корреляции Пирсона (2) будет оставаться высоким, если все оценки сравниваемых пользователей отличаются на постоянную величину (один пользователь оценивает более строго, другой – ставит более высокие оценки). Косинус (1) в этом случае выдаст меньшее значение. Но, если оценки двух пользователей отличаются пропорционально, то косинус (1) будет больше, корреляция (2) – меньше. В обоих случаях оценки пользователей похожи. Поэтому, какую из этих метрик использовать – зависит от конкретной задачи.

Существуют и другие меры близости [6,7]. В данной работе они рассматриваться не будут.

Второй шаг. Теперь нужно выбрать множество *K* наиболее похожих на *a* пользователей. Можно выбрать всех пользователей. Но так как пользователей с непохожими или несильно пересекающимися интересами довольно много, то они будут отрицательно влиять на точность предсказания оценки для объекта *i*. Кроме того, количество пользователей влияет на объем вычислений на третьем шаге.

Одно из возможных решений – установить порог меры близости, вычисленной на первом шаге: (1), (2). Пользователи с мерой близости, превышающей порог, войдут во множество *K*. Остальные – нет. Но чаще всего выбирается целая константа *k*. Затем все пользователи сортируются по убыванию меры близости. И во множество *K* входят *k* пользователей, наиболее близких к *a*.

Третий шаг. Имея множество *K* близких пользователей нужно вычислить оценку, которую поставил бы пользователь *a* объекту *i*. Напомним, что рассматриваются только те пользователи, которые оценили объект *i*.

Нужная оценка вычисляется по формуле:

$$p_{a,i} = \overline{r_a} + \frac{\sum_{u \in K} (r_{u,i} - \overline{r_u}) \times sim(a,u)}{\sum_{u \in K} | sim(a,u) |} \quad (3)$$

$p_{a,i}$ – это предсказываемая оценка пользователя *a* для объекта *i*. За основу берется его средняя оценка $\overline{r_a}$, а затем прибавляется среднее отклонение оценки других пользователей из множества *K* для объекта *i* от их средней оценки. Чем ближе пользователь *u* к пользователю *a* (согласно мере близости *sim(a,u)*, вычисленной на первом шаге), тем сильнее его вклад в предсказание оценки.

Таким образом, описанный алгоритм предсказывает оценки для объектов, которые текущий пользователь еще не оценил. Для того чтобы сделать рекомендацию для данного пользователя, достаточно предсказать оценки для всех неоцененных объектов и выбрать объекты с наибольшей предсказанной оценкой.

3.2. Альтернативные алгоритмы и улучшения

Описанный выше алгоритм основан на сравнения между собой пользователей. Он называется User-based Collaborative filtering. Существует также другой подход: вместо пользователей сравнивать объекты (Item-based Collaborative filtering). В этом случае алгоритм почти такой же, включает 3 шага:

1. Для каждого объекта *j* вычислим, насколько он похож на объект *i*, для которого предсказывается оценка.
2. Выберем множество объектов, наиболее близких к *i*
3. Предскажем оценку на основе оценок выбранных на втором шаге объектов пользователем *a*.

Сравнение объектов на первом шаге происходит также как и в (1) и (2)

$$sim(i, j) = \frac{\sum_{u \in U} (r_{u,i} - \overline{r_i}) \cdot (r_{u,j} - \overline{r_j})}{\sqrt{\sum_{u \in U} (r_{u,i} - \overline{r_i})^2 \cdot \sum_{u \in U} (r_{u,j} - \overline{r_j})^2}} \quad (4)$$

Здесь суммирование ведется по множеству U пользователей, оценивших объекты i и j. $\overline{r_i}$ – средняя оценка объекта i.

Второй шаг аналогичен второму шагу алгоритма, основанного на сравнении пользователей. Основное отличие в том, что K – множество близких к i ***объектов***. На третьем шаге считается взвешенная средняя оценка похожих объектов, полученная пользователем a:

$$p_{a,i} = \frac{\sum_{j \in K} r_{u,j} \times sim(i, j)}{\sum_{j \in K} |sim(i, j)|} \quad (5)$$

В алгоритме, основанном на сравнении объектов, использовалась более сложная формула (3). Это связано с тем, что средняя оценка объекта показывает, насколько хорошим является объект, а средняя оценка пользователя показывает, насколько строго пользователь оценивает объекты. Формула (3) вычисляет оценку с учетом того, что строгость оценки разных пользователей, вообще говоря, различна.

Алгоритм, основанный на сравнении объектов, имеет несколько преимуществ. Так как интересы пользователей часто меняются, а средняя оценка объекта остается постоянной, то при сравнении объектов достаточно посчитать меру близости каждого с каждым и сохранить в памяти. В частом обновлении этих данных нет необходимости. Причем эти меры близости можно посчитать в оффлайне, т.е. когда в системе нет активных пользователей.

В современных системах рекомендаций постоянно растет число пользователей и объектов, поэтому наступает момент, когда необходимо использовать распределенную архитектуру системы. При этом возникает проблема хранения данных: было бы неплохо похожих пользователей или похожие объекты хранить рядом. Для ее решения можно применить алгоритмы кластеризации для пользователей или объектов, используя меру близости между ними: (1), (2), (4). Разбиение пользователей или объектов на группы дает еще одно преимущество: так как в кластере находятся похожие объекты, то нужный объект достаточно на первом шаге сравнивать с объектами из своего кластера. Это значительно сокращает количество вычислений.

Описанные выше методы коллаборативной фильтрации называются методами, основанными на работе с памятью, так как необходимо хранить матрицу R. Есть и другая группа алгоритмов – методы, основанные на моделях. Идея заключается в том, что имея матрицу R можно построить модель, которая будет меньше по объему занимаемой памяти. Используя полученную модель, можно будет вычислять рекомендации. Примерами таких моделей являются Байесовский алгоритм [2], при котором исходная задача сводится к задаче классификации; Марковские модели [8], которые позволяют учитывать не только похожесть объектов, но и их ценность; а также другие модели [3].

4. Анализ содержимого

Коллаборативная фильтрация использует данные обратной связи пользователя (оценки, отзывы). Однако, не всегда доступно необходимое количество оценок. Кроме того, существует проблема "холодного старта": система не знает, что рекомендовать новому пользователю и кому рекомендовать новый объект.

В случаях, когда коллаборативная фильтрация не дает результата, приходится использовать информацию об объектах. Например, был снят фильм – продолжение какого-то фильма. Он добавляется в систему. Возникает вопрос, кому его рекомендовать. Коллаборативная фильтрация ответ не даст из-за проблемы «холодного старта». Но есть смысл рекомендовать новый фильм тем, кто посмотрел первую часть фильма.

4.1. Представление данных об объектах

Каждый объект в любой системе рекомендаций имеет название, а также и другую текстовую информацию, которую можно использовать. В табл. 1 представлен пример доступной информации о книгах на английском языке в интернет-магазине.

Название	**Жанр**	**Автор**	**Цена**	**Ключевые слова**
The Night of the Gun	Memoir	David Carr	29.90	press and journalism, drug addiction, personal memoirs, New York
The Lace Reader	Fiction, Mystery	Brunonia Barry	49.90	American contemporary fiction, detective, historical
Into the Fire	Romance, Suspense	Suzanne Brockmann	45.90	American fiction, murder, neo-Nazism

Табл. 1 *Информация о книгах в интернет-магазине.*

Пользователи системы имеют свой профиль. Там они указывают информацию о себе и своих интересах: любимые жанры, любимые авторы и т.д. Имея эти данные и информацию о книгах из таблицы 1 можно сделать выводы, интересна ли конкретная книга пользователю. Но, в этом случае мера похожести интересов пользователя и книги принимает два возможных значения: 1 – если жанр книги есть в профиле пользователя, 0 – в противном случае.

Для более точных рекомендаций используется другой подход: зная информацию о книгах, купленных человеком, ему можно посоветовать похожие книги. Похожесть двух книг измеряется с помощью сравнения

данных из табл. 1. Например, если *keywords(B$_i$)* – множество ключевых слов книги B_i, то сравнить ее с книгой B_j можно следующим образом:

$$dice(B_1, B_j) = \frac{|keywords(B_i) \bigcap keywords(B_j)|}{|keywords(B_i)| + |keywords(B_j)|} \quad (6)$$

Формула (6) называется коэффициентом Дайса. Он принимает значения из отрезка *[0,1]*. Он показывает, насколько близки два множества.

Но множество ключевых слов почти никогда не доступно явно. Их можно получить, анализируя непосредственно содержание объектов. Естественно, что извлекать ключевые слова можно только из книг, статей и других текстовых объектов. Для выделения ключевых слов используются различные алгоритмы: с использованием Википедии [9], TF-IDF и др.

4.2. TF-IDF

Пусть имеется набор текстов (документов). Все тексты разбиты на слова. Коэффициент TF-IDF показывает, насколько слово важно в тексте. Основная идея в том, что слово считается более важным, если оно часто встречается в одном тексте и редко в других.

TF (term frequency) -- это нормализованная частота слова в тексте:

$$TF(x, D) = \frac{freq(x, D)}{\max_{y \in D} freq(y, D)} \quad (7)$$

Здесь *freq(x,D)* – количество слов *x* в документе *D*. *TF* принимает значения из отрезка *[0,1]*.

IDF (inverse document frequency) – обратная частота документов.

$$IDF(x) = \frac{N}{n(x)} \quad (8)$$

Здесь *N* – количество документов в наборе, *n(x)* – количество документов, в которых встречается слово *x*.

Коэффициент *TF-IDF* вычисляется как произведение *TF* на *IDF*.

Для сравнения двух текстов, их можно представить в виде векторов в евклидовом многомерном пространстве. Каждому измерению соответствует слово, а значение каждой компонент вектора равно *TF-IDF*. А затем посчитать косинус между этими векторами.

Для уменьшения вычислительных затрат, затрат памяти и шума в вычислениях используются различные эвристики:

1. **Стоп-слова**. Такие слова, как предлоги, артикли, междометия встречаются в текстах очень часто, хотя не несут никакой полезной информации. Поэтому эти слова заносится в список

стоп-слов и не рассматриваются при построении векторной модели.

2. **Стемминг**. Известно, что одно и то же слово может принимать множество форм (разные падежи, времена и т.д.). Поэтому существуют алгоритмы, позволяющие превести слово к нормальной форме, чтобы отождествлять слова
3. **Уменьшение количества компонент вектора**. Различных слов в документе очень много. Но можно уменьшить их количество, оставив только с наибольшим весом *TF-IDF*.

Более подробно методы сравнения текстов рассмотрены в [10].

5. Методы, основанные на знаниях

Методы коллаборативной фильтрации и анализа содержимого хорошо работают, когда человек довольно часто покупает товары или слушает музыку. Но если это происходит редко (например, при покупке дома, автомобиля, компьютера), эти методы не помогут, потому что система не имеет нужного количества информации о покупаемых пользователем товарах.

Для решения такой задачи используются методы, основанные на знаниях [11]. Они делятся на две группы:

- Использование жестких ограничений
- Выбор близких объектов

Идея обоих методов следующая: пользователи формулируют свои требования к товару, система пытается найти нужный товар.

В первом случае ищутся и рекомендуются только те объекты, которые точно соответствуют всем требованиям пользователя. Во втором случае ищутся объекты с характеристиками, близкими к требованиям (с использованием некоторых мер близости).

5.1. Знания об объектах и запросы пользователей

Информация об объекте обычно представляется в виде набора параметров. Например, для фотокамеры это размер дисплея, зум, количество мегапикселей и т.д. Пусть в магазине имеются следующие фотокамеры (см. табл. 2).

номер	цена	Мега-пиксели	опт. зум	размер экрана	видео	звук	водостойкость
1	148	8.0	4x	2.5	нет	нет	да
2	182	8.0	5x	2.7	да	да	нет
3	189	8.0	10x	2.5	да	да	нет
4	196	10.0	12x	2.7	да	нет	да
5	151	7.1	3x	3.0	да	да	нет
6	199	9.0	3x	3.0	да	да	нет
7	259	10.0	3x	3.0	да	да	нет
8	279	9.1	10x	3.0	да	да	да

Табл. 2 *Фотокамеры в интернет-магазине.*

Задача системы – выбрать камеры, удовлетворяющие запросам пользователя. Но не всегда эти запросы выражаются в виде множества значений параметров, например, "камера ценой не более $200, не менее 8 мегапикселей, с поддержкой видео". Достаточно часто пользователя интересуют некоторые функциональные особенности. Тогда его запросы принимают вид: "камера для спортивной съемки", или "камера для печати больших фотографий". Поэтому выделяются два типа свойств: свойства товара и пользовательские свойства.

В нашем примере с камерами свойствами товара являются:

цена(0...1000), мегапиксели(3.0...12.0), опт-зум(4x-12x), дисплей(2.5...3.0), видео(да,нет), звук(да,нет), водостойкость(да,нет).

Пользовательские свойства:

максимальная_цена(0...1000), использование(цифровое, печать_больших_фото, печать_маленьких_фото), фотографии (спортивные, пейзажи, портреты, макросъемка).

Т.е. в своем запросе пользователь может указать максимальную цену, какого размера он будет печатать фотографии, что он будет фотографировать (пейзажи, портреты).

Помимо знаний о товарах (см. табл. 2) в базе знаний присутствуют ограничения. Ограничения могут представлять собой разрешенные значения пользовательских свойств, например:

использование=печать_больших_фото →максимальная_цена>200, т.е. система не позволяет пользователю указать минимальную цену меньше 200, если он хочет печатать большие фотографии.

Другой вид ограничений позволяет преобразовать свойства пользователя в свойства объектов:

использование=печать_больших_фото → мегапиксели>5.0, т.е. если пользователь хочет большие фотографии, то не следует ему рекомендовать камеры, меньше чем с 5 мегапикселями.

Итак, задача системы – выбрать объекты, удовлетворяющие запросам пользователя, с помощью знаний об объектах и ограничениях.

5.2. Использование жестких ограничений

Идея метода проста: найти объекты в базе данных, полностью удовлетворяющие требованиям пользователя. Но, поиск объектов не является основной проблемой. Главной задачей является получение требований пользователя. Нужно определить, как взаимодействовать с пользователем, какие вопросы и в какой последовательности задавать, какие вопросы можно не задавать. Таким образом, основная задача – получить систему, удобную для пользователя.

Для того чтобы ограничить количество задаваемых пользователю вопросов, используются значения параметров по умолчанию. Они делятся на 3 группы:

- **Статические умолчания.** Пример: *использование = печать_больших_фото.* Так как большинство пользователей требуют возможности печати больших фотографий, то можно установить такое значение параметра *использование* по умолчанию
- **Зависимые умолчания.** Пример: *использование = печать_маленьких_фото* → *максимальная_цена = 300.* Т.е. если пользователь указал, что будут печататься маленькие фотографии, то его можно не спрашивать о максимальной цене и указать ее равной 300
- **Производные умолчания.** Если первые два типа умолчаний определяются заранее, то производные умолчания вычисляются с использованием логов взаимодействия системы с пользователями

Пользователи указывают не все требуемые параметры, а только те, которые считают наиболее существенными. Логи взаимодействия предыдущих пользователей с системой используются также для предсказывания, какой следующий параметр будет интересен пользователю. Табл. 3 содержит информацию о том, в какой последовательности пользователи устанавливали атрибуты.

№	1 атрибут	2 атрибут	3 атрибут	4 атрибут	…
1	*цена*	*оптический зум*	*мегапиксели*	*видео*	…
2	*цена*	*оптический зум*	*мегапиксели*	*видео*	…
3	*цена*	*мегапиксели*	*оптический зум*	*размер дисплея*	…
4	*мегапиксели*	*цена*	*оптический зум*	*размер дисплея*	…
5	*мегапиксели*	*цена*	*размер дисплея*	*оптический зум*	…

Табл. 3 *Последовательность ответов пользователей на вопросы системы.*

Параметры предсказываются с учетом их популярности. Первый предсказываемый параметр – самый популярный первый параметр среди предыдущих пользователей (в нашем примере – *цена*). Если пользователь указал в качестве первого параметра *цену*, а в качестве второго – *оптический зум*, то ищется самый популярный третий атрибут в строках с номерами 1 и 2 (см. табл. 3).

Если пользователь указал слишком жесткие требования, то может оказаться, что искомых объектов нет в системе. В этом случае система последовательно и автоматически ослабляет ограничения пользователя на объекты. Методы выбора минимального множества ослабляемых параметров рассмотрены в [10]. В данной работе они рассматриваться не будут.

5.3. Выбор близких объектов

В отличие от методов, использующих жесткие ограничения, методы данного класса рекомендуют объекты, наиболее близкие к требованиям пользователей. В этом случае возможна рекомендация объектов, которые не полностью удовлетворяют требованиям.

Мера близости между требованиями и свойствами вычисляется по формуле:

$$similarity(p, REQ) = \frac{\sum_{r \in REQ} w_r \cdot sim(p, r)}{\sum_{r \in REQ} w_r} \qquad (9)$$

Здесь p – объект, REQ – множество требований пользователя. r – требование. w_i – вес данного требования, $sim(p,r)$ – мера близости объекта p к требованию r.

Численные свойства для пользователя делятся на три группы:

- «больше - лучше» (обозначаются MIB). Например, чем больше мегапикселей в камере, тем она считается лучше
- «меньше - лучше» (обозначаются LIB). Например, чем цена ниже, тем лучше для пользователя
- «ближе - лучше». Например, пользователю нужен монитор с определенной диагональю, больше или меньше – хуже.

Поэтому для каждого типа мера близости определяется следующим образом:

«Больше – лучше»:

$$sim(p,r) = \frac{\varphi_r(p) - \min(r)}{\max(r) - \min(r)} \quad (10)$$

«Меньше – лучше»:

$$sim(p,r) = \frac{\max(r) - \varphi_r(p)}{\max(r) - \min(r)} \quad (11)$$

«Ближе – лучше»:

$$sim(p,r) = 1 - \frac{|\varphi_r(p) - r|}{\max(r) - \min(r)} \quad (12)$$

Здесь *sim(p,r)* – мера близости объекта *p* к требованию *r*. *min(r), max(r)* – минимальные и максимальные значения свойства, соответственно. *r* в (12) – точное значение требования пользователя. $\varphi_r(p)$ – значение соответствующего свойства у объекта *p*.

6. Гибридные методы

Итак, выше были рассмотрены 3 основных вида систем рекомендаций. Каждый из них имеет свои плюсы и минусы [12]:

Метод	Плюсы	Минусы
Коллаборативная фильтрация	• достаточно иметь только данные о рейтингах	• проблема «холодного старта» (при добавлении новых пользователей и объектов) • нужно много данных о разных пользователях

Анализ содержимого	• достаточно иметь профиль текущего пользователя и содержимое объектов	• проблема «холодного старта» (только для новых пользователей) • работают преимущественно только с текстовыми данными
Основанный на знаниях	• не обязательно иметь много данных о покупках	• необходимость использования базы знаний

Табл. 4*. Положительные и отрицательные стороны методов рекомендаций.*

Крупные системы рекомендаций (СР), которые используют различные данные об объектах, пользователях, связях между ними, представляют собой сложную гибридную систему. Она является объединением различных подходов и алгоритмов: коллаборативной фильтрации, анализа содержимого, анализа социальных связей и связей между объектами, и т.д.

Выделяется три типа гибридных систем [9]:

- С монолитной организацией
- С параллельной организацией
- С конвейерной организацией

Схемы организации гибридных СР представлены на рисунках (2-4):

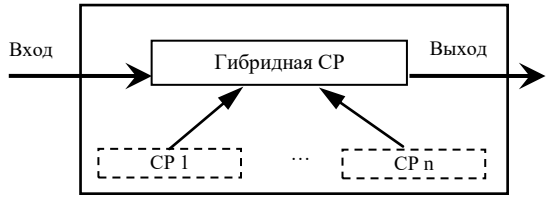


Рис. 2. *Монолитная организация гибридной СР.*

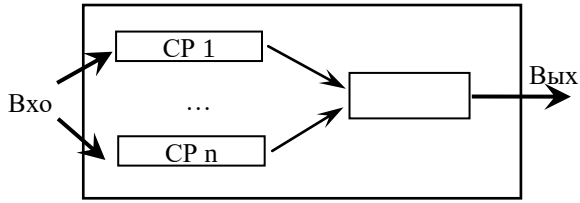


Рис. 3. *Параллельная организация гибридной СР.*

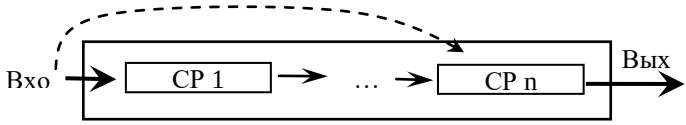


Рис. 4. *Конвейерная организация гибридной СР.*

6.1. Монолитная организация

В отличие от параллельной и конвейерной организаций гибридных систем, системы с монолитной организацией представляют собой один модуль, включающий в себя различные методы обработки имеющихся данных (см. рис. 2).

Большинство гибридных систем такого рода реализуют коллаборативную фильтрацию, усиленную особенностями содержимого. Например, если алгоритм коллаборативной фильтрации рекомендует некоторому пользователю несколько книг, то среди них можно выбрать наиболее подходящую с помощью сравнения жанров этих книг с популярными жанрами данного пользователя. Т.е. если пользователь предпочитает фантастику (что легко узнать, просмотрев жанры всех оцененных/прочитанных пользователем книг или его профиль), то среди книг двух жанров: история и фантастика, система предложит второй вариант.

Многие системы используют псевдооценки объектов пользователями:

$$v_{u,i} = \begin{cases} r_{u,i}, \text{если пользователь } u \text{ оценилобъект } i \\ c_{u,i}, \text{иначе} \end{cases} \quad (13)$$

Здесь $c_{u,i}$ – предсказание оценки на основе анализа содержимого.

Иногда пользователь запрашивает рекомендации объектов, удовлетворяющих некоторым критериям. Здесь уже могут использоваться методы, основанные на знаниях. Но, сочетание систем, использующих базы знаний с коллаборативными методами, пока не изучено и практически не используется.

6.2. Параллельная организация

Схема гибридной системы с параллельной организацией представлена на рис. 3. Исходные данные поступают на каждый модуль системы. После того, как все подсистемы выдадут ответ, начинает работать агрегирующий модуль. Есть три основных подхода выбрать множество наиболее релевантных рекомендаций.

Первый подход – смешать все рекомендации выдать их пользователю:

$$rec(u) = \bigcup_{k=1}^{n} rec_k(u) \quad (14)$$

rec(u) – результат работы «агрегатора» для пользователя *u*, $rec_k(u)$ – результат работы k-ого модуля системы для пользователя *u*.

Второй подход – для каждого объекта подсчитать сумму рейтингов, полученных параллельными модулями и вывести объекты с наибольшим рейтингом:

$$r(u,i) = \sum_{k=1}^{n} \beta_k \cdot r_k(u,i) \quad (15)$$

Здесь *r(u,i)* – нужная предсказанная оценка объекта *i* пользователем *u*, $r_k(u,i)$ – оценка, полученная *k*-м модулем, β_k – весовые коэффициенты.

И, наконец, третий подход – выбрать одну подходящую систему рекомендации, реализованную в одном из модулей для каждой конкретной ситуации. Например, если модуль коллаборативной фильтрации не дает ответ из-за нехватки данных (проблема «холодного старта»), система выбирает метод анализа содержимого.

6.3. Конвейерная организация

Схема гибридной системы с конвейерной организацией представлена на рис. 4. Задача рекомендации делится на этапы, которые выполняются последовательно. Результат работы очередного этапа используется в последующих.

7. Заключение

В данной работе были кратко изложены основные алгоритмы, используемые в системах рекомендаций. Различные методы используют различные данные о пользователях и об объектах. Каждый подход имеет свои достоинства и недостатки. Например, метод коллаборативной фильтрации рекомендует объекты, не имея никакого представления о том, что они собой представляют. Однако алгоритм не позволяет рекомендовать новые объекты. Проблему рекомендации новых объектов решают методы анализа содержимого. Но для их хорошей работы требуются текстовые данные об объектах. Если информации о пользователях, объектах и оценках недостаточно для этих алгоритмов, применяются методы, использующие базы знаний. При этом интерактивно выявляются требования пользователя.

Чем больше доступно данных, тем более точную систему рекомендации можно разработать, используя различные гибридные методы рекомендации.

Список литературы

[1]. P. Melville, V. Sindhwani; Recommender systems; Encyclopedia of Machine Learning, 2010

[2]. X. Su, T.M. Khoshgoftaar; A Survey of Collaborative Filtering Techniques; Advances in Artificial Intelligence, 2009

[3]. К. В. Воронцов; Методы коллаборативной фильтрации и тематического моделирования; 2011; [PDF] http://www.machinelearning.ru/wiki/images/9/95/Voron-ML-CF.pdf

[4]. А.Г. Гомзин; Collaborative filtering (Коллаборативная фильтрация); 2011; [PDF] http://modis.ispras.ru/seminar/wp-content/uploads/2011/11/CFFinal.pdf

[5]. D. Turdakov; Recommender system based on user-generated content; SYRCODIS, 2007

[6]. W.P. Jones, G.W. Furnas; Pictures of Relevance: A Geometric Analysis of Similarity Measures; Journal of the American society for information science. 38(6): 420-442, 1987; стр. 420-442.

[7]. S. Choi, S. Cha, C.C. Tappert; A Survey of Binary Similarity and Distance Measures; Journal of Systemics, Cybernetics and Informatics, Vol 8 No 1 2010, стр. 43-48

[8]. G. Shani, D. Heckerman, R. I. Brafman; An MDP-Based Recommender System; Journal of Machine Learning Research 6, 2005; стр. 1265–1295

[9]. А.В. Коршунов; Извлечение ключевых терминов из сообщений микроблогов с помощью Википедии; Труды Института системного программирования РАН, том 20, 2011. стр. 283-296

[10]. A. Huang; Similarity Measures for Text Document Clustering; NZCSRSC, 2008

[11]. D. Jannach, M. Zanker, A. Felfernig, G. Friedrich; Recommender Systems. An Introduction; Cambridge University Press 32 Avenue of the Americas, New York, NY 10013-2473, USA, 2011; 352 страниц

[12]. R. Burke; Hybrid Recommender Systems: Survey and Experiments; User Modeling and User-Adapted Interaction 12, 2002; стр. 331-370

Recommender systems: a survey of modern approaches

Andrey G. Gomzin, Anton V. Korshunov
gomzin@ispras.ru, korshunov@ispras.ru
ISP RAS, Moscow, Russia

Annotation. This paper is a survey of algorithms that are used in recommender systems. All related methods may be divided into three large groups: collaborative filtering, content-based and knowledge-based methods. These approaches are based on different source data that is used for recommendations computing. Collaborative filtering is based on analyzing ratings that user provide for items. The idea here is to recommend items that got high rating by the users that are similar with the user, who receives recommendations from the system. Content-based methods use available content of recommended items. Predominantly the content is represented as texts from items' names, descriptions and so on. Methods that analyze media content (images, videos, audio) are not considered in this paper. Knowledge-based methods analyze user requests that include information about what items do they want. The most popular methods are considered in this paper. Advantages and drawbacks of recommender technics of each group of methods are described. Moreover, composite techniques that allow drawbacks to be eliminated are considered in the paper. Composite technics assume that a lot of information about items is available. Hence, they combine collaborative filtering, content- and knowledge-based technics to provide more accurate recommendations for users.

Keywords: recommender system, collaborative filtering, knowledge bases, hybrid recommender system

References

[1]. P. Melville, V. Sindhwani; Recommender systems; Encyclopedia of Machine Learning, 2010
[2]. X. Su, T.M. Khoshgoftaar; A Survey of Collaborative Filtering Techniques; Advances in Artificial Intelligence, 2009
[3]. K. V. Vorontsov; Metody kollaborativnoj fil'tratsii i tematicheskogo modelirovaniya [Collaborative filtering and topic modeling techniques]; 2011; PDF: http://www.machinelearning.ru/wiki/images/9/95/Voron-ML-CF.pdf
[4]. A.G. Gomzin; Kollaborativnaya fil'tratsiya [Collaborative filtering] 2011; PDF: http://modis.ispras.ru/seminar/wp-content/uploads/2011/11/CFFinal.pdf
[5]. D. Turdakov; Recommender system based on user-generated content; SYRCODIS, 2007
[6]. W.P. Jones, G.W. Furnas; Pictures of Relevance: A Geometric Analysis of Similarity Measures; Journal of the American society for information science. 38(6): 420-442, 1987; стр. 420-442.
[7]. S. Choi, S. Cha, C.C. Tappert; A Survey of Binary Similarity and Distance Measures; Journal of Systemics, Cybernetics and Informatics, Vol 8 No 1 2010, стр. 43-48
[8]. G. Shani, D. Heckerman, R. I. Brafman; An MDP-Based Recommender System; Journal of Machine Learning Research 6, 2005; стр. 1265–1295

[9]. А.В. Коршунов; Izvlechenie klyuchevykh terminov iz soobshhenij mikroblogov s pomoshh'yu Vikipedii [Keyterm extraction from microblogs' messages using Wikipedia]. Trudy ISP RAN [Proceedings of ISP RAS], vol. 20, 2011. pp. 283-296
[10]. A. Huang; Similarity Measures for Text Document Clustering; NZCSRSC, 2008
[11]. D. Jannach, M. Zanker, A. Felfernig, G. Friedrich; Recommender Systems. An Introduction; Cambridge University Press 32 Avenue of the Americas, New York, NY 10013-2473, USA, 2011; 352 страниц
[12]. R. Burke; Hybrid Recommender Systems: Survey and Experiments; User Modeling and User-Adapted Interaction 12, 2002; стр. 331-370

Случайные графы, модели и генераторы безмасштабных графов

М.М. Берновский. mikhail.bermovskiy@phystech.edu
Н.Н. Кузюрин nnkuz@ispras.ru

Аннотация. В статье рассматриваются различные модели случайных графов, описывающие реальные сети, возникающие в различных областях: биологии, компьютерных науках, инженерии, социологии. Особое внимание уделено моделям, описывающим социальные сети.

Ключевые слова случайный граф, безмасштабная сеть, генератор случайных графов, Эрдеш, Боллобаш, Чунг, Янсон, Лучак.

1. Введение

Теория случайных графов стала интенсивно развиваться с конца 50-х годов прошлого века после публикации статей Эрдеша-Реньи об эволюции случайных графов. В этой модели все ребра появляются случайно и независимо с одинаковой вероятностью p и под эволюцией понимается изменение свойств графов с ростом вероятности p. Оказалось, что в некоторых значениях p происходит так называемый фазовый переход и свойства графа кардинально меняются. В этом направлении было получено много интересных и глубоких результатов. Однако, в начале 2000-х выяснилось, что модель Эрдеша-Реньи плохо описывает реальные графы, возникающие в различных областях, в частности в графы таких социальных сетей как Facebook, Twitter и т.п.

Это породило много новых исследований математических моделей случайных графов, о некоторых из которых и будет идти речь в данной статье.

В задачах описания динамиики социальных сетей основное значение имеет правильный выбор математической модели. На данный момент известно множество моделей случайных графов и безмасштабных (scale-free) сетей, некоторые из которых показали удовлетворительные результаты при сравнении с экспериментальными данными. Наиболее полным справочником по моделям безмасштабных сетей является [1].

Вообще говоря, модели социальных сетей можно разделить на три класса: модели случайных графов (модель Эрдеша-Реньи [2] и ее обобщения), простейшие модели безмасштабных сетей (модель Боллобаша [9] и ее

обобщения, модель копирования [8] и др.) и более гибкие модели безмасштабных сетей (модель Чунг-Лу [3], модель Янсона-Лучака [4] и др.) На наш взгляд третий класс моделей представляет наибольший интерес при моделировании больших реальных социальных сетей, таких как Facebook.
Данный обзор не претендует на полноту и служит в основном для обозначения трендов в обозначенной теме.

2. Модель Эрдеша-Реньи

Зафиксируем натуральное число n и рассмотрим множество $V = \{1,\ldots,n\}$. Таким образом мы задали множество вершин случайного графа. Зададим полный граф K_n на множестве вершин V. Пронумеруем ребра K_n: $e_1,\ldots,e_N$, где $N = \binom{n}{2}$. Зададим некотое $p \in [0,1]$ и будем выбирать ребра из множества $e_1,\ldots,e_n$ согласно схеме Бернулли. Таким образом мы получили случайный граф $G = (V,E)$. Формально выражаясь, мы имеем вероятностное пространство $G(n,p) = (\Omega_n, F_n, P_{n,p})$, в котором:

$$|\Omega_n| = 2^N, P_{n,p}(G) = p^{|E|} q^{\binom{n}{2}-|E|}.$$

Таким образом, в модели Эрдеша-Реньи каждое ребро *независимо* от других ребер входит в случайный граф с вероятностью p. Модель Эрдеша-Реньи на данный момент является самой изученной моделью случайных графов. Приведем несколько наиболее значительных с нашей точки зрения фактов о ней.

Будем далее говорить, что случайный граф обладает некоторым свойством почти наверное, если вероятность обладания этим свойством стремится к единице.

- **Треугольники в случайном графе.** Обозначим через $T_{3,n}$ случайную величину на пространстве $G(n,p)$, равную количеству треугольников в случайном графе G. Тогда верны следующие три теоремы:

Теорема 1 *Пусть* $\alpha(n) \to 0$ *при* $n \to \infty$. *Если* $p(n) = \frac{\alpha(n)}{n}, n \in \boldsymbol{N}$, *то почти наверное* $T_{3,n} = 0$ *(т.е. граф не содержит треугольников).*

Теорема 2 *Пусть* $p(n) \sim \frac{c}{n}$, *где* $c > 0$ – *константа. Тогда* $T_{3,n}$ *имеет асимптотически пуассоновское распределение с параметром* $\lambda = \frac{c^3}{6}$.

Теорема 3 *Пусть* $\omega \to \infty$ *при* $n \to \infty$. *Тогда если* $p(n) = \frac{\omega(n)}{n}, n \in \boldsymbol{N}$, *то* $T_{3,n} \geq 1$.

• **Связность случайного графа.** Одно из самых интересных свойств модели Эрдеша-Реньи – наличие фазового перехода:

Теорема 4 *Пусть* $p = \frac{c \ln n}{n}$. *Если* $c > 1$, *то почти наверное случайный граф связен. Если* $c < 1$, *то почти наверное случайный граф связным не является.*

Теорема 5 *Пусть* $p = \frac{c}{n}$. *Тогда при любом* $c < 1$ *существует такая константа* $\beta = \beta(c) > 0$, *что почти наверное каждая компонента случайного графа имеет не более* $\beta \ln n$ *вершин. При любом* $c > 1$ *существует такая константа* $\gamma = \gamma(c) \in (0,1)$, *что почти наверное среди компонент случайного графа есть одна (гигантская), число вершин которой не меньше* γn.

Т.о., модель Эрдеша-Реньи и ее простейшие обобщения являются слишком негибкими для моделирования больших социальных сетей, т.к., например, в них нет соответствующих фазовых переходов и распределения числа треугольников.

3. Наблюдения Барабаши-Альберт

В своих статьях [5], [6] и [7] авторы заметили следующие закономерности в веб-графе (графе, вершинами которого являются сайты, а ребра соответствуют ссылкам):

- Веб-граф разрежен (на n вершинах у него mn ребер, $m \in N$)
- Веб-граф подчиняется феномену «малого мира» (его диаметр $\approx 5-7$)
- Он подчиняется степенному закону:

$$\frac{|\{v : \deg(v) = d\}|}{n} \approx \frac{c}{d^{\lambda}}, c = const, \lambda \sim 2.1$$

На основании своих наблюдений авторы ввели понятие **предпочтительного присоединения (preferential attachment)**. Рассмотрим процесс генерации

графа. На n-ом шагу мы добавляем новую вершину n с m ребрами, инцидентыми ей, причем вероятность ребра к вершине i пропорциональна степени вершины i (см. Рисунок 1):

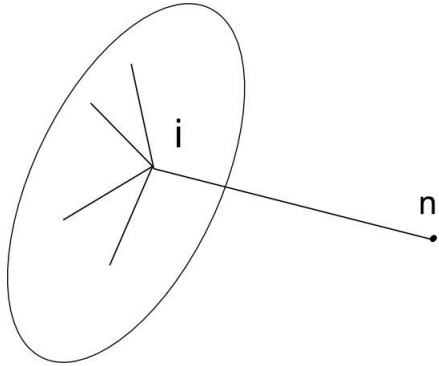


Рис. 1:Добавление вершины на n*-ом шагу.*

$$P(\textit{ребро из } n \textit{ в } i) = \frac{\deg(i)}{\Sigma_j \deg(j)}. \qquad (1)$$

Основных проблем со спецификацией модели Барабаши-Альберт две.

Во-первых, результирующий граф зависит от начального параметра m. Например, при $m = 1$ модель Барабаши-Альберт описывает генрацию дерева, если начальный граф – тоже дерево. Если начальный граф несвязный, то и все последующие тоже будут таковыми.

Во-вторых, трудность с предпочительным присоединением заключается в случайном выборе вершин (если $m \geq 2$), к которым присоединится новая вершина. Например, верна следующая теорема:

Теорема 1 *Пусть* $f(n), n \geq 2$ *– произвольная целочисленная функция, такая что:* $f(2) = 0$, $f(n) \leq f(n+1) \leq f(n)+1$ *для любых* $n \geq 2$ *и* $f(n) \to \infty$ *при* $n \to \infty$. *Тогда существует такой процесс генерации случайного графа* $T^{(n)}$, *удовлетвовряющий* (1), *что с вероятностью 1 в* $T^{(n)}$ *ровно* $f(n)$ *треугольников для достаточно больших* n.

Говоря менее формально, теорема **1** говорит о том, что если вы хотите иметь в графе с n вершинами $\log n$ треугольников, есть модель Барабаши-Альберт, которая выдаст такой результат.

4. Модель Боллобаша-Риордана

4.1. Генерация графа

Боллобаш и Риордан предложили следующую спецификацию модели Барабаши-Альберта. Построим последовательность случайных графов $\{G_1^n\}$, в которой у графа с номером n число вершин и ребер равно n. Преобразуем ее в последовательность $\{G_k^n\}$, в которой у графа с номером n число вершин равно n, а число ребер равно $kn, k \in N$.

Пусть $G_1^1 = (\{1\},\{(1,1)\})$. Предположим, что граф G_1^{n-1} уже построен. Ребер и вершин у него по $n-1$. Добавим вершину n и ребро (n,i), у которого $i \in \{1,\ldots,n\}$. Ребро (n,n) появится с вероятностью $\frac{1}{2n-1}$, ребро (n,i) – с вероятностью $\frac{\deg(i)}{2n-1}$, причем $\deg(i)$ – степень вершины i в графе G_1^{n-1}. Распределение вероятностей задано корректно, т.к.

$$\sum_{i=1}^{n-1} \frac{\deg(i)}{2n-1} + \frac{1}{2n-1} = 1.$$

Т.о., граф G_1^n построен, и он удовлетворяет принципу предпочтительного присоединения. Теперь перейдем к G_1^{kn}, у которого по kn вершин и ребер. Делим множество его вершин на последовательные куски размера k:

$$\{1,\ldots,k\},\{k+1,\ldots,2k\},\ldots,\{k(n-1)+1,\ldots,kn\}.$$

Каждый кусок примем за новую вершину, а ребра сохраним (ребра внутри куска становятся кратными петлями, ребра между разными кусками – кратными ребрами).

4.2. Основные результаты

Оказалось, что модель Боллобаша-Риордана довольно хорошо сходится с эмпирическими данными. За время изучения этой модели было получено огромное множество полезных результатов, мы же приведем только некоторые из них.

Теорема 1. Для любого $k \geq 2$ и любого $\varepsilon > 0$

$$P\left((1-\varepsilon)\frac{\ln n}{\ln\ln n} \leq diam\, G_k^n \leq (1+\varepsilon)\frac{\ln n}{\ln\ln n}\right) \to 1, n \to \infty. \qquad (2)$$

При $n: 10^7$, что соответствует Интернету образца 1999 года, имеем в (2) $\frac{\ln n}{\ln\ln n} \approx 6$, что совпадает с наблюдениями Барабаши-Альберт.

Теорема 2. Для любого $k \geq 1$ и для любого $d \leq n^{1/15}$

$$E\left(\frac{|\{i=1,\ldots,n : indeg_{G_k^n}(i)=d\}|}{n}\right) \sim \frac{2k(k+1)}{(d+k)(d+k+1)(d+k+2)}, \quad (3)$$

где $indeg_{G_k^n}(i)$ – количество ребер, имеющих вершину i своим левым концом в графе G_k^n.

Т.е. мы получили степенной закон d^{-3}. От условия $d \le n^{1/15}$ смогли избавиться сравнительно недавно, а для того чтобы получить степень 2.1 (соответствующую реальной степени веб-графа несколько лет назад) вместо 3, надо отойти от модели Боллобаша-Риордана.

Пусть H – фиксированный граф. Обозначим через $\#(H, G_k^n)$ случайную величину, равную количеству подграфов графа G_k^n, изоморфных графу H. В работе [9] приведены результаты о математическом ожилании этой величины.

Теорема 2 *Пусть* $k \ge 2$. *Пусть также* K_3 – *полный граф на трех вершинах. Тогда*

$$E\ \#(K_3, G_k^n) = (1+o(1)) \cdot \frac{k(k-1)(k+1)}{48} \cdot (\ln n)^3 \quad (4)$$

при $n \to \infty$.

Теорема 3 *Пусть фиксированы* $k \ge 2$ *и* $l \ge 3$. *Пусть также* C_l – *цикл на* l *вершинах. Тогда*

$$E\ \#(C_l, G_k^n) = (1+o(1)) \cdot c_{k,l} \cdot (\ln n)^l \quad (5)$$

при $n \to \infty$, где $c_{k,l}$ – положительная константа. Более того, при $k \to \infty$ имеем $c_{k,l} = \Theta(k^l)$.

А. Рябченко и Е. Самосват из Яндекса в модели, близкой к модели Боллобаша-Риордана, установили следующий факт:

Теорема 4 *Пусть задан граф* H, *степени вершин в котором равны* $d_1, \ldots, d_s$. *Обозначим через* $\#(d_i = m)$ *число вершин в* H, *степень каждой из которых равна* m. *Тогда*

$$E\ \#(H, G_k^n) = \Theta\ n^{\#(d_i=0)} \cdot (\sqrt{n})^{\#(d_i=1)} \cdot (\ln n)^{\#(d_i=2)}. \quad (6)$$

Зависимость от k занесена в константу Θ.

По (6)

$$E\ \#(K_3, G_k^n) = \Theta((\ln n)^3),$$

что согласуется с теоремой **2**. А для K_4 теорема **4** говорит, что его средняя частота в веб-графе постоянна. Т.о., «тетраэдров» в веб-графе почти нет.

Следует отметить, что последнее утверждение имеет мало общего со свойствами реального веба: в нем встречаются и тетраэдры, и клики большей мощности. Это связано с действием спамеров и агентств по раскрутке сайтов (групп в социальных сетях). Спам в модели Боллобаша-Риордана не учитывается.

5. Модель LCD [10]

Выделим в пространстве ось абсцисс и зафиксируем на ней $2n$ точек: 1, 2, 3, …, $2n$. Разобьем эти точки на пары, и элементы каждой пары соединим дугой, лежащей в верхней полуплоскости. Полученный объект назовем *линейной хордовой диаграммой (LCD)*. Дуги в LCD могут как пересекаться, так и лежать друг под другом, но *не* могут иметь общих вершин. Количество различных диаграмм равно

$$l_n = \frac{(2n)!}{2^n n!}. \qquad (7)$$

По каждой диаграмме построим граф с n вершинами и n ребрами. Процесс построения описан в алгоритме 1 и показан на рис. 2.

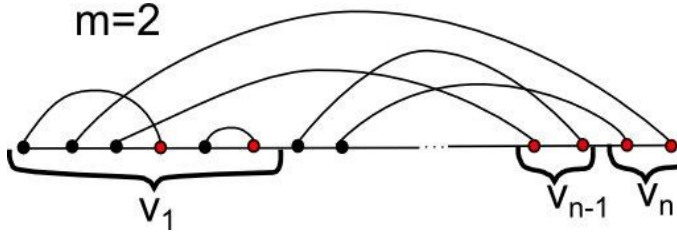


Рис. 2: LCD модель

Algorithm 1 Алгоритм построения графа по LCD

Идем слева направо по оси абсцисс, пока не встретим правый конец какой-либо дуги.
Пусть этот конец имеет номер i_1.
Объявляем набор $\{1, \ldots, i_1\}$ **первой вершиной** графа.
Снова повторяем шаг 1.
Пусть новый конец имеет номер i_2.
Объявляем набор $\{i_1 + 1, \ldots, i_2\}$ **второй вершиной** графа.
Продолжаем процедуру до прохода по всем точкам.
Ребра порождаем дугами.

Алгоритм 1

Теперь считаем LCD случайной, т.е. полагаем вероятность каждой диаграммы равной $1/l_n$, где l_n – общее число диаграмм из (7). Т.о. мы получаем случайные графы. В [7] показано, что такие графы по своим вероятностным

характеристикам почти неотличимы от графов G_1^n (см. предыдущий параграф). Графы с n вершинами и kn ребрами получаются так же, как и в модели Боллобаша-Риордана.

6. Модель Buckley-Osthus

Buckley, Osthus и другие исследователи предложили модификацию модели Барабаши-Альберт, в которой вершины обладают «*изначальной привлекательностью» («initial attractiveness»)*: вероятность того, что старая вершина будет выбрана соседом новой вершины пропорциональна ее входящей степени (in-degree) плюс константе, т.е. «изначальной привлекательности», т.е. am, где m – число ребер, входящих в новую вершину. Если $a=1$, то мы просто получаем модель Барабаши-Альберт, т.к. там используется полная степень, а каждая исходящая степень равна m. Buckley и Osthus в работе [11] обобщили результат для произвольной привлекательности a.

Для произвольного фиксированного положительного целого a граф $H_{a,1}^t s$ определяется так же, как и G_1^t в модели Боллобаша-Риордана, но при этом вероятность выбора старой вершины v_i определяется по-другому (см. статью [11] или обзор [9]). Оказывается, в этой модели распределение степеней тоже подчиняется степенному закону. Если обозначить через $\#_{a,m}^n(d)$ число вершин графа $H_{a,m}^n$ с входящей степенью d, то при определенных условиях:

$$\frac{\#_{a,m}^n(d)}{n} = \Theta(d^{-2-a}).$$

7. Модель копирования [8]

7.1. Гененерация графа

Фиксируем $\alpha \in (0,1)$ и $d \geq 1, d \in N$. В качестве начального графа возьмем любой d-регулярный граф. Пусть построен граф с номером t. Обозначим его через $G_t = (V_t, E_t)$, где $V_t = \{u_1, \ldots, u_s\}$, причем s отличается от t на число вершин начального графа, т.е. на константу, выражаемую через d. Добавим к G_t новую вершину u_{s+1} и d ребер, выходящих из u_{s+1}. Для этого сначала случайно выберем вершину $p \in V_t$. Затем строим ребра из u_{s+1} в V_t. На шаге с номером $i, 1 \leq i \leq d$, бросаем неоднородную монетку (падает решкой с вероятностью α, орлом – с вероятностью $1-\alpha$). Если выпала решка, то выпускаем ребро из u_{s+1} в случайную вершину из V_t. Если выпал орел, то

берем i-го по номеру соседа p. Последнее действие всегда возможно, т.к. исходный граф d-регулярен.

7.2. Основной результат

Теорема. Пусть $N_{t,r}$ – математическое ожидание числа вершин степени r в графе G_t. Тогда

$$\lim_{t\to\infty}\frac{N_{t,r}}{t}=\Theta\left(r^{-\frac{2-\alpha}{1-\alpha}}\right). \qquad (8)$$

Если вероятность копирования близка к 1 (α близка к 0), то показатель степени в (8) близок к 2.1, что соответствует наблюдениям Барабаши-Альберт. Также в модели копирования есть плотные двудольные графы, которые соответствуют спамерским структурам, отсутствующим в модели Боллобаша-Риордана.

8. Ориентированные безмасштабные графы

Большинство безмасштабных моделей описывают только неориентированные графы. Но поскольку большинство реальных сетей ориентировано, то логично создать ориентированные модели, в которых предпочтительное присоединение зависит от входящих и исходящих степеней. Такая модель была предложена Боллобашем, Риорданом и др. в работе [12]. В этой модели тоже получается степенной закон для входящих и исходящих степеней вершин.

9. Модель Чунг-Лу

9.1. Генерация графа

Пусть нам задано некоторое конечное множество вершин $V=\{v_1,\ldots,v_n\}$ и степень каждой вершины $d_i, i=\overline{1,n}$. Генерация графа $G=(V,E)$ происходит следующим образом:

• Формируем множество L, состоящее из d_i копий v_i для каждого i от 1 до n.

• Задаем случайные паросочетания на множестве L.

• Для вершин u и v из Vs количество ребер в графе G, соединяющее их, равно числу паросочетаний между копиями u и v в L.

Сгенерированный таким образом граф соответствует степенной модели $P(\alpha,\beta)$, описывающей графы, для которых:

$$|\{v\,|\,\deg(v)=x\}|=e^{\alpha}/x^{\beta}.$$

9.2. Основные результаты

• Почти наверное при $\beta < 1$ граф связен.

• При $1 < \beta < 2$ в графе есть гигантская компонента, при этом все остальные компоненты имеют размер $O(1)$.

• При $2 < \beta < \beta_0$ в графе есть гигантская компонента, при этом все остальные компоненты имеют размер $O(\log n)$. $\beta_0 \approx 3.47$ – решение уравнения $\zeta(\beta-2) - 2\zeta(\beta-1) = 0$.

• При $\beta = 2$ меньшие компоненты имеют размер $O(\log n / \log\log n)$

• При $\beta > \beta_0$ в графе почти наверное нет гигантской компоненты.

9.3. Сравнение с реальными сетями

Авторы проверяли гипотезу на графе звонков (максимальное число узлов 47×10^6). Качественно, прогнозы авторов верны (см. иллюстрации к статье [3] и ее последний раздел).

10. Модель Янсона-Лучака

10.1. Генерация графа

Рассмотрим упорядоченный набор $[n]s$ из ns вершин. Каждой вершине i присваивается вес W_i. Для простоты и понятности положим их независимо и одинаково распределенными случайными величинами со степенным хвостом:

$$P(W > x) = ax^{-\alpha},\ x \geq x_0, \quad (9)$$

с некоторыми константами $a > 0$ и $\alpha > 0$, и некоторым $x_0 > 0$. Мы обозначаем **наибольший** вес $W_{\max} = \max_i W_i$. Из (9) следует:

$$P(W_{\max} > tn^{1/\alpha}) \leq nP(W > tn^{1/\alpha}) = O(t^{-\alpha}). \quad (10)$$

Следует отметить, что $EW^\beta < \infty$, только если $\alpha > \beta$. В частности, при $\alpha < 2$ в случае распределения с экспоненциально неограниченным хвостом (heavy tail case) имеем $EW^2 = \infty$.

При условии, что нам дан набор весов $\{W_i\}_1^n$, соединим кажду пару вершин $\{i, j\}$ посредством E_{ij} параллельных ребер, где E_{ij} – независимые пуассоновские случайные величины с ожиданием:

$$E(E_{ij}) = \lambda_{ij} = b\frac{W_i W_j}{n}, \quad (11)$$

$b > 0$ – константа. В результате мы получим *мультиграф* $\hat{G}(n,\alpha)$. Далее мы можем стянуть параллельные ребра в одно, т.о. получив простой случайный граф $G(n,\alpha)$, в котором вершины i и j соединены с вероятностью

$$p_{ij} = 1 - \exp(-\lambda_{ij}), \qquad (12)$$

независимой для всех пар (i, j) таких, что $1 \le i < j \le n$.

10.2. Основные результаты

Обозначим за $\omega(G(n,\alpha))$ размер максимальной клики в графе $G(n,\alpha)$. В этиз обозначениях в модели Янсона-Лучака получены следующие результаты:

Теорема 5

(i) Если $0 < \alpha < 2$, то

$$\omega(G(n,\alpha)) = (c + o_p(1))n^{1-\alpha/2}(\log n)^{-\alpha/2},$$

где

$$c = ab^{\alpha/2}(1-\alpha/2)^{-\alpha/2}. \qquad (13)$$

(ii) Если $\alpha = 2$, то $\omega(G(n,\alpha)) = O_p(1)$.

(iii) Если $\alpha > 2$, то почти наверное $\omega(G(n,\alpha)) \in \{2,3\}$. Более того, при $n \to \infty$:

$$P(\omega(G(n,\alpha)) = 2) \to e^{-\frac{1}{6}(bE(W^2))^3} \qquad (14)$$

$$P(\omega(G(n,\alpha)) = 3) \to 1 - e^{-\frac{1}{6}(bE(W^2))^3} \qquad (15)$$

Основной задачей при работе с веб-графом является поиск клик в нем. Опишем несколько разных типов клик в модели Янсона-Лучака:

- Жадная клика (greedy clique)

$$\mathrm{K}_{gr} = \{i : i \sim j \text{ для всех } j \text{ с } W_j > W_i \text{ и } j \in \mathrm{K}_{gr}\}.$$

- Quasi top clique

$$\mathrm{K}_{qt} = \{i : i \sim j \text{ для всех } j \text{ с } W_j > W_i\}$$

- Full top clique

$$\mathrm{K}_{ft} = \{i : j \sim k \text{ для всех различных } j,k \text{ с } W_j, W_k \ge W_i\}.$$

Имеем $\boldsymbol{K}_{ft} \subseteq \boldsymbol{K}_{qt} \subseteq \boldsymbol{K}_{gr}$. Обозначим за $\boldsymbol{K}_{\max}$ максимальную клику в графе. Тогда

$$|\boldsymbol{K}_{ft}| \le |\boldsymbol{K}_{qt}| \le |\boldsymbol{K}_{gr}| \le |\boldsymbol{K}_{\max}| = \omega(G(n,\alpha)). \qquad (16)$$

Тогда верны следующие теоремы:

Теорема 6 *Если* $0 < \alpha < 2$, *то* $\boldsymbol{K}_{gr}$ *и* $\boldsymbol{K}_{qt}$ *имеют размер* $(1+o_p(1))\omega(G(n,\alpha))$. *С другой стороны,*

$$|\boldsymbol{K}_{ft}|/|\boldsymbol{K}_{\max}| \to^p 2^{-\alpha/2}.$$

Теорема 7 *Для любого* $\alpha > 0$ *сушествует алгоритм, который почти наверное находит в* $G(n,\alpha)$ *клику размером* $(1+o_p(1))\omega(G(n,\alpha))$ *за полиномиальное время.*

10.3. Основные результаты для схожих моделей

10.3.1 Детерминированные веса

Вместо того, чтобы выбирать веса независимо согласно распределению W, можно выбрать их из подходящей детерменированной последовательности W_i (как в модели Чунг-Лу), например

$$W_i = a^{1/\alpha}\frac{n^{1/\alpha}}{i^{1/\alpha}}, i = 1,\ldots,n. \quad (17)$$

В данной модели верными остаются все результаты для модели Янсона-Лучака.

11. Получение и обработка экспериментальных данных из социальных сетей

Большие социальные сети (например, Facebook и MySpace) не публикают свой веб-граф, поэтому на текущий момент его характеристики приходится получать экспериментально. Наиболее популярный способ для этого – создание популярных приложений в социальной сети (с числом пользователей не менее нескольких миллионов) и последующий сбор данных их пользователей. Сотрудники центра RUBINET [17] написали три приложения для Facebook с общим числом пользователей около 8 миллионов. Анонимизированные данные, полученные с помощью этих приложений, доступны on-line. Проанализировав различные характеристики пользователей (их географическое расположение, связи друг с другом и другие) и как они различаются в зависимости от типа приложения, исследователи пришли к следующим выводам:

- Граф отношений (interaction graph) пользователей обладает свойством «малого мира».
- При этом пользователи неигровых приложений образуют более плотные сообщества, чем пользователи игр. При этом

популярность приложений характеризуется степенным законом с экспоненциальным угасанием.

12. Генераторы синтетических веб-графов

Поскольку мы считаем разумным использование безмасштабных моделей типа Чунг-Лу [3] и Янсона-Лучака [4], то остановимся на генераторах, способных воспроизводить эти или похожие модели. В известных нам свободно распространяемых генераторах [13], [14], [15], [16] паросочетания реализуются не напрямую, а приближенно (например через введение понятия *потенцтального соседа* в [16]). Качество работы генератора определяется в сравнении некоторых параметров (например, размера плотных подграфов) получаемого графа с их теоретическими оценками. Наиболее полный отчет приведен авторами генератора DIGG [13].

13. Заключение

Существующие модели случайных безмасштабных графов можно условно разделить на три класса. В первый класс попадают модели со строгим математическим обоснованием выполнения степенного закона для порождаемых графов, для которых также доказан ряд важных свойств (подсчитан диаметр графа, коэффициент кластеризации, и т.п.), однако, показатель степени фиксирован и не может выбираться заранее. Типичным примером такой модели является модель Боллобаша-Риордана (степень равна 3) и ее обобщения (степень не меньше 2). Ко второму классу можно отнести модели, в которых показатель степенной зависимости может задаваться произвольно, что позволяет изучать эффекты фазовых переходов при его изменении. К числу таких моделей можно отнести модель Чунг-Лу и модель Лучака-Янсона. Доказательство наличия фазового перехода в них по свойству содержать большую клику при переходе степени через значение 2 представляет несомненный интерес и стимулирует исследование других свойств случайных графов с этой точки зрения. Наконец к третьему, самому многочисленному классу, относятся модели, в которых характеристики и свойства графов определяются эмпирически. Про такие модели не доказано никаких содержательных результатов (однако, возможно это дело недалекого будущего). Примерами таких моделей могут являться модель Forest Fire [18] и ряд других.

Кроме очевидной задачи моделирования динамики социальных сетей, модели безмасштабных сетей также применимы к таким популярным сейчас практическим задачам, как *поиск сообщетсв, максимизация распространения влияния, оценка безопасности и выносливости сложных сетей* и другие. Они позволяют сравнивать качество предлагаемых алгоритмов на целых семействах случайных графов и сравнивать получаемые результаты с предсказанными теоретически.

В целом, можно резюмировать, что данное направление исследований переживает период интенсивного развития и привлекает внимание математиков, инженеров, специалистов по компьютерным наукам, социологов и других специалистов.

Список литературы

[1] Mark Newman, Albert-László Barabási, Duncan J. Watts. The Structure and dynamics of networks. Princeton University Press, 2006.
[2] P. Erdös and A. Rényi. On the evolution of random graphs // Publ. Math. Debrecen. 1959. V. 6. P. 290-297.
[3] W. Aiello, F. Chung, L. Lu. A Random Graph Model for Massive Graphs. *STOC*'2000.
[4] S. Janson, T. Łuczak, I. Norros, (2010), Large cliques in a power-law random graph, J. Appl. Probab. V. 47, N. 4 1124-1135.
[5] L.-A. Barabási, R. Albert. Emergence of scaling in random networks. Science, 1999. V.286. P. 509-512
[6] L.-A. Barabási, R. Albert, H. Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. Physica, 2000. V. A281. P. 69-77.
[7] L.-A. Barabási, R. Albert, H. Jeong. Diameter of the world-wide web. Nature, 1999. V. 401. P. 130-131.
[8] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, E. Upfal. Stochastic models for the web graph. Proc. 41st Symposium on Foundations of Computer Science. 2000.
[9] B. Bollobás, O. Riordan. Mathematical results on scale-free random graphs. // Handbook of graphs and networks. Weinheim: Wiley-VCH, 2003. P. 1-34.
[10] А.М. Райгородский. Модели случайных графов. МЦНМО, 2011.
[11] P.G. Buckley, D. Osthus. Popularity based random graph models leading to scale-free degree sequence. Discrete Mathematics. Volume 282, Issues 1–3, 6 May 2004, Pages 53–68.
[12] B. Bollobás, C. Borgs, T. Chayes, O.M. Riordan. Directed scale-free graphs. ProceedingSODA '03 Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms
[13] Arthur R. Brady. A compact routing scheme for power-law networks using empirical discoveries in power-law graph topology. http://digg.cs.tufts.edu/readings/pdf/021.pdf
[14] Deepayan Chakrabarti et al. R-MAT: A Recursive Model for Graph Mining. http://www.cs.cmu.edu/ christos/PUBLICATIONS/siam04.pdf
[15] A. Medina et al. BRITE: Universal Topology Generation from a User's Perspective. http://www.cs.bu.edu/brite/publications/usermanual.pdf
[16] Joel C. Miller, Aric Hagberg. Efficient Generation of Networks with Given Expected Degrees. WAW, 2011.
[17] Atif Nazir, Saqib Raza, Chen-Nee Chuah. Unveiling Facebook: a measurement study of social network based applications. IMC'08.
[18] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. ACM Transact. on Knowledge Discovery from Data, 1(1), 2007.

Random graphs, models and generators of scale-free graphs

MM. Bernovskiy (ISP RAS, Moscow, Russia),
N.N. Kuzyurin (ISP RAS, Moscow, Russia, nnkuz@ispras.ru)

Abstract. In this paper various models of random graphs describing real networks arising in different research fields: biology, computer science, engineering, sociology are considered. Particular attention is paid to models describing social networks. We start with observation of classical Erdos-Renyi model and basic facts concerning properties of random graphs depending on the value p of the probability of appearance each edge in a graph. Then we observe so-called scale-free model of random graphs proposed by Barabassi and Albert and some its generalizations. Some mathematical results about properties of random graphs in such models are described. For one such model of Bollobas-Riordan we describe results concerning the diameter of random graph, the number of verticies of given degree (power-law distribution) and other different properties. This class of models can be characterized by the property that so-called power law constant cannot be chosen in advance and has one fixed value. For example in Bollobas-Riordan model this constant is 3. Finely we observe more general models of random graphs such as Aiello-Chung-Lu model where the power law constant can be chosen as a parameter. In such models there are interesting results concerning evolution properties of random graphs depending on the value of parameter of power law constant. The main example of such property is the size of maximum clique in random graph which we consider in this paper.

Keywords: random graph, scale-free network, random graph generator, Erdos, Bollobas, Chung, Janson, Luczak models

References

[1]. Newman M., Barabási A.L., Watts D. J.. The Structure and dynamics of networks. Princeton University Press, 2006.
[2]. Erdös P., Rényi A. On the evolution of random graphs. Publ. Math. Debrecen. 1959. V. 6. P. 290-297.
[3]. Aiello W., Chung F., Lu L. A Random Graph Model for Massive Graphs. STOC '2000.
[4]. Janson S., Łuczak T., Norros I., Large cliques in a power-law random graph, J. Appl. Probab. 2010. V. 47, N. 4, pp. 1124-1135.
[5]. Barabási L.A., Albert R. Emergence of scaling in random networks. Science, 1999. V.286. P. 509-512
[6]. Barabási L.A., Albert R., Jeong H.. Scale-free characteristics of random networks: the topology of the world-wide web. Physica, 2000. V. A281. P. 69-77.
[7]. Barabási L.A., Albert R., Jeong H. Diameter of the world-wide web. Nature, 1999. V. 401. P. 130-131.
[8]. Kumar R., Raghavan P., Rajagopalan S., Sivakumar D., Tomkins A., Upfal E. Stochastic models for the web graph. Proceedings of the 41st Symposium on Foundations of Computer Science. 2000.

[9]. Bollobás B., Riordan O. Mathematical results on scale-free random graphs. Handbook of graphs and networks. Weinheim: Wiley-VCH, 2003. P. 1-34.
[10]. Raigoroskii A.M. Modeli sluchainykh grafov [Random graph models]. MCNMO [Moscow center of continuous mathematical education], 2011. 136 , p. (in Russian)
[11]. Buckley P.G., Osthus D. Popularity based random graph models leading to scale-free degree sequence. Discrete Mathematics. Volume 282, Issues 1–3, 6 May 2004, pp. 53–68.
[12]. Bollobás B., Borgs C., Chayes T., Riordan O.M. Directed scale-free graphs. Proceedings SODA '03 Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms.
[13]. Brady A.R. A compact routing scheme for power-law networks using empirical discoveries in power-law graph topology. http://digg.cs.tufts.edu/readings/pdf/021.pdf
[14]. Chakrabarti D., et al. R-MAT: A Recursive Model for Graph Mining. http://www.cs.cmu.edu/ christos/PUBLICATIONS/siam04.pdf
[15]. Medina A., et al. BRITE: Universal Topology Generation from a User's Perspective. http://www.cs.bu.edu/brite/publications/usermanual.pdf
[16]. Miller J. C., Hagberg A. Efficient Generation of Networks with Given Expected Degrees. WAW, 2011.
[17]. Nazir A.,, Raza C., Chuah C.-N.. Unveiling Facebook: a measurement study of social network based applications. IMC'08.
[18]. Leskovec J., Kleinberg J., Faloutsos C. Graph evolution: Densification and shrinking diameters. ACM Transact. on Knowledge Discovery from Data, 1(1), 2007.

Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ

В.А.Захаров (ИСП РАН), Т.А. Новикова (ф-т ВМК МГУ им. М.В. Ломоносова)
{zakh@cs.msu.su} {taniaelf@mail.ru}

Аннотация. Логико-термальная эквивалентность программ – это одно из наиболее слабых отношений эквивалентности программ, аппроксимирующих отношение функциональной эквивалентности и обладающих разрешающим алгоритмом. В данной статье предложена новая модификация алгоритма проверки логико-термальной эквивалентности программ, основанная на операции вычисления точной нижней грани в решетке конечных подстановок. Показано, что трудоемкость предложенного алгоритма оценивается величиной $O(n^6)$, где n - размер анализируемых программ. Полученная верхняя оценка сложности задачи проверки логико-термальной эквивалентности программ существенно меньше ранее известных полиномиальных оценок сложности указанной задачи.

Ключевые слова: программа, логико-термальная эквивалентность программ, подстановка, антиунификация, сложность.

1. Введение

Две программы π_1 и π_2 считаются логико-термально эквивалентными, если для любой синтаксически допустимой трассы tr' в одной из программ существует такая трасса tr'' в другой программе, что в обеих трассах логические условия (предикаты) проверяются в одной и той же последовательности для одних и тех же наборов значений переменных. Логико-термальная (л-т) эквивалентность программ была введена в статье [1]. Интерес к л-т эквивалентности программ был обусловлен двумя ее свойствами:

1) л-т эквивалентность программ π_1 и π_2 влечет функциональную эквивалентность этих программ [2],

2) отношение л-т эквивалентности программ разрешимо за полиномиальное время [2,3].

Благодаря этим качествам алгоритмы проверки л-т эквивалентности программ можно использовать для решения многочисленных задач анализа поведения и

преобразования программ, включая задачу верификации [4-8], оптимизации [9,10] и реорганизации (рефакторинга) программ [11-13] .

Было установлено также, что л-т эквивалентность – это одно из наиболее слабых разрешимых отношений эквивалентности программ, аппроксимирующих отношение функциональной эквивалентности. Первый алгоритм, разрешающий л.-т. эквивалентность стандартных схем программ был предложен в статье [1]. Сложность его оценивается двойной экспонентой, зависящей от размера проверяемых программ. Впоследствии были созданы более эффективные алгоритмы [4,6]; в частности, в работах [2,3] (см. также [16]) был разработан алгоритм проверки л.-т. эквивалентности программ за время $O(n^7)$, где n - размер анализируемых программ.

Как было показано в статьях [14,15], л-т эквивалентность программ лежит на самой границе, отделяющей разрешимые виды эквивалентности программ от неразрешимых. Поскольку определение л-т эквивалентности не опирается на интерпретации базовых компонентов программ (функций и предикатов), задача проверки л-т эквивалентности программ может быть отнесена к задачам статического анализа программ, и для ее решения может быть привлечены методы вычисления неподвижных точек монотонных операторов на решетках [16]. На основании этих методов и строятся алгоритмы проверки л-т эквивалентности программ. В настоящей статье в качестве такой решетки была выбрана решетка конечных подстановок, исследованная в статьях [17,18]. В многочисленных работах (см. [17-23]) были предложены различные алгоритмы антиунификации подстановок, вычисляющие точные нижние грани в решетке подстановок. На основании этих алгоритмов в статье [24] был предложен новый алгоритм проверки л-т эквивалентности программ и показано его применение для решения новой задачи анализа программ – задачи унификации программ. В настоящей статье мы показываем, что верхняя оценка сложности предложенного алгоритма составляет величину $O(n^6)$, где n - размер анализируемых программ, что существенно улучшает ранее известные верхние оценки сложности задачи проверки л-т эквивалентности программ.

Содержание настоящей статьи таково. Во втором ее разделе приводятся основные понятия алгебры конечных подстановок с операциями композиции и антиунификации. В третьем разделе статьи коротко определяются модель стандартных схем программ и отношение л-т эквивалентности программ в этой модели, а также описан алгоритм проверки л-т эквивалентности программ в том виде, в котором он был предложен в статье [24]. Поскольку в этом алгоритме интенсивно используются операции композиции и антиунификации подстановок, сложность этих операций и самого алгоритма существенно зависят от способов представления этих подстановок. В четвертом разделе рассмотрены графовые способы представления конечных подстановок, приведены верхние и нижние оценки сложности указанных операций для подстановок, реализованных в виде конечных ациклических

ориентированных графов. Эти оценки свидетельствуют о том, что для повышения эффективности алгоритма проверки л-т эквивалентности программ класс подстановок, используемых в этом алгоритме, должен быть существенно сокращен. Поэтому в пятом разделе статьи вводится специальный подкласс редуцированных подстановок (определение 1), которые используются в дальнейшем в модифицированном алгоритме проверки л-т эквивалентности программ. В этом же разделе описана процедура редукции подстановок и исследованы некоторые алгебраические свойства редуцированных подстановок (утверждения 3-5 и теорема 6). В шестом разделе статьи введена операция редуцированной антиунификации (определение 3) и установлены некоторые ее алгебраические свойства (теоремы 7-9), необходимые для применения этой операции в модифицированном алгоритме проверки л-т эквивалентности программ. В заключительном седьмом разделе статьи описана процедура вычисления редуцированной антиунификации для подстановок, представленных в виде размеченных ациклических ориентированных графов. В теореме 10 показано, что время выполнения этой процедуры и размер графа, представляющего результат редуцированной антиунификации, ограничены величиной, линейно зависящей от размеров графов, представляющих подстановки-операнды. На основе этой оценки в теореме 11 последнего раздела статьи получен основной результат работы – верхняя оценка сложности по времени $O(n^6)$ для модифицированного алгоритма проверки л-т эквивалентности программ.

2. Алгебра конечных подстановок

Для заданных множеств переменных Var, функциональных символов F и предикатных символов P обозначим записью $Term(F,Var)$ множество термов, а записью $Atom(P,F,Var)$ множество атомарных формул (атомов), которые определяются обычным образом (см. [17,18]). Для каждого терма t обозначим записью Var_t множество переменных, входящих в терм t.

Пусть X и Y – два конечных множества переменных. Подстановкой назовем всякое отображение $\theta: X \to Term(F,Y)$, сопоставляющее каждой переменной из X некоторый терм из $Term(F,Y)$. Множество всех таких подстановок условимся обозначать записью $Subst(X,F,Y)$. Если $X=\{x_1,x_2,\ldots,x_n\}$ и $\theta(x_i)=t_i$ для всех i, $1\le i\le n$, то подстановка θ однозначно определяется множеством (списком) пар $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}s$. Если $t_i = x_i s$, то пара (связка) $\{x_i/t_i\}$ может быть опущена в этом списке. Запись Dom_θ будет обозначать множество переменных $\{x:\theta(x)\ne x\}$, а запись Var_θ будет использоваться для обозначения множества переменных $\bigcup_{i=1}^{n} Var_{t_i} s$, входящих в

состав всех термов подстановки θ. Результатом применения подстановки θ к терму t является терм $t\theta$, получающийся одновременной заменой в t каждой переменной x_i термом $\theta(x_i)$. Композиция $\theta\eta$ подстановок $\theta \in Subst(X,F,Y)$, $\eta \in Subst(Y,F,Z)$ – это подстановка из множества $Subst(X,F,Z)$, которая определяется равенством $\theta\eta(x) = (\theta(x))\eta$ для каждой переменной $x, x \in X$. Всякая биекция $\theta : Y \to Y$ называется переименованием.

На множестве подстановок $Subst(X,F,Y)$ определим отношения предпорядка $\prec$ и эквивалентности $\approx$. Для пары подстановок θ_1, θ_2 отношение $\theta_1 \prec \theta_2$ выполняется, если есть такая подстановка $\eta \in Subst(Y,F,Y)$, что $\theta_2 = \theta_1\eta$, и отношение $\theta_1 \approx \theta_2$ выполняется, если $\theta_2 = \theta_1\rho$ для некоторого переименования ρ. Если для пары подстановок θ_1, θ_2 выполняется отношение $\theta_1 \prec \theta_2$, то подстановку θ_1 будем называть *прототипом* подстановки θ_2, а подстановку θ_2 – *примером* подстановки θ_1. Отношение предпорядка $\prec$ порождает на множестве классов эквивалентности $Subst(X,F,Y)/\approx$ отношение частичного порядка $\leq$. Частично упорядоченное множество $(Subst(X,F,Y)/\approx, \leq)$ образует решетку, наименьшим элементом которой является класс эквивалентности, порожденный так называемой пустой подстановкой $\{x_1 / y_1, x_2 / y_2, \ldots, x_n / y_n\}s$. Чтобы сделать решетку подстановок полной, добавим к множеству подстановок в качестве наибольшего элемента специальную мнимую подстановку τ, удовлетворяющую равенствам $\tau\theta = \theta\tau = \tau s$ и $E_1\tau = E_2\tau s$ для любой подстановки θ и термов t_1, t_2. Операция взятия точной нижней грани называется антиунификацией подстановок и обозначается символом $\downarrow$. Решетка $(Subst(X,F,Y)/\approx, \leq)s$ удовлетворяет условию обрыва убывающих цепей. Далее запись $\theta_1 \downarrow \theta_2 s$ будем использовать для обозначения произвольной подстановки из классов эквивалентности $\theta_1^{\approx} \downarrow \theta_2^{\approx}$. Для любой пары подстановок $\theta', \theta' \in Subst(X',F,Y)$ и $\theta'', \theta'' \in Subst(X'',F,Y)$ в случае $X' \cap X'' = \varnothing$, мы будем использовать запись $\theta' \cup \theta''$ для обозначения подстановки ηs, которая представляет собой теоретико-множественное объединение связок подстановок θ' и θ''.

3. Модель последовательных императивных программ

В качестве математической модели последовательных императивных программ мы рассматриваем множество конечных размеченных ориентированных графов. Опишем вкратце устройство этой модели. Пусть задано некоторое конечное множество переменных X. Каждой вершине v

графа π, моделирующего императивную программу, приписана атомарная формула A_v из множества $Atom(P,F,X)$. Из каждой вершины исходят две дуги, одна из которых помечена символом 0, а другая – символом 1. Кроме того, каждой дуге, ведущей в графе π из вершины u в вершину v, приписана подстановка θ_{uv} из множества $Subst(X,F,X)$. Одна из вершин v_{in} графа π особо выделена в качестве входа в программу, а другая вершина v_{out} играет роль выхода из программы. Предполагается также, что через каждую вершину графа π проходит некоторый маршрут, ведущий из входа программы в ее выход.

Вершины графа π соответствуют точкам программы, в которых проводится проверка условий и ветвление потока управления программы. Атомарные формулы, приписанные вершинам, соответствуют тестам (предикатам), а дуги графа π - линейным участкам программы. Если линейный участок представляет собой последовательность операторов присваивания $x_{i_1} := t_1; x_{i_2} := t_2; \ldots x_{i_k} := t_k;$, то соответствующей дуге приписана подстановка $\theta = \{x_{i_1} / t_1\}\{x_{i_2} / t_2\}\cdots\{x_{i_k} / t_k\}$, представляющая собой композицию односвязочных подстановок для всех операторов этого линейного участка.

Пусть задан некоторый маршрут из входа в программу π в ее выход

$$w = v_0 \xrightarrow{\sigma_1,\theta_1} v_1 \xrightarrow{\sigma_2,\theta_2} v_2 \xrightarrow{\sigma_1,\theta_1} \cdots v_{n-1} \xrightarrow{\sigma_n,\theta_n} v_n$$

Тогда последовательность пар

$$lth(w) = (A_{v_0}\mu_0,\sigma_1),(A_{v_1}\mu_1,\sigma_2),\ldots,(A_{v_{n-1}}\mu_{n-1},\sigma_n),(A_{v_n}\mu_n,1),$$

где $\mu_0 = \varepsilon$ и $\mu_i = \theta_1\theta_2\cdots\theta_i$ для каждого $i, 1 \le i \le n,$ называется *логико-термальной историей* маршрута ws. *Детерминантом* программы π называется множество

$Det(\pi) = \{lth(w) : w$ - маршрут в программе πs из входа в выход$\}$.

Две программы π_1 и $\pi_2 s$ считаются *логико-термально* (*л-т*) *эквивалентными*, если справедливо равенство $Det(\pi_1) = Det(\pi_2)$. Отношение л-т эквивалентности программ является отношением эквивалентности, которое аппроксимирует функциональную эквивалентность программ, т.е. две л-т эквивалентные программы π_1 и π_2 для любой интерпретации и для любого начального значения переменных вычисляют одинаковые результаты (значения переменных при достижении вычислением выхода из программы). Более подробно об отношениях функциональной эквивалентности и л-т эквивалентности программ можно прочесть в статьях [1-10] и монографии [16]. Задача проверки л-т эквивалентности программ состоит в том, чтобы для

произвольной заданной пары программ π_1 и π_2 проверить, являются ли эти программы л-т эквивалентными.

В статье [24] был предложен следующий алгоритм проверки л-т эквивалентности программ. Предположим, что задана пара программ π_1 и π_2 над множеством переменных $X = \{x_1, x_2, \ldots, x_n\}$. Работа алгоритма проверки л-т эквивалентности этих программ состоит из четырех этапов.

На первом этапе проводится переименование переменных в этих программах: все вхождения переменных $x_1, x_2, \ldots, x_n$ в программу π_1 заменяются на переменные $x_1', x_2', \ldots, x_n'$ соответственно, а все вхождения переменных $x_1, x_2, \ldots, x_n$ в программу π_2 заменяются на переменные $x_1'', x_2'', \ldots, x_n''$ соответственно. Получившиеся в результате указанного переименования переменных варианты программ π_1 и π_2 условимся обозначать π' и π''.

На втором этапе для пары программ π' и π'' строится граф логически совместных трасс $\Gamma[\pi', \pi'']$. Его устройство таково. Вершинами графа $\Gamma[\pi', \pi'']$ служат всевозможные пары (u', u''), где u' - вершина программы π', а u'' - вершина программы π''. Каждой вершине (u', u'') приписывается пара атомов $(A_{u'}, A_{u''})$, которыми были помечены вершины u' и u'' в программах π' и π''. В том случае, если в программе π' из вершины u' в вершину v' ведет дуга с пометками σ, θ', где $\sigma \in \{0,1\}$, и $\theta' = \{x_1'/t_1', x_2'/t_2', \ldots, x_n'/t_n'\}$, а и в программе π'' из вершины u'' в вершину v'' ведет дуга с пометками σ, θ'', где $\theta'' = \{x_1''/t_1'', x_2''/t_2'', \ldots, x_n''/t_n''\}$, то в графе $\Gamma[\pi', \pi'']$ из вершины (u', u'') в вершину (v', v'') ведет дуга с пометкой $\theta = \theta' \cup \theta'' = \{x_1'/t_1', \ldots, x_n'/t_n', x_1''/t_1'', \ldots, x_n''/t_n''\}$.

На третьем этапе применяется процедура вычисления разметки вершин графа $\Gamma[\pi', \pi'']$ подстановками из множества $Subst(X' \cup X'', F, Y)$, где $Y = \{y_1, y_2, \ldots\}$ - бесконечное множество переменных, отличных от переменных множеств $X' = \{x_1', x_2', \ldots, x_n'\}$ и $X'' = \{x_1'', x_2'', \ldots, x''\}_n$. В начале работы этой процедуры вершина (u_{in}', u_{in}'') графа $\Gamma[\pi', \pi'']$, где u_{in}' и u_{in}'' - точки входа в программы π' и π'', помечается подстановкой $\eta_0 = \{x_1'/y_1, x_1''/y_1, x_2'/y_2, x_2''/y_2, \ldots, x_n'/y_n, x_n''/y_n\}$, а все остальные вершины графа $\Gamma[\pi', \pi'']$ помечаются максимальной в решетке подстановок $(Subst(X' \cup X'', F, Y)/\approx, \leq)$ подстановкой τ. Далее, для каждой пары вершин (u', u'') и (v', v''), помеченных подстановками $\eta_{u'u''}$ и $\eta_{v'v''}$ и соединенных

дугой, которой приписана подстановка θ, вычисляется подстановка $\mu = \theta\eta_{u'u''} \downarrow \eta_{v'v''}$, которая приписывается вершине (v', v'') вместо подстановки $\eta_{v'v''}$. Эта процедура выполняется до тех пор, пока в графе $\Gamma[\pi', \pi'']$ для каждой пары вершин (u', u'') и (v', v'') не будет выполняться равенство $\eta_{v'v''} \approx \theta\eta_{u'u''} \downarrow \eta_{v'v''}$.

На последнем этапе работы алгоритма проводится проверка выполнимости следующих двух условий:

1. В графе $\Gamma[\pi', \pi'']$ из вершины (u'_{in}, u''_{in}) достижима хотя бы одна из вершин вида (u'_{out}, u'') или (u', u''_{out}), где $u' \neq u'_{out}, u'' \neq u''_{out}$;
2. Некоторая вершина (u', u'') графа $\Gamma[\pi', \pi'']$ помечена подстановкой $\eta_{u'u''}$, для которой справедливо соотношение $A_{u'}\eta_{u'u''} \neq A_{u''}\eta_{u'u''}$.

Если хотя бы одно из этих условий выполнено, то исходные программы π_1 и π_2 признаются л-т неэквивалентными. В противном случае рассматриваемые программы л-т эквивалентны.

Завершаемость описанной процедуры вычисления разметки вершин графа $\Gamma[\pi', \pi'']$ следует из свойства фундированности отношения порядка в решетке подстановок $(Subst(X' \cup X'', F, Y)/\approx, \leq)$. Корректность и полнота алгоритмы проверки л-т эквивалентности программ следуют из двух теорем, в которых установлены алгебраические свойства операции антиунификации подстановок.

Теорема 1. [24] Для любой подстановки $\theta \in Subst(X, F, X)$ и пары подстановок $\eta_1, \eta_2 \in Subst(X, F, Y)$ справедливо равенство

$\theta\eta_1 \downarrow \theta\eta_2 = \theta(\eta_1 \downarrow \eta_2)$.

Теорема 2. [24] Для любой пары атомов $A, B \in Atom(P, F, X)$ любой пары подстановок $\eta_1, \eta_2 \in Subst(X, F, Y)$ справедливо соотношение

$A\eta_1 = B\eta_1 \wedge A\eta_2 = B\eta_2 \quad \Leftrightarrow \quad A(\eta_1 \downarrow \eta_2) = B(\eta_1 \downarrow \eta_2)$.

Для того чтобы оценить сложность предложенного алгоритма проверки л-т эквивалентности программ, необходимо рассмотреть некоторые способы представления подстановок и оценить сложность операций композиции и антиунификации в зависимости от этих представлений.

4. Графовые представления подстановок

Эффективность алгоритмов вычисления композиции и антиунификации подстановок существенно зависит от тех структур данных, которые используются для представления термов и подстановок. Для представления

термов целесообразно использовать графовые конструкции, которые легко моделируются ссылочными структурами данных. Опишем общий принцип, положенный в основу этих конструкций.

Рассмотрим произвольный конечный ациклический ориентированный граф $G=(V,E)$ с множеством вершин V и множеством дуг E. Размером $|G|$ графа G будем считать суммарное количество его вершин и дуг. *Разметкой* АОГ G назовем отображение, которое приписывает каждой вершине графа v символ s_v, который является либо функциональным символом из множества F или переменной из множества Var, а каждой дуге – некоторое натуральное число. *Правильной разметкой* АОГ G назовем *разметку*, удовлетворяющую следующим двум требованиям:

- если из вершины v не исходит ни одной дуги, то эта вершина помечена либо константой, либо переменной;
- если из вершины v исходят n дуг, где $n>0$, то эта вершина помечена функциональным символом местности n, а все исходящие дуги помечены попарно различными числами из множества $\{1,2,\ldots,n\}$.

Если в правильно помеченном АОГ G из вершины v в вершину u ведет дуга, помеченная числом m, то вершину u будем называть *m-наследником* вершины v. Каждой вершине v правильно размеченного АОГ G можно однозначно сопоставить терм t_v, который *реализуется в вершине* v, руководствуясь следующими правилами:

- если из вершины v не исходит ни одной дуги, то в этой вершине реализуется терм s_v;
- если из вершины v исходят n дуг, ведущие в вершины $u_1,u_2,\ldots,u_n$, и при этом для любого $i, 1\le i\le n,$ дуга, ведущая в вершину u_i, помечена натуральным числом i, то в вершине v реализуется терм $s_v(t_{u_1},t_{u_2},\ldots,t_{u_n})$.

Правильно размеченный АОГ G *реализует* конечное множество термов $T, T\subseteq Term(F,Var),$ если для каждого терма $t, t\in T,$ в графе G существует вершина v, для которой $t_v=t$.

Среди различных способов реализации множеств термов T, посредством правильно размеченных АОГ, наиболее широкое распространение получили два способа – *древесный* и *приведенный*.

В первом случае АОГ G представляет собой конечное множество деревьев (лес), и каждому терму из множества T взаимно однозначно соответствует дерево из этого леса, в корне которого реализуется этот терм. Это самый

простой способ графового представления множеств термов: древесная реализация множества термов легко строится по строковой записи термов и легко получается из произвольного АОГ, реализующего множество T, путем «расклейки» вершин этого графа.

АОГ G, реализующий множество термов T, называется *приведенным*, если он удовлетворяет двум требованиям:

- в каждой вершине АОГ G реализуется некоторый подтерм какого-либо терма из множества T;
- в разных вершинах АОГ G реализуются разные термы.

Для построения приведенного АОГ из произвольного графового представления множеств термов нужно выполнить специальную процедуру редукции. Эта процедура заключается в удалении всех вершин, которые не удовлетворяют первому из указанных выше требований и в применении к АОГ (до тех пор пока это возможно) следующего правила: если две вершины u и v одинаково помечены, и при этом m-наследник вершины u совпадает m-наследником вершины v для каждого натурального числа m, то эти вершины необходимо склеить, т.е. удалить одну из этих вершин вместе с исходящими из нее дугами, а все ведущие в эту вершину дуги направить в другую вершину рассматриваемой пары. Нетрудно видеть, что в результате применения описанной процедуры к АОГ, реализующему множество термов T, будет построен приведенный граф, реализующий то же самое множество термов. Процедура может быть осуществлена за время, линейное относительно размеров исходного АОГ.

Приведенные АОГ - наиболее компактный способ представления множеств термов. Существуют такие последовательности термов, размер древесного представления которых оценивается экспонентой, зависящей от размеров приведенного представления этих же термов.

Размеченные АОГ, реализующие множества термов, можно использовать также и для реализации подстановок. Пусть заданы подстановка $\theta = \{x_1 / t_1, x_2 / t_2, \ldots, x_n / t_n\}$ из множества $Subst(X, F, Y)$ и АОГ G, реализующий множество термов $\{t_1, t_2, \ldots, t_n\}$. Сопоставим каждой переменной $x_i, 1 \le i \le n$, ту вершину v графа G, в которой реализуется соответствующий терм t_i; переменную x_i будем называть *заголовком* вершины v. Размеченный таким образом АОГ G будем называть *графовой реализацией* подстановки θ.

Графовое представление подстановок оказывает существенное влияние на сложность задач вычисления операций композиции и антиунификации подстановок. Если подстановки θ и η имеют древесные реализации G_θ и G_η, то

- размер и сложность вычисления древесной реализации $G_{\theta\eta}$ композиции этих подстановок оценивается величиной $\Omega(|G_\theta||G_\eta|)$,
- размер древесной реализации $G_{\theta\downarrow\eta}$ антиунификации этих подстановок оценивается величиной $\Omega(\min(|G_\theta|,|G_\eta|))$, а сложность ее вычисления оценивается величиной $\Omega(|G_\theta||G_\eta|)$.

Если же подстановки θ и η имеют приведенные реализации H_θ и H_η, то

- размер и сложность вычисления приведенной реализации $H_{\theta\eta}$ композиции этих подстановок оценивается величиной $\Omega(|H_\theta|+|H_\eta|)$,
- размер и сложность приведенной реализации $H_{\theta\downarrow\eta}$ антиунификации этих подстановок оценивается величиной $\Omega(|H_\theta||H_\eta|)$ (см. [22]).

В статье [24] был предложен алгоритм проверки логико-термальной эквивалентности программ. Этот алгоритм представляет собой итеративную процедуру, в которой на каждом шаге для тройки подстановок $\theta_1, \theta_2, \theta_3$ вычисляется значение $(\theta_1\theta_2)\downarrow\theta_3$. Таким образом, при любом из двух рассмотренных способе графовой реализации подстановок размер АОГ, представляющего подстановку $(\theta_1\theta_2)\downarrow\theta_3$, оценивается величиной, пропорциональной произведению размеров АОГ, представляющих подстановки-операнды. Поскольку в худшем случае число итераций, которые должен выполнить процедура проверки эквивалентности, пропорционально размеру анализируемых программ, алгоритм, предложенный в работе [25] имеет экспоненциальную сложность. Один из способов преодоления возникшей трудности состоит в ограничении класса рассматриваемых подстановок и замене операции антиунификации более простой операцией, которая вычисляет некоторую нижнюю грань двух подстановок, но при этом сохраняет справедливость всех вспомогательных утверждений, представленных в статье [24] и обеспечивающих корректность процедуры проверки логико-термальной эквивалентности программ.

5. Редуцированные подстановки

Далее мы будем рассматривать множество подстановок $Subst(X,F,Y)$; переменные из конечной области определения $X=\{x_1,x_2,\ldots,x_n\}$ этих подстановок будем называть *главными переменными*, а переменные из бесконечного множества Y, используемые для построения термов, – *вспомогательными переменными*. Для каждой подстановки θ множество переменных Var_θ может быть разбито на два подмножества

$HVar_\theta = \{y : \exists x\,(x \in X \wedge \theta(x) = y)\}$ и $NVar_\theta = Var_\theta \setminus HVar_\theta$. В графовой реализации подстановки θ все вершины, помеченные вспомогательными переменными из множества $HVar_\theta$, имеют заголовки. В множестве $Subst(X,F,Y)$ выделим подкласс редуцированных подстановок.

Определение 1. Подстановка θ из множества $Subst(X,F,Y)$ называется *редуцированной подстановкой*, если $Var_\theta = HVar_\theta$.

Исследуем некоторые характеристические свойства редуцированных подстановок.

Как видно из определения 1, характеристическое свойство редуцированных подстановок состоит в том, что в их приведенном АОГ, реализующем редуцированную подстановку, все листовые вершины, которым приписаны вспомогательные переменные, озаглавлены. Очевидно также, что каждая подстановка θ имеет хотя бы один редуцированный прототип (таковым является, например, пустая подстановка $\{x_1 / y_1, x_2 / y_2, \ldots, x_n / y_n\}$). Пусть задана некоторая подстановка $\theta = \{x_1 / t_1, x_2 / t_2, \ldots, x_n / t_n\}$ и пусть y - некоторая вспомогательная переменная, не принадлежащая множеству Var_θ. Предположим, что терм t_1 содержит подтерм $f(y_1, y_2, \ldots, y_k)$ и при этом $y_j \notin HVar_\theta$ для некоторого $j, 1 \le j \le k$. Тогда для каждого $i, 1 \le i \le k$, обозначим записью t_i' терм, который либо совпадает с термом t_i в том случае, когда t_i не содержит подтерма $f(y_1, y_2, \ldots, y_k)$, либо получен из терма t_i заменой всех вхождений подтерма $f(y_1, y_2, \ldots, y_k)$ на переменную y в том случае, когда указанный подтерм содержится в терме t_i. Рассмотрим новую подстановку $\theta' = \{x_1 / t_1', x_2 / t_2', \ldots, x_n / t_n'\}$. Как видно из приведенного выше описания подстановки θ', имеет место равенство $\theta = \theta'\{y / f(y_1, y_2, \ldots, y_k)\}$. Покажем, что для пары подстановок θ и θ' и произвольной редуцированной подстановки η справедливо

Утверждение 3. Редуцированная подстановка η является прототипом подстановки θ тогда и только тогда, когда η является прототипом подстановки θ'.

Доказательство. Поскольку $\theta = \theta'\{y / f(y_1, y_2, \ldots, y_k)\}$, каждый прототип подстановки θ' является прототипом подстановки θ. Рассмотрим теперь произвольный редуцированный прототип η подстановки θ. Тогда для некоторой подстановки $\rho, \rho \in Subst(Y,F,Y)$, верны равенства $\eta\rho = \theta = \theta'\{y / f(y_1, y_2, \ldots, y_k)\}$. Без ограничения общности будем считать, что подстановка ρ не содержит связок-переименований вида $\{y' / y''\}$. Тогда терм

$f(y_1, y_2, \ldots, y_k)$ должен входить в состав либо одного из термов в подстановке η, либо в состав одного из термов подстановки ρ. Поскольку подстановка η является редуцированной, а переменная y_j не содержится в множестве $HVar_\theta$, терм $f(y_1, y_2, \ldots, y_k)$ не содержится ни в одном из термов подстановки η. Значит, выражение $f(y_1, y_2, \ldots, y_k)$ входит только в состав термов подстановки ρ. В этом случае подстановку ρ можно представить в виде композиции $\rho = \rho'\{y / f(y_1, y_2, \ldots, y_k)\}$, где ρ' - это подстановка из класса $Subst(Y, F, Y)$, ни один терм которой не содержит вхождений выражения $f(y_1, y_2, \ldots, y_k)$. Ввиду того, что это выражение также не входит в состав ни одного из подстановки θ', приходим к заключению о выполнимости равенства $\eta\rho' = \theta'$. Оно свидетельствует о том, что редуцированная подстановка η является прототипом подстановки θ'. □

Определение 2. Редуцированная подстановка θ' называется *наиболее специальной редукцией* подстановки θ (обозначается $msr(\theta)$), если θ' является прототипом θ, и каждая редуцированная подстановка, являющаяся прототипом подстановки θ, является также примером подстановки θ'.

Непосредственно из утверждения 3 вытекает, что для каждой подстановки θ существует ее наиболее специальная редукция $msr(\theta)$, и эта редукция единственна (с точностью до переименования переменных). Опишем алгоритм построения $msr(\theta)$. Этот алгоритм работает с АОГ G_θ, реализующим подстановку θ, проводя его раскраску.

Все вершины графа G_θ, имеющие заголовок, окрашиваются в синий цвет, а все вершины, которым приписана вспомогательные переменные из множества $NVar_\theta$, окрашиваются в красный цвет. Далее строим раскраску графа, руководствуясь следующими правилами:

1. каждая красная вершина окрашивает в красный цвет все входящие в нее дуги;
2. если из вершины исходит хотя бы одна красная дуга, то и все исходящие из этой вершины дуги окрашиваются в красный цвет;
3. если вершина еще не окрашена, а все исходящие из нее дуги – красные, то эта вершина окрашивается в красный цвет;
4. все перечисленные выше действия повторяются до тех пор, пока возможно применение хотя бы одного из них.

После завершения раскраски графа

- для каждой вершины, у которой имеется заголовок и из которой исходят только красные дуги, вводится новая вспомогательная переменная, которой помечается эта вершина;
- из графа удаляются все вершины и дуги окрашенные в красный цвет.

В результате этого построения получаем размеченный граф, который является реализацией подстановки $msr(\theta)$. Время работы предложенного алгоритма пропорционально размеру графа G_θ.

Выделим несколько важных свойств, которыми обладает операция редукции подстановок.

Утверждение 4. Для любых двух подстановок θ_1 и θ_2 таких, что $\theta_1 \le \theta_2$ верно, что их наиболее специальные редукции находятся в соотношении $msr(\theta_1) \le msr(\theta_2)$.

Доказательство. Заметим, что из выполнимости отношения $\theta_1 \le \theta_2$ следует, что существует такая подстановка ρ, $\rho \in Subst(Y, F, Y)$, что выполнено равенство $\theta_2 = \theta_1\rho$. Тогда для обоснования справедливости леммы достаточно показать справедливость неравенства $msr(\theta_1) \le msr(\theta_1\rho)$. Справедливость этого отношения следует непосредственно из описания алгоритма построения редуцированных подстановок, описанного выше. □

Свойства операции редукции относительно композиции подстановок устанавливает следующее

Утверждение 5. Пусть $\theta \in Subst(X, F, X)$, $\eta \in Subst(X, F, Y)$. Тогда $msr(\theta\eta) = msr(\theta\, msr(\eta))$.

Доказательство. Справедливость утверждения вытекает из утверждения 1 и описания алгоритма построения наиболее специальных редукций подстановок. □

Теорема 6. Для любой пары атомарных формул A, B из множества $Atom(P, F, X)$ и для любой редуцированной подстановки θ из множества $Subst(X, F, Y)$ выполняется соотношение

$A\theta = B\theta \quad \Leftrightarrow \quad Amsr(\theta) = Bmsr(\theta)$.

Доказательство. Поскольку подстановка $msr(\theta)$ является прототипом подстановки θ, равенство $Amsr(\theta) = Bmsr(\theta)$ очевидно влечет за собой равенство $A\theta = B\theta$. С другой стороны, равенство $A\theta = B\theta$ означает, что подстановка θ является унификатором атомов A и B. Коль скоро эти атомы унифицируемы, они имеют наиболее общий унификатор. Так как атомы A и

B принадлежат классу $Atom(P,F,X)$, любой из алгоритмов унификации, описанных в работах [18-24], построит для этих атомов наиболее общий унификатор η, принадлежащий классу подстановок $Subst(X,F,X)$ и удовлетворяющий соотношению $Dom_\eta \cap Var_\eta = \varnothing$, т.е. $\eta = \{x_{i_1} / t_1, x_{i_2} / t_2, \ldots, x_{i_k} / t_k\}$ и при этом ни один терм $t_1, t_2, \ldots, t_k$ не содержит переменные из множества $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$. Пусть $Var_\theta = \{x_{j_1}, x_{j_2}, \ldots, x_{j_m}\}$; рассмотрим переименование $\lambda = \{x_{j_1} / y_1, x_{j_2} / y_2, \ldots, x_{j_m} / y_m\}$. Как известно (см. [19]), всякая подстановка, эквивалентная наиболее общему унификатору двух атомов, также является наиболее общим унификатором той же пары атомов. Поэтому подстановка $\theta' = \eta\lambda = \{x_{i_1} / t_1\lambda, x_{i_2} / t_2\lambda, \ldots, x_{i_k} / t_k, x_{j_1} / y_1, x_{j_2} / y_2, \ldots, x_{j_m} / y_m\}$ является

1. редуцированной подстановкой из класса $Subst(X,F,Y)$,
2. наиболее общим унификатором атомов A и B.

Последнее означает, что $A\theta' = B\theta'$ и для некоторой подстановки $\rho,\ \rho \in Subst(Y,F,Y)$, верно равенство $\theta = \theta'\rho$, и, следовательно, подстановка θ' является редуцированным прототипом подстановки. Но в таком случае для некоторой подстановки $\rho',\ \rho' \in Subst(Y,F,Y)$, выполняется равенство $msr(\theta) = \theta'\rho'$, и поэтому справедливо равенство $Amsr(\theta) = Bmsr(\theta)$. □

6. Редуцированная антиунификация подстановок

Определение 3. Редуцированной антиунификацией двух подстановок θ_1 и θ_2 из класса $Subst(X,F,Y)$ назовем такую подстановку θ_0 которая удовлетворяет равенству $\theta_0 = msr(\theta_1 \downarrow \theta_2)$.

Для обозначения редуцированной антиунификации двух подстановок будем использовать запись $\theta_0 = \theta_1 \Downarrow \theta_2$. Операция редуцированной антиунификации обладает определенным набором свойств, которые позволяют использовать ее для проверки логико-термальной эквивалентности.

Теорема 7. Для любой пары подстановок $\eta_1, \eta_2 \in Subst(X,F,Y)$ справедливо равенство

$\eta_1 \Downarrow \eta_2 \approx msr(\eta_1) \Downarrow msr(\eta_2)$.

Доказательство. Заметим, что $msr(\eta_1) \le \eta_1$ и $msr(\eta_2) \le \eta_2$. Поэтому $msr(\eta_1) \downarrow msr(\eta_2) \le \eta_1 \downarrow \eta_2$. На основании утверждения 4 отсюда следует неравенство $msr(\eta_1) \Downarrow msr(\eta_2) \le \eta_1 \Downarrow \eta_2$.

Покажем теперь, что каждый редуцированный прототип подстановки $\eta_1 \downarrow \eta_2$ является также редуцированным прототипом подстановки $msr(\eta_1) \downarrow msr(\eta_2)$. Рассмотрим произвольную редуцированную подстановку θ из класса $Subst(X,F,Y)$, являющуюся прототипом подстановки $\eta_1 \downarrow \eta_2$. Тогда θ также является редуцированным прототипом обеих подстановок η_1 и η_2. Но в этом случае верны соотношения $\theta \le msr(\eta_1)$ и $\theta \le msr(\eta_2)$, и, следовательно, соотношение $\theta \le msr(\eta_1) \downarrow msr(\eta_2)$.

Таким образом, наиболее специальный редуцированный прототип подстановки $\eta_1 \downarrow \eta_2$ является некоторым редуцированным прототипом подстановки $msr(\eta_1) \downarrow msr(\eta_2)$. Отсюда следует неравенство $\eta_1 \Downarrow \eta_2 \le msr(\eta_1) \Downarrow msr(\eta_2)$.

Неравенства $msr(\eta_1) \Downarrow msr(\eta_2) \le \eta_1 \Downarrow \eta_2$ и $\eta_1 \Downarrow \eta_2 \le msr(\eta_1) \Downarrow msr(\eta_2)$ означают, что подстановки $\eta_1 \Downarrow \eta_2$ и $msr(\eta_1) \Downarrow msr(\eta_2)$ эквивалентны. □

Теорема 8. Для любой подстановки $\theta \in Subst(X,F,X)$ и пары подстановок $\eta_1, \eta_2 \in Subst(X,F,Y)$ справедливо равенство

$\theta\eta_1 \Downarrow \theta\eta_2 = msr(\theta(\eta_1 \Downarrow \eta_2))$.

Доказательство. Согласно определению редуцированной антиунификации и теореме 1 (см. [24]) верны равенства

$\theta\eta_1 \Downarrow \theta\eta_2 = msr(\theta\eta_1 \downarrow \theta\eta_2) = msr(\theta(\eta_1 \downarrow \eta_2))$.

Как следует из утверждения 5,

$msr(\theta(\eta_1 \downarrow \eta_2)) = msr(\theta\, msr(\eta_1 \downarrow \eta_2)) = msr(\theta(\eta_1 \Downarrow \eta_2))$. □

Теорема 9. Для любой пары атомов $A, B \in Atom(P,F,X)$ и любой пары подстановок $\eta_1, \eta_2 \in Subst(X,F,Y)$ справедливо соотношение

$A\eta_1 = B\eta_1 \wedge A\eta_2 = B\eta_2 \quad \Leftrightarrow \quad A(\eta_1 \Downarrow \eta_2) = B(\eta_1 \Downarrow \eta_2)$.

Доказательство. Как гласит теорема 2 (см. [24]),

$A\eta_1 = B\eta_1 \wedge A\eta_2 = B\eta_2 \quad \Leftrightarrow \quad A(\eta_1 \downarrow \eta_2) = B(\eta_1 \downarrow \eta_2)$.

Согласно утверждению 6

$A(\eta_1 \downarrow \eta_2) = B(\eta_1 \downarrow \eta_2) \quad \Leftrightarrow \quad A(\eta_1 \Downarrow \eta_2) = B(\eta_1 \Downarrow \eta_2)$. □

Обратимся теперь к алгоритму проверки л-т эквивалентности программ, описанному в разделе 3 и обоснованному в статье [24]. Заменим в процедуре разметки графа логически совместных трасс $\Gamma[\pi', \pi'']$ операцию антиунификации $\downarrow$ на операцию редуцированной антиунификации $\Downarrow$. Тогда

доказанные выше теоремы 7, 8 и 9 обеспечивают корректность и полноту модифицированного таким образом алгоритма.

7. Сложность вычисления редуцированной антиунификации подстановок

Покажем, что переход от операции унификации $\downarrow$ к операции редуцированной антиунификации $\Downarrow$ позволяет существенно повысить эффективность алгоритма проверки л-т эквивалентности программ. Прежде всего, оценим сложность построения и размер графовой реализации редуцированной антиунификации двух подстановок $\eta_1 \Downarrow \eta_2$. Для вычисления редуцированной антиунификации можно воспользоваться композицией двух алгоритмов: алгоритма вычисления антиунификации двух подстановок, описанного в статьях [13, 23], и алгоритма построения наиболее специальной редукции, описанного в разделе 3 настоящей работы. Однако оценка сложности вычисления и размера построенной таким образом подстановки $\theta = \eta_1 \Downarrow \eta_2$ может иметь квадратичную зависимость от размера графовой реализации подстановок η_1, η_2. Такие оценки неприемлемы для нашей цели – построения полиномиального по времени алгоритма проверки л-т эквивалентности программ. Поэтому ниже описан алгоритм редуцированной антиунификации подстановок, имеющий линейную сложность по времени.

Пусть заданы приведенные размеченные АОГ G_{η_1} и G_{η_2}, реализующие пару подстановок η_1, η_2. Построение приведенного размеченного АОГ G_θ, реализующего подстановку $\theta = \eta_1 \Downarrow \eta_2$, проведем в три этапа. Вершинами графа G_θ будут упорядоченные пары (u, v), где u - вершина АОГ G_{η_1}, а v - вершина графа $G_{\eta 2}$.

На первом этапе построения введем множество вершин графа G_θ, имеющих заголовки. Чтобы осуществить это, для каждой переменной $x_i, x_i \in X$, рассмотрим пару вершин $w_i = (u_i, v_i)$, где u_i - вершина АОГ G_{η_1}, озаглавленная переменной x_i, а v_i - вершина АОГ G_{η_2}, озаглавленная той же переменной x_i. Все указанные вершины w_i входят в состав графа G_θ; при этом каждая такая вершина w_i считается озаглавленной переменной x_i. Возможно, что некоторые из указанных вершин могут быть озаглавлены несколькими различными переменными из множества X.

На втором этапе множество вершин графа G_θ пополняется новыми вершинами при помощи следующей процедуры. Предположим, что u_0 и v_0 - это вершины в АОГ G_{η_1} и G_{η_2} соответственно, помеченные одним и тем же

функциональным символом f, $f \in F$, местности m. Предположим также, что для любого натурального числа $i, 1 \le i \le m$, пара (u_i', v_i'), где u_i' - i-наследник вершины u_0 в АОГ $G_{\eta 1}$, а v_i' - i-наследник вершины v_0 в АОГ $G_{\eta 2}$, уже внесена в состав вершин графа G_θ. Тогда к множеству вершин графа G_θ добавляется вершина (u_0, v_0), которая помечается функциональным символом f. При этом для каждого $i, 1 \le i \le m$, из вершины (u_0, v_0) в вершину (u_i', v_i') проводится дуга, помеченная числом i. В том случае, если вершина (u_0, v_0) была введена ранее на первом этапе построения графа G_θ, осуществляется только проведение дуг из этой вершины. Применение описанной процедуры прекращается, как только к графу G_θ уже не удается добавить ни одной новой вершины и ни одной новой дуги.

На третьем этапе все листовые вершины, озаглавленные переменными из множества X (к их числу относятся только некоторые из вершин $w_i = (u_i, v_i)$, введенных на первом этапе построения графа G_θ) помечаются попарно различными вспомогательными переменными из множества Y, и после этого применяется процедура редукции, описанная в разделе 4.

Из описания процедуры построения размеченного АОГ G_θ видно, что реализуемая этим графом подстановка θ обладает следующими свойствами:

1. подстановка θ является редуцированной (это следует из замечания к определению 1 редуцированной подстановки);
2. подстановка θ является прототипом каждой из подстановок η_1, η_2;
3. каждый редуцированный прототип обеих подстановок η_1, η_2 является также и прототипом подстановки θ.

Таким образом, построенный АОГ G_θ является приведенной графовой реализацией редуцированной антиунификации $\eta_1 \Downarrow \eta_2$.

На первом этапе построения АОГ G_θ алгоритм вычисления редуцированной антиунификации совершает $|X|$ действий. Число итераций, совершаемых процедурой пополнения множества вершин графа G_θ на втором этапе его построения, не превосходит числа вершин в каждом из АОГ $G_{\eta 1}$ и $G_{\eta 2}$. Поэтому время построения АОГ G_θ можно оценить сверху величиной

$$O(|X| + \min(|G_{\eta 1}|, |G_{\eta 2}|)).$$

Чтобы оценить размер графа, реализующего результат редуцированной антиунификации двух подстановок, введем в рассмотрение еще одну сложностную характеристику АОГ для редуцированных подстановок –

сложность редуцированной подстановки. Пусть η – произвольная редуцированная подстановка из класса $Subst(X,F,Y)$, и G_η – приведенный АОГ, реализующий подстановку η. Тогда *сложностью* подстановки η назовем величину $\ell(\eta) = N_1(\eta) + |X| - N_2(\eta)$, где $N_1(\eta)$ – количество вершин в графе G_η, помеченных функциональными символами, а $N_2(\eta)$ – количество вершин в графе G_η, озаглавленных переменными из множества X.

Утверждение 10. Пусть $\theta = \eta_1 \Downarrow \eta_2$. Тогда

1. $\ell(\theta) \le \min(\ell(\eta_1), \ell(\eta_2))$;
2. Если $\ell(\theta) = \ell(\eta_1)$, то $\theta = \eta_1$.

Доказательство. Вершины АОГ G_θ, озаглавленные переменными из множества X, строятся на первом этапе. Из описания алгоритма видно, что $N_2(\theta) \ge max(N_2(\eta_1), N_2(\eta_2))$.

Каждая вершина, которая добавляется в АОГ G_θ на втором этапе вычисления редуцированной антиунификации, взаимно однозначно соответствует одной вершине графа G_{η_1} и одной вершине графа G_{η_2}. Таким образом, верно неравенство

$N_1(\theta) \le \min(N_1(\eta_1), N_1(\eta_2))$.

Из полученных неравенств следует, что $\ell(\theta) \le \min(\ell(\eta_1), \ell(\eta_2))$. Кроме того, равенство $\ell(\theta) = \ell(\eta_1)$ означает, что графы G_θ и G_{η_1} изоморфны, и поэтому $\theta = \eta_1$. □

Оценки трудоемкости вычисления редуцированной антиунификации двух подстановок и размеров ее графовой реализации, установленные выше, позволяют оценить трудоемкость решения задачи проверки л-т эквивалентности программ.

Теорема 11. Существует алгоритм, разрешающий задачу проверки л-т эквивалентности произвольной пары программ π_1 и π_2 за время $O(n^6)$, где $n = \max(|\pi_1|, |\pi_2|)$.

Доказательство. Воспользуемся модифицированным алгоритмом проверки л-т эквивалентности программ, описанным в разделе 3 и уточненным в разделе 6. Очевидно, что граф логически совместных трасс $\Gamma[\pi_1, \pi_2]$ имеет не более n^2. Максимальная сложность подстановки $\eta_{u'u''}$, приписанной любой вершине (u', u'') этого графа, также не превосходит величины n^2. Как

следует из утверждения 10, на каждой итерации алгоритма проверки л-т эквивалентности программ сложность хотя бы одной из подстановок, приписанных вершине графа логически совместных трасс $\Gamma[\pi_1, \pi_2]$, уменьшается. Поэтому число итераций алгоритма ограничено величиной $O(n^4)$. При этом трудоемкость вычисления редуцированной антиунификации $\mu = \theta\eta_{u'u''} \Downarrow \eta_{v'v''}$ на каждой итерации алгоритма проверки л-т эквивалентности программ ограничено величиной, которая пропорциональна максимальной сложности подстановок, приписанных вершинам графа. Отсюда следует оценка сложности алгоритма, заявленная в теореме. □

Список литературы

[1] Иткин В.Э. Логико-термальная эквивалентность схем программ // Кибернетика. — 1972. — N 1. — с. 5-27.

[2] Сабельфельд В.К. Полиномиальная оценка сложности распознавания логико-термальной эквивалентности // ДАН СССР. — 1979. — т. 249, N 4. — с. 793-796.

[3] Sabelfeld V.K. The logic-termal equivalence is polynomial-time decidable // Information Processing Letters. — 1980 — v. 10, N 2. — p. 57-62.

[4] Буда А.О., Иткин В.Э. Сводимость эквивалентности программ к термальной эквивалентности // Системное и теоретическое программирование. Сборник статей. Т. 1. — Кишенев, 1974. — с. 293-324.

[5] Буда А.О. Об одной модели вычислений над памятью с разрешимой проблемой эквивалентности // Доклады АН СССР — 1979. — т. 245, N 1.

[6] Буда А.О. Канонизация логической структуры логико-термально эквивалентных схем программ // Трансляция и модели программ. — Новосибирск, 1980.

[7] Годлевский А.Б. О связи логико-термальной эквивалентности схем программ с одним случаем специальной проблемы эквивалентности дискретных преобразователей // Кибернетика. — 1975. — N 5. — с. 32-34.

[8] Хачатрян В.Е. Логико-термальная эквивалентность в классе схем над сильно невырожденным базисом // Программирование. — 1977. — N 3. — с. 20-27.

[9] Сабельфельд В.К. Некоторые оптимизирующие преобразования для стандартных схем программ // Языки и системы программирования. — Новосибирск, 1979.

[10] Син Мен Де. Логико-термально эквивалентные преобразования схем программ // Кибернетика. — 1976 — N 5.

[11] Фаулер М. Рефакторинг. Улучшение существующего кода. — Символ-Плюс, 2008. — 432 с.

[12] Komondoor R., Horwitz S. Using slicing to identify duplication in source code // Proceedings of the 8th International Symposium on Static Analysis. — Springer-Verlag, 2001. — p. 40–56.

[13] Bulychev P., Kostylev E., Zakharov V. Anti-unification algorithms and their applications in program analysis // Proceedings of the 7th International Conference “Perspectives of System Informatics”, June 15-19, 2009, Novosibirsk. — 2009. — p. 82-89.

[14] Luckham D.C., Park D.M., Paterson M.S., On formalized computer programs // Journal of Computer and System Science — 1970. — v.4, N 3. — p. 220-249.

[15] Itkin V.E., Zwinogrodski Z. On program schemata equivalence // Journal of Computer and System Science. — 1972. — v. 6, N 1. — p. 88-101.
[16] Котов В.Е., Сабельфельд В.К. Теория схем программ. — М.:Наука, 1991. — 348 с.
[17] Eder E. Properties of substitutions and unifications // Journal of Symbolic Computations. — v. 1. — 1985. — p. 31-46.
[18] Palamidessi C. Algebraic properties of idempotent substitutions // Lecture Notes in Computer Science — v. 443 — 1990. — p. 386-399.
[19] Plotkin G.D. A note on inductive generalization // Machine Intelligence. — 1970. — v. 5, N 1. — p. 153-163.
[20] Sorensen M. H., Gluck. R. An algorithm of generalization in positive supercompilation // Proceedings of the 1995 International Symposium on Logic Programming._MIT Press. — 1995. — p. 465-479.
[21] Захаров В.А., Костылев Е.В. О сложности задачи антиунификации // Дискретная математика. — 2008. — т. 20, N 1. — с. 131-144.
[22] Zakharov V.A. On the refinement of logic programs by means of anti-unification // Proceedings of the 2nd Panhellenic Logic Symposium, Delphi, Greece. — 1999. — p. 219-224.
[23] В.А. Захаров, Т.А. Новикова. Применение алгебры подстановок для унификации программ. Труды Института системного программирования РАН, том 21, 2011 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

Polynomial time algorithm for checking strong equivalence of program

T.A. Novikova (Kazakhstan Branch of Lomonosov Moscow State University, Astana, Kazakhstan)
V.A.Zakharov (ISP RAS, Moscow, Russia),
{zakh@cs.msu.su} {taniaelf@mail.ru}

Annotation. Strong (logic&term) equivalence of programs is the weakest decidable equivalence relation which approximates the functional equivalence of programs. In this paper we develop a new variant of the algorithm for checking strong equivalence of programs. A distinguished feature of our algorithm is that it relies completely on the algebra of finite substitutions which includes the operations of composition and antiunification. The paper begins with a short introduction to the theory of first-order substitutions. In the next section the formal first-order model of sequential programs and strong equivalence are defined. We also describe a preliminary variant of equivalence checking algorithm. This algorithm deals with composition and antiunification operations on finite substitutions. Since the complexity of these operations depends on data structures that represent substitutions, we consider in Section 4 graph-theoretic representations of substitutions as well as upper and lower bounds on the complexity of basic operations on substitutions. To reduce the complexity of equivalence checking algorithm for sequential programs we introduce in Section 5 a new class of truncated substitutions and study the basic algebraic properties of composition and antiunification operations on truncated substitutions. In Section 6 we showed that both operations on truncated substitutions can be performed in time $O(n^2)$. Finally, in Section 7 using the properties of truncated substitutions we introduced an improved version of the equivalence checking algorithm and proved that its time complexity is $O(n^6)$ on the size of programs to be checked.

Keywords: program, strong equivalence, equivalence checking, substitution, composition, antiunification, complexity.

References

[1]. Itkin V.E. Logical-termal equivalence of program schemata. Proceedings of the International Sympoisum on Theoretical Programming, 1972, p. 127-143.

[2]. Sabel'fel'd V.K. Polinomial'naja ocenka slozhnosti raspoznavanija logiko-termal'noj jekvivalentnosti [Polynomial time algorithm for checking strong equivalence of program] DAN SSSR [Reports of USSR Academy of Science], 1979, t. 249, N 4, s. 793-796 [in Russian].

[3]. Sabelfeld V.K. The logic-termal equivalence is polynomial-time decidable. Information Processing Letters, 1980, v. 10, N 2, p. 57-62.

[4]. Buda A.O., Itkin V.E. Svodimost' ekvivalentnosti program k termal'noy ekvivalentnosti [On the reducibility of program equivalence to termal equivalence]. Sistemnoye I teoreticheskoye programirovaniye [Proceeding of the conference on System and Theoretic Programming], vol. 1, Cisinau, 1974, p. 293-324 [in Russian].

[5]. Buda A.O. Ob odnoy modeli vychisleniy nad pamat'yu s razreshimoy problemoy ekvivalentnosti [On a random access memory model of computation with decidable equivalence checking problem]. Doklady Akademii Nauk [Reports of USSR Academy of Science], 1979, vol. 245, no 1 [in Russian].
[6]. Buda A.O. Kanonizaciaya logicheskoy struktuty logiko-termal'no ekvivalentnyh shem program [On canonization of logical structure of logic-termal equivalent program schemata]. Translaciya i Modeli program [Translation and program models], Novosibirsk, 1980 [in Russian].
[7]. Godlevsky A.B. O svyazi logiko-termal'noy ekvivalentnosti shem program s odnim sluchaem special'noy problem ekvivalentnosti diskrenyh preobrazovateley [On the relationship of logic-termal equivalence of program schemata with some special case of equivalence problem for discrete transducers]. Cybernetics, 1975, N 5, p. 32-34 [in Russian].
[8]. Khachatryan V.Y. Logiko-termal'naya ekvivalentost' v klasse shem nad sil'no nevyrizhdennym bazisom [Logic-termal equivalence in a class of schemata over strongly nondegenerated basis]. Programirovaniye [Programming and Computer Software], 1977, N 3, p. 20-27 [in Russian].
[9]. Sabelfeld V.K. Nekotoriye optomiziruyushchiye preobrazovaniya dly standartnyh shem program [Some optimization transformations for standard program schemata]. Yazyki i sistemy programirovaniya [Languages and programming systems]. Novosibirsk, 1979 [in Russian].
[10]. Sin Min De. Logiko-termal'niye probrazovaniya shem program [Logic-termal transformations of program schemata]. Kibernetika [Cybernetics], 1976, N 5 [in Russian].
[11]. Fauler M. Refactoring: Improving the design of existing code. 1999, Addison Wesley.
[12]. Komondoor R., Horwitz S. Using slicing to identify duplication in source code. Proceedings of the 8th International Symposium on Static Analysis. Springer-Verlag, 2001, p. 40–56.
[13]. Bulychev P., Kostylev E., Zakharov V. Anti-unification algorithms and their applications in program analysis // Proceedings of the 7th International Conference "Perspectives of System Informatics", June 15-19, 2009, Novosibirsk, 2009, p. 82-89.
[14]. Luckham D.C., Park D.M., Paterson M.S., On formalized computer programs. Journal of Computer and System Science, 1970, v.4, N 3, p. 220-249.
[15]. Itkin V.E., Zwinogrodski Z. On program schemata equivalence. Journal of Computer and System Science, 1972, v. 6, N 1, p. 88-101.
[16]. Kotov V.E. Sabelfeld V.K. Teoriya skhem program [Theory of program schemata]. Moscow, "Nauka", 1991, 348 p.
[17]. Eder E. Properties of substitutions and unifications. Journal of Symbolic Computations, v. 1, 1985, p. 31-46.
[18]. Palamidessi C. Algebraic properties of idempotent substitutions. Lecture Notes in Computer Science, v. 443, 1990, p. 386-399.
[19]. Plotkin G.D. A note on inductive generalization. Machine Intelligence, 1970, v. 5, N 1, p. 153-163.
[20]. Sorensen M. H., Gluck. R. An algorithm of generalization in positive supercompilation. Proceedings of the 1995 International Symposium on Logic Programming._MIT Press, 1995, p. 465-479.
[21]. Zakharov V.A., Kostylev E.V. On the complexity of anti-unification. Discrete Mathematics, 2008, vol. 20, N 1, p. 131-144.

[22]. Zakharov V.A. On the refinement of logic programs by means of anti-unification. Proceedings of the 2nd Panhellenic Logic Symposium, Delphi, Greece, 1999, p. 219-224.
[23]. Zakharov V.A., Novikova T.A. Primininie algebry podstanovok dlya unifikacii program [On the application of substitution algebra to program unification]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, p. 141-166 [in Russian].

Об одной задаче Коффмана-Шора, связанной с упаковкой прямоугольников в полосу

М.А. Трушников

Аннотация. Предложен новый онлайновый алгоритм упаковки прямоугольников в полосу, существенно превосходящий известный алгоритм Коффмана-Шора по качеству получаемой упаковки.

1. Введение

Задача упаковки в полосу (в англоязычной литературе Strip packing problem) состоит в размещении множества открытых прямоугольников внутри полубесконечной вертикальной полосы единичной ширины, при этом стороны прямоугольников должны быть параллельны сторонам полосы (вращения запрещены) и прямоугольники не должны пересекаться. Нужно минимизировать «высоту упаковки» --- расстояние от основания полосы до верхней грани верхнего прямоугольника в упаковке.

Эта задача рассматривалась с 1980 г. [1,2]. Она близка к известной задаче упаковки в контейнеры (bin packing) [3,4,10].

Задача имеет следующую естественную интерпретацию. Каждый прямоугольник --- вычислительная задача, ширина прямоугольника соответствует количеству процессоров необходимых для вычисления задачи, высота --- времени. Эффективное размещение прямоугольников внутри полосы требуется также в задачах разработки СБИС и раскройки материалов. Частным случаем упаковки в полосу при равенстве высот всех прямоугольников является NP-трудная задача упаковки в контейнеры. Поэтому для общей задачи упаковки в полосу интерес представляют приближенные полиномиальные алгоритмы.

Особый интерес представляют «онлайновые» алгоритмы, размещающие прямоугольники в полосе по мере их поступления, без знания параметров всех последующих прямоугольников.

Алгоритмы упаковки в полосу обычно оцениваются по худшему случаю (worst-case analysis) или в среднем (average-case analysis). При анализе **по худшему случаю** минимизируют «высоту упаковки». При анализе **в среднем** минимизируемой функцией является математическое ожидание

незаполненной прямоугольниками площади полосы от основания полосы до верхней грани самого верхнего прямоугольника в упаковке. При этом ширины и высоты всех прямоугольников являются независимыми в совокупности равномерно распределенными на $(0,1]$ случайными величинами.

Известно, что математическое ожидание незаполненной площади полосы у оптимальной упаковки есть $O(N^{1/2})$, где N --- число прямоугольников. В 1993 году Коффман и Шор предложили «офлайновый» (информация о всех прямоугольниках известна заранее) алгоритм [6], для которого по порядку достигается нижняя оценка $O(N^{1/2})$. В той же работе они предложили онлайновый алгоритм, требующий знания числа прямоугольникой заранее, с оценкой $O(N^{2/3})$ для математического ожидания незаполненной площади полосы. Этот алгоритм относится к классу так называемых шельфовых алгоритмов, активно изучавшихся ранее [2,5,7].

В 2010 году Кузюрин и Поспелов предложили онлайновый алгоритм с той же оценкой $O(N^{2/3})$ [9], но не требующий знания числа прямоугольников заранее. Тем не менее вопрос о возможности улучшения оценки $O(N^{2/3})$ в классе онлайновых алгоритмов оставался открытым с 1993 г. Ранее были получены результаты о том, что в классе шельфовых алгоритмов (в котором была получена оценка Шором и Кофманом) существенно улучшить оценку качества упаковки невозможно [7].

В данной работе предложен принципиально новый онлайновый алгоритм упаковки прямоугольников. Проведенные эксперименты показали, что предложенный алгоритм обеспечивает верхнюю оценку $O(N^{1/2})$ для математического ожидания незаполненной площади полосы, что существенно лучше, чем у алгоритма Коффмана-Шора.

2. Постановка задачи

Вход: N --- число прямоугольников; $w_i, h_i, i = 1,\ldots,N$ --- ширины и высоты прямоугольников, являющиеся значениями независимых в совокупности равномерно распределенных на $(0,1]$ случайных величин;

Выход: $x_i, y_i, i = 1,\ldots,N$ -- координаты центров прямоугольников, удовлетворяющие условию: прямоугольники без вращений и пересечений размещены внутри полубесконечной полосы единичной ширины. Основание полосы совпадает с отрезком $[(0,0),(1,0)]$ в R^2, а боковые стороны полосы параллельны оси y.

Высотой упаковки назовем величину

$$H = max_i \left(y_i + \frac{h_i}{2} \right).$$

Требуется минимизировать величину

$$H - \sum_{i=1}^{N} w_i * h_i,$$

равную незаполненной площади полосы.

Алгоритм

Предлагается алгоритм для описанной задачи со следующими свойствами. Алгоритму известно число прямоугольников N заранее, в остальном алгоритм является полностью онлайновым: прямоугольники подаются алгоритму по очереди, упаковка i-го прямоугольника зависит только от прямоугольников с номерами с 1-го по i-й. Более формально

$$\forall\, i = 1, \ldots, N$$
$$x_i = x_i(h_1, w_1, h_2, w_2, \ldots, h_i, w_i)$$
$$y_i = y_i(h_1, w_1, h_2, w_2, \ldots, h_i, w_i).$$

Где x_i, y_i --- координаты центров прямоугольников в упаковке, производимой данным алгоритмом.

Обозначим некоторые величины.

$$d = \left\lfloor \frac{N/4}{\sqrt{N}} \right\rfloor,\ U = \frac{N/4}{d} = \sqrt{N} + O(1), \delta = \frac{1}{d}$$

В основании полосы выделяется $d+1$ горизонтальная область, каждая высоты U в соответствии с рисунком.



Рис. 1.

Выделенную область в основании полосы можно разбить на две одинаковые (с точностью до поворота на 180 градусов) "пирамиды" вида:

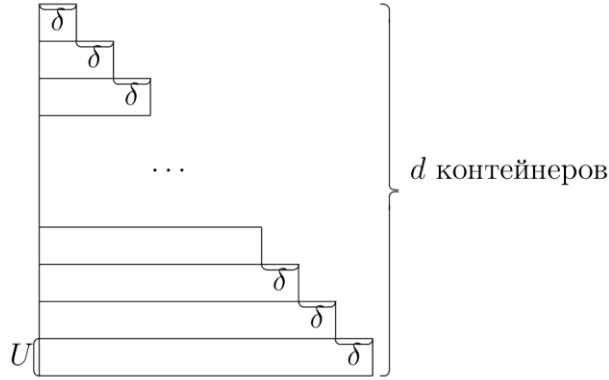


Рис. 2.

Каждый четный прямоугольник будем упаковывать в одну пирамиду, каждый нечетный --- в другую.

Далее следует описание упаковки прямоугольника в пирамиду. Прямоугольники, из которых состоит пирамида будем называть **контейнерами**. Пронумеруем контейнеры внутри пирамиды числами от 1 до

d так, что i-й имеет ширину $i\delta$. Прямоугольники внутри контейнеров будут упаковываться просто друг над другом: первый упаковываемый прямоугольник кладется на дно контейнера, следующий --- поверх первого и так далее.

Пусть некоторое количество прямоугольников упаковано в пирамиду и следующим для упаковки в данную пирамиду приходит прямоугольник ширины w.

• Найдем такое i, что $(i-1)\delta < w \le i\delta$. Будем говорить, что этот прямоугольник **назначен** в i-й контейнер (тем не менее он не обязательно будет упакован именно в i-й контейнер).

• Далее ищем минимальное такое j, что $i \le j \le d$ и в j-м контейнере достаточно места, чтобы поместить туда данный прямоугольник.

• Если такое j существует --- помещаем данный прямоугольник в j-й контейнер.

• Если нет --- просто кладем прямоугольник сверху текущей упаковки. Такие прямоугольники, которым не нашлось места ни в одном из контейнеров, в которые они помещаются по ширине будем называть **выпавшими**.

3. Анализ алгоритма

Корректность и онлайновость алгоритма очевидны. Чтобы показать, что алгоритм обеспечивает высокое качество упаковки достаточно показать, что число выпавших прямоугольников мало (порядка $N^{1/2}$).

Мы приведем оценки экспериментального анализа предложенного алгоритма и оценки незаполненной площади полосы для него.

Для этого была реализована версия описанного алгоритма, эффективность которой исследовалась на случайных последовательностях прямоугольников. Результаты экспериментов приведены ниже. При этом через N обозначено число прямоугольников, величина незаполненной площади представлена в виде $CN^{1/2}$.

N	C
80000	1.5655
150000	1.5716
500000	1.5798
1000000	1.5798
4000000	1.5878
15000000	1.5975

30000000	1.5897
100000000	1.5934
300000000	1.6006
800000000	1.5912
1000000000	1.6044
1500000000	1.6027
2000000000	1.5949

Как видно из представленных экспериментальных результатов предложенный алгоритм существенно превосходит по качеству упаковки алгоритм Коффмана-Шора и обеспечивает оценку незаполненной площади порядка $O(\sqrt{N})$ с небольшой мультипликативной константой (меньшей двух). Получение аналогичных теоретических оценок для рассмотренного алгоритма представляется интересной исследовательской задачей.

Ссылки

[1] Baker B. S., Coffman E. J., Rivest R. L. Orthogonal packings in two dimensions. SIAM J. Computing. 1980. V. 9. 4. P. 846-855.

[2] Baker B. S., Schwartz J. S. Shelf algorithms for two dimensional packing problems. SIAM J. Computing. 1983. V. 6. 2. P. 508-525.

[3] Karp R. M., Luby M., Marchetti-Spaccamela A. A probabilistic analysis of multidimensional bin packing problems. Proc. Annu, ACM Symp. on Theory of Computing. New-York: ACM. 1984. P 289-298.

[4] Shor P. W. The average-case analysis of some on-line algorithms for bin packing. Combinatorica. 1986. V. 6. 2. P. 179-200.

[5] Csirik J., Woeginger G. J. Shelf algorithm for on-line strip packing, Inf. Process. Lett. 1997. V. 63. 4, P. 171-175.

[6] Coffman E. G., Jr, Shor P. W. Packing in two dimensions: Asymptotic average-case analysis of algorithms. Algorithmica. 1993. V. 9. 3. P. 253-277.

[7] Кузюрин Н. Н., Поспелов А. И. Вероятностный анализ шельфовых алгоритмов упаковки прямоугольников в полосу. Дискретная математика. 2006. Т. 18. 1. С. 76-90.

[8] X. Han, K. Iwama, D. Ye, G. Zhang. Strip Packing vs. Bin Packing. Algorithmic Aspects in Information and Management. Lecture Notes in Computer Science. 2009. V. 4508/2007. P. 358-367.

[9] Кузюрин Н. Н., Поспелов А. И. Вероятностный анализ нового класса алгоритмов упаковки прямоугольников в полосу. ЖВМиМФ. 2011. Т. 51, N 10, с. 1931-1936.

[10] Seiden S. S., On the online bin packing problem. J. ACM 49. 2002. P. 640-671.

On one problem of Koffman-Shor connected to strip packing problem

M.A. Trushnikov (ISP RAS, Moscow, Russia)

Abstract. In 1993 Coffman and Shor proposed on-line strip packing algorithm with expected unpacked area (waste area) of order O(n^{2/3}) in a standard probabilistic model, where n is the number of rectangles. In the standard probabilistic model the width and height of each rectangle are independent random variables with uniform distribution in [0,1]. It is well-known that optimal packing has expected waste of order O(n^{1/2}) and there exists off-line polynomial algorithm that provides this bound.. During more than 10 years the question concerning the possibility to obtain similar upper bound in the class of on-line strip packing algorithms was open. It was also known that in the class of popular so-called shelf algorithms the bound O(n^{2/3}) cannot be improved. In this paper we propose new strip packing algorithm which differs from all known on-line strip packing algorithms. We analyze our strip packing algorithm experimentally and show that the upper bound of expected unpacked area is of order O(n^{1/2}) which is optimal up to constant factor. Our experimnets show that this constant factor is close to 1.5-1.6. Our experiments were done with up to 2000000000 random rectangles. The only restriction is the following: we must know the number n of rectangles in advance. In a popular terminology concerning on-line algorithms this means that our algorithm is closed-end on-line algorithm.

The paper proposes a new online strip packing algorithm that provides a quality of packing significantly higher than well-known algorithm of Koffman-Shor.

Keywords: online strip packing algorithm, algorithm of Koffman-Shor

References

[1]. Baker B. S., Coffman E. J., Rivest R. L. Orthogonal packings in two dimensions. SIAMJ. Computing. 1980. V. 9. 4. P. 846-855.

[2]. Baker B. S., Schwartz J. S. Shelf algorithms for two dimensional packing problems.SIAM J. Computing. 1983. V. 6. 2. P. 508-525.

[3]. Karp R. M., Luby M., Marchetti-Spaccamela A. A probabilistic analysis ofmultidimensional bin packing problems. Proc. Annu, ACM Symp. on Theory of Computing. New-York: ACM. 1984. P 289-298.

[4]. Shor P. W. The average-case analysis of some on-line algorithms for bin packing.Combinatorica. 1986. V. 6. 2. P. 179-200.

[5]. Csirik J., Woeginger G. J. Shelf algorithm for on-line strip packing, Inf. Process. Lett. 1997. V. 63. 4, P. 171-175.

[6]. Coffman E. G., Jr, Shor P. W. Packing in two dimensions: Asymptotic average-case analysis of algorithms. Algorithmica. 1993. V. 9. 3. P. 253-277.

[7]. Kuzyurin N.N., Pospelov A. I.. Veroyatnostnyi analiz shelfovyh algoritmov upakovki pryamougolnikov v polosu [Probabilistic analysis of shelf strip packintg algorithms]. Discretnaya matematika [Discrete mathematics]. 2006. V. 18. N 1. P. 76-90. (in Russian)

[8]. Han X., Iwama K., Ye D., Zhang G. Strip Packing vs. Bin Packing. Algorithmic Aspects in Information and Management. Lecture Notes in Computer Science. 2009. V. 4508/2007. P. 358-367.
[9]. Kuzyurin N.N., Pospelov A. I. Veroyatrnostnyi analiz novogo klassa algoritmov upakovki pryamougolnikov v polosu [Probabilistic analysis of a new class of strip packing algorithms]. JVMiMF [Journal of computational mathematics and mathematicak physics]. 2011. V. 51, N 10, p. 1931-1936. (in Russian)
[10]. Seiden S. S., On the online bin packing problem. J. ACM 49. 2002. P. 640-671.

Сравнение сложностей задач нахождения базиса Гребнера идеала и решений этого идеала

А. В. Шокуров

Аннотация. Сравниваются сложности задач нахождения решения системы алгебраических уравнений и базисов Гребнера идеалов этих систем.

1. Введение

В диссертации Б. Бухбергера [Buc06] в 1965 году был предложен новый метод исследования идеалов в кольцах многочленов. В основании метода положено использование базиса идеала специального вида — базиса Гребнера, названного им так в честь своего научного руководителя. Предложенный подход позволил доказать алгоритмическую сводимость задачи нахождения решений системы алгебраических уравнений от многих переменных к задаче нахождения решений алгебраического уравнения от одной переменной. В дальнейшем этот метод нашел множество применений в различных теоретических и прикладных разделах математики, механики, криптографии и др. (см. [Buc01], [Laz83]).

Однако практическое использование метода сопряжено с колоссальными вычислительными затратами. Первые оценки трудоемкости этого метода были получены еще тогда, когда понятие базиса Гребнера еще не было известно. В работе [Her25] еще в 1925 г. была получена необходимая верхняя оценка в виде двойной экспоненты от числа переменных и максимальных степеней входящих описание задачи многочленов для степеней разложения элемента идеала по его произвольному базису. Как оказалось позднее [Dube] эта оценка не может быть улучшена.

Поскольку базис Гребнера зависит от допустимого упорядочения на множестве термов, естественно был поставлен вопрос о возможной зависимости трудоемкости вычисления такого базиса от такого прядочения. Поэтому сначала потребовалось описать такие упорядочения. Полное описание таких опорядочений было дано в работе [Rob85]. Однако этот же результат можно извлечь из более ранних работ [МИ53] и [Gio52].

В дальнейшем эти упорядочения использовались при построении специальных алгоритмов нахождения базиса Гребнера. Выделим особо алгоритмы F_4 и F_5 предложенные Фожеро ([Fau99], [Fau02]).

Экспоненциальная оценка для сложности задачи нахождения базиса Гребнера в общем случае получена в [KM96]. Среди других работ в этом направлении выделим [Bar04], [Bro87], [Giu84], [HL11], [Laz83], [May89], [May97], [MM82].

Для булевых идеалов в статье [KM96] доказана

Теорема. 1 *Для заданных базиса булева идеала* $I \subseteq Q[x_1,\ldots,x_n]$ *и допустимого порядка на термах* $<$ *единственный приведенный базис Гребнера для* I *относительно порядка* $<$ *(задача 1) может быть вычислен алгоритмом, объем используемой памяти которого ограничен сверху полиномом от длины входных данных.*

В данной работе доказано, что размер базиса Гребнера G булева идеала I может иметь экспоненциальный размер относительно размера описания базиса идеала I , т.е. выполняется

Теорема. 2 *Задача построения базиса Гребнера булева идеала не принадлежит классу* FP .

2. Основные определения

Пусть M — произвольное множество. Отображение

$$M \times M \to M \qquad (1)$$

будем называть операцией. Для операции могут использоваться мультипликативная и аддитивная запись — xy и $x+y$ соответственно. Аддитивная запись обычно используется только для коммутативной операции, т.е. когда $x+y=y+x$ для всех пар (x,y).

Операция называется ассоциативной, если для всех троек (x,y,z) выполняются соотношения $(xy)z = x(yz)$.

Если для всех пар x,y из M выполняются соотношения $xy = yx$, то операция называется коммутативной.

Элемент $e \in M$ называется единичным элементом, если для всех $x \in M$ выполняются соотношения $ex = x = xe$. (В случае аддитивной записи операции единичный элемент обозначается через 0 и называется нулевым элементом.) Единичный элемент единствен, если он существует. Действительно, пусть e' — другой единичный элемент. Тогда выполняются соотношения

$$e = ee' = e'.$$

Определение 1. 3 *Множество* M *с ассоциативной операцией называется полугруппой. Полугруппа, содержащая единицу, называется моноидом.*

Определение 2. 4 *Отношение* $\prec$ *на моноиде* M *называется допустимым упорядочением моноида* M, *если для этого отношения выполняются следующие свойства:*

• Для любых различных элементов ν_1 и ν_2 моноида M всегда выполненяется в точности одно из соотношений $\nu_1 \prec \nu_2$ или $\nu_2 \prec \nu_1$.

• Если $\nu_1 \prec \nu_2$, то для любого $\nu \in M$ выполняется соотношение $\nu \cdot \nu_1 \prec \nu \cdot \nu_2$.

• В любом непустом подмножестве $M_0 \subset M$ всегда существует наименьший элемент $\nu_0 \in M_0$:

$$\forall \nu \in M_0 \quad \nu_0 \prec \nu \ \textbf{\textit{или}} \ \nu_0 = \nu.$$

Отношение $\nu \prec \mu$ будем записывать также как $\mu \succ \nu$.

Пусть $X = \{x_1, \ldots, x_n\}$ — конечное множество. Обозначим через $\mathbb{Z}_+$ множество неотрицательных целых чисел. Выражения вида $x_1^{\omega_1} \ldots x_n^{\omega_n}$, где

$$\omega = \{\omega_1, \ldots, \omega_n\} \in \mathbb{Z}_+^n,$$

будем называть термами на множестве X и обозначать $T\langle X \rangle$. Множество $T\langle X \rangle$ является моноидом относительно операции умножения, заданной формулой

$$x_1^{\omega_1} \ldots x_n^{\omega_n} \cdot x_1^{\eta_1} \ldots x_n^{\eta_n} = x_1^{\omega_1 + \eta_1} \ldots x_n^{\omega_n + \eta_n}.$$

На моноиде термов $T\langle X \rangle$ существует бесконечно много допустимых упорядочений. Из определения моноида термов следует, что достаточно описать допустимое упорядочение на множестве $\mathbb{Z}_+^n$.

Примеры допустимых упорядочений на множестве векторов из $\mathbb{Z}_+^n$.

• **Лексикографический порядок.** Для $\alpha, \beta \in \mathbb{Z}_+^n$ будем считать, что $\alpha \succ \beta$, если в векторе разности $\alpha - \beta$ первая слева ненулевая координата положительна.

• **Лексикографический порядок в градуированном моноиде.** Градуируем моноид $M = \bigcup_{i=0}^{\infty} M_i$, где $M_i = \{\alpha \in \mathbb{Z}_+^n \mid \alpha_1 + \ldots + \alpha_n = i\}$. При $i > j$ всякий элемент множества M_i больше любого элемента множества M_j. Порядок в каждом M_i зададим как лексикографический.

• **Обратный лексикографический порядок в градуированном моноиде.** Градуируем моноид $M = \bigcup_{i=0}^{\infty} M_i$, где $M_i = \{\alpha \in \mathbb{Z}_+^n \mid \alpha_1 + \ldots + \alpha_n = i\}$. При $i > j$ всякий элемент множества M_i больше любого элемента множества M_j. Порядок в каждом M_i зададим как обратный лексикографический, т. е. для элементов $\alpha, \beta \in M_i$ выполняется соотношение $\alpha \succ \beta$ тогда и только тогда, когда первая справа ненулевая координата в векторе разности $\alpha - \beta$ отрицательная.

Фиксируем некоторый допустимый порядок $\prec$ на множестве термов $T\langle X\rangle$. Фиксируем также некоторое поле K. Элементами кольца многочленов на множестве переменных $X = \{x_1, \ldots, x_n\}$ являются выражения вида

$$\sum_{t \in T\langle X\rangle} a_t \cdot t,$$

где $a_t \in K$ и $a_t \neq 0$ только для конечного множества термов $t \in T\langle X\rangle$.

Определение. 5 *Пусть A — поле. Конечное множество G называется базисом Гребнера идеала I кольца $A[X]$, если*

- G — базис идеала I,
- $\forall f \in I \; \exists g \in G \mid HT(g) \mid HT(f)$.

Поскольку A — поле, старший моном любого элемента идеала делится на старший моном некоторого элемента базиса Гребнера этого идеала. Отметим также, что базисы Гребнера идеала для различных допустимых порядков на множестве термов $T\langle X\rangle$ различны.

3. Простая система уравнений со сложным базисом Гребнера

Теорема 1. 6 *Задача построения базиса Гребнера булева идеала не принадлежит классу FP.*

Рассмотрим кольцо многочленов $K[X]$ от переменных множества $X = \{x_1, \ldots, x_n\}$ над произвольным полем K. Фиксируем целое число $0 < s \leq n$. Для целого $0 < i \leq s$ обозначим через $\sigma_i^{(s)}(x_1, \ldots, x_s)$ i-й симметрический многочлен. Далее будем предполагать, что хактерика поля K либо 0, либо больше s.

Фиксируем неотрицательное целое число $k \leq n$, где n — число переменных в кольце многочленов. Рассмотрим идеал I, порожденный многочленами

$$f_0(x_1, \ldots, x_n) = x_1 + \ldots + x_n - k, \qquad f_i(x_1, \ldots, x_n) = x_i^2 - x_i, \; i = 1, \ldots, n. \quad (2)$$

Рассмотрим множество многочленов $F = \{ f_i \mid i = 1,\ldots,n \}$.

Лемма 1. 7 *Пусть идеал* I *определен многочленами из формулы (2) и заданы два целых числа* $0 < i \le s < n$. *Тогда существует такой многочлен* $P_{i,s}(x_{s+1},\ldots,x_n)$ *степени не выше* i *от переменных* $x_{s+1},\ldots,x_n$, *что*

$$\sigma_i^{(s)}(x_1,\ldots,x_s) + P_{i,s}(x_{s+1},\ldots,x_n) \in I.$$

Доказательство. Воспользуемся индукцией по i.

При $i = 1$ определим многочлен $P_{i,s}(x_{s+1},\ldots,x_n)$ формулой

$$P_{1,s}(x_{s+1},\ldots,x_n) = x_{s+1} + \ldots + x_n - k = \sigma_1^{(n-s)}(x_{s+1},\ldots,x_n) - k.$$

Тогда

$$f_0(x_1,\ldots,x_n) = \sigma_1^{(s)}(x_1 + \ldots, x_s) + P_{1,s}(x_{s+1},\ldots,x_n) \in I.$$

Предположим, что существование многочленов $P_{i,s}$ доказано для всех $i < j \le s$. Докажем существование многочлена $P_{j,s}$. Заметим, что для любых a и b в силу равенства

$$a^j - b^j = (a-b)(a^{j-1} + \ldots + b^{j-1})$$

и определения идеала из соотношения $a - b \in I$ следует, что $a^j - b^j \in I$.

Положим

$$\begin{array}{lll} a = x_1 + \ldots + x_s & = & \sigma_1^{(s)}(x_1,\ldots,x_s) b = -x_{s+1} - \ldots - x_n + k \\ = & -\sigma_1^{(n-s)}(x_{s+1},\ldots,x_n) + k. & \end{array}$$

Легко видеть, что

$$a^j = j! \cdot \sigma_j^{(s)}(x_1,\ldots,x_s) + L\ \sigma_1^{(s)}(x_1,\ldots,x_s),\ldots,\sigma_{j-1}^{(s)}(x_1,\ldots,x_s)\ (\mathrm{mod}\, F),$$

где L — линейная форма. Тогда при некотором $h \in I$ выполняется соотношение

$$\begin{array}{lll} a^j & = & j! \cdot \sigma_j^{(s)}(x_1,\ldots,x_s) + L\ \sigma_1^{(s)}(x_1,\ldots,x_s),\ldots,\sigma_{j-1}^{(s)}(x_1,\ldots,x_s) \\ & + & h(x_1,\ldots,x_n) \end{array}$$

Согласно предположению индукции для для всех $i = 1,\ldots, j-1$ существуют $h_i \in I$ для которых выполняются соотношения

$$\sigma_i^{(s)}(x_1,\ldots,x_s) = h_i(x_1,\ldots,x_n) - P_{i,s}(x_{s+1},\ldots,x_n).$$

Поскольку $a^j - b^j \in I$, и $b^j = Q_{j,s}(x_{s+1},\ldots,x_n)$ и $\deg Q_{j,s} \le j$, то при некотором $g \in I$ выполняется соотношение

$$a^j - b^j = j!\cdot\sigma_j^{(s)}(x_1,\ldots,x_s) - L\ P_{i,s}(x_{s+1},\ldots,x_n),\ldots,P_{i,s}(x_{s+1},\ldots,x_n) - Q_{j,s}(x_{s+1},\ldots,x_n) + g(x_1,\ldots,x_n)$$

и, следовательно, можно определить многочлен $P_{j,s}$ формулой

$$P_{j,s}(x_{s+1},...,x_n) = \frac{1}{j!}(Q_{j,s}(x_{s+1},...,x_n) + L(P_{i,s}(x_{s+1},...,x_n),...,P_{i,s}(x_{s+1},...,x_n)))$$

Введем обозначение

$$B^s = \underbrace{\{0,1\}\times\ldots\times\{0,1\}}_{s\ \textit{сомножителей}}.$$

Для $\omega = (i_1,\ldots,i_s) \in B^s$ положим

$$|\omega| = i_1 + \ldots + i_s.$$

Для любого целого $0 < k < n$ определим разбиение множества X на два множества $Y = \{x_1,\ldots,x_{k-1}\}$ (в случае $k = 1$ это множество пустое) и множество $Z = \{x_k,\ldots,x_n\}$. Напомним, что для любого $\omega \in B^{k-1}$ определен терм

$$Y^\omega = x_1^{i_1}\ldots x_{k-1}^{i_{k-1}},$$

где $\omega = (i_1,\ldots,i_{k-1})$, и для любого $\eta \in B^{n-k+1}$ определен терм

$$Z^\eta = x_k^{j_1}\ldots x_n^{j_{n-k+1}},$$

где $\eta = (j_1,\ldots,j_{n-k+1})$.

Лемма 2. 8 *Пусть $0 < k < 2k < n$ и характеристика поля K больше n или равна 0. Тогда в кольце многочленов $K[x_k,\ldots,x_n]$ существует единственный, с точностью до умножения на ненулевую константу, ненулевой многочлен вида*

$$\sum_{\omega\in B^{n-k+1}}^{|\omega|\le k} \lambda_\omega Z^\omega,$$

принадлежащий идеалу I.

Доказательство. Вначале докажем существование указанного многочлена, воспользовавшись леммой 1.

Положим $s = k-1$. Согласно лемме 1 для всех $i < k$ определены многочлены $P_{i,k-1}$, удовлетворяющие условиям

$$\sigma_i^{(k-1)}(x_1,\ldots,x_{k-1})+P_{i,k-1}(x_k,\ldots,x_n)\in I.$$

Поскольку $\deg P_{i,k-1}\le i$, а идеал I — булев (см. определение 3), то существуют такие многочлены $h_{i,k}\in I$ и

$$Q_{i,k}(x_k,\ldots,x_n)=\sum_{\omega\in B^{n-k+1}}^{|\omega|\le i}\lambda_{\omega,k}Z^{\omega}\in K[Z],$$

что

$$P_{i,k-1}(x_k,\ldots,x_n)=Q_{i,k}(x_k,\ldots,x_n)+h_{i,s}(x_k,\ldots,x_n).$$

Тогда

$$\sigma_i^{(k-1)}(x_1,\ldots,x_{k-1})+Q_{i,s}(x_k,\ldots,x_n)\in I \qquad (3)$$

для всех пар i,k, удовлетворяющих условию $0<i\le k-1<n$.

Заметим, что в силу равенства

$$a^k-b^k=(a-b)(a^{k-1}+\ldots+b^{k-1})$$

и определения идеала из соотношения $a-b\in I$ следует, что $a^k-b^k\in I$.

Положим

$$a=x_1+\ldots+x_{k-1}b=-x_k-\ldots-x_n+k.$$

Раскрывая скобки и используя соотношения $x_i^2=x_i$ при всех $i=1,\ldots,n$ получаем равенства

$$a^k=\sum_{i=0}^{k-1}\lambda_i\sigma_i^{(k-1)}(x_1,\ldots,x_{k-1})\pmod I b^k=\sum_{i=0}^{k-1}\mu_i\sigma_i^{(n-k+1)}(x_k,\ldots,x_n)+$$
$$k!\cdot\sigma_k^{(n-k+1)}(x_k,\ldots,x_n)\pmod I.$$

Тогда в силу соотношений (3) выполняется равенство

$$a^k=\sum_{\omega\in B^{n-k+1}}^{|\omega|<k}\nu_{\omega}Z^{\omega}\pmod I$$

и, учитывая, что $a^k-b^k\in I$, получаем

$$k!\cdot\sigma_k^{(n-k+1)}(x_k,\ldots,x_n)+\sum_{i=0}^{k-1}\mu_i\sigma_i^{(n-k+1)}(x_k,\ldots,x_n)-\sum_{\omega\in B^{n-k+1}}^{|\omega|<k}\nu_{\omega}Z^{\omega}\in I.$$

Поскольку характеристика поля K больше k, многочлен

$$k!\cdot\sigma_k^{(n-k+1)}(x_k,\ldots,x_n)+\sum_{i=0}^{k-1}\mu_i\sigma_i^{(n-k+1)}(x_k,\ldots,x_n)-\sum_{\omega\in B^{n-k+1}}^{|\omega|<k}\nu_{\omega}Z^{\omega}$$

не равен 0, принадлежит кольцу многочленов $K[Z]$ и идеалу I.

Далее покажем, что указанный многочлен единствен. Выберем любой многочлен

$$p(x_k,\ldots,x_n)=\sum_{\omega\in B^{n-k+1}}^{|\omega|\le k}\lambda_\omega Z^\omega,$$

удовлетворяющий условиям леммы 2. Необходимым условием принадлежности многочлена p идеалу I является равенство этого многочлена нулю во всех корнях идеала I.

Разобьем множество M всех корней идеала I на непересекающиеся подмножества M_i, где $i=0,1,\ldots,k-1$. Для этого определим множество M_i как множество корней $(\alpha_1,\ldots,\alpha_n)$ идеала, удовлетворяющих условию $\alpha_1+\ldots+\alpha_{k-1}=i$. Множества M_i, очевидно, не пересекаются. Поскольку идеал I булев, то координаты корней могут быть равными только 0 или 1. Поэтому $\alpha_1+\ldots+\alpha_{k-1}<k$. Следовательно, множество корней M идеала I представимо в виде разбиения

$$M=\bigcup_{i=0}^{k-1}M_i.$$

Докажем, что

$$\lambda_\omega=(-1)^{|\omega|}\lambda_{\omega_0},\quad (4)$$

где

$$\omega_0=(0,\ldots,0).$$

Доказательство проведем индукцией по $i=|\omega|$.

Пусть $|\omega|=1$. Рассмотрим множество корней $M_{k-1}\subset M$. Поскольку $\alpha_1+\ldots+\alpha_{k-1}=k-1$, то только одна из координат $(\alpha_k,\ldots,\alpha_n)$ равна 1, а остальные нулевые. Все эти корни находятся во взимнооднозначном соответствии с такими векторами $\omega\in B^{n-k+1}$, что $|\omega|=1$. Поскольку многочлен p обращается в нуль на таких векторах, то для всех $|\omega|=1$ выполняется равенство $\lambda_\omega=-\lambda_{\omega_0}$.

Пусть равенство (4) доказано для всех $|\omega|<m\le k$. Докажем равенство (4) для всех $|\omega|=m$.

Рассмотрим множество корней $M_{k-m}\subset M$. Поскольку $\alpha_1+\ldots+\alpha_{k-1}=k-m$, то из координат $(\alpha_k,\ldots,\alpha_n)$ в точности m равны 1, а остальные нулевые. Все такие решения находятся во взимнооднозначном соответствии с такими

$\omega \in B^{n-k+1}$, что $|\omega| = m$. Заметим, что согласно определению множества M_{k-m} и предположения индукции для таких решений выполняется равенство

$$p(\alpha_k,\ldots,\alpha_n) = \sum_{\omega\in B^{n-k+1}}^{|\omega|\le m} \lambda_\omega A^\omega = \sum_{\omega\in B^{n-k+1}}^{|\omega|< m} (-1)^{|\omega|}\lambda_{\omega_0} A^\omega + \sum_{\eta\in B^{n-k+1}}^{|\eta|= m} \lambda_\eta A^\eta,$$

где $A = (\alpha_k,\ldots,\alpha_n)$. Согласно определению множества M_{k-m} и ввиду неравенства $2k < n$ при $0 \le i < m$ на решениях из M_{k-m} справедливо равенство

$$\sum_{\omega\in B^{n-k+1}}^{|\omega|= i} (-1)^{|\omega|}\lambda_{\omega_0} A^\omega = \sum_{\omega\in B^{n-k+1}}^{|\omega|= i} (-1)^i \lambda_{\omega_0} A^\omega = (-1)^i \cdot \binom{m}{i} \cdot \lambda_{\omega_0}.$$

Также выполнено равенство

$$\lambda_{\eta_0} = \sum_{\eta\in B^{n-k+1}}^{|\eta|= m} \lambda_\eta A^\eta,$$

где $\eta_0 = (\alpha_k,\ldots,\alpha_n)$. Заметим, что справедливы равенства

$$\sum_{i=0}^{m} \binom{m}{i} (-1)^i = (1-1)^m = 0. \qquad (5)$$

Поскольку многочлен p равен нулю на таких векторах $(\alpha_k,\ldots,\alpha_n)$, то выполняются равенства

$$p(\alpha_k,\ldots,\alpha_n) = \sum_{i=0}^{m-1} (-1)^i \cdot \binom{m}{i} \cdot \lambda_{\omega_0} + \lambda_{\eta_0} = 0.$$

Учитывая теперь соотношение (5) получаем равенство $\lambda_{\eta_0} = (-1)^m \cdot \lambda_{\omega_0}$. Следовательно, для всех $|\omega| = m$ выполняется равенство $\lambda_\omega = (-1)^{|\omega|} \cdot \lambda_{\omega_0}$.

Поэтому

$$p(\alpha_k,\ldots,\alpha_n) = \lambda_{\omega_0} \cdot \left(\sum_{i=0}^{k} \sigma_i^{(n-k+1)}(x_k,\ldots,x_n) \right).$$

Следствие 1. 9 *Пусть выполняются предположения леммы 2. Тогда многочлен*

$$p(\alpha_k,\ldots,\alpha_n) = \sum_{i=0}^{k} \sigma_i^{(n-k+1)}(x_k,\ldots,x_n). \qquad (6)$$

является неприводимым элементом базиса Гребнера идеала I для лексикографического упорядочения термов на множестве переменных $X = \{x_1, \ldots, x_n\}$.

Доказательство. Предположим, что многочлен p приводим. Тогда существует элемент g неприводимого базиса Гребнера идеала I, старший терм которого делит один из термов многочлена p. Следовательно, этот старший терм многочлена g содержит только переменные из множества $Z = \{x_k, \ldots, x_n\}$. Поскольку термы многочлена g упорядочены лексикографически, все его остальные термы также зависят только от переменных из множества Z. Поэтому многочлен g удовлетворяет условиям леммы 8, и многочлен $p \in K[x_k, \ldots, x_n]$ отличается от g ненулевым множителем, т.е. p неприводим.

Следующее утверждение завершает доказательство теоремы 1.

Следствие. 10 *При* $k = \left\lfloor \frac{n}{2} \right\rfloor$ *размер базис Гребнера идеала* I *экспоненциален относительно описания идеала* I *формулами (2).*

Доказательство. Многочлен p является неприводимым элементом базиса Гребнера идеала I и содержит $\binom{n}{k+1} > 2^{n/2}$ термов с коэффициентами 1.

4. Нахождение решений систем алгебраических уравнений с использованием базиса Гребнера

Определение 3. 11 *Идеал* I *в кольце многочленов* $K[x_1, \ldots, x_n]$ *над полем* K *называется булевым, если* $x_i^2 - x_i \in I$ *для любого* $0 < i \le n$. *Система уравнений, определяющая такой идеал, называется булевой.*

Теорема 2. 12 *Если булева система уравнений над полем* K *имеет единственное решение, то приведенный базис Гребнера идеала этой системы определен однозначно с точностью до коэффициентов и не зависит от выбора допустимого порядка на множестве мономов.*

Доказательство. В силу соотношений $x_i^2 = x_i$ все x_i принимают значения в множестве $\{0,1\}$, и следовательно, решения, принадлежащие алгебраическому замыканию поля K, лежат в поле K. Пусть единственным решением булевой системы уравнений является набор $x_i = a_i, i = 1, \ldots, n$. Докажем вначале, что идеал

$$I = \{x_1 - a_1, \ldots, x_n - a_n\}$$

совпадает в этом случае с идеалом, заданным булевыми уравнениями исходной системы уравнений. Действительно, поскольку $x_i - a_i$ обращается в нуль во всех решениях исходной системы, то согласно теореме Гильберта о нулях идеала при некотором натуральном k многочлен $(x_i - a_i)^k$ принадлежит идеалу, заданному исходной булевой системой уравнений, а тогда и сам многочлен $x_i - a_i$ принадлежит этому идеалу, поскольку выполняется равенство $x_i^2 = x_i$. Следовательно, идеал I лежит в идеале J, заданном булевыми уравнениями исходной системы уравнений. Очевидно, что для любого многочлена $f \in J$ выполняется соотношение $f \to_{F^*} \underline{a}$, где $a \in \{0,1\}$. Если этот элемент ненулевой, то система не имеет решения. Если этот элемент нулевой, то это означает, что $f \in I$. Поэтому $I = J$.

Пусть F_0 — приведенный базис Гребнера, относительно некоторого допустимого порядка. Пусть x_n — наименьший моном положительной степени относительно этого порядка. Тогда из $x_n - a_n \in I$ следует существование многочлена $h \in F_0$ старший моном которого делит моном x_n. Поскольку моном x_n — наименьший, этот моном делится на старший моном многочлена h. Следовательно, $x_n - a_n \in F_0$. Переменная x_n не входит ни в один оставшихся многочленов базиса Гребнера F_0. Следовательно, рассуждая как выше получаем, что в базис Гребнера входит также $x_{n-1} - a_{n-1}$, где x_{n-1} следующая по старшинству переменная.

Следствие 3. 13 *Задачи нахождения базиса Гребнера и единственного решения для булевых идеалов, имеющих единственное решение эквивалентны.*

Литература.

[Bar04] Bardet, M. and Faugere, J.C and Salvy, B. “On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations”.In: International Conference on Polynomial System Solving - ICPSS. Paris, France, Nov. 2004, pp. 71 – 75.

[Bro87] D. Brownawell. “Bounds for the degrees in the Nullstellensatz”. In: Annals of Math. Second Series 126.3 (1987), pp. 577–591.

[Buc06] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal).* PhD thesis, Universität Innsbruck, 1965. English translation in J. of Symbolic Computation, Special Issue on Logic, Mathematics, and Computer Science: Interactions. 41(3/4):475--511, 2006.

[Buc01] B. Buchberger. Gröbner Bases : A Short Introduction for Systems Theorists, Computer Aided Systems Theory—EUROCAST 2001 (2001) Volume: 330, Issue: 9, Publisher: Springer, Pages: 1–19.

[Gio52] Trevisan Giorgio. “Classificazione dei semplici ordinamenti di un gruppo libero commutativo con n generatori”. In: vol. 22. CEDAM, 1952, pp. 143–156.

[Giu84] Marc Giusti.“Some Effectivity Problems in Polynomial Ideal Theory”. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation. EUROSAM’84. London, UK: Springer-Verlag, 1984, pp. 159–171.

[Fau02] J.-C. Faugere. “A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)”. In: Proceedings of the 2002 international symposium on Symbolic and algebraic computation. ISSAC ’02. Lille, France: ACM, 2002, pp. 75–83.

[Fau99] J.-C. Faugere. “A new efficient algorithm for computing Gröbner bases (F4).” In: Journal of Pure and Applied Algebra 139. 1–3(June 1999), pp. 61–88.

[Her25] G. Hermann. Die Frage der endlich vielen Schritte in der Theorie der Polynomideale: Unt. Benutzung nachgelassener Satze v .Kurt Hentzelt. Springer, 1925.

[HL11] Amir Hashemi and Daniel Lazard. “Sharper Complexity Bounds for Zero-Dimensional Gröbner Bases and Polynomial System Solving”. In: IJAC 21.5 (2011), pp. 703–713.

[KM96] Klaus Kühnle and Ernst W. Mayr. “Exponential space computation of Gröbner bases”. In: Proceedings of the 1996 international symposium on Symbolic and algebraic computation. ISSAC’96. Zurich, Switzerland: ACM, 1996, pp. 63–71.

[Laz83] Daniel Lazard. “Gröbner-Bases, Gaussian elimination and resolution of systems of algebraic equations”. In: Proceedings of the European Computer Algebra Conference on Computer Algebra. London, UK: Springer-Verlag, 1983, pp. 146–156.

[May89] Ernst W. Mayr. “Membership in Polynomial Ideals over Q Is Exponential Space Complete”. In: STACS. Ed. by Burkhard Monien and Robert Cori. Vol. 349. Lecture Notes in Computer Science. Springer, 1989, pp. 400–406.

[May97] Ernst W. Mayr. “Some Complexity Results for Polynomial Ideals”. In: J. Complexity 13.3 (1997), pp. 303–325.

[MM82] E. Mayr and A. Meyer. “The complexity of the word problems for commutative semigroups and polynomial ideals”.English.In:Adv.Math., Beijing 46.3 (Dec. 1982), pp. 305–329.

[Riq10] C. Riquier. Les systèmes d’équations aux dérivées partielles. Cornell University Library historical math monographs. Gauthier-Villars, 1910.

[Rob85] Lorenzo Robbiano. Term orderings on the polynomial ring. Computer algebra, EUROCAL ’85, Proc. Eur. Conf., Linz/Austria 1985, Vol. 2, Lect. Notes Comput. Sci. 204, 513-517 (1985). 1985.

[МИ53] Зайцева М.И.. “О совокупности упорядочений абелевой группы”. Успехи математических наук 8 (1953), pp. 135–137.

[Dube] T. W. Dube, The structure of polynomial ideals and Grobner bases, SIAM Jornal of Computing, 19: 750-773, 1990.

Comparing complexities of problems of determining of Grebner's basis of ideal and solving this ideal

Shokurov A.V. ISP RAS, Moscow, Russia

Abstract. A new method of investigation of ideals in the rings of polynomials was proposed by B. Buchberger in 1965. He proposed to use special basis in such rings named "Gröbner Basis" in the honor of his teacher. The proposed approach has allowed to prove the algorithmic reducibility of the task of finding solutions of a system of algebraic equations in many variables to the problem of solving algebraic equation of one variable. However, the practical use of the method needs enormous computational cost. The first estimate of the computational complexity of this method was obtained by G. Hermann in 1925, when the concept of Gröbner basis was not yet known. Was obtained the required upper estimate in the form of dual exponent of the number of variables and the maximum degree of incoming task description polynomials of degree of decomposition element ideal for an arbitrary basis. As it turned out later by T.W. Dube this estimate cannot be improved.

In 1996 K. Kühnle and E. W. Mayr proved, that for Boolean ideals the upper bound of memory capacity is exponential on input size. For zero-dimensional ideals A. Hashemi and D. Lazard proved the exponential complexity of finding Gröbner basis. Here we prove the lower bound of computation of such bases, by giving an example of an ideal having Gröbner basis of exponential size on input.

References.

[Bar04] Bardet, M. and Faugere, J.C and Salvy, B. "On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations".In: International Conference on Polynomial System Solving - ICPSS. Paris, France, Nov. 2004, pp. 71 – 75.

[Bro87] D. Brownawell. "Bounds for the degrees in the Nullstellensatz". In: Annals of Math. Second Series 126.3 (1987), pp. 577–591.

[Buc06] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal).* PhD thesis, Universität Innsbruck, 1965. English translation in J. of Symbolic Computation, Special Issue on Logic, Mathematics, and Computer Science: Interactions. 41(3/4):475--511, 2006.

[Buc01] B. Buchberger. Gröbner Bases : A Short Introduction for Systems Theorists, Computer Aided Systems Theory—EUROCAST 2001 (2001) Volume: 330, Issue: 9, Publisher: Springer, Pages: 1–19.

[Gio52] Trevisan Giorgio. "Classificazione dei semplici ordinamenti di un gruppo libero commutativo con n generatori". In: vol. 22. CEDAM, 1952, pp. 143–156.

[Giu84] Marc Giusti.“Some Effectivity Problems in Polynomial Ideal Theory”. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation. EUROSAM’84. London, UK: Springer-Verlag, 1984, pp. 159–171.

[Fau02] J.-C. Faugere. “A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)”. In: Proceedings of the 2002 international symposium on Symbolic and algebraic computation. ISSAC ’02. Lille, France: ACM, 2002, pp. 75–83.

[Fau99] J.-C. Faugere. “A new efficient algorithm for computing Gröbner bases (F4).” In: Journal of Pure and Applied Algebra 139. 1–3(June 1999), pp. 61–88.

[Her25] G. Hermann. Die Frage der endlich vielen Schritte in der Theorie der Polynomideale: Unt. Benutzung nachgelassener Satze v .Kurt Hentzelt. Springer, 1925.

[HL11] Amir Hashemi and Daniel Lazard. “Sharper Complexity Bounds for Zero-Dimensional Gröbner Bases and Polynomial System Solving”. In: IJAC 21.5 (2011), pp. 703–713.

[KM96] Klaus Kühnle and Ernst W. Mayr. “Exponential space computation of Gröbner bases”. In: Proceedings of the 1996 international symposium on Symbolic and algebraic computation. ISSAC’96. Zurich, Switzerland: ACM, 1996, pp. 63–71.

[Laz83] Daniel Lazard. “Gröbner-Bases, Gaussian elimination and resolution of systems of algebraic equations”. In: Proceedings of the European Computer Algebra Conference on Computer Algebra. London, UK: Springer-Verlag, 1983, pp. 146–156.

[May89] Ernst W. Mayr. “Membership in Polynomial Ideals over Q Is Exponential Space Complete”. In: STACS. Ed. by Burkhard Monien and Robert Cori. Vol. 349. Lecture Notes in Computer Science. Springer, 1989, pp. 400–406.

[May97] Ernst W. Mayr. “Some Complexity Results for Polynomial Ideals”. In: J. Complexity 13.3 (1997), pp. 303–325.

[MM82] E. Mayr and A. Meyer. “The complexity of the word problems for commutative semigroups and polynomial ideals”.English.In:Adv.Math., Beijing 46.3 (Dec. 1982), pp. 305–329.

[Riq10] C. Riquier. Les systèmes d’équations aux dérivées partielles. Cornell University Library historical math monographs. Gauthier-Villars, 1910.

[Rob85] Lorenzo Robbiano. Term orderings on the polynomial ring. Computer algebra, EUROCAL ’85, Proc. Eur. Conf., Linz/Austria 1985, Vol. 2, Lect. Notes Comput. Sci. 204, 513-517 (1985). 1985.

[МИ53] Зайцева М.И.. “О совокупности упорядочений абелевой группы”. Успехи математических наук 8 (1953), pp. 135–137.

[Dube] T. W. Dube, The structure of polynomial ideals and Grobner bases, SIAM Jornal of Computing, 19: 750-773, 1990.

О построении аппроксимирующих функций характеристик малого числа систем

Хетагуров Я.А. <tshe@bk.ru>
Концерн АГАТ

Аннотация. Выявляется взаимное влияние характеристик на показатели системы, части, устройства посредством построения общих математических моделей (ММ), использующих одинаковые характеристики систем, частей, устройств. Для получения уравнений ММ применяются линейные и гиперболические аппроксимирующие зависимости, основанные на данных характеристик систем, частей и устройств, близких по назначению и применению.

Ключевые слова: характеристики систем, математические модели, аппроксимирующие зависимости

Проблема учета влияния большого числа факторов на основные показатели цифровых управляющих систем реального времени с вычислительными машинами на начальном этапе проектирования решается в основном выбором характеристик, которые комплексно учитывают области работы системы, требования к системе и затратам материальных ресурсов, выпускаемым системой продуктам и объемам их производства.

На величины этих характеристик оказывают влияние различные факторы, связанные с условиями построения системы и её частей, с производственными возможностями предприятия, с особенностями эксплуатации, внешними условиями (финансирование, сроки, важность системы), особенно с опытом ведущих разработчиков и субъективными условиями организации работы на предприятии или предприятиях, создающих систему.

Для выявления взаимного влияния характеристик на показатели системы, части, устройства будем осуществлять построение общих математических моделей (ММ), использующих одинаковые характеристики систем, частей, устройств. Для получения уравнений ММ применим линейные и гиперболические аппроксимирующие зависимости, использующие характеристики близких по назначению и применению систем, частей и устройств.

Выделим две группы ММ.

В первой группе используются ММ, связывающие две или три характеристики двух систем.

Во второй группе используются ММ, связывающие три или четыре характеристики трех систем.

Для повышения точности результатов увеличивается число пар или троек используемых близких по назначению систем.

Для выбранных наборов характеристик строятся в прямоугольной системе координат математические модели

первой группы $x= f (y)$ и $x= f (y,z)$

второй группы $x=f (y, z)$ и $x= f (y,z,q)$

При построении зависимостей в моделях первой группы по данным двух систем между двумя характеристиками $x= f (y)$ необходимы пары координат (точек) $x1y1$, $x2y2$,… $xiyi$,…; между тремя характеристиками $x= f (y,z)$ необходимы тройки координат для каждой точки $x1y1Z1$, $x2y2Z2$, … $xiyiZi$…Каждая точка представляет систему.

При построении зависимостей в моделях второй группы по данным трех систем между характеристиками $x=f (y, z)$ необходимы данные трех и более групп координат $x1y1Z1$, $x2y2Z2$, $x3y3$ $Z3$, … $xiyi$ Zi, …; между четырьмя характеристиками $x= f (y,z,q)$ необходимы данные четырех координат для каждой точки $x1y1Z1q1$, $x2y2Z2q2$, $x3y3Z3q3$, … $xiyiZiq$….. Каждая тройка или четверка координат представляет одну систему.

Увеличение числа анализируемых систем в моделях первой группы для выбранных характеристик увеличивает соответственно количество пар или троек координат – точек, что обеспечивает возможность применения аппроксимирующих уравнений второго порядка для трех точек; третьего порядка для четырех точек. Увеличение порядка аппроксимирующих уравнений повышает точность решения аппроксимирующего уравнения. Однако увеличение степени уравнения существенно усложняет определение коэффициентов уравнения, а также повышает требования к точности представления характеристик.

Увеличение числа, анализируемых систем в моделях второй группы также приводит к увеличению порядка аппроксимирующих уравнений и существенно усложняет и увеличивает объемы вычислений для получения коэффициентов аппроксимирующего уравнения. Также усиливается влияние точности представления коэффициентов.

Используем для вычислений аппроксимирующих коэффициентов для 3-х систем (3-х точек) три уравнения (линейные или гиперболические) с усреднением получаемых коэффициентов.

Основным показателем, оценивающим качество аппроксимирующей зависимости, является точность соответствия вычисленной величины по аппроксимирующей формуле фактическому значению величины.

Оценка точности математической модели, представленной аппроксимирующей функцией, в теории эксперимента проводится с использованием 25 – 30 данных опытов. Для нашего применения это соответствует данным о 25 – 30 системах. Учитывая, что для построения аппроксимирующего выражения используются данные всего 2 – 5 систем, применение методов оценки точности из теории эксперимента для наших условий невозможно.

Рассмотрим аппроксимацию и оценки точности ММ первой группы с использованием двух систем с двумя или тремя характеристиками. При проведении аппроксимации применяются два вида уравнений: линейные и гиперболические.

Для выполнения линейной или гиперболической аппроксимации необходимы данные двух точек, характеризующие работу двух систем. При использовании двух характеристик в каждой системе точки имеют координаты x_1y_1, x_2y_2, при использовании трех характеристик – координаты точек $x_1y_1z_1$, $x_2y_2z_2$. На основе приведенных координат двух точек определяются две ММ с двумя аппроксимирующими коэффициентами, связывающие характеристики y=f(x) и y = f(x,z) и представляемые соответственно линейными уравнениями

$$y = a_1x + a_0 \quad \text{и} \quad y = a_1x + a_0z$$

и гиперболическими уравнениями

$$y = \frac{a_0}{1+ a_1x} \quad \text{или} \quad y = \frac{a_0x}{1+ a_1x}$$

и

$$y = \frac{a_0z}{1+ a_1x}$$

Определение коэффициентов при двух характеристиках системы производится, решая систему из двух уравнений.

Коэффициенты аппроксимации a0, a1 для линейной функции при двух характеристиках системы определяются решением системы уравнений:

$$\begin{cases} y_1 = a_1x_1 + a_0 \\ y_2 = a_1x_2 + a_0 \end{cases}$$

где

$$a_1 = \frac{y_1 - y_2}{x_1 - x_2}$$

$$a_0 = \frac{y_2x_1 - y_1x_2}{x_1 - x_2}$$

При трех характеристиках систем определение коэффициентов a0, a1 производится решением системы уравнений

$$\begin{cases} y_1 = a_1x_1 + a_0\, z_1 \\ y_2 = a_1x_2 + a_0z_2 \end{cases}$$

где

$$a_1 = \frac{y_1z_2 - y_2z_1}{x_1z_2 - x_2z_1}$$

$$a_0 = \frac{y_2x_1 - y_1x_2}{x_1z_2 - x_2z_1}$$

Коэффициенты a0, a1 для гиперболической функции $y = \frac{a_0}{1+ a_1x}$ при двух характеристиках систем определяются решением системы

$$\begin{cases} y_1 = \dfrac{a_0}{1 - a_1x_1} \\ y_2 = \dfrac{a_0}{1 - a_1x_2} \end{cases}$$

где

$$a_1 = \frac{y_2 - y_1}{y_2x_2 - y_1x_1}$$

$$a_0 = \frac{y_1y_2(x_2 - x_1)}{y_2x_2 - y_1x_1}$$

Коэффициенты a0, a1 для гиперболической функции $\mathrm{y} = \frac{\mathrm{a_0x}}{1+ \mathrm{a_1x}}$ определяются решением системы

$$\begin{cases} y_1 = \dfrac{a_0x_1}{1 + a_1x_1} \\ y_2 = \dfrac{a_0x_2}{1 + a_1x_2} \end{cases}$$

где

$$a_1 = \frac{y_1x_2 - y_2x_1}{x_1x_2(y_1 - y_2)}$$

$$a_0 = \frac{y_1 y_2 (x_1 - x_2)}{x_1 x_2 (y_1 - y_2)}$$

При трех характеристиках системы эти коэффициенты определяются решением системы уравнений

$$\begin{cases} y_1 = \dfrac{a_0 z_1}{1 - a_1 x_1} \\ y_2 = \dfrac{a_0 z_2}{1 - a_1 x_2} \end{cases}$$

где

$$a_1 = \frac{y_1 z_1 - y_2 z_2}{y_1 x_1 z_2 - y_2 x_2 z_1}$$

$$a_0 = \frac{y_1 y_2 (x_1 - x_2)}{y_1 x_1 z_2 - y_2 x_2 z_1}$$

Для повышения точности получения коэффициентов аппроксимации линейного или гиперболического уравнения используются дополнительные системы (точки). На их основе, решая системы уравнений, определяются аппроксимирующие коэффициенты дополнительных уравнений. Суммируя эти одинаковые по наименованию коэффициенты уравнения, рассчитываются их средние значения. Использование средних значений коэффициентов уравнений (линейных или гиперболических), повышает точность решения.

Например, при наличии трех систем с тремя переменными и соответственно точек с координатами x1y1z1, x2y2z2, x3y3z3 строятся три системы линейных уравнений (x1y1z1, x2y2z2); (x2y2z2 , x3y3z3) и (x1y1z1, x3y3z3) для получения трех уравнений 1, 2, 3 (рис. 1) с использованием точек 1,2; 2,3 и 3.1

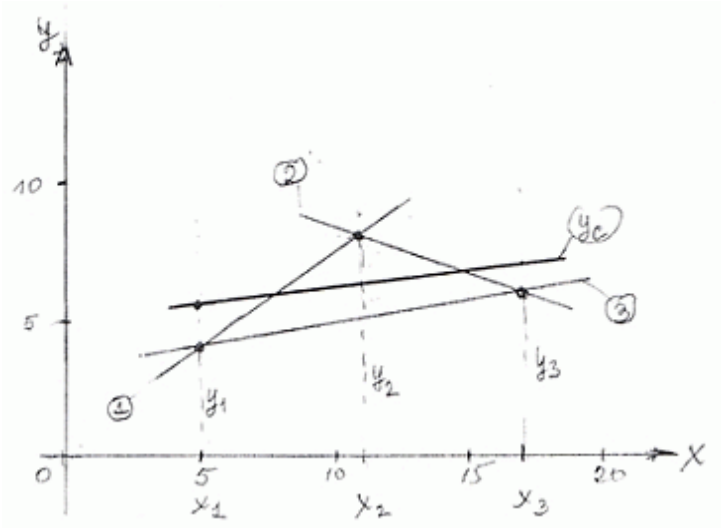


Рис 1

$$\begin{cases} y_1 = a_{11}x_1 + a_{01}z_1 \\ y_2 = a_{11}x_2 + a_{01}z_2 \end{cases}$$

$$\begin{cases} y_2 = a_{12}x_2 + a_{02}z_2 \\ y_3 = a_{12}x_3 + a_{02}z_3 \end{cases}$$

$$\begin{cases} y_1 = a_{13}x_1 + a_{03}z_1 \\ y_3 = a_{13}x_3 + a_{03}z_3 \end{cases}$$

где

$$a_{11} = \frac{y_1z_2 - y_2z_1}{x_1z_2 - x_2z_1} \qquad a_{01} = \frac{y_2x_1 - y_1x_2}{x_1z_2 - x_2z_1}$$

$$a_{12} = \frac{y_2z_3 - y_3z_2}{x_2z_3 - x_3z_2} \qquad a_{02} = \frac{y_2z_3 - y_3z_2}{x_2z_3 - x_3z_2}$$

$$a_{13} = \frac{y_3z_1 - y_1z_3}{x_3z_1 - x_1z_3} \qquad a_{03} = \frac{y_3z_1 - y_1z_3}{x_3z_1 - x_1z_3}$$

$$a_{03} = \frac{y_1 x_3 - y_3 x_1}{x_3 z_1 - x_1 z_3}$$

Для первой системы уравнений с данными y1x1z1 и y2x2z2 получим коэффициенты аппроксимации a11, a01.

Для второй системы уравнений с данными y2x2z2, y3x3 z3 – коэффициенты аппроксимации a12, a02.

Для третей системы уравнений y1x1z1, y3x3z3 – коэффициенты аппроксимации a13, a03.

Среднее значение коэффициентов аппроксимации запишем в виде

$$a_{1c} = \frac{\sum_1^3 a_{1i}}{3} \qquad a_{0c} = \frac{\sum_1^3 a_{0i}}{3}$$

и аппроксимирующую функцию с усредненными коэффициентами:

$$y_c = a_{1c}\, x + a_{0c}\, z$$

При четырех системах (точках) возможно построение до 6 линейных уравнений или гиперболических уравнений, использующих (для случая двух характеристик) следующие пары точек (x1y1, x2y2); (x2y2, x3y3); (x3y3, x4y4), (x1y1 , x4y4), (x3y3, x1y1), (x2y2 , x4y4) (1.2; 2.3; 3.4; 1.4; 3.1; 2.4).

При использовании не более двух раз одинаковых точек в парах для определения коэффициентов систем достаточно построить четыре системы линейных уравнений с характеристиками (x1y1, x2y2); (x2y2, x3y3), (x3y3, x4y4),(x4y4, x1y1), (1.2; 2.3; 3.4; 4.1). Запишем четыре системы уравнений 1, 2, 3, 4 (рис. 2) для определения коэффициентов аппроксимации для систем 1.2; 2.3; 3.4; 4.1.

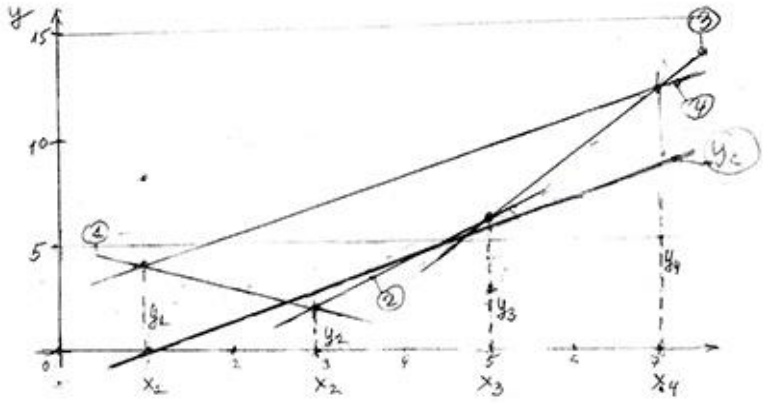


Рис 2

$$\begin{cases} y_1 = a_{11}x_1 + a_{01} \\ y_2 = a_{11}x_2 + a_{01} \end{cases}$$

$$\begin{cases} y_2 = a_{12}x_2 + a_{02} \\ y_3 = a_{12}x_3 + a_{02} \end{cases}$$

$$\begin{cases} y_3 = a_{13}x_3 + a_{03} \\ y_4 = a_{13}x_4 + a_{03} \end{cases}$$

$$\begin{cases} y_4 = a_{14}x_4 + a_{04} \\ y_1 = a_{14}x_1 + a_{04} \end{cases}$$

где

$$a_{11} = \frac{y_1 - y_2}{x_1 - x_2} \qquad a_{12} = \frac{y_2 - y_3}{x_2 - x_3}$$

$$a_{13} = \frac{y_3 - y_4}{x_3 - x_4} \qquad a_{14} = \frac{y_4 - y_1}{x_4 - x_1}$$

$$a_{01} = \frac{y_2x_1 - y_1x_2}{x_1 - x_2} \qquad a_{02} = \frac{y_3x_2 - y_2x_3}{x_2 - x_3}$$

$$a_{03} = \frac{y_4x_3 - y_3x_4}{x_3 - x_4} \qquad a_{04} = \frac{y_1x_4 - y_4x_1}{x_4 - x_1}$$

Среднее значение коэффициентов аппроксимации:

$$a_{1c} = \frac{\sum_1^4 a_{1i}}{4} \qquad a_{0c} = \frac{\sum_1^4 a_{0i}}{4}$$

Аппроксимирующую функцию с усредненными коэффициентами запишем в виде:

$$y_{c4} = a_{1c}x + a_{0c}$$

Определим аппроксимирующие коэффициенты гиперболических функций для трех систем 1, 2, 3.

Составим три группы уравнений для определения коэффициентов аппроксимации для каждой группы (рис. 3).

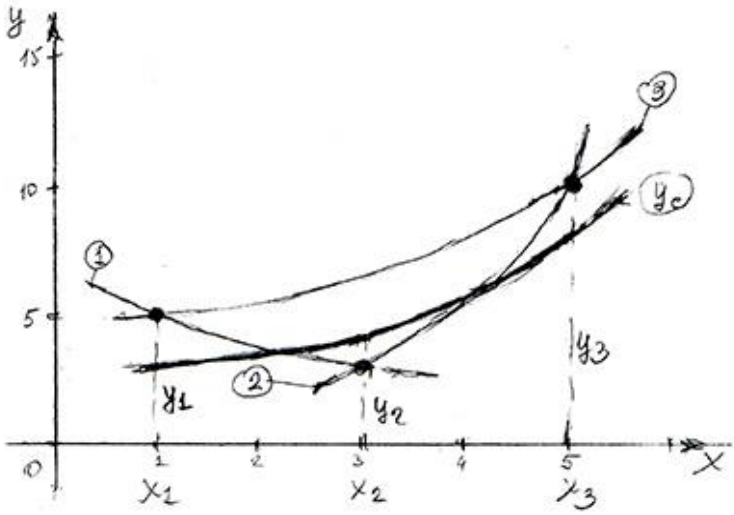


Рис 3

$$\begin{cases} y_1 = \dfrac{a_{01}}{1 - a_{11}x_1} \\ y_2 = \dfrac{a_{01}}{1 - a_{11}x_2} \end{cases}$$

$$\begin{cases} y_2 = \dfrac{a_{02}}{1 - a_{12}x_2} \\ y_3 = \dfrac{a_{02}}{1 - a_{12}x_3} \end{cases}$$

$$\begin{cases} y_1 = \dfrac{a_{03}}{1 - a_{13}x_3} \\ y_2 = \dfrac{a_{03}}{1 - a_{13}x_1} \end{cases}$$

где

$$a_{11} = \frac{y_1 - y_2}{y_2\, x_2 - y_1\, x_1}$$

$$a_{12} = \frac{y_2 - y_3}{y_3 x_3 - y_2 x_2}$$

$$a_{13} = \frac{y_3 - y_1}{y_1 x_1 - y_3 x_3}$$

$$a_{01} = \frac{y_1 y_2 (x_2 - x_1)}{y_2 x_2 - y_1 x_1}$$

$$a_{02} = \frac{y_2 y_3 (x_3 - x_2)}{y_3 x_3 - y_2 x_2}$$

$$a_{03} = \frac{y_1 y_3 (x_1 - x_3)}{y_1 x_1 - y_3 x_3}$$

Среднее значение коэффициентов

$$a_{1c} = \frac{\sum_1^3 a_{1i}}{3} \qquad a_{0c} = \frac{\sum_1^3 a_{0i}}{3}$$

Аппроксимирующая функция с усредненными коэффициентами

$$y_{c3} = \frac{a_{0c}}{1 - a_{1c} x}$$

При данных о пяти системах (точках) возможно построение 10 аппроксимирующих линейных или гиперболических уравнений. Для определения усредненных коэффициентов линейного или гиперболического уравнений достаточно использования пяти пар точек (x1y1, x2y2); (x2y2, x3y3); (x3y3, x4y4); (x4y4, x5y5), (x5y5, x1y1), характеризующих пять систем, исключая повторное использование данных точек. При решении уравнений определяются пять пар коэффициентов a11a01; a12a02, a13a03; a14a04; a15a05. Суммируя их, получим усредненные коэффициенты

$$a_{1c} = \frac{\sum_1^5 a_{1i}}{5} \qquad a_{0c} = \frac{\sum_1^5 a_{0i}}{5}$$

При увеличении числа систем с гиперболическими функциями аппроксимации увеличивается число систем уравнений аналогично линейной аппроксимации.

На основании приведенных примеров запишем последовательность действий для формирования линейной и гиперболической аппроксимирующих функций, использующих 3 и более систем.

В первую очередь производится построение систем уравнений для определения коэффициентов аппроксимации для каждой пары систем (по двум точкам). Количество систем уравнений соответствует числу систем (точек). Сумма этих коэффициентов аппроксимации делится на число систем - полученные величины являются усредненными коэффициентами аппроксимирующего уравнения для анализируемого числа систем.

Оценим точность линейных и гиперболических аппроксимирующих математических моделей.

Точность аппроксимирующей функции математической модели, учитывая малое число данных, оценим отклонениями решения усредненного аппроксимирующего уравнения от исходных данных и определением их средней величины. Чем меньше средняя величина отклонения, тем точнее аппроксимация математической модели.

Для линейной математической модели точность аппроксимации – средняя величина отклонения решения аппроксимирующего уравнения при использовании данных от трех систем определяется отклонениями

$$y_1 - y_{1c} = \Delta y_1$$

$$y_2 - y_{2c} = \Delta y_2$$

$$y_3 - y_{3c} = \Delta y_3$$

где

$$y_1 = a_{11}x_1 + a_{01} \quad y_2 = a_{12}x_2 + a_{02} \quad y_3 = a_{13}x_3 + a_{03}$$

$$y_{1c} = a_{1c}x_1 + a_{0c} \quad y_{2c} = a_{1c}x_2 + a_{02} \quad y_{3c} = a_{1c}x_3 + a_{0c}$$

Точность оценивается усредненной суммой отклонений

$$\delta = \frac{\sum_1^3 \Delta y_i}{3}$$

Чем меньше величина δ, тем выше точность аппроксимирующей функции.

Усредненную оценку точности аппроксимации линейной математической модели для 4-х систем запишем в виде

$$\delta = \frac{\sum_1^4 \Delta y_i}{4} \text{ где } \quad \Delta y_i = y_i - y_{ic}$$

Для определения точности аппроксимирующей гиперболической функции используется такой же подход, как и для аппроксимирующей линейной функции.

Рассмотрим аппроксимацию и оценки точности ММ второй группы с использованием трех систем с тремя характеристиками. При проведении аппроксимации применяются два вида уравнений: линейные и гиперболические с тремя коэффициентами аппроксимации.

При линейной и гиперболической зависимостях характеристик аппроксимирующая математическая модель может иметь два базовых варианта:

$$y = a_1 x + a_2 z + a_3$$

$$y = a_1 x + \frac{a_2}{1 - a_3 z}$$

Определение коэффициентов аппроксимации у первого базового варианта производим путем последовательного решения двух уравнений

$$y = a_1 x + A \qquad \text{и} \qquad A = a_2 z + a_3$$

по трем характеристикам от каждой из трех систем

$x_1, y_1, z_1; x_2, y_2, z_2; x_3, y_3, z_3$.

Для первого уравнения запишем три системы уравнений для определения коэффициента a1 по определенным коэффициентам a11, A1; a12,A2 и a13,A3.

$$\begin{cases} y_1 = a_{11} x_1 + A_1 \\ y_2 = a_{11} x_2 + A_1 \end{cases}$$

$$\begin{cases} y_2 = a_{12} x_2 + A_2 \\ y_3 = a_{12} x_3 + A_2 \end{cases}$$

$$\begin{cases} y_3 = a_{13} x_3 + A_3 \\ y_1 = a_{13} x_1 + A_3 \end{cases}$$

Вычисляем среднее значение коэффициента a1, которое запишем

$$a_{1c} = \frac{\sum_1^3 a_{1i}}{3}$$

Для второго уравнения запишем три системы уравнений для определения коэффициентов a2 и a3

$$\begin{cases} A_1 = a_{21} z_1 + a_{31} \\ A_2 = a_{21} z_2 + a_{31} \end{cases}$$

$$\begin{cases} A_2 = a_{22} z_2 + a_{32} \\ A_3 = a_{22} z_3 + a_{32} \end{cases}$$

$$\begin{cases} A_3 = a_{23} z_3 + a_{33} \\ A_1 = a_{23} z_1 + a_{33} \end{cases}$$

По определенным коэффициентам a21a31; a22a32; a23a33 вычислим средние значения коэффициентов a2c и a3c

$$a_{2c} = \frac{\sum_1^3 a_{2i}}{3}$$

$$a_{3c} = \frac{\sum_1^3 a_{3i}}{3}$$

Аппроксимирующие выражения математической модели запишем

$$y_c = a_{1c}\, x + a_{2c} z + a_{3c}$$

При увеличении числа анализируемых систем на одну систему число систем уравнений увеличивается с 3х - 12, 23, 31 до шести 12, 23, 34, 41, 24, 31, однако, используется только 4 системы уравнений 12, 23, 34, 41, учитывая наличие более двух раз одинаковых характеристик в парах.

Рассмотрим построение аппроксимирующей гиперболической ММ для трех переменных y, x, Z вида

$$y = a_1 x + \frac{a_2}{1 - a_3 z}$$

Эта модель может связывать, например, затраты на создание системы с быстродействием и объемом аппаратуры или отражать функцию, связывающую быстродействие аппаратуры с частотой работы элементов и объемом аппаратуры вычислительного устройства.

Для получения коэффициентов a1, a2, a3 необходимо иметь данные трех систем. Определение коэффициентов будем проводить в два этапа для проведения усреднений.

Первый этап проводится для определения коэффициента аппроксимации. Для этого основное уравнение математической модели представляется в виде:

$$y_i = a_{1j} x_i + A_j$$

$$A_j = \frac{a_{2j}}{1 - a_{3j} z_i}$$

Определение коэффициентов аппроксимации производится по трем системам уравнений:

$$\begin{cases} y_1 = x_1 a_{11} + A_1 \\ y_2 = x_2 a_{11} + A_1 \end{cases}$$

$$\begin{cases} y_2 = x_2 a_{12} + A_2 \\ y_3 = x_3 a_{12} + A_2 \end{cases}$$

$$\begin{cases} y_3 = x_3 a_{13} + A_3 \\ y_1 = x_1 a_{13} + A_3 \end{cases}$$

В результате решения трех систем уравнений получим три группы коэффициентов: a11, A1; a12, A2; a13, A3 и определим среднее значение коэффициента аппроксимации переменной «x».

$$a_{1c} = \frac{\sum_1^3 a_{1j}}{3}$$

Для проведения второго этапа математическую модель представим в виде:

$$A_j = \frac{a_{2j}}{1 - a_{3j} z_i}$$

Для определения коэффициентов аппроксимации используем три системы уравнений, в которых применим коэффициенты Aj, определенные на первом этапе

$$\begin{cases} A_1 = \dfrac{a_{21}}{1 - a_{31} z_1} \\ A_2 = \dfrac{a_{21}}{1 - a_{31} z_2} \end{cases}$$

$$\begin{cases} A_2 = \dfrac{a_{22}}{1 - a_{32} z_2} \\ A_3 = \dfrac{a_{22}}{1 - a_{32} z_3} \end{cases}$$

$$\begin{cases} A_3 = \dfrac{a_{23}}{1 - a_{33} z_3} \\ A_1 = \dfrac{a_{23}}{1 - a_{33} z_1} \end{cases}$$

Решением этих трех систем уравнений будут три группы коэффициентов: a21, a31; a22, a32; a23, a33.

На основании полученных коэффициентов определим их средние величины

$$a_{2c} = \frac{\sum_1^3 a_{2j}}{3}$$

$$a_{3c} = \frac{\sum_1^3 a_{3j}}{3}$$

В результате, подставив средние значения коэффициентов a_{1c}, a_{2c} и a_{3c}, получим аппроксимирующую функцию математической модели с усредненными коэффициентами вида:

$$y_c = a_{1c}x + \frac{a_{2c}}{1 - a_{3c}z}$$

Точность решения аппроксимирующего уравнения оценивается усредненной суммой отклонений характеристик относительно усредненных значений функции:

$$\delta = \frac{\sum_1^3 \Delta y_i}{3}$$

где

$$\Delta y_i = y_i - y_{ic}$$

При увеличении числа систем, используемых для аппроксимации, число систем уравнений увеличивается на одно. Полученные коэффициенты аппроксимации при решении четырех уравнений усредняются, и по ним оценивается точность.

Для ряда математических моделей работы системы используется уравнение 3-х переменных с четырьмя коэффициентами аппроксимации

$$y = x(a_1 - a_2 z) + \frac{a_3}{1 - a_4 z}$$

При определении коэффициентов аппроксимации необходимы данные четырех близких систем. Определение коэффициентов аппроксимации для их усреднения будем проводить в три этапа.

Уравнение системы для проведения первого этапа представим в виде:

$$y_i = x_i a_{ij} - B_i$$

где

$$B_i = x_i z_i a_{2j} - \frac{a_{3j}}{1 - a_{4j} z_i}$$

Для определения коэффициентов a1j и Bj составим четыре системы уравнений.

$$\begin{cases} y_1 = x_1 a_{11} + B_1 \\ y_2 = x_2 a_{11} + B_1 \end{cases}$$

$$\begin{cases} y_2 = x_2 a_{12} + B_2 \\ y_3 = x_3 a_{12} + B_2 \end{cases}$$

$$\begin{cases} y_3 = x_3 a_{13} + B_3 \\ y_4 = x_4 a_{13} + B_3 \end{cases}$$

$$\begin{cases} y_4 = x_4 a_{14} + B_4 \\ y_1 = x_1 a_{14} + B_4 \end{cases}$$

Решая эти системы уравнений, получим коэффициенты аппроксимации a11, B1; a12, B2; a13, B3; a14, B4. Величину коэффициента a1 после усреднения запишем в виде:

$$a_{1c} = \frac{\sum_1^4 a_{1j}}{4}$$

На втором этапе определим коэффициенts «a_{2j}». Для этого используем уравнение

$$B_i = a_{2j} x_i z_i - A_i$$

Составим четыре системы уравнений для определения коэффициентов a2j и Aj:

$$\begin{cases} B_1 = a_{21} x_1 z_1 - A_1 \\ B_2 = a_{21} x_2 z_2 - A_1 \end{cases}$$

$$\begin{cases} B_2 = a_{22} x_2 z_2 - A_2 \\ B_3 = a_{22} x_3 z_3 - A_2 \end{cases}$$

$$\begin{cases} B_3 = a_{23} x_3 z_3 - A_3 \\ B_4 = a_{23} x_4 z_4 - A_3 \end{cases}$$

$$\begin{cases} B_4 = a_{24} x_4 z_4 - A_4 \\ B_1 = a_{24} x_1 z_1 - A_4 \end{cases}$$

Усредненный коэффициент a2c определим

$$a_{2c} = \frac{\sum_1^4 a_{2j}}{4}$$

На третьем этапе определяются коэффициенты a3j и a4j и проводится их усреднение. Для этого этапа используется уравнение.

$$A_i = \frac{a_{3j}}{1 - a_{4j} z_i}$$

Составим четыре системы уравнений для определения коэффициентов a_{3j} и a_{4j} и их усредненных величин

$$\begin{cases} A_1 = \dfrac{a_{31}}{1 - a_{41}z_1} \\ A_2 = \dfrac{a_{31}}{1 - a_{41}z_2} \end{cases}$$

$$\begin{cases} A_2 = \dfrac{a_{32}}{1 - a_{42}z_2} \\ A_3 = \dfrac{a_{32}}{1 - a_{42}z_3} \end{cases}$$

$$\begin{cases} A_3 = \dfrac{a_{33}}{1 - a_{43}z_3} \\ A_4 = \dfrac{a_{33}}{1 - a_{43}z_4} \end{cases}$$

$$\begin{cases} A_4 = \dfrac{a_{34}}{1 - a_{44}z_4} \\ A_1 = \dfrac{a_{34}}{1 - a_{44}z_1} \end{cases}$$

Усреднение величин запишем в виде

$$a_{3c} = \frac{\sum_1^4 a_{3j}}{4}$$

$$a_{4c} = \frac{\sum_1^4 a_{4j}}{4}$$

Исходное уравнение, подставив значения коэффициентов, запишем в виде:

$$y_c = x(a_{1c} - a_{2c}z) + \frac{a_{3c}}{1 - a_{4c}z}$$

Точность аппроксимирующей зависимости

$$\delta = \frac{\sum_1^4 \Delta y_i}{4}$$

где

$$\Delta y_i = y_i - y_{ic}$$

Приведенные последовательности формирования аппроксимирующих функций на основе усредненных коэффициентов аппроксимации обеспечивают получение усредненных значений этих функций.

Для получения усредненных коэффициентов аппроксимации основное уравнение разделяют на несколько простых аппроксимирующих уравнений, использующих две координаты для последовательного вычисления усредненного коэффициента каждого уравнения. Число уравнений определяется числом членов основного уравнения.

On creation approximating functions of characteristics of small number of systems

Ya. A. Khetagurov
Concern Agat, Moscow, Russia
tshe@bk.ru

Abstract: Problem of accounting for the influence of many factors on the basic parameters of real-time computer based digital control systems at the initial design stage is solved mainly by selecting characteristics that take into account the complex area of work , system requirements and cost of material resources , characteristics of the product system, and the amount of their production. Values of these characteristics are influenced by various factors associated with the terms of building the system and its parts, with the production capacity of the enterprise, their applications, external conditions (funding, timing , the importance of the system) , especially with the experience of the leading developers and subjective conditions for the organization of the enterprise or enterprises producing system. Identifying mutual influence of these characteristics on the performance parameters of the system, parts, devices defined by above factors will be implemented by the construction of general mathematical models (MM) using the same characteristics of systems, parts, devices. Linear and hyperbolic approximating dependences will be applied to linear equations MM using performance data close to the destination and use of the systems, parts and devices.

Keywords: system characteristics, mathematical models, approximating dependencies