ИСП Институт Системного Программирования
Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 27, выпуск 5

Volume 27, issue 5

Москва 2015

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge.

Proceedings of ISP RAS is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/

Содержание

Современные методы аспектно-ориентированного анализа эмоциональной окраски
И. А. Андрианов, В. Д. Майоров, Д. Ю. Турдаков *5*

Балансировка нагрузки в системе Unihub на основе предсказания поведения пользователей
Д.А. *Грушин, Н.Н Кузюрин* *23*

Реализация сервиса для выполнения Apache Spark задач и создания Apache Spark кластеров на основе Openstack Sahara
А.В. Алексиянц, О.Д. Борисенко, Д. Ю. Турдаков, А. В. Шер, С.Д. Кузнецов *35*

Метод тестирования производительности и стресс-тестирования центральных сервисов идентификации облачных систем на примере Openstack Keystone
И.В. Богомолов, А.В. Алексиянц, А. В. Шер, О.Д. Борисенко, А.И. Аветисян *49*

Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя
В.К. Кошелев, И.А. Дудина, В.И. Игнатьев, А.И. Борзилов *59*

Метод легковесного статического анализа для поиска состояний гонок
П.С. Андрианов, В.С. Мутилин, А.В. Хорошилов *87*

Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях
М.У. Мандрыкин, В.С. Мутилин *117*

Использование языка программирования Python для описания ограничений на архитектурные модели
Е.В. Корныхин, А.В. Хорошилов *143*

Использование симуляции сбоев при тестировании компонентов ядра ОС Linux
А.В. Цыварев, А.В. Хорошилов *157*

Об интеграции формальных методов в задачах верификации операционных систем
А.К.Петренко, В.В.Кулямин, А.В.Хорошилов и др. *175*

Приближенный алгоритм для хроматической раскраски двудольных графов за полиномиальное в среднем время
А.С. Асратян, Н.Н. Кузюрин. *191*

Table of Contents

Modern Approaches to Aspect-Based Sentiment Analysis
I. Andrianov, V. Mayorov, D. Turdakov *5*

Load Balancing in Unihub SaaS System Based on User Behavior Prediction
D.A. Grushin, N.N. Kuzyurin *23*

Implementing Apache Spark jobs Execution and Apache Spark cluster creation for Openstack Sahara
A. Aleksiyants, O. Borisenko, D. Turdakov, A. Sher, S. Kuznetsov. *35*

A Performance Testing and Stress Testing of Cloud Platform Central Identity: Openstack Keystone Case Study
I. V. Bogomolov, A. Aleksiyants, A. Sher, O. Borisenko, A. Avetisyan *49*

Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference
V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov *59*

Lightweight Static Analysis for Data Race Detection in Operating System Kernels
P.S. Andrianov, V.S. Mutilin, A.V. Khoroshilov *87*

Modeling Memory with Uninterpreted Functions for Predicate Abstractions
M.U. Mandrykin, V.S. Mutilin *117*

Python-Based Constraint Language for Architecture Models
E. Kornykhin, A. Khoroshilov..... *143*

Using Fault Injection for Testing Linux Kernel Components
A.Tsyvarev, A.Khoroshilov *157*

Integration Points of Operating System Verification Techniques
A. K. Petrenko, V. V. Kuliamin, A. V. Khoroshilov. *175*

Approximating Chromatic Sum Coloring of Bipartite Graphs in Expected Polynomial Time
A.S. Asratian, N.N. Kuzyurin. *191*

Современные методы аспектно-ориентированного анализа эмоциональной окраски*

[1]И. А. Андрианов <ivan.andrianov@ispras.ru>
[1]В. Д. Майоров <vmayorov@ispras.ru>
[1,2,3]Д. Ю. Турдаков <turdakov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. Данная работа посвящена обзору методов решения актуальной на сегодняшний день задачи аспектно-ориентированного анализа эмоциональной окраски текстов. Данная задача решалась в рамках нескольких конференций, посвященных автоматическому анализу текстов на естественном языке. Организаторы конференций предлагали участникам площадки для сравнительного тестирования методов. В рамках данной работы рассмотрены методы решения задачи аспектно-ориентированного анализа эмоциональной окраски, предложенные участниками двух таких международных площадок: SemEval-2015 и SentiRuEval-2015.

Ключевые слова: анализ эмоциональной окраски; извлечение аспектных терминов; обработка текстов на естественном языке; машинное обучение

DOI: 10.15514/ISPRAS-2015-27(5)-1

Для цитирования: Андрианов И.А., Майоров В.Д., Турдаков Д.Ю. Современные методы аспектно-ориентированного анализа эмоциональной окраски. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 5-22. DOI: 10.15514/ISPRAS-2015-27(5)-1.

1. Введение

Последние годы ознаменовались бурным ростом торговых площадок в интернете. Потребители при выборе товара рассматривают не только описание его фактических характеристик, но и отзывы других потребителей.

* Работа поддержана грантом РФФИ 15-37-20375 мол_а_вед

Производители товаров также заинтересованы в получении информации об успешности продукта от потребителей.

Это приводит к появлению все большего числа площадок для размещения отзывов в интернете. Помимо собственно возможности оставить отзыв в свободной форме, подобные площадки нередко предоставляют и возможность оценить различные аспекты товара (например, качество исполнения или надежность) по какой-либо шкале.

В связи с этим приобрела актуальность задача аспектно-ориентированного анализа эмоциональной окраски. Задача предполагает анализ текстов отзывов с целью автоматического извлечения из них информации об эмоциональной окраске тех или иных аспектов товара.

Актуальность задачи привела к росту числа площадок для сравнительного тестирования подходов, организаторы которых предлагают участникам разработать системы, решающие подзадачи аспектно-ориентированного анализа эмоциональной окраски. При этом организаторы вначале предоставляют участникам тренировочные данные, а затем, непосредственно перед проведением тестирования, данные для тестирования. Примерами таких площадок могут служить: SemEval, проводимый в рамках сообщества ACL, а также SentiRuEval, проводимый в рамках конференции “Диалог”.

В рамках данной статьи рассматриваются методы решения задачи аспектно-ориентированного анализа эмоциональной окраски, предложенные участниками площадок SemEval-2015 [1] и SentiRuEval-2015 [2].

В рамках SemEval-2015 предлагалась задача аспектно-ориентированного анализа эмоциональной окраски (задача 12). Эта задача была разделена на две независимые подзадачи, первая из которых была сформулирована как аспектно-ориентированный анализ эмоциональной окраски внутри предметной области (и обучающая, и тестовая коллекции отзывов относились к одной предметной области), вторая - между предметными областями (обучающая коллекция относилась к предметным областям “автомобили” и “рестораны”, тестовая коллекция относилась к неизвестной заранее предметной области - “отели”). Первая подзадача делилась в свою очередь на секции. Одна из секций была посвящена извлечению аспектных терминов, то есть упоминаний целевых объектов и их характеристик, другая - определению полярности эмоциональной окраски для каждого аспектного термина.

В рамках SentiRuEval-2015 задача аспектно-ориентированного анализа эмоциональной окраски рассматривалась только внутри предметной области и разделялась на секции похожим образом: две подзадачи касались извлечения аспектных терминов, еще одна была посвящена определению полярности эмоциональной окраски.

2. SemEval-2015: Извлечение аспектных терминов

2.1 Описание задачи

Задача извлечения аспектных терминов предлагалась участникам SemEval-2015 как задача 12.1.2 [1]. В качестве входных данных в данной задаче выступали тексты отзывов. На выходе требовалось выявить аспектные термины и их аспекты.

Аспектными терминами (Opinion Target Expression) считались упоминания объекта, для которого написан отзыв, или его аспекта. Например, в предложении "*The pizza was delicious.*" имеется аспектный термин "*pizza*" для аспекта *FOOD#QUALITY*. Упоминания, касающиеся других объектов (например, при сравнении двух объектов) не рассматривались.

Участникам были предоставлены тренировочные данные на английском языке для одной предметной области: 254 размеченных отзыва для ресторанов. Тестовые данные, предоставленные позднее, содержали 96 размеченных отзывов для ресторанов. Балансировка данных по аспектам аспектных терминов не производилась.

2.2 Методы, предложенные участниками

Методы решения задачи извлечения аспектных терминов, предложенные участниками SemEval-2015, можно разбить на 3 категории: методы, основанные на разметке последовательности (использовались большинством участников); методы, основанные на выявлении предметно-специфичной терминологии; методы обучения без учителя.

2.2.1 Методы, основанные на разметке последовательности

Разметка последовательности - это подход, хорошо зарекомендовавший себя в задачах обработки текста, например, NERC [3]. Идея подхода во многом схожа с подходом, основанном на модели классификации, и заключается в том, чтобы построить признаковое описание предложения как последовательность признаковых описаний слов, составляющих предложение. Далее производится обучение модели путем подачи ей на вход меток слов предложений и их признаковых описаний. При обработке новых предложений модель ставит их признаковым описаниям в соответствие наиболее вероятную последовательность меток.

В рамках SemEval-2015 участники применяли как одну из наиболее популярных моделей разметки последовательности CRF [4], так и различные эмуляции разметки последовательности с помощью последовательной классификации слов часто применяемыми в обработке текстов моделями линейного SVM [5] и перцептроном [6] с применением дополнительных признаков: меток, присвоенных классификатором предыдущим словам.

Необходимо отметить, что, как и в случае задач NERC и других задач обработки текста, при решении задачи извлечения аспектных терминов с помощью разметки последовательности важную роль играет кодирование. Кодирование обеспечивает возможность различить два подряд идущих термина с одинаковыми аспектами от одного “длинного” термина. Все участники SemEval-2015, применявшие модели разметки последовательности, применяли BIO-кодирование [7], предполагающее наличие трех видов меток: *B-aspect* (начало аспектного термина для аспекта *aspect*), *I-aspect* (середина/конец аспектного термина для аспекта *aspect*), *O* (отсутствие аспектного термина).

Участниками был предложен широкий набор признаков слов, которые можно разделить на следующие категории: лексические и морфологические признаки, признаки на кластерах слов, синтаксические признаки, признаки именованных сущностей.

Лексические и морфологические признаки, использовавшиеся в том или ином виде большинством участников, включали в себя: слова [8], [9], [10]; части речи слов [10], [11], [12]; леммы слов [10], [11]; префиксы / суффиксы слов [8], [11]; наличие в слове прописных, строчных букв, а также цифр [8], [10], [11]; маски слов (замена символов слова на метки, такие как “строчная буква”, “цифра” и т.п.) [11].

Признаки на кластерах слов основывались на результатах кластеризации слов в неразмеченном корпусе текстов. В качестве корпуса участники применяли Википедию [8], а также автоматически собранные из Web отзывы для соответствующей предметной области [8], [9]. Участники применяли следующие подходы для кластеризации: Brown-кластеры [8], [9], [10], [13]; K-means-кластеры векторов word2vec [8], [9], [14], [15]; а также Clark-кластеры [8], [16].

Синтаксические признаки, использовавшиеся участниками, варьировались от простых: стоит ли слово в начале предложения [8], тип фрагмента (например, “именная фраза”) [11], в который входит слово, до более сложных, основанных на синтаксическом разборе предложений по грамматике зависимостей [17]: головное слово [9], метка входящего ребра [10], метка первого исходящего ребра [10].

Признаки именованных сущностей представляли собой BIO-разметку именованных сущностей [11].

2.2.2 Методы, основанные на выявлении предметно-специфичной терминологии

Методы, основанные на выявлении предметно-специфичной терминологии, строятся на предположении о том, что аспектные термины являются предметно-специфичными терминами [18]. Идея методов состоит в выявлении предметно-специфичных терминов каким-либо из известных методов и построении модели классификации терминов по аспектам.

В качестве модели классификации участники применяли SVM. В качестве признаков для данной модели - вхождение в соответствующие целевой предметной области (выбраны вручную) категории DBPedia [19], число гипонимов / гиперонимов по WordNet [20], число терминов, совместно встречающихся с целевым термином, в Brown-кластерах, а также в тренировочных данных, слова предложения, в которое входит термин.

2.2.3 Методы обучения без учителя

Методы обучения без учителя используют для обучения коллекции неразмеченных аспектными терминами отзывов. Важными преимуществами подобных методов являются возможность автоматического сбора большого набора отзывов для целевой предметной области из Веб и отсутствие необходимости привлекать экспертов для трудоемкой разметки отзывов.

Одним из участников был предложен метод, основанный на графах [21]. Вначале осуществлялась выборка всех существительных из тренировочных данных как кандидатов в аспектные термины, а также прилагательных как кандидатов в оценочные слова.

Далее строился граф, в котором узлами выступали выбранные слова двух типов. Ребрам в данном графе назначались веса по следующей схеме: между однотипными узлами - косинусная близость [22] между векторами word2vec, соответствующими словам-узлам; между разнотипными узлами - частота нахождения слов-узлов в тренировочных данных в одном из выбранных вручную синтаксических отношений.

Узлы-кандидаты в аспектные термины, имевшие наибольшие значения меры PageRank [23], вычисленной на построенном графе, считались аспектными терминами и размечались в тестовых данных путем поиска совпадений по леммам. Аспектом всех аспектных терминов предложения считался аспект, для которого сумма косинусных близостей векторов word2vec вручную выбранных слов и слов предложения была наибольшей.

У данного метода есть существенный недостаток: он не может находить аспектные термины, содержащие более одного слова. Для устранения данного недостатка было предложено выявить самые устойчивые словосочетания в тренировочных данных с помощью меры Log-Likelihood Ratio [24] и рассматривать их как одно слово при обработке.

2.3 Анализ результатов

Для оценки результатов организаторы использовали F1-меру. В качестве базового метода использовался метод, который собирал словарь аспектных терминов по обучающей коллекции, и искал в тестовой коллекции аспектные термины по полному совпадению со словарными.

Базовый метод получил оценку 48,06%, методы участников - оценки от 33,86% до 70,05%. Лучшие результаты показали методы на основе разметки

последовательности, большинство из них превзошло результаты базового метода. Лучшие результаты среди методов на основе разметки последовательности показали методы, использовавшие признаки на кластерах слов. Важными для показа высоких результатов оказались также лексические и морфологические признаки. Методы на основе выявления предметно-специфичной терминологии и методы обучения без учителя показали результаты, схожие с результатами базового метода: от 45,67% до 49,97%.

3. SentiRuEval-2015: Извлечение аспектных терминов

3.1 Описание задачи

Задача извлечения аспектных терминов предлагалась участникам SentiRuEval-2015 как задача A (явные) и задача B (явные, неявные и факты) [2]. В качестве входных данных в данной задаче выступали тексты отзывов. На выходе требовалось выявить аспектные термины и их аспекты.

Явными аспектными терминами назывались термины, содержащие упоминание объекта, для которого написан отзыв, или его аспекта. Неявными аспектными терминами назывались термины, включающие оценочные слова, специфичные для какого-то аспекта: например, “*комфортный*” представляет позитивную оценку аспекта *интерьер*. Эмоциональными фактами назывались термины, которые представляют объективную информацию, но неявно выражают мнение автора отзыва: например, “*была вежлива*” представляет позитивную оценку аспекта *обслуживание*.

Участникам были предоставлены тренировочные коллекции на русском языке для двух предметных областей: 201 размеченный отзыв для ресторанов, 217 - для автомобилей. Тестовые данные, предоставленные позднее, содержали 203 отзыва и 201 соответственно. Балансировка данных по аспектам и типам (явные, неявные, факты) аспектных терминов не производилась. Число явных аспектных терминов превышало число неявных/фактов в 4 - 6 раз.

3.2 Методы, предложенные участниками

Участники SentiRuEval-2015 так же, как и участники SemEval-2015, в основном использовали подходы, основанные на разметке последовательности. Так, Рубцова Ю. В. и Кошельников С. А. [25] предложили применять для решения задачи извлечения аспектных терминов CRF с использованием лексических и морфологических признаков (словоформы предыдущего и следующего слова, часть речи и лемма слова). Для задачи A было предложено обучать классификатор на трех классах (B, I, O), а для задачи B, в которой необходимо определять аспектные термины трех типов, было предложено использовать использовать 7 классов (B и I для каждого из трех типов аспектных терминов и класс O).

Майоров В. Д. и др. [26] предложили использовать линейный SVM с дополнительными признаками - метками классов предыдущих слов. Также ими был предложен намного более обширный список использованных признаков классификации, в том числе лексические, морфологические, синтаксические признаки, признаки, кластеры слов (кластеры векторных представлений слов, тематическое моделирование) и признаки, использующиеся в задаче извлечения терминологии [27]. В отличие от решения, предложенного Рубцовой, авторы метода предлагают использовать оригинальный метод (решение задачи A) независимо для каждого из типов аспектов для решения задачи B.

Блинов П. Д. и Котельников Е. В. [28] предложили альтернативный метод решения задачи. Идея метода заключается в использовании семантической близости между словами для определения аспектных терминов. Каждому слову из тестовой выборки ставился в соответствие вектор word2vec и вычислялась косинусная мера близости к словам, которые вошли в состав аспектных терминов в обучающей выборке. Если результат оказывался выше экспериментально определенного порога, то слово помечалось аспектным термином. Для определения многословных аспектных терминов авторами был предложен набор правил для объединения рядом стоящих аспектных терминов, например, найденные однословные аспектные термины объединялись в один многословный, если были разделены предлогом.

Также был предложен метод, основанный на использовании рекуррентных нейронных сетей [29]. Авторами метода были предложены различные варианты решения задачи A: рекуррентная нейронная сеть Эльмана [30], двунаправленная рекуррентная нейронная сеть [31], использовалась длинная короткая память [32]. Для преобразования входных слов в сигналы применяется языковая модель рекуррентных нейронных сетей [33]. Обучение нейронных сетей всех типов проводится с использованием метода обратного распространения ошибки [34]. При обучении, в исходных данных все числа были заменены на специальный маркер. Дополнительно было предложено добавлять во входные данные небольшой Гауссовский шум. Лучшие результаты, согласно данным авторов статьи, показал метод, основанный на двунаправленных рекуррентных нейронных сетях с использованием длинной короткой памяти.

3.3 Анализ результатов

При оценке результатов организаторы использовали F1-меру с макро-усреднением 2 видов: для полного и для частичного совпадения аспектных терминов. В качестве базового метода использовался метод, который размечал как аспектный термин для заданного аспекта слова и словосочетания, которые выступали в качестве аспектных терминов для данного аспекта в тренировочных данных.

Предложенные участниками решения задачи A не смогли показать результаты, значительно превышающие показатели базового метода. При вычислении качества по точному совпадению аспектных терминов лучшим решением оказался метод, основанный на разметке последовательности слов с использованием SVM. Он получил для предметных областей “рестораны” и “автомобили” оценки 63.1% и 67.6% соответственно, в то время как базовый метод получил оценки 60.8% и 59.4%. При учете частичного совпадения аспектных терминов для предметной области “рестораны” лучшими методами оказались подход на основе семантической близости слов (72.8%) и метод на основе рекуррентных нейронных сетей (71.9%). Базовый метод получил оценку 66.5%. Для предметной области “автомобили” лучшие результаты показали метод на основе рекуррентных нейронных сетей (74.8%) и метод, использующий SVM с большим числом лексических, морфологических, синтаксических и семантических признаков (73.0%). Базовый метод получил оценку 69.6%.

Лучшие решения задачи B также незначительно превосходят показатели базового метода. Для обеих предметных областей (“рестораны” и “автомобили”) по точному совпадению аспектных терминов только два метода смогли преодолеть базовый (58.7% и 58.9%): метод на основе нейронных сетей (60.0%, 63.0%) и метод на основе SVM (59.6% и 63.6%). По частичному совпадению аспектных терминов базовый метод (61.9% и 67.4%) смог превзойти лишь метод на основе рекуррентных нейронных сетей (66.8% и 71.5%).

4. SemEval-2015: Анализ полярности аспектных терминов

4.1 Описание задачи

Задача аспектно-ориентированного анализа полярности предложений предлагалась участникам SemEval-2015 как задача 12.1.3 (для известных заранее предметных областей) и задача 12.2 (для неизвестной заранее предметной области) [1]. В качестве входных данных в данной задаче выступали тексты отзывов, у которых каждое предложение было размечено множеством аспектов и аспектных терминов, в отношении которых в предложении выражаются мнения. На выходе требовалось выявить полярность для каждого из указанных аспектных терминов.

Рассматривалось 3 класса полярности: позитивная, негативная и нейтральная. Одновременное наличие как позитивной, так и негативной полярности рассматривалось как нейтральная.

Участникам были предоставлены тренировочные данные на английском языке для двух предметных областей: 254 размеченных отзыва для ресторанов, 277 - для ноутбуков. Тестовые данные, предоставленные позднее, содержали 96 размеченных отзывов для ресторанов, 173 - для ноутбуков, а также 30 - для неизвестной заранее предметной области (отелей). Балансировка данных по

классам полярности не производилась. Позитивный класс был мажорным с относительной частотой вхождений от 53% до 73% в зависимости от предметной области. Нейтральный класс был самым малочисленным с относительной частотой вхождений от 3% до 9%.

4.2 Методы, предложенные участниками

Методы решения задачи аспектно-ориентированного анализа полярности предложений, предложенные участниками SemEval-2015, можно разбить на 2 категории: методы, основанные на классификации; методы обучения без учителя.

4.2.1 Методы, основанные на классификации

Идея методов, основанных на классификации, состоит в выборе контекста аспектного термина и построении модели классификации выбранного контекста по полярности.

В простейшем случае участники выбирали все предложение, в которое входит аспектный термин, в качестве контекста [10], [18]. Такой подход обладает серьезным недостатком: все аспектные термины в предложении получают одинаковую полярность, что в общем случае неверно. Например, в предложении "*I like the somosas, chai, and the chole, but the dhosas and dhal were kinda dissapointing.*" термины "*somosas*" и "*dhosas*" имеют положительную и отрицательную полярности соответственно. Для устранения данного недостатка некоторые участники применяли разбиение предложений по пунктуации [11]. Также применялся и типовой для задач обработки текста способ выбора контекста: окно слов слева-справа [8] и его модификация, использующая не позиции в тексте, а позиции в синтаксическом дереве [35], [36]. Кроме того, применялся подход, выбирающий из предложения и соседних с ним предложений слова с наивысшими весами, например, tf-idf [36], [37].

В качестве моделей классификации участники применяли линейный SVM, логистическую регрессию [38]. Признаки для моделей классификации, предложенные участниками можно разделить на следующие категории: лексические, морфологические и синтаксические признаки, признаки на основе внешних источников данных.

Лексические, морфологические и синтаксические признаки включают в себя: униграммы [10], [11], [18], [35], [36]; биграммы [11], [36]; части речи [8], [35], [36]; леммы [8]; наличие повторяющихся букв [10] и восклицательных знаков [10], [36]; вхождение слова в список вопросительных или условных слов [10]; метки входящих ребер синтаксического дерева [35], [36].

Наиболее часто в качестве внешнего источника данных участники задействовали как составленные вручную, так и собранные автоматически, лексиконы оценочных слов: Bing Liu [8], [11], [18], [35], [36]; General Inquirer

[8], [18], [36]; SentiWordNet [8], [11], [35], [36]; MPQA [11], [35], [36]; Sentiment140 [11], [36]; NRC Hashtag [11], [36]. В качестве признаков на основе лексиконов выступали: полярности, суммы полярностей, максимумы / минимумы полярностей, относительное число позитивных / негативных полярностей. Некоторые участники инвертировали полярность слова, если рядом с ним встречалось слово-отрицание [11], [18], [35], [36]. Помимо лексиконов оценочных слов участники применяли признаки на основе WordNet [10] и Brown-кластеров [11].

4.2.2 Методы обучения без учителя

Одним из участников был предложен следующий метод обучения без учителя [21]. Были вручную выбраны слова, имеющие заданную полярность в любой предметной области: например, “*excellent*”. Далее из тренировочных данных были извлечены слова, имеющие самые высокие значения косинусной близости векторов word2vec с выбранными вручную словами. Нахождение подобных слов рядом с аспектным термином считалось признаком соответствующей полярности аспектного термина. Кроме того, во внимание принимались и слова-отрицания, их наличие рядом с аспектным термином инвертировало его полярность.

4.3 Анализ результатов

При оценке результатов организаторы использовали достоверность, т.е. число аспектов с верно определенной полярностью к общему числу аспектов в предложениях тестовых данных. В качестве базового метода использовался линейный SVM с признаками “мешок слов” и “номер аспекта” (всем аспектам были предоставлены уникальные номера).

Для предметной области ресторанов базовый метод получил оценку 63,55%, методы участников - оценки от 60,71% до 78,69%. Для предметной области ноутбуков базовый метод получил оценку 69,96%, методы участников - оценки от 51,84% до 79,34%.

Лучшими методами на основе классификации стали методы, которые использовали различные оценки слов контекста аспектного термина на основе лексиконов оценочных слов, а также учитывали слова-отрицания. Лучшими подходами к выбору слов контекста аспектного термина стали выбор слов с наивысшим tf-idf в текущем и соседних предложениях, а также разбиение предложения по пунктуации. Методы обучения без учителя показали результаты, близкие к результатам базового метода: от 68,38% до 69,46% в зависимости от предметной области.

Необходимо отметить, что практически все методы, включая базовый, показали для предметной области “ноутбуки” более высокий результат, нежели для предметной области “рестораны”. Это можно объяснить тем, что тестовые данные для предметной области “рестораны” содержали 53,72% аспектных терминов с позитивной полярностью, в то время как

тренировочные – 72,43%. Для предметной области “ноутбуки” же данные показатели – 57% и 55,87% соответственно. Однако чем выше был результат метода, тем меньшей становилась разница результатов для двух предметных областей. Для лучших методов разница результатов была менее 1%, для базового – более 6%.

5. SentiRuEval-2015: Анализ полярности аспектных терминов

5.1 Описание задачи

Задача анализа полярности аспектных терминов предлагалась участникам SentiRuEval-2015 как задача C [2]. В качестве входных данных в данной задаче выступали тексты отзывов, размеченные аспектными терминами и их аспектами. На выходе требовалось выявить полярность аспектных терминов.

Рассматривалось 4 класса полярности: позитивная, негативная, нейтральная, обе. Нейтральная использовалась также и в случае отсутствия эмоциональной окраски: например, в предложении “*Так-же при покупке авто стоит сразу-же надеть чехлы на сиденья так как обивка сидений тоже бюджетная.*”, где мнение выражается в отношении аспектного термина “*обивка сидений*”, но не термина “*чехлы на сиденья*”.

Участникам были предоставлены тренировочные коллекции на русском языке для двух предметных областей: 201 размеченный отзыв для ресторанов, 217 - для автомобилей. Тестовые данные, предоставленные позднее, содержали 203 отзыва и 201 соответственно. Балансировка данных по классам полярности не производилась. Позитивная полярность была мажорным классом для всех предметных областей, вслед за ней располагались негативная и нейтральная, класс “обе” был крайне редким (не более 115 терминов в каждом наборе данных). Отношение числа терминов в позитивном классе к их числу в негативном/нейтральном располагалось в интервале от 3 до 6.

5.2 Методы, предложенные участниками

Участниками были предложены методы на основе word2vec [28] и рекуррентных нейронных сетей [29].

Рассмотрим подробнее метод на основе word2vec, предложенный одним из участников. Вначале были выбраны эталонные слова положительной / отрицательной полярностей для каждой из предметных областей. Затем для каждого прилагательного и глагола из обучающих данных была подсчитана сумма косинусных близостей векторов word2vec данного слова и слов-эталонов. В качестве полярности слова выбиралась та, для которой такая сумма была наибольшей. В случае если перед словом стояло слово отрицание, полярность инвертировалась.

Далее были использованы дополнительные коллекции отзывов соответствующих предметных областей, собранные участником, с оценками по десятибалльной шкале. Коллекции были разбиты по оценкам на резко положительные и резко отрицательные отзывы, на каждом фрагменте была вычислена мера PMI [39] для каждого прилагательного / глагола из обучающих данных.

Вычисленные с помощью word2vec полярности и значения PMI всех прилагательных и глаголов, находящихся рядом с аспектным термином, использовались как признаки для модели классификации Gradient Boosting [40].

Полярность “оба” выбиралась для аспектных терминов, рядом с которыми в рамках предложения находилось слово “но”.

5.3 Анализ результатов

При оценке результатов организаторы использовали F1-меры с макро- и микро-усреднением. В качестве базового метода использовался метод, который назначал аспектным терминам наиболее распространенный класс полярности для их аспекта в тренировочных данных (это был “позитивно” во всех случаях).

По F1-мере с макро-усреднением базовый метод получил оценку 26,7%, лучший из 7 методов участников (Gradient Boosting) - оценку 55,4%. Остальные методы участников либо не смогли превысить оценку базового метода, либо превысили ее незначительно. Схожая картина наблюдалась и в оценках по F1-мере с микро-усреднением: базовый метод получил оценку 71,0%, лучший из методов участников (также Gradient Boosting) - 82,4%.

6. Заключение

Участники международных площадок для сравнительного тестирования методов аспектно-ориентированного анализа эмоциональной окраски в основном предлагали методы, основанные на машинном обучении с учителем. Однако результаты тестирования показали незначительное улучшение качества по сравнению с простыми базовыми методами решения этой задачи. Таким образом, в результате обзора методов, предложенных в рамках SemEval-2015 и SentiRuEval-2015, можно сделать вывод, что задачи автоматического извлечения аспектных терминов и определения полярности эмоциональной окраски все еще далеки от окончательного решения.

Организаторы SemEval-2015 предложили следующий этап сравнительного тестирования в рамках площадки SemEval-2016[*], где рассматривается та же задача, но расширен перечень предметных областей, и существенно расширен список целевых языков.

[*] http://alt.qcri.org/semeval2016/task5/

Список литературы

[1]. Pontiki M., Galanis D., Papageorgiou H., Manandhar S., Androutsopoulos I. SemEval-2015 Task 12: Aspect Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 486–495.

[2]. Loukachevitch N., Blinov P., Kotelnikov E., Rubtsova Y., Ivanov V., Tutubalina E. SentiRuEval: Testing Object‑oriented Sentiment Analysis Systems in Russian. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 12–24.

[3]. Ratinov L., Roth D. Design challenges and misconceptions in named entity recognition. Proceedings of the Thirteenth Conference on Computational Natural Language Learning, 2009, pp. 147–155.

[4]. Lafferty J., McCallum A., Pereira F. Conditional random fields: probabilistic models for segmenting and labeling sequence data. Proceedings of the Eighteenth International Conference on Machine Learning, 2001, pp. 282–289.

[5]. Vapnik V. Statistical Learning Theory. Wiley, New York, NY, 1998.

[6]. Collins M. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. Proceedings of the ACL-02 conference on Empirical methods in natural language processing, 2002, volume 10, pp. 1–8.

[7]. Ramshaw L. A., Marcus M. P. Text chunking using transformation-based learning. Natural language processing using very large corpora, Springer Netherlands, 1999, pp. 157-176.

[8]. San Vicente I., Saralegi X., Agerri R. EliXa: A modular and flexible ABSA platform. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 748–752.

[9]. Toh Z., Su J. NLANGP: Supervised Machine Learning System for Aspect Category Classification and Opinion Target Extraction. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 496–501.

[10]. Guha S., Joshi A., Varma V. SIEL: Aspect Based Sentiment Analysis in Reviews. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 759–766.

[11]. Hamdan H., Bellot P., Bechet F. Lsislif: CRF and Logistic Regression for Opinion Target Extraction and Sentiment Polarity Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 753–758.

[12]. Koppula A., Pallelra R., Repaka R., Movva V. UMDuluth-CS8761-12: A Novel Machine Learning Approach for Aspect Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 742–747.

[13]. Brown P. F., Desouza P. V., Mercer R. L., Pietra V. J. D., Lai J. C. Class-based n-gram models of natural language. Computational linguistics, 1992, volume 18, issue 4, pp. 467–479.

[14]. Hartigan J. A., Wong M. A. Algorithm AS 136: A k-means clustering algorithm. Applied statistics, 1979, pp. 100-108.

[15]. Mikolov T., Yih W., Zweig G. Linguistic Regularities in Continuous Space Word Representations. Proceedings of HLT-NAACL, 2013, pp. 746-751.

[16]. Clark A. Combining distributional and morphological information for part of speech induction. Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics, 2003, volume 1, pp. 59–66.

[17]. Eisner J. M. Three new probabilistic models for dependency parsing: An exploration. Proceedings of the 16th conference on Computational linguistics, 1996, volume 1, pp. 340-345.

[18]. De Clercq O., Van de Kauter M., Lefever E. and Hoste V. LT3: Applying Hybrid Terminology Extraction to Aspect-Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 719–724.

[19]. Auer S., Bizer C., Kobilarov G., Lehmann J., Cyganiak R., Ives Z. Dbpedia: A nucleus for a web of open data. Springer Berlin Heidelberg, 2007, pp. 722-735.

[20]. Miller G. A. WordNet: a lexical database for English. Communications of the ACM, 1995, volume 38, issue 11, pp. 39-41.

[21]. Garcia-Pablos A., Cuadros M., Rigau G. V3: Unsupervised Aspect Based Sentiment Analysis for SemEval-2015 Task 12. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 714–718.

[22]. Yates R. B., Neto B. R. Modern Information Retrieval. ADDISON-WESLEY, New York, 1999, pp. 501.

[23]. Page L., Brin S., Motwani R., Winograd T. The PageRank Citation Ranking: Bringing Order to the Web. Stanford InfoLab, 1999, pp. 17.

[24]. Owen A. B. Empirical likelihood ratio confidence intervals for a single functional. Biometrika, 1988, volume 75, issue 2, pp. 237-249.

[25]. Rubtsova Y. V., Koshelnikov S. A. Aspect Extraction Using Conditional Random Fields. http://www.dialog-21.ru/digests/dialog2015/materials/pdf/RubtsovaYVKoshelnikovSA.pdf, 2015.

[26]. Mayorov V., Andrianov I., Astrakhantsev N., Avanesov V., Kozlov I., Turdakov D. A High Precision Method for Aspect Extraction in Russian. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 58–67.

[27]. Астраханцев Н. А. Автоматическое извлечение терминов из коллекции текстов предметной области с помощью Википедии. Труды ИСП РАН, 2014, том 26, выпуск 4, с. 7–20.

[28]. Blinov P. D., Kotelnikov E. V. Semantic Similarity for Aspect-Based Sentiment Analysis. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 36–45.

[29]. Tarasov D. S. Deep Recurrent Neural Networks for Multiple Language Aspect-based Sentiment Analysis of User Reviews. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 77–88.

[30]. Elman J. Finding structure in time. Cognitive science, 1990, volume 14(2), pp. 179–211.

[31]. Schuster M., Kuldip K. P. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 1997, volume 45(11), pp. 2673–2681.

[32]. Hochreiter S., Schmidhuber J.. Long short-term memory. Neural computation, 1997, volume 9(8), pp. 1735 –1780.

[33]. Mikolov T., Karafiat M., Burget L., Cernocky J., Khudanpur S. Recurrent neural network based language model. In INTERSPEECH, 2010, pp. 1045–1048.

[34]. Werbos P. J. Backpropagation through time: what it does and how to do it. Proceedings of the IEEE, 1990, volume 78(10), 1550–1560.

[35]. Jimenez-Zafra S., Martınez-Camara E., Martın-Valdivia M., Urena-Lopez L. SINAI: Syntactic approach for Aspect Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 730–735.

[36]. Zhang Z., Lan M. ECNU: Extracting Effective Features from Multiple Sequential Sentences for Target-dependent Sentiment Analysis in Reviews. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 736–741.
[37]. Ramos J. Using tf-idf to determine word relevance in document queries. Proceedings of the first instructional conference on machine learning, 2003, pp. 45–65.
[38]. Press S. J., Wilson S. Choosing between logistic regression and discriminant analysis. Journal of the American Statistical Association, 1978, volume 73, issue 364, pp. 699-705.
[39]. Church K. W., Hanks P. Word association norms, mutual information, and lexicography. Computational linguistics, 1990, volume 16, issue 1, pp. 22-29.
[40]. Friedman J. H. Stochastic gradient boosting. Computational Statistics & Data Analysis, 2002, volume 38, issue 4, pp. 367-378.

Modern Approaches to Aspect-Based Sentiment Analysis

[1]I. Andrianov <ivan.andrianov@ispras.ru>
[1]V. Mayorov <vmayorov@ispras.ru>
[1,2,3]D. Turdakov <turdakov@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper presents a survey of methods solving the actual task of aspect-based sentiment analysis. Solutions for this task were proposed at multiple natural language processing conferences. Organizers of these conferences proposed evaluation platforms for methods for aspect-based sentiment analysis. This paper describes methods proposed by participants of two international evaluation platforms: SemEval-2015 focusing on English texts and SentiRuEval-2015 focusing on Russian texts.

SemEval-2015 organizers provided participants with the task 12.2 for English language and two domains: restaurants and laptops. The task was split to multiple subtasks two of which are described in this paper: opinion target expression (both explicit and implicit ones) extraction and sentiment polarity detection. Described methods for opinion target expression can be split to the following categories: sequence labeling, domain-specific terminology extraction and unsupervised learning. Methods for sentiment polarity detection varied from classification-based to unsupervised learning.

SentiRuEval-2015 organizers provided participants with the tasks A, B and C for Russian language and two domains: restaurants and automobiles. Task A was devoted to explicit aspect term extraction, task B – to explicit, implicit and factual aspect term extraction. Sentiment polarity detection was subject of the Task C. Described methods for aspect term

extraction can be classified as following: sequence labeling, word2vec-based and neural network-based. Methods for sentiment polarity detection varied from word2vec-based to neural network-based.

Keywords: sentiment analysis; aspect extraction; text processing; machine learning

DOI: 10.15514/ISPRAS-2015-27(5)-1

For citation: Andrianov I., Mayorov V., Turdakov D. Modern Approaches to Aspect-Based Sentiment Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 5-22 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-1.

References

[1]. Pontiki M., Galanis D., Papageorgiou H., Manandhar S., Androutsopoulos I. SemEval-2015 Task 12: Aspect Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 486–495.

[2]. Loukachevitch N., Blinov P., Kotelnikov E., Rubtsova Y., Ivanov V., Tutubalina E. SentiRuEval: Testing Object‑oriented Sentiment Analysis Systems in Russian. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 12–24.

[3]. Ratinov L., Roth D. Design challenges and misconceptions in named entity recognition. Proceedings of the Thirteenth Conference on Computational Natural Language Learning, 2009, pp. 147–155.

[4]. Lafferty J., McCallum A., Pereira F. Conditional random fields: probabilistic models for segmenting and labeling sequence data. Proceedings of the Eighteenth International Conference on Machine Learning, 2001, pp. 282–289.

[5]. Vapnik V. Statistical Learning Theory. Wiley, New York, NY, 1998.

[6]. Collins M. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. Proceedings of the ACL-02 conference on Empirical methods in natural language processing, 2002, volume 10, pp. 1–8.

[7]. Ramshaw L. A., Marcus M. P. Text chunking using transformation-based learning. Natural language processing using very large corpora, Springer Netherlands, 1999, pp. 157-176.

[8]. San Vicente I., Saralegi X., Agerri R. EliXa: A modular and flexible ABSA platform. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 748–752.

[9]. Toh Z., Su J. NLANGP: Supervised Machine Learning System for Aspect Category Classification and Opinion Target Extraction. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 496–501.

[10]. Guha S., Joshi A., Varma V. SIEL: Aspect Based Sentiment Analysis in Reviews. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 759–766.

[11]. Hamdan H., Bellot P., Bechet F. Lsislif: CRF and Logistic Regression for Opinion Target Extraction and Sentiment Polarity Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 753–758.

[12]. Koppula A., Pallelra R., Repaka R., Movva V. UMDuluth-CS8761-12: A Novel Machine Learning Approach for Aspect Based Sentiment Analysis. Proceedings of the

9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 742–747.

[13]. Brown P. F., Desouza P. V., Mercer R. L., Pietra V. J. D., Lai J. C. Class-based n-gram models of natural language. Computational linguistics, 1992, volume 18, issue 4, pp. 467–479.

[14]. Hartigan J. A., Wong M. A. Algorithm AS 136: A k-means clustering algorithm. Applied statistics, 1979, pp. 100-108.

[15]. Mikolov T., Yih W., Zweig G. Linguistic Regularities in Continuous Space Word Representations. Proceedings of HLT-NAACL, 2013, pp. 746-751.

[16]. Clark A. Combining distributional and morphological information for part of speech induction. Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics, 2003, volume 1, pp. 59–66.

[17]. Eisner J. M. Three new probabilistic models for dependency parsing: An exploration. Proceedings of the 16th conference on Computational linguistics, 1996, volume 1, pp. 340-345.

[18]. De Clercq O., Van de Kauter M., Lefever E. and Hoste V. LT3: Applying Hybrid Terminology Extraction to Aspect-Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 719–724.

[19]. Auer S., Bizer C., Kobilarov G., Lehmann J., Cyganiak R., Ives Z. Dbpedia: A nucleus for a web of open data. Springer Berlin Heidelberg, 2007, pp. 722-735.

[20]. Miller G. A. WordNet: a lexical database for English. Communications of the ACM, 1995, volume 38, issue 11, pp. 39-41.

[21]. Garcia-Pablos A., Cuadros M., Rigau G. V3: Unsupervised Aspect Based Sentiment Analysis for SemEval-2015 Task 12. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 714–718.

[22]. Yates R. B., Neto B. R. Modern Information Retrieval. ADDISON-WESLEY, New York, 1999, pp. 501.

[23]. Page L., Brin S., Motwani R., Winograd T. The PageRank Citation Ranking: Bringing Order to the Web. Stanford InfoLab, 1999, pp. 17.

[24]. Owen A. B. Empirical likelihood ratio confidence intervals for a single functional. Biometrika, 1988, volume 75, issue 2, pp. 237-249.

[25]. Rubtsova Y. V., Koshelnikov S. A. Aspect Extraction Using Conditional Random Fields. http://www.dialog-21.ru/digests/dialog2015/materials/pdf/RubtsovaYVKoshelnikovSA.pdf, 2015.

[26]. Mayorov V., Andrianov I., Astrakhantsev N., Avanesov V., Kozlov I., Turdakov D. A High Precision Method for Aspect Extraction in Russian. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 58–67.

[27]. Astrakhantsev N. Avtomaticheskoe izvlechenie terminov iz kollektsii tekstov predmetnoi oblasti s pomoshch'yu Vikipedii [Automatic term acquisition from domain-specific text collection by using Wikipedia]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, volume 26, issue 4, pp. 7–20.

[28]. Blinov P. D., Kotelnikov E. V. Semantic Similarity for Aspect-Based Sentiment Analysis. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 36–45.

[29]. Tarasov D. S. Deep Recurrent Neural Networks for Multiple Language Aspect-based Sentiment Analysis of User Reviews. Proceedings of the 21st International Conference on Computational Linguistics Dialog-2015, 2015, volume 2, pp. 77–88.

[30]. Elman J. Finding structure in time. Cognitive science, 1990, volume 14(2), pp. 179–211.

[31]. Schuster M., Kuldip K. P. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 1997, volume 45(11), pp. 2673–2681.
[32]. Hochreiter S., Schmidhuber J.. Long short-term memory. Neural computation, 1997, volume 9(8), pp. 1735 –1780.
[33]. Mikolov T., Karafiat M., Burget L., Cernocky J., Khudanpur S. Recurrent neural network based language model. In INTERSPEECH, 2010, pp. 1045–1048.
[34]. Werbos P. J. Backpropagation through time: what it does and how to do it. Proceedings of the IEEE, 1990, volume 78(10), 1550–1560.
[35]. Jimenez-Zafra S., Martınez-Camara E., Martın-Valdivia M., Urena-Lopez L. SINAI: Syntactic approach for Aspect Based Sentiment Analysis. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 730–735.
[36]. Zhang Z., Lan M. ECNU: Extracting Effective Features from Multiple Sequential Sentences for Target-dependent Sentiment Analysis in Reviews. Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval-2015), 2015, pp. 736–741.
[37]. Ramos J. Using tf-idf to determine word relevance in document queries. Proceedings of the first instructional conference on machine learning, 2003, pp. 45–65.
[38]. Press S. J., Wilson S. Choosing between logistic regression and discriminant analysis. Journal of the American Statistical Association, 1978, volume 73, issue 364, pp. 699-705.
[39]. Church K. W., Hanks P. Word association norms, mutual information, and lexicography. Computational linguistics, 1990, volume 16, issue 1, pp. 22-29.
[40]. Friedman J. H. Stochastic gradient boosting. Computational Statistics & Data Analysis, 2002, volume 38, issue 4, pp. 367-378.

Балансировка нагрузки в системе Unihub на основе предсказания поведения пользователей.

Д.А. Грушин <grusin@ispras.ru>
Н.Н Кузюрин <nnkuz@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Разработанная в ИСП РАН программная система Unihub является облачной вычислительной системой типа SaaS и предоставляет пользователям возможность удаленной работы через Web-браузер с интерактивными графическими Linux-приложениями, запускаемыми в изолированных Docker контейнерах. Контейнеры имеют динамические требования к вычислительным ресурсам. Обычный способ размещения, при котором контейнеры распределяются равномерно по всем имеющимся в распоряжении серверам, приводит к неравномерной загрузке системы: некоторые сервера перегружены, а некоторые простаивают. В данной работе мы предлагаем собирать данные о поведении пользователей и характере работы различных приложений с целью прогнозирования создаваемой контейнерами нагрузки. Наши наблюдения показывают, что полученная информация позволяет равномернее загрузить имеющееся в распоряжении оборудование и увеличить предельную производительность системы в целом**.**

Ключевые слова: облачные системы типа SaaS. система Unihub. прогнозирование поведения пользователей

DOI: 10.15514/ISPRAS-2015-27(5)-2

Для цитирования: Грушин Д.А., Кузюрин Н.Н. Балансировка нагрузки в системе Unihub на основе предсказания поведения пользователей. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 23-34. DOI: 10.15514/ISPRAS-2015-27(5)-2.

1. Введение

Одной из основных тенденций развития информационных технологий в настоящее время является массовое внедрение "облачных" вычислений. Важнейшим преимуществом для пользователей облачной инфраструктуры является возможность отказаться от необходимости самостоятельно содержать дорогостоящее вычислительное оборудование. За счет использования технологий виртуализации, одним набором физического

оборудования, облако способно удовлетворить различные требования большого числа пользователей. Облако также дает возможность "масштабирования с помощью кредитной карты" (ориг. "scale by credit card") – что означает незамедлительное предоставление по требованию дополнительных ресурсов на некоторое время за плату. При этом, потребности пользователя ограничиваются только его финансовыми возможностями, в отличие от физических ограничений при добавлении узлов в кластер или накладных расходов на содержание неиспользуемых ресурсов. Системы управления облачными инфраструктурами дают новые возможности для администраторов, так как включают в себя механизмы для мониторинга и управления огромными массивами физических серверов. Различают несколько моделей облачных вычислений:

1. Программное обеспечение как услуга (SaaS, англ. Software-as-a-Service) — модель, в которой пользователю предоставляется возможность использования программного обеспечения, работающего в облачной инфраструктуре и доступного из различных клиентских устройств, например, из браузера.
2. Платформа как услуга (PaaS, англ. Platform-as-a-Service) — модель, когда пользователь использует облачную инфраструктуру для размещения базового программного обеспечения для последующего размещения на нем новых или существующих приложений. В состав таких платформ входят инструментальные средства создания, тестирования и выполнения прикладного программного обеспечения — системы управления базами данных, среды исполнения языков программирования и др., предоставляемые облачным провайдером.
3. Инфраструктура как услуга (IaaS, англ. IaaS or Infrastructure-as-a-Service)предоставляется как возможность использования облачной инфраструктуры для самостоятельного управления ресурсами обработки, хранения, сетями и другими фундаментальными вычислительными ресурсами. Например, потребитель может устанавливать и запускать произвольное программное обеспечение, которое может включать в себя операционные системы и прикладное программное обеспечение. Потребитель может контролировать операционные системы, виртуальные системы хранения данных и установленные приложения, а также набор доступных сервисов (например, межсетевой экран, DNS).

Разработанная в ИСП РАН программная система Unihub [10] является облачной вычислительной системой типа SaaS и предоставляет пользователям возможность удаленной работы через Web-браузер с интерактивными графическими Linux-приложениями.

Популярность подобных систем, предоставляющих доступ к удаленным рабочим столам посредством тонкого клиента, в настоящее время сильно выросла. Это произошло благодаря повсеместному внедрению облачных

вычислений и доступности скоростного Интернет доступа [3]. При выполнении приложений в виртуальных рабочих столах на удаленных серверах, пользователи получают доступ к любому приложению из любого места и с любого устройства. Такая модель использования вычислительных ресурсов создает новую задачу оптимизации распределения нагрузки.

Этому направлению в настоящее время уделяется большое внимание (см. например [13-15]). В системе Unihub все задачи интерактивные, с динамическими требованиями к ресурсам. При появлении задачи планировщик не может знать когда она завершится и сколько ресурсов она будет потреблять в определенный момент времени. Если поместить несколько столов на один сервер, и приложения при этом начнут активно использовать процессор, то пользователи сразу заметят замедление работы, что нежелательно. Эта ситуация в точности соответствует работе нескольких приложений на одном компьютере - если вы запускаете много тяжеловесных приложений, то процессор делится между ними, и все приложения работают заметно медленнее.

Для оптимизации размещения рабочих столов планировщику необходима дополнительная информация. Источником такой информации может служить, например, история загрузки серверов за последнее время, количество запущенных приложений на каждом сервере, и т.д. Мы предлагаем собирать и хранить информацию о загрузке серверов вместе со статистикой запусков задачи и поведения пользователей в системе для оптимизации принятия решений планировщиком.

2. Управление вычислительными ресурсами в системе Unihub

Unihub запускает приложения в Docker контейнерах [11], которые могут размещаться в виртуальных машинах или на серверах (далее будем использовать термин сервер для обозначения обоих случаев).

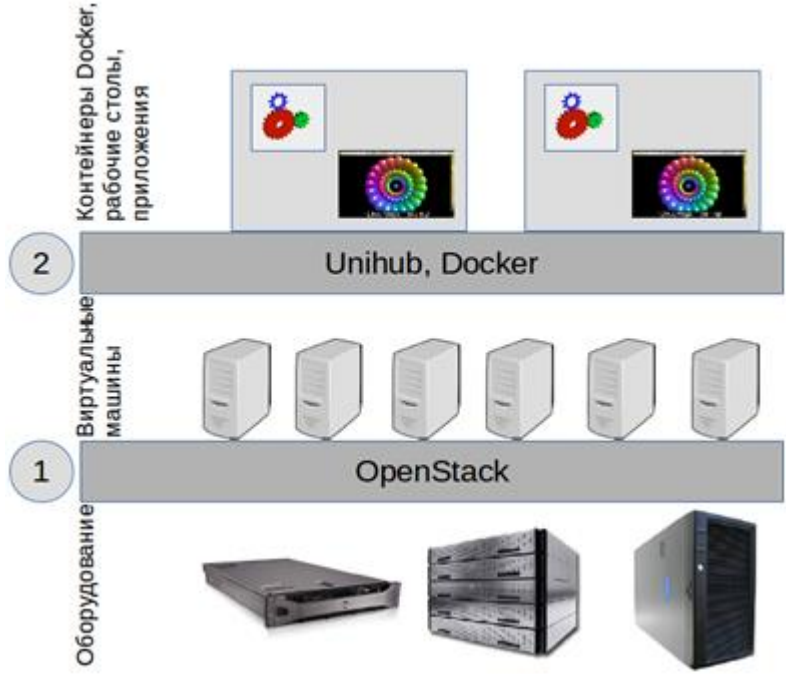
Контейнеры изолируют несколько экземпляров операционной системы Linux на одном сервере друг от друга и позволяют ограничивать процессорное время, память и другие ресурсы, потребляемые отдельным экземпляром.

Каждый контейнер содержит работающий процесс графического рабочего стола Xorg. Внутри рабочего стола пользователь может одновременно работать с несколькими графическими приложениями. В то время, когда рабочие столы продолжают постоянно работать на серверах Unihub, пользователь может отключаться и подключаться к системе в любое время и с любого совместимого устройства.

Работающие на сервере контейнеры контролируются Docker агентом. По умолчанию, если несколько контейнеров работают на одном сервере, то процессорное время разделяется между ними поровну. Однако, если другие контейнеры бездействуют, то один контейнер может использовать процессор полностью. Docker не дает возможности жестко ограничить использование

процессора одним контейнером, например, в размере 1/5, зато позволяет привязать контейнер к определенному процессорному ядру на все время работы контейнера.

Рассмотрим схему управления вычислительными ресурсами в системе Unihub.



На первом уровне вычислительным оборудованием управляет OpenStack. Он предоставляет вышестоящему уровню вычислительные серверы: как виртуальные машины, так и bare metal серверы.* Распределение серверов происходит на этапе инициализации системы, и управляется администратором. В дальнейшем распределении задач пользователей планировщик OpenStack не принимает участия.

На втором уровне планировщик системы Unihub создает и распределяет Docker контейнеры с рабочими столами пользователей по серверам. Контейнер использует квоты (ограничения) на количество разрешенной памяти и процессорного времени. Зная характеристики сервера и требования контейнеров, планировщик решает какое количество контейнеров следует разместить на данном сервере.

* bare metal сервер - это вычислительная система, в которой виртуальная машина устанавливается непосредственно на оборудование, а не поверх гостевой операционной системы.

В системе Unihub все задачи интерактивные, с динамическими требованиями к ресурсам. При появлении задачи планировщик не может знать когда она завершится и сколько ресурсов она будет потреблять в определенный момент времени. Это означает, что, если мы поместим несколько контейнеров на один сервер, и контейнеры при этом начнут активно использовать процессор, то пользователи сразу заметят замедление работы своих приложений,[*] что нежелательно.

Исходя из этого, представляется логичным распределять контейнеры равномерно по всем имеющимся в распоряжении узлам. Таким образом, замедление работы должно будет происходить при общей перегрузке системы - превышении некоторого количества одновременно работающих пользователей. В таких ситуациях говорят о достижении предельной производительности системы и необходимости наращивания оборудования.

Однако, очевидно, что пользователи используют рабочие столы по-разному. Характер нагрузки, создаваемой двумя произвольными контейнерами будет различаться. Это означает, что, разместив одинаковое количество контейнеров на каждом сервере, возникнет ситуация при которой некоторые сервера будут перегружены, а некоторые будут простаивать.

В данной ситуации, оптимальным решением является миграция работающего контейнера на менее загруженный сервер. В последней версии Docker возможность миграции контейнера появилась [12], однако все еще находится на стадии разработки.

Если миграция невозможна, и контейнер должен быть размещен только один раз и работать до конца на одном узле, необходимо следить за тем, чтобы предсказать нагрузку и распределить ее по серверам равномерно. Это значит, планировщику необходима дополнительная информация. Источником такой информации может служить, например, история загрузки серверов за последние несколько минут, количество запущенных приложений на каждом сервере, и т.д. Мы предлагаем использовать статистику запусков задачи и поведения пользователей в системе для оптимизации принятия решений о размещении контейнеров.

Рассмотрим пример работы контейнера в системе Unihub.

1. Пользователь авторизуется в системе.
2. Некоторое время пользователь работает с информационными материалами и не запускает приложений.
3. Пользователь запускает приложение X и некоторое время активно использует его.
4. Пользователь прекратил работать с приложением, но не закрыл его. Приложение работает, но не создает нагрузки.
5. Пользователь вышел из системы.

[*] QoE - quality of experience, качество восприятия.

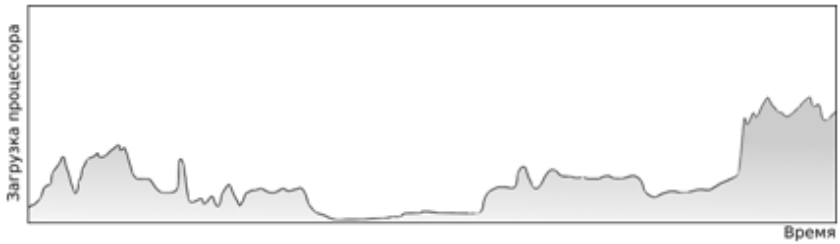
Учитывая то, что пользователи регулярно пользуются системой, данная модель поведения будет с некоторыми вариациями повторятся. Эта информация может быть использована планировщиком. Например, как только пользователь зашел в систему, планировщик уже может предсказать его дальнейшие действия. При создании контейнера планировщик также будет обладать информацией о статистике использования данного приложения как текущим, так и другими пользователями. Очевидно, редко используемые контейнеры можно группировать плотнее и изолировать их от более активных контейнеров, тем самым не вызывая заметных пользователям замедлений в работе приложений.

Использование данных о поведении пользователей в распределенных вычислительных системах неоднократно рассматривалось разными авторами, например, для оптимизации потребления энергии [6], для задачи размещения данных [5], оптимизации расписаний в Grid [7], снижения латентности тонких клиентов [8], запуска виртуальных машин [4] и др.

Во всех работах подтверждается периодичность, характерная для создаваемой пользователями нагрузки. В системе Unihub мы также провели анализ поведения пользователей и выяснили, что больше половины постоянных пользователей работают с системой по расписанию. Можно выделить несколько групп пользователей, работающих одновременно с разной периодичностью.

2.1 Сбор данных и размещение контейнеров

Рассмотрим пример графика создаваемой контейнером нагрузки.



С момента создания контейнера на некотором промежутке времени измеряется загрузка процессора. Мы предлагаем вычислять три величины: L - средняя загрузка на всем интервале, B - средняя продолжительность интервала высокой загрузки, I - средняя продолжительность интервала бездействия (низкой загрузки). В качестве промежутка времени предлагается использовать сессию пользователя.

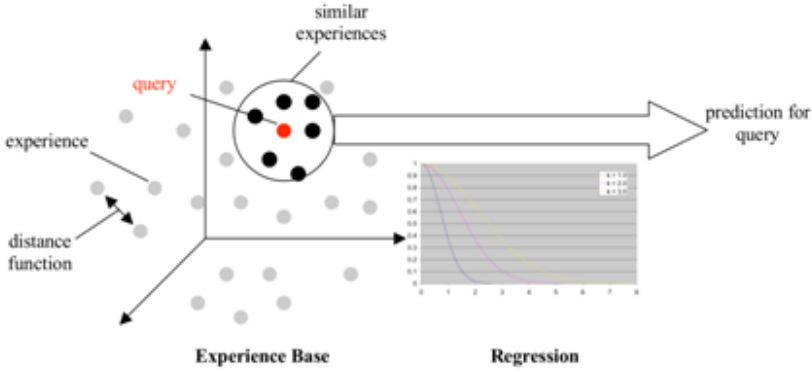
Вместе с указанными величинами сохраняется дополнительная информация:

1. идентификатор пользователя,
2. дата и время,
3. запущенные приложения в других контейнерах пользователя,

4. характеристики контейнера (квота, образ),
5. и др.

Располагая собранными данными, планировщику необходимо при размещении нового контейнера определить предполагаемый характер его нагрузки. Существует несколько способов решения данной задачи. Наиболее часто применяемыми на практике являются способы, основанные на машинном обучении [2]. В данной работе мы применили один из методов основанных на машинном обучении - instance-based learning (IBL), который иногда называется memory-based learning [1].

Данный метод работает следующим образом. В базе данных хранится информация об "опытах". В нашем случае это информация, связанная с историей загрузки контейнера. Опыт состоит из входных и выходных объектов. Входные объекты описывают условия, при которых опыт проводился, а выходные - то, что произошло при этих условиях. Каждый объект, как правило, состоит из имени и значения, где значением может быть простой тип: целое число, число с плавающей точкой, или строка. Запросом к базе является массив входных объектов, по которым мы хотим получить соответствующий опыт.



Близость двух опытов определяется функцией расстояния (distance function). Существует много вариантов вычисления данной функции [9].

Заключительным этапом формирования прогноза является построение массива полученных из базы близких опытов и доверительного интервала.

Получая таким образом значения L,B,I - предполагаемой средней загрузки и интервалов высокой и низкой активности, для размещения контейнера выбирается сервер, для всех контейнеров которого сумма значений предполагаемой средней загрузки в заданном интервале времени не превышала бы значения заданной константной величины L_max (например 0,8), и у которых отношение длины среднего интервала бездействия к средней длине интервала сильной загрузки (I/B) минимально.

В случае, когда планировщик не может определить предполагаемые значения загрузки для данного контейнера, размещение происходит на наименее загруженный за последнее время сервер.

Данные о принятом планировщиком решении также сохраняются. Если в процессе работы сервера величина загрузки превышает предполагаемую, то параметр L_max корректируется. Таким образом, планировщик будет обладать возможность адаптироваться к различным потокам задач.

2.2 Эксперименты

Для сравнения поведения указанных стратегий проводилось моделирование на основе собранной информации из работающей системы Unihub. Мы использовали ранее разработанную в ИСП РАН систему Gridme с некоторыми необходимыми изменениями для моделирования облачных систем.

Для сравнения была выбрана базовая стратегия - размещение контейнеров равным количеством по всем доступным серверам, при котором загрузка сервера не учитывается.

Второй стратегией мы выбрали размещение на наименее загруженный за последние сутки сервер.

Третья стратегия - размещение с учетом собранной дополнительной информации о загрузке контейнеров.

Эксперименты подтвердили, что дисбаланс при использовании базовой стратегии наступает очень быстро. Единственным плюсом базовой стратегии является простота реализации.

Стратегия размещения на наименее загруженный за последние сутки сервер показала лучший результат по отношению к базовой. Вероятность возникновения дисбаланса меньше, и зависит от периода времени за который суммируется загрузка сервера. Наилучшие результаты были показаны при значении интервала в 3 суток. Дальнейшее увеличение интервала приводило к ухудшению результатов.

Стратегия размещения с учетом собранной дополнительной информации показала лучший результат. В среднем, по сравнению с базовой стратегией удалось увеличить количество одновременно работающих контейнеров без ухудшения QoE на 12%.

3. Заключение

Система Unihub является облачной вычислительной системой типа SaaS, предоставляющей доступ к интерактивным приложениям. Основной характеристикой такой системы может являться величина QoE - качество восприятия. В настоящее время данный термин становится все более популярным. Довольно много исследований ведется по созданию новых методов определения качества восприятия пользователями того или иного сервиса или услуги, однако точных стандартизованных методик по

определению QoE, описывающих все случаи, пока не существует. Для системы Unihub величину QoE определяет отсутствие видимого замедления в работе приложений пользователя.

В системе Unihub пользователь, заходя через Web-браузер с любого совместимого устройства, может запустить несколько рабочих столов. Рабочий стол - это графическое окружение Linux, изолированное внутри Docker контейнера. В то время, как рабочие столы постоянно работают на серверах Unihub, пользователь может отключаться и подключаться к системе в любое время.

Такая модель характеризуется динамической нагрузкой. Учитывая, в настоящее время, невозможность миграции работающих контейнеров, основной задачей планировщика является равномерное распределение нагрузки по серверам. В случае дисбаланса значение QoE будет уменьшаться.

В данной работе мы предложили собирать и использовать информацию о характере активности пользователей для оптимизации размещения контейнеров. Наблюдения показали, что для создаваемой пользователями нагрузки характерна периодичность - больше половины постоянных пользователей работают с системой по расписанию, причем по этому показателю всех пользователей можно разделить на несколько групп.

Результаты моделирования предложенных алгоритмов показали, что использование информации о поведении пользователей позволяет поднять плотность упаковки контейнеров на одном сервере без значительных, заметных пользователю потерь производительности приложений. В сравнении с базовой стратегией (размещение контейнеров равным количеством по всем доступным серверам) удалось увеличить плотность упаковки контейнеров в среднем на 12%.

Мы планируем в дальнейшей работе исследовать степень влияния различных параметров на точность предсказания создаваемой контейнером загрузки. Необходимо также изучить устойчивость предсказания и, следовательно, эффективность балансировки, к резкому изменению поведения пользователей, например, при добавлении в систему большой группы новых пользователей.

Список литературы

[1]. Atkeson C., Moore A., Schaal S. Locally weighted learning. Artificial Intelligence Review. 1997. № 1-5 (11). С. 11–73.

[2]. Blum A. On-line algorithms in machine learning Springer, 1996. 306–325 с.

[3]. Deboosere L. [и др.]. Cloud-based desktop services for thin clients. Internet Computing, IEEE. 2012. № 6 (16). С. 60–67.

[4]. Jiang Y. [и др.]. ASAP: A self-adaptive prediction system for instant cloud resource demand provisioning 2011. 1104–1109 с.

[5]. Kavulya S. [и др.]. An analysis of traces from a production mapReduce cluster 2010. 94–103 с.

[6]. Nguyen T.-D. [и др.]. Prediction-based energy policy for mobile virtual desktop infrastructure in a cloud environment. Inf. Sci. 2015. № C (319). С. 132–151.

[7]. Smith W. Prediction services for distributed computing 2007. 1–10 с.
[8]. Suznjevic M., Skorin-Kapov L., Humar I. User behavior detection based on statistical traffic analysis for thin client services Advances in intelligent systems and computing. Под ред. Á. Rocha [и др.]., Springer International Publishing, 2014. 247–256 с.
[9]. Wilson D.R., Martinez T.R. Improved heterogeneous distance functions. J. Artif. Int. Res. 1997. № 1 (6). С. 1–34.
[10]. Unihub: Технологическая платформа программы университетский кластер. http://unihub.ru.
[11]. Docker: An open platform for distributed applications for developers and sysadmins. http://www.docker.com.
[12]. CRIU: A project to implement checkpoint/restore functionality for linux in userspace. http://criu.org/Docker.
[13]. Mohammadi F., Jamali S., Bekvari M., Survey on Job Scheduling algorithms in Cloud Computing, Int. Journal of Emergency Trends of Technology in Computer Science, 2014, v. 3, Issue 2, C. 151-154.
[14]. Tian W., Zhao E., Zhong V., Xu M., Jing C., A dynamic and integrated load-balancing scheduling algorithm for cloud datacenters, Cloud computing and Intelligence Systems (CCIS), 2011, IEEE Int. Conference15-17 September 2011, C. 311-315.
[15]. Duy T.V.T., Sato Y., Inogushi Y., Performance evaluation of a green scheduling algorithm for energy savings in cloud computing, Parallel and Distributed Processing, Workshops and PhD forum, Atlanta, 19-23 April 2010, C. 1-8.

Load Balancing in Unihub SaaS System Based on User Behavior Prediction

D.A. Grushin <grushin@ispras.ru>
N.N. Kuzyurin <nnkuz@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. ISP RAS develops SaaS cloud computing system Unihub. The system provides possibility for users to work via Web-browser with interactive Linux-applications, running in isolated Docker containers. Each container runs Xorg process and variable amount of user applications. Containers share cpu time and have dynamical demands to computational resources. Conventional way of placement when containers are distributed uniformly among all servers can lead to bad result: some servers have too many running applications at some moment but others are almost empty.
Quality of Experience (QoE) is a measure of a customer's experiences with a service. In Unihub we evaluate QoE as an amount of visual slowdown of graphical applications.
In this paper we propose to collect information about users behavior and investigate how different applications work in order to predict containers load and provide this information to the scheduler.

Our observations showed that more then a halt of Unihub users work with the system on scheduled times, and depending on the schedule all users can be divided into several groups. The simulation results of the proposed algorithms have shown that the use of information about the user's behavior allows to run more containers on a given set of servers without significant loss of QoE.

Keywords: cloud computing SaaS systems, Unihub system. Users behavior prediction

DOI: 10.15514/ISPRAS-2015-27(5)-2

For citation: Grushin D.A., Kuzyurin N.N. Load Balancing in Unihub SaaS System Based on User Behavior Prediction. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 23-354 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-2.

References

[1]. Atkeson C., Moore A., Schaal S. Locally weighted learning. Artificial Intelligence Review. 1997. № 1-5 (11). С. 11–73.
[2]. Blum A. On-line algorithms in machine learning Springer, 1996. 306–325 с.
[3]. Deboosere L. [и др.]. Cloud-based desktop services for thin clients. Internet Computing, IEEE. 2012. № 6 (16). С. 60–67.
[4]. Jiang Y. [и др.]. ASAP: A self-adaptive prediction system for instant cloud resource demand provisioning 2011. 1104–1109 с.
[5]. Kavulya S. [и др.]. An analysis of traces from a production mapReduce cluster 2010.С. 94–103.
[6]. Nguyen T.-D. [и др.]. Prediction-based energy policy for mobile virtual desktop infrastructure in a cloud environment. Inf. Sci. 2015. № С (319). С. 132–151.
[7]. Smith W. Prediction services for distributed computing 2007. С. 1–10.
[8]. Suznjevic M., Skorin-Kapov L., Humar I. User behavior detection based on statistical traffic analysis for thin client services Advances in intelligent systems and computing. Springer International Publishing, 2014. С. 247–256.
[9]. Wilson D.R., Martinez T.R. Improved heterogeneous distance functions. J. Artif. Int. Res. 1997. № 1 (6). С. 1–34.
[10]. Unihub: Technological platform for university cluster program. http://unihub.ru.
[11]. Docker: An open platform for distributed applications for developers and sysadmins. http://www.docker.com.
[12]. CRIU: A project to implement checkpoint/restore functionality for linux in userspace. http://criu.org/Docker.
[13]. Mohammadi F., Jamali S., Bekvari M., Survey on Job Scheduling algorithms in Cloud Computing, Int. Journal of Emergency Trends of Technology in Computer Science, 2014, v. 3, Issue 2, C. 151-154.
[14]. Tian W., Zhao E., Zhong V., Xu M., Jing C., A dynamic and integrated load-balancing scheduling algorithm for cloud datacenters, Cloud computing and Intelligence Systems (CCIS), 2011, IEEE Int. Conference15-17 September 2011, C. 311-315.
[15]. Duy T.V.T., Sato Y., Inogushi Y., Performance evaluation of a green scheduling algorithm for energy savings in cloud computing, Parallel and Distributed Processing, Workshops and PhD forum, Atlanta, 19-23 April 2010, C. 1-8.

Implementing Apache Spark Jobs Execution and Apache Spark Cluster Creation for Openstack Sahara[1]

[1]A. Aleksiyants <aleksiyantsa@gmail.com>
[1]O. Borisenko <al@somestuff.ru>
[1,2,4]D. Turdakov <turdakov@ispras.ru>
[1]A. Sher <sher-ars@ispras.ru>
[1,2,3]S. Kuznetsov <kuzloc@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. In this paper the problem of creating virtual clusters in clouds for big data analysis with Apache Hadoop and Apache Spark is discussed. Both clouds and MapReduce models are popular nowadays for a bunch of reasons: cheapness and efficient big data analysis respectively. For these thoughts, having an open source solution for building clusters is important. The article gives an overview on existing methods for Apache Spark cluster creation in clouds. We consider two open source cloud engines OpenStack and Eucalyptus and the most popular proprietary cloud service Amazon Web Services and examine cloud related features presented by these systems. Afterwards, we regard possible ways of creating virtual clusters for big data processing in OpenStack and describe their pros and cons. In the second part we describe in details one of these solutions that uses service Sahara. Sahara represents a cluster management system for OpenStack and it is used for setting up virtual clusters and executing MapReduce jobs. Sahara did not support contemporary versions of Apache Spark. The article introduces the results of our work that led to a Sahara modification, describes its idea and implementation details. By virtue of our modification, Sahara is able to create and use virtual clusters with contemporary versions of Apache Spark in OpenStack clouds.

Keywords: Apache Spark, Openstack, Openstack Sahara, IaaS, PaaS

DOI: 10.15514/ISPRAS-2015-27(5)-3

[1] The work is supported by the RFBR, grant No 14-07-00602

For citation: Aleksiyants A., Borisenko O., Turdakov D., Sher A., Kuznetsov S. Implementing Apache Spark Jobs Execution and Apache Spark Cluster Creation for Openstack Sahara. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 35-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-3.

1. Introduction

Lots of modern computational tasks are implemented using MapReduce[1] parallel paradigm. The computational process in this paradigm consists of two stages: Map and Reduce. Before task execution all the data is distributed over nodes of the cluster. On the Map stage, the master sends the executable task to the other nodes. Then every worker (slave) processes its data. The next step is called Reduce which means the master gets all results from workers and returns the final results which depend on worker's results.

Since MapReduce paradigm is very popular there are a lot of frameworks implementing this model of computations, such as Apache Hadoop, Apache Spark, Infinispan, Gridgain, Riak and other solutions that provide Big Data Processing abilities.[2][3][4][5] Moreover, such frameworks are not usually used as a standalone system; there exists huge stack of related solutions and frameworks that provide another abstraction levels such as jobs isolation, data distribution, task scheduling and so on. Lots of efforts are needed to make these levels work together. Some software companies (such as Cloudera, HortonWorks, MapR etc.) provide their own distributions that include all the components for the whole stack of related technologies.

The most popular MapReduce for today is Apache Hadoop framework. The main goal of this project is implementation of MapReduce. Hadoop uses its file system called HDFS (Hadoop Distributed File System). HDFS is a distributed, fault tolerance block storage system that is the main priority in Apache Hadoop project now.

Another example of MapReduce implementation is Apache Spark[6] that is a successor for Apache Hadoop project. This system is probably the fastest implementation of MapReduce[7][8]. In contrast to Hadoop, Spark performs most computations in main memory boosting the performance. Apache Spark supports most of the common big data stack technologies (primarily – Apache technology stack). It seems that Apache Spark stands for replacement of Apache Hadoop since it's much faster and it was designed to fix Apache Hadoop issues.

As mentioned before, nowadays a software engineer should know not only how to develop applications for Spark but also how to configure this framework with all the needed related technologies for his particular task. It seems to be a problem since configuration part is not really related to developing process.

At the beginning of our project, there were Spark deployment applications but none of them were able to deploy Spark in cloud environments [9][10] with full flexibility of related technology stack. There are different definition of cloud computing but we will use the following: "Cloud is a parallel and distributed

computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the service provider and consumers."[11]

There are 3 most common types of cloud services provided nowadays:

1. Infrastructure as a Service (IaaS) provides access to the whole virtual machine. A client is able to use all functionality and memory given to the machine. In other words, a client maintains the operating system and software of each particular machine himself.
2. Platform as a Service (PaaS). This service provides a platform allowing clients to run and manage certain types of applications supported by the provider. Client can deploy his applications and he doesn't really concern about virtual machine configuration.
3. The last type is Software as a Service (SaaS). This type of service stands for client getting access to applications that are already deployed in the cloud just as a user.

The most common IaaS systems that could be used for clusters creation for big data processing are OpenStack, Amazon EC2 and Eucalyptus. Our project has the following goals:

1. Automated Apache Spark clusters construction in cloud environment.
2. Provide user an ability to choose what version of Apache Spark framework to deploy in a cloud.
3. Provide user an ability to use Apache Spark in couple with related technologies within the same virtual cluster.
4. Provide PaaS access for constructed cluster which stands for simple Spark jobs execution without any knowledge about the real cluster configuration (just click&run).
5. Provide IaaS access for constructed clusters which stands for user ability to login to any virtual machine in cluster and get full control for managing his cluster if one wants to.

The next section is a brief overview of modern IaaS systems that we have mentioned and their main capabilities.

2. Platform overview

2.1 Openstack

OpenStack is an open source cloud platform with modular pluggable architecture and multitenancy support. OpenStack consists of several services that communicate via REST API. Let's take a look at the main OpenStack services.

Nova

Nova is the main module of OpenStack. Nova is responsible for running and maintaining VMs that are controlled by hypervisor. Nova consists of independent daemons responsible for different tasks. Daemons use reliable message queue for communications. The main daemons are Scheduler and Compute. At the initialization step Scheduler decides which node should host new VM. Compute provides communication between hypervisors and central component that manages requests for VMs creations/deletions/etc. All the states changes are sent to the central component that also manages primary database. Nova also stores all the supplementary data about VMs in the cloud.

Keystone

Keystone is the second major subsystem. Keystone is responsible for user authentication and providing lists of services with their endpoints. Any interaction of services and users happens through Keystone. When a client (service or user) wants to interact with any service it sends its credentials to Keystone and if they are correct Keystone subsystem provides a token that is valid for some time span. After that the client can send requests to services with the given token.

Glance

Glance is an OS image management service. Glance can use various storage backends for images: file systems or object storage systems like Swift. The main task of the project is providing boot images for VMs in Nova.

Neutron

Neutron is a service for network management. Neutron is a rather complicated service which provides virtual network connectivity to VMs in Nova. Neutron allows users to organize multilevel networks with complicated topologies, complicated security in- and outbound rules. It provides tenants networks isolation and external network services such as floating IP pools and traffic balancing. Neutron can use various network hardware resources directly through specially designed for each hardware resource plugins.

Cinder

Cinder is a block storage service. Cinder is responsible for providing persistent block storage to VMs on-demand. These block volumes are attached to compute nodes via iSCSI protocol and then they could be used by VMs as a local storage volumes that don't disappear after VM shutdown.

Swift

Swift is a highly available, distributed, eventually consistent object storage [12]. Object stands for file plus unstructured user-defined metadata for each file. Swift has support for stored objects compressing. Swift provides access via REST API much common to Amazon S3 which makes them easily interchangeable.

Horizon

Horizon is a Web GUI application for OpenStack services. It works as a client for each Openstack service and aggregates the capabilities of the services in one place for common usage of all the subsystems. Horizon is based on Django framework and uses native client libraries for each Openstack project.

Heat

Heat is a configuration management tool for OpenStack. For node management Heat uses templates which represent some sort of scenarios. These templates should be written in AWS format (Amazon Web Services) or HOT (Heat Orchestration Template). It also provides its own metadata server which tries to convert Nova metadata in terms of Amazon EC2 metadata but there is no full compatibility and it provides not all of the Amazon EC2 metadata terms.

Sahara

Sahara (formerly «Savanna») provides PaaS access for most common big data tools using Heat or its own engine for VM configuration. This project will be observed later in more detailed way.

Ceilometer

Ceilometer is a resource usage monitoring system. This system is used for gathering information about using VMs. The service is mostly used in business projects for controlling VMs of clients and billing.

2.2 Eucalyptus

Yet another open source cloud platform. Like OpenStack, Eucalyptus consists of several loosely coupled components communicating over WSDL (Web Service Description Language) [13]. Let's take a look at the Eucalyptus architecture and its main components.

Node controller

Node Controller is deployed on every physical node that is intended to be used for VM hosting. It is responsible for launching, maintaining and destroying VMs. The service represents a communication point between Cluster Controller and operating systems on VMs. Also Node Controller knows all the configurations of VMs on its

host. When one wants to create a VM the Cluster Controller sends a signal to some Node Controller and after that the Node Controller creates a VM from image on the node and passes control to hypervisor of VM.

Cluster controller

This component gathers information about running VMs and manages the network. Cluster Controller could be run on any machine that is connected to the Cloud Controller and the network of Node Controllers. Cluster controller gets commands from high level controllers and transmits them to specific Node Controllers. Gathered cluster data sends back to Cloud Controller.

Virtual network overlay

This component represents the network for Eucalyptus components. All VMs should be interconnected and at least one of them should have a connection to the Internet. At the start VMs interfaces are attached to software Ethernet bridge of a physical machine. Then VMs get IP and MAC addresses. VMs can communicate freely inside their cluster but if it is needed to send a message to another node then Virtual network overlay works as a router.

Storage controller (Walrus)

Walrus is a storage with Amazon S3 compatible API. It is used for storing user data and VM images.

Cloud controller

Cloud Controller is an entry point to the system for users. Cloud Controller is responsible for resource provisioning and monitoring, managing user and system data, GUI and authentication.

Eucalyptus does not have any tools for deploying Hadoop or Spark on clusters[14]. There are articles about Hadoop performance on Eucalyptus clusters but tuning and configuring were done by scripts[15].

2.3 Amazon Elastic Cloud (EC2)

Amazon EC2 is the most popular proprietary cloud platform. EC2 uses hypervisor Xen for creating VMs. At the same time EC2 has a lot of options for configuring. This platform has a MapReduce service called Amazon Elastic MapReduce (Amazon EMR) for big data processing which can deploy and run Spark as PaaS. This service can create clusters with different extensions like Hive, Pig, Impala and others. Since this project is proprietary we can use it just as it is provided without any control.

2.4 Conclusions

We have observed the most popular modern IaaS systems that are suitable for big data processing clusters creation: Eucalyptus, Amazon EC2 and OpenStack. Amazon EC2 is a proprietary platform with its own closed-source MapReduce service. Eucalyptus platform has no special support for creating big data processing clusters at all, unlike Openstack, which has Sahara project. As a result, we decided to concentrate on the latter. In the rest of this paper we describe existing Apache Spark clusters deployment existing approaches, our proposed solution and its implementation.

3. Apache Spark clusters deployment existing approaches

At the start of our project, OpenStack had a subsystem called Heat[16]. One of the goals of Heat project was to make OpenStack behave similar to Amazon EC2. This would have made possible to use Amazon cluster deploying scripts (bundled with Spark sources) for OpenStack. In the last-year work[17] we have managed to adapt it for Openstack environment but it's still not flexible, doesn't provide an ability to use other components of big data processing tools they need to use specially prepared images for each Apache Spark release.

At the present time there are three ways of building Apache Spark clusters.

3.1 Manual configuration

The first way is to configure all the components manually which means that we need to deploy all the needed components separately and for each VM we use. In the minimal configuration that means that we should install JVM, Scala, Apache Spark on each node; manually set connectivity and roles in that cluster and run all the nodes hoping that no errors were made during configuration. If we need more than just Apache Spark, we should do all the steps for each separate component.

NB: some helper tools such as Cloudera manager exist for that task, but that still means that you should setup all the base requirements for Cloudera manager on each VM in your cluster and then manually provision the cluster with Cloudera web-interface which is still complicated and long process.

3.2 Amazon EC2 IaaS approach

We have already mentioned the PaaS for big data processing that Amazon provides (EMR). Nevertheless there is another approach which is used by Apache Spark developers team and which we have used earlier [17].

This approach stands for deploying clusters in Amazon EC2[18][19] (or using similar adapted scripts in Openstack). In this case one should use pre-configured images with installed components and configuring scripts. To support HDFS, the images should use two types of storage devices for storing persistent and temporary data.

The script executes the following steps:

1. The launching process is initiated from Amazon EC2 API. This process launches VMs with chosen OS from image based on Red Hat Enterprise Linux 5.3.
2. The script sets up security groups via API.
3. The script waits for 5 minutes.
4. The script gets nodes IP addresses via API.
5. Configuration files are created for all nodes.
6. A script is run on the master node over ssh. This script downloads configuration templates from repositories.
7. The master node sends a file containing master and worker addresses over ssh to other nodes.
8. The master node creates worker's configuration files from templates.
9. The script sends created configuration files to workers.
10. The master node runs HDFS and Apache Spark.

After these steps the cluster are ready to work.

Drawbacks for this approach are the following:

1. Pre-configured images for Amazon EC2 are private. We used to prepare images for Openstack based on CentOS but it takes a lot of time and Apache Spark releases appear too fast.
2. Our scripts need a lot of tune-up for particular Openstack environment: we should provide networks ids for internal and external networks, external IP address allocation method, we should know what networking system uses particular Openstack installation (it could be Quantum/Neutron or legacy Nova networking), we should implement separate adaptors for communications for each version of API.
3. This approach is not extendable and doesn't allow using other than Apache HDFS and Apache Spark frameworks.

These actions could be managed more easily with orchestration tools such as Ansible/Salt/Chef/Puppet but the main problems of this approach would remain the same.

3.3 Openstack Sahara approach

The third option is using Sahara project in OpenStack. The main goals of Sahara are creating clusters for big data processing in OpenStack, deploying MapReduce and running jobs on clusters.[20] Before our project succeeded, Sahara could create clusters with only vanilla Apache Spark 1.0 which was already obsolete at that moment and there was no choice of other frameworks for such clusters. Also it provided access to created clusters in IaaS terms only.

At the present time Amazon EC2 lets users to create clusters with Apache Hadoop or Apache Spark automatically. But this project is not open source and does not satisfy us.

Considering the last two approaches, the most attractive is direct integration with OpenStack because of Sahara existence.

At the start of the project all these solutions either did not satisfy our requirements or needed a lot of supplementary work during every cluster launch. Therefore there was no open source system that would create clusters with current Apache Spark version from different distributors.

We decided to implement the missing functionality in Openstack Sahara project.

4. Proposed solution

Sahara works with different distributions for big data. Spark support in Sahara was limited to creation of clusters with vanilla Spark 1.0 only with basic support for running jobs via Sahara REST API with custom methods for Spark.

At the same time Sahara could create clusters with other MapReduce frameworks like Hortonworks, Cloudera and MapR. These distributors frequently update versions for distributions components including Spark. Since Sahara tries to keep up with versions of these frameworks it made sense to add Spark support as a general adapter for all the distributions and to make one specific implementation as an example for others. We have chosen to implement Spark support for Cloudera and this choice is quite random – it could be done in the same way for the others.

MapReduce implementation is defined by plugin system in Sahara. A client can create clusters using quite simple interface. Moreover, it is possible to add and remove cluster nodes online. You can see module interaction in Figure 1.

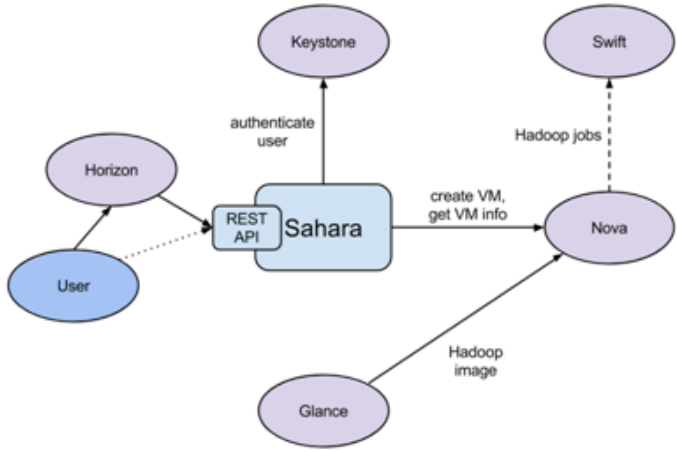


Figure 1

Cluster creation happens as follows:

1. Choose Hadoop version.
2. Choose image. Any image with cloud-init could be used.

3. Configure cluster and nodes. During this step, master and worker nodes are defined by choosing running processes for nodes.
    a. Create Node Group Template. Node group stands for group of VMs that use the same set of frameworks and tools onboard. You can assign roles to Node Groups such as HDFS Namenode, HDFS Datanode etc.
    b. Create Cluster Node Template. Cluster configuring stands for selecting Node Groups that should be included to cluster and the quantity of VMs for each Node Group.
4. Run cluster

At the start of the project you could run different MapReduce jobs (Pig, Hive, Java) through Horizon web-interface on running cluster depending on Hadoop version. Spark jobs were not on the list that time.

4.1 Sahara Architecture Overview

Figure 2 shows Openstack Sahara architecture. We have marked with colors the components we have modified to achieve our goals.

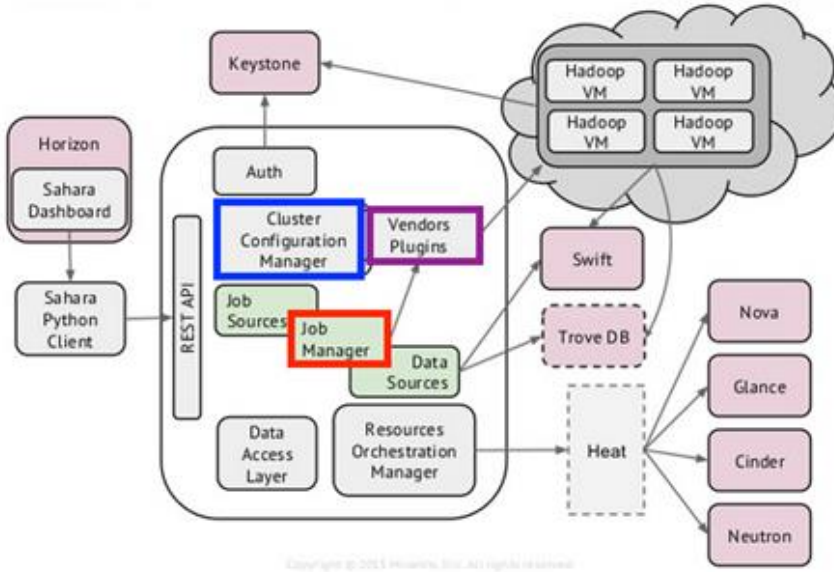


Figure 2

1. Auth component is responsible for client authentication and authorization, communicates with the OpenStack Identity service (Keystone).
2. DAL - Data Access Layer, persists internal models in DB.

3. Provisioning Engine is a component responsible for communication with the Openstack Compute (Nova), Orchestration (Heat), Block Storage (Cinder) and Image (Glance) services.
4. Vendor Plugins is pluggable mechanism responsible for configuring and launching data processing frameworks on provisioned VMs. Existing management solutions like Apache Ambari and Cloudera Management Console could be utilized for that purpose as well.
5. EDP - Elastic Data Processing is responsible for scheduling and managing data processing jobs on clusters provisioned by Sahara.
6. REST API - exposes Sahara functionality via REST HTTP interface.
7. Python Sahara Client - like other OpenStack components, Sahara has its own Python client
8. Sahara pages - a GUI for the Sahara is located in the OpenStack Dashboard (horizon).[21]

Sahara gets requests over REST API with web framework Flask. As mentioned before, support for different MapReduce distributions is implemented via plugins. Sahara uses SQLAlchemy ORM to be able to use different RDBMS for internal data handling (such as storing jobs and their state, clusters configurations etc).

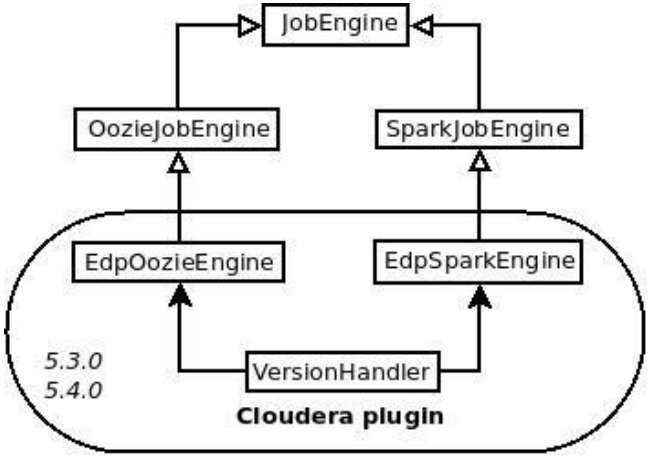


Figure 3

Sahara runs jobs on cluster through JobEngine class. This class is the base class for all types of jobs. Spark jobs are run from SparkJobEngine, which contains the most technical part. Every distribution with Spark support has its own implementation of this class. We have added another implementation for Cloudera called EdpSparkEngine. Previously Cloudera plugin could run Hadoop jobs only and they were run over Oozie.

We have made possible for Cloudera plugin to choose 'engine' class depending on the type of a job one wants to run. Now plugin's class VersionHandler returns either

Oozie or Spark engine. In addition to this we have changed common SparkJobEngine class so it could be used for all the plugins (Vanilla Spark and Cloudera were implemented during the project). This feature is supported in Cloudera 5.3.0 and 5.4.0.

Also we have changed the way Cloudera clusters are configured and made possible to create Spark clusters on Yarn using Cloudera plugin in Spark. That feature worked in standalone Cloudera version before (not in Sahara version).

Spark jobs are supposed to use storage for input and output. Originally Cloudera Spark could work with HDFS but not with Swift. We have modified classes responsible for configuring Cloudera cluster so now such clusters are able to work with Swift using Cloudera HDFS to Swift adapter.

You can check all the implementation details in Sahara official repository; all the code is available under open-source license.

6. Results

As a result of this project we have now full support for Apache Spark cluster creation in Openstack environments with support for running Spark jobs via web-interface. Also Swift object storage can be used by Spark applications transparently in Cloudera plugin for Sahara instead of HDFS. All the code is included in Openstack Liberty release and available in the official Openstack Sahara repository.

References

[1]. Jeffrey D., Sanjay G. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
[2]. Official Hadoop homepage - http://hadoop.apache.org/
[3]. Official Infinispan homepage - http://infinispan.org/
[4]. Official Cloudera CDH Apache Hadoop homepage - http://www.cloudera.com/content/cloudera/en/productsand-services/cdh.html
[5]. Official BashoRiak homepage - http://basho.com/riak/
[6]. Official ApacheSpark homepage - http://spark.apache.org/
[7]. M. Chowdhury, M. Zaharia, I. Stoica. Performance and Scalability of Broadcast in Spark. 2010.
[8]. VMWare Serengeti page - http://www.vmware.com/hadoop/serengeti
[9]. Official Cloudera Manager homepage - http://www.cloudera.com/content/cloudera/en/products-andservices/cloudera-enterprise/cloudera-manager.html
[10]. Buyya R., Broberg J., Goscinski D. Cloud Computing: Principles and Paradigms. Wiley, 2011, 664 P.
[11]. Buyya R., Yeo C. S., Venugopal S. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. CoRR, (abs/0808.3558), 2008
[12]. Swift Architectural Overview - http://docs.openstack.org/developer/swift/overview architecture.html
[13]. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language - http://www.w3.org/TR/wsdl20/

[14]. Nurmi, D. The Eucalyptus Open-Source Cloud-Computing System. Cluster Computing and the Grid. 2009. 10.1109/CCGRID.2009.93
[15]. Nilson J. Hadoop MapReduce in Eucalyptus Private Cloud. Bachelor's Thesis in Computing Science. Umea, Sweden, 2011
[16]. Official Openstack Heat homepage - https://wiki.openstack.org/wiki/Heat
[17]. O. Borisenko, D. Turdakov, S. Kuznetsov. Avtomaticheskoe sozdanie virtual'nȳkh klasterov Apache Spark v oblachnoĭ srede Openstack. [Automating cluster creation and management for Apache Spark in Openstack cloud] Trudy ISP RAN [The Proceedings of ISP RAS], volume 26, issue 4, p. 33-43, 2014. (In Russian)
[18]. Official Amazon Elastic Compute Cloud (EC2) homepage - http://aws.amazon.com/ec2/
[19]. Creeger, Mache. Cloud Computing: An Overview. ACM Queue 7. 5. 2009
[20]. OpenStack Sahara roadmap - https://wiki.openstack.org/wiki/Sahara/Roadmap
[21]. OpenStack Sahara Architecture - http://docs.openstack.org/developer/sahara/architecture.html

Реализация сервиса для выполнения Apache Spark задач и создания Apache Spark кластеров на основе Openstack Sahara[1]

[1]А.В. Алексиянц <aleksiyantsa@gmail.com>
[1]О.Д. Борисенко < al@somestuff.ru >
[1,2,4]Д. Ю. Турдаков <turdakov@ispras.ru>
[1]А. В. Шер <sher-ars@ispras.ru>
[1,2,3]С. Д. Кузнецов <kuzloc@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В работе рассматривается задача создания виртуальных Apache Spark и Apache Hadoop кластеров для обработки больших данных в облачных средах. Проведен обзор существующих методов создания Apache Spark кластеров. Также

[1] Работа поддержана грантом РФФИ 14-07-00602 А Исследование и разработка методов автоматизации масштабирования и разворачивания виртуальных кластеров для обработки сверхбольших объёмов данных в облачной среде Openstack

описывается реализованный способ создания Apache Spark кластеров и сервиса для выполнения Apache Spark задач в среде OpenStack. Предложенное решение включено в проект OpenStack Sahara и доступно начиная с релиза OpenStack Liberty.
Ключевые слова: Apache Spark, Openstack, Openstack Sahara, IaaS, PaaS
DOI: 10.15514/ISPRAS-2015-27(5)-3
Для цитирования: Алексиянц А.В., Борисенко О.Д., Турдаков Д. Ю., Шер А.В., Кузнецов С. Д. Реализация сервиса для выполнения Apache Spark задач и создания Apache Spark кластеров на основе Openstack Sahara. Труды ИСП РАН, том 27 (выпуск 5), 2015 г. стр. 35-48. DOI: 10.15514/ISPRAS-2015-27(5)-3.

Список литературы

[1]. Jeffrey D., Sanjay G. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
[2]. Официальная страница проекта Hadoop - http://hadoop.apache.org/
[3]. Официальная страница проекта Infinispan - http://infinispan.org/
[4]. Официальная страница Cloudera CDH Apache Hadoop - http://www.cloudera.com/content/cloudera/en/productsand-services/cdh.html
[5]. Официальная страница BashoRiak - http://basho.com/riak/
[6]. Официальная страница Apache Spark - http://spark.apache.org/
[7]. M. Chowdhury, M. Zaharia, I. Stoica. Performance and Scalability of Broadcast in Spark. 2010.
[8]. Официальная страница VMWare Serengeti - http://www.vmware.com/hadoop/serengeti
[9]. Официальная страница Cloudera Manager - http://www.cloudera.com/content/cloudera/en/products-andservices/cloudera-enterprise/cloudera-manager.html
[10]. Buyya R., Broberg J., Goscinski D. Cloud Computing: Principles and Paradigms. Wiley, 2011, 664 P.
[11]. Buyya R., Yeo C. S., Venugopal S. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. CoRR, (abs/0808.3558), 2008
[12]. Обзор архитектуры Swift - http://docs.openstack.org/developer/swift/overview architecture.html
[13]. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language - http://www.w3.org/TR/wsdl20/
[14]. Nurmi, D. The Eucalyptus Open-Source Cloud-Computing System. Cluster Computing and the Grid. 2009. 10.1109/CCGRID.2009.93
[15]. Nilson J. Hadoop MapReduce in Eucalyptus Private Cloud. Bachelor's Thesis in Computing Science. Umea, Sweden, 2011
[16]. Официальная страница Openstack Heat - https://wiki.openstack.org/wiki/Heat
[17]. О. Д. Борисенко, Д. Ю. Турдаков, С. Д. Кузнецов. Автоматическое создание виртуальных кластеров Apache Spark в облачной среде OpenStack. Труды Института системного программирования РАН, том 17, 2009 г. Стр 31-50.
[18]. Официальная страница Amazon Elastic Compute Cloud (EC2) - http://aws.amazon.com/ec2/
[19]. Creeger, Mache. Cloud Computing: An Overview. ACM Queue 7. 5. 2009
[20]. OpenStack Sahara roadmap - https://wiki.openstack.org/wiki/Sahara/Roadmap
[21]. Архитектура OpenStack Sahara - http://docs.openstack.org/developer/sahara/architecture.html

Метод тестирования производительности и стресс-тестирования центральных сервисов идентификации облачных систем на примере Openstack Keystone[1]

[1]И.В. Богомолов <igor95n@gmail.com>
[1]А.В. Алексиянц <aleksiyantsa@gmail.com>
[1]А. В. Шер <sher-ars@ispras.ru>
[1]О.Д. Борисенко <al@somestuff.ru >
[1,2,3]А.И. Аветисян <arut@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. На сегодняшний день платформа OpenStack является одной из лидирующих технологий в области построения облачных сервисов. Keystone - это один из основных сервисом платформыOpenStack. Данный компонент отвечает за аутентификацию и авторизацию пользователей в системе. Keystone является сильнонагруженным сервисом в инфраструктуре OpenStack, поскольку любое взаимодействие между сервисами происходит через Keystone. Таким образом, увеличение числа пользователей облачным сервисом значительно усиливает нагрузку на Keystone. В этой статье мы обращаем внимание на проблему деградации данного сервиса при постоянной нагрузке. Чтобы найти причину такого поведения, мы протестировали сервис Keystone с различными СУБД (MariaDB, PostgreSQL), HTTP-серверами (Apache2, nginx) и физическими устройствами хранения данных (HDD, SSD и tmpfs в оперативной памяти). Использование различных конфигураций для запуска Keystone было необходимо, чтобы сузить поиск причины возникающей деградации. Все тесты запускались через тестирующую систему OpenStack Rally. В качестве результата мы обнаружили, что система начинает довольно быстро деградировать при достаточно слабых нагрузках. Так же было отмечено, что подобная деградация возникает во всех возможных конфигурациях. С целью подтвердить, что

[1] РФФИ **15-29-07111 офи_м** Исследование методов обеспечения масштабируемости систем в облачных средах и разработка высокопроизводительного отказоустойчивого центрального сервиса идентификации

подобное поведение Keystone является неудовлетворительным, мы реализовали макетный сервис, который выполняет простейшие функции Keystone. Подобный сервис отказался значительно быстрее самого Keystone. Мы предполагаем, что проблема с Keystone может возникать в результате реализации внутренней логики приложения, либо неэффективного взаимодействия с другими компонентами. Поиск причин является следующим этапом нашего исследования.

Ключевые слова: OpenStack, Keystone, Rally

DOI: 10.15514/ISPRAS-2015-27(5)-4

Для цитирования: Богомолов И.В., Алексиянц А.В., Шер А.В., Борисенко О.Д., Аветисян А.И. Метод тестирования производительности и стресс-тестирования центральных сервисов идентификации облачных систем на примере Openstack Keystone. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 49-58. DOI: 10.15514/ISPRAS-2015-27(5)-4.

1. Введение

На текущий момент облачные технологии становятся все более востребованными. Наиболее популярным решением в области открытых облачных сред является платформа OpenStack [1][2][3]. К важнейшим компонентам OpenStack относится сервис Keystone [4], который служит для авторизации и аутентификации пользователей. В его обязанности также входит каталогизация доступных в облаке сервисов. Поскольку любое взаимодействие с облаком, требующее аутентификации и/или авторизации проходит через Keystone, его производительность имеет значительное влияние на качество работы облака в целом.

В ходе эксплуатации OpenStack нами были замечены серьезные проблемы в производительности Keystone. На их существование также указывают, например, следующие источники [5][6].

Таким образом, актуальной видится задача сбора статистики производительности Keystone на различных конфигурациях, которую далее можно будет использовать для поиска узких мест в работе сервиса. Данная работа посвящена достижению этой цели.

В следующем разделе описываются конфигурации тестов и общая методика тестирования. Далее рассматриваются выбранные метрики. В четвертом разделе анализируются результаты.

2. Методика тестирования

С точки зрения последовательности обработки запросов архитектуру Keystone можно разделить на три уровня: верхний, принимающий запросы (веб-сервер), центральный, реализующий основную логику обработки, и нижний, отвечающий за ввод-вывод (базы данных). Проблемы с производительностью могут происходить на любом из них или между ними. В качестве веб-сервера тестировались Apache2 с mod_wsgi и nginx с uwsgi. В качестве слоя баз

данных использовались MariaDB и PostgreSQL. Отметим, что при этом настройки Keystone и используемых компонентов оставались по умолчанию и не оптимизировались под каждую тестовую конфигурацию. Это решение обосновывается тем, что цель данной работы заключается не в оптимизации Keystone под конкретные конфигурации, а в выявлении общих проблем производительности Keystone. Разнообразие выбранных компонентов позволит нам в случае деградации скорости работы на всех выбранных конфигурациях утверждать, что источник проблемы находится не в этих компонентах, а в самом Keystone.

Место для хранения базы данных выбиралось из тех же соображений – тестировались разные конфигурации, чтобы с большей точностью определить узкие места Keystone. При этом были использованы обычные жесткие диски (HDD), твердотельные накопители (SSD) и, наконец, файловая система в оперативной памяти tmpfs.

Таким образом, получается набор конфигураций, показанный на табл. 1.

Табл. 1 Набор тестовых конфигураций

№	Хранение БД	Веб-сервер	БД
1	HDD	mod_wsgi + Apache2	MariaDB
2	HDD	mod_wsgi + Apache2	PostgreSQL
3	HDD	uwsgi + nginx	MariaDB
4	HDD	uwsgi + nginx	PostgreSQL
5	SSD	mod_wsgi + Apache2	MariaDB
6	SSD	mod_wsgi + Apache2	PostgreSQL
7	SSD	uwsgi + nginx	MariaDB
8	SSD	uwsgi + nginx	PostgreSQL
9	tmpfs	mod_wsgi + Apache2	MariaDB
10	tmpfs	mod_wsgi + Apache2	PostgreSQL
11	tmpfs	uwsgi + nginx	MariaDB
12	tmpfs	uwsgi + nginx	PostgreSQL

Для проведения тестов был использован OpenStack Rally [7]. Rally создан специально для нагрузочного тестирования OpenStack облаков. Весь процесс тестирования автоматизирован с помощью инструмента оркестрации Ansible[8].

2.1 Конфигурация тестового стенда

Keystone был развернут на одном вычислительном узле с конфигурацией, указанной на табл. 2.

Табл. 2. Описание тестового стенда

Компонент	**Описание**
Процессор	Intel Core i7-4790 CPU 3.60GHz
Оперативная память	16 GB DDR3, 1600MHz

HDD	Seagate ST2000DM001-1ER164 7200 rpm
SSD	3 диска Kingston V300 450MB/s, все диски объединены в RAID 0 под управлением аппаратного контроллера
Операционная система	Ubuntu Server 14.04.3 LTS

3. Обзор метрики

Мы считаем, что система деградирует при определенной нагрузке, если растет время отклика на запросы, или если происходят ошибки транспортного или HTTP уровня. Таким образом, основной метрикой в нашем тестировании является наличие или отсутствие деградации производительности системы, что свидетельствует о её работоспособности. Работоспособность системы зависит от нагрузки на систему и компонент, с которыми она взаимодействует. Набор используемых компонент определяются описанными выше конфигурациями, а нагрузка числом запросов в секунду к Keystone (Requests per second, далее RPS).

Rally может работать как консольное приложение, так и в качестве демона. Шаблоны тестов называются сценариями и задаются в виде json или yaml файла. Сценарии группируются по темам, каждая тема в коде Rally соответствует классу в Python (например, KeystoneBasic), унаследованному от base.Scenario. Методы этого класса и являются сценариями, а его аргументы - аргументами сценария; таким образом, разработка собственных сценариев достаточно проста.

Кроме того, у описания всех сценариев есть еще три раздела, задающие общее для всех тестов поведение: runner, context и sla. Раздел runner определяет, какого типа будет нагрузка и ее интенсивность. С помощью раздела context указывается контекст нагрузки, например, количество пользователей и тенантов, создающих ее. В опциональном разделе sla (service-level agreement) можно задать условия, при которых тестирование заканчивается. Для тестирования Keystone мы выбрали самый простой существующий сценарий — выдача токена.

В первую очередь нас интересуют результаты работы Keystone при горячем старте. Для этого каждый очередной тест будет запускаться таким образом, чтобы система работала не менее 6 минут. В качестве результатов тестирования рассматриваем поведение системы в течение 5 минут после завершения разогревочного этапа, равного одной минуте.

Будем считать, что N – максимальное значение RPS, при котором не наблюдается деградации в течение 5 минут. Считалось, что система деградирует, если за время выполнения теста был хотя бы один ответ с ошибкой Keystone, или если время отклика начало возрастать. Способ обнаружения спада производительности будет описан ниже.

Для нахождения значения N для каждой комбинации «конфигурация+сценарий» необходимо решить три задачи:

Первая — это непосредственный поиск N. Нахождение N выполняется бинарным поиском, поскольку в выбранной конфигурации факт наличия или отсутствия деградации монотонно зависит от RPS. В качестве верхней границы было выбрано значение 200 RPS, при котором система гарантированно имела деградацию во всех наших конфигурациях.

Следующей задачей является построение сценария для очередного запуска теста. Как было сказано выше, каждый тест должен выполняться не менее 6 минут. В таком случае для каждого теста будет вычисляться значение числа итераций, которое бы нагрузило систему в течение не менее 6 минут при выбранном значении RPS. Для этого необходимо выбранную частоту умножить на 360 секунд. Например, для 200 RPS это равно 360 сек. * 200 RPS = 72000 запросов.

Последней проблемой поиска N является обнаружение факта деградации системы или её отсутствия. Для этого строится модель линейной регрессии по выполненным запросам вида $y = ax + b$. Будем считать, что деградация отсутствует, если a близок к 0 (для нашего тестирования мы выбрали $a < 0.01$). В случае возникновения хотя бы одной ошибки при ответе считалось, что система деградирует.

После получения N проводятся эксперименты для N-1 и для N+1, чтобы на них визуально убедиться, что полученное N верно. После этого выполняются тесты при N+1 и собираются данные о выполнении тестов.

Результатом тестирования будут являться выходные файлы Rally, которые могут быть двух видов: JSON и HTML. JSON удобно использовать для обработки, а HTML, помимо результатов, содержит код на javascript, предназначенный для их наглядного отображения.

Написанное нами тестирующее приложение для каждого выполненного запроса проверяет отсутствие ошибок в разделе errors и считывает значения duration и timestamps для их последующей обработки.

4. Анализ данных

На графиках представлены результаты Keystone для двух конфигураций: SSD, MariaDB, uWSGI и HDD, PostgreSQL, Apache2 для N и N+1 запросов в секунду.

Для первой представленной конфигурации N=37, для второй N=99. Как видно из графиков, при переходе через определенное значение RPS система показывает постепенную деградацию производительности. Даже изменение RPS на единицу сильно меняет среднее время ответа на запрос в рассматриваемом нами диапазоне. Аналогичная картина наблюдается во всех остальных конфигурациях.

Для сравнения мы реализовали простой сервис, написанный на языке Python с использованием микрофреймворка для веб-приложений Flask[9], который

принимает запрос на выдачу токена, генерирует токен и запоминает информацию о выданных токенах. Этот простой сервис отработал значительно лучше Keystone. Для такого микросервера N оказалось равным 711. Так как все сгенерированные токены за время работы хранились в памяти программы, то можно сравнивать эти результаты с любой конфигурацией Keystone с использованием tmpfs в качестве устройства хранения.

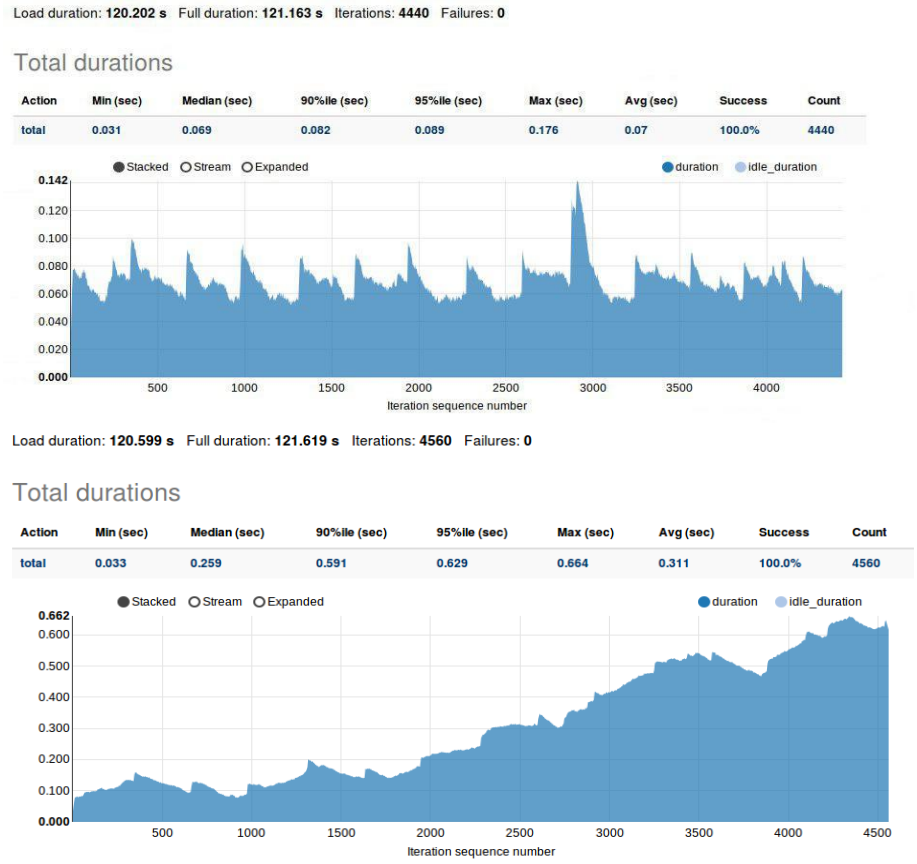


Рис. 1 tmpfs, MariaDB, uWSGI. На оси ординат обозначено время отклика в секундах.

Для нашего тестового сервиса на первом графике N=711, а на втором N=713. Результаты такого сервиса оказались значительно лучше результатов работы Keystone в аналогичной конфигурации. Также стоит отметить, что все тестируемые сценарии проводились с параметром "resource_management_workers": 30, который отвечает за число рабочих

процессов, используемых Rally. При тестировании Keystone для Rally всегда хватало ресурсов на всех экспериментах, поэтому этот параметр не был критичным. В то время как при тестировании нашего сервера Rally не хватало производительности процессора при высоких значениях RPS, что ухудшило результаты, поскольку Rally и тестируемое приложение располагались на одной машине.

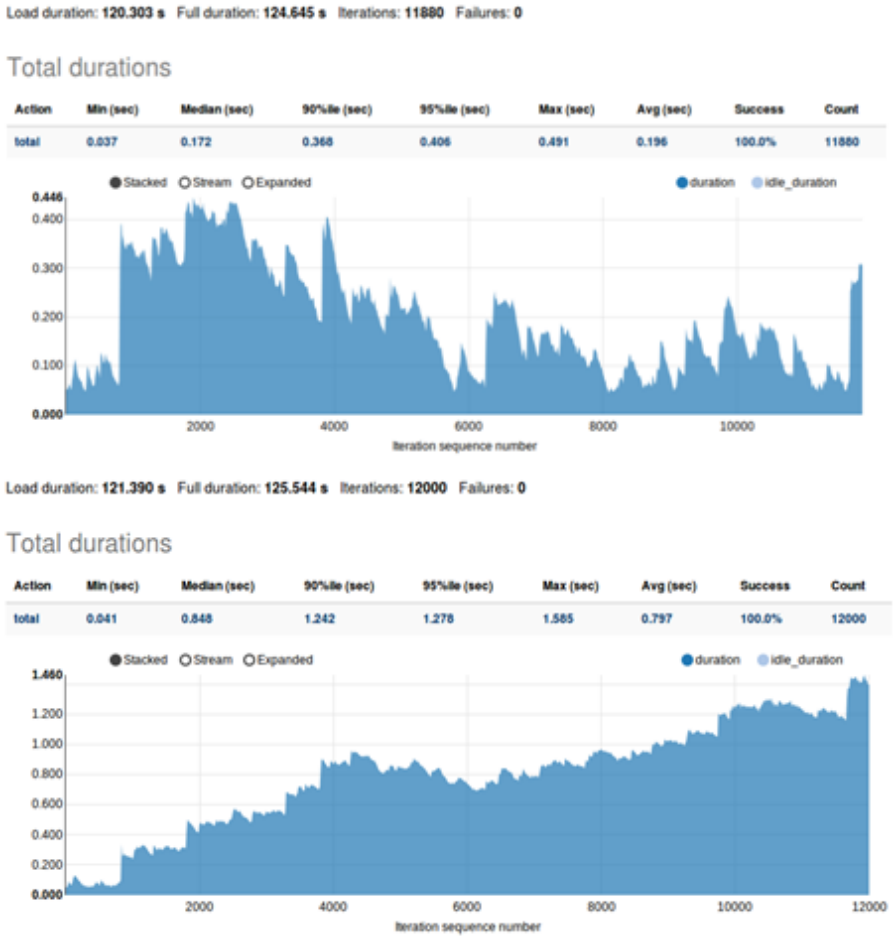
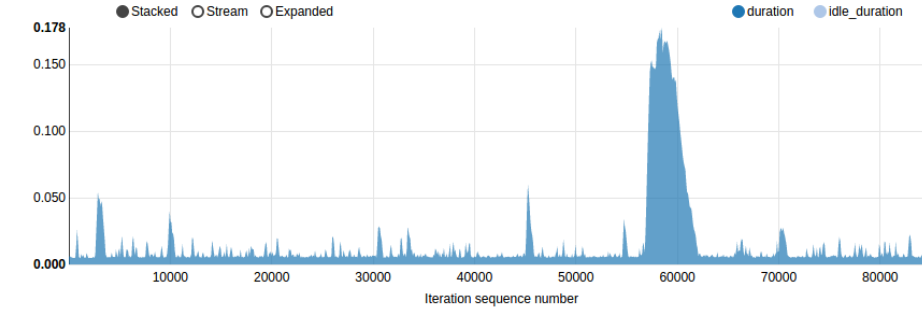


Рис. 2 HDD, PostgreSQL, Apache2. На оси ординат обозначено время отклика в секундах.

Load duration: **120.026 s** Full duration: **120.056 s** Iterations: **85320** Failures: **0**

Total durations

Action	Min (sec)	Median (sec)	90%ile (sec)	95%ile (sec)	Max (sec)	Avg (sec)	Success	Count
total	0.003	0.006	0.022	0.054	0.245	0.015	100.0%	85320



Load duration: **123.624 s** Full duration: **123.654 s** Iterations: **85560** Failures: **0**

Total durations

Action	Min (sec)	Median (sec)	90%ile (sec)	95%ile (sec)	Max (sec)	Avg (sec)	Success	Count
total	0.003	0.021	0.286	0.423	8.396	0.106	100.0%	85560

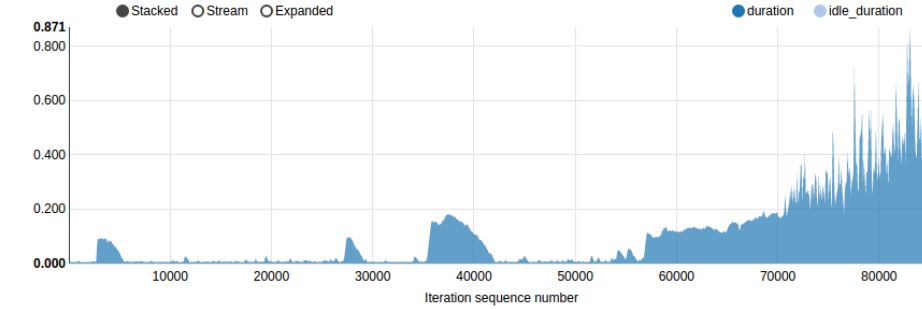


Рис. 3 Результаты тестирования нашего сервиса при N=711 и N=713. На оси ординат обозначено время отклика в секундах.

5. Вывод

Абсолютно на всех перечисленных конфигурациях найденное значение N для Keystone оказывалось значительно меньше, чем на реализованном нами простом сервере. Такое низкое значение N может быть следствием проблем в бизнес-логике Keystone или же неправильной работы с другими компонентами. Выявление причины такого поведения системы является темой для дальнейших исследований. В будущем мы также намерены улучшить модель тестирования таким образом, чтобы производительность Rally не

влияла на работу тестируемого сервера. Для этого планируется подобрать более справедливую конфигурацию для Rally, а также разнести Rally и сервер на разные физические машины.

Список литературы

[1]. Официальная страница проекта OpenStack— https://www.openstack.org/
[2]. M. Bist, M. Wariya, A. Agarwal. Comparing Delta, Open Stack and Xen Cloud Platforms: A Survey on Open Source IaaS. 3rd IEEE International Advance Computing Conference (IACC) , 2013
[3]. T. Rosado, J. Bernardino. An overview of openstack architecture. Proceedings of the 18th International Database Engineering & Applications Symposium. ACM, 2014.
[4]. Keystone project web page — http://docs.openstack.org/developer/keystone/
[5]. Alexander Maretskiy. Finding a Keystone bug while benchmarking 20 node HA cloud performance at creating 400 VMs, December 15 2015. (https://rally.readthedocs.org/en/0.1.1/stories/nova/boot_server.html)
[6]. Neependra Khare. 4x performance increase in Keystone inside Apache using the token creation benchmark, December 15 2015. (https://rally.readthedocs.org/en/0.1.1/stories/keystone/authenticate.html)
[7]. Официальная страница проекта Rally — https://wiki.openstack.org/wiki/Rally
[8]. Официальная страница проекта Ansible — http://www.ansible.com/
[9]. Официальная страница проекта Flask — http://flask.pocoo.org/

A Performance Testing and Stress Testing of Cloud Platform Central Identity: Openstack Keystone Case Study[2]

[1]I. V. Bogomolov <igor95n@gmail.com>
[1]A. Aleksiyants <aleksiyantsa@gmail.com>
[1]A. Sher <sher-ars@ispras.ru>
[1]O. Borisenko <al@somestuff.ru>
[1,2,3]A. Avetisyan <arut@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia

[2] RFBR 15-29-07111 ofi_m

Abstract. Nowadays OpenStack platform is a leading solution in cloud computing field. Keystone, the OpenStack Identity Service, is one of its major components. This service is responsible for authentication and authorization of users and services of the system. Keystone is a high-load service since all interactions between services happen through it. This leads us to the following problem: increasing the number of cloud service users results to significant load growth. In this paper we demonstrate the problem of Keystone performance degradation during constant load. In order to find source of the problem we have tested Keystone with different backends (PostgreSQL, MariaDB), frontends (Apache2, ngnix) and keeping the database on different hardware (HDD, SSD and tmpfs on RAM). Using different configuration sets is necessary because it helps us to narrow the amount of possible reasons causing degradation. Tests were conducted with Rally. As a result, in all test cases we have seen inadequate quick degradation under relatively light load. We have also implemented a mock service which represents the simplest Keystone tasks. Our service turned out to be much faster than Keystone. The problem with Keystone might be related to either its internal logic implementation or incorrect interaction with other components; it is the subject of further research.

Keywords: OpenStack, Keystone, Rally

DOI: 10.15514/ISPRAS-2015-27(5)-4

For citation: Bogomolov I.V., Aleksiyants A., Sher A., Borisenko O., Avetisyan A. A Performance Testing and Stress Testing of Cloud Platform Central Identity: Openstack Keystone Case Study. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 49-58 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-4.

References

[1]. OpenStack project web page — https://www.openstack.org/
[2]. M. Bist, M. Wariya, A. Agarwal. Comparing Delta, Open Stack and Xen Cloud Platforms: A Survey on Open Source IaaS. 3rd IEEE International Advance Computing Conference (IACC) , 2013
[3]. T. Rosado, J. Bernardino. An overview of openstack architecture. Proceedings of the 18th International Database Engineering & Applications Symposium. ACM, 2014.
[4]. Keystone project web page — http://docs.openstack.org/developer/keystone/
[5]. Alexander Maretskiy. Finding a Keystone bug while benchmarking 20 node HA cloud performance at creating 400 VMs, December 15 2015. (https://rally.readthedocs.org/en/0.1.1/stories/nova/boot_server.html)
[6]. Neependra Khare. 4x performance increase in Keystone inside Apache using the token creation benchmark, December 15 2015. (https://rally.readthedocs.org/en/0.1.1/stories/keystone/authenticate.html)
[7]. Rally project web page — https://wiki.openstack.org/wiki/Rally
[8]. Ansible project web page — http://www.ansible.com/
[9]. Flask project web page — http://flask.pocoo.org/

Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя

В.К. Кошелев <vedun@ispras.ru >
И.А. Дудина <eupharina@ispras.ru >
В.Н. Игнатьев <valery.ignatyev@ispras.ru>
А.И. Борзилов <helendile@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В данной работе рассматривается построение масштабируемого чувствительного к путям анализа дефектов в программах на языке C#. Предложенный метод анализа является адаптацией набора подходов, используемых в инструментах для статического анализа C/C++-программ Saturn, Calysto и Svace. Особое внимание уделяется связи рассматриваемой модели с реальным выполнением программы. Производится формализация дефекта разыменования нулевого указателя и сведение задачи поиска данного дефекта к задаче выполнимости формул логики предикатов. Для проведения формализации вводится модельный язык, на который может быть оттранслирована любая программа на языке C#. Язык представляет собой подмножество языка C#, избавленное от синтаксического сахара. Для упрощения формализации из числовых типов рассматривается только один целочисленный тип. Для решения проблемы анализа циклов используется метод развертки графа потока управления. Предлагается модель абстрактного состояния программы, описывающая множество возможных состояний в данной точке программы. Абстрактное состояние задается состоянием памяти программы, описанным с помощью набора неизвестных входных переменных, и предикатом пути над тем же набором переменных. Для каждой инструкции модельного языка определяются формальная семантика и передаточная функция, отражающая изменение абстрактного состояния в соответствии с семантикой. Каждая функция считается точкой входа в программу, а межпроцедурный анализ основан на методе резюме. Резюме строятся по результатам внутрипроцедурного анализа функций. Поиск дефектов происходит при помощи добавления дополнительной информации к абстрактному состоянию. Решение о выдаче предупреждения принимается на основе выполнимости формулы, описывающей условие возникновения ошибки. Предложенный метод реализован в анализаторе SharpChecker, разработанном в ИСП РАН. Приведены результаты тестирования, подтверждающие применимость метода для анализа промышленных программ.

Ключевые слова: Статический анализ; разыменование нулевого указателя; чувствительность к путям; резюме функции; поиск дефектов.

DOI: 10.15514/ISPRAS-2015-27(5)-5

Для цитирования: Кошелев В.К., Дудина И.А., Игнатьев В.Н., Борзилов А.И. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59-86. DOI: 10.15514/ISPRAS-2015-27(5)-5.

1. Введение

Автоматический поиск дефектов в программах является одной из важнейших задач программной инженерии. В рамках данной работы рассматривается задача автоматического поиска дефектов в программах на языке программирования C#. Для поиска дефектов используется метод статического анализа исходного кода программ. К анализируемой программе предъявляется единственное требование: она должна успешно компилироваться. Для промышленного применения анализатора он должен поддерживать как полный анализ проекта за время, ограниченное временем ночной сборки, так и инкрементальный анализ небольших изменений, например на лету при разработке в интегрированной среде. Такой анализ дефектов должен за отведенное ему время находить как можно больше срабатываний, сохраняя при этом процент истинных срабатываний на приемлемом уровне. Учитывая имеющиеся ограничения на время работы анализатора, фактически время анализа одной функции не должно превышать нескольких десятков секунд.

В отличие от языков C/C++, язык C# не подвержен некоторым характерным типам уязвимостей, как переполнение буфера и уязвимость форматной строки, с помощью которых осуществляется большинство атак, приводящих к исполнению произвольного кода. Однако, большее внимание требуется уделять специфичным типам эксплуатируемых ошибок, таким как утечка ресурсов, которые могут привести, например, к отказу в обслуживании.

При анализе, дефект формулируется либо с помощью конструкции абстрактного синтаксического дерева, либо с помощью ошибочного пути выполнения программы. Между данными формулировками существует принципиальная разница: поиск конструкции абстрактного синтаксического дерева алгоритмически разрешим, а точный и полный анализ путей выполнения в общем случае неразрешим.

Промышленные анализаторы ищут оба вида дефектов, поэтому они применяют два типа анализа: один используется для поиска потенциально опасных конструкций, другой – для анализа путей выполнения. Задачей анализа второго типа является поиск таких дефектов, как разыменование невалидного указателя, переполнение буфера, утечка памяти, неверное использование примитивов синхронизации, применение невалидных дескрипторов и т.д. Данная работа посвящена поиску дефектов второго типа.

Для их поиска предлагается использовать внутрипроцедурный чувствительный к путям анализ. Нарушения контракта использования функций при внутрипроцедурном анализе функций будем моделировать с помощью резюме. Использование резюме является неизбежным компромиссом между производительностью и качеством анализа. С одной стороны, суммарное время, необходимое на применение резюме намного меньше времени повторного анализа в каждой точке вызова. С другой стороны, резюме отражает только некоторые эффекты вызова функции, поэтому такое моделирование может быть причиной как ложных срабатываний, так и пропуска ошибок. Кроме того, с помощью резюме можно просто реализовать инкрементальный анализ.

Идея реализации чувствительного к путям анализа заключается в построение для рассматриваемых путей формулы логики предикатов над значениями переменных, отражающей истинность условий переходов, характеризующие данные пути. В том случае, если на рассматриваемых путях произошла ошибка и построенная формула пути выполнима, то данный путь может быть представлен пользователю в качестве примера пути, на котором происходит ошибка. Выполнимость формулы пути проверяется с помощью SAT-решателей или SMT-решателей. Анализ путей выполнения осуществляется с использованием точек слияния, задающих состояние одновременно нескольких путей выполнения. Использование точек слияния позволяет избежать комбинаторного взрыва, возникающего при переборе всех возможных путей даже в случае отсутствия циклов.

В предлагаемом методе анализ функций производится в обратном топологическом порядке графа вызовов и сопровождается построением резюме для каждой проанализированной функции. Таким образом при анализе очередной функции для всех вызываемых ею функции их резюме уже построены. Рекурсия при данном подходе игнорируется за счет удаления обратных ребер графа вызова. При внутрипроцедурном анализе каждая функция считается точкой входа в независимую подпрограмму, поэтому её входные параметры могут принимать произвольные значения.

Так как в данном методе параметры функции являются неизвестными значениями, то в формуле пути они участвуют как свободные переменные. Значения этих переменных в случае наличия модели являются значениями входных переменных, при которых произойдет ошибка. Так как параметры функции могут использоваться для доступа к куче, то для проведения анализа необходимо некоторым образом задать состояние кучи в точке входа анализируемой функции. Для этого в данной работе рассматривается формализация, позволяющая сопоставить начальному состоянию кучи и входным параметрам набор независимых переменных, которые в дальнейшем используются при построении формул пути.

Для поиска ошибок в проверяемой точке программы рассматривается формула над значениями переменных, выполнимая в случае, если ошибка

возможна. Например, для разыменования нулевого указателя можно рассмотреть формулу выполнимую, если в данной переменной может быть записан null. Тогда для поиска разыменования нулевого указателя достаточно убедиться, что в точке разыменования переменной условие того, что ее значения равно null, и формула пути до текущей точки совместны. Тогда модель, на которой они совместны, можно предъявить пользователю в качестве примера входных данных, приводящих к разыменованию нулевого указателя.

Особенную сложность при анализе программ на языке C# представляет собой полная поддержка следующих конструкций: исключения, виртуальные вызовы, асинхронные вызовы, лямбда-выражения и некоторые другие особенности языка. Особенности поддержки данных конструкций в рамках предлагаемого метода планируется рассмотреть в дальнейших работах.

Данный метод был реализован в анализаторе CSCC, разработанном в ИСП РАН, и получил практическое подтверждение.

Текст работы организован следующим образом. Во второй части производится формализация понятий входных данных и пути выполнения. При помощи введенных понятий формулируется ситуация разыменования нулевого указателя. В третьей части приводится описание внутрипроцедурного анализа, позволяющего находить разыменования нулевого указателя. В четвертой части рассматривается применение резюме функции для реализации межпроцедурного анализа. Пятая часть посвящена особенностям реализации предложенного метода. В шестой части описываются результаты применения данного метода для анализа промышленных проектов. Седьмая часть посвящена обзору схожих решений. В заключительной части подводятся итоги данной работы.

2. Постановка задачи

В данной работе предлагается рассматривать программы на модельном языке вместо программ на языке C#. Такое решение мотивировано тем, что реализованный анализатор не имеет собственного внутреннего представления, которое можно было бы использовать в качестве модельного языка, используя вместо инструкций в графе потока управления вершины абстрактного синтаксического дерева, соответствующие операторам языка C#. Введение модельного языка позволяет избежать избыточности, вызванной наличием в языке C# синтаксического сахара, и исключить не полностью поддерживаемые анализатором конструкции языка. Будем считать, что программа, написанная на подмножестве C#, поддерживаемым анализатором, может быть однозначно оттранслирована в программу на модельном языке с сохранением семантики.

2.1 Модельный язык

В модельном языке присутствует только два типа: integer — 32–битное знаковое число и object — ссылка на объект, находящийся в динамической памяти. К переменным типа integer могут применяться арифметические и битовые операции. К переменным типа object применяются операции сравнения и взятия поля. Определение поля для конкретной ссылки на объект в памяти, как и в скриптовых языках, происходит в момент первого присваивания в него значения. В реальных программах на языке C# используются и другие целочисленные типы, однако для просты повествования ограничимся только типов integer.

Рассмотрим грамматику модельного языка, записанную в БНФ. Выражение < текст > обозначает строковый литерал, а символ <> означает отсутствие каких-либо символов. Сразу оговоримся, что, так как рекурсия исходным анализатором не поддерживается, в модельном языке она также запрещена.

- *identifier* ::= строка
- *const* ::= число |*null*
- *type* ::= *integer* | *object*
- *var* ::= *identifer*;
- *field* ::= *identifer*;
- *param* ::= type identifer;
- *assignConst* ::= *var* = *const*;
- *assign* ::= *var* = *var*;
- *setfield* ::= *var.field* = *var*;
- *getfield* ::= *var* = *var.field*;
- *binop* ::= *var* = *var* ◇ *var*;
- *unop* ::= *var* = □*var*;
- *if* ::=< *if* > (*var*){*statements*}|
  < *if* > (*var*){*statements*} < *else* > {*statements*}
- *loop* ::=< *do* > {*statements*} < *while* > (*var*);
- *control* ::=< *break* > | <*continue* > | < *return* > *var*;
- *statement* ::= *assign* | *assignConst* | *setfield* |
  getfield | *binop* | *unop* | *if* | *loop* | *control* | *functioncall*

- *statements* ::= *statement*|*statementstatements*| <>
- *paramlist* ::= *param, paramlist* | *param* | <>
- *arglist* ::= *var* | *var, arglist* | <>
- *functioncall* ::= *var* = *identifer*(*arglist*);
- *function* ::= *type identifier*(*paramlist*){*statements*}
- *program* ::= *function* | *function program*

Определение переменных в данном языке также происходит «лениво», в момент первого присваивания значения в данную переменную. Тип переменной выводится из типа правой части присваивания. Аналогично определяются поля объектов. В данном языке все функции возвращают значение. Если в исходной программе функция имела тип void, то будем считать, что она возвращает null. В модельном языке отсутствуют глобальные переменные, т.к. в программах на языке C# они, как правило, не используются. В операторах if и loop, в качестве операндов допускаются только переменная типа integer, переход по истинной ветке осуществляется в том случае, если значение переменной отлично от нуля. Из циклов в данном языке представлен только цикл *do...while*. Данное решение принято для того, чтобы при построении графа потока управления выход из цикла мог осуществляться без прохождения по обратному ребру.

Построим для каждой функции на модельном языке граф потока управления. Вершинами графа являются операторы, определяемые правилами *assign*, *assignConst*, *setfield*, *getfield*, *binop*, *unop* и *functioncall*. Далее будем называть данные операторы инструкциями. Для учета условий переходов добавим специальную операцию «*assume* ::=< *assume* > (*var*);», продолжающую выполнение только в случае, если значение переменной *var* отличается от нуля. Для всех вершин графа потока управления, имеющих двух потомков, потребуем, чтобы ребра вели в инструкции *assume* с взаимоисключающими условиями, соответствующими исходному условию перехода.

Символом $\diamond$ обозначается некоторый бинарный оператор, а символом $\square$ - некоторый унарный оператор.

2.2 Конкретные состояния

При анализе будем считать, что каждая функция на модельном языке может являться точкой входа в программу. На возможные значения её параметров не накладывается никаких ограничений кроме того, что все ссылочные значения указывают на разные объекты. Для описания входных значений функции определим следующие множества:

- W — множество переменных, определяемых в функции;

- $P, P \subset W$ — множество параметров функции;
- N — множество значений типа integer;
- R — множество значений типа object, включая *null*;
- $V = N \cup R$ — множество значений;
- $F = F_N \cup F_R$ — множество полей, по которым может производиться взятие поля;
- F_N, F_R – множества полей типа integer и типа object соответственно.

Тогда состояние программы в момент входа в анализируемую функцию можно задать следующей парой отображений:

- $Params : P \to V$ — значения параметров в момент вызова функции;
- $HEAP : R \times F \rightharpoonup V$ — задание значений кучи.

Данную пару отображений будем называть начальным состоянием, множество всех возможных начальных состояний определим как *EState*. Состояние программы в конкретной точки анализируемой функции данной парой отображений:

- $Vars : W \to V \cup \perp$ — значения переменных; символ $\perp$ означает, что переменная не была инициализирована;
- $HEAP : R \times F \rightharpoonup V$ — задание значений кучи.

Вторую из заданных пар отображений будем называть конкретным состоянием, множество всех возможных конкретных состояний обозначим как *CState*. Нетрудно заметить, что состояние программы в момент входа в функцию является частным случаем конкретного состояния, в котором $Vars(v) \mapsto \perp, \forall v \in W \setminus P$.

Если выполнение очередной инструкции возвращает управление, будем считать, что инструкция определяет отображение $CState \to CState$ в соответствии с семантикой данной инструкции. Особенно важно, что все функции в данном модельном языке являются чистыми, т.е. их выполнение определяется конкретным состоянием в момент вызова, поэтому, если они вернут управление, их действие также будет задаваться некоторым отображением $CState \to CState$.

Данное отображение для инструкции *instr* будем обозначать как $\xrightarrow{instr}$.

Рассмотрим некоторое начальное состояние $estate \in EState$. Тогда последовательно применяя отображения $\xrightarrow{instr}$ для очередной инструкции к текущему конкретному состоянию, получим некоторый путь выполнения.

Если у вершины есть два исходящих ребра, то, из-за взаимоисключающих условий последующих инструкций *assume*, ребро, по которому продолжится выполнение, всегда выбирается однозначно.

Тогда каждому $estate \in EState$ можно поставить в соответствие некоторый, возможно бесконечный, путь выполнения, задающийся последовательностью пар $\langle cstate_i, e_i \rangle$, для которых верно, что:

- последовательность e_i образует путь в графе потока управления – $cstate_0 = estate$, e_0 – ребро, выходящее из точки входа.
- $cstate_i \xrightarrow{instr} .cstate_{i+1}$, где $instr(e)$, инструкция в вершине, в которую входит ребро e.

Данную последовательность пар будем называть конкретным путем. Как видно из построения, каждое начальное состояние определяет возможно бесконечный набор вложенных друг в друга конкретных путей выполнения.

Тогда сформулируем задачу поиска дефекта как задачу поиска конкретного пути выполнения, на котором произошла ошибка. Для эффективного решения данной задачи необходимо ограничить длину рассматриваемых путей, причём достаточно ограничить количество вхождений обратных ребер в рассматриваемые конкретные пути выполнения некоторой константой k. Поэтому вместо графа потока управления предлагается рассматривать ациклический граф развертки, полученный из графа потока управления.

2.3 Развертка графа потока управления

Построим бесконечную развертку $UnRoll = \langle V_u, E_u \rangle$ графа потока управления $G = \langle V, E \rangle$. Множество обратных рёбер обозначим как $BE, BE \subset E$. Введем операцию *Reachable*, которая для графа потока управления G и вершины $v \in V$ строит новый граф $G' = \langle V', E' \rangle$, изоморфный подграфу G, состоящему из вершин и ребер, достижимых из v только по прямым ребрам. Изоморфизм между вершинами G' и G задается отображением $T : V' \to V$.

Первую компоненту данной развертки построим как $Comp_1 = Reachable(entry)$, где *entry* – точка входа в CFG, соответственно, T_{Comp1} обозначает соответствие между вершинами $Comp_1$ и CFG. Рассмотрим алгоритм построения из данной компоненты графа развертки $Comp$, достижимого из неё компонент. Для этого рассмотрим все вершины $Comp = \langle V_{comp}, E_{comp} \rangle$, такие, что $v \in V_{comp} \langle T_{Comp}(v), u \rangle \in BE$. Таким образом, каждой из рассматриваемых вершин было поставлено в соответствие некоторое ребро $be \in BE$, обозначим множество данных ребер как BE_{comp}. Пронумеруем рассматриваемые обратные ребра в порядке топологической сортировки, i-тое ребро данного множества обозначим как $be_i = \langle T(v_i), u_i \rangle$. Тогда компоненту $Comp_i$ из компоненты $Comp$ получим применением операции R к вершине u_i. Соответственно, добавим к графу $UnRoll$ граф $Reachable(u_i)$ и соединим его с компонентой $Comp$ при помощи

ребра $\langle v_i, entry \rangle$, где *entry* – точка входа в граф $R(u_i)$. Ребро $\langle v_i, entry \rangle$ соответствует ребру be_i.

На Рисунке 1 изображен пример применения развертки к графу потока управления. Обратные ребра обозначены пунктирными линиями. Имена компонент содержат полную информацию о том, как данная компонента была построена. Например, имя «Компонента 1.2» означает, что данная компонента была построена из компоненты «Компонента 1» при помощи обратного ребра номер 2 в топологическом порядке.

Введем понятие глубины для графа развертки. Первая компонента графа развертки имеет глубину ноль. Если компонента графа развертки построена из компоненты глубины n, то она имеет глубину $n + 1$. Таким образом, для того чтобы ограничиться конкретными путями, проходящими по обратным ребрам не более k раз, достаточно рассмотреть развертку, содержащую все компоненты глубиной не более k.

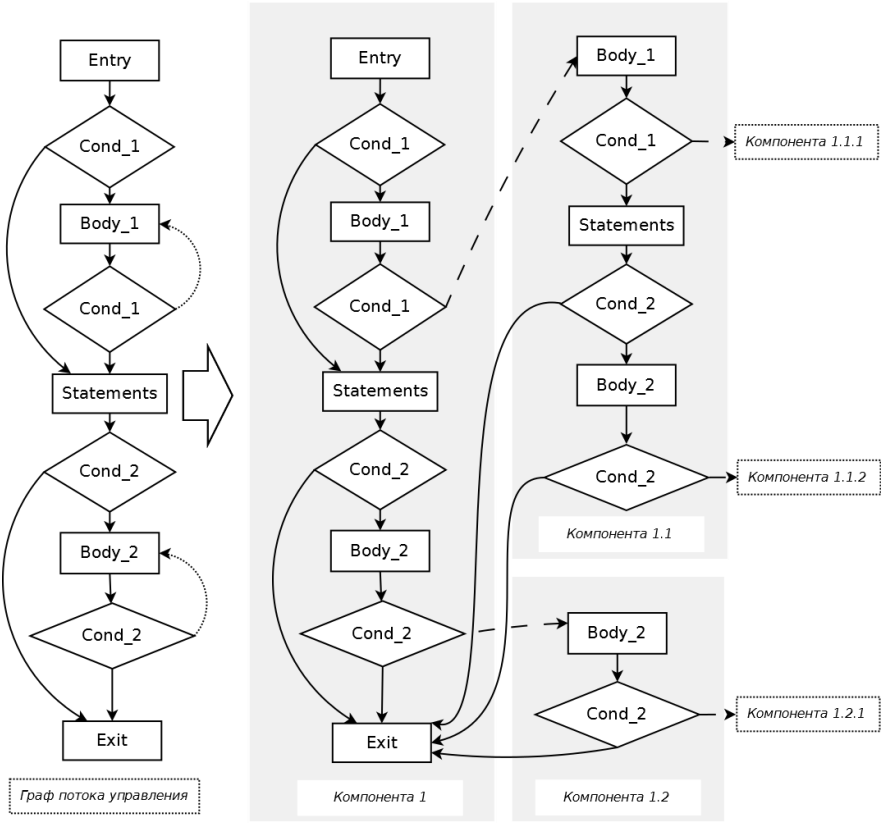


Рис. 1 — Развертка графа

Переход от путей на графе потока управления является аппроксимацией снизу, т.к. происходит ограничение рассматриваемых путей. Данная аппроксимация уменьшает множество ошибок, которые можно обнаружить с помощью анализа.

Ограничение на рассматриваемые пути действует также для вызовов функций. Теперь отображение *CState* → *CState* задается только для таких *cstate*, что вызываемая функция возвращает управление, пройдя не более, чем через *k* обратных ребер в вызываемой функции.

2.4 Определения понятия ошибки

При поиске ошибок нулевого указателя наибольший интерес будут представлять следующие две ситуации: во-первых, разыменование значения, полученного из присваивания переменной константы *null*, и во-вторых, взятие поля у входного параметра функции после успешного сравнения его на равенство с константой *null*. Выразим данные ситуации на языке конкретных путей.

Для этого введем два значения для обозначения нулевого указателя: *null* и *nil*. Данные значения являются эквивалентными в том смысле, что *null == nil* вычисляется в истину. Тем не менее, в начальном состоянии могут быть использованы только значения *nil*, а все константы, используемые в программе, имеют значение *null*. Задачу поиска ошибки вида разыменования нулевого указателя разделим на две подзадачи. Во-первых, будем искать ситуации, в которых на некотором конкретном пути значение *null* используется для взятия поля. С разыменованием значения *nil* ситуация обстоит сложнее. Если во всех случаях разыменования *nil* сообщать о разыменовании нулевого указателя, то количество ложных срабатываний будет чрезмерно велико, т.к. скорее всего анализируемая функция обладает неявным неизвестным контрактом, который запрещает данному параметру принимать значение *nil*. Поэтому предлагается выдавать предупреждение только в том случае, когда находится такой конкретный путь, что значение *nil* становится разыменованным после сравнения именно этого значения с *null*. Рассмотрим следующий пример:

```
string foo ( object a , object b) {
if (a == null && b == null ) return "";
return b.ToString( ) ;
}
```

С точки зрения данных выше определений в данном примере отсутствует ошибка разыменования нулевого указателя, т.к. поле входного параметра *b* берется только в том случае, когда *b* не участвует в сравнении с *null*.

3. Алгоритм поиска ошибок

Идея алгоритма поиска ошибок достаточно проста. Рассмотрим некоторую инструкцию взятия поля в графе развертки. Данная инструкция достижима на некотором наборе входных состояний. Если в данном наборе найдется такое входное состояние, что, когда соответствующий ему конкретный путь доберется до данной инструкции, в разыменованной переменной будет находиться значение *null*, и произойдет ошибка.

Для поиска ошибок необходимо ввести описание начальных состояний, из которых может быть достигнута конкретная точка развертки. В данном разделе рассматривается только внутрипроцедурный анализ. Межпроцедурному анализу посвящен следующий раздел.

Рассмотрим входные параметры функции. Путем доступа назовем последовательность вида $p.f_1.f_2....f_n$, где $p \in P, f_i \in F_R$, $f_n \in F, 0 \le i < n$, $n \ge 0$. Данная последовательность описывает доступ к куче через параметр функции p. Тип пути доступа будем определять по типу последнего поля. Так как развертка является конечной, то количество путей доступа, начинающихся с параметров функции, к которым может осуществляться доступ, также является конечным. Обозначим описанное множество путей доступа как AP. Пусть $|AP| = m$, тогда пронумеруем пути доступа из AP от 1 до m. Каждому пути доступа ap_i поставим в соответствие некоторую символьную переменную x_i, тип которой совпадает с типом пути доступа. Если символьная переменная x_i имеет тип integer, то она может принимать любое значение данного типа. Если же символьная переменная x_i имеет тип object, то она может принимать только два значения: *null* или r_i, где $r_i \in R$. Все r_i не равны один другому.

По аналогии с конкретным начальным состоянием, введем символьное начальное состояния. Пусть X — множество символьных переменных, поставленных в соответствие путям доступа AP. Тогда символьным начальным состоянием назовем пару отображений:

- $RV_S : P \to X$ — отображение параметров функции в соответствующие символьные переменные;
- $\mathrm{HEAP}_S : R \times F \rightharpoonup X$ — состояние памяти в момент входа в функцию.

Отображение RV_S строится по путям доступа, не содержащих обращения к полям. Рассмотрим все $ap_i = p$, $p \in P$, тогда $RV_S(p) \mapsto x_i$.

Отображение HEAP_S строится, исходя из связей между путями доступа. Если пути доступа ap_i и ap_j такие, что $ap_i.f = ap_j$, то $\mathrm{HEAP}_S(\langle r_i , f \rangle) \mapsto x_j$. Тогда, подставляя вместо символьных переменных конкретные значения, получим множество начальных состояний, задаваемое данным символьным состоянием.

Дальнейший поиск ошибок будем проводить только среди конкретных начальных состояний, задаваемых символьным состоянием. Потери истинных

срабатываний не произойдет, поскольку построенные таким образом состояния содержат все варианты входных данных для рассматриваемой развертки.

Для описания всех начальных состояний, соответствующих данной вершине графа развертки, можно воспользоваться формулами, построенными над символьными переменными. Данные формулы будем называть символьными выражениями. Так же как и символьные переменные, символьные выражения могут иметь либо тип integer, либо тип object. Построение символьных выражений задается следующими правилами:

- все числовые константы и *null* являются символьными выражениями;
- все символьные переменные являются символьными выражениями;
- если x и y — символьные выражения типа integer, то $x \diamondsuit y$ символьное выражение типа integer; если x — символьное выражение типа integer, то $\square x$ также является символьным выражением типа integer;
- если x, y – символьные выражения одного типа, а *cond* – символьное выражение типа integer, то $ite(c,x,y)$ – символьное выражение того же типа. Функция *ite* является условным оператором и возвращает свой второй аргумент, если первый не равен нулю, или третий аргумент, если первый аргумент равен нулю.

Множество всех символьных выражения обозначим как *SE*. По аналогии с конкретным состоянием, введем понятие символьного состояния, определяемого следующим образом:

- $RV_S : W \to SE$ — отображение переменных функции в соответствующие символьные выражения;
- $HEAP_S : R \times F \rightharpoonup SE$ — текущее состояние памяти функции.
- $\mathcal{G}, \quad \in SE$ — символьное выражения, задающее ограничения на символьные переменные.

Как и в случае с конкретными состояниями, начальное символьное состояние является частным случаем символьного состояния, в котором $\mathcal{G} = \top$.

Определим правила вывода, позволяющие из начального символьного состояния построить символьные состояния для всех вершин графа развертки с учетом семантики инструкций. Символьное состояние будем обозначать тройкой $\langle RV,HEAP,\mathcal{G}\rangle$, начальное символьное состояние обозначим как Г. Обозначим e_1,e_2 два последовательных ребра развертки, такие что $e_1 = \langle v,u\rangle, e_2 =$

$\langle u,w\rangle$. Инструкцию, содержащуюся в вершине, на которую указывает ребро e, обозначим $\mathcal{I}(e)$.

$$\frac{\langle \Gamma,\ e_1\rangle \vdash \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(\ e_1) = (\text{"}a\ =\ b;\text{"})}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ RV\ (b)\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ const;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ const\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ b \diamondsuit c;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ RV\ (b) \diamondsuit RV\ (c)\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ \square b;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ \square b\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle\ \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}assume(c);\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV, HEAP, \mathcal{G}\ \wedge\ RV\ (c)\rangle)}$$

Здесь и далее операция $\uplus$ обозначает переопределение отображения для заданных значений:

$$(A\ \uplus\ B)(x)\ = \begin{cases} B(x), \text{если } x\ \in\ dom(B); \\ A(X), \text{иначе} \end{cases}$$

Введём операцию $PT : SE \rightarrow 2^{R\times SE}$, которая для символьного выражения типа object строит пары из ссылки и условия, при котором данное выражение равно данной ссылке. Все условия являются попарно несовместными. Также введём операцию разыменования DEREF : $2^{R\times SE} \times f \rightarrow SE$, которая определяется следующим образом:

$$pt = \{ri,ci\},\ |pt| = n,\ pt \in 2R\times SE\ vi = HEAP(ri,f)$$

$$DEREF(pt,f) = \begin{cases} ite\left(c_1, v_1, ite\left(c_2, v_2, \ldots\ ite(c_{n-1}, v_{n-1}, v_n)\right)\right), \text{для } n > 1 \\ v_1, \text{для } n = 1 \end{cases}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ b.f;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ DEREF(PT(RV\ (b)), f)\}, HEAP, \mathcal{G}\ \wedge\ RV\ (b)\ \neq\ null\ \rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a.f\ =\ b;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV, HEAP\ \uplus_{i=1}^{n} \left\{\{s_i, f\}\ \mapsto \text{ite}\left(c_i, RV(b), HEAP(si, f)\right)\right\}, \mathcal{G}\ \wedge\ RV(a) \neq null\ \rangle}$$

Определим правило вывода для точки слияния. Будем считать, что в графе потока управления точки слияния выделены в отдельные вершины, которые

имеют входящую степень два. Будем считать, что все остальные вершины имеют степень один. Для точек слияния определим условия того, по какому из ребер пришло управление. Для этого воспользуемся подходом, аналогичным построению Gated SSA [1]. Таким образом, получим символьное выражение *cond*, которое точно определяет ребро, по которому управление пришло в точку слияния.

Будем считать, что, если условие *cond* верно, переход произошел по ребру e_l, иначе – по ребру e_r. Ребро, ведущее из точки слияния, обозначим как e_j. Тогда его символьное состояние задается следующим образом:

$$\frac{\langle \Gamma, e_{\mathrm{l}}\rangle \vdash \langle RV_{\mathrm{l}}, HEAP_{\mathrm{l}}, \mathcal{G}_{\mathrm{l}}\rangle \quad \langle \Gamma, e_{\mathrm{r}}\rangle \vdash \langle RV_{\mathrm{r}}, HEAP_{\mathrm{r}}, \mathcal{G}_{r}\rangle \quad \mathcal{I}(e_{\mathrm{l}}) = \mathcal{I}(e_{\mathrm{r}}), \quad e_{\mathrm{l}} \neq e_{\mathrm{r}}}{\begin{gathered}\langle \Gamma, e_{\mathrm{j}}\rangle \vdash \langle \cup\, v\{v \mapsto ite(cond, RV_{\mathrm{l}}(v), RV_{\mathrm{r}}(v))\},\\ \cup\, \langle sr, f\rangle\{\langle sr, f\rangle \mapsto ite\big(cond, HEAP_{\mathrm{l}}(sr, f), HEAP_{\mathrm{r}}(sr, f)\big)\},\\ \mathcal{G}_l \vee \mathcal{G}_r\rangle\end{gathered}}$$

Для реализации различных проверок будем использовать следующий механизм пометок. К каждому символьному выражению, использующемуся в программе, алгоритм проверки может прибавить или убрать некоторую пометку, установленную для некоторого множества начальных состояний. Данное действие можно формализовать, считая для каждого ребра графа развертки отображение T : SE×*TAG* → SE, причём множество *TAG* обозначает множество пометок, используемых анализами. Данное отображение ставит соответствие между парой «символьное выражение, тип пометки» и условием, при котором данное символьное выражение будет помечено. Отображение является частичным, отсутствие соответствующего элемента для пары ⟨*se*,*tag*⟩ означает отсутствие пометки *tag* для выражения *se*.

При помощи данного отображения произвольный анализ дефектов может доопределить правила вывода для передачи пометок через инструкции, таким образом, возможна реализация поиска произвольных дефектов. Зачастую для объединения пометок может быть использовано следующее правило вывода:

$$\frac{\langle \Gamma, e_{\mathrm{l}}\rangle \vdash \langle se, tag\rangle \mapsto c_l \quad \langle \Gamma, e_{\mathrm{r}}\rangle \vdash \langle se, tag\rangle \mapsto c_r \quad \mathcal{I}(e_{\mathrm{l}}) = \mathcal{I}(e_{\mathrm{r}}), \quad e_{\mathrm{l}} \neq e_{\mathrm{r}}}{\langle \Gamma, e_{\mathrm{j}}\rangle \vdash \langle se, tag\rangle \mapsto c_l \vee c_r}$$

Так же как и ранее, *cond* обозначает критерий выбора ребра. Если отображение отсутствует по одной из веток объединения, но присутствует по другой, будем считать, что его условием, в случае отсутствия, является ложь.

Дополнительно потребуем согласованности пометок для операции *ite*, иными словами:

$$\forall a,b,c \in \mathrm{SE}, \forall t \in TAG, T(ite(c,a,b),tag) = T(a,t) \wedge c \vee T(b,c) \wedge \neg c$$

Далее будем считать, что условия пометок для *ite* выражений определяются автоматически по приведённому выше правилу.

Для поиска ошибок вида разыменования нулевого указателя достаточно ввести три вида пометок. Пометку *nullable* будем использовать для обозначения того, что выражение равно *null*. Данную пометку достаточно разместить только на значение *null*. Для всех производных *ite* выражений значение пометки определится автоматически.

Тогда для инструкции, берущей поле у символьного выражения *se*, верно, что все решения $T(se,nullable) \wedge \mathcal{G}$ задают начальные конкретные состояния, на которых произойдет разыменования нулевого указателя. Для поиска ошибок разыменования нулевого указателя после сравнения с *null* необходимо ввести два дополнительных типа пометок: *compared* и *deref*. Соответственно, пометкой *compared* будем помечать такие выражения, которые были сравнены с *null, deref* — те, что были разыменованы.

Ошибка для пометки *compared* определяется точно так же, как и для пометки *nullable*. Пометка *deref* используется при межпроцедурном анализе.

4. Межпроцедурный анализ

Для проведения межпроцедурного анализа при построении начального символьного состояния необходимо рассматривать пути доступа, по которым происходят обращения в вызванных функциях. Ввиду отсутствия рекурсии и циклов, данное множество путей доступа также будет конечным. Поэтому, учитывая его при построении начального состояния, анализ вызываемых функций можно проводить при помощи встраивания. Однако подход встраивания функций не масштабируется на промышленные приложения, поэтому вместо встраивания функций предлагается использовать резюме.

4.1 Резюме функции

Для построения резюме функции воспользуемся результатами, полученными при её анализе. Будем предполагать, что каждая функция имеет одну точку выхода. Тогда в качестве резюме функции возьмем символьное состояние, полученное в точке выхода. Таким образом, полученное резюме будет описывать состояние после выполнения функции, при условии, что все входные ссылки не являются псевдонимами друг друга.

Для межпроцедурного поиска дефектов необходимо также сохранить в резюме условия пометок *nullable* и *deref*, при этом важно, что значение пометок *compared* сохранять не требуется, т.к. наличие сравнения с *null* символьного значения внутри вызываемой функции не говорит ничего о возможности равенства нулю данного значения в контракте вызывающей функции.

Однако ввиду того, что результат применения резюме при вызове некоторой функции входит в резюме вызывающей функции, существует проблема

разрастания размера резюме при анализе промышленных проектов. Для решения данной проблемы предлагается ограничить максимальный размер сохраняемого резюме. При реальной реализации больше всего памяти требуется для сохранения символьных выражений, поэтому предлагается ограничить количество символьных выражений, которые войдут в резюме. Будем считать, что размер сохраняемых символьных выражений ограничен константой *M*. Тогда вместо символьных выражений, не вошедших в резюме, будем использовать новые символьные переменные. Если в резюме имеются символьные переменные, то его применение к конкретному состоянию становится недетерминированным. При наличии недетерминированных резюме начальное конкретное состояние перестает однозначно задавать конкретный путь выполнения. Однако для проводимого анализа данное свойство начального конкретного состояния критично, поэтому для каждого вызова функции в развертке введем свой набор неявных параметров p_i, таких, что результат применения резюме однозначно определяется по конкретному состоянию и значениям параметров p_i. Так как количество вызовов функции в развертки составляет конечное число, включим все их неявные параметры функции в входные параметры анализируемой функции. Дополненный таким образом набор входных параметров анализируемой функции снова однозначно определяет дальнейшее выполнение.

Для вычисления всех возможных путей доступа при построении начального символьного состояния потребуем от резюме, чтобы они явно задавали использующиеся в них пути доступа. Чтобы определить набор неявных параметров для текущего вызова, также, как и в случае с символьными переменными, необходимо произвести анализ возможных путей доступа для результата применения резюме.

При помощи недетерминизма в резюме можно моделировать результат выполнения неизвестных функций. На практике такое моделирование необходимо при анализе промышленных программ.

4.2 Построение резюме

Как уже говорилось ранее, в качестве резюме можно использовать символьное состояние и пометки, полученные после выполнения функции. Отдельно необходимо сохранить символьное выражение, соответствующее возвращаемому функцией значению. Также необходимо определиться с тем, какие символьные выражения должны входить в итоговое резюме.

Прежде всего постараемся сохранить те символьные выражения, которые необходимы для описания условий пометок. Если очередное условие пометки нельзя добавить к резюме, не превысив *M*, такая пометка не включается в резюме. В случае, когда все символьные выражения, требуемые для формул пометок, не могут быть сохранены, можно рассмотреть задачу аппроксимации данных снизу более строгими формулами. Аппроксимация осуществляется именно снизу для того, чтобы не допустить появления ложных срабатываний.

С точки зрения такой аппроксимации корректно заменять формулы на ложь, т.е. игнорировать формулы, не подходящие под ограничение по размеру.

Все вхождения символьных выражений, не вошедших в резюме, в символьное состояние, предлагается заменять на новые символьные переменные. Символьное состояние, полученное в результате таких замен, является аппроксимацией сверху исходного символьного состояния.

4.3 Применение резюме

Рассмотрим инструкцию вызова функции. Будем считать, что инструкция вызова функции имеет следующий вид:

$$a = foo(x_1,x_2,...,x_n);$$

где x_i – некоторая переменная, определённая в программе. Будем считать, что резюме содержит начальное состояние $entry = \langle RV_{entry}, \mathrm{HEAP}_{entry}, \top\rangle$ и конечное состояние $exit = \langle RV_{exit}, \mathrm{HEAP}_{exit}, \mathcal{G}_{exit}\rangle$. Точки применения резюме соответствует символьное состояние $\langle RV, \mathrm{HEAP}, \mathcal{G}\rangle$. Символьные выражения в entry и exit сформулированы над символьными переменными вызванной функции. Для применения резюме считается, что задано отображение $W2\underline{P}$: $W \to \underline{P}$, которое задает отображение переменных, используемых при вызове во входные параметры вызываемой функции $\underline{P}$, в том числе неявные переменные резюме. С помощью отображения $W2\underline{P}$ построим отображение $\underline{S}2S : \underline{SE} \to SE$, где $\underline{SE}$ — символьные выражения вызываемой функции. Данное отображение ставит в соответствие входным символьным значениям вызываемой функции символьные выражения вызывающей функции. Для построения $\underline{S}2S$ воспользуемся следующим алгоритмом. Во-первых, выполним следующее сопоставление:

$$\underline{S}2S : RV_{entry}(W2\underline{P}(x_i)) \mapsto RV(x_i), \forall i \in [1,n]$$

Обозначим за R множество ссылок вызываемой функции. Рассмотрим $\langle \underline{r}, f\rangle, r \in \underline{R}, f \in F$, таких, что для $\langle \underline{r}, f\rangle$ определено отображение HEAP_{entry}. Пусть x – символьная переменная вызванной функции, которая может принимать значение $\underline{r}$. По построению символьных переменных, такая переменная только одна. Тогда, если $\underline{S}2S$ определено для x, то доопределим $\underline{S}2S$ следующим образом:

$$\underline{S}2S: \mathrm{HEAP}_{entry}(\langle r, f\rangle) \mapsto \mathrm{HEAP}(\langle \underline{S}2S(x), f\rangle)$$

Также доопределим в $S2S$ отображение для символьных выражений вызываемой функции следующим образом:

$$\underline{S}2S\,(a \diamondsuit b) = \underline{S}2S\,(a) \diamondsuit \underline{S}2S\,(b)$$

$$\underline{S}2S\,(\square a) = \square \underline{S}2S\,(a)$$

$$\underline{S}2S\ (ite(cond, a, b)) = ite(\underline{S}2S\ (cond), \underline{S}2S\ (a), \underline{S}2S\ (b))$$

Рассмотрим отображения $HEAP_{entry}$ и $HEAP_{exit}$. Найдем для них такое множество пар $\langle r,f\rangle$, что $HEAP_{entry}(\langle r,f\rangle) /= HEAP_{post}(\langle r,f\rangle)$. Данное множество пар обозначим за Diff. Тогда действие резюме определим как набор присваиваний. Для этого рассмотрим множество всех пар $\langle r,f\rangle \in$ Diff и выполним присваивание значения $\underline{S}2S(HEAP_{exit}(\langle r,f\rangle))$ в поле $\underline{S}2S(x).f$ по аналогии с инструкцией $a.f = b$. Для выполнения присваивания возвращаемого функцией значения необходимо просто произвести отображение сохраненного для него символьного выражения.

Сразу оговоримся, что в случае, если параметры вызывающей функции являются псевдонимами один для другого, то результат такого применения резюме будет частично некорректным, т.к. при построении функции предполагалось, что все входные значения не являются псевдонимами один для другого. Однако на практике практически не встречаются ложные срабатывания, вызванные подобной ситуацией. Для переноса условий из резюме достаточно воспользоваться построенным отображением $\underline{S}2S$.

5. Особенности реализации

Обратим внимание на некоторые особенности реализации данного метода.

Построение символьных переменных для начального состояния можно проводить «лениво», по мере их использования во время основной фазы анализа. В данном случае, начальное символьное состояние будет доступно только по завершении анализа функции. Разворот цикла предлагается разделить на две итерации, чтобы ошибка, причина которой возникает на первой итерации, могла быть обнаружена на второй.

Для поиска допустимых начальных состояний предлагается использовать SMT-решатель. Решение формул предлагается производить в логике QF_BV согласно классификации SMT-LIB [2]. Данная логика описывает пропозициональные формулы над битовыми векторами, что позволяет достичь битовой точности при анализе. Тем не менее, запросы к решателям могут занимать достаточно долгое время, поэтому важной задачей является минимизация количества запросов. Для минимизации количества запросов вместе с точным анализом, основанным на формулах, можно также производить неточный анализ, который в простых случаях способен доказывать наличие либо отсутствие ошибки. Тогда, если неточный анализ смог доказать наличие, либо же отсутствие ошибки, запрос к SMT-решателю не является обязательным.

Например, для задачи разыменования нулевого указателя можно рассмотреть следующий анализ потока данных над символьными выражениями. Каждому символьному выражению поставим в соответствие одно из трех абстрактных значений: *Nil, NotNil, Unknown*. Данные абстрактные значения образуют

нижнюю полурешетку: *Unknown* ≺ *Nil,Unknown* ≺ *NotNil*. Данными абстрактными значениями помечены символьные выражения при прохождении соответствующих инструкций *assume*(*x*).

Операция объединения происходит в соответствии с порядком на полурешетке. Важным отличием данных пометок от пометок, используемых раннее, является то, что из равенства *null ite* выражения не может быть сделано никаких выводов о его операндах.

Соответственно, в случае разыменования символьного выражения, помеченного меткой *nullable* или *compared*, вызов SMT-решателя нужен только в том случае, когда данное символьное выражение помечено как *Unknown*.

6. Практические результаты

Данный подход реализован в рамках инструмента Svace [3] в стороннем анализаторе CSCC. Анализатор CSCC разработан на базе компиляторной инфраструктуры Roslyn. Детали реализации данного метода в CSCC планируется описать в последующих работах. Тестирование CSCC проводилось на программном обеспечении с открытым исходным кодом. В ходе тестирования была установлена хорошая масштабируемость предложенного метода на промышленных приложениях с открытым исходным кодом. Результаты тестирование приведены в таблице 1. В первом столбце таблицы указано название анализируемого проекта, второй столбец содержит количество строк кода. Последующие четыре столбца содержат число срабатываний соответствующего типа на данной проекте. Последний столбец указывает время работы анализатора. Данные результаты показывают, что число срабатываний и время работы анализатора коррелируют с объемом исходного кода. Расхождение между объемом кода и временем работы для проекта OpenBVE связано с наличием в проекте большого числа сложных для анализа функций. Большое число срабатываний на проекте Jil связано с дублированием ошибочного паттерна. Процент истинных срабатываний для реализованных проверок составил более 50%. Дальнейший анализ результатов показал, что основной причиной ложных срабатываний является неточность используемых резюме при анализе.

Табл. 1 Результаты работы анализатора на проектах с открытым исходным кодом

Проект	LOC	Разыменование константы null	Разыменование константы null в вызванной функции	Разыменование после сравнения с null	Разыменование после сравнения с null в вызванной функции	Время
Sake	1093	0	0	0	0	00:08
Polly	4828	0	0	0	0	00:13

BobBuilder	6426	1	1	0	0	00:21
Shadowsocks	17862	0	1	0	0	00:44
Perspective	20590	0	0	0	0	00:39
CSParser	21087	2	0	5	0	01:16
NetMQ	30258	0	1	1	1	01:06
Jil	49333	0	0	32	0	02:10
LibGit	51032	0	5	0	0	01:15
OpenBVE	56859	0	2	0	2	10:26
Cassandra	62757	1	1	1	1	02:12
OpenRA	104505	9	5	10	1	06:01
FSpot	110765	8	5	3	2	03:17
ShareX	144641	2	5	5	2	04:49
Banshee	167828	3	5	13	2	06:52
Lucene.Net	528120	13	12	21	2	22:01
SharpDevelop	1224434	32	186	57	99	2:11:42
Roslyn	1356367	114	222	70	25	1:46:45
NetOffice	2560115	2	56	10	245	3:31:52

7. Обзор существующих подходов

Точный и полный алгоритм поиска дефектов на путях выполнения невозможен в силу неразрешимости задач останова и анализа псевдонимов [4]. На практике используется некоторая комбинация аппроксимаций множества рассматриваемых путей сверху и снизу, что приводит к возникновению ложных срабатываний (при аппроксимации сверху) и пропуску дефектов (при аппроксимации снизу). Исходя из анализа работ по статическим анализаторам, предлагается следующий набор характеристик, по которым можно классифицировать статические анализаторы:

- покрытие — множество рассматриваемых путей выполнения;
- точность — процент истинных срабатываний, выдаваемых анализатором;
- масштабируемость — возможность анализа промышленных проектов с миллионами строк кода;

- автономность — возможность осуществления анализа без какой-либо дополнительной информации от пользователя.

Далее будем рассматривать только автономные методы поиска дефектов. В задачах автономной верификации на моделях зачастую требуется покрытие, как можно более близкое к реальному поведению программы. Самыми точными подходами в данной области являются подходы, основанные на bounded model checking(CBMC [5], LLBMC [6]), т.к. они моделируют каждый возможный путь выполнения с точностью до бита. Однако применение данных инструментов всегда связанно с поиском компромисса между временем работы и покрытием путей, т.к. данные инструменты заведомо неспособны перебрать все возможные пути выполнения за конечное время. Стоит отметить, что попытка использовать инструмент CBMC для автоматической верификации, предпринятая авторами Calysto, показала полную неприменимость данного анализатора для промышленных приложений [7]. Аналогичная попытка применить LLBMC, произведённая в рамках подготовки данной работы, также показала его неприменимость.

Одним из способов автономного доказательства корректности программ являются методы на основе подхода CEGAR [8] (SLAM [9], Blast [10], CPAchecker [11]). Идея данных методов заключается в итеративном поиске ошибочного пути выполнения с помощью уточнения абстракции. Однако для промышленных приложений в некоторых случаях такой итеративный подход принципиально не будет сходиться. Зацикливание, как правило, происходит из-за невозможности представления инварианта цикла в виде конечной формулы логики высказываний. Однако, в отличие от bounded model checking, данный подход может доказывать отсутствие ошибки сразу для всех путей выполнения. По сравнению с bounded model checking, алгоритмы, основанные на CEGAR, имеют лучшее покрытие и немного более низкую точность (например, CPAchecker обычно не поддерживает битовые операции).

Типичным применением инструментов, основанных на методе CEGAR, является верификация драйверов операционных систем, размер которых ограничен несколькими тысячами строк кода. Для моделирования поведения ядра операционной системы используются модельные реализации интерфейсов. Таким образом, несмотря на то, что сами методы являются автономными, для их эффективного применения требуется описание инфраструктуры, в рамках которой происходит анализ. Однако, что важно для автономности, генерация такой инфраструктуры также может производиться автоматически. В качестве примера можно привести разработанный в ИСП РАН проект LDV [12], который автоматически генерирует модельное окружение драйвера для последующего анализа при помощи инструмента CPAchecker.

Для достижения масштабируемости, ввиду алгоритмической неразрешимости решаемых задач, необходимо вводить различные упрощающие

предположения. Данные предположения влияют прежде всего на покрытие и точность проводимого анализа. Однако предположения, значительно ухудшающие точность проводимого анализа, делают его результаты бесполезными, т.к. анализ срабатываний пользователем анализатора является крайне трудоемким процессом, поэтому большинство упрощающих предположений направлено именно на уменьшение области покрытия. В отношении снижения точности важно понимать, что упрощающее предположение можно считать допустимым только в случае, если оно не приводит к резкому увеличению срабатываний определённого типа на интересующих программах. Упрощающие предположения направлены прежде всего на решение следующих проблем:

- поддержка циклов;
- поддержка рекурсии;
- контекстная чувствительность.

В данном случае считается, что решение таких проблем, как неограниченность кучи и анализ псевдонимов, следует из того, как анализ решает проблему циклов и рекурсии. При решении задачи циклов и рекурсии можно выделить два основных подхода: развертку циклов и итеративный алгоритм.

Итеративный алгоритм используется в таких подходах, как абстрактная интерпретация [13] и анализ потока данных. При использовании данных подходов, как правило, уменьшается точность анализа при сохранении покрытия. Основным применением для анализаторов, основанных на абстрактной интерпретации, таких как Astree [14], является анализ программ для встраиваемых систем, в которых отсутствует динамическое выделение памяти, рекурсия, а структура графа потока управления более простая, нежели у обычных программам. К сожалению, для программ общего назначения данный применение подхода затруднено из-за недостаточной точности анализа, применения аппроксимации снизу, и проблем, связанных с временем сходимости итеративного алгоритма. Более перспективным для программ общего назначения видится применение развертки циклов.

Набор подходов, посимвольно интерпретирующих выполнение программы, называется символьным выполнением [15]. К нему относятся такие анализаторы, как Saturn [16], Calysto [7], Varvel [17], CSA [18]. Данная группа примечательна тем, что, сохраняя точность близкую к CBMC, она может производить анализ промышленных приложений. Все перечисленные анализаторы производят развертку циклов и не учитывают, либо разворачивают рекурсию. Разница между данными анализаторами касается прежде всего использующихся компромиссов, которые заключаются в применении различных подходов для реализации контекстной чувствительности.

Рассматривая CSA в сравнении с CBMC, можно заметить, что данные анализаторы объединяет то, что они используют встраивание для реализации межпроцедурного анализа. Однако CSA, в отличие от CBMC анализатора, анализирует каждую функцию вне контекста, считая её точкой входа. Встраивание CSA также применяется только в том случае, когда размер встраиваемой функции невелик, иначе он считает данный вызов неизвестным, что зачастую эквивалентно его игнорированию. Кроме того, CSA использует упрощенный решатель вместо полноценных SAT/SMT решателей, что негативно сказывается на точности анализа. Однако, благодаря данным ограничениям, CSA может быть легко использован для анализа промышленных проектов. Отдельную проблему для CSA представляет межмодульный анализ, т.к. CSA работает на внутреннем представлении компилятора clang, который за одну итерацию обрабатывает один модуль трансляции.

Подход инструмента Saturn идейно близок к CBMC, с той разницей, что, во-первых, вместо встраивания функций используются резюме, во-вторых, при анализе точно объединяет символьные состояния путей в точках слияния. Для представления условий достижимости также используется предикатная абстракция. При межпроцедурном анализе Saturn также использует все функции в обратном порядке топологической сортировки графа вызовов. Каждая функция анализируется как точка входа, а также используется контекстно-нечувствительное упрощающее предположение о том, что входные данные функции могут быть произвольными. Благодаря данным решениям, разработчикам Saturn удалось добиться масштабируемости их подхода для программ, состоящих из нескольких сотен тысяч строк.

Инструмент Calysto является идейным продолжателем анализатора Saturn, однако по сравнению с Saturn, Calysto имеет следующие отличия. Во-первых, Calysto использует специализированный SMT-решатель SPEAR [19], в то время как Saturn использует SAT-решатели. Во-вторых, в качестве резюме Calysto сохраняет не наборы предусловий и постусловий, а граф символьный значений, схожий по структуре с классическим графом значений [20]. Благодаря представлению резюме функции в виде графа значений, становится возможным последовательное уточнение формул при межпроцедурном анализе по аналогии с методом CEGAR. Главной задачей инструмента Calysto было достижение большей точности из-за улучшения контекстной чувствительности. Исходя из данных, приведённых в статье, анализатор Calysto работает в среднем быстрее Saturn, имеет намного более низкий процент ложных срабатываний и находит куда больше истинных срабатываний. К сожалению, авторы Calysto не распространяют данный инструмент, что делает невозможным проверку предоставленных результатов. Кроме того, ни Calysto, ни Saturn являются исследовательскими проектами, а не промышленными анализаторами.

В отличие от двух предыдущих анализаторов, Varvel является промышленным анализатором, использующимся в NEC. Анализатор Varvel состоит из трех основных компонент: абстрактного интерпретатора, BMC-анализатора, подобного CBMC с возможностью объединения состояний, и инфраструктуры создания резюме функций. Идея метода заключается в применении абстрактной интерпретации для доказательства отсутствия ошибки. При помощи слайсинга из программы исключаются инструкции, не влияющие на недоказанные потенциально уязвимые инструкции. По словам авторов, при помощи такой комбинации удается сократить объем анализируемого кода на 60%. Далее, подобно Saturn, в Varvel все функции рассматриваются как возможные точки входа с произвольными входными параметрами. Для каждой такой функции запускается BMC-анализ с ограничением на глубину стека вызова. Все функции, имеющие глубину вызова относительно данной функции больше заданной величины, заменяются на свои резюме, содержащие предусловия и постусловия их выполнения. Авторы утверждают, что данный подход может быть эффективно применён для анализа проектов, состоящих из нескольких десятков миллионов строк кода. Опыт Varvel доказывает, что анализаторы, основанные на BMC, могут масштабироваться на промышленные проекты практически любых размеров.

Рассматриваемый в данной работе анализатор имеет общий набор упрощающих предположений с такими анализаторами, как Saturn и Varvel, а именно:

- обратные ребра в графе вызовов игнорируются;
- развертка циклов происходит на определенное число итераций;
- при анализе каждая функция считается точкой входа, а её аргументы могут принимать произвольные значения;
- все неизвестные указатели не являются псевдонимами один другого.

Предложенный в данной работе метод имеет следующие отличия от рассмотренных инструментов:

1. В отличии от Saturn, для решения формул используется SMT, а не SAT решатель, что существенно упрощает построение формул.
2. Запрос к SMT решателю происходит только в том случае, когда более простой анализ не смог доказать отсутствие ошибки. Данный подход существенно уменьшает количество запросов для проверки разыменования нулевого указателя.

3. В отличие от инструмента Valvel, для анализа и для построения резюме используется один и тот же анализ.
4. Ввиду ограничения на время работы и размер потребляемой памяти, в отличие от инструмента Calysto, размер сохраняемого резюме дополнительно ограничивается. Данное ограничение необходимо для анализа проектов, состоящих из миллионов строк кода.

8. Заключение

В данной работе был предложен метод межпроцедурного анализа программ на языке C#, позволяющий производить поиск различных дефектов в программе. В частности, были разобраны вопросы построения внутрипроцедурного чувствительного к путям анализа и использование резюме функции для реализации межпроцедурного анализа. Предложенный метод является масштабируемым и может быть использован для анализа промышленных проектов. На основе данного метода был предложен способ поиска ошибок разыменования нулевого указателя. Данный метод был реализован в анализаторе CSCC. Реализованный метод показал хорошую производительность и приемлемое количество ложных срабатываний. В последующих работах планируется более подробно описать особенности реализации данного метода в анализаторе CSCC.

Список литературы

[1]. P. Tu, D. Padua. Efficient Building and Placing of Gating Functions, SIGPLAN Not. 1995. Vol. 30, no. 6. pp. 47–55. doi: 10.1145/223428.207115
[2]. B. Clark, F. Pascal, T. Cesare. The SMT-LIB Standard: Version 2.5. Department of Computer Science, The University of Iowa, 2015. www.SMT-LIB.org.
[3]. В.П. Иванников А.А. Белеванцев А.Е. Бородин В.Н. Игнатьев Д.М. Журихин А.И. Аветисян М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН. 2014. Том. 26, выпуск 1. pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7
[4]. G. Ramalingam. The Undecidability of Aliasing. ACM Trans. Program. Lang. Syst. 1994. Vol. 16, no. 5. pp. 1467–1471. doi: 10.1145/186025.186041
[5]. E. Clarke, D. Kroening. Hardware Verification using ANSI-C Programs as a Reference. Proceedings of ASP-DAC. 2003, pp. 308–311.
[6]. Falke Stephan, Merz Florian, Sinz Carsten. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. Tools and Algorithms for the Construction and Analysis of Systems. pp. 623–626. doi: 10.1007/978-3-642-36742-7_48
[7]. Babic D., Hu A.J. Calysto. Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. 2008. May. pp. 211–220.
[8]. E. Clarke, O. Grumberg, S. Jha at al. Counterexample-Guided Abstraction Refinement. Computer Aided Verification. 2000. pp. 154–169. doi: 10.1007/10722167_15

[9]. SLAM2: Static Driver Verification with Under 4% False Alarms, T. Ball, E. Bounimova, R. Kumar, V. Levin. Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. 2010. pp. 35–42.

[10]. D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The software model checker Blast. International Journal on Software Tools for Technology Transfer. 2007. Vol. 9, no. 5-6. Pp. 505–525.

[11]. D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. Computer Aided Verification. 2011. pp. 184–190.

[12]. И.С. Захаров М.У. Мандрыкин В.С. Мутилин Е.М. Новиков А.К. Петренко А.В. Хорошилов. Конфигурируемая система статической верификации модулей ядра операционных систем. Программирование. 2015. Том. 41, номер. 1. сс. 44–67.

[13]. P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1977. pp. 238–252. doi: 10.1145/512950.512973

[14]. P. Cousot, R. Cousot, J. Feret at al. The ASTREÉ Analyzer. Programming Languages´ and Systems Vol. 3444 of Lecture Notes in Computer Science. Pp. 21–30.

[15]. J. King. Symbolic Execution and Program Testing. Commun. ACM. 1976. Vol. 19, no. 7. pp. 385–394.

[16]. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.

[17]. F. Ivančić, G. Balakrishnan, A. Gupta at al. Scalable and scope-bounded software verification in Varvel. Automated Software Engineering. 2015. Vol. 22, no. 4. pp. 517–559.

[18]. Z. Xu, and T. Kremenek and J. Zhang. A memory model for static analysis of C programs. Leveraging Applications of Formal Methods, Verification, and Validation. 2010. Pp 535–548.

[19]. D. Babic, F. Hutter. Spear theorem prover. Proc. of the SAT. 2008. pp. 187–201.

[20]. J. Reif, H. Lewis. Symbolic evaluation and the global value graph. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977. pp. 104–118.

Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference

V. Koshelev <vedun@ispras.ru>
I. Dudina <eupharina@ispras.ru>
V. Ignatyev <valery.ignatyev@ispras.ru>
A. Borzilov <helendile@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Annotation. This paper proposes an approach for detecting bugs in C# programs and uses null pointer deference as the main example. The approach employs a scalable path-sensitive analysis, which involves symbolic execution with state merging and function summary methods. C/C++ program analyzers like Saturn Software Analysis Project, Calysto or Svace use similar approaches. We analyze functions in backward topological order with account for previously calculated summaries. For summary construction, we use the same analysis engine as for bug detection. The paper contains a formal description of the proposed approach applied to reduced "sugar-free" subset of C# language. For each instruction of the considered language, we define a formal semantics and transfer function according to the symbolic state. During the path-sensitive analysis, we store additional information related to possible bugs in the symbolic state, and the decision whether the warning should be reported is made upon the satisfiability of the corresponding formula. Therefore, we reduce the problem of bug detection to satisfiability of a first-order logical formula defined on atoms, which are arithmetic expressions on function input values. It can be efficiently solved with modern SMT solvers. We have implemented the approach in our Roslyn-based analyzer, called SharpChecker. Evaluation of SharpChecker on open-source commodity applications has shown acceptable scalability and reasonable amount of warnings.

Keywords: static analysis; null pointer dereference; path-sensitive analysis; function summary; bug detection.

DOI: 10.15514/ISPRAS-2015-27(5)-5

For citation: Koshelev V., Dudina I., Ignatyev V., Borzilov A. Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5.

References

[1]. P. Tu, D. Padua. Efficient Building and Placing of Gating Functions, SIGPLAN Not. 1995. Vol. 30, no. 6. pp. 47–55. doi: 10.1145/223428.207115

[2]. B. Clark, F. Pascal, T. Cesare. The SMT-LIB Standard: Version 2.5. Department of Computer Science, The University of Iowa, 2015. www.SMT-LIB.org.

[3]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode programm. [Static analyzer Svace for finding of defects in program source code] // Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[4]. G. Ramalingam. The Undecidability of Aliasing. ACM Trans. Program. Lang. Syst. 1994. Vol. 16, no. 5. pp. 1467–1471. doi: 10.1145/186025.186041

[5]. E. Clarke, D. Kroening. Hardware Verification using ANSI-C Programs as a Reference. Proceedings of ASP-DAC. 2003, pp. 308–311.

[6]. Falke Stephan, Merz Florian, Sinz Carsten. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. Tools and Algorithms for the Construction and Analysis of Systems. pp. 623–626. doi: 10.1007/978-3-642-36742-7_48

[7]. Babic D., Hu A.J. Calysto. Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. 2008. May. pp. 211–220.

[8]. E. Clarke, O. Grumberg, S. Jha at al. Counterexample-Guided Abstraction Refinement. Computer Aided Verification. 2000. pp. 154–169. doi: 10.1007/10722167_15

[9]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms, Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. 2010. pp. 35–42.

[10]. D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The software model checker Blast. International Journal on Software Tools for Technology Transfer. 2007. Vol. 9, no. 5-6. Pp. 505–525.

[11]. D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. Computer Aided Verification. 2011. pp. 184–190.

[12]. I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, A.V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software. January 2015, Volume 41, Issue 1, pp 49-64.

[13]. P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1977. pp. 238–252. doi: 10.1145/512950.512973

[14]. P. Cousot, R. Cousot, J. Feret at al. The ASTREÉ Analyzer. Programming Languages´ and Systems Vol. 3444 of Lecture Notes in Computer Science. Pp. 21–30.

[15]. J. King. Symbolic Execution and Program Testing. Commun. ACM. 1976. Vol. 19, no. 7. pp. 385–394.

[16]. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.

[17]. F. Ivančić, G. Balakrishnan, A. Gupta at al. Scalable and scope-bounded software verification in Varvel. Automated Software Engineering. 2015. Vol. 22, no. 4. pp. 517–559.

[18]. Z. Xu, and T. Kremenek and J. Zhang. A memory model for static analysis of C programs. Leveraging Applications of Formal Methods, Verification, and Validation. 2010. Pp 535–548.

[19]. D. Babic, F. Hutter. Spear theorem prover. Proc. of the SAT. 2008. pp. 187–201.

[20]. J. Reif, H. Lewis. Symbolic evaluation and the global value graph. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977. pp. 104–118.

Метод легковесного статического анализа для поиска состояний гонок[1]

[1]П.С. Андрианов <andrianov@ispras.ru>
[1]В.С. Мутилин <mutilin@ispras.ru>
[1,2,3,4]А.В. Хорошилов <khoroshilov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В этой статье представлен подход легковесного статического анализа для поиска состояний гонок, названный CPALockator. Он учитывает такую специфику ядер операционных систем, как сложный параллелизм и особые примитивы синхронизации. Метод основан на алгоритме Lockset, но использует две эвристики, которые призваны уменьшить количество ложных предупреждений: модель памяти и модель параллелизма. В качестве примитивов синхронизации рассматриваются блокировки. Основным предметом нашего исследования являются ядра операционных систем, но предложенный подход может быть применен также и для других программ. Метод основан на идее адаптивного статического анализа (Configurable Program Analysis, CPA) и реализован в инструменте CPAchecker. Реализация метода состоит из двух стадий: сначала определяется множество разделяемых переменных для каждой точки программы, затем производится анализ примитивов синхронизации. На второй стадии собирается информация о всех возможных доступах к разделяемым переменным и захваченных примитивах синхронизации. После этого создается отчет, содержащий предупреждения для тех переменных, для которых было найдено хотя бы два доступа, образующие потенциальное состояние гонки. Для каждого доступа приводится один из возможных путей выполнения программы, который ведет к нему. Инструмент был апробирован на ядре операционной системы реального времени объемом приблизительно 200 000 строк кода. Он позволил найти 20 новых состояний гонки, признанных разработчиками. Кроме того, был произведен пилотный запуск инструмента на драйверах операционной системы Linux с помощью инфраструктуры

[1] Исследование проводилось при финансовой поддержке РФФИ в рамках проекта №13-01-00694

инструмента LDV, который использовался для подготовки заданий для инструмента CPALockator. Дальнейшим направлением исследований является разработка более точной модели потоков, интеграция с более точными тяжеловесными техниками анализа.

Ключевые слова: статический анализ, состояние гонки, ядро операционной системы, разделяемые данные.

DOI: 10.15514/ISPRAS-2015-27(5)-6

Для цитирования: Андрианов П.С., Мутилин В.С., Хорошилов А.В. Метод легковесного статического анализа для поиска состояний гонок. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 87-116. DOI: 10.15514/ISPRAS-2015-27(5)-6.

1. Введение

Несмотря на большой прогресс в области верификации программного обеспечения, ошибки, связанные с параллельным выполнением кода остаются одними из наиболее труднообнаруживаемых. Более того, такие ошибки довольно многочисленны, например, в среднем они составляют около 20% от всех ошибок в файловых системах ядра операционной системы Linux [1]. Наиболее частыми причинами ошибок, связанных с параллельным выполнением ядра операционной системы, являются состояния гонки, при которых происходит одновременный доступ к разделяемым данным из нескольких потоков. В частности анализ исправлений за год разработки ядра Linux показал, что ошибки, связанные с состояниями гонки, образуют наиболее многочисленный класс и составляют 17% от всех типичных ошибок [2].

Существуют два пути для поиска состояний гонки автоматически: динамический анализ и статический анализ. Техники динамического анализа позволяют получить относительно небольшой процент ложных срабатываний. Примерами инструментов, реализующих методы динамического анализа, являются Eraser [3], RaceHound [4] и DataCollider [5]. Они способны находить потенциальные состояния гонки только на тех путях выполнения программы, которые происходят в течение реального исполнения. Состояние гонки определяется двумя практически одновременными доступами к одним и тем же данным, что усложняет ее обнаружение. Инструменты, которые используют методы на основе векторных часов, могут работать с двумя разнесенными во времени доступами к памяти, но они чувствительны к порядку операций. Кроме того, известно, что значительную часть путей исполнения программы достаточно сложно воспроизвести в тестовом окружении.

Методы статического анализа имеют свои проблемы. Тяжеловесные техники достаточно точны, но требуют большого количества времени для анализа. В случае задачи поиска гонок общее число мест, где может возникнуть состояние гонки, слишком велико. Проводились некоторые эксперименты по

верификации модулей ядра, и результаты показали, что тяжеловесный подход не масштабируется[6]. Происходит комбинаторный взрыв состояний, поэтому даже для небольших модулей количество затраченного времени и памяти было гигантским.

Методы легковесного статического анализа, например, метод, реализованный в инструмента Locksmith [7], работают очень быстро, но число ложных срабатываний обычно бывает очень велико. Для инструмента Locksmith средний процент ложных сообщений об ошибках составляет 73% на некоторых POSIX приложениях и около 96% на нескольких драйверах [8]. Существующие методы не принимают во внимание некоторую специфику ядра операционной системы, описанную ниже, поэтому большинство драйверов и особенно ядро само по себе анализировать существующими инструментами, предназначенными для пользовательских приложений, очень сложно.

Параллелизм в ядрах операционных систем устроен сложным образом, так как они являются асинхронными. Многие функции ядра могут быть выполнены параллельно друг с другом, и определить, когда начинается параллельное исполнение, очень сложно. Кроме того в ядрах операционных систем используются дополнительные примитивы синхронизации, такие как отключение прерываний или планирования. Еще одна важная особенность — это активное использование адресной арифметики. Как результат, поиск состояний гонок в ядрах операционных систем является более сложной задачей, чем в пользовательских приложениях.

В этой статье мы предлагаем новый метод легковесного статического анализа для обнаружения состояний гонок, названный CPALockator. Он может легко масштабироваться на большие объемы исходного кода, оставляя процент ложных срабатываний на приемлемом уровне и принимая во внимание специфику ядра операционной системы.

Оставшаяся часть статьи организована следующим образом. В разделе 2 даются необходимые определения. В разделе 3 описывается основная идея предложенного метода. После этого рассказывается о подходе адаптивного статического анализа. Реализация метода обсуждается в разделе 5. Следующий раздел посвящен интеграции в систему LDV [12]. В разделе 7 мы рассказываем о полученных результатах, потом в разделе 8 — о близких работах. В заключении кратко описываются планы на будущее.

2. Определения

В этой статье термин **поток** используется, чтобы обозначить независимый поток выполнения инструкций в ядре операционной системы, например, прерывание от аппаратуры или выполнение системных вызовов от имени пользовательского потока. Если некоторый системный вызов может быть

прерван прерыванием от аппаратуры, мы считаем, что этот системный вызов и прерывание могут выполняться параллельно друг с другом.

Блокировка — это объект, использующийся для предотвращения одновременного доступа к памяти. Если некоторая блокировка захвачена из одного потока, то другой поток, пытающийся захватить ту же самую блокировку, не может продолжить свое выполнение до тех пор, пока блокировка не будет освобождена. Типичными примерами блокировок могут быть мьютексы и спинлоки. Мы считаем такие примитивы синхронизации ядра, как отключение прерываний или планирования, специальными блокировками. Например, функция `irq_disable()` отключает планирование и тем самым запрещает любое параллельное исполнение, поэтому мы считаем, что воображаемая глобальная блокировка `irq_disable` захвачена. Некоторые блокировки могут быть захвачены несколько раз, в этом случае имеет место рекурсивный захват блокировки.

Разделяемые данные — это область памяти, которая доступна из нескольких потоков. В языке Си разделяемые данные представлены глобальными переменными и указателями на память, доступную из нескольких потоков через корректные конструкции языка Си. Важно отметить, что разделяемость данных является характеристикой, зависящей от времени. Локальные данные могут стать разделяемыми в некоторой точке программы и вернуть статус локальности позже.

Использование данных — чтение или запись данных.

Состояние гонки — это ситуация, при которой происходят два неупорядоченных использования одних и тех же разделяемых данных и по крайней мере одно из них является записью. Состояние гонки не всегда приводит к ошибке (так называемое доброкачественное состояние гонки), но является симптомом ее.

3. Легковесный метод для поиска состояний гонки

Метод CPALockator основан на алгоритме Lockset [3], который строит множество *C(v)* потенциальных блокировок для каждой разделяемой ячейки памяти *v*. Это множество содержит в себе те блокировки, которые защищают v для дальнейших действий. Блокировка *l* находится в *C(v)* в текущий момент времени, если для каждого потока доступ к *v* всегда происходил при захваченной блокировке *l*. *C(v)* инициализируется всеми возможными блокировками. Когда происходит доступ к данным, *C(v)* обновляется, как пересечение *C(v)* и того множества блокировок, которые захвачены на данный момент в текущем потоке. Если *C(v)* становится пустым, имеет место потенциальное состояние гонки.

Чтобы задать алгоритм поиска состояний гонок мы должны ответить на следующие вопросы:

- Когда начинается параллельное исполнение?

- Что такое данные?
- Какие данные считаются одинаковыми?
- Что такое блокировки и какие существуют правила для операций с ними?
- Какие блокировки считаются одинаковыми?

Инструмент динамической верификации Eraser, который первым реализовал Lockset, использует точки создания потоков, чтобы определить, когда начинается параллельное выполнение. Для ядра операционной системы определить, когда начинается параллельное выполнение, достаточно сложно. Мы считаем, что каждый системный вызов или обработчик прерывания может выполняться параллельно с любым другим, включая себя. В реальности взаимосвязь между ними более сложная. Модель потоков в методе CPALockator представлена функцией main, которая содержит вызовы всех системных вызовов и обработчиков прерываний.

Eraser оперирует ячейками памяти при реальном выполнении. Метод CPALockator считает все переменные и поля структур единицей данных по умолчанию. Существуют ситуации, в которых доступ к различным полям структур должен быть защищен блокировкой. Предположим, что у нас есть тип структуры, представляющий разделяемый связный список с полями next и prev. Пусть у нас есть два доступа: к полю next одной переменной этого типа и к полю prev другой переменной этого же типа. Все статические методы, оперирующие с ячейками памяти столкнутся с проблемой в этом случае, так как всегда очень сложно понять, что два различных указателя могут указывать на одну и ту же область памяти. В нашем методе мы имеем возможность считать этот случай двумя доступами к одним данным и выдать предупреждение об ошибке. Этот способ требует ручной аннотации, тем не менее достаточно прост в использовании. Подробнее он будет описан в Разделе 5.3.

Так как Eraser оперирует ячейками памяти при реальном исполнении программы, то в нем данные считаются разделяемыми, если два доступа происходят по одному адресу. Статический инструменты, такие как Locksmith, строят граф потоков данных, чтобы определить, какие указатели указывают на одну и ту же память. Однако для ядра операционной системы сложно построить граф потоков данных из-за активного использования адресной арифметики и массивного параллелизма. Поэтому такой метод работает не так хорошо, как для пользовательских программ. В методе CPALockator равенство ячеек памяти следует только из синтаксических правил. Глобальный указатель всегда считается указывающим на одну и ту же область памяти. Аналогичное предположение действует и для локального указателя в заданной функции. Считается, что два поля структуры указывают на одну область памяти, если совпадает тип структуры и имена полей. Важно отметить, что имена самих переменных в этом случае не учитываются. Так, если указатели на структуры A и B имеют одинаковый тип, доступы A->x и B-

>x будут рассматриваться, как доступы к одной и той же памяти. Если структуры A и B не связаны друг с другом, это предположение приведет к ложному сообщению об ошибке. На практике из-за этой эвристики происходит 18% всех ложных предупреждений.

Eraser рассматривает блокировки, как объект, который может быть захвачен. Он поддерживает только две операции с ним: захват и освобождение. Метод CPALockator позволяет описать блокировку: задать функции захвата и освобождения (возможны несколько вариантов), их аргументы, рекурсивность. Равенство блокировок следует из равенства имен объектов (переменных) в обоих методах.

4. Адаптивный статический анализ

Для реализации метода CPALockator был выбран инструмент адаптивного статического анализа CPAchecker (Configurable Program Analysis, CPA) [10]. Он позволяет комбинировать различные техники анализа, встраивать дополнительные подходы, такие как CEGAR, BMC. В этом заключается одно из важных отличий описываемого метода от существующих техник легковесного анализа. Рассмотрим кратко теорию адаптивного статического анализа.

Адаптивный статический анализ может быть составлен из нескольких алгоритмов, предлагающих различные типы анализа. Кроме того, возможна дополнительная настройка алгоритмов путем выбора оператора слияния и способа проверки необходимости завершения анализа.

Адаптивный статический анализ (`D`, `transfer`, `merge`, `stop`) состоит из абстрактного домена `D`, отношения переходов `transfer`, оператора слияния `merge` и оператора останова `stop`. Эти четыре компонента задают алгоритм анализа и влияют на его точность и потребление ресурсов.

Абстрактный домен `D` определяет множество абстрактных состояний. Каждому абстрактному состоянию соответствует его абстрактное значение, то есть множество конкретных состояний, которое оно представляет. Конкретное состояние программы — это отображение переменных программы во множество значений этих переменных.

Отношение переходов `transfer` определяет для каждого состояния **e** потенциальные следующие абстрактные состояния {**e'**}, для которых каждый переход помечен соответствующей дугой графа потока управления (ГПУ).

Оператор `merge` позволяет объединить информацию от нескольких путей анализа. Он определяет, когда два узла дерева достижимости сливаются в один, а когда они должны быть рассмотрены по-отдельности. В классических легковесных подходах объединение всегда происходит, в случае если абстрактные состояния относятся к одной вершине ГПУ. В классических тяжеловесных техниках абстрактные состояния никогда не объединяются.

Оператор `stop` проверяет, является ли текущее состояние покрытым данным множеством уже проанализированных состояний. Он определяет, когда анализ пути завершается в текущей вершине. В классических легковесных подходах останов происходит когда полученное абстрактное состояние не содержит новых конкретных, то есть достигнута неподвижная точка. В тяжеловесных техниках останов происходит, в случае если множество конкретных состояний полученного абстрактного состояния является подмножеством множества конкретных состояний, соответствующих некоторому уже проанализированному абстрактному состоянию.

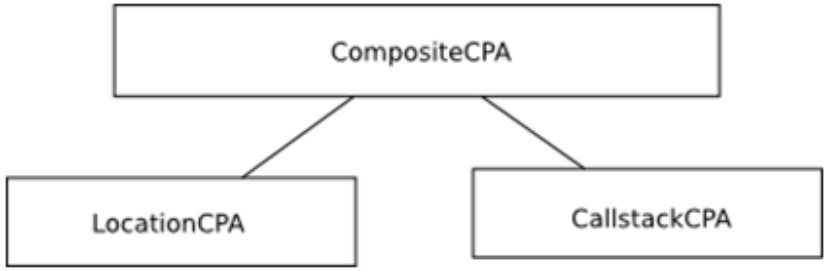


Рис. 1. Пример дерева конфигураций CPA

Рассмотрим пример дерева конфигурации CPA. В нем изображены три анализа. Основным является *CompositeCPA*. Он включает в себя *LocationCPA* и *CallstackCPA*.

Состнояние *LocationCPA* содержит в себе вершину ГПУ, то есть номер строки с исходным кодом. Таким образом, его абстрактный домен является множеством всех возможных вершин ГПУ. Оператор `transfer` меняет номер строки текущего состояния на номер строки вершины, в которую входит соответствующая дуга. Для этого анализа оператор `merge` никогда не объединяет состояния. Останов происходит только если состояние уже было проанализировано ранее.

Состояние *CallstackCPA* состоит из стека вызовов функций. В случае если вызывается новая функция, ее имя помещается на вершину стека. Когда производится возврат, имя функции удаляется из стека. Это описание работы оператора перехода. Операторы stop и merge такие же, как и для предыдущего анализа.

Задача *CompositeCPA* — объединение анализов, описанных выше. Ее абстрактный домен является декартовым произведением доменов *LocationCPA* и *CallstackCPA*. Оператор перехода *CompositeCPA* вызывает операторы переходов вложенных анализов. Сначала он получает новое состояние от *LocationCPA*, затем — от *CallstackCPA* и объединяет их вместе. Это объединение состояние является новым состоянием *CompositeCPA*.

`Merge` и `stop` операторы также объединяют соответствующие операторы вложенных анализов. Для того, чтобы объединить два состояния *CompositeCPA*, нужно сначала объединить состояния *LocationCPA*, которые включены в данные состояния *CompositeCPA*, а потом — состояния *CallstackCPA*. Оператор stop работает похожим образом: если все вложенные CPA решают остановить анализ, *CompositeCPA* также останавливает его.

Рассмотрим, как такая композиция CPA анализирует простую программу (Рис. 2). На рисунке 3 изображен абстрактный граф достижимости для этой программы.

```
1  int g(int a) {
2    int b = 0;
3    if (a == 0) {
4      b++;
5    }
6    return b;
7  }
8  int f() {
9    return 0;
10 }
11 int main() {
12   int t;
13   t = f();
14   g(t);
15 }
```

Рисунок 2 — Пример программы

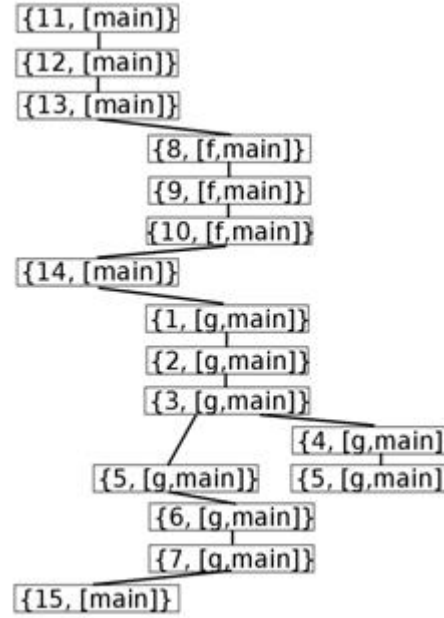


Рисунок 3 — Пример анализа. Первое число в скобках представляет собой состояние LocationCPA (номер строки) и затем идет стек вызовов функций

Инструмент начинает из функции `main`, затем он анализирует функцию `f`, после этого — переходит в функцию `g`. В этой фунции он встречает условие на строке 3. Он анализирует две ветви и получает одинаковые состояния на строке 5. Это означает, что одно состояние покрывается другим, поэтому нет необходимости анализировать оба, а анализ продолжается только для одного из них.

Классические анализы, реализующие подход CPA решают задачу достижимости, то есть, доказывают достижимость некоторой метки в программе (вызова функции и т.п.). Состояние гонки характеризуется двумя точками и для того, чтобы пойти по пути классических анализов, нужно было бы определить состояние нашего анализа, как декартово произведение состояний в двух потоках, то есть получить семантику чередования (англ. interleaving). Мы же используем модульную абстракцию для описания взаимодействия нескольких потоков. Этот подход предполагает использование упрощенной модели взаимодействия потоков. При необходимости эта модель может быть уточнена, но так как в текущем варианте анализа не учитываются условия, то взаимодействие потоков не может быть учтено ни при какой его модели.

5. Реализация

Реализация метода CPALockator состоит из двух этапов. Сначала определяются разделяемые данные, затем для каждого использования разделяемых данных сохраняется множество захваченных блокировок. На рисунке 4 представлены эти этапы.

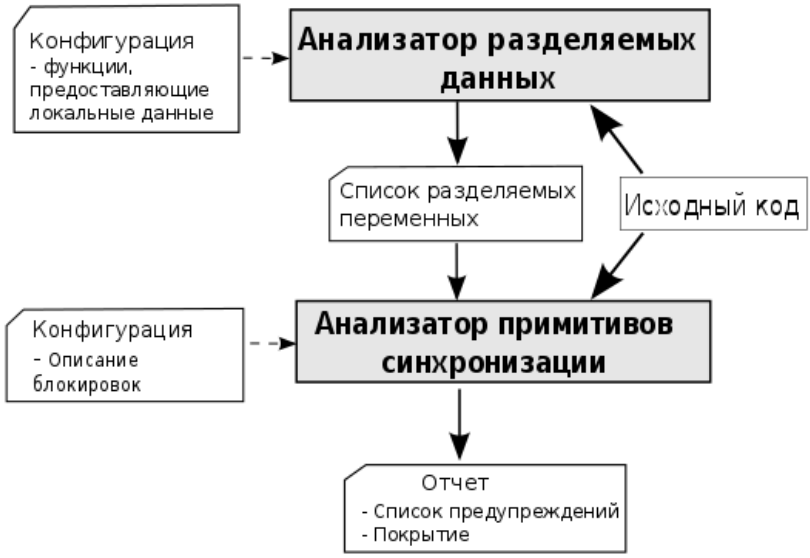


Рисунок 4 — Этапы CPALockator

Конфигурация CPA для Анализатора разделяемых данных состоит из функций, предоставляющих локальные данные, например, `calloc`, `malloc` и др. Мы предполагаем, что указатели, возвращаемые этими функциями,

указывают на локальные данные, которые не могут быть разделяемыми в соответствующей точке программы. Конфигурация для Анализатора блокировок включает в себя описание блокировок и аннотации, которые описываются в секции 5.3.

5.1 Конфигурация CPA для Анализатора разделяемых данных

Анализатор разделяемых данных используется для формирования списка разделяемых переменных для каждой точки программы (см. Рис 5).

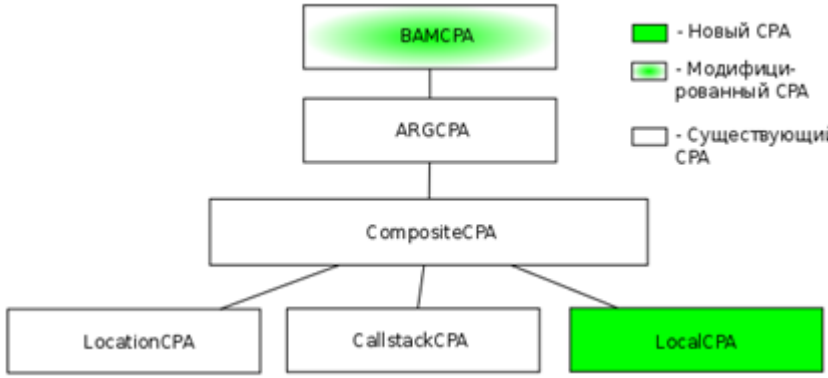


Рисунок 5 — Конфигурация Анализатора разделяемых данных

BAMCPA (Block Abstraction Memorization) [11] отвечает за модульность анализа. Если функция была уже проанализирована с некоторым состоянием до текущего вызова и было сохранено множество конечных состояний, то она не анализируется еще раз, а используются сохраненные состояния. Мы добавили в оригинальный *BAMCPA* возможность обработки рекурсии и некоторые способы для взаимодействия *BAMCPA* и наших новых CPA.

ARGCPA (Abstract Reachability Graph) отвечает за возможность восстановления пути из любого состояния до начального. Он хранит для каждого состояния множество предыдущих и последующих, и предоставляет эти данные для обхода и восстановления пути.

CompositeCPA, LocationCPA и *CallstackCPA* были уже описаны.

LocalCPA отвечает за определение локальности всех переменных, доступных в данной точке программы. Отдельну следует отметить, что под переменными мы также понимаем указатели, то есть, анализ также учитывает информацию о том, куда они указывают. Оператор перехода `transfer` распространяет информацию о разделяемости данных через операторы присваивания и вызовы функций. Например, если указатель `b` указывает на разделяемую область памяти и существует присваивание `a = b`, тогда разделяемость памяти

*b переностится на область памяти *a. После присваивания считается, что a также указывает на разделяемую память. На точках объединения оператор merge объединяет результаты. В случае неопределенности всегда выбирается наихудший результат, то есть, статус shared. Рассмотрим следующий пример:

```
if (condition) {
    a = b;
} else {
    a = c;
}
```

Если b указывает на локальные данные, а c — на разделяемые, то после анализа этого блока кода считается, что a указывает на разделяемые данные.

Результатом этого этапа анализа является список разделяемых переменных для каждой точки программы.

5.2 Конфигурация CPA для Анализатора примитивов синхронизации

Анализатор блокировок используется для определения множества захваченных блокировок для каждого использования разделяемых данных, которые были представлены на предыдущем этапе анализа.

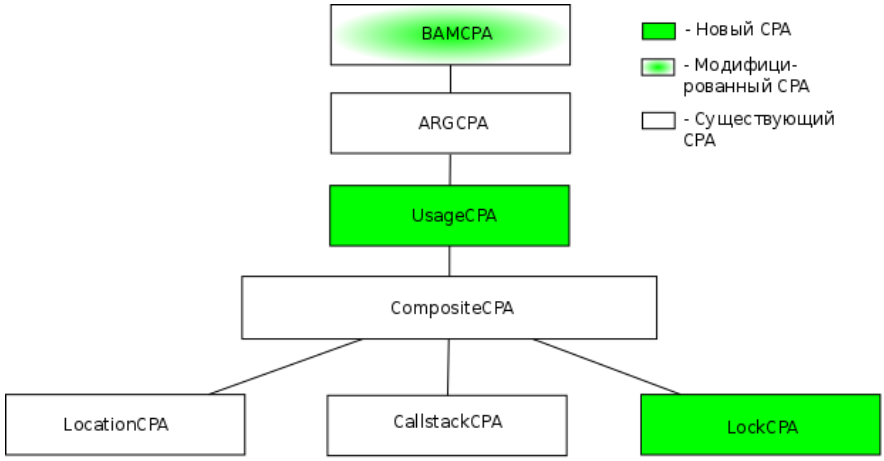


Рисунок 6 — Конфигурация Анализатора примитивов синхронизации

BAMCPA, *ARGCPA*, *LocationCPA* и *CallstackCPA* являются теми же самыми.

UsageCPA собирает статистику использования данных. Оператор transfer определяет переменные, использованные в выражении в некотором доступе к

данным, сохраняет стек вызовов и множество захваченных блокировок для каждого использования.

В конце анализа мы получаем информацию обо всех использованиях для каждой разделяемой переменной. Каждое использование состоит из:

- множества захваченных блокировок;
- стека вызовов функций;
- номера строки;
- типа дуги ГПУ (вызов функции, присваивание и т.д.);
- тип доступа (чтение или запись).

LockCPA следит за множеством захваченных блокировок. Его состояние содержит множество блокировок, которые были захвачены на текущем пути анализа. Для каждой блокировки содержится информация о:

- имя блокировки;
- счетчик рекурсивного захвата;
- стек вызовов функций.

Оператор перехода меняет состояние, которое содержит множество захваченных блокировок. Когда вызывается функция захвата блокировки, соответствующая блокировка добавляется во множество или увеличивается счетчик, если она уже присутствует. При вызове функции освобождения функции счетчик уменьшается, а если он становится равным нулю, блокировка удаляется из множества.

Состояния всех CPA в этой конфигурации никогда не объединяются. Анализ останавливается, если это состояние уже было проанализировано.

5.3 Аннотации

Аннотации используются для более точной настройки метода на специфический код. Всего есть три класса аннотаций:

- Аннотации влияния функции на примитивы синхронизации.
- Аннотации влияния функции на разделяемость данных.
- Аннотации целевых данных.

Рассмотрим следующий пример для объяснения первого типа аннотации.

```
int f() {
  if (isGlobalPointer) {
    lock();
  }
  (*pointer)++;
  if (isGlobalPointer) {
    unlock();
```

```
  }
}
```

В этом примере увеличение разделяемого счетчика всегда происходит под блокировкой. В случае же локального указателя захват блокировки не нужен. Наш анализ рассматривает четыре пути, так как для каждого из двух условных операторов анализируются ветви then и else. Два из этих путей оканчиваются с захваченной блокировкой, а оставшиеся два оканчиваются в состоянии с пустым множеством блокировок. Пара состояний с захваченной блокировкой является недостижимой, так как условия в условных операторах одинаковы.

Такие ситуации не так часто происходят, но каждая из них порождает значительное число ложных сообщений об ошибке, так как финальное состояние в функции с захваченной блокировкой сильно влияет на дальнейший пути анализа. Аннотации функций используются для того, чтобы разобраться с такими случаями. Это лишь способ подсказать анализу, что функция всегда освобождает или захватывает блокировку.

В приведенным выше примере достаточно добавить аннотацию, что функция f всегда освобождает блокировку.

Аннотации описывают функции в терминах состояний *LockCPA*. После того, как функция была проанализирована, состояние уточняется в соответствии с аннотацией.

В данный момент используется 4 типа аннотаций используются:

- Захват блокировки — функция всегда захватывает блокировку.
- Освобождение блокировки — функция всегда освобождает блокировку.
- Сброс блокировки — если блокировка может быть захвачена несколько раз рекурсивно, функция полностью освобождает ее.
- Восстановление блокировки — функция может модифицировать множество блокировок, но все изменения будут забыты при выходе из функции.

К этому типу аннотаций можно добавить конфигурацию блокировок. Поддерживается возможность определить функции захвата, освобождения и сброса, а также глубину рекурсивного захвата. Эти аннотации обрабатываются в *LockCPA*.

Следующий тип аннотаций описывает влияние на разделяемые данные. Функция может вернуть разделяемые данные или инициализировать указатель, передаваемый, как аргумент, локальными данными. Также данные могут стать разделяемыми после вызова функции. Все эти случаи могут быть специфицированы аннотациями, чтобы повысить точность анализа. В данный момент поддерживаются только описание функций, возвращающих локальные данные. Эти аннотации обрабатываются в *LocalCPA*.

Третий тип аннотаций используется для установки равенства между переменными так, чтобы они рассматривались, как одинаковые данные. Это требуется, например, для списков, в которых элементы обычно имеют одинаковые имена *next*. Если не удается различить элементы различных списков, будет выдано большое количество ложных предупреждений, так как доступ к различным спискам может защищаться различными множествами блокировок. Поэтому мы связываем переменные, представляющие элементы списка, с самим списком. Для этой цели конфигурация содержит функции, которые используются для работы со списком. Например, выражение `e = getElement(list)` связывает переменную *e* с переменной *list*, передаваемой как параметр. Эти аннотации обрабатываются в *UsageCPA*.

Для операционной системы реального времени, на которой метод был апробирован, было аннотировано около 2% функций, что в абсолютных величинах составило 90 функций.

6. Интеграция инструмента

Метод CPALockator был интегрирован в систему LDV (Linux Driver Verification), разработанную в рамках проекта по верификации драйверов операционной системы Linux (см. Рис. 7) [12].

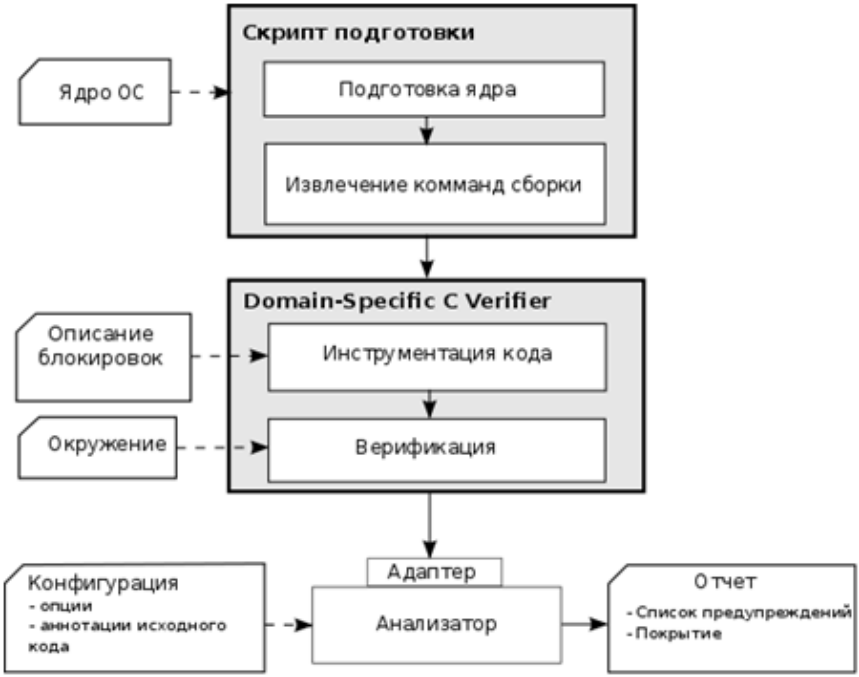


Рисунок 7 — Интеграция с архитектурой LDV

Сначала подготавливается ядро операционной системы. В течение этой стадии вызовы компилятора заменяются на вызовы нашего инструмента для извлечения команд сборки. Затем поток команд сборки извлекается с помощью специальных скриптов. Этот поток передается в компонент DSCV (Domain-Specific C Verifier). Он инструментирует исходный код, используя описания блокировок. Например, он заменяет макросы, используемые для захвата и освобождения блокировок на вызовы модельных функций, аннотированных в конфигурации. Это делается из-за того, что макросы могут быть развернуты в очень сложную последовательность команд, тогда как модельные функции анализировать значительно проще.

После этого включается модель окружения. Она представлена функцией main, содержащей системные вызовы и обработчики прерываний. Считается, что все они могут выполняться параллельно.

После всей подготовки исходный код анализируется нашим инструментом поиска гонок. Он генерирует отчет, содержащий список предупреждений с детальной информацией о каждом из них.

Для визуализации трасс ошибок используется другой компонент LDV. Когда инструмент генерирует предупреждение о состоянии гонки, оно должно быть показано пользователю. Более того, пользователь должен иметь возможность проверить, является ли это предупреждение истинной ошибкой или ложным предупреждением. Таким образом, необходимо представлять трассу ошибки и ее связь с исходным кодом. Состояние гонки характеризуется по крайней мере двумя использованиями с непересекающимися наборами блокировок. Визуализированная трасса ошибки содержит стек вызовов функций для двух использований с точками захватов блокировок. Визуализатор преобразует данные, полученные от верификатора и связывает их с исходным кодом. Для представления результатов генерируется HTML отчет. Главная страница отчета содержит общую статистику (Таблица 1).

Табл. 1 — пример отчета для драйвера Linux floppy.ko

Статистика	Общая	Предупреждения
Глобальные:	195	29
Переменные	122	23
Указатели	73	6
Локальные:	3	0
Переменные	0	0

Указатели	3	0
Поля структур:	118	24
Переменные	105	24
Указатели	13	0
Всего:	316	53

На ней отображены общее число переменных для каждой из трех категорий: глобальные, локальные и поля структур. Во втором столбце указано число переменных, для которых были получены предупреждения. Обозначение «Указатели» означает доступ по указателю, а «Переменные» — доступ к самой переменной. После этого идет список всех найденных блокировок. После этого расположен список всех предупреждений, для каждого из которых содержится пара использований с непересекающимся множеством блокировок.

Пример выдаваемого предупреждения об ошибке представлен на рисунках 8-9.

Source code

Race_example.c

```
 1 #line 1 "../cil-files/Race_example.c"
 2 int global;
 3
 4 int print() {
 5   if (global % 2 == 0) {
 6           printf("global is even: %d", global);
 7     } else {
 8           printf("global is odd: %d", global);
 9   }
10 }
11
12 int increase() {
13   lock();
14   global++;
15   unlock();
16 }
17
18 int main() {
19   switch(undef_int()) {
20           case 0:
21                   print();
22                   break;
23
24           case 1:
25                   increase();
26                   break;
27     }
28 }
```

Рисунок 8 — Пример исходного кода. Параллельное выполнение print() и increase() может привести к состоянию гонки

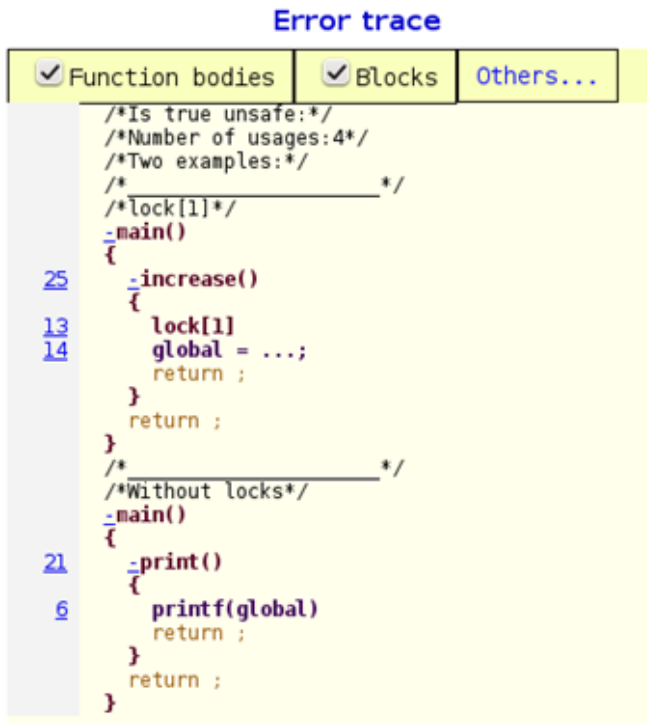


Рисунок 9 — Пример сообщения об ошибке. Трасса ошибки с точками захвата блокировок и точками вызова функций, каждая из которых ссылается на соответствующую строку исходного кода (см. рис. 8)

Еще раз подчеркнем, что мы используем модель параллелизма, и две функции, вызываемые из функции *main*, считаются выполняющимися параллельно.

Функция print печатает информацию о переменной `global`, а `increase` увеличивает ее значение под блокировкой. В таком примере возможно состояние гонки, потому что функция `increase` может записывать переменную одновременно с ее проверкой в функции `print`. Наш инструмент генерирует предупреждение для переменной `global` с трассой ошибки, показанной на рисунке 9.

В первой строке указано общее число найденных использований. Только два из них визуализируются далее. Первое использование означает, что доступ к `global` был в вызове функции. Второе происходит в присваивании при захваченной блокировке `lock`.

Кроме того имеется опция для генерации покрытия исходного кода. Оно показывает код, который был проанализирован верификатором, и его связь с общим кодом ядра.

7. Результаты

Инструмент был применен для анализа операционной системы реального времени, которая уже была протестирована и находилась в использовании уже несколько лет. Общий объем кода был около 200 000 строк кода, но только 50 000 были проанализированы. Основной причиной является то, что не все возможные функции были включены в функцию main. Мы обнаружили 20 новых состояний гонки, которые были подтверждены разработчиками. Общее число предупреждений было 139. Без Анализатора разделяемых данных число предупреждений было 378. В данный момент большая часть ложных сообщений об ошибках происходит из-за неточности в анализе выражений. Например, сейчас анализ не полностью поддерживает условные операторы.

Один запуск инструмента на машине с 6 Гб памяти и восьмиядерным процессором 2.80 ГГц занял 3 минуты на ядре описанной выше операционной системы. Кроме этого, был пробный запуск инструмента на ядре операционной системы Linux 3.8 на директории `drivers`. Общее число проанализированных модулей было около 3500. Инструмент сообщил о 900 предупрежденях. Некоторые из них были проанализированы, и одна реальная ошибка была найдена, однако в текущей версии эта ошибка была уже исправлена.

8. Похожие работы

Мы не будем рассматривать методы динамического анализа, которые имеют свои преимущества. Также мы рассмотрим только методы анализа программ на языке Си.

8.1 Тяжеловесные техники анализа

Традиционно идеи тяжеловесных подходов применяются к небольшим программам. Зачастую в таких подходах предполагается, что взаимодействие между потоками производится только через глобальные переменные. Так как в реальных программных системах число предупреждений, которое приходится на поля структур, сопоставимо с предупреждениями, полученными для глобальных переменных, а иногда даже выше, то для нас было важно учитывать в анализе поля структур.

Еще одной важной особенностью тяжеловесных инструментов, нацеленных на анализ многопоточного программного обеспечения, в том, что они проверяют другое свойство, а именно свойство достижимости ошибочной метки в условиях параллельного окружения. Для того чтобы проверить наличие состояния гонки, необходимо дополнительно изощряться, чтобы записать это

требование, как условие достижимости. Например, после каждого присваивания в разделяемую переменную добавить проверку, что сейчас в данной переменной находится именно то, что мы только что туда записали. Это значительно усложняет анализ. Более того, операции захвата и освобождения блокировки тоже могут быть представлены, как операции с переменными. Например, в инструменте ESBMC [13] моделирование примитивов синхронизации происходит с помощью конструкции assume(P), которая рассматривает пути выполнения программы, для которых в данный момент выполнен предикат P. Операция захвата блокировки m представляет собой атомарную последовательность assume(m==0); m=1. Моделирование захвата блокировки таким образом усложняет анализ, так как теперь приходится учитывать значение еще одной переменной.

Методы тяжеловесного анализа обыкновенно рассматривают все варианты параллельной работы программы, то есть анализируются возможные варианты переключений потоков (англ. interleavings). Одним из подходов является анализ параллельной композиции потоков, при котором так или иначе строится дерево достижимых состояний (ART), которые представляют собой произведение состояний каждого потока. Далее неизбежно возникает проблема комбинаторного взрыва, которую различные инструменты решают по-своему. Например, наблюдение, что состояние гонки проявляется уже при небольшом числе переключений контекста, позволило ограничить число переключений некоторым фиксированным K — подход, получивший название проверка моделей с ограниченным переключением контекста (англ. context-bounded model checking). Примерами инструментов, реализующих методы тяжеловесного анализа, могут быть ESBMC [13] и SATABS [14], которые используют различные подходы к анализу программ (BMC[15] и CEGAR [16, 17] соответственно)

В предыдущих инструментах число потоков было ограничено несколькими экземплярами и известно заранее. Однако большие программные системы, такие как ядро операционной системы, работают со значительно большим количеством потоков. Для анализа таких систем был предложен метод абстракции по счетчику состояний, который реализован, например, в инструменте Boom[18]. Идея метода заключается в том, чтобы рассматривать не состояния каждого потока, а количество потоков в том или ином состоянии. Кроме того, были предложены такие оптимизации как символическое представление, то есть состояние локального потока описывается не конкретным значением, а некоторым символическим. Однако такой подход применим только в случае, если один и тот же код выполняется в несколько потоков. В сложных программных системах большая часть кода, выполняющегося параллельно, является уникальным.

Еще одним вариантом борьбы с большим пространством состояний является использование редукции частичных порядков.

Вторым большим классом тяжеловесных подходов является построение модульной абстракции, реализованное в инструменте Threader [19, 20, 21] или BLAST[22]. Такие подходы не рассматривают все возможные чередования, а строят абстракцию взаимодействия между потоками. Сначала используется неточная модель, которая предполагает, что все потоки могут изменять любые разделяемые данные произвольным образом, затем эта модель уточняется в процессе анализа. Возможно уточнение как множества точек переключения контекста, так и влияния потоков друг на друга. Потенциально такой подход должен дать лучшие результаты нежели те, которые были получены для инструментов, типа ESBMC, однако разработчики приводят результаты только на тестах порядка нескольких сотен строк исходного кода, поэтому остаются вопросы о масштабируемости такого подхода.

Задача верификации многопоточных программ может решаться с помощью трансляции параллельной программы в последовательную с последующей ее верификацией с помощью существующих инструментов или какую-нибудь другую промежуточную форму, например, формулу, которая затем может быть проверена решателем. В основном, методы трансляции опираются на контекстно-ограниченную проверку моделей (англ. Context-Bounded Model Checking [23, 24, 25, 26]) - подход к проверке многопоточных программ, при котором число активных потоков ограничивается некоторым числом в процессе верификации. Для преобразования параллельной программы фиксируется число возможных потоков n и число возможных переключений контекста K. Требуется построить такую последовательную программу, которая покрывала бы все возможные варианты выполнения n потоков, каждый из которых прерывался бы не более K раз.

Некоторые тяжеловесные инструменты предпочитают работать не с программой на языке Си, а на некотором более простом языке. В таком случае сначала применяется трансляция исходной программы в другой язык. Например, для инструмента Storm [27] — это язык Boogie [28], в котором нет таких конструкций, как динамическое выделение памяти, арифметика указателей, преобразование типов. Параллельная программа преобразуется в последовательную, и для нее уже строятся формулы, которые проверяются с помощью SMT-решателя. Для моделирования переключения контекста используется понятие карты памяти, которое означает локальную копию всей памяти, занимаемой глобальными переменными. В данном подходе для каждого потока создается копия карты памяти. Переключение контекста моделируется переключением работы с одной картой памяти на другую. Все потоки выполняются один за другим. Для того чтобы повысить масштабируемость метода, применяется слайсинг по полям структур. Такая идея базируется на предположении, что для проверки конкретного свойства необходимо наблюдение за очень небольшим количеством полей. Построение множества отслеживаемых полей производится с помощью метода CEGAR. Для верификации циклы разворачиваются на конечную глубину, что может

привести к потере точности, а вызовы функций заменяются на их тела. Кроме того, принимается предположение о том, что в системных программах редко встречается рекурсия, чтобы можно было заменить вызов функции на ее тело.

В статье [24] авторы предлагают три варианта трансляции параллельной программы в логическую формулу.

Первый вариант - это метод Explicit Program Counter (EMC) [29], использование явного счетчика команд. В этом случае состояние программы включает в себя все локальные переменные и счетчик команд для каждого потока. В возможных точках переключения контекста вставляется специальный оператор недетерминированного выбора, который выбирает поток для переключения и корректную строку кода, на которой закончилось его выполнение в прошлый раз. Это самый простой способ, но он требует много времени и памяти.

Второй вариант - Lal-reps(LR) [23]. В этом случае каждый поток символически выполняется независимо от других, с учетом того, что значения глобальных переменных могут измениться случайным образом. Задача упрощается тем, что заранее задается, когда происходит переключения контекста, а значит, задаются и точки, когда глобальным переменным присваиваются новые значения.

Третий вариант - LMP [30]. Его основное отличие от предыдущего подхода в том, что он присваивает глобальным переменным не случайные значения, а только те, которые возможны во время выполнения программы. После переключения контекста текущее локальное состояние потока сбрасывается, и чтобы его восстановить приходится заново исполнять поток из начального состояния с учетом того, что становятся известны значения глобальных переменных до переключения.

Попытки применения подхода с контекстно-ограничиваемыми проверками показали, что он плохо масштабируется и на реальных программах пока не может быть использован [27]. Авторы сравнивают свой способ верификации, основанный на генерации логических формул со способом логической проверки моделей, при котором все переменные программы преобразуются к бинарным, для которых уже генерируются формулы для проверки решателем [31]. Такой подход, например, использован в статье [18]. Эксперименты показали, что эти две парагдимы сильно отличаются, и для них нужны различные способы трансляции. LR - лучший способ для проверки логических формул. Что интересно, самый простой способ EMC оказался лучше, чем LMP почти во всех тестах.

Еще два варианта трансляции рассматриваются в статье [32]. В ней представлен подход ленивой трансляции параллельной программы в последовательную с ограниченным числом переключений контекста. Важной особенностью описываемого метода является то, что множество достижимых состояний полученной программы является тем же, что и для исходной программы. Известные методы сохраняют локальное состояние потока перед

переключением контекста. Для того чтобы избежать этого, предлагается использовать копии глобальных переменных для каждого из переключений контекста. Таким образом, после каждого переключения ведется работа со своей картой памяти, а условия на равенство их между собой будут выписаны позднее. При ленивой трансляции выполнение происходит постепенно, с реальным переключением. Инициализируется только первая карта памяти. Далее, при переключении контекста она копируется во вторую и происходит анализ второго потока. При переключении контекста обратно в первый поток происходит его выполнение заново, но уже до следующей точки переключения контекста.

Кроме того, была реализована вторая стратегия, противоположная, - нетерпеливая трансляция, которая предполагает, что поток выполняется сразу от начала и до конца. В начале карты памяти инициализируются произвольными значениями. В точках переключения контекста происходит смена карты памяти. Для сравнения этих стратегий использовались несколько тестов, которые показали, что нетерпеливая трансляция тратит больше времени и памяти на проверку.

8.2 Методы легковесного статического анализа кода

Легковесный статический анализ традиционно применяется на больших программных системах, на которых нельзя провести формальную проверку моделей. Он отличается большей скоростью, но не может претендовать на формальное доказательство отсутствия ошибок.

Статья [6] представляет инструмент Locksmith для статического поиска гонок. Этот метод является наиболее близким к нашему. Он также основан на алгоритме Lockset, но имеет другие подходы для анализа блокировок и разделяемых данных.

Сначала происходит построение графа потока данных — такого графа, в узлах которого стоят переменные, а дуги ведут в ту область памяти, на которую они указывают. Этот анализ чувствителен к полям структур (field-sensitive), это значит, что каждое поле моделируется независимо от других. Такой способ точного анализа потоков данных неизбежно приводит к проблемам с адресной арифметикой. Например, возникают различные проблемы с моделированием указателей типа void, и в статье было рассмотрено несколько вариантов их решения. Важно отметить, что блокировки также считаются данными, для которых строится граф потоков данных. Такой подход имеет как плюсы, так и минусы. Имея граф потоков данных, инструмент может понять, что две различные переменные указывают на одну блокировку и тем самым повысить точность анализа. Однако, если блокировке соответствует очень сложный граф, например, она берется из поля несколько раз вложенной структуры, которая берется из списка, то анализ этого графа сам по себе становится нетривиальной задачей.

Анализ разделяемых данных определяет те области памяти, которые доступны из нескольких потоков. Важной особенностью такого алгоритма определения разделяемых данных является то, что разделяемые переменные фиксируются для всей программы (location-insensitive), и становится невозможным учитывать случаи, в которых некоторая локальная переменная становится разделяемой после некоторых действий.

Еще одним инструментом, который основан на алгоритме Lockset, является Relay[33]. Он описывает изменения во множествах блокировок и доступах к областям памяти относительно точки входа в функцию. Модель потоков очень похожа на вариант, используемый в методе CPALockator — некоторое количество функций, отобранных вручную, считается выполняемыми параллельно. После анализа может быть применен ряд эвристик, которые удалят некоторые предупреждения, возникшие из-за неточной модели окружения.

Одним из важных отличий этого подхода от подхода CPALockator является определение разделяемых данных с учетом алиасов, то есть, данные являются разделяемыми, если равны адресные выражения или одно адресное выражение входит в множество возможных (may)-алиасов другого адресного выражения. Блокировки являются объектами, которые могут быть захвачены и освобождены специальными функциями. Для каждой функции вычисляется множество захваченных блокировок относительно точки входа в функцию. Оно представляет собой пару из точно захватываемых блокировок и возможно освобождаемых. Важно отметить, что блокировки также являются данными, вычисляемыми относительно точки входа в программу. Поэтому, если блокировка является аргументом вызова функции, то для дальнейшего анализа она будет обновлена в терминах переменных вызываемой функции. Наконец, для каждой функции сохраняется множество доступов с захваченными блокировками и эффект функции, который включает в себя относительное множество блокировок.

Метод, предложенный в [34], фокусируется на быстром вычислении must-алиасов, то есть множества таких переменных, которые являются алиасами при любых путях выполнения программы. В этом подходе используются три модели потоков. Первый напоминает модель в подходе CPALockator — каждая функция выполняется параллельно с другими. Вторая модель основана на функциях работы с потоками, такими, как `fork` и `join`. Параллельное выполнение начинается после создания нового потока операцией `fork`, а заканчивается на точке слияния (операция `join`). Третья модель отличается от предыдущих отсутствием точек слияния и ограничением на количество задач, способных выполняться на одном потоке. Данными являются области памяти, а равные области памяти определяются множеством must-алиасов. Блокировки также определяются множеством must-алиасов. Таким образом, основным отличием от нашего метода является акцент на анализе алиасов.

Статья [35] является развитием предыдущей и посвящена проблеме построения графа потока управления для многопоточной программы, если в ней имеются вызовы по функциональным указателям, а также проблеме анализа алиасов в этом случае. В случае рекурсии граф потока управления раскручивается до тех пор, пока не будет найдена неподвижная точка в терминах разбиения Стинсгаарда [36]. Полученный граф состояний обходится и анализируется на предмет наличия состояний гонок с помощью стандартного алгоритма Lockset. Разделяемые переменные определяются, как алиасы к глобальным переменным.

9. Заключение

В данной статье мы предложили новый легковесный метод для поиска состояний гонок, который реализован на основе инструмента CPAchecker. Метод CPALockator учитывает такую специфику ядра операционной системы, как сложный параллелизм, примитивы синхронизации и активное использование адресной арифметики. Еще одной особенностью является возможность масштабируемости на большие объемы исходного кода. Основными отличиями описанного метода является модель памяти, модель параллелизма и способ определения разделяемых данных.

Основной проблемой данного метода является большое количество ложных предупреждений об ошибках. Чтобы уменьшить их количество, планируется интегрировать идеи метода CEGAR (Counterexample Guided Abstraction Refinement), который позволяет учесть условия с помощью предикатной абстракции. В случае задачи поиска состояний гонки метод CEGAR должен быть адаптирован для анализа двух потоков.

Еще одним направлением дальнейшего развития является аппробация предложенного метода на ядре операционной системы Linux.

Список литературы

[1]. Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, A Study of Linux File System Evolution, 11th USENIX Conference on File and Storage Technologies (FAST ’13)

[2]. Мутилин В.С., Новиков Е.М., Хорошилов А.В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, том 22, С. 349-374, 2012.

[3]. Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[4]. Герлиц Е.А., Кулямин В.В., Максимов А.В., Петренко А.К., Хорошилов А.В., Цыварев А.В. Тестирование операционных систем. Труды ИСП РАН, том 26, С. 73-107, 2014.

[5]. John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk. Effective Data-Race Detection for the Kernel Operating System Design and Implementation (OSDI’10), 2010, USENIX.

[6]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher. Model checking concurrent linux device drivers. ASE'07, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp 501-504, ACM, New York, NY, USA, 2007.

[7]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Practical Static Race Detection for C, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 33(1):Article 3, January 2011.

[8]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 320 - 331, ACM New York, 2006

[9]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher, Model Checking Concurrent Linux Device Drivers, ASE'07, November 4–9, 2007.

[10]. Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz, Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[11]. Daniel Wonisch, Block Abstract Memorization for CPAchecker, TACAS 2012, LNCS 7214, pp. 531-533.

[12]. Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е. Архитектура Linux Driver Verification. Труды ИСП РАН, том 20, 2011. С. 163-187.

[13]. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)

[14]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 570-574, 2005.

[15]. CBMC A Tool for Checking ANSI-C Programs, Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pages 193–207, 1999.

[16]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), pp. 154-169, 2000.

[17]. Хорошилов А.В., Мандрыкин М.У., Мутилин В.С. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды ИСП РАН, том 24, С. 219-292, 2013.

[18]. Gerard Basler, Michele Mazzucchi1, Thomas Wahl, Daniel Kroening. Symbolic Counter Abstraction for Concurrent Software CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp. 64-78, Springer-Verlag, Berlin, Heidelberg, 2009.

[19]. A. Gupta, C. Popeea, A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.

[20]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: a constraint-based verifier for multi-threaded programs In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.

[21]. C. Popeea, A. Rybalchenko. Threader: a verifier for multi-threaded programs In Proceedings of the 19th International Conference on Tools and Algorithms for the

Construction and Analysis of Systems (TACAS 2013), LNCS, vol. 7795, pp. 633-636, 2013.

[22]. A. Malkis, A. Podelski, A. Rybalchenko. Thread-modular counterexample-guided abstraction refinement, SAS'10 Proceedings of the 17th international conference on Static analysis, pp. 356-372, Springer-Verlag, Berlin, Heidelberg, 2010.

[23]. S. Qadeer, D. Wu. KISS: Keep it simple and sequential. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04), pp. 14 - 24, 2004.

[24]. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In Conf. on Computer-Aided Verification (CAV), pages 82–97, 2005.

[25]. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In PLDI, pages 446–455, 2007.

[26]. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In Intl. SPIN Workshop on Model Checking Software, pages 114–133, 2008.

[27]. Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp 509-524, Springer-Verlag Berlin, Heidelberg, 2009

[28]. R. DeLine, K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[29]. Naghmeh Ghafari1, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: an empirical evaluation. SPIN'10 Proceedings of the 17th international SPIN conference on Model checking software, pp. 227-244, Springer-Verlag Berlin, Heidelberg, 2010

[30]. S. La Torre, P. Madhusudan, G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), pp. 211-222, 2009.

[31]. Gerard Charly Basler, Model Checking Boolean Programs, abhandlung zur Erlangung des titels Doktor der Wissenschaften der ETH Zurich, 28 October 1978.

[32]. S. La Torre, P. Madhusudan, G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09), LNCS 5643, pp. 477-492, 2009.

[33]. Jan Wen Voung, Ranjit Jhala, Sorin Lerner, RELAY: Static Race Detection on Millions of Lines of Code. ESEC/FSE'07, 2007

[34]. Kahlon V., Yang Y., Sankaranarayanan S., Gupta A.: Fast and accurate static data-race detection for concurrent programs. In: CAV'07. LNCS, vol. 4590, pp. 226-239. Springer (2007)

[35]. Vineet Kahlon, NishantSinha, Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. ESEC/FSE '09 Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 13-22 ACM New York, NY, USA, 2009

[36]. B. Steensgaard. Points-to analysis in almost linear time. Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96), pp. 32-41, 1996

Lightweight Static Analysis for Data Race Detection in Operating System Kernels

[1]P.S. Andrianov <andrianov@ispras.ru>
[1]V.S. Mutilin <mutilin@ispras.ru>
[1,2,3,4]A.V. Khoroshilov <khoroshilov@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper presents an approach to lightweight static data race detection called CPALockator. It takes into account the specifics of operating system kernels such as complex parallelism and kernel specifics synchronization mechanisms. The method is based on the Lockset one but it implements two heuristics that are aimed to reduce amount of false alarms: a memory model and a model of parallelism. We consider locks as synchronization primitives. The main target of our research and evaluation is operating system kernels but the approach may be applied to the other programs as well. The method is based on idea of Configurable Program Analysis (CPA) and was implemented in CPAchecker tool. The implementation of the method was done in two stages. Firstly, the set of shared data is determined for every program location, then the analysis of synchronization primitives is performed. On the second stage, information about all potential accesses to shared variables and acquired synchronization primitives is also collected. After analysis the report with results is created. The tool was applied to the kernel of Real-Time operating system. It has about 200 000 lines of source code. CPALocator found 20 new race conditions, which are confirmed by developers. Also, the tool was launched on the Linux device drivers using the LDV framework for preparing tasks for CPALockator. The future investigations will be related to development more precise thread model and integration with more precise heavyweight methods of static analysis.

Keywords: static analysis, data race, operating system kernel, shared data

DOI: 10.15514/ISPRAS-2015-27(5)-6

For citation: Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Lightweight Static Analysis for Data Race Detection in Operating System Kernels. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 5, 2015, pp. 87-116 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-6.

References

[1]. Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, A Study of Linux File System Evolution, 11th USENIX Conference on File and Storage Technologies (FAST '13)

[2]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analiz tipovyh oshibok v drajverah operacionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349–374, 2012 (in Russian).

[3]. Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[4]. Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. Testirovanie operacionnyh sistem [Testing of Operating Systems]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, pp. 73–107, 2014 (in Russian).

[5]. John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk Effective Data-Race Detection for the Kernel Operating System Design and Implementation (OSDI'10), 2010, USENIX.

[6]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher. Model checking concurrent linux device drivers. ASE'07, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp 501-504, ACM, New York, NY, USA, 2007.

[7]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Practical Static Race Detection for C, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 33(1):Article 3, January 2011.

[8]. Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 320 - 331, ACM New York, 2006

[9]. Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher, Model Checking Concurrent Linux Device Drivers, ASE'07, November 4–9, 2007.

[10]. Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz, Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[11]. Daniel Wonisch, Block Abstract Memorization for CPAchecker, TACAS 2012, LNCS 7214, pp. 531-533.

[12]. 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).

[13]. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)

[14]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 570-574, 2005.

[15]. CBMC A Tool for Checking ANSI-C Programs, Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pages 193–207, 1999.

[16]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), pp. 154-169, 2000.

[17]. Khoroshilov A.V., Mandrykin M.U., Mutilin V.S. Vvedenie v metod CEGAR — utochnenie abstrakcii po kontrprimeram [Introduction to CEGAR — Counter-Example Guided Abstraction Refinement], Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian)

[18]. Gerard Basler, Michele Mazzucchi1, Thomas Wahl, Daniel Kroening. Symbolic Counter Abstraction for Concurrent Software CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp. 64-78, Springer-Verlag, Berlin, Heidelberg, 2009.

[19]. A. Gupta, C. Popeea, A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.

[20]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: a constraint-based verifier for multi-threaded programs In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.

[21]. C. Popeea, A. Rybalchenko. Threader: a verifier for multi-threaded programs In Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013), LNCS, vol. 7795, pp. 633-636, 2013.

[22]. A. Malkis, A. Podelski, A. Rybalchenko. Thread-modular counterexample-guided abstraction refinement, SAS'10 Proceedings of the 17th international conference on Static analysis, pp. 356-372, Springer-Verlag, Berlin, Heidelberg, 2010.

[23]. S. Qadeer, D. Wu. KISS: Keep it simple and sequential. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04), pp. 14 - 24, 2004.

[24]. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In Conf. on Computer-Aided Verification (CAV), pages 82–97, 2005.

[25]. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In PLDI, pages 446–455, 2007.

[26]. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In Intl. SPIN Workshop on Model Checking Software, pages 114–133, 2008.

[27]. Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification, pp 509-524, Springer-Verlag Berlin, Heidelberg, 2009

[28]. R. DeLine, K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[29]. Naghmeh Ghafari1, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: an empirical evaluation. SPIN'10 Proceedings of the 17th international SPIN conference on Model checking software, pp. 227-244, Springer-Verlag Berlin, Heidelberg, 2010

[30]. S. La Torre, P. Madhusudan, G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), pp. 211-222, 2009.

[31]. Gerard Charly Basler, Model Checking Boolean Programs, abhandlung zur Erlangung des titels Doktor der Wissenschaften der ETH Zurich, 28 October 1978.

[32]. S. La Torre, P. Madhusudan, G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09), LNCS 5643, pp. 477-492, 2009.

[33]. Jan Wen Voung, Ranjit Jhala, Sorin Lerner, RELAY: Static Race Detection on Millions of Lines of Code. ESEC/FSE'07, 2007

[34]. Kahlon V., Yang Y., Sankaranarayanan S., Gupta A.: Fast and accurate static data-race detection for concurrent programs. In: CAV'07. LNCS, vol. 4590, pp. 226-239. Springer (2007)

[35]. Vineet Kahlon, NishantSinha, Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. ESEC/FSE '09 Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 13-22 ACM New York, NY, USA, 2009

[36]. B. Steensgaard. Points-to analysis in almost linear time. Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96), pp. 32-41, 1996

Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях[1]

М.У. Мандрыкин <mandrykin@ispras.ru>
В.С. Мутилин <mutilin@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Одна из фундаментальных проблем в современных методах статической верификации программ состоит в точном учете семантики выражений с указателями. От точности анализа данных выражений зависит достоверность вердикта верификации. В данной работе описывается метод верификации с моделями памяти на основе неинтерпретируемых функций, который позволяет анализировать программы, содержащие выражения с указателями, в том числе указателями на структуры, массивы и выражения с адресной арифметикой. Ограничениями метода является конечность размера массивов и конечная глубина рекурсии для динамических структур данных. Предложенный метод был реализован в инструменте CPAchecker, основанном на подходе CEGAR с использованием булевой предикатной абстракции и интерполяции Крейга для получения новых предикатов при уточнении абстракции. Для решения задач проверки выполнимости формулы пути и интерполяции Крейга в CPAchecker используются интерполирующие решатели, поддерживающие бесквантворные теории вещественной или целочисленной линейной арифметики и равенства с неинтерпретируемыми функциями. В разработанном подходе состояние памяти программы представляется в виде неинтерпретируемой функции, отображающей некоторые условные адреса переменных в памяти в их значения. При записи значения по какому-либо адресу происходит смена версии неинтерпретируемой функции, представляющей состояние памяти. Эксперименты проводились на наборах международных соревнований по верификации программ SV-COMP'2016, содержащих практически значимый набор из драйверов устройств ОС Linux. На этих наборах метод показывает приемлемые результаты по времени, сокращая при этом количество упущенных ошибок и ложных предупреждений. Среди возможных дальнейших направлений исследований отметим возможность более точного разбиения на регионы памяти, так что для этих регионов выделяются независимые неинтерпретируемые функции.

1 Исследование проводилось при финансовой поддержке РФФИ в рамках проекта №15-01-03934

Ключевые слова: модель памяти, предикатная абстракция, метод уточнения абстракций по контрпримеру.

DOI: 10.15514/ISPRAS-2015-27(5)-7

Для цитирования: Мандрыкин М.У., Мутилин В.С. Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 117-142. DOI: 10.15514/ISPRAS-2015-27(5)-7.

1. Введение

Одна из фундаментальных проблем в современных методах статической верификации программ состоит в точном учете семантики выражений с указателями. От точности анализа данных выражений зависит достоверность вердикта верификации. В данной работе описывается метод верификации с моделями памяти, учитывающими семантику выражений с указателями.

Для анализа выражений с указателями существует несколько известных подходов, применяемых в инструментах статической верификации.

Подход ограничиваемой проверки моделей (от англ. Bounded Model Checking, BMC) [1, 2] основывается на разворачивании циклов программы на конечное фиксированное число шагов k и последующей проверке возможности нарушения проверяемого свойства на глубине, не большей, чем эти k шагов. Процесс верификации повторяется для все больших значений числа k, пока не будут обнаружены все возможные нарушения проверяемого свойства или при помощи специальных проверок не будет доказано, что выполнение циклов более чем k раз в данной программе невозможно. Принудительное ограничение сверху числа k является основным ограничением на применимость этого метода для формального доказательства корректности.

В методе BMC ограничение сверху числа k существенно используется при моделировании памяти. Так как программа представляется конечным множеством конечных путей, то на каждом из них можно раскрыть выражения с разыменованием указателей, подставив вместо переменных-указателей выражения, определяющие их значения на этом пути [3]. Так как путь конечен, то можно с помощью таких подстановок прийти к выражению, в котором отсутствуют указатели. Исключения составляют лишь массивы, для моделирования которых вводятся неинтерпретируемые функции.

Метод BMC позволяет поддерживать для языка программирования широчайший набор конструкций, включая выражения с указателями, в том числе указатели на структуры, массивы, адресную арифметику, но так как рассматриваются только конечные пути, то рекурсивные структуры данных поддерживаются на конечную глубину (см. Табл. 1). На сегодняшний день метод BMC является наилучшим методом для поиска неглубоких (в смысле длины пути выполнения) ошибок. Инструменты, реализующие этот метод,

такие как CBMC[1] и F-SOFT[4], с успехом используются для поиска ошибок при разработке полупроводниковых устройств, для которых схемы часто моделируются программой на языке высокого уровня (например, C), а также в системном программном обеспечении, например, в автомобильной промышленности [5].

Вторая группа методов основана на подходе уточнения абстракции по контрпримерам (от англ. CounterExample-Guided Abstraction Refinement, CEGAR) [6, 7]. Инструменты, основанные на данном подходе, пытаются доказать некоторые свойства исходно заданной системы, предварительно ее упростив. Упрощенная система при этом, как правило, не обладает всеми свойствами исходной и поэтому в дальнейшем может потребоваться процесс ее уточнения. В общем случае к упрощенной системе — абстракции — предъявляется требование корректности (англ. soundness). Оно заключается в том, что свойства, доказанные для абстракции, должны быть верны и на исходной системе. Однако не все свойства, верные для исходной системы, верны для абстракции. В абстракции может быть обнаружено нарушение одного из проверяемых свойств, то есть контрпример, который не является осуществимым в исходной системе. Для исключения таких контрпримеров используется уточнение абстракции.

Процесс верификации с использованием уточнения абстракции по контрпримерам начинается с построения неточной (грубой) абстракции исходной системы и продолжается одним или несколькими уточнениями. Когда в абстракции обнаруживается контрпример, инструмент проверяет его на осуществимость в исходной системе (то есть проверяет, является ли данное нарушение подлинным или возникает вследствие неточности построенной абстракции). Если нарушение осуществимо, то верификация завершается и выдается сообщение об ошибке, если нет, то доказательство неосуществимости контрпримера используется для уточнения абстракции, и ее проверка снова повторяется.

В качестве абстракции программы будем рассматривать *предикатные абстракции* [8]. Предикатная абстракция программы основывается на разбиении всего множества состояний программы на подмножества с одинаковыми наборами значений выбранных предикатов. При этом предикаты в абстракции могут комбинироваться либо только с использованием конъюнкций (в таком случае предикатная абстракция называется *декартовой*), либо в том числе с использованием дизъюнкций (в таком случае абстракция называется *булевой*). Проверка контрпримера и пересчет абстракции осуществляется на основе представления последовательностей инструкций исходной программы в виде некоторых логических формул. Для решения формул используются различные решатели формул в теориях (англ. Satisfiability Modulo Theories, SMT). Для уточнения абстракции новые предикаты извлекаются из невыполнимых формул. Среди способов их извлечения имеются *синтаксические* [9] методы, а также *интерполяция*

Крейга [10, 11]. Для поиска интерполянтов Крейга на практике применяются интерполирующие решатели формул в теориях, такие как CSISAT [12] и MATHSAT [13, 14].

Рассмотрим методы реализации поддержки указателей в CEGAR-инструментах. Первый метод основан на анализе *алиасов*. В программировании словом алиас (от англ. alias — имя, прозвище) описывается ситуация, при которой какая-либо ячейка с данными в памяти программы оказывается доступна в исходном тексте (коде) под различными обозначениями (именами). Таким образом, изменение данных с использованием одного из таких обозначений неявно ведет к изменению значений, доступных по всем остальным обозначениям той же ячейки памяти.

Информация об алиасах используется для генерации ограничений, накладываемых на состояние памяти операцией присваивания, в формулах контрпримера и при пересчете абстракции. Для каждого возможного алиаса цели операции присваивания генерируется проверка, является ли его адрес равным адресу цели. В зависимости от результата этой проверки с помощью логических формул выражается либо обновление соответствующего выражения-алиаса, либо сохранение прежнего значения этого выражения.

Такой метод был реализован в инструменте BLAST 2.5 [15-17]. Анализ алиасов в нем осуществляется в терминах анализа Андерсена [18]. Для хранения и выполнения запросов к информации об алиасах используется представление BDD и алгоритм, похожий (но не совпадающий) с алгоритмом в [19]. Для того чтобы выписывать ограничения в формуле на состояния памяти, которые доступны через несколько разыменований, рассматривается *замыкание алиасов* по операции разыменования. Для каждой переменной программы выписываются алиасы на не более чем заданное число ее разыменований. Такой метод ограничивает доступы к структурам на конечную глубину (см. Табл. 1).

В инструменте BLAST 2.7 [20] метод улучшен так, что в формулу для оператора присваивания добавляются только алиасы, влияющие на ее выполнимость, которые определяются по выражениям, использованным в последующих операторах на пути. Таким образом стало возможно поддерживать произвольную глубину разыменований для указателей и структур. Однако в инструменте все еще не поддерживаются массивы, адресная арифметика и рекурсивные структуры данных.

Анализ рекурсивных структур данных (shape analysis) — это метод статического анализа программ, который позволяет находить и проверять свойства динамически выделяемых структур данных. Он обладает достаточно большой точностью при анализе различных соотношений между указателями в программе и динамически размещаемыми в памяти структурами данных, в том числе при анализе выражений с указателями. Однако, имея высокую точность, этот подход является достаточно плохо масштабируемым. Так, например, представленный на соревновании SV-COMP 2014 [21] инструмент

PREDATOR [22], который реализует данный вид анализа на основе символьных графов памяти (англ. Symbolic Memory Graphs, SMG), набрал 50 баллов из 2766 в категории, соответствующей реальным драйверам ОС Linux. Разработка другого инструмента, BLAST, реализующего *ленивый анализ рекурсивных структур данных* [23] на основе предикатов троичной логики (three-valued logic) [24], к моменту написания данной статьи не велась уже в течение нескольких лет.

Таблица 1. Сравнение инструментов статической верификации.

Инструмент/подход	Указатели	Структуры	Массивы	Рек. структуры данных	Адресная арифметика	Масштабир.
Ограничиваемая проверка моделей (BMC)	+	+	+	± (конечная глубина)	+	-
BLAST 2.5, алиасы с замыканием	± (конечная глубина)	± (конечная глубина)	-	-	-	+
BLAST 2.7, алиасы с бесконечным замыканием	+	+	-	± (конечная глубина)	-	+
BLAST, ленивый анализ рекурсивных структур данных	+	+	-	+	-	?
CPAchecker, модель памяти на основе неинтерпретируемых функций	+	+	± (огранич. размер)	± (конечная глубина)	+	+

Третий метод, который предложен в данной работе, основан на использовании теории неинтерпретируемых функций без кванторов. По определению неинтерпретируемая функция — это функция, для которой задано только имя и количество аргументов. Соответственно, для одних и тех же значений аргументов функция выдает один и тот же результат. Неинтерпретируемые функции используются для представления состояния памяти программы как отображения некоторых условных адресов переменных в памяти в их значения. В зависимости от конкретной реализации данный подход позволяет достигать различной точности анализа выражений с указателями, при этом использование теории неинтерпретируемых функций дает возможность его применения только для объектов с наперед заданными размерами. Данный подход обладает ограничениями по сравнению с анализом рекурсивных структур данных для анализа динамических структур данных, так как в нем отсутствуют средства сворачивания длинных последовательностей элементов, однако для структур данных относительно небольшого фиксированного размера его использование может быть вполне оправдано.

2. Обзор метода

Для реализации метода был выбран инструмент верификации CPAchecker, основанный на подходе CEGAR с использованием булевой предикатной абстракции и интерполяции Крейга для получения новых предикатов при уточнении абстракции. Для решения задач проверки выполнимости формулы пути и интерполяции Крейга в CPAchecker используются интерполирующие решатели MATHSAT 5, SMTINTERPOL, Z3, PRINCESS. Все эти решатели поддерживают бескванторные теории вещественной или целочисленной линейной арифметики и равенства с неинтерпретируемыми функциями, но не поддерживают (либо не полностью поддерживают, как MATHSAT 5) теорию массивов. Поэтому в данной работе разрабатывался и исследовался способ построения формулы пути с использованием только неинтерпретируемых функций.

В разработанном подходе состояние памяти программы представляется в виде неинтерпретируемой функции *f*, отображающей некоторые условные адреса переменных в памяти в их значения. Для неинтерпретируемых функций выполнена аксиома конгруэнтного замыкания отношения эквивалентности $\forall a. \forall b. ((a = b) \rightarrow f(a) = f(b))$. Эта аксиома моделирует равенство значений, полученных после разыменования равных указателей при одном и том же состоянии памяти программы. То есть если есть два равных указателя *p1* и *p2*, то значения **p1* и **p2* также равны.

При записи значения по какому-либо адресу (**e = expr*) происходит смена версии неинтерпретируемой функции, представляющей состояние памяти. При этом в получаемую логическую формулу пути необходимо явно добавлять равенства между значениями соседних версий неинтерпретируемых функций для не участвовавших в присваивании адресов. Поскольку адрес в таком присваивании часто может вычисляться динамически и быть в общем случае неизвестен в точке присваивания, данные равенства в логической формуле будут представлены в виде дизъюнкций следующего вида:

$$e = a \vee f_i(a) = f_{i-1}(a),$$

где e — адресное выражение, a — адрес, для которого выписывается равенство, f_{i-1} и f_i — соответственно старая и новая версия неинтерпретируемой функции, обновляемой в результате присваивания.

Для представления адресов в памяти предлагается использовать суммы вида $b + n$, где b — переменная (неинтерпретируемая константа), соответствующая адресу некоторой сущности (переменной, структуры, объединения или массива) верхнего уровня, называемая базовым адресом, n — смещение рассматриваемой переменной относительно сущности верхнего уровня. При таком представлении для адресов сущностей верхнего уровня необходимо выполнение условий положительности и непересечения внутренних адресов. Эти условия предлагается записывать в виде двух аксиом модели памяти:

$$b>0 \qquad (A1)$$

$$B(b+n) = k, \qquad (A2)$$

где b — переменная, представляющая базовый адрес сущности, k — целое число, уникальное для каждой такой переменной, n — смещение относительно начала сущности, принимающее значения от 0 до $s-1$ включительно, где s — размер сущности.

Экземпляры (A2) позволяют задать всевозможные попарные неравенства внутренних адресов. Докажем, что из $B(b_1 + n_1) = k_1, B(b_2 + n_2) = k_2$ и $k_1 \neq k_2$ следует, что $b_1 + n_1 \neq b_2 + n_2$. Пусть $b_1 + n_1 = b_2 + n_2$, тогда из (A2) получаем $B(b_1 + n_1) = B(b_2 + n_2)$, $k_1 = k_2$ – противоречие с $k_1 \neq k_2$.

2.1 Полнота предлагаемого метода

Одним из важнейших требований, предъявляемых к подходу уточнения и построения абстракции в инструменте, основанном на использовании метода CEGAR, является требование полноты этого подхода, или иначе говоря, требование надежного уточнения абстракции. Оно означает, что на каждой следующей итерации метода CEGAR, после уточнения абстракции на основе какого-либо ложного контрпримера, этот контрпример обязательно оказывается исключенным из уточненной абстракции. Таким образом достигается постоянное уточнение абстракции с точки зрения уменьшения числа порождаемых ею ложных контрпримеров. Для выполнения данного требования в той реализации CEGAR, которая используется инструментом CPAchecker, достаточно выполнения требования индуктивности получаемых интерполянтов и независимого построения различных частей формулы пути.

Независимость построения различных частей формулы пути означает, что каждой из этих частей будет соответствовать одна и та же логическая формула вне зависимости от контекста ее построения (при проверке контрпримера, интерполяции или пересчете абстракции). Это требование могло бы быть нарушено, например, при использовании эвристик, учитывающих наличие либо отсутствие в контрпримере каких-либо неиспользуемых переменных. В предлагаемом подходе такие эвристики не используются. В случае же выполнения обоих требований полноту подхода можно показать, воспользовавшись определением интерполянта. Предлагаемый подход удовлетворяет этим требованиям, так как предполагает последовательное получение индуктивных интерполянтов и для одинаковых блоков операторов строит формулы пути, совпадающие с точностью до имен индексированных символов.

3. Определения

3.1 Программы и поток управления

Мы ограничимся рассмотрением простого императивного языка программирования (аналогично [25, 26]), в котором все переменные имеют типы *int* и *int**, а все операции — это либо присваивания, либо предположения *assume*, представленные в Таблице 2. Мы рассмотрим программы без вызовов функций, хотя описанный подход может быть расширен на программы с несколькими функциями[2].

Программа представляется *автоматом потока управления* (АПУ) (от англ. control-flow automaton, CFA). АПУ *A=(L,G)* состоит из множества точек программы L, моделирующих счетчик команд, и множества дуг потока управления $G \subseteq L \times Ops \times L$, которые моделируют действия, выполняемые при переходе из одной точки программы в другую. Множество переменных, встречающихся в операциях *Ops*, обозначим как *X*. Программа $P = (A, l_0, l_E)$ состоит из АПУ *A=(L,G)* (моделирующего поток управления программы), начальной точки программы $l_0 \in L$ (моделирует точку входа), и целевой точки программы l_E (моделирует ошибочное состояние).

Конкретное состояние данных программы – это состояние памяти программы, как динамической выделяемой, так памяти под переменные *X*, выделенные на стеке и в статической памяти. Далее мы не будем разделять разные виды пямяти, а будем считать, что для каждой переменной из множества X выделена память, адрес которой обозначается именем переменной. Состояние памяти задается функцией $f : Z \to Z$, отображающей адрес ячейки памяти в значение, содержащееся в ней. Так как переменные $x \in X$ обозначают адреса ячеек памяти, то значения для переменной *x* будут представляться как *f(x)*. Обозначим множество конкретных состояний программы как $\mathcal{E}$.

Множества $r \subseteq \mathcal{E}$ назовем регионами, которые будем представлять с помощью формул алгебры логики φ. Формулы будут содержать переменные из множества *X*, а также неинтерпретируемые функции из множества *F*, заданные над целыми числами ($Z \to Z$), которые служат для моделирования состояния памяти. Формула φ представляет множество $[[\varphi]]$ конкретных состояний данных *c*, для которых выполнено φ (т. е. $[[\varphi]] = \{c \in \mathcal{E} \mid c \vDash \varphi\}$).

Конкретное состояние программы это пара *(l,c)*, где $l \in L$ точка программы, а *c* — это конкретное состояние данных. Пара *(l, φ)* представляет множество $\{(l, c) \mid c \vDash \varphi\}$ конкретных состояний. Конкретная семантика операции $op \in Ops$ определяется оператором сильнейшего постусловия $SP_{op}(\cdot)$: для

[2] Реализация в инструменте CPAchecker работает с программами на языке C и поддерживает вызовы функций

формулы φ оператор $SP_{op}(\varphi)$ представляет наименьшее по включению множество состояний, содержащее все состояния, получаемые хотя бы из одного состояния региона, представленного φ, после выполнения оператора op.

Путь σ — это последовательность $\langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ пар из операции и точки программы. Путь σ называется *путем программы*, если он начинается в l_0 (см. определение P) и для каждого i, такого что $0 < i \leq n$, существует дуга АПУ $g = (l_{i-1}, op_i, l_i)$. Таким образом, σ представляет синтаксический путь в АПУ.

Конкретная семантика для пути программы σ = $\langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ определяется как последовательное применение оператора сильнейшего постусловия для каждой операции: $SP_\sigma(\varphi) = SP_{op_n}(\ldots SP_{op_1}(\varphi)\ldots)$. Формула $SP_\sigma(\varphi)$ называется *формулой пути*.

Множество конкретных состояний, являющихся результатом выполнения пути программы σ представляется парой $(l_n, SP_\sigma(true))$. Путь программы называется *достижимым*, если формула $SP_\sigma(true)$ выполнима. Конкретное состояние программы (l_n, c_n) называется *достижимым*, если существует достижимый путь σ, заканчивающийся в точке l_n, и такой, что $c_n \vDash SP_\sigma(true)$. Точка программы l достижима, если существует конкретное состояние c, такое что *(l, c)* достижимо. Программа корректна (safe), если l_E недостижимо.

3.2 Булевы предикатные абстракции

Пусть F — множество неитерпретируемых функций. Пусть $\wp$ — множество предикатов из теории без кванторов над переменными программы X и неинтерпретируемыми функциями F. Формула φ — это булева комбинация предикатов из $\wp$. *Точность для формул* — это конечное подмножество $\pi \in \wp$. *Точность для программы* — это функция $\Pi : L \to 2^{\wp}$, которая задает точность для формул в каждой точке программы.

Булева предикатная абстракция $(\varphi)^\pi$ для формулы φ — это сильнейшая булева комбинация предикатов из точности π, которая следует из φ. Данная предикатная абстракция формулы φ, которая представляет регион конкретных состояний программы, используется как *абстрактное состояние данных* (абстрактное представление региона) в верификации программ. Для формулы φ и точности π булева предикатная абстракция $(\varphi)^\pi$ может быть вычислена с помощью запросов к SMT решателю с поддержкой ALL-SAT следующим образом. Каждому предикату $p_i \in \pi$ сопоставим булеву переменную v_i. Затем сделаем запрос к решателю для выдачи всех векторов решений $v_1, \ldots, v_{|\pi|}$ формулы $\varphi \wedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$. Для каждого вектора решений мы строим конъюнкцию всех предикатов из π, которые входят в вектор решения

как истина. Дизъюнкция всех таких конъюнкций будет булевой предикатной абстракцией для формулы φ.

Абстрактный оператор сильнейшего постусловия с точностью π и операцией *op*, который преобразует абстрактное состояние φ в следующее состояние φ', может быть определен с помощью применения оператора сильнейшего постусловия и последующего вычисления предикатной абстракции, т. е. $\varphi' = \left(SP_{op}(\varphi)\right)^{\pi}$. Более детально предикатные абстракции описаны в работах [27, 28, 7].

3.3 Кодирование с настраиваемым размером блока

В кодировании с настраиваемым размером блока (Adjustable-Block Encoding, ABE) предикатные абстракции не вычисляются при каждом переходе по дуге из АПУ, а напротив, вычисляются только в некоторых абстрактных состояниях, которые будем называть *состояниями абстракции* (другие абстрактные состояния будем называть состояниями *без абстракции*). На пути между двумя состояниями вычисления абстракции сильнейшее постусловие пути хранится во втором компоненте состояния, который мы назовем *дизъюнктивной формулой пути*. Таким образом, абстрактное состояние ABE содержит два компонента-формулы $\langle \psi, \varphi \rangle$, где формула абстракции ψ – это результат вычисления абстракции, а дизъюнктивная формула φ представляет сильнейшее постусловие с момента вычисления последнего состояния абстракции. Для заданной дуги АПУ $g = (l, op, l')$ и абстрактного состояния $\langle \psi, \varphi \rangle$, следующее состояние либо только расширяет формулу пути φ, либо вычисляет новую формулу абстракции ψ и сбрасывает φ. Точки вычисления абстракции (и, таким образом, размер блока) определяется так называемым оператором настройки блока *blk*. Если оператор $blk(e, g)$ возвращает *false* (нет вычисления абстракции, т. е. абстрактное состояние e без абстракции), то следующее состояние $\langle \psi', \varphi' \rangle$ содержит $\psi' = \psi$ (неизмененное) и $\varphi' = SP_{op}(\varphi)$. Если оператор $blk(e, g)$ возвращает *true* (e — состояние абстракции), то следующее состояние $\langle \psi', \varphi' \rangle$ содержит результат вычисления абстракции по формуле $\psi \wedge \varphi$, $\psi' = \left(SP_{op}(\varphi \wedge \psi)\right)^{\pi}$ и $\varphi' = true$ как новую дизъюнктивную формулу. Если $\varphi \wedge \psi$ невыполнимо для состояния e, то e недостижимо.

4. Модель памяти на основе неинтерпретируемых функций

Для моделирования памяти будем использовать неинтерпретируемые функции f (f_m) и B. Имя неинтерпретируемой функции f_m определим как конкатенацию имени f и индекса m: *f#m*. Над формулами ψ для

неинтерпретируемых функций f и f_m определим операцию подмены всех таких функций, содержащихся в формуле ψ на новое имя $f_{m'}$: $\psi[f_{m'}]$ — заменяет все вхождения f и f_m на $f_{m'}$.

Для задания модели памяти нам потребуются вспомогательные компоненты, которые будут храниться в абстрактном состоянии и изменяться при переходе к следующему состоянию. Во-первых, $Alloc$ — множество пар $(A, n) \in \mathcal{A} \times int$, где константа $A \in \mathcal{A}$ — представляет базовый адрес выделенной памяти, а число n — смещение относительно базового адреса. Во-вторых, m — индекс функции памяти f. В-третьих, k — последний индекс базовых адресов.

В модели памяти оператор сильнейшего постусловия задается как $SP_{op}(\varphi) = \varphi \wedge \Gamma(op)$, где $\Gamma(op)$ определяется Таблицей 2.

Будем использовать вспомогательную функцию:

- $mem\_update(p, m', m, addrs) = \bigwedge_{(a,i)\in addrs}\Big((p = a + i) \vee \big(f_{m'}(a + i) = f_m(a + i)\big)\Big)$,

где e — это выражение без побочных эффектов и без разыменований указателей.

В выражениях **(s1 + i)*, i имеет тип int.

Размер целого и указателей принимается равным 1 байту.

Иначе нужно в выражении **(s + i)* в i указывать размер как количество элементов, помноженное на соответствующий размер элемента.

4.1 Расширение подхода для структур

Подход к построению ограничений Г может быть легко расширен на случай использования в программе структурных типов. Основная идея состоит в том, что структура рассматривается как участок памяти размера, равного сумме размеров полей. Обращения к полям транслируются как сумма указателя на начало структуры и смещения поля относительно начала. Обозначим как $\omega(A, f)$ смещение поля с именем f в структуре A.

1. При выделении памяти *alloc(size)* для структуры A, размер *size* определяется по размеру структуры — сумма размеров полей, и, возможно, с учетом выравнивания.

Таблица 2. Правила построения ограничений Γ

Операция (op)	*Индекс функции памяти m'*	*Множество адресных переменных Alloc'*	*Индекс базового адреса k'*	*Ограничения Γ*
Выделение переменной на стеке int s; или int *s;	Не меняется	*A'* — новое имя переменной. $Alloc' = Alloc \cup \{(A', 0)\}$	*k'* — новый индекс	*s = A'* ∧ *A' > 0* ∧ *B(A')=k'*
Выделение памяти размера size в куче s = alloc(size)	*m'* — новый индекс	*A'* — новое имя переменной. *Alloc' = Alloc ∪ {(A',0)...(A',size-1)}*	*k'* — новый индекс	$f_{m'}(s) = A'$ ∧ *A' > 0* ∧ *B(A'+i)=k'*, где *i* от *0* до *size-1* ∧ *mem_update(s, m', m, Alloc)*
s = e	*m'* — новый индекс	Не меняется	Не меняется	$f_{m'}(s) = \Gamma(e)$ ∧ *mem_update(s, m', m, Alloc)*, где Γ(e) для выражения e вычисляется по следующим правилам: $\Gamma(const)$: $const$ $\Gamma(s)$: $f_m(s)$ $\Gamma(s1\ op\ s2)$, $op \in \{'+', '-', '*', '/'\}$: $f_m(s1)\ op\ f_m\ (s2)$
*(s1 + i) = s2	*m'* — новый индекс	Не меняется	Не меняется	$f_{m'}(f_m(s1) + f_m(i)) = f_m(s2)$ $\wedge\ mem\_update(f_m(s1) + f_m(i), m', m, Alloc)$
s1 = *(s2 + i)	*m'* — новый индекс	Не меняется	Не меняется	$f_{m'}(s1) = f_m(f_m(s2) + f_m(i))$ $\wedge\ mem\_update(s1, m', m, Alloc)$
assume p	Не меняется	Не меняется	Не меняется	Γ(p) для предиката p вычисляется по следующим правилам: $\Gamma(const)$: $const$ $\Gamma(s)$: $f_m(s)$ $\Gamma(p1 == p2)$: $\Gamma(p1) = \Gamma(p2)$ $\Gamma(p1 < p2)$: $\Gamma(p1) < \Gamma(p2)$ $\Gamma(p1 <= p2)$: $\Gamma(p1) \leq \Gamma(p2)$ $\Gamma(p1 \,||\, p2)$: $\Gamma(p1) \vee \Gamma(p2)$ $\Gamma(p1 \,\&\&\, p2)$: $\Gamma(p1) \wedge \Gamma(p2)$ $\Gamma(!\,p)$: $\neg\Gamma(p)$

2. Выражение доступа к полю $s->f$ рассматривается как $*(s + \omega(A, f))$.
3. В присваивании структур по значению *s1 = s2* необходимо выписывать ограничения для присваивания всех полей структуры.

4.2 Пример построения формулы пути

В примере программы используется макрос *container_of(p, type, field_name)*, который раскрывается как $(type)(p + (0 - \omega(type, field\_name)))$. То есть для указателя на вложенное поле структуры *field_name*, мы получаем указатель на структуру, содержащую это поле. Макрос *container_of* часто используется в коде ядра ОС Linux и составляет большую сложность для инструментов, не поддерживающих адресную арифметику. Из-за этого возникают как ложные срабатывания, так и упущенные ошибки.

Пусть задана следующая программа:

```
struct B { int a; int b; };

struct B *p; struct B *q; int *x;

p =alloc(sizeof(B));

p->b = 1;

x = &(p->a);

q = container_of(x, struct B, a);

assume(p->b != q->b);
```

Программа содержит единственный путь, являющийся недостижимым, так как *p->b == q->b* и условие в последнем *assume* не выполнено. В данном примере при моделировании памяти требуется учитывать семантику арифметики указателей, иначе возможно ложное срабатывание, из-за того что путь может быть признан достижимым.

Для данного пути предварительно мы заменим операции со структурами на операции с указателями и построим для него формулу пути.

Таблица 3. Пример построения формулы пути

Путь	Построение формулы пути			
	m'	Alloc'	k'	Γ
struct B *p;	*0*	*{(A0,0)}*	*1*	*p = A0* *∧ A0 > 0 ∧ B(A0) = 1*
struct B *q;	*0*	*{(A0,0), (A1,0)}*	*2*	*q = A1* *∧ A1 > 0 ∧ B(A1) = 2*

int *x;	*0*	*{(A0,0), (A1,0), (A2,0)}*	*3*	$x = A2$ $\wedge\ A2 > 0 \wedge B(A2) = 3$
p =alloc(sizeof(struct B));	*1*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_1(p) = A3$ $\wedge\ A3>0$ $\wedge\ B(A3+i) = 4, 0 \le i < sizeof(struct\ B)$ $\wedge\ ((p = A_i + o_i) \vee (f_1(A_i + o_i) = f_0(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
*(p + $\omega(B,b)$) = 1;	*2*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_2(f_1(p) + \omega(B,b)) = 1$ $\wedge$ $((f_1(p) + \omega(B,b) = A_i + o_i) \vee (f_2(A_i + o_i) = f_1(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
x = p + $\omega(B,a)$;	*3*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_3(x) = f_2(p) + \omega(B,a)$ $\wedge\ ((x = A_i + o_i) \vee (f_3(A_i + o_i) = f_2(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
q = x + (0 - $\omega(B,a)$);	*4*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_4(q) = f_3(x) - \omega(B,a)$ $\wedge\ ((q = A_i + o_i) \vee (f_4(A_i + o_i) = f_3(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
assume *(p + $\omega(B,b)$) != *(q + $\omega(B,b)$)	*4*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$\neg(f_4(f_4(p) + \omega(B,b)) = f_4(f_4(q) + \omega(B,b)))$

Таким образом, мы получим формулу пути $SP_\sigma(true)$, являющуюся конъюнкцией формул в столбце Г. Эта формула является невыполнимой, что подтверждает недостижимость данного пути, что и требовалось показать в приведенном примере.

4.3 Конфигурируемый анализ (CPA) с моделью памяти на основе неинтерпретируемых функций

Формализуем анализ с моделью памяти на основе неинтерпретируемых функций в виде *конфигурируемого анализа* (Configurable program analysis, CPA) [29]. Это позволяет использовать гибкость операторов CPA для описания анализа без изменения основного алгоритма (см. Алгоритм CPA на Рис. 1).

Конфигурируемый анализ $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$ состоит из абстрактного домена D, отношения перехода $\rightsquigarrow$, оператора *merge*, оператора *stop*, определяемых следующим образом.

Пусть задана программа $P = (A, l_0, l_E)$, где X — обозначает множество переменных используемых в программе P, F — множество неинтерпретируемых функций, используемых для моделирования памяти, $\wp$ — множество предикатов без кванторов над переменными X и функциями F, и $\Pi: L \to 2^{\wp}$ — точность предикатной абстракции.

1. **Абстрактный домен** $D = (C, R, [[\cdot]])$ — это тройка, состоящая из множества конкретных состояний C, полурешетки $R = (E, \top, \sqsubseteq, \sqcup)$ и функции конкретизации $[[\cdot]] : E \to C$.

Элементы решетки также называются абстрактными состояниями и являются семерками $(l, \psi, l^{\psi}, \varphi, Alloc, k, m)$, где первые четыре компонента являются стандартными компонентами анализа АВЕ, $l, l^{\psi} \in (L \cup \{l_{\top}\})$; $\psi, \varphi \in \wp$. Компонент l моделирует счетчик команд, формула абстракции ψ — булева комбинация предикатов, заданных в Π, l^{ψ} — точка в программе, в которой была вычислена абстракция ψ, а φ — это дизъюнктивная формула, представляющая некоторые или все пути из точки l^{ψ} в l. Заметим, что в состоянии абстракции всегда $l = l^{\psi}$ и $\varphi = true$.

Для уточненной модели памяти мы ввели три новых компонента. Во-первых, $Alloc$ — множество пар $(A, n) \in \mathcal{A} \times int$, где $A \in \mathcal{A}$ — представляет базовый адрес выделенной памяти, а число n — смещение относительно базового адреса. Во-вторых, m — индекс функции памяти f. В-третьих, k — последний индекс базовых адресов.

Верхний элемент решетки — это абстрактное состояние $\top = (l_{\top}, true, l_{\top}, true, \{\}, 0, 0)$. Частичный порядок $\sqsubseteq \subseteq E\ x\ E$ определяется так, что для любых двух состояний $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1, Alloc_1, k_1, m_1)$ и $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2, Alloc_2, k_2, m_2)$ из E выполнено:

$$e_1 \sqsubseteq e_2 \equiv (e_2 = \top) \vee ((l_1 = l_2) \wedge (\varphi_1 = \varphi_2 = true) \wedge (\psi_1[f] \Rightarrow \psi_2[f]))$$

Заметим, что мы подменяем все вхождения версий памяти в формулах абстракции $\psi_1[f]$ и $\psi_2[f]$ на одинаковую версию f.

Оператор соединения $\sqcup: E\ x\ E \to E$ выдает наименьшую верхнюю грань двух операндов в соответствии с частичным порядком $\sqsubseteq$.

Алгоритм CPA($\mathbb{D}, e_0$) (взят из работы [29])
Вход: CPA $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$, начальное состояние $e_0 \in E$, где E обозначает множество элементов решетки D
Выход: множество достижимых абстрактных состояний
Переменные: множество $reached$ достигнутых элементов из E, множество $waitlist$ элементов из E
1: $waitlist$:= $\{e_0\}$
2: $reached$:= $\{e_0\}$
3: **пока** $waitlist \neq 0$ **делать**
4: выбрать элемент e из $waitlist$

```
5:  waitlist := waitlist \ {e}
6:  для каждого e' такого что e ⇝ e' делать
7:    для каждого e'' ∈ reached делать
8:      //слияние с существующими абстрактными состояниями
9:      e_new := merge(e', e'')
10:     если e_new ≠ e'' то
11:       waitlist := (waitlist ∪ {e_new}) \ {e''}
12:       reached := (reached ∪ {e_new}) \ {e''}
13:   если ¬stop(e', reached) то
14:     waitlist := waitlist ∪ {e'}
15:     reached := reached ∪ {e'}
16: вернуть reached
```

Рис. 1. Алгоритм конфигурируемого анализа (CPA)

2. **Отношение перехода** $\rightsquigarrow \subseteq E \times G \times E$ содержит все дуги (e, g, e'), где $e = (l, \psi, l^{\psi}, \varphi, Alloc, k, m)$, $e' = (l', \psi', l^{\psi'}, \varphi', Alloc', k', m')$ и $g = (l, op, l')$ для которых выполнено:

$$\begin{cases} (\varphi' = \text{true}) \wedge \left(\psi' = \left(\mathrm{SP}_{\mathrm{op}}(\varphi \wedge \psi)\right)^{\Pi(l')}[\mathrm{f}_{\mathrm{m}'}]\right) \wedge (l^{\psi'} = l'), \text{если } \mathrm{blk}(e, g) \vee (l' = \mathrm{l}_{\mathrm{E}}) \\ (\varphi' = \mathrm{SP}_{\mathrm{op}}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi'} = l^{\psi}) \text{ иначе} \end{cases}$$

В представленной модели памяти оператор сильнейшего постусловия задается как $SP_{op}(\varphi) = \varphi \wedge \Gamma(op)$, где $\Gamma(op)$ определяется таблицей 2. При этом значения $Alloc', k', m'$ в следующем состоянии определяются также по таблице 2.

Таким образом, мы имеем отношения перехода, которое работает в двух режимах, определяемых оператором $blk: E \times G \to B$, который отображает абстрактное состояние e и дугу g АПУ в *true* или *false*. Оператор *blk* задается как параметр анализу. В первом режиме строится абстракция, а во втором вычисляется только сильнейшее постусловие.

3. **Оператор слияния** $merge: E \times E \to E$ для двух абстрактных состояний $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1, Alloc_1, k_1, m_1)$ и $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2, Alloc_2, k_2, m_2)$ определяется следующим образом: $merge(e_1, e_2) =$

$$\begin{cases} \text{если } (\mathrm{l}_1 = \mathrm{l}_2) \wedge (\psi_1[\mathrm{f}] = \psi_2[\mathrm{f}]) \wedge (l^{\psi_1} = l^{\psi_2}), \text{то } (\mathrm{l}_2, \psi_2, l^{\psi_2}, (\varphi_1 \wedge \mathrm{mem\_unchanged}(\mathrm{m}', \mathrm{m}_1, \mathrm{Alloc}_1)) \\ \quad \vee (\varphi_2 \wedge \mathrm{mem\_unchanged}(\mathrm{m}', \mathrm{m}_2, \mathrm{Alloc}_2)), \mathrm{Alloc}_1 \cup \mathrm{Alloc}_2, \max(\mathrm{k}_1, \mathrm{k}_2), \mathrm{m}') \\ \quad e_2, \text{иначе} \end{cases}$$

где m' — новый индекс функции памяти, а функция *mem_unchanged* определяется как:

- $mem\_unchanged(m', m, addrs) = \wedge_{(a,i) \in addrs}(f_{m'}(a + i) = f_m(a + i))$

Таким образом, множества адресных переменных объединяются, а при построении новой дизъюнктивной формулы вводится новый индекс функции памяти, к которой приравниваются функции в каждом из состояний.

4. **Оператор останова** $stop\colon E \times 2^E \to B$ проверяет, покрывается ли состояние e другим состоянием из пройденных состояний R (множество *reached*):

$$\forall e \in E, R \subseteq E\colon stop(e,R) \;=\; \exists e' \in R : (e \sqsubseteq e')$$

5. Оптимизации

Эффективность модели памяти сильно зависит от количества дизъюнкций, выписываемых в *mem_update* из Таблицы 2. В предлагаемом подходе используются следующие оптимизации для сокращения числа этих дизъюнкций:

1. Разделение области памяти по типам, так что каждая неинтерпретируемая функция задает отображение адресов переменных в их значения для одного соответствующего ей примитивного типа данных. Например, вводятся функции *f_long_int, f_char*, f_struct_B**. Примитивными, то есть не составными типами данных в данном подходе, считаются символьный и целочисленный типы, а также любой тип указателя. Таким образом, неявно предполагается, что значение, записанное по какому-либо адресу в качестве значения какого-то типа данных не может быть впоследствии считано по этому же адресу как значение другого типа. Это предположение является одним из ограничений подхода.
2. Выделение среди множества всех переменных программы подмножества «чистых» переменных, к которым возможен доступ лишь по именам (то есть переменных, не имеющих алиасов). Для таких переменных нет необходимости в использовании неинтерпретируемых функций для представления значений. Поэтому для них используется сходное с используемым инструментами BLAST и CPAchecker SSA-представление (для них не выписывается разыменование и имя переменной представляет значение переменной, а не ее адрес). Определение «чистых» переменных возможно проводить не для всей программы, а на заданном пути, до тех пор пока не встретиться взятие адреса для этой переменной на этом пути или на другом пути при выполнении оператора *merge*.
3. Использование эвристики для полей структур. В предлагаемом подходе предполагается, что указатель на поле структуры, получаемый сложением адреса структуры с соответствующим смещением поля, может принимать только значения адресов того же самого поля в других структурах того же самого типа, что и структура, которой это поле принадлежит. Такой указатель не может,

в частности, быть равным адресу элемента массива или отдельной переменной, не являющейся полем структуры. Например, обновление поля *skb1->next* не может повлиять ни на какой *skb2->prev*, даже если *next* и *prev* одного типа. В этом случае в качестве оптимизации мы опускаем посыл импликации в *mem_update*, если смещения заранее не равны. Такое предположение также является одним из ограничений метода.

4. Инициализация константами. Оптимизация заключается в том, что мы выписываем одно обновление mem_update для нескольких присваиваний. Например, при выделении памяти, заполненной нулями под структуру *kzalloc(sizeof(*info), GFP_KERNEL)*, *mem_update* выписывается только один раз после инициализации всех полей нулями.
5. Сокращение множества *Alloc*, за счет сохранения смещений только для тех полей, которые были использованы в пути. Соответственно для неиспользованных полей не будут выписываться дизъюнкции в *mem_update*.

6. Результаты

Предложенный метод был реализован в инструменте CPAchecker версии 1.4 (для экспериментов была взята ревизия *18237*). За включение разработанной модели памяти на основе неинтерпретируемых функций отвечает опция *cpa.predicate.handlePointerAliasing*. Эксперименты проводились на наборах международных соревнований по верификации программ SV-COMP'2016 (http://sv-comp.sosy-lab.org/2016).

В первую очередь была рассмотрена категория по работе со структурами данных в куче *Heap Data Structures*. Задачи в этой категории требуют поддержки анализа указателей.

Запуски проводились в двух конфигурациях предикатного анализа *predicateAnalysis* с поддержкой модели памяти на основе неинтерпретируемых функций (с НФ) и без нее (без НФ). Как можно видеть в Таблице 4, на представленном наборе анализ с использованием предложенного метода не дает некорректных результатов на представленном наборе по сравнению с не использующей его версией, которая давала 24 некорректных результата.

Отметим, что вердикт *unknown* возникает, когда верификатор не может найти ошибку или доказать ее отсутствие, в силу превышения лимитов по времени, памяти, обнаружению неподдерживаемых конструкций (например, рекурсии) или в силу ограничений самого CEGAR.

Количество вердиктов *unknown* составило 14 шт. для НФ, большая часть из них из-за превышения лимита по времени 10 шт. В результате общее время работы анализа с НФ составило 9514 секунд, что почти в десять раз больше чем без НФ (см. Табл. 5). Однако если рассматривать только время на

корректные результаты (не включающее время на вердикты *unknown*), то время работы оказывается сравнимо.

Таблица 4. Результаты запуска на наборе HeapDataStructures в конфигурации предикатного анализа, с лимитом времени 15 минут и 15 Gb памяти.

Модели памяти	без НФ	с НФ
Общее количество	81	81
Корректные результаты	53	67
Доказано отсутствие ошибки	34	44
Ошибка найдена	19	23
Некорректные результаты	24	0
Упущенная ошибка	5	0
Ложное предупреждение	19	0

Таблица 5. Время работы CPU (в секундах) запуска на наборе HeapDataStructures в конфигурации предикатного анализа, с лимитом времени 15 минут и 15 Gb памяти.

Модели памяти	без НФ	с НФ
Общее время	680	9514
Время для корректных результатов	476	487

В качестве второго набора был выбран *DeviceDriversLinux64*, состоящий из драйверов устройств ядра операционной системы Linux, большая часть которого подготовлена в рамках проекта LDV [30, 31]. Для этого набора была использована конфигурация *ldv*, использующаяся в проекте LDV. Видно, что время работы для этого набора сравнимо как на всех тестах, так и на тестах, для которых получен корректный результат (см. Табл. 7). Причем корректных результатов без НФ получено на 11 шт. больше (см. Табл. 6).

Таблица 8 показывает изменения вердиктов при переходе от запуска без НФ к запуску с НФ. Видно, что модель памяти с НФ теряет 31 корректный результат, из них 29 из-за вердикта *unknown*, одно ложное срабатывание получено из-за того, что для точной работы модели памяти с НФ в тесте не хватает явного выделения памяти, кроме того, еще в одном тесте анализ без НФ находит ложную трассу ошибки, так как считает возможным равенство нулю адреса переменной на стеке. С другой стороны, получается 20 новых корректных результатов, а 5 некорректных результатов становятся *unkown*.

Таблица 6. Результаты запуска на наборе DeviceDriversLinux64 в конфигурации ldv, с лимитом по времени 15 минут и 15 Gb по памяти.

Модели памяти	без НФ	с НФ
Общее количество	2121	2121
Корректные результаты	1654	1643
Доказано отсутствие ошибки	1450	1450
Ошибка найдена	204	193
Некорректные результаты	17	6
Упущенная ошибка	5	3
Ложное предупреждение	12	3

Таблица 7. Время работы CPU (в часах) запуска на наборе DeviceDriversLinux64 в конфигурации ldv, с лимитом по времени 15 минут и 15 Gb по памяти.

Модели памяти	без НФ	с НФ
Общее время	140 часов	144 часа
Время для корректных результатов	22,2 часов	22,7 часов

Таблица 8. Результаты изменения вердиктов на наборе DeviceDriversLinux64 в конфигурации ldv.

Переходы из модели памяти без НФ в модель памяти с НФ	Количество
Корректный результат → Некорректный результат	2
Корректный результат → unknown	29
Некорректный результат → Корректный результат	8
Некорректный результат → unknown	5
Unknown → Корректный результат	12
Unknown → Некорректный результат	0

7. Заключение

Метод моделирования памяти на основе неинтерпретируемых функций позволяет анализировать программы, содержащие выражения с указателями, в том числе указателями на структуры, массивы, и выражения, содержащие адресную арифметику. Ограничениями метода является конечность размера массивов и конечная глубина рекурсии для динамических структур данных. Метод является масштабируемым, так как показывает приемлемые результаты по скорости на практически значимом наборе из драйверов устройств ОС Linux. Вместе с тем, ряд корректных результатов не может быть получен, по причине замедления работы анализа. Это требует дальнейшего развития метода.

Среди возможных направлений отметим, возможность более точного разбиения на регионы памяти, так что для этих регионов выделяются независимые неинтерпретируемые функции. Например, отдельные функции могут быть использованы для каждого поля структуры, для которого не берется адрес. Регионы также могут быть выделены на основе предварительного анализа кода программы. Кроме того, в качестве направления исследований может быть рассмотрено объединение инструкций программы, например, объединение последовательности присваиваний, меняющих независимые участки памяти, и их рассмотрение как одного обновления памяти.

Литература

[1]. Edmund Clarke, Daniel Kroening, Flavio Lerda. A Tool for Checking ANSI-C Programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, volume 2988, pp. 168-176, 2004.

[2]. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pp. 193–207, 1999.

[3]. Daniel Kroening, Edmund Clarke, Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In Proceedings of DAC 2003, pp. 368–371, ACM Press, 2003.

[4]. F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, P. Ashar. F-Soft: Software Verification Platform. In CAV, LNCS, volume 3576, pp. 301-306, 2005.

[5]. H. Post, C. Sinz, F. Merz, T. Gorges, T. Kropf. Linking Functional Requirements and Software Verification. In 17th IEEE International Requirements Engineering Conference, pp.295-302, 2009.

[6]. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. Counterexample-Guided Abstraction Refinement. In CAV, LNCS, volume 1855, pp. 154-169, 2000.

[7]. М.У. Мандрыкин, В.С. Мутилин, А.В. Хорошилов. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды Института системного программирования РАН, том 24, стр. 219-292, 2013.

[8]. S. Graf, H. Saidi. Construction of abstract state graphs with PVS. In CAV, LNCS, volume 1254, pp. 72-83, 1997.

[9]. Ranjit Jhala, Rupak Majumdar. Software model checking. ACM Computing Surveys, volume 41, issue 4, article 21, pp.1-54, 2009.
[10]. William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. The Journal of Symbolic Logic, vol. 22, no. 3, pp. 269-285, 1957.
[11]. Roger Lyndon. An interpolation theorem in the predicate calculus. Pacific Journal of Mathematics, vol. 9, no. 1, pp. 129-142, 1959.
[12]. Dirk Beyer, Damien Zufferey, Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In CAV, LNCS, volume 5123, pp. 304-308, 2008.
[13]. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani. The MathSAT 4 SMT Solver. In CAV, LNCS, volume 5123, pp. 299-303, 2008.
[14]. Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. Journal on Satisfiability, Boolean Modeling and Computation, vol. 8 pp. 1-27, 2012.
[15]. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer, volume 9, issue 5, pp. 505-525, 2007.
[16]. Швед П. Е., Мутилин В. С., Мандрыкин М. У. Опыт развития инструмента статической верификации BLAST. Программирование, том 3, стр. 24–35, 2012.
[17]. P. E. Shved, V. S. Mutilin, M. U. Mandrykin. Experience of improving the BLAST static verification tool. Programming and Computer Software, volume 38, issue 3, pp. 134-142, 2012.
[18]. Andersen L. O. Program Analysis and Specialization for the C Programming Language. Københavns Universitet, Datalogisk Institut, DIKU, 1994.
[19]. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. Points-to analysis using BDDs. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM, volume 38, issue 5, pp. 103-114. 2003.
[20]. Pavel Shved, Mikhail Mandrykin, Vadim Mutilin. Predicate Analysis with BLAST 2.7. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 7214, pp. 525–527, 2012.
[21]. Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 8413, pp. 373-388, 2014.
[22]. K. Dudka, P. Peringer, T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In CAV, LNCS, volume 6806, pp. 372-378, 2011.
[23]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Lazy Shape Analysis. In CAV, LNCS, volume 4144, pp. 532-546, 2006.
[24]. Tal Lev-Ami, Mooly Sagiv. TVLA: A System for Implementing Static Analyses. In Static Analysis, LNCS, volume 1824, pp. 280-301, 2000.
[25]. Dirk Beyer, M. Erkan Keremoglu, Philipp Wendler. Predicate abstraction with adjustable-block encoding. In Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, pp. 189-198, 2010.
[26]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Kenneth L. McMillan. Abstractions from proofs. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, volume 39, issue 1, pp. 232-244, 2004.

[27]. T. Ball, A. Podelski, S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 2031, pp. 268-283, 2006.
[28]. S. K. Lahiri, R. Nieuwenhuis, A. Oliveras. SMT techniques for fast predicate abstraction. In CAV, LNCS, volume 4144, pp. 424-437, 2006.
[29]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In CAV, LNCS, 4590, pp. 504-518, 2007.
[30]. M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, P. E. Shved. Using Linux Device Drivers for Static Verification Tools Benchmarking. Programming and Computer Software, volume 38, issue 5, pp. 245-256, 2012.
[31]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Alexander Petrenko, Ilya Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, n.1, pp. 49-64, 2015.

Modeling Memory with Uninterpreted Functions for Predicate Abstractions

M.U. Mandrykin <mandrykin@ispras.ru>
V.S. Mutilin <mutilin@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. One of the key problems in modern static verification methods is a precise model for semantics of expressions containing pointers. The trustworthiness of the verification verdict highly depends on the analysis of these expressions. In the paper, we describe the verification methods with memory models based on uninterpreted functions, allowing to analyze programs containing expressions with pointers, including pointers to structures, arrays and pointer arithmetic. The approach is limited finite array size and finite recursion depth for dynamic data structures. The method was implemented in CPAchecker tool, based on CEGAR with boolean predicate abstractions and Craig interpolation for inferring new predicates used in abstraction refinement. For solving safisfability of path formulas and Craig interpolation CPAchecker uses interpolation solvers supporting theories of linear integer and real inequalities and equalities with uninterpreted functions. In the method, the memory state is represented as uninterpreted function mapping some variable addresses in memory to its values. After each write to memory for a pointer the version of uninterpreted function is changed. The experiments were performed on the benchmarks of International Competition on Software Verification (SV-COMP'2016) containing industrial size benchmarks of device drivers of Linux operating system. On these benchmarks, the method demonstrates reasonable verification times, reducing the number of missed defects and false alarms. Among the future work directions we consider deviding memory into region thus having a separate uninterpreted for each region.

Keywords: memory model, predicate abstraction, counterexample-guided abstraction refinement

DOI: 10.15514/ISPRAS-2015-27(5)-7

For citation: Mandrykin M.U., Mutilin V.S. Modeling Memory with Uninterpreted Functions for Predicate Abstractions. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 117-142 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-7.

References

[1]. Edmund Clarke, Daniel Kroening, Flavio Lerda. A Tool for Checking ANSI-C Programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, volume 2988, pp. 168-176, 2004.

[2]. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pp. 193–207, 1999.

[3]. Daniel Kroening, Edmund Clarke, Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In Proceedings of DAC 2003, pp. 368–371, ACM Press, 2003.

[4]. F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, P. Ashar. F-Soft: Software Verification Platform. In CAV, LNCS, volume 3576, pp. 301-306, 2005.

[5]. H. Post, C. Sinz, F. Merz, T. Gorges, T. Kropf. Linking Functional Requirements and Software Verification. In 17th IEEE International Requirements Engineering Conference, pp.295-302, 2009.

[6]. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. Counterexample-Guided Abstraction Refinement. In CAV, LNCS, volume 1855, pp. 154-169, 2000.

[7]. Khoroshilov A.V., Mandrykin M. U., MutilinV. S. Vvedenie v metod CEGAR — utochnenie abstrakcii po kontrprimeram [Introduction to CEGAR — Counter-Example Guided Abstraction Refinement]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian).

[8]. S. Graf, H. Saidi. Construction of abstract state graphs with PVS. In CAV, LNCS, volume 1254, pp. 72-83, 1997.

[9]. Ranjit Jhala, Rupak Majumdar. Software model checking. ACM Computing Surveys, volume 41, issue 4, article 21, pp.1-54, 2009.

[10]. William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. The Journal of Symbolic Logic, vol. 22, no. 3, pp. 269-285, 1957.

[11]. Roger Lyndon. An interpolation theorem in the predicate calculus. Pacific Journal of Mathematics, vol. 9, no. 1, pp. 129-142, 1959.

[12]. Dirk Beyer, Damien Zufferey, Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In CAV, LNCS, volume 5123, pp. 304-308, 2008.

[13]. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani. The MathSAT 4 SMT Solver. In CAV, LNCS, volume 5123, pp. 299-303, 2008.

[14]. Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. Journal on Satisfiability, Boolean Modeling and Computation, vol. 8 pp. 1-27, 2012.

[15]. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer, volume 9, issue 5, pp. 505-525, 2007.

[16]. P. E. Shved, V. S. Mutilin, M. U. Mandrykin. Opyt razvitiya instrumenta staticheskoy verifikatsii BLAST. [Experience of improving the BLAST static verification tool]. Programmirovanie [Programming and Computer Software], vol. 38, issue 3, pp. 25-34, 2012 (in Russian).

[17]. P. E. Shved, V. S. Mutilin, M. U. Mandrykin. Experience of improving the BLAST static verification tool. Programming and Computer Software, volume 38, issue 3, pp. 134-142, 2012.

[18]. Andersen L. O. Program Analysis and Specialization for the C Programming Language. København Universitet, Datalogisk Institut, DIKU, 1994.

[19]. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. Points-to analysis using BDDs. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM, volume 38, issue 5, pp. 103-114. 2003.

[20]. Pavel Shved, Mikhail Mandrykin, Vadim Mutilin. Predicate Analysis with BLAST 2.7. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 7214, pp. 525–527, 2012.

[21]. Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 8413, pp. 373-388, 2014.

[22]. K. Dudka, P. Peringer, T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In CAV, LNCS, volume 6806, pp. 372-378, 2011.

[23]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Lazy Shape Analysis. In CAV, LNCS, volume 4144, pp. 532-546, 2006.

[24]. Tal Lev-Ami, Mooly Sagiv. TVLA: A System for Implementing Static Analyses. In Static Analysis, LNCS, volume 1824, pp. 280-301, 2000.

[25]. Dirk Beyer, M. Erkan Keremoglu, Philipp Wendler. Predicate abstraction with adjustable-block encoding. In Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, pp. 189-198, 2010.

[26]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Kenneth L. McMillan. Abstractions from proofs. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, volume 39, issue 1, pp. 232-244, 2004.

[27]. T. Ball, A. Podelski, S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 2031, pp. 268-283, 2006.

[28]. S. K. Lahiri, R. Nieuwenhuis, A. Oliveras. SMT techniques for fast predicate abstraction. In CAV, LNCS, volume 4144, pp. 424-437, 2006.

[29]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In CAV, LNCS, 4590, pp. 504-518, 2007.

[30]. M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, P. E. Shved. Using Linux Device Drivers for Static Verification Tools Benchmarking. Programming and Computer Software, volume 38, issue 5, pp. 245-256, 2012.

[31]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Alexander Petrenko, Ilya Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, n.1, pp. 49-64, 2015.

Использование языка программирования Python для описания ограничений на архитектурные модели[1]

[2]Е.В. Корныхин <kornevgen@cs.msu.ru>
[1,2,3,4]А.В. Хорошилов <khoroshilov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В данной статье предлагается подход к описанию и верификации структурных ограничений на архитектурные модели, в основе которого лежит переиспользование возможностей языка программирования Python, инструментов, библиотек, документации и методических материалов для языка Python. Использование в качестве основы широко известного языка должно уменьшить порог вхождения в предлагаемый подход. Ограничения становятся частью архитектурной модели, в идеале они разрабатываются вместе с моделью. Ограничения записываются на языке программирования Python в виде функций с одним аргументом (он обозначает проверяемый компонент модели) и возвращаемым значением логического типа, снабженных специальным декоратором (исполнимой аннотацией). Чтобы проверить выполнение ограничений для модели, генерируется и выполняется программа на языке Python. В этой программе создается архитектурная модель и затем выполняются нужные функции-ограничения.

Подход был реализован в среде MASIW Framework – это среда моделирования программно-аппаратных систем на языке AADL, выполненная на базе широко известной среды разработки Eclipse. В среду MASIW Framework были интегрированы инструменты разработки на языке программирования Python – это инструмент PyDev, хорошо известный разработчикам на Python в среде Eclipse. Этот инструмент упрощает выполнение программ на Python, содержит в себе мощный редактор программ на Python с раскраской кода и автодополнением. Такие возможности удалось

[1] Исследование проводилось при финансовой поддержке РФФИ в рамках проекта №14-07-00443

задействовать из-за особенностей генерируемых исходных кодов на Python: классы строятся из компонентов модели, поля классов – из подкомпонентов, соединений и т.п., методы – из ограничений, иерархия классов и пакетов – из иерархии компонентов и пакетов исходной архитектурной модели.

Ключевые слова: архитектурное моделирование; верификация моделей; язык программирования Python, язык моделирования AADL.

DOI: 10.15514/ISPRAS-2015-27(5)-8

Для цитирования: Корныхин Е.В., Хорошилов А.В. Использование языка программирования Python для описания ограничений на архитектурные модели. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 143-156. DOI: 10.15514/ISPRAS-2015-27(5)-8.

1. Введение

Архитектурное моделирование становится важным этапом разработки сложных систем ответственного применения. Оно позволяет на ранних этапах разработки выявлять ошибки и, тем самым, уменьшать затраты на разработку. Обычно архитектурное моделирование выполняется с использованием специализированных языков и сред моделирования, важной составляющей которых является функциональность по описанию ограничений на модели и верификации моделей. Поскольку основной фокус архитектурных моделей заключается в описании структуры целевой системы и связей между ее компонентами, то и ограничения на модели в первую очередь рассматриваются структурные в отличие от ограничений на поведение целевой системы в динамике.

В разных средах моделирования проверка структурных ограничений строится на основе различных подходов. Например, среда может проверить модель по тем требованиям, которые включены в язык моделирования. Такие требования обычно описываются в руководствах по языку моделирования, и их поддержка реализована непосредственно в среде моделирования. Любая модель должна выполнять все требования языка моделирования. Например, если в модели описана некоторая иерархическая структура из компонентов, обозначающая, что компонент одного уровня иерархии является частью компонента более высокого уровня, и для каждого компонента определен размер, то размер объемлющих компонентов не должен превышать сумму размеров вложенных компонентов.

Однако таких встроенных требований недостаточно для тщательной проверки: необходимо учитывать функциональные требования, требования внутренней согласованности модели, требования из предметной области, которые могут быть различными для разных моделей. При архитектурном моделировании проверка таких требований играет ключевую роль. Поскольку они могут быть различными для разных моделей, они не могут быть предопределенными, поэтому необходим способ записи этих требований пользователем. Способ

записи должен быть однозначен, понятен среде моделирования и достаточен для автоматической проверки модели.

Например, для языка архитектурного моделирования AADL (Architecture Analysis and Design Language) для этой цели был предложен специализированный язык REAL (Requirements and Enforcements Analysis Language) [1]. Спецификация на языке REAL состоит из набора «теорем», которые итеративно обходят модель, делают промежуточные вычисления и проверку различных условий. REAL проектировался как простой язык для описания ограничений. В частности, мы в своей практике проверки моделей столкнулись с недостатками этого языка: в нем нет ряда структур данных и операций, удобных, а иногда и необходимых для записи сложных ограничений. Некоторые из недостатков были решены в других языках, развивающих идеи языка REAL, но далеко не все [2]. Например, сложные проверки имеет смысл разбивать на более мелкие, особенно если более мелкие проверки повторяются в разных больших проверках. Причем необходима возможность отдельного описания часто используемых проверок или промежуточных вычислений и их использования для более крупных проверок и вычислений (то, что языках программирования достигается за счет механизмов подпрограмм и рекурсии). Таких возможностей нет ни в REAL, ни в его последователях, что ограничивает их практическое применение.

Гибкость – не единственное важное требование к языку описания ограничений. Приходится учитывать, насколько сложно изучить его конечным пользователям.

В данной статье мы предлагаем подход к построению языка описания ограничений, ключевая идея которого состоит в переиспользовании мощных возможностей традиционных языков программирования в удобном для пользователя виде. Также мы рассмотрим вопросы реализации подхода на примере языка архитектурного моделирования AADL.

2. Требования к языку описания ограничений

Язык описания ограничений нуждается в развитых возможностях, привычных для многих языков программирования, в частности, в богатом наборе выражений, операторов, переиспользуемых компонентов кода (функций и библиотек функций). Эти возможности можно разработать специально для того или иного языка описания ограничений, а можно воспользоваться возможностями существующего языка программирования, если разработать на его основе язык описания ограничений. Последний подход позволяет не только воспользоваться продуманными мощными возможностями языка, но и существующими инструментами разработки, библиотеками компонентов кода и опытом разработчиков. Пользователи могут быть уже знакомыми с данным языком программирования или они смогут воспользоваться имеющейся литературой, обучающими курсами или знаниями коллег по данному языку программирования.

С другой стороны, язык описания ограничений должен быть интегрирован с языком архитектурного моделирования. Он должен позволять описывать ограничения в терминах языка архитектурного моделирования, должен обеспечивать удобный синтаксис для доступа к компонентам модели и указания, для каких компонентов нужно проверять выполнение тех или иных ограничений. В языке AADL последнее решается за счет встроенного механизма расширений (annex). Решение остальных упомянутых проблем требует дополнительного исследования, но в качестве основы можно взять саму модель на языке AADL.

Язык описания ограничений может по-разному использовать язык программирования. Например, язык описания ограничений может переиспользовать только некоторые синтаксические конструкции (выражения, операторы и т.п.), а остальное в этом языке (в частности, семантика конструкций) будет сделано по-своему. Но возможно и более глубокое переиспользование, в котором правильные тексты на языке программирования будут правильными текстами и на языке описания ограничений. Последний подход позволит переиспользовать среды разработки, компиляторы, интерпретаторы, средства документирования, средства отладки и т.п. Однако при этом возможно чрезмерное усложнение синтаксиса и ухудшение читабельности кода, особенно если в языке программирования нет достаточных средств расширения.

Кроме вышеперечисленного при выборе языка программирования следует учитывать его распространенность и сложность изучения. Пользователям будет проще изучить и начать правильное использование расширения (известного им) языка программирования, чем полноценного нового языка спецификации.

Поскольку многие из перечисленных выше характеристик выполнены для языка программирования Python, мы предлагаем его в качестве основы для языка описания ограничений для языка архитектурного моделирования AADL. Мы реализовали поддержку этого языка в рамках среды архитектурного моделирования MASIW Framework [3, 4].

Мы считаем возможности языка программирования Python достаточно развитыми, существует большое число библиотек компонентов, реализованных на Python, существует большое число обучающих курсов и методической литературы. Инструменты разработки Python имеются практически для всех широко используемых сред разработки (в частности, и для среды Eclipse, на основе которой реализована среда MASIW Framework). В нашем подходе язык описания ограничений является полным расширением языка Python, т.е. любую программу на Python можно считать программой на нашем языке описания ограничений. Более того, это расширение реализовано непосредственно на самом языке Python и для пользователя выглядит как библиотека классов и функций.

Ограничения на компоненты могут быть размещены непосредственно в определениях компонентов. В языке AADL для этого есть такой механизм расширений, как annex. Ограничение представляется функцией на языке Python, помеченной декоратором. Проверяемая архитектурная модель доступна функции как объект, структура которого повторяет структуру модели. При программировании функций можно использовать известные навыки императивного, функционального и объектно-ориентированного программирования для языка Python. Проверка модели, снабженной ограничениями, выполняется достаточно просто – это выполнение всех функций, помеченных нужным декоратором.

При разработке языка описания ограничений и инструментов проверки модели возникают следующие задачи: какими конструкциями языка программирования следует пользоваться для описания ограничений, как интегрировать инструменты разработки на языке программирования в среду архитектурного моделирования. Далее рассмотрим эти задачи подробнее.

3. Интеграция языка Python в среду моделирования

От среды моделирования, в которой имеется поддержка языка описания ограничений, нужны такие возможности как удобное написание ограничений (с раскраской кода ограничений, с контекстными подсказками и автодополнением и т.п.), удобная проверка модели (скрывающая в себе технические особенности запуска компиляторов или интерпретаторов и показывающая результат запуска в виде, привычном при архитектурном моделировании), удобная отладка и т.п.

Среда MASIW Framework реализована на базе широко известной среды разработки Eclipse. Для получения удобного инструмента написания ограничений достаточно воспользоваться одним из плагинов для Eclipse для разработки на языке Python (например, плагином PyDev). Для проверки моделей готового плагина нет. От него требуется, чтобы он мог выполнить код ограничений над моделью, которой обладает среда моделирования, при помощи интерпретатора языка программирования. Поскольку этот интерпретатор не является частью среды моделирования, необходимо их специальным образом интегрировать. Эта интеграция может быть выполнена одним из трех следующих способов:

- вся проверка модели проводится средой моделирования, т.е. без использования отдельных компиляторов или интерпретаторов языка программирования; в этом подходе проверка модели встроена в среду моделирования, в частности, сама среда моделирования проводит синтаксический анализ и выполнение текстов ограничений, используя внутренние протоколы доступа к проверяемой модели; от языка программирования используется только синтаксис и семантика;
- вся проверка модели проводится вне среды моделирования; в этом подходе проверяемая модель отчуждается от среды моделирования в

виде кода на языке программирования, на котором написаны ограничения; ограничения без изменений встраиваются в этот код; полученный код выполняется также вне среды моделирования существующими компиляторами и интерпретаторами, а результат передается в среду моделирования;

- среда моделирования работает параллельно с интерпретатором языка программирования и предоставляет части модели по запросу интерпретатора; предполагается наличие в среде моделирования модуля, выполняющего преобразование запросов интерпретатора в запросы к модели в среде моделирования по ее внутренним протоколам.

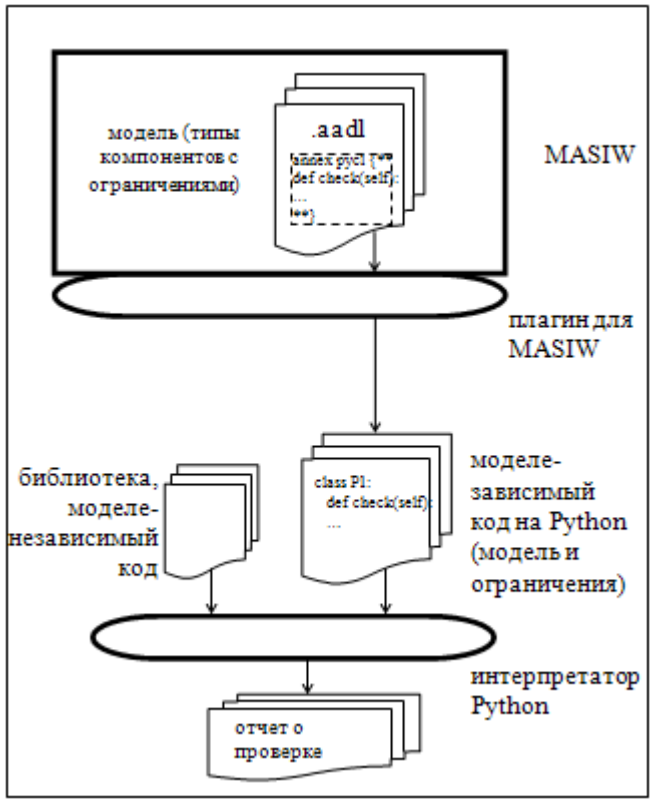


Рис. 1. Проверка модели в MASIW Framework.

Процесс проверки модели изображен на рис. 1. Ограничения пишутся внутри определений типов компонентов. Затем на основе этих определений специальный плагин для среды MASIW Framework генерирует набор моделе-

зависимых исходных текстов на языке Python. Эти тексты включают в себя код, повторяющий типы компонентов (сюда включены и ограничения, т.к. они находились в определениях типов компонентов), и генератор целевого проверяемого объекта (это объект одного из типов компонентов). Моделе-зависимый код соединяется с заранее подготовленным моделе-независимым кодом и библиотеками на Python, которые использованы в ограничениях. Полученный код выполняется обычным интерпретатором Python: этот код строит объект с проверяемой моделью и, итерируясь по его подкомпонентам, вызывает все функции-ограничения. В итоге получается отчет о сделанной проверке модели.

4. Представление проверяемой модели на языке программирования

Архитектурная модель на языке AADL состоит из определений типов компонентов. Этот тип описывает, что каждый компонент этого типа будет содержать определенные подкомпоненты и различные атрибуты. Ограничения зачастую формулируются для каждого компонента того или иного типа. Поэтому удобно включить ограничения в определения типа. Более того, один тип может наследовать другой тип (что означает включение всего, что описано в типе-предке в тип-потомок).

Все это приводит к выводу о родстве типов компонентов в AADL с классами в объектно-ориентированном программировании. Мы решили следовать этому и генерировать класс для каждого типа компонента. Атрибуты типа компонента становятся полями класса, функции-ограничения – методами класса.

Рассмотрим пример типа-процесса, в котором есть входной порт данных, и возможный код на языке Python для него:

```
process P1
features
    input: in data port FastPort;
…
end P1;
data FastPort
…
end FastPort;

class FastPort:
    …
class P1:
    def __init__(self):
        self.features = {}
```

```
            self.features["input"] = self.input =
    InDataPort(FastPort)
        …
```

Класс P1 соответствует типу компонента-процесса P1. Метод __init__ класса P1 – это конструктор объектов класса P1. Он создает все поля объекта (в данном случае, инициализирует таблицу для feature, создает объект класса FastPort и помещает его в эту таблицу).

У разных классов будут повторяющиеся части. Поэтому их можно вынести в отдельные классы и сократить код генерируемых классов:

```
    class P1(ComponentTypeInstance, Process):
        def __init__(self):
            super(P1, self).__init__()
        def init_features(self):
            self.features["input"] = self.input =
    InDataPort(FastPort)
…
```

В последнем случае метод init_features вызывается конструктором класса ComponentTypeInstance, который в свою очередь вызывается конструктором класса P1. Метод init_features переопределен в классе P1. В нем записана инициализация полей для features именно для типа P1.

Использования словарей для подкомпонентов, feature и т.п. позволяет использовать существующий синтаксис языка Python для итерирования по содержимому словаря (т.е. для итерирования по всем подкомпонентам, feature и т.п.). Для обращения к отдельным подкомпонентам, feature и т.п. заводятся дополнительные поля, имя которых совпадает с именем подкомпонента, feature и т.п. (в примере выше это input). Согласно стандарту языка AADL имена подкомпонентов, feature и т.п. должны различаться внутри одного и того же типа компонента и в них могут входить только такие символы, которые встречаются в идентификаторах в языках программирования.

Кроме полей из определений типов компонентов, для повышения удобства генерируются дополнительные поля и методы классов. Например, для соединений генерируются дополнительные поля, означающие концы соединения.

После того, как в классе сгенерированы конструкторы и поля из определения типа и удобные служебные поля и методы, в определение класса вставляется содержимое annex с ограничениями. Поскольку это содержимое уже написано на языке Python, то не требуется какая-либо сложная обработка и трансляция этого содержимого, чтобы им можно было пользоваться. Возможна лишь необходимость подстроить отступы в сгенерированном коде, чтобы он соответствовал отступам в содержимом из annex.

Чтобы приблизить язык Python и язык AADL, нужно решить еще одну задачу: Python – регистро-зависимый язык, а AADL – регистро-независимый, поэтому

необходимо сделать так, чтобы в коде ограничений можно было использовать произвольный регистр идентификаторов. Эта задача была решена, благодаря тому, что в Python можно управлять тем, как по имени поля получить объект-поле.

Наконец, поскольку цель ограничения – проверить модель, а не изменить ее, был реализован запрет изменения объектов, представляющих модель, из кода ограничений.

5. Описание ограничений на языке Python

Практика показывает, что среди различных требований очень востребованы требования внутренней согласованности компонентов того или иного типа. Компоненты могут включать подкомпоненты, сами компоненты и их подкомпоненты могут обладать свойствами и интерфейсными сущностями – они должны быть согласованы. Такое требование логично сопоставить с типом компонента и описать в виде функции («функции-ограничения»), которая получает ссылку на проверяемый объект соответствующего типа и возвращает булевское значение («истина», если требование выполнено для данного ей компонента, «ложь», иначе). Одним из способов сопоставления является включение функции в конструкцию определения типа компонента. Важно, что функция-ограничение может делать все то, что может делать программа на языке Python.

Для промежуточных вычислений тоже могут потребоваться функции, но эти функции не должны автоматически вызываться для проверки и не обязаны иметь указанную выше сигнатуру: нужен способ разграничения функций внутри определения одного и того же типа компонента, какие функции будут обозначать ограничения, а какие функции будут вспомогательными. В языке Python для этого есть такое средство, как декоратор. В частности, функция-ограничение – это будет функция с декоратором @constraint. Вспомогательная функция – это функция без этого декоратора.

Рассмотрим пример функции-ограничения check из типа P1, проверяющей следующее требование: «каждый компонент типа P1 должен иметь свойство Size и это свойство должно быть положительным числом». Посмотрите код этой функции (обратите внимание, что он записан на чистом Python):

```
annex pycl {**
…
  @constraint
  def check(self):
    return "Size" in self.properties and
self.properties["Size"] > 0
…
**};
```

Обратите внимание на то, что функция-ограничение check имеет аргумент для ссылки на объект, представляющий проверяемый компонент. Альтернативный подход – это безаргументные функции-ограничения (как сделано в языке REAL). Такие функции самостоятельно выполняют итерирование по всей модели и выбор подходящих объектов для проверки. Наличие явного аргумента дает возможность получить более детальную статистику проверки модели, а именно иметь для каждого проверяемого компонента множество функций-ограничений, равных истине («выполненные требования») и множество функций-ограничений, равных лжи («невыполненные требования»). Такая информация позволяет упростить локализацию ошибки и ее исправление.

Декораторы могут сами иметь аргументы. Декоратор @constraint может иметь декоратор с текстовым описанием проверяемого ограничения, удобное для чтения человеком. Такое неформальное описание может быть включено в отчет о проверке или использовано для организации трассировки требований:

```
  @constraint("Each P1 component must have a
Size property which must be positive")
  def check(self):
    return "Size" in self.properties and
self.properties["Size"] > 0
```

В следующем примере используется вспомогательная рекурсивная функция. Функция-ограничение может вызывать другие вспомогательные функции и функции-ограничения, что можно использовать для организации цепочек проверок:

```
  @constraint("Each P1 component must have a
Size property which must be a power of 2")
  def check(self):
      def ispower2(n):
          if n < 2 or n % 2 != 0:
              return False
          else:
              return ispower2(n/2)
    return "Size" in self.properties and
ispower2(self.properties["Size"])
```

Функция на языке Python может иметь несколько декораторов одновременно. Для примера мы определили декоратор @precondition. Он позволяет определить условие, при невыполнении которого проверка ограничения не имеет смысла. Например, требование «свойство Size должно быть степенью двойки» не имеет смысла, если такого свойства нет у компонента:

```
  @precondition(lambda self: "Size" in
self.properties)
```

```
  @constraint("Each P1 component must have a
Size property which must be a power of 2")
  def check(self):
       def ispower2(n):
            if n < 2 or n % 2 != 0:
                 return False
            else:
                 return ispower2(n/2)
     return ispower2(self.properties["Size"])
```

Для отладки программ часто пользуются т.н. «отладочной печатью». Это дополнительные операторы печати некоторых фраз, по наличию которых при выполнении программы судят о выполнении тех или иных участков программы и о значениях переменных в момент их выполнения. Во время выполнения проверки модели отладочная печать возможна, но для ее использования при отладке необходимо ее структурировать с привязкой к контексту проверки, т.к. отладочная печать разных вызываемых функций-ограничений сливается в единый текст. Чтобы избежать этой необходимости, мы добавили метод info(). Он имеет 1 аргумент для некоторой строки. Она будет вставлена в отчет, если выполнение дойдет до вызова этого метода. Этот метод автоматически запоминает контекст, в котором он вызван:

```
  @precondition(lambda self: "Size" in
self.properties)
  @constraint("Each P1 component must have a
Size property which must be a power of 2")
  def check(self):
       def ispower2(n):
            if n < 2 or n % 2 != 0:
                 return False
            else:
                 return ispower2(n/2)
     if self.properties["Size"] <= 0:
          self.info("Value of a Size property
is not positive")
          return False
     if not ispower2(self.properties["Size"]):
          self.info("Value of a Size property
is not a power of 2")
          return False
     return True
```

6. Заключение

В данной статье обсуждается построение языка описания ограничений на базе языка программирования. Предложенный подход позволяет переиспользовать богатые возможности языка программирования и существующую инфраструктуру (среды разработки, компиляторы, интерпретаторы, отладчики, средства документирования). Кроме того, данный подход должен позволить понизить порог вхождения в язык описания ограничений, если этот язык построен на базе хорошо известных концепций и синтаксиса. Существующих возможностей расширения языков программирования достаточно, чтобы описывать ограничения в терминах модели без изменения среды моделирования. Наконец, этот подход позволяет минимизировать затраты на разработку инструментов выполнения и разработки ограничений.

В качестве примера в статье использовалась реализация этого подхода в среде моделирования MASIW Framework. Для нее был разработан плагин, позволяющий проверять, выполнены ли в модели ограничения, описанные на расширении языка программирования Python. В качестве основы для реализации плагина использовался широко известный плагин PyDev к среде Eclipse, позволяющий разрабатывать программы на языке Python в этой среде.

Список литературы

[1]. O. Gilles, J. Hugues. Expressing and enforcing user-defined constraints of AADL models. Proceedings of the 5th UML&AADL Workshop. 2010. PP. 337-342.

[2]. D. Cofer, A.Gacek, S.Miller, M.W. Whalen, B. LaValley, L. Sha. Compositional Verification of Architectural Models. NASA Formal Methods, Proceedings of the 4th International Symposium. 2012. PP. 126-140.

[3]. D. Albitskiy, A. Khoroshilov, I. Koverninskiy, M. Olshanskiy, A. Petrenko, A. Ugnenko. AADL-Based Toolset for IMA System Design and Integration. SAE International Journal of Aerospace. 2012, 5. PP. 294-299. doi:10.4271/2012-01-2146.

[4]. Д.В. Буздалов, С.В. Зеленов, Е.В. Корныхин, А.К. Петренко, А.В. Страх, А.А. Угненко, А.В. Хорошилов. Инструментальные средства проектирования систем интегрированной модульной авионики. Труды Института системного программирования РАН. 2014, т.26, вып. 1. с.201-230. doi:10.15514/ISPRAS-2014-26(1)-6.

Python-Based Constraint Language for Architecture Models

[2]E. Kornykhin <kornevgen@cs.msu.ru>
[1,2,3,4]A. Khoroshilov <khoroshilov@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

[3] *Moscow Institute of Physics and Technology (State University)*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] *National Research University Higher School of Economics (HSE)*
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper presents an approach to specify constraints on AADL models in Python-based language and a toolset allowing to verify that constraints. The goal of the approach is to enable reusing of existing rich facilities of Python language, tools, and libraries as well as to reduce learning curve of engineers. Constraints must be placed into component annexes. These constraints must be written in Python programming language as functions with one argument (an object to be checked), Boolean result, and special decorator. A plugin for a modeling environment generates a program in Python from the model components declarations. While it is executing this program creates an object with the model instance and checks the object by functions from annexes. This approach is implemented in MASIW Framework that allows checking constraints on model instance. The implementation is made upon PyDev, a well-known Eclipse-plugin for Python developing in Eclipse and reuses integration of Eclipse with Python from PyDev. The Python sources generated are enough to involve automatically such PyDev tools as code coloring, auto-completion for classes (components in AADL), fields of classes (subcomponents, connections, features, etc. in AADL), methods of classes (constraints from annexes), hierarchy of packages and classes (according to components hierarchy in AADL).

Keywords: architecture modeling; constraints; Python; AADL; model verification.

DOI: 10.15514/ISPRAS-2015-27(5)-8

For citation: E. Kornykhin, A. Khoroshilov. Python-Based Constraint Language for Architecture Models. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 143-156 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-8.

References

[1]. O. Gilles, J. Hugues. Expressing and enforcing user-defined constraints of AADL models. Proceedings of the 5th UML&AADL Workshop. 2010. PP. 337-342.

[2]. D. Cofer, A.Gacek, S.Miller, M.W. Whalen, B. LaValley, L. Sha. Compositional Verification of Architectural Models. NASA Formal Methods, Proceedings of the 4th International Symposium. 2012. PP. 126-140.

[3]. D. Albitskiy, A. Khoroshilov, I. Koverninskiy, M. Olshanskiy, A. Petrenko, A. Ugnenko. AADL-Based Toolset for IMA System Design and Integration. SAE International Journal of Aerospace. 2012, 5. PP. 294-299. doi:10.4271/2012-01-2146.

[4]. D.V. Buzdalov, S.V. Zelenov, E.V. Kornykhin, A.K. Petrenko, A.V. Strakh,A.A. Ugnenko, A.V. Khoroshilov. Instrumental'nye sredstva proektirovanija sistem integrirovannoj modul'noj avioniki [Tools for System Design of Integrated ModularAvionics]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol. 26, n.1, pp. 201-230 (in Russian). doi: 10.15514/ISPRAS-2014-26(1)-6.

Использование симуляции сбоев при тестировании компонентов ядра ОС Linux

[1]А.В. Цыварев <tsyvarev@ispras.ru>
[1,2,3,4]А.В. Хорошилов <khoroshilov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В статье рассматриваются методы тестирования компонентов ядра ОС Linux с использованием симуляции сбоев. Основная цель таких методов – проверка поведения модуля при возникновении нештатных ситуаций, таких как сбои в аппаратуре или нехватка ресурсов.

Эти ситуации на практике встречаются достаточно редко и непредсказуемо, что существенно затрудняет их обнаружение и локализацию.

Единственным распространённым подходом к поиску таких проблем является случайное внесение сбоев в ходе выполнения обычных тестов.

Но из-за недетерминированной природы такой способ тестирования не обеспечивает достаточной воспроизводимости тестовых сценариев, что не позволяет давать никаких гарантий корректности поведения драйвера даже в случае успешного выполнения всех тестов.

В статье предлагаются новые методы систематического тестирования устойчивости к сбоям, решающие проблему воспроизводимости тестирования.

В основе этих методов лежит систематический перебор точек для симуляции сбоев с отбрасыванием потенциально "неинтересных" тестов, которые не проверяют ничего нового по сравнению с другими тестами. Применимость и эффективность конкретного систематического метода зависит от способа определения "неинтересных" тестов: это может быть как ручное выделение уникальных участков базовых тестов, ограничивающих множество точек для симуляции сбоев, так и автоматическое определение дублирующих точек по стеку вызовов функций тестируемого модуля. Была проведена апробация предложенных систематических методов на одном из драйверов Linux и двух тестовых наборах. Результаты апробации показывают, что систематические методы не уступают по эффективности методу со случайным внесением сбоев, а в чем-то и превосходят его.

Ключевые слова: Linux; компоненты ядра Linux; драйвер; тестирование; симуляция сбоев.

DOI: 10.15514/ISPRAS-2015-27(5)-9

Для цитирования: А.В. Цыварев, А.В. Хорошилов. Использование симуляции сбоев при тестировании компонентов ядра ОС Linux. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 157-174. DOI: 10.15514/ISPRAS-2015-27(5)-9.

1. Введение

Операционные системы, основанные на ядре Linux, широко используются в мире. Они обеспечивают работу более 90% мощнейших суперкомпьютеров из международного рейтинга TOP500 и преобладают на рынке операционных систем для мобильных устройств в составе ОС Android. Существенная часть компонентов ядра Linux является драйверами устройств - одними из ключевых компонентов Linux, отвечающих за организацию взаимодействия пользователя с различными внешними устройствами компьютеров: жесткими дисками, съемными накопителями, сетевыми адаптерами, и т. д. Кроме того, в Linux компонентами ядра реализованы и некоторые программные сущности, такие как сетевые протоколы и файловые системы. В некотором роде такие компоненты - "драйвера программных сущностей". Далее в статье будет использоваться название *драйвер* как общее обозначение для драйверов устройств и "драйверов программных сущностей".

В статье [1] на примере драйверов файловых систем были описаны основные требования к системе тестирования, позволяющей качественно проверять работоспособность драйверов. В качестве одного из требований приводилось тестирование работы драйвера при нехватке памяти и в случае сбоев устройства. Как эффективный способ такого тестирования предлагалось использовать симуляцию таких условий в функциях, вызываемых из тестируемого драйвера. Для драйвера это выглядит как будто вызванная им функция не смогла выделить память, не смогла записать данные на диск или считать данные с диска. Для обобщения такую симуляцию будем называть симуляцией сбоев.

В данной статье исследуются существующие методы тестирования с использованием такого рода симуляции, а также предлагаются новые методы.

Во втором разделе статьи будет рассмотрен наиболее прямолинейный способ тестирования с использованием симуляции сбоев – целенаправленная разработка каждого теста вместе с сценарием симуляции сбоев. Из-за трудоемкости разработки таких тестов, они практически не применяются. В третьем разделе будет описан уже существующий метод с использованием рандомизированного сценария симуляции сбоев. Хотя качественные показатели метода неплохи, недетерминированность сценария симуляции сбоев не позволяет использовать его результаты тестирования в качестве обоснования корректности драйвера. Поэтому в четвертом разделе

предлагаются методы систематической симуляции сбоев, которые могли бы преодолеть недостатки рандомизированного метода. В пятом разделе будут приведены результаты испытания описанных методов с существующими тестовыми наборами на одном из драйверов ОС Linux. В шестом разделе будет проведено сравнение описанных методов тестирования с симуляцией сбоев, и будут сделаны выводы об их применимости на практике.

2. Целенаправленная разработка тестов с симуляцией сбоев

Самый прямолинейный способ тестирования с симуляцией сбоев – разработка каждого теста вместе со сценарием симуляции сбоев. Под сценарием симуляции сбоев подразумевается выбор конкретных вызовов функций из драйвера для симуляции сбоев в этих вызовах. В дальнейших рассуждениях на такие вызовы функций будем ссылаться как на точки симуляции. Такая разработка тестов может основываться на поведении аналогичных тестов, но без симуляции сбоев. При выборе сценария симуляции можно учитывать исходный код драйвера и его логику, результаты измерения покрытия кода при запуске теста без сбоев и пр.

Теоретически, таким способом можно заставить драйвер работать во всех случаях поведения функций, сбои которых эмулируются. И таким образом исследовать корректность обработки всех возможных ошибок.

На практике же такой метод симуляции сбоев малоэффективен. Во-первых, разработка каждого теста вместе со сценарием симуляции сбоев очень трудоемка: необходимо исследовать исходный код драйвера, исследовать исходный код самого ядра или на основе опытных данных определить его точную логику. Во-вторых, получившиеся тесты будут применимы и обеспечивать необходимое покрытие только для конкретного драйвера и его конкретной версии и для конкретной версии ядра. Для других драйверов, других версий данного драйвера или других версий ядра такие тесты с большой вероятностью будут либо некорректны, либо бессмысленнены, так как их запуск не будет вынуждать драйвер пройти по задуманному пути. Такое ограничение тестов возникает из-за того, что их разработка должна учитывать реализацию драйвера и ядра, с которым драйвер очень тесно взаимодействует.

По причине трудоемкости разработки, целенаправленная разработка тестов со сценариями симуляции практически неприменима для поиска новых ошибок в драйверах. Таким способом можно только проверить наличие уже найденных ошибок, которые плохо выявляются другими методами, и их исправление (регрессионное тестирование). Далее в статье этот метод не рассматривается.

3. Рандомизированный метод симуляции сбоев

Наиболее распространенным методом симуляции сбоев при тестировании ядра Linux и его компонентов является использование рандомизированного сценария симуляции сбоев. Такой метод используется, например, в проекте Linux Test Project (LTP)[2]. При рандомизированном сценарии, в процессе выполнения теста каждый вызов функции выделения памяти, работы с диском и др. возвращает ошибку с некоторой вероятностью p ($0<p<1$). В таком окружении каждый тест выполняется несколько раз, что позволяет наблюдать реакцию драйвера на разные ошибки.

Применение такого метода симуляции сбоев не требует каких-либо специальных свойств от базовых тестов, которые запускаются в сбойном окружении. Поэтому в качестве базового набора тестов часто берут уже существующий набор тестов, который работает с данным драйвером.

Теоретически, выполняя конкретный (базовый) тест большое количество раз, с таким методом симуляции сбоев можно наблюдать поведение драйвера при всех последовательностях сбоев в тестовом сценарии. На практике же, количество повторений теста ограничено временем, отведенным на тестирование. Поэтому за одну итерацию тестов будет наблюдаться только часть из возможных последовательностей сбоев.

Вероятностная база сценария симуляции сбоя определяет недостаток такого подхода: итерация тестов не дает гарантии, что то или иное *конкретное* поведение драйвера будет проверено.

С другой стороны, при рандомизированном сценарии симуляции сбоев есть вероятность наблюдать одинаковые последовательности сбоев при разных запусках одного теста. Это ведет к увеличению времени тестирования без повышения качества, то есть уменьшается эффективность тестирования.

Хотя использованием рандомизированного метода симуляции сбоев обеспечивает неплохое качество тестирования (при адекватном базовом наборе тестов), слабая воспроизводимость тестовых сценариев не позволяет давать гарантий корректности поведения драйвера. Основным источником проблем с воспроизводимостью является, понятно, факт недетерминированности сценариев сбоев. Поэтому логичным способом улучшения воспроизводимости результатов тестирования было бы использование детерминированных сценариев симуляции.

4. Систематическая симуляция сбоев

Вместо того, чтобы запускать каждый тест несколько раз с одним и тем же рандомизированным сценарием симуляции сбоев, можно систематически перебирать сценарии симуляции из заранее сформированного множества, и запускать тест с каждым из таких сценариев.

Таким способом определяется класс методов т.н. систематической симуляции сбоев. Формально, каждый метод этого класса из набора *нормальных* тестов N (не использующих симуляцию сбоев) автоматически генерирует набор тестов S с симуляцией сбоев, каждый тест которого есть пара $s=(n, c)$, где $n \subset N$, а c выбирается из множества сценариев симуляции сбоев C. Как и в случае с рандомизированным сценарием, в качестве набора N выбирается уже существующий набор тестов.

При выборе метода систематической симуляции сбоев должны учитываться следующие критерии:

1. Качество тестирования с использованием набора тестов S.
2. Эффективность тестирования с использованием набора тестов S.
3. Применимость набора тестов S для большого множества драйверов и их версий.
4. Применимость метода для широкого класса базовых тестов.

Введем две гипотезы, описывающие тестируемый драйвер и исходный набор тестов.

Первая гипотеза предполагает, что драйвер работает по принципу «все или ничего»: если сбой не дает успешно выполнить запрос, то состояние драйвера возвращается к состоянию, которое драйвер имел до выполнения запроса. При этом считается, что в ходе возврата к исходному состоянию драйвера функции, допускающие сбой, не вызываются.

Это гипотеза выполняется для большей части драйверов и запросов к ним; при этом инициатору запроса, обнаружившему сбой, часто возвращается некоторый индикатор ошибки.

Вторая гипотеза уже касается исходных тестовых сценариев: трасса выполнения любого исходного теста не зависит от запуска этого теста. Своего рода, это предположение о детерминированности теста и драйвера.

Как и первая гипотеза, гипотеза о детерминированности выполняется или почти выполняется для большинства драйверов Linux. На тесты же эта гипотеза накладывает существенные ограничения. Которые, впрочем, выполняются для многих существующих тестов.

В условиях гипотезы «все или ничего» достаточно рассматривать сценарии с однократной симуляцией сбоев; тестирование с использованием более сложных сценариев может быть сведено к однократным сбоям. Например, корректность драйвера при запуске исходного теста с симуляцией сбоев в точках A и B можно проверить, выполнив запуск исходного теста с симуляцией с точке A, а затем повторив запуск исходного теста, но уже без запроса драйверу, включающего точку A, с симуляцией сбоев в точке B. (Здесь неявно считается, что эквивалент исходному тесту без запроса с точкой A также присутствует в исходном тестовом наборе.)

Из гипотезы детерминированности следует, что набор точек для однократной симуляции сбоев в исходном тесте устойчив. Этот набор точек можно выделить, запустив исходный тест без симуляции сбоев, а затем ссылаться на

каждую точку с возможной симуляцией сбоев, например, по ее порядковому номеру в тесте.

Все эти рассуждения приводят к следующему методу систематической симуляции сбоев.

4.1 Метод перебора всех возможных точек симуляции сбоев

Алгоритм тестирования:

1. Запускаем каждый тест n исходного набора N без симуляции сбоев, но собирая данные о возможных точках симуляции сбоев в процессе выполнения теста. В простейшем случае достаточно узнать общее количество таких точек $K(n)$.
2. Запускаем каждый тест n исходного набора N $K(n)$раз. В запуске номер k используем сценарий симуляции сбоя в точке номер k.

Исходя из рассуждений выше, такой метод тестирования с симуляцией сбоев дает *полный* тестовый набор: он проверяет драйвер во всех ситуациях, в каких его можно проверить, запуская тесты исходного набора с различными сценариями симуляции сбоев.

На первый взгляд, получившийся тестовый набор с симуляцией сбоев обладает также свойством минимальности: ни один тест из него нельзя выкинуть без ухудшения качества тестирования. Ведь запуск исходного теста с каждым новым сценарием однократной симуляции сбоев позволяет выполнить код драйвера, отвечающий за выполнение обработки сбоя в новом участке кода драйвера. На самом деле, это рассуждение неверно.

Во-первых, разные точки возможных сбоев в трассе выполнения исходного теста могут соответствовать одному и тому же участку кода драйвера. Это происходит, например, когда исходный тест повторяет запрос драйверу для достижения какого-то особенного состояния драйвера (например, исчерпания свободных блоков в файловой системе).

Во-вторых, разные тесты в исходном тестовом наборе могут выполнять одинаковую последовательность операций с драйвером в качестве подготовительной работы. Например, как для создания файла, так и для создания директории в файловой системе, ее требуется сначала примонтировать. В итоге тесты со сбоями в этих общих последовательностях операций дублируют друг друга.

На практике выходит, что подавляющая часть тестов в получившемся наборе тестов с симуляцией сбоев – «неинтересная», то есть в плане тестирования эти тесты не дают ничего нового по сравнению с остальными. Применимость метода быстро упирается в практическое ограничение по времени выполнения тестов при увеличении сложности исходных тестов и/или их количества.

Рассмотрим способы уменьшения количества «неинтересных» тестов, порожденных данным методом симуляции сбоев.

4.2 Выделение уникальных участков в коде тестов

Количество заведомо «неинтересных» тестов можно сократить, если в каждом исходном тесте выделить участки кода, выполняющие операции с драйвером, характерные только для данного теста, и никаких других. Такие участки кода теста будем называть "уникальными". Используя разметку уникальных участков кода тестов, метод простого перебора можно модифицировать следующим образом:

1. Запускаем каждый тест n исходного набора N без симуляции сбоев, но собирая данные о возможных точках симуляции сбоев в процессе выполнения уникальных участков кода теста. В простейшем случае достаточно узнать общее количество таких точек $U(n)$.
2. Запускаем каждый тест n исходного набора N $U(n)$ раз. В запуске номер u используем сценарий симуляции сбоя в точке номер u.

Если разметка выполнена корректно, то набор тестов, порожденный модифицированным методом, сохранит свойство полноты, но за счет значительно сокращенного общего количества тестов метод уже будет применим в реальных условиях.

Выделение уникальных участков кода в тестах должно выполняться человеком. Эта деятельность усложняется необходимостью учитывать информацию обо всех тестах в наборе. Тем не менее, это существенно проще, чем разработка сценариев симуляции для каждого теста вручную (см. главу 2).

Альтернативой ручному выделению "уникальных" участков кода является использование базовых тестов, в которых уже выделены 3 стандартных этапа теста: *подготовка*, *основная часть* и *освобождение ресурсов*. В этом случае логично ограничить симуляцию сбоев *основной частью* теста, которая отличает каждый тестовый сценарий от других. Понятно, что основная часть теста не обязана быть уникальной с точки его воздействия на драйвер, но сокращение времени тестирования по сравнению с неограниченной симуляцией сбоев может оказаться существенным для применимости метода.

4.3 Учет стека вызовов для отбрасывания неинтересных сценариев симуляции сбоев

Вместо того, чтобы вручную выделять «уникальные» участки кода тестов, можно использовать различные характеристические функции, использующие информацию о точке возможного сбоя, для автоматического определения дублирующих тестов с симуляцией сбоев. При этом метод простого перебора точек симуляцией сбоев будет обобщенно модифицирован следующим образом(считаем, что множество исходных тестов N упорядочено):

1. Запускаем каждый тест n исходного набора N без симуляции сбоев, но собирая данные о возможных точках симуляции сбоев в процессе

выполнения теста. Помимо общего количества $K(n)$ таких точек, для каждой точки вычисляем значение характеристической функции $h(n, k)$.

2. Множество Q - множество собранных значений характеристической функции h. Изначально пустое.
3. Для каждого n из N пробегаем числа k от 1 до $K(n)$. Если $h(n,k) \in Q$ (тест с таким значением характеристической функции уже выполнялся), то ничего не делаем. Иначе выполняем тест n с симуляцией сбоя в точке номер k и добавляем значение $h(n, k)$ в множество Q.

Одна из простых, и в тоже время эффективных характеристических функций - стек вызовов(call stack) в возможной точке симуляции:

$CS(n,k)$ = {стек вызовов в возможной точке симуляции номер k в процессе выполнения теста n}.

Под стеком вызовов в точке выполнения программы подразумевается рекурсивно генерируемая цепочка(массив) из точек в коде в формате

$$pair\ \{< \text{имя функции} >; < \text{смещение в коде функции} >\}$$

где первый элемент цепочки соответствует заданной точке выполнения, а каждый последующий - точке вызова функции, содержащей предыдущую точку. Генерация цепочки обрывается на точке входа в ядро.

Такая характеристическая функция одинаково классифицирует (возвращает одно и тоже значение) точки возможных сбоев, которые соответствуют одной и той же инструкции исходного кода ядра, цепочки вызовов функций для которых также совпадают.

Модификация метода тестирования с симуляцией сбоев, использующая такую характеристическую функцию, гарантированно выявляет случаи симуляции сбоев в общих участках инициализации различных тестов, и случаи симуляции сбоев в повторяющихся запросах при инициализации одного теста. За счет этого набор тестов с симуляцией сбоев значительно сокращается, и становится пригодным для использования.

Однако, при сравнении стеков вызовов в точках симуляции сбоев не учитываются возможные различия в классификации этих сбоев в коде их *обработки*. Одна из часто встречающихся ситуаций, когда такое различие существенно, это выделение памяти в цикле:

```
for(i = 0; i < 10; i++) {
   p[i] = kmalloc(…);
   if(!p[i]) {
      for(i--;i >= 0; i--) {
         kfree(p[i]);
   }
}
```

Рис. 1. Выделение памяти с цикле.

Хотя стеки вызовов при выделении памяти в первой(i = 0) и второй (i = 1) итерациях не отличаются, трасса обработки сбоев в этих выделениях памяти отличается: вложенный цикл выполняется только во втором случае.

Другая ситуация - зависимость кода обработки сбоев от переменных состояния:

```
p = kmalloc(…);
if(!p) {
   if(panic_on_error) panic(…);
   …
}
```

Рис 2. Зависимость обработки сбоя от переменной состояния.

В этом примере стек вызовов в выделении памяти не зависит от значения переменной *panic_on_error*. Но код обработки сбоя в этом выделения памяти зависит от этой переменной.

Как видно их этих примеров, использование $CS(n,k)$ в качестве характеристической функции для метода простого перебора точек симуляцией сбоев приведет к потере качества тестирования. А именно, теряются тесты, проверяющие код обработки сбоя в разных условиях.

Для того, чтобы смягчить эти потери, можно использовать следующие вариации характеристической функции $CS(n,k)$:

1) $CS\_1(n,k)$ = pair{$CS(n,k)$; 1, если $\exists m<k$: $CS(n,m) = CS(n,k)$, 0 иначе}. Такая функция отличает точку возможной симуляции сбоев, в которой заданный стек вызовов наблюдался впервые в тесте, от точки, стек вызовов которой наблюдался в какой-то предыдущей точке теста.
2) $CS\_set(n,k)$ = pair{$CS(n,k)$; $\cup CS(n,m)$, $m<k$}.

Характеристическая функция *CS_1* отличает (возвращает различные значения) точку возможной симуляции сбоев, в которой заданный стек вызовов наблюдался впервые в тесте, от точки, стек вызовов которой наблюдался в какой-то предыдущей точке теста. Таким образом, характеристика сценария симуляции сбоев в первой итерации цикла отлична от характеристики следующей итерации (пример на рис. 1).

Характеристическая функция *CS_set* также позволяет отличать сбои в первой и последующих итерациях циклов. Но помимо этого, эта функция позволяет отличать сбои с разными переменными состояниями (пример на рис.2), если эти переменные влияли на классификацию предыдущих запросов драйверу, в которых возможны сбои.

5. Апробация методов и ее результаты

Описанные методы тестирования с симуляцией сбоев (кроме целенаправленной разработки таких тестов, описанной в разделе 2) были применены для тестирования драйвера файловой системы ext4.

Для реализации сценариев симуляции сбоев была выбрана платформа KEDR[3] и основанный на ней инструмент KEDR Fault Simulation. Как описано в статье[4], этот инструмент предназначен для симуляции сбоев в функциях, вызываемых драйвером, и он уже содержит все необходимое для сценария однократной симуляции в каждой точке, в том числе с ограничением по уникальному коду тестов. Сценарий для рандомизированной симуляции сбоев был также реализован с использованием уже существующей функциональности этого инструмента. Реализация сценариев, учитывающих стек вызовов, потребовала написания новых модулей ядра (т.н. индикаторы для KEDR Fault Simulation).

В качестве меры качества того или иного метода тестирования с симуляцией сбоев использовался прирост покрытия по коду драйвера ext4. А именно, количество строчек кода драйвера, покрытых в результате тестирования с использованием симуляции сбоев, но не покрытых при простом запуске тестов (без симуляции).

В приведенных ниже таблицах и их анализе используются следующие обозначения методов симуляции сбоев:

- *fsim_rnd* – рандомизированный сценарий (вероятность сбоя и количество повторений указываются дополнительно)
- *fsim_all* – однократная симуляция сбоев в каждой возможной точке
- *fsim_restricted* - однократная симуляция сбоев в каждой точке из заранее выбранного участка теста
- *fsim_stack_1* – однократная симуляция сбоев в каждой точке с отбрасыванием дублирующих сценариев, используя характеристическую функцию *CS_1*.

- *fsim_stack_set* – однократная симуляция сбоев в каждой точке с отбрасыванием дублирующих сценариев, используя характеристическую функцию *CS_set*.

В качестве исходных тестовых наборов были взяты:

1. Linux File System Verification Tests[5].
2. 10 тестов из набора Xfstests[6]. Тесты были отобраны с учетом применимости предположения о детерминированности тестов.

5.1 Linux File System Verification Tests (LFSVT)

Этот тестовый набор включает модульные тесты(unit tests), использующие достаточно простые сценарии. Эти особенности позволили в каждом таком тесте выделить участки кода, присущие именно этому тесту. Используя такие участки кода как «уникальные» (см. раздел 3.2), можно применять различные методы симуляции сбоев с ограничением области симуляции этими участками.

С другой стороны, количество тестов LFSVT и время их запуска достаточно велико, что делает неактуальным применение рандомизированных сценариев симуляции сбоев. Например, при вероятности сбоев 1% и повторении каждого базового теста 100 раз общее количество тестов с симуляцией сбоев будет больше 1 млн., а оценка времени их выполнения – 7 дней.

Табл. 1. Применение методов симуляции сбоев для тестов LFSVT

Метод симуляции сбоев	Ограничение симуляции уникальными участками кода теста	Прирост покрытия, строк кода	Время тестирования, мин	Стоимость прироста покрытия, мин/строк
(без симуляции)	-	-	110	-
fsim_restricted	да	311	92	0,30
fsim_stack_1	да	266	2	0,0075
fsim_stack_set	да	266	3	0,011
fsim_all (теор. оценка)	нет	-	5000	-
fsim_stack_1	нет	333	4	0,012
fsim_stack_set	нет	354	9	0,025

Как видно из таблицы, при учете разметки уникальных участков кода тестов автоматическое отбрасывание «похожих» сценариев симуляции сбоев действительно ухудшает качество тестирования: прирост покрытия от методов *fsim_stack_1* и *fsim_stack_set* меньше, чем у *fsim_restricted*.

С другой стороны, такое отбрасывание делает возможным тестирование с симуляцией сбоев без учета уникальных участков кода тестов: запуск всех (~500 тыс.) тестов с однократной симуляцией потребовал бы несколько дней, в то время как тестирование с методами *fsim_stack_1* и *fsim_stack_set* выполняется меньше, чем за 10 минут.

Более того, использование методов *fsim_stack_1* и *fsim_stack_set* без ограничения области сбоев дает больший прирост покрытия (при значительно меньшей стоимости), чем использование метода *fsim_restricted* с учетом размеченных «уникальных» участков кода тестов. Это говорит о неточности разметки. Детальный анализ покрытия и кода тестов подтвердил недостатки в разметке.

5.2 Тестовый набор xfstests

Набор xfstests состоит из тестов системного уровня, которые используют более сложные сценарии, чем тесты Spruce. Не все тесты из этого набора обеспечивают детерминированное воздействие на драйвер ФС ext4, поэтому для экспериментов использовалась только часть из них (10 тестов), воздействие которых на драйвер ФС ext4 близко к детерминированному.

Так как разметки «уникальных» участков кода в этих тестах нет, то метод простого перебора точек симуляции сбоев для таких тестов потребовал бы недопустимо много времени. Однако систематические методы, использующие фильтрацию сценариев на основе стека вызовов, вполне применимы.

Сложные сценарии тестов и небольшое количество этих тестов позволяют использовать рандомизированные сценарии для симуляции сбоев в этих тестах. Количество повторений тестов с рандомизированными сценариями было выбрано 200, что обеспечило сопоставимость суммарного времени такого тестирования с детерминированными методами.

Несмотря на то, что тестовые сценарии в наборе Xfstests вполне детерминированные, алгоритм работы драйвера ext4 под воздействием этих тестов не является в точности детерминированным. Для более точного сравнения методов, тестовые наборы, основанные на этих методах, выполнялись несколько раз, а результаты усреднялись. Для этого использовалась следующая методика испытаний:

1. Базовый набор (без использования симуляции сбоев) запускался 10 раз. В базовое покрытие включались строки, покрытые хотя бы в одной из запусков.
2. Каждый набор тестов с симуляцией сбоев выполнялся 5 раз. В качестве усредненного прироста покрытия учитывались строки, покрытые как минимум в половине запусков этого набора (3 из 5), но не включенные в базовое покрытие. В качестве времени тестирования бралось среднее время по 5 запускам.

Так как качество тестирования с использованием рандомизированного метода зависит от вероятности сбоя, то вначале теоретико-экспериментальным образом был выделен промежуток вероятностей (в процентах), на котором достигается максимум покрытия по коду драйвера. Этим промежутком оказался [0.5; 2.0]. Затем с шагом 0.1% на этом промежутке каждая вероятность испытывалась в соответствии с п.2 приведенной выше методики. Следующий график отражает результат этих испытаний.

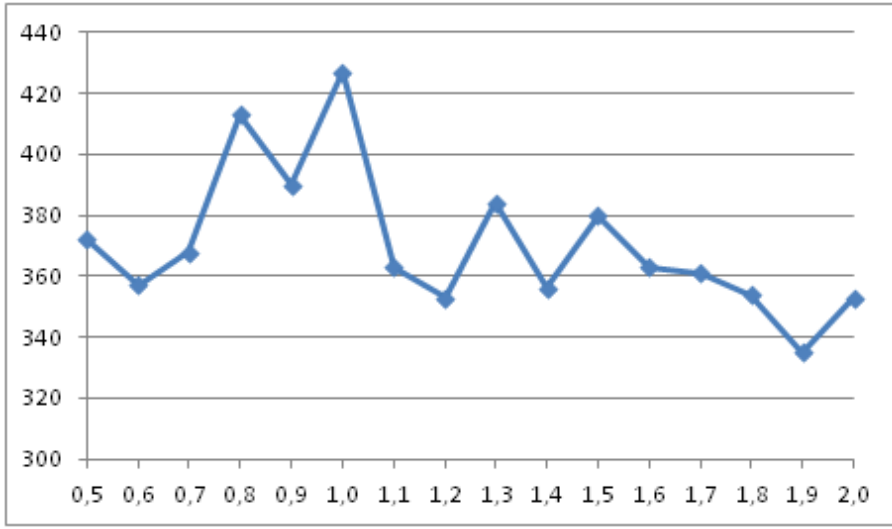


Рис.3 Прирост покрытия по коду в зависимости от вероятности рандомизированного метода

Как видно из графика, наибольшее покрытие по коду обеспечивается при вероятности сбоя 1%. Результаты запуска с такой вероятностью сбоя и сравнивались с результатами испытаний остальных методов.

Табл. 2. Применение методов симуляции сбоев для тестов Xfstests

Метод симуляции сбоев	Прирост покрытия, строк кода	Время тестирования, мин	Стоимость прироста покрытия, мин/строк	Дисперсия прироста покрытия, строк кода
(без симуляции)	-	2	-	-
fsim_all (теор. оценка)	-	10 000	-	-
fsim_rnd (p=1%)	427	137	0,32	47,2
fsim_stack_1	397	64	0,16	7,3
fsim_stack_set	439	209	0,48	13,6

Как видно из таблицы, прирост покрытия с использованием детерминированных методов симуляции сбоев не сильно отличается от

прироста покрытия при использовании рандомизированного метода. Но дисперсия - показатель воспроизводимости тестирования - у детерминированных методов значительно меньше (в 4 раза у метода *fsim_stack_set*). Различия между детерминированными методами, *fsim_stack_1* и *fsim_stack_set*, заключаются в существенно ускоренном тестировании (в 3 раза) первым из них, но в немного большем приросте покрытия от второго (30 строк кода или 8%).

6. Сравнение методов тестирования с симуляцией сбоев

Объединяя результаты испытаний с нашими рассуждениями, получим следующую таблицу, отражающую сравнительные характеристики каждого из метода.

Табл. 3. Сравнение методов тестирования с симуляцией сбоев

Критерий	*fsim_rnd*	*fsim_restricted*	*fsim_stack_1*	*fsim_stack_set*
Простота разработки и поддержки тестов	+	±	+	+
Полнота тестирования (прирост покрытия)	+	±	±	+
Стоимость прироста покрытия	±	∓	+	±
Стабильная воспроизводимость тестовых сценариев	∓	±	±	±

Хотя стоимость прироста покрытия для метода *fsim_restricted* по абсолютной величине (0,30. см. табл. 1) примерно совпадает с соответствующей характеристикой метода *fsim_rnd* (0,32, см. табл. 2), эти величины получены для разных тестовых наборов (Spruce и Xfstests соответственно). А характеристики методов *fsim_stack_1* и *fsim_stack_set* в применении к этим наборам сильно отличаются. Поэтому отличаются и сравнительные оценки методов *fsim_restricted* и *fsim_rnd* по этой характеристике.

Как видно из сравнительной таблицы, метод однократной симуляции сбоев с различными модификациями является хорошей альтернативой рандомизированному методу, обеспечивая гораздо большую воспроизводимость тестовых сценариев. Из минусов этого метода существенным является его требование детерминированности базовых тестов. Хотя в случае небольших отклонений от этого требования метод не теряет своей эффективности (см. табл. 2).

Различия между модификациями метода однократной симуляции сбоев более тонкие и прослеживаются из численных результатов апробации (табл. 1):

- *fsim_restricted* обеспечивает лучшее качество тестирования(полноту), требуя взамен (ручное) выделение «уникальных» участков кода в базовых тестах, что довольно трудоемко.
- Даже в случае небольших ошибок при выделении "уникальных" участков кода тестов в методе *fsim_restricted*, этот метод может проиграть по качеству тестирования методам *fsim_stack_1* и *fsim_stack_set*, которые вообще не требуют ручной предобработки базовых тестов.

Выбор между методами *fsim_stack_set* и *fsim_stack_1* – это выбор между немного лучшим покрытием по коду, обеспечивающимся первым из них, и существенно лучшей эффективностью второго.

7. Заключение

В статье были рассмотрены методы тестирования драйверов ОС Linux с симуляцией сбоев, позволяющие проверять работу драйвера в ситуациях, редко встречающихся в реальной работе. Существующий метод, использующий рандомизированные сценарии сбоев, обеспечивает достаточно качественное тестирование, но его недетерминированная природа не позволяет делать выводы о корректности протестированного драйвера. Для решения этих проблем в статье предлагаются методы, основанные на систематическом переборе точек возможной симуляции. Хотя полный перебор таких точек в тестах является неэффективным, методы, основанные на несложных ограничениях этих точек симуляции, вполне применимы на практике и по качеству сравнимы с рандомизированным методом. Что и было подтверждено экспериментами, проведенных на двух существующих тестовых наборах.

Список литературы

[1]. А.В. Цыварев, В.А. Мартиросян. Тестирование драйверов файловых систем в ОС Linux. Труды Института системного программирования РАН, том 23, 2012 г. ISSN 2079-8156. Стр. 413-426.

[2]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.

[3]. KEDR Project, http://linuxtesting.org/kedr.

[4]. Е.А. Герлиц, В.В. Кулямин, А.В. Максимов, А.К. Петренко, А.В. Хорошилов, А.В. Цыварев. Тестирование операционных систем. Труды Института системного программирования РАН, том 26, Выпуск 1, 2014 г. ISSN 2079-8156. Стр. 73-108.

[5]. Linux File System Verification Project, http://linuxtesting.org/spruce

[6]. Xfstests source code, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git

Using Fault Injection for Testing Linux Kernel Components

[1]*A.Tsyvarev <tsyvarev@ispras.ru>*
[1,2,3,4]*A.Khoroshilov <khoroshilov@ispras.ru>*
[1]*Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Moscow Institute of Physics and Technology (State University)*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] *National Research University Higher School of Economics (HSE)*
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper considers methods of using fault injection for testing components of Linux kernel. The main goal of the methods is to check how the kernel behaves in abnormal situations such as lack of resources and hardware faults. Such situations happens quite rarely and unpredictably, that makes very difficult to detect and to localize bugs in the code responsible for their handling. The most widely-used approach to find the bugs is random fault injection during execution of normal tests. Random fault injection works much better than absence of abnormal testing at all, but it has a number of drawbacks. Its random nature does not allow to reproduce test scenarios and it does not allow to estimate absence of bugs even if all tests are passed.

The paper presents a method of systematic fault injection for robustness testing that also enables reproducibility of test scenarios out of box.The basic idea of the method is to detect all execution points where a fault can happen, to filter out the points on the base of some equivalence relation and to execute tests bringing code into the chosen execution points with injecting faults at that points.Applicability and efficiency of the approach depends on the equivalence relation. The equivalence relation can be defined using manual markup of important segments of tests or it can be defined completely automatically on the base of a set of stack traces describing a fault injection.Experimental data were collected using test suites for Linux kernel file system implementations augmented with random and systematic fault injection.The experiments demonstrate the efficiency of systematic approach and acknowledge benefits coming from its deterministic nature.

Keywords: Linux; Linux kernel; file system; driver; testing; fault simulation.

DOI: 10.15514/ISPRAS-2015-27(5)-9

For citation: A.Tsyvarev, A.Khoroshilov. Using Fault Injection for Testing Linux Kernel Components. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 157-174 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-9.

References

[1]. A.V. Tsyvarev, V.A. Martirosyan. Testirovanie drajverov fajlovyh sistem v OS Linux [Testing of Linux File System Drivers]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2012, vol. 23, pp. 413-426 (in Russian). doi: 10.15514/ISPRAS-2012-23-24.

[2]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.

[3]. KEDR Project, http://linuxtesting.org/kedr.

[4]. E.A. Gerlits, V.V. Kuliamin, A.V. Maksimov, A.K. Petrenko, A.V. Khoroshilov, A.V. Tsyvarev. Testirovanie operacionnyh sistem [Testing of Operating Systems]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2014, vol. 26, n.1, pp. 27-72 (in Russian). doi: 10.15514/ISPRAS-2014-26(1)-3.

[5]. Linux File System Verification Project, http://linuxtesting.org/spruce

[6]. Xfstests source code, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git

Об интеграции формальных методов в задачах верификации операционных систем[1]

[1,2,3,4]А. К. Петренко <petrenko@ispras.ru>
[1,2,4]В. В. Кулямин <kuliamin@ispras.ru>
[1,2,3,4]А. В. Хорошилов <khoroshilov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В данной работе ставится задача разработки методов качественной верификации операционных систем, перспективным подходом к решению которой является интеграция различных техник верификации. Результатом успешного решения этой задачи может считаться комбинация методов верификации, которая позволяет проверить всю систему в целом и при этом более тщательно верифицировать наиболее важные ее компоненты и функции, используя для этого более строгие и формальные подходы. На основе опыта ИСП РАН, полученного при выполнении многочисленных проектов по верификации операционных систем с помощью различных методов выявлены артефакты разработки, являющиеся удобными кандидатами на роль точек сопряжения методов статической и динамической верификации ядра операционной системы на основе формальных спецификаций.

Выделение общих (разделяемых) артефактов предлагается провести на основе рассмотрения типовых задач верификации. Типовые задач, встречающиеся при использовании различных техник верификации, дают основу для интеграции техник и процессов верификации. К таким типовым задачам относятся: определение программного контракта модулей (функций), построение модели окружения, построение пути, демонстрирующего ошибку, увязка уровней абстракции и оценка полноты верификации. В наибольшей степени на роль артефактов представляющих собой точки интеграции претендуют контрактные спецификации, модели окружения и метрики (измерения) полноты верификации.

[1] Работа поддержана грантом РФФИ 14-01-00484.

Ключевые слова. Дедуктивная верификация, проверка моделей, тестирование на основе моделей, операционная система, интеграция методов верификации, программные контракты.

DOI: 10.15514/ISPRAS-2015-27(5)-10

Для цитирования: Петренко А.К., Кулямин В.В., Хорошилов А.В. Об интеграции формальных методов в задачах верификации операционных систем. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 175-190. DOI: 10.15514/ISPRAS-2015-27(5)-10.

1. Введение

Размеры и сложность современного программного обеспечения (ПО) обычно не позволяют провести верификацию достаточно высокого качества. Это связано как с большим количеством разнородных функций и компонентов, входящих в типичную программную систему, так и с практическими ограничениями самих применяемых технологий верификации. Подобные ограничения в первую очередь касаются формальных методов верификации, дающих самые надежные гарантии корректности верифицированных систем. Причины таких ограничений разнообразны – это могут быть низкая выразительность или, наоборот, чрезмерная усложненность формального аппарата, лежащего в основе метода, ограниченность конкретных инструментов, а также, что является на практике важнейшим фактором, сложность, высокая стоимость и длительность выполнения самих формальных методов верификации.

Интеграция различных формальных методов верификации является перспективным подходом к решению проблемы преодоления таких ограничений. На практике бывает важно совместить результаты верификации различных компонентов сложной системы, проведенные разными командами с помощью различных техник. Другая часто встречающая задача - найти приемлемую комбинацию методов и технологий верификации, которые позволяют проверить всю систему в целом и, одновременно, сфокусировать усилия на отдельных, наиболее критических компонентах и функциях, используя в их верификации наиболее строгие, но и более сложные и дорогостоящие методы. В данной статье мы рассматриваем вторую задачу применительно к ядру операционной системы как к системе с повышенными требованиями по показателям надежности и корректности.

2. Достижения в направлении интеграции методов верификации

При решении задач интеграции методов верификации просматривается как минимум два аспекта. Первый – это повторное использование (reuse) артефактов верификации и возможностей инструментов разных технологий, которые применялись к одним и тем компонентам проверяемой системы. Второй – это стыковка процессов верификации взаимодействующих компонентов, каждый из которых в отдельности верифицировался при

помощи разных методов и инструментов, и увязывание получаемых при этом артефактов.

Проблемы интеграции методов верификации и, в частности, формальных методов рассматриваются исследовательским сообществом достаточно давно - международная конференция по интеграции формальных методов проводится уже с 1999 года [1]. Пожалуй, наибольшим продвижением в этом направлении можно назвать широкое использование инструментов поиска решений систем логических утверждений, так называемых решателей (solvers), таких как Z3 [2,3], CVC4 [4,5] и др., в рамках различных технологий верификации. Они успешно применяются как компоненты многих инструментов динамической верификации (например, SpecExpolrer [6], Pex [7], SAGE [8]), инструментов верификации программ с извлечением их моделей (software model checking, например, BLAST [9], Static Driver Verifier [10], CPAchecker [11]), а также инструментов дедуктивной верификации (например, Frama-C [12], Rodin [13]).

Авторы тоже обращались к теме интеграции методов верификации в нескольких работах, например в [14] и [15]. В данной статье рассматривается еще одно направление интеграции, которому ранее не уделялось должного внимания.

Анализируя публикации на тему интеграции формальных методов, можно прийти к выводу, что наиболее активно разрабатывается тема интеграции подходов, построенных на различных формализмах моделирования поведения. Задача собственно стыковки самих методов рассматривается на примерах частных приложений. Мы считаем, что есть еще одно измерение интеграции, на которое следует обратить внимание, – это комплексная верификация некоторого важного класса систем, обладающих специфическими особенностями архитектуры и имеющих некоторую общую область характеристик функциональности, определяющую их назначение и являющуюся по существу объектом анализа в ходе верификации. В рамках данной статьи в качестве такого класса систем мы рассматриваем операционные системы, в более узком смысле – ядра операционных систем.

3. Терминология и особенности предметной области

В этом разделе мы кратко перечислим особенности операционных систем (ОС) как объекта верификации и так же кратко рассмотрим терминологию, используемую в рамках методов и техник верификации, чтобы используемые далее термины понимались однозначно.

Имеется две основные схемы верификации ядра ОС.

- *Модульная верификация* (unit verification). Все ядро разбивается на модули со строго заданными интерфейсами, при помощи которых осуществляется взаимодействие модулей между собой, с пользователями и устройствами, включая сетевые устройства. Интерфейсы специфицируются, и каждый модуль проверяется (верифицируется) на соответствие заданным спецификациям.

Результат верификации при условии осуществления всех проверок гарантирует формальную корректность модуля в предположении, что спецификации сами по себе корректны и полны.

- *Системная верификация* (system verification). Ядро ОС рассматривается как «черный ящик» (иногда как «серый ящик», т.е., используется определенная информация о его структуре и потоках данных внутри ядра), определяются только видимые извне интерфейсы ядра, в частности, системные вызовы и информационные потоки на уровне аппаратных интерфейсов (процессор, средства хранения информации, сетевые карты и др.). В случае «серого ящика» можно рассматривать интерфейсы на уровне драйверов и/или на уровне аппаратных абстракций (HAL, Hardware Abstraction Layer). Задается спецификация правильного видимого извне поведения ОС. Ставится задача верификации соответствия кода ОС данной спецификации. Можно сказать, что статически такая задача верификации может решаться только для небольших ОС (размер ее кода в 10 тыс. строк – практический предел, более реалистично – 1-2 тыс. строк), также важно условие полной спецификации аппаратуры. Как система в целом, так и отдельные модули могут рассматриваться с большим или меньшим количеством деталей. Как правило, имеет смысл сначала строить модель модуля/системы, верифицировать ее, а потом переходить к верификации либо более детальной модели, получаемой с помощью уточнения абстрактной, либо собственно реализации. При наличии некоторых условий верификация более детальных моделей может опираться на верификационные артефакты, полученные при верификации более абстрактных моделей, что снижает общую трудоемкость работ. Однако это не всегда удается обеспечить.

Основные причины возникновения специфических проблем при верификации операционных систем следующие.

- Выполнение различных функций ОС управляется *внешними событиями* (ОС являются примером event driven system). Этими событиями являются, в первую очередь, обращения к системным вызовам из приложений, а также обращения к обработчикам прерываний со стороны аппаратного обеспечения. По сути, отсутствует некоторый четкий сценарий обращений к различным функциям ядра ОС, хотя обычно существуют некоторые плохо формализованные ограничения на последовательности вызовов и возможные параллельные обращения к ним (а также повторные обращения из обработчиков прерываний, сработавших во время их выполнения, и пр.).
- Большинство ОС должны поддерживать *параллелизм*, т.е., наличие одновременно многих процессов и потоков управления в

приложениях, разделение ресурсов между процессами и параллельные обращения из разных потоков к различным функциям ядра. В том числе, должна учитываться реальная параллельность работы центрального процессора, различных контроллеров и внешних устройств.

- Современные ОС поддерживают *многоядерность*, т.е., должны распределять выполнение процессов и потоков на несколько управляющих устройств процессора, работающих параллельно, с учетом тех ограничений, которые накладываются на корректное распределение задач.
- Многие ОС имеют требования по соблюдению *ограничений реального времени*, связанных как с ограничениями по времени выполнения административных функций самой ОС, так с мониторингом временных ограничений на работу приложений.
- ОС имеют определенные требования к *доступности и надежности* работы, обуславливающие необходимость обработки разнообразных отказов оборудования.

Основной вывод, который важен для нас, состоит в том, провести качественную верификацию операционной системы на практике одним методом невозможно. По этой причине операционные системы являются хорошим полигоном для экспериментов по применению перспективных методов верификации и, в частности, интегрированных подходов к верификации.

Дадим краткие определения используемых терминов. Более полное описание терминов и понятий верификации можно найти в обзоре [16].

- *Верификация* проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов правилам и стандартам разработки. В частности, верификация проверяет соответствие между нормами стандартов, описанием требований (техническим заданием) к ПО, проектными решениями, исходным кодом, пользовательской документацией и функционированием самого ПО.
- *Спецификация* в широком толковании это достаточно подробное, полное и точное описание. В контексте формальных методов под спецификацией понимается формальное описание требований к программной системе, ее поведению, ее функциональных и нефункциональных свойств (например, требований к устойчивости, несмотря на ошибки оборудования, или к производительности и пропускной способности). Поскольку спецификация практически всегда охватывает лишь часть свойств и характеристик реальной системы, часто говорят о спецификации определенной модели, или просто о модели. В таком контексте термины «спецификация» и

«модель» обычно используются как синонимы.
Объектами спецификации могут быть поведение (функциональные свойства), нефункциональные свойства (защищенность, надежность, производительность, переносимость, совместимость и пр.), язык описания чего-либо (в этом случае выделяют синтаксис, статическую и динамическую семантика, прагматические правила и шаблоны), протокол взаимодействия (в телекоммуникационных и интерфейсных стандартах) и др.

- *Реализация* – это проверяемая система, являющаяся объектом верификации. Рассматриваются различные формы представления реализации (например, исходный код, двоичный код, наблюдаемое поведение и внутренняя динамика работы кода).
- *Статический анализ* – анализ свойств системы, выполняемый (обычно специализированными инструментами) без ее реальной работы за счет ее моделирования на основе анализа/интерпретации исходного кода или др. артефактов разработки; *динамический анализ* – анализ свойств системы, выполняемый в ходе ее реальной работы и с использованием наблюдения за этой работой. Во многих случаях эти виды анализа используются совместно и разделяются с трудом (например, для анализа используется симуляция работы кода системы в специализированной виртуальной машине), тогда говорят о *статико-динамическом анализе*.
- Виды динамического анализа.
    - *Тестирование* – оценка соответствия поведения реализации требованиям по ее работе в рамках заранее выбранного (сгенерированного) набора сценариев или ситуаций, который получен для обеспечения определенного критерия полноты (связанного с покрытием всех типичных ситуаций, задаваемых кодом и/или описанных в требованиях).
    - *Мониторинг* (run-time verification или пассивное тестирование, passive testing) – оценка соответствия поведения реализации требованиям по ее работе в рамках реальной эксплуатации или некоторой похожей деятельности (в которой возникающие ситуации выбираются, не для достижения покрытия этих ситуаций, а для достижения каких-то с точки зрения верификации целей). Сам анализ выявленного поведения может выполняться в ходе работы системы (online) или после ее завершения (offline или post-mortem).
- *Средства наблюдения и управления* – доступные средства контроля поведения реализации в ходе ее реальной работы или симуляции, используемые обычно при динамическом анализе, например, внешний интерфейс, обратный интерфейс (набор вызываемых ей

операций в других компонентах и системах), различные журналы (трассы), снимки памяти и др.

- *Дедуктивная верификация* или *дедуктивный анализ* (theorem proving) - проверка того, что набор утверждений, представляющий спецификацию, формально выводится из формального представления реализации и определенного набора гипотез о поведении окружения системы.
- Генератор верификационных задач (verification conditions generator) – инструмент, анализирующий верифицируемую программу и строящий набор утверждений, формально описывающих необходимые и достаточные условия выполнения некоторого заданного свойства программы. Используется в рамках дедуктивной верификации.
- Инструмент автоматизированного доказательства (prover, proof assistant) – инструмент для автоматизированного или полностью автоматического поиска доказательства задаваемых ему на входе утверждений.
- *Решатель* (solver) – инструмент для поиска решений системы утверждений. Такая система может задавать верифицируемое свойство или, наоборот, нарушение проверяемого свойства.
- *Проверка моделей* (model checking, software model checking) – проверка соответствия поведения автоматной формальной модели, описывающей реализацию, формальной модели (обычно декларативной), описывающей требования к системе. Как правило, модель реализации несколько упрощенно представляет проверяемую систему, что облегчает верификацию. В рамках классической проверки моделей построение модели реализации является дополнительной работой, выполняемой вручную, из-за чего модель может оказаться неадекватной целевой системе. Проверка программных моделей (software model checking) предполагает автоматическое построение модели реализации на основе кода системы, при этом используются эвристики, которые также могут привести к нарушению адекватности. По этой причине в некоторых методах в процесс построения модели включаются итеративные проверка адекватности и уточнение моделей, не прошедших проверку (например, в методе CEGAR [17,18] для этого выполняется попытка сгенерировать контрпример, демонстрирующий, что ошибочная ситуация достижима).
- *Предикатная абстракция* – использование в качестве модели состояния системы некоторого набора булевских характеристик (характеризующих как набор текущих внутренних данных, так и состояние выполнения программного кода, включая возможный параллелизм).

- Точность анализа, характеристики и аспекты качества верификации.
    - Важными характеристиками техники верификации являются возможность пропуска ошибок определенного типа или же гарантированность их выявления, а также возможность (или невозможность) ложных сообщений об ошибках определенного типа. Во многих случаях выявление абсолютно всех ошибок без ложных сообщений требует недоступных на практике трудозатрат и времени, приходится, в лучшем случае, выбирать между полнотой обнаружения ошибок и отсутствием ложных сообщений, минимизируя другой показатель.
    - При проведении верификации важно выявить и зафиксировать полный и точный набор предположений, в которых она выполняется. Неадекватность этих условий реальности (также как и наличие неявных предположений) может привести к необнаружению ошибок.
    - В случае сложных систем полная верификация может оказаться недостижимой. В этой ситуации возможно использование метрик полноты верификации, связанных с долей проверенных утверждений по отношению ко всем и проверенной части реализации по отношению ко всему объему ее кода. Для адекватного учета полноты также важно, чтобы наиболее существенные ограничения и элементы кода проходили верификацию как можно раньше.
    - Технические результаты верификации также могут задаваться по-разному, в зависимости от различных предположений об окружении и применяемых методов. Это могут быть либо подозрительные/опасные ситуации, возможные дефекты, которые нужно еще подвергнуть дополнительному анализу на предмет присутствия в них ошибки, либо «настоящие ошибки», для которых в результате верификации становится известен способ демонстрации нарушения требований при работе системы.

4. Возможные сценарии интеграции верификационных техник

В данном исследовании мы основываемся на опыте, полученном в многочисленных проектах по верификации программного и аппаратного обеспечения за последние 20 лет работы в отделе Технологий программирования ИСП РАН. Коротко перечислим эти проекты. Начнем с тех, которые базировались на использовании одного вида формальных методов, а затем перейдем к проектам, где проводились эксперименты с синтезом различных подходов.

Первые проекты полностью опирались на тестирование на основе формальных моделей (model based testing, MBT). В них были использованы следующие формализмы для представления моделей [19].

- Программные контракты интерфейсов (interface contracts) в форме ограничений - пред- и постусловия операций или событий и инварианты типов данных.
- Расширенные автоматы (extended finite state machines, EFSM) в так называемой неявной форме для описания моделей тестовых сценариев.
- Расширенные грамматики для моделирования деревьев разбора, абстрактных синтаксических деревьев с ограничениями, а также для представления частей документов с семантическими зависимостями.

В этих проектах было проведено промышленное тестирование нескольких операционных систем, включая ОС Linux и отечественные операционные системы реального времени, оптимизирующего компилятора компании Intel, некоторых компонентов информационной инфраструктуры компании крупного телекоммуникационного оператора, моделей отечественных микропроцессоров.

В рамках данных проектов решался следующий типовой набор задач, связанных с построением соответствующих компонентов верифицирующей (в данном случае тестовой) системы.

- Построение спецификаций программного контракта.
- Построение отображения реализационных событий и данных в модельные (так называемые адаптеры).
- Построение модели тестового сценария, задающего логику генерации последовательности тестовых воздействий.
- Генерация тестовых данных
- Построение тестовых оракулов – компонентов, которые на основе поведения тестируемой системы дают оценку (вердикт) его корректности.
- Определение метрик полноты тестирования и построение системы сбора и анализа достигнутого тестового покрытия.

Следующая группа проектов использовала технику проверки программных моделей (software model checking). Верифицируемой системой в данном случае являлись компоненты ядра ОС Linux, в первую очередь, драйвера устройств [20]. В рамках этих проектов типовой состав работ был таков.

- Формализация правил взаимодействия компонентов программной системы (их можно рассматривать как некоторый вид программных контрактов) [21].
- Генерация модели окружения, описывающей возможные сценарии вызовов функций проверяемой системы [22].
- Генерация верификационных задач.

- Решение верификационных задач.
- Сбор информации о проанализированных путях в графе потока управления проверяемой системы.

Еще одна группа проектов связана с использованием дедуктивной верификации [23]. Проверяемой системой в данном случае является система защиты информации (СЗИ) ОС AstraLinux. В рамках этой группы проектов проводилась верификация модели требований к СЗИ, в качестве такой модели была использована мандатная сущностно-ролевая модель доступа и потоков данных (МРОСЛ ДП) [24], а затем нужно было провести верификацию кода модуля безопасности ядра ОС Linux (Linux Security Module, LSM).

В рамках этих проектов решались следующие задачи.

- Построение формальных спецификаций операций и инвариантов типов данных для интерфейсов модели и модуля LSM в форме программных контрактов.
- Формализация инвариантов информационной безопасности, определяющих требования обеспечения защищенности проверяемой системы.
- Собственно верификация модели и модуля LSM.

Заметим, что здесь работы с моделью и модулем LSM рассматриваются совместно, так как между этими объектами верификации много концептуально близкого – они описывают механизмы обеспечения контроля доступа к информации. Однако модель и модуль реализованы на разных языках (Event-B и C) и верифицируются разными инструментами (Rodin и Frama-C/Jessie/Why3). Кроме того, в части интерфейсов модель и модуль не совпадают.

Помимо различия в используемых языках и инструментах, задачи верификации модели СЗИ и модуля LSM существенно отличаются друг от друга тем, что модель представляет на соответствующем уровне абстракции целевую систему в целом, а LSM лишь часть ядра ОС, ответственную за выполнение проверок при обращениях к ней других компонентов ядра. Это означает необходимость, помимо решения других задач, описать формально предположения о поведении окружении модуля (т.е., ядра и всех приложений в целом, по отношению к вызовам операций модуля), без таких предположений невозможно верифицировать сам модуль. Отметим, что корректность таких предположений тоже надо проверять, то есть верифицировать.

В выделенных списках работ в каждой из трех рассмотренных групп можно отметить работы, нацеленные на решение одних и тех же задач, хотя пока, в силу технических различий между инструментами верификации, их приходится решать по-разному. Мы разместили схожие задачи в приведенной ниже таблице. Всего получилось пять групп.

Группы задач	Тестирование на основе моделей	Проверка моделей	Дедуктивная верификация
Определение программного контракта	Построение спецификаций программного контракта (тесно связано с построением тестовых оракулов)	Формализация правил взаимодействия компонентов программной системы	Построение формальных спецификаций операций и типов данных в форме программных контрактов
Построение модели окружения	Построение модели тестового сценария	Генерация модели окружения	Формальное описание предположений о поведении окружения проверяемого модуля
Построения пути, демонстрирующего ошибку	Выполнение обхода сценарного автомата, генерация тестовых данных для всех воздействий	Генерация и решение верификационных задач	(эта задача не ставилась)
Увязка уровней абстракции	Построение отображения реализационных событий и данных в модельные	(эта задача не ставилась)	(эта задача не ставилась)
Оценка полноты верификации	Определение метрик полноты тестирования и построение системы сбора и анализа достигнутого тестового покрытия	Сбор информации о проанализированных путях в графе потока управления системы	(эта задача не ставилась)

Таблица 1. Соответствие между задачами в разных методах верификации.

На основании сказанного можно выделить узлы, в которых различные методы и реализующие их инструменты имеют возможные точки сопряжения. Примерами таких точек могут быть следующие артефакты:

- Контрактные спецификации поведения.
- Модель окружения проверяемой системы, состоящего из компонентов, которые вызывают операции системы, вызываются ею или взаимодействуют с ней каким-либо другим способом.
- Метрики полноты верификации.

5. Заключение

В этой статье мы формулируем задачу создания методов верификации ядер операционных систем, дающих достаточно высокие гарантии надежности и пригодных для практического применения. Результирующие методы, скорее всего, будут интегрировать элементы различных техник верификации, и, будучи применимы ко всей системе в целом, позволят тщательнее проверять наиболее важные ее функции и компоненты, используя при этом самые строгие формальные подходы.

Проанализировав имеющийся в ИСП РАН опыт проведения верификации операционных систем с помощью различных методов – тестирования на основе моделей, проверки программных моделей, дедуктивного анализа, мы выделили набор артефактов разработки, которые могут быть наиболее удобными точками интеграции различных верификационных техник. Развитие проведенного исследования будет связано с созданием техники верификации, интегрирующей статический и динамический анализ при помощи использования выявленных артефактов – модели поведения, модели окружения и модели ситуаций.

Литература

[1]. Proceedings of the 1st International Conference on Integrated Formal Methods. Edited by K. Araki, A. Galloway, K. Taguchi, York, 28-29 June 1999. Springer-Verlag, 1999. ISBN:1-85233-107-0.

[2]. L. De Moura, N. Bjørner. Z3: an Efficient SMT Solver. Proceedings of TACAS'2008. LNCS 4963:337-340, Springer-Verlag, 2008.

[3]. http://github.com/Z3Prover/z3

[4]. C. Barret, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli. CVC4. Proceedings of CAV'2011, LNCS 6806:171-177, Springer, 2011.

[5]. http://cvc4.cs.nyuedu/web

[6]. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson. Model-based Testing of Object-oriented Reactive Systems with SpecExplorer. Proceedings of FORTEST'2008, LNCS 4949:39-76, Springer-Verlag, 2008.

[7]. N. Tillmann, J. de Halleux. Pex – White Box Test Generation for .NET. Proceedings of TAP'2008, LNCS 4966:134-153, Springer-Verlag, 2008.

[8]. P. Godefroid. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. Proceedings of 2-nd International Workshop on Random Testing, p. 1-1, ACM, 2007.

[9]. T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Software Verification with BLAST. Proceedings of SPIN'2003, Model Checking Software, LNCS 2648:235-239, Springer-Verlag, 2003.

[10]. T. Ball, B. Cook, V. Levin, S. K. rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Proceedings of IFM'2004, LNCS 2999:1-20, Springer-Verlag, 2004.

[11]. D. Beyer, M. E. Keremoglu. CPAchecker: a Tool for Configurable Software Verification. Proceedings of CAV'2011, LNCS 6806:184-190, Springer, 2011.

[12]. G. Canet, P. Cuoq, B. Monate. A Value Analysis for C Programs. Proceedings of SCAM'2009, p. 123-124, IEEE, 2009.

[13]. J.-R. Abrial, M. Butler, S. Hallerstede, L. Voisin. An Open Extensible Tool Environment for Event-B. Proceedings of ICFEM'2006, Formal Methods and Software Engineering, LNCS 4260:588-605, Springer-Verlag, 2006.
[14]. А. К. Петренко. Унификация в автоматизации тестирования. Позиция UniTESK. Труды Института системного программирования РАН 14(1):7-22, 2008.
[15]. В. В. Кулямин. Интеграция методов верификации программных систем. Программирование, 35(4):41-55, 2009.
[16]. В. В. Кулямин. Методы верификации программного обеспечения. Статья-победитель конкурса обзорно-аналитических статей по направлению «Информационно-телекоммуникационные системы», 2008 (http://www.ispras.ru/publications/2008/methods_of_software_verification/).
[17]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Proceedings of CAV'2000, LNCS 1855:154-169, Springer-Verlag, 2000.
[18]. М. У. Мандрыкин, В. С. Мутилин, А. В. Хорошилов. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды Института системного программирования РАН 24: 219-292, 2013. DOI: 10.15514/ISPRAS-2013-24-12
[19]. В. В. Кулямин, А. К. Петренко. Развитие подхода к разработке тестов UniTESK. Труды Института системного программирования РАН 26(1):9-26, 2014. DOI: 10.15514/ISPRAS-2014-26(1)-1
[20]. И.С. Захаров, М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.К. Петренко, А.В. Хорошилов. "Конфигурируемая система статической верификации модулей ядра операционных систем". Труды Института Системного Программирования РАН, Том 26(2):5-42, 2014. DOI: 10.15514/ISPRAS-2014-26(2)-1
[21]. Е. М. Новиков. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени к.ф.-м.н., Москва, 2013.
[22]. И.С. Захаров, В.С. Мутилин, А.В. Хорошилов. "Моделирование окружения с использованием шаблонов для статической верификации модулей ядра Linux". // Программирование, Том 41, 2015, №3, сс. 3-19.
[23]. P. N. Devyanin, A. V. Khoroshilov, V. V. Kuliamin, A. K. Petrenko, and I. V. Shchepetkov. Using Refinement in Formal Development of OS Security Model. Proceedings of PSI'2015.
[24]. П. Н. Девянин. Ролевая ДП-модель управления доступом и информационными потоками в операционных системах семейства Linux. ПДМ, 2012, № 1, 69–90.

Integration Points of Operating System Verification Techniques[2]

[1,2,3,4]A. K. Petrenko <petrenko@ispras.ru>
[1,2,4]V. V. Kuliamin <kuliamin@ispras.ru>
[1,2,3,4]A. V. Khoroshilov <khoroshilov@ispras.ru>
[1]Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.

[2] Supported by RFBR grant # 14-01-00484.

[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Moscow Institute of Physics and Technology (State University)*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] *National Research University Higher School of Economics (HSE)*
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. In this work the problem of high quality verification techniques applicable for operating systems is formulated. A perspective approach to solve this problem is integration of various verification methods. The solution technique can be considered successful if it allows to check the whole operating system and to verify in more accurate way the most important functions and components of the system, using more strict and formal methods for it. Based on the ISP RAS experience in operating system verification projects conducted using various verification techniques we determine development artifacts, that can be suitable integration point candidates for integration of formal specification based static and dynamic verification techniques for operating systems.
The article proposes to define common (shared) artifacts based on the consideration of typical verification tasks. Typical problems encountered in the use of different techniques of verification, provide the basis for integration techniques and verification processes. These types of problems are: the definition of a software contracts of modules (functions); the construction of the model environment; building the path, demonstrating the error; bringing levels of abstraction and assessment of the completeness of the verification. To the greatest extent on the role of artifacts representing the points of integration claim: software contract specifications, environment models and measurement of verification completeness.

Keywords. Theorem proving, software model checking, model-based testing, operating system, integration of verification methods, software contracts.

DOI: 10.15514/ISPRAS-2015-27(5)-10

For citation: Petrenko A. K., Kuliamin V. V., Khoroshilov A. V. Integration Points of Operating System Verification Techniques. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 175-190 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-10.

References

[1]. Proceedings of the 1st International Conference on Integrated Formal Methods. Edited by K. Araki, A. Galloway, K. Taguchi, York, 28-29 June 1999. Springer-Verlag, 1999. ISBN:1-85233-107-0.
[2]. L. De Moura, N. Bjørner. Z3: an Efficient SMT Solver. Proceedings of TACAS'2008. LNCS 4963:337-340, Springer-Verlag, 2008.
[3]. http://github.com/Z3Prover/z3
[4]. C. Barret, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli. CVC4. Proceedings of CAV'2011, LNCS 6806:171-177, Springer, 2011.
[5]. http://cvc4.cs.nyuedu/web
[6]. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson. Model-based Testing of Object-oriented Reactive Systems with SpecExplorer. Proceedings of FORTEST'2008, LNCS 4949:39-76, Springer-Verlag, 2008.

[7]. N. Tillmann, J. de Halleux. Pex – White Box Test Generation for .NET. Proceedings of TAP'2008, LNCS 4966:134-153, Springer-Verlag, 2008.

[8]. P. Godefroid. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. Proceedings of 2-nd International Workshop on Random Testing, p. 1-1, ACM, 2007.

[9]. T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Software Verification with BLAST. Proceedings of SPIN'2003, Model Checking Software, LNCS 2648:235-239, Springer-Verlag, 2003.

[10]. T. Ball, B. Cook, V. Levin, S. K. rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Proceedings of IFM'2004, LNCS 2999:1-20, Springer-Verlag, 2004.

[11]. D. Beyer, M. E. Keremoglu. CPAchecker: a Tool for Configurable Software Verification. Proceedings of CAV'2011, LNCS 6806:184-190, Springer, 2011.

[12]. G. Canet, P. Cuoq, B. Monate. A Value Analysis for C Programs. Proceedings of SCAM'2009, p. 123-124, IEEE, 2009.

[13]. J.-R. Abrial, M. Butler, S. Hallerstede, L. Voisin. An Open Extensible Tool Environment for Event-B. Proceedings of ICFEM'2006, Formal Methods and Software Engineering, LNCS 4260:588-605, Springer-Verlag, 2006.

[14]. A. K. Petrenko. Unifikatsia v avtomotizatsii testirovania. Positsia UniTESK [Testing Unification and Automation. UniTESK Viewpoint]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 14(1):7-22, 2008 (in Russian).

[15]. V. V. Kuliamin. Integration of Software Verification Methods. Programming and Computer Software, 35(4):41-55, 2009. DOI 10.1134/S0361768809040057

[16]. V. V. Kuliamin. Software Verification Methods. Paper on Analytical Survey Contest on Information and Communication Systems, 2008 (http://www.ispras.ru/publications/2008/methods_of_software_verification/, in Russian).

[17]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Proceedings of CAV'2000, LNCS 1855:154-169, Springer-Verlag, 2000.

[18]. Khoroshilov A.V., Mandrykin M. U., MutilinV. S. Vvedenie v metod CEGAR — utochnenie abstrakcii po kontrprimeram [Introduction to CEGAR — Counter-Example Guided Abstraction Refinement]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian). DOI: 10.15514/ISPRAS-2013-24-12

[19]. V. V. Kuliamin, A. K. Petrenko. Razvitie Podkhoda k Razrabotke Testov UniTESK [Evolution of UniTESK Test Development Approach]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26(1):9-26, 2014 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-1

[20]. A. V. Khoroshilov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and I. S. Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software 41(1):49-64, 2015. DOI 10.1134/S0361768815010065

[21]. E. M. Novikov. Development of Software Contracts for Verification of Linux Operating System Kernel Modules. PhD Thesis, Moscow, 2013 (in Russian).

[22]. A. V. Khoroshilov, V. S. Mutilin, I. S. Zakharov. Pattern-based environment modeling for static verification of Linux kernel modules. Programming and Computer Software 41(3):183-195, 2015. DOI 10.1134/S036176881503007X

[23]. P. N. Devyanin, A. V. Khoroshilov, V. V. Kuliamin, A. K. Petrenko, and I. V. Shchepetkov. Using Refinement in Formal Development of OS Security Model. Proceedings of PSI'2015.

[24]. P. N. Devyanin. The role DP-model of access and information flows control in operating systems of Linux sets. Prikladnaia i Diskretnaia Matematika (Applied and Discrete Mathematics), 2012, № 1, 69–90 (in Russian).

Approximating Chromatic Sum Coloring of Bipartite Graphs in Expected Polynomial Time

[1] *A.S. Asratian <arasr@mai.liu.se>*
[2] *N.N. Kuzyurin <nnkuz@ispras.ru>*
[1] *Linkopings Universitet, Department of Mathematics, Sweden, 581 83,*
[2] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., 109004, Moscow, Russia*

Abstract. It is known that if complexity class P is not equal to NP the sum coloring problem cannot be approximated within 1+epsilon for some positive constant epsilon.
We consider finite, undirected graphs without loops and multiple edges. Let G=(V,E) be a graph. By a coloring of G we mean a mapping c of V to the numbers 1,2, ..., |V| . A coloring c is proper if c(v) is not equal to c(u) whenever the vertices u and v are adjacent.
Let S(G,c) is the sum_of c(v) over all vertices v. By a chromatic sum of G we mean the number S(G)=min S(G,c) where minimum is taken over all proper colorings c of G.
The problem of finding S(G) is called the sum coloring problem.
It was shown that the sum coloring problem is NP-complete.
A graph G is called bipartite if the set of vertices of G can be partitioned into two non-empty sets V1 and V2 such that every edge of G has one end in each of the sets.
For a number b, we say that an algorithm A approximates the chromatic sum within factor b over graphs on n vertices, if for every such graph G the algorithm A outputs a proper coloring c, such that S(G,c) is not greater than b S(G).
It is known that there exists 27/26-approximation polynomial algorithm for the chromatic SUM COLORING PROBLEM on any bipartite graph. On the other side, it was shown that here exists epsilon>0, such that there is no (1+epsilon)-approximation polynomial algorithm for the sum coloring problem on bipartite graphs, unless P is not equal to NP.
In this paper we consider the problem of developing an (1+epsilon)-approximation algorithm for the sum coloring of bipartite graphs which is polynomial in the average case for arbitrary small epsilon. We prove the existence of such algorithm.

Keywords: sum coloring problem, bipartite graphs, expected polynomial time

DOI: 10.15514/ISPRAS-2015-27(5)-11

For citation: Asratian A.S., Kuzyurin N.N. Approximating Chromatic Sum Coloring of Bipartite Graphs in Expected Polynomial Time. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 191-198 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-11.

1. Introduction

Let $G = (V_1, V_2, E)$ be a bipartite graph with $n+m$ vertices such that $|V_1| = m$, $|V_2| = n$, $m \le n$. By a coloring we mean a mapping:

$$c : V_1 \cup V_2 \to \{1,2,\ldots,n+m\}.$$

A coloring is proper if $c(v) \ne c(u)$ whenever $(u,v) \in E$.

Let $S(G,c) = \sum_{v \in V} c(v)$. By a chromatic sum we mean $S(G) = \min_c S(G,c)$ where minimum is taken over all proper colorings of G. The problem of finding $S(G)$ is called the SUM COLORING PROBLEM.

The notion of chromatic sum was first introduced in [6] where it was shown that the SUM COLORING PROBLEM is NP-complete on arbitrary graphs. A few b-approximation algorithms which find a coloring c with $S(G,c) \le b \cdot S(G)$ were presented. In [7] a $10/9$-approximation polynomial algorithm for the SUM COLORING PROBLEM on any bipartite graph was described. This result was improved in [8] where an $27/26$-approximation algorithm for the same problem was constructed. On the other side, in [7] the authors have shown that there exists $\varepsilon > 0$, such that there is no $(1+\varepsilon)$-approximation polynomial algorithm for the SUM COLORING PROBLEM on bipartite graphs, unless $P = NP$.

In this paper we present for any positive ε an $(1+\varepsilon)$-approximation algorithm for this problem with expected polynomial time. The probabilistic distribution is uniform over all bipartite graphs with N vertices, $N = n+m$, $m \le n$. Note that the first example of approximation algorithm with expected polynomial time guaranteeing approximation ratio better than inapproximability threshold in the worst case was presented in [9]. Probabilistic analysis of algorithms for random graphs is the focus of much research now [1-5, 9].

2. Approximation scheme with expected polynomial time

Let $N = n+m$. We consider now a straightforward approach testing all possible colorings of G and choosing the one with the best possible color sum.

Algorithm 1. Test all possible vertex colorings of a bipartite graph and choose a proper coloring with minimum color sum.

Lemma 1. The time complexity of Algorithm 1 is $O(N^N) = O((2n)^{2n})$.

Let δ be a positive number, $0 < \delta < 1$ and

$$V_1' = \{v \in V_1 : (1-\delta)\frac{m}{2} \le \deg v \le (1+\delta)\frac{m}{2}\},$$

$$V_2' = \{v \in V_2 : (1-\delta)\frac{n}{2} \le deg\ v \le (1+\delta)\frac{n}{2}\},$$

$$\overline{V}'_1 = V_1 \setminus V_1',$$

$$\overline{V}'_2 = V_2 \setminus V_2'.$$

2.1 Algorithm VERTEX-COLOR.

Input: A bipartite graph $G = (V_1, V_2, E)$ such that $|V_1| = m$, $|V_2| = n$, $m \le n$, and a parameter $\varepsilon > 0$.

Output: A proper coloring c of G such that $S(G) \le S(G,c) \le (1+\varepsilon)S(G)$.

1. If $\varepsilon \le \max\{40n^{-0.5}, n^{-0.2}, 50n^{-0.3}\}$ then goto 7.

2. If $m \le n^{0.8}$ then goto 7.

3. Set $\delta = \min\{\frac{1}{50}, \frac{\varepsilon}{50} - n^{-0.3}\}$.

4. Count the number $t_1 = |\overline{V}'_1|$, and $t_2 = |\overline{V}'_2|$.

5. If $t_1 > \sqrt{n}$ or $t_2 > n^{0.4}$ then goto 7.

6. Color V_2 by color 1 and color V_1 by color 2 and STOP.

7. Run Algorithm 1 and STOP.

Theorem 1. For any fixed $\varepsilon > 0$ Algorithm **VERTEX-COLOR** finds a proper coloring within $1+\varepsilon$ of the optimum color sum in expected polynomial time.

Proof. Note that at step 2 and step 5 of the algorithm we get $S(G,c) = n + 2m$ using very simple coloring strategy. The main idea of the proof is to extract sufficiently large almost regular bipartite subgraph $G' = (V_1', V_2', E')$ of G such that for any $v \in V_1'$ $(1-\delta')r \le deg\ v \le (1+\delta')r$, and for any $v \in V_2'$ $(1-\delta')k \le deg\ v \le (1+\delta')k$. Such an almost regular subgraph can guarantee a tight lower bound on $S(G)$ close to the upper bound $S(G) \le n + 2m$. The main difficulty is to estimate the probability that the size of such subgraph is large enough.

We use m' and n' for denoting $|V_1'|$ and $|V_2'|$ respectively.

Lemma 2. For any $0 < \delta' < \frac{1}{2}$ and an induced subgraph $G' = (V_1', V_2', E')$ as above

$$n' + 2m' - 10\delta' m' \le S(G') \le n' + 2m'.$$

Proof of Lemma 2. The upper bound is evident (we color V_1' by color 2 and color V_2' by color 1). To prove the lower bound we use the folowing inequalities

$$(1+\delta')r\sum_{v\in V_1'} c(v) + (1+\delta')k\sum_{v\in V_2'} c(v) \ge \sum_{e=(u,v)\in E'} (c(u)+c(v)) \ge$$

$$\ge 3|E'| \ge 3r(1-\delta')m'.$$

This implies the inequality

$$\sum_{v\in V_1'} c(v) + \frac{k}{r}\sum_{v\in V_2'} c(v) \ge 3m'\frac{1-\delta'}{1+\delta} \ge 3m'(1-2\delta').$$

Adding to both parts of the inequality $(1-\frac{k}{r})\sum_{v\in V_2'} c(v)$ and taking into account that $c(v) \ge 1$ for any v we obtain that for any proper coloring c of G'

$$S(G',c) = \sum_{v\in V_1'} c(v) + \sum_{v\in V_2'} c(v) \ge 3m' - 6\delta' m' + (1-\frac{k}{r})\sum_{v\in V_2'} c(v) \ge$$

$$\ge 2m' + m' - 6\delta' m' + (1-\frac{k}{r})n' = 2m' + n' + m' - 6\delta' m' - \frac{k}{r}n' \ge$$

$$2m' + n' + m' - 6\delta' m' - m' - 4\delta' m' = n' + 2m' - 10\delta' m'.$$

Here we used the inequality $m'r(1+\delta') \ge n'k(1-\delta')$ which for any $0 < \delta' < \frac{1}{2}$ implies

$$\frac{k}{r}n' \le m'\frac{1+\delta'}{1-\delta'} = m'(1+\frac{2\delta'}{1-\delta'}) \le m'(1+4\delta').$$

The proof of Lemma 2 is complete.

Now we estimate the size of G'.

Lemma 3. There is $c > 0$ depending on δ such that

$$Pr\{|\overline{V}'_2| \ge \sqrt{n}\} \le \exp\{\sqrt{n}\log n - cn^{3/2}\}.$$

$$Pr\{|\overline{V}'_1| \ge n^{0.4}\} \le \exp\{n^{0.4}\log n - cn^{1.2}\}.$$

Proof. We need the following lemma.

Lemma ([5]). Let $x_1,\ldots,x_n$ be independent random variables such that x_i takes two values: 0 and 1, and $Pr\{x_i=1\}=p$, $Pr\{x_i=0\}=1-p$.

Let $X=\sum_{i=1}^{n}x_i$ and $EX=np$. Then the following inequalities hold:

for any $\delta>0$

$$Pr\{X-EX<-\delta EX\}\le\exp\{-(\delta^2/2)EX\},$$

for any $0<\delta<1$

$$Pr\{X-EX>\delta EX\}\le\exp\{-(\delta^2/3)EX\}.$$

Using this Lemma we have for $v\in V_1'$:

$$Pr\{d(v)\le n(1-\delta)2\}\le\exp\{-(\delta^2/2)n/2\},$$
$$Pr\{d(v)\ge n(1+\delta)2\}\le\exp\{-(\delta^2/3)n/2\}.$$

We give the proof for $\overline{V'}_2$. The proof for $\overline{V'}_1$ is similar.
To do this we estimate the following probability:

$$Pr\{|\overline{V'}_2|\ge k\}\le\binom{n}{k}(Pr\{fixed\ k_1\ vertices\ v\ in\ \overline{V'}_2\ have\ d(v)\le(1-\delta)n/2\}\cdot$$
$$Pr\{fixed\ k_2\ vertices\ v\ in\ \overline{V'}_2\ have\ d(v)\ge(1+\delta)n/2\}),$$

where $k=k_1+k_2$. Using the Lemma and taking into account independence of the corresponding events we have

$$Pr\{fixed\ k_1\ vertices\ v\ in\ \overline{V'}_2\ have\ d(v)\le(1-\delta)n/2\}\le$$
$$\exp\{-(\delta^2/3)k_1m/2\}\le\exp\{-cmk_1\},$$
$$Pr\{fixed\ k_2\ vertices\ v\ in\ \overline{V'}_2\ have\ d(v)\ge(1+\delta)m/2\}\le$$
$$\exp\{-(\delta^2/3)k_2m/2\}\le\exp\{-cmk_2\},$$

where c depends on δ.

Letting in the last inequalities $k=n^{0.4}$ we obtain

$$Pr\{|\overline{V'}_2|\ge k\}\le\binom{n}{k}\exp\{-cm(k_1+k_2)\}\le$$
$$\exp\{k\log n-cmk\}\le\exp\{n^{0.4}\log n-cn^{1.2}\}.$$

To finish the proof of Theorem 1 it is necessary to estimate the approximation ratio of the algorithm **VERTEX-COLOR** and its expected running time.

2.2 Approximation ratio

If the algorithm terminates at step 2 then we use the inequality

$$n+m \le S(G) \le n+2m.$$

This gives that for the proper coloring c obtained at step 2

$$S(G,c) = n+2m \le S(G)\cdot\frac{n+2m}{n+m} = S(G)(1+\frac{m}{n+m}) \le$$

$$\le S(G)(1+n^{-0.2}) \le S(G)(1+\varepsilon),$$

because $\varepsilon > n^{-0.2}$ (in the opposite case the algorithm always finds an optimal solution at step 7).

Because at step 7 we always find an optimal solution it is sufficient to estimate approximation ratio for step 6. To do this we use Lemma 2. If the algorithm terminates at step 6 then $t_1 \le \sqrt{n}$ and $t_2 \le n^{0.4}$. Thus we have $n' = n - t_1 \ge n - \sqrt{n}$, $m' = m - t_2 \ge m - \sqrt{n}$. Because the degree of a vertex in G' can decrease by at most $\sqrt{n}$ we can estimate δ' as follows:

$$degv \ge (1-\delta)\frac{m}{2} - \sqrt{n} = (1-\delta')\frac{m}{2},$$

which implies $\delta' = \delta + \frac{2\sqrt{n}}{m}$.

By Lemma 2

$$n+2m-10\delta' m - t_1 - t_2 \le S(G') \le S(G) \le n+2m.$$

This implies the inequality

$$n+2m-10\delta m - 23\sqrt{n} \le S(G) \le n+2m,$$

and then the inequality

$$(n+2m)(1-10\delta - \frac{25}{\sqrt{n}}) \le S(G) \le n+2m.$$

Thus, for the coloring c that the algorithm outputs at step 6 the following inequality holds

$$S(G,c) \le S(G)(1-10\delta - \frac{25}{\sqrt{n}})^{-1}.$$

Now we use the following technical lemma.

Lemma. Let $0 < \delta < \min\{\frac{1}{50}, \frac{\varepsilon}{50}\}$, $\varepsilon > 40n^{-0.5}$. Then

$$(1-10\delta-\frac{25}{\sqrt{n}})^{-1} \le 1+\varepsilon.$$

Proof. We have

$$(1-10\delta-\frac{25}{\sqrt{n}})\cdot(1+\varepsilon) \ge 1$$

This is equivalent to

$$\varepsilon-10\delta(1+\varepsilon)-\frac{25}{\sqrt{n}}(1+\varepsilon) =$$

$$\varepsilon-(1+\varepsilon)(10\delta+\frac{25}{\sqrt{n}}) \ge 0.$$

This implies

$$\frac{\varepsilon}{1+\varepsilon} \ge 10\delta+\frac{25}{\sqrt{n}}.$$

Taking into account the inequality $\delta < \varepsilon/50$ we have

$$n \ge \frac{1200}{\varepsilon^2}.$$

This inequality follows from the condition of the Lemma: $\varepsilon > 40n^{-0.5}$.

2.3 Expected running time

Step 4 is performed in quadratic (in n) time. By Lemmas 1 and 3 the expected time of step 7 is at most

$$O((2n)^{2n})\exp\{\sqrt{n}\log n - cn^{1.2}\} \le$$

$$c\exp\{2n\log 2n + \sqrt{n}\log n - cn^{1.2}\} \to 0$$

as n tends to infinity.

References

[1]. B. Bollobas, Random graphs, second ed., Cambridge Univ. Press, Cambridge, 2001.
[2]. B. Bollobas, The chromatic number of random graphs, Combinatorica, 1988, v. 8, c. 49-55.
[3]. A. Frieze,C. McDiarmid, Algorithmic theory of random graphs, Random Structures and Algorithms, Jan.-March 1997, v. 10, n. 1-2, c. 5-42.

[4]. L. Kucera, The greedy coloring is a bad probabilistic algorithm, J. of Algorithms, Dec. 1991, v. 12, n. 4, c. 674-684.
[5]. R. Motwani and P. Raghavan, Randomized algorithms, Cambridge Univ. Press, 1995.
[6]. E. Kubicka, A.J. Schwenk, An introduction to chromatic sums, Proc. of ACM Computer Science Conference, 1989, c. 39-45.
[7]. A. Bar-Noy, G. Kortsarz, Minimum color sum of bipartite graphs, J. of Algorithms, 1998, v. 28, c. 339-365.
[8]. K. Giaro, R. Janczewski, M. Kubale, M. Malafiejski, A 27/26-approximation algorithm for the chromatic sum coloring of bipartite graphs, Proc. APPROX 2002, LNCS v. 2462, c. 135-145.
[9]. M. Krivelevich , V.H. Vu, Approximating the independence number and the chromatic number in expected polynomial time, J. Combin. Optimization, 2002, v. 6, c. 143-155.

Приближенный алгоритм для хроматической раскраски двудольных графов за полиномиальное в среднем время

[1]А.С. Асратян <arasr@mai.liu.se>
[2]Н.Н. Кузюрин <nnkuz@ispras.ru>
[1] Линчёпингский университет, факультет математики, Швеция, 581 83
[2] Институт системного программирования РАН, Россия, 109004, Москва, ул. Солженицына, 25

Аннотация. Известно что если P≠NP то задача аппроксимации суммарной раскраски двудольных графов не может быть осуществлена в полиномиальное время с точностью $\mathbf{1}+\boldsymbol{\varepsilon}$ для некоторой константы $\boldsymbol{\varepsilon}$. Мы предлагаем для сколь угодно малого $\boldsymbol{\varepsilon}>0$ приближенный алгоритм для данной проблемы который работает за полиномиальное в среднем время.

Ключевые слова: проблема хроматической раскраски, двудольные графы, полиномиальное в среднем время

DOI: 10.15514/ISPRAS-2015-27(5)-11

Для цитирования: Асратян А.С., Кузюрин Н.Н. Приближенный алгоритм для хроматической раскраски двудольных графов за полиномиальное в среднем время. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 191-198. DOI: 10.15514/ISPRAS-2015-27(5)-11.

Литература

[1]. B. Bollobas, Random graphs, second ed., Cambridge Univ. Press, Cambridge, 2001.

[2]. B. Bollobas, The chromatic number of random graphs, Combinatorica, 1988, v. 8, с. 49-55.
[3]. [1] A. Frieze,C. McDiarmid, Algorithmic theory of random graphs, Random Structures and Algorithms, Jan.-March 1997, v. 10, n. 1-2, с. 5-42.
[4]. L. Kucera, The greedy coloring is a bad probabilistic algorithm, J. of Algorithms, Dec. 1991, v. 12, n. 4, с. 674-684.
[5]. R. Motwani and P. Raghavan, Randomized algorithms, Cambridge Univ. Press, 1995.
[6]. E. Kubicka, A.J. Schwenk, An introduction to chromatic sums, Proc. of ACM Computer Science Conference, 1989, с. 39-45.
[7]. A. Bar-Noy, G. Kortsarz, Minimum color sum of bipartite graphs, J. of Algorithms, 1998, v. 28, с. 339-365.
[8]. K. Giaro, R. Janczewski, M. Kubale, M. Malafiejski, A 27/26-approximation algorithm for the chromatic sum coloring of bipartite graphs, Proc. APPROX 2002, LNCS v. 2462, с. 135-145.
[9]. M. Krivelevich , V.H. Vu, Approximating the independence number and the chromatic number in expected polynomial time, J. Combin. Optimization, 2002, v. 6, с. 143-155.