ИСП Институт Системного Программирования
Российской Академии наук

Труды Института Системного Программирования

Том 18

Москва 2010

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

Труды Института Системного Программирования

Том 18

Под редакцией
академика РАН В.П. Иванникова

Москва 2010

УДК51

Труды Института системного программирования: Том 18.
/Под ред. В.П. Иванникова/ – М.: ИСП РАН, 2010. – 260 с.

В восемнадцатом томе Трудов Института системного программирования публикуются статьи, содержащие результаты работ, проводимых в Институте в 2009-2010 годах.

ISBN 978-0-543-57631-6

Содержание

П р е д и с л о в и е

В 18-м томе Трудов института системного программирования публикуются 9 статей сотрудников Института, посвященных различным аспектам тестирования программных и аппаратных средств на основе формальных спецификаций, методам виртуализации и моделирования.

В статье В.В. Кулямина «Архитектура среды тестирования на основе моделей, построенная на базе компонентных технологий» представлен подход к построению архитектуры инструментария для тестирования на основе моделей, использующего современные компонентные технологии. Одна из основных идей, лежащих в его основе – применение техник неинвазивной композиции, позволяющих с минимальными затратами интегрировать множество независимо разработанных компонентов в сложную систему и переконфигурировать ее, не изменяя кода компонентов. Также описывается прототипная реализации предложенного подхода на базе свободно доступных библиотек и пример ее использования для построения тестов.

Статья С.Г. Грошева «Технология создания гетерогенных трасс, их анализа и генерации из них отчётов» содержит описание архитектуры модульной системы трассировки и анализа трасс, поддерживающей трассировку сложноструктурированной информации, расширение видов трассируемой информации, изоляцию и выборку нужных данных при анализе полученных трасс и облегчающей связывание разнородных данных между собой. Рассматриваются методы связывания данных, в том числе разнородных по содержанию и по времени, а также дальнейшего преобразования их в разнообразные отчёты.

Статья Е.В. Корныхина «Метод зеркальной генерации ограничений для построения тестовых программ по тестовым шаблонам» относится к области системного функционального (core-level) тестирования микропроцессоров, более точно модулей управления памяти. В статье описывается метод построения тестов (тестовой программы) для нацеленной генерации. Такая генерация предполагает систематичное построение тестов специального вида. В конце приводятся результаты апробации реализации метода для тестирования модулей управления памяти микропроцессоров архитектуры MIPS64.

В статье П.М. Довгалюка, М.А. Климушенковой и А.М. Мухиной «Создание модулей поддержки архитектур для среды TrEx с помощью специализированного языка описания процессоров» рассматривается предложенный авторами подход к разработке модулей поддержки архитектур для среды TrEx на основе языка описания процессорных архитектур. Проанализированы достоинства и недостатки подхода по сравнению с уже существующими методами разработки.

Статья Д.Н. Воробьева и А.С. Камкина «Генерация тестовых программ для микропроцессоров на основе шаблонов конвейерных конфликтов» посвящена описанию методики автоматизированного построения тестовых программ для верификации управляющей логики микропроцессоров. Методика основана на формальной спецификации системы команд и описании шаблонов конфликтных ситуаций возможных в работе конвейера тестируемого микропроцессора. Использование формальных спецификаций позволяет автоматизировать разработку генератора тестовых программ и систематично протестировать управляющую логику.

В статье М.М. Чупилко «Автоматизация системного тестирования моделей аппаратуры на основе формальных спецификаций» обсуждается проблема системного тестирования взаимосвязанных модулей аппаратуры, когда их сложность уже не позволяет применять подходы к тестированию на уровне модулей. В работе приводится краткий анализ возможностей построения тестовой системы на основе использования формальных спецификаций, а также предлагается метод верификации, являющийся расширением модульного подхода, основанного на технологии UniTESK.

В статье А.С. Камкина «Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров» рассматриваются вопросы автоматизированного построения тестовых программ, предназначенных для функциональной верификации модулей обработки переходов микропроцессоров. Формулируются задачи, возникающие при создании таких программ, и предлагаются техники их автоматизированного решения. Предложенные методы можно использовать в промышленных генераторах тестовых программ.

Статья А.В. Никешина, Н.В. Пакулина и В.З. Шнитмана «Разработка тестового набора для верификации реализаций протокола безопасности IPsec v2» посвящена разработке тестового набора для проверки соответствий реализаций узлов Интернет спецификациям нового протокола безопасности IPsec v2. Для построения тестового набора использовалась технология автоматического тестирования UniTESK и программный пакет CTesK, реализующий эту технологию. В статье описаны метод формализации требований IPsec v2, процесс создания тестового набора, а также результаты тестирования существующих реализаций.

В статье И.Б. Бурдонова и А.С. Косачева «Тестирование конформности на основе соответствия состояний» обсуждаются проблемы тестирования соответствия (конформности) реализации требованиям спецификации. Идея безопасного тестирования, предложенная авторами для конформности, основанной на трассах наблюдений, распространяется на случай (слабой) симуляции – конформности, основанной на соответствии состояний реализации и спецификации. Строится теория безопасной симуляции и ее тестирования. Предлагаются общий алгоритм полного тестирования и его

модификация для практического применения, опирающаяся на некоторые ограничения на реализацию и спецификацию.

Статья П.Н. Яковенко «Прозрачный механизм удаленного обслуживания системных вызовов» содержит описание подхода к удаленному обслуживанию системных вызовов, не требующий модификации кода пользовательского приложения и операционной системы. Использование технологии аппаратной виртуализации для перехвата системных вызовов, чтения их параметров и записи результатов позволяет делегировать обслуживание перехваченных системных вызовов другой системе. Возможность предоставлять отдельным процессам контролируемый доступ к ресурсам, к которым сама операционная система доступа не имеет, позволяет эффективно решать некоторые задачи компьютерной безопасности.

Сборник завершает статья Д.А. Грушина и А.И. Поспелова «Система моделирования Grid: реализация и возможности применения», в которой описывается разработанная в ИСП РАН система моделирования распределенных вычислительных сред Grid. С помощью этой системы проведен анализ реальной вычислительной среды Sharcnet. На основе анализа были выявлены возможные способы существенного увеличения эффективности работы среды.

Академик РАН В.П. Иванников

Архитектура среды тестирования на основе моделей, построенная на базе компонентных технологий

Кулямин В. В.
kuliamin@ispras.ru

Аннотация. В статье представлен подход к построению архитектуры инструментария для тестирования на основе моделей, использующего современные компонентные технологии. Одна из основных идей, лежащих в его основе — применение техник неинвазивной композиции, позволяющих с минимальными затратами интегрировать множество независимо разработанных компонентов в сложную систему и переконфигурировать ее, не изменяя кода компонентов. Также описывается прототипная реализации предложенного подхода на базе свободно доступных библиотек и пример ее использования для построения тестов.

1. Введение

Увеличение разнообразия и количества задач, возлагаемых в современном мире на вычислительные системы, приводит к постоянному росту их сложности и усложнению всех видов деятельности, связанных с их построением и сопровождением. На решение проблем, связанных с этим ростом сложности, нацелены активно развивающиеся в последние десятилетия *компонентные технологии* [1,2] построения как программных, так и аппаратных составляющих вычислительных систем.

При их применении системы выстраиваются из относительно независимых *компонентов*, каждый из которых решает узкий набор задач и взаимодействует с окружением только через четко определенный интерфейс, т.е. является *модулем*. Помимо этого, программные компонентные технологии обеспечивают совместную работу и облегченную интеграцию компонентов без необходимости специальных операций сборки и связывания (точнее, эти операции выполняются автоматически и часто на лету, без прерывания работы системы), только за счет помещения готовых компонентов в бинарном виде в рамки технологической инфраструктуры системы. В результате компоненты могут создаваться и поддерживаться совершенно независимыми организациями и разработчиками, что дает возможность построения весьма сложных систем без существенных расходов на поддержку их интегрированной разработки и сопровождения.

Компонентные технологии позволили значительно повысить сложность создаваемых продуктов и изделий, однако их развитие и распространение пока не сопровождается соответствующим прогрессом в технологиях контроля качества программного обеспечения (ПО). В связи с этим проблемы обеспечения качества компонентных систем, включающие обеспечение и проверку надежности и корректности их работы, совместимости и защищенности, не получают адекватных технологических решений, а лишь усугубляются с возрастанием сложности систем и ответственности решаемых ими задач.

Источником таких проблем служит обычно чрезвычайно высокая трудоемкость проверки корректности поведения систем при нелинейно растущем с увеличением числа компонентов количестве возможных сценариев их взаимодействия и сложных, часто неявных связей и зависимостей между ними. В свою очередь, эти трудности имеют следующие причины.

- Сложившаяся практика неполного и неточного описания интерфейсов компонентов. Эти неточности и неполнота и приводят к возникновению множества трудноуловимых зависимостей между компонентами, проявляющихся при замене компонентов на их новые версии или на аналоги от других поставщиков. Дело в том, что полный интерфейс модуля, как он понимался в работах Парнаса [3], одного из основоположников модульного подхода к разработке ПО, должен задавать не только его синтаксическую структуру (имена реализуемых операций, типы их параметров и результатов, и соответствующие структуры данных), но и семантику (правила использования операций и ожидаемое в ответ поведение модуля). На практике же обычно ограничиваются только фиксацией синтаксиса, семантика описывается, да и то, чаще всего не полностью и не вполне однозначно, только для интерфейсов, входящих в системные библиотеки и имеющих достаточно долгую историю использования. С такими интерфейсами обычно связаны высокие риски и накладные расходы в случае несовместимости или некорректного взаимодействия компонентов, использующих их. Однако «менее рискованных» интерфейсов, для которых семантические требования не описываются совсем или фиксируются в виде одной фразы, указывающей их основную функциональность в нормальной ситуации, гораздо больше, и меньшие расходы, связанные с отдельными ошибками, в сумме дают очень значительную величину. В 2002 году годовые потери одной только экономики США от многочисленных ошибок в ПО были оценены в 60 миллиардов долларов [4].

- Трудности интеграции в технологические процессы разработки ПО средств верификации и тестирования на соответствие требованиям.

Такие средства по большей части представлены в виде «монолитных» инструментов, реализующих большое число разнообразных функций, от оформления требований до создания отчетов о проведенном тестировании. Причем набор деятельностей, выполняемых в ходе контроля качества, связи между ними, а также поддерживаемые техники верификации и тестирования, определяются разработчиками инструмента и не могут быть изменены и расширены сколь-нибудь существенным образом. Более того, в последнее время многие компании, предоставляющие средства разработки ПО, предлагают интегрированные решения, охватывающие практически весь процесс разработки, причем включение в них посторонних, заранее не предусмотренных техник и средств невозможно, либо очень трудоемко. Вместе с тем, процессы разработки ПО в разных организациях сильно различаются, в связи с отличиями в создаваемых ими программах, в требованиях клиентов и способах взаимодействия с ними, в используемых бизнес-моделях и т.д. Поэтому все более востребованы инструменты разработки, имеющие модульную архитектуру, способные с минимальными накладными расходами интегрироваться с другими инструментами и включать модули, реализующие передовые, более эффективные техники выполнения отдельных видов деятельности по разработке ПО. Эти требования можно суммировать так: инструменты разработки сложных систем тоже должны быть компонентными и облегчать свое включение в разнообразные технологические цепочки. Среди средств верификации такими свойствами пока обладают лишь инструменты *модульного тестирования* (unit testing) [5], самым известным из которых является JUnit [6], модульных инструментов, поддерживающих более сложные виды верификации или тестирование на соответствие требованиям, практически нет.

- Отсутствие поддержки масштабного многократного использования возникающих в ходе контроля качества артефактов — тестов, верифицированных утверждений, моделей и пр. Эта проблема особенно важна именно для компонентных технологий, поскольку верификация одного компонента чаще всего имеет приемлемую стоимость, в ее ходе создаются многочисленные вспомогательные артефакты, а использовать их далее при проверке взаимодействующих компонентов или гораздо более дорогостоящем контроле качества систем, включающих проверенный компонент, уже не удается. Обеспечение многократного использования таких артефактов могло бы значительно снизить издержки верификации компонентного ПО.

Преодолеть возникающие трудности можно с помощью *компонентных технологий верификации* программных и аппаратных систем. Такие

технологии позволили бы независимо проверять качество отдельных компонентов в соответствии с требованиями к ним, и использовать получаемые при этом модели, тесты и пр. для более аккуратного контроля корректности их интеграции и качества компонентных систем, постепенно наращивая количество проверенных компонентов, подсистем, различных требований к системе в целом. Более детально, подобная технология должна обладать следующими характеристиками.

- Предоставлять средства определения требуемых свойств отдельного компонента, включающих требования компонента к его окружению и его ответные обязательства, предоставляемые им службы и их характеристики.
- Давать возможность использовать те же средства для фиксации необходимых свойств подсистем, объединяющих в себе несколько компонентов (и других подсистем).
- Предоставлять инструментарий для независимой проверки компонентов и подсистем на соответствие требованиям к ним.
- Обеспечивать контроль взаимного соответствия взаимных требований и обязательств компонентов и подсистем при их интеграции и совместной работе.
- Позволять эффективно использовать результаты проверки отдельных компонентов и подсистем и наработанные в ее ходе материалы в виде формальных моделей, ограничений использования, верифицированных утверждений, тестов и т.п. в неизменном виде при контроле качества включающих их подсистем и систем.

Помимо этих свойств, чтобы получить признание на практике, компонентная технология верификации должна быть расширением одной из существующих компонентных технологий разработки и задействовать по большей части ее элементы, уже знакомые и привычные для разработчиков.

Свойства технологии, связанные с ее гибкостью, возможностью использования для решения широкого множества задач, во многом обуславливаются лежащей в ее основе архитектурой, определяющей общую организацию работ, входные и выходные данные на каждом этапе. Для компонентной технологии архитектурные решения еще более значимы, поскольку они непосредственно определяют систему правил создания, развития и добавления новых компонентов, их разновидности, шаблоны организации их взаимодействий, возможные конфигурации и пр.

В данной работе представлены некоторые элементы компонентной технологии верификации, использующей методы тестирования на основе моделей и базирующейся на компонентных технологиях платформы Java, однако представленная технология находится в процессе создания и в окончательном виде пока не сформирована. Основная же часть статьи посвящена архитектурному каркасу (framework) этой технологии,

включающему ее инструментальные библиотеки, а также набор основных типов компонентов и правила их интеграции. Этот каркас может быть использован в несколько более широком контексте, чем сама технология, поэтому целесообразно иметь его отдельное описание.

Изложение организовано следующим образом. Во втором разделе рассматривается подход к тестированию на основе моделей, анализируются его требования к архитектуре поддерживающего инструментария и реализация этих требований в уже имеющихся инструментах. Затем формулируются принципы компонентной разработки и представляются основные элементы предлагаемого архитектурного каркаса. Далее описан небольшой пример создания тестов с его помощью. Заключение статьи резюмирует ее содержание и обрисовывает направления дальнейшего развития изложенных идей.

2. Тестирование на основе моделей и инструменты тестирования

Тестирование на основе моделей (model based testing) [7,8] представляет собой подход к тестированию, в рамках которого тесты строятся вручную, автоматизированным образом или генерируются полностью автоматически на основе *модели поведения* тестируемой системы и *модели ситуаций*, связанных с ее работой.

- *Модель поведения* (behavior model) формализует требования к тестируемой системе, т.е. описывает, какие внешние воздействия на нее в каких ситуациях допустимы, и как она должна реагировать на эти воздействия, как она должна работать.
  Модель поведения служит основой для *тестового оракула* [9-11] — компонента или процедуры, производящей оценку поведения системы во время тестирования.
  Чаще всего при тестировании используются модели в виде автоматов разного вида: наиболее простые конечные автоматы, расширенные автоматы, системы помеченных переходов [7], Statecharts [12,13], временные автоматы [14,15] и т.д. Однако в таком качестве можно использовать другие разновидности моделей: контрактные спецификации в виде пред- и постусловий операций, алгебраические спецификации в виде правил эквивалентности различных цепочек вызовов операций, трассовые или сценарные модели, описывающие возможные последовательности воздействий и реакций системы.

- *Модель ситуаций* (situation model) формализует структуру возможных тестовых ситуаций, составляющими которых являются внешние воздействия и состояния самой системы и ее окружения, определяет различные классы ситуаций и их важность с точки зрения контроля качества. Обычно такая модель определяет конечный набор классов

эквивалентности ситуаций, подразумевая, что при тестировании достаточно проверить работу системы хотя бы в одной ситуации из каждого класса. Достаточно часто также для задания модели ситуаций используется конечный набор элементарных событий, при этом каждая ситуация соответствует некоторому множеству таких событий. Примером такого события может являться выполнение заданной инструкции в коде тестируемой системы. В более сложных случаях определяемые классы ситуаций могут пересекаться, и им могут приписываться приоритеты и веса, показывающие, насколько важно проверить одну из ситуаций такого класса.

Модель ситуаций используется для решения двух тесно связанных задач: определения *критерия адекватности* или *полноты тестирования* [16] и определения числовых метрик полноты тестирования. Заметим, что под полнотой тестирования здесь имеется в виду удовлетворение определенному критерию (скажем, достижение 80% покрытия кода), а совсем не исчерпывающий перебор всех практически возможных ситуаций. Критерий полноты тестирования задает свойства набора тестов, позволяющие считать его достаточно хорошо представляющим разнообразное поведение проверяемой системы для решения некоторого круга задач и больше тестов не создавать. Метрики полноты тестов представляются в терминах процентной доли проверяемых ими классов ситуаций разных видов. Обычно в качестве критерия полноты тестирования устанавливается достижение какого-то значения метрики, но при этом само значение метрики бывает полезно знать дополнительно. Чаще всего полнота тестирования определяется на основе покрытия классов эквивалентных ситуаций или элементарных событий, и в этих случаях говорят о *критерии тестового покрытия*, *метрике тестового покрытия*, а достигнутый в ходе тестирования процент покрытых классов ситуаций называют *тестовым покрытием*.

Построение тестов на основе моделей заключается в том, что создается (извлекается из проектных документов или, реже всего, берется откуда-то в готовом виде) модель поведения тестируемой системы, создается модель ситуаций, отражающая основные приоритеты и риски проекта и, чаще всего, использующая структурные элементы модели поведения, и затем строится (генерируется автоматически, создается вручную, или разрабатывается с существенной помощью инструментов) комплект тестов, проверяющих соответствие между реальным поведением тестируемой системы и ее моделью поведения. При этом тестовый набор создается так, чтобы он удовлетворял критерию полноты тестирования, заданному моделью ситуаций.

Используемые для построения тестов методы можно разделить три типа.

- *Вероятностные методы* используют псевдослучайную генерацию данных элементарных типов, псевдослучайное комбинирование их в

более сложные структуры и псевдослучайную же генерацию последовательностей тестовых воздействий, если необходимо проверять поведение системы в разных состояниях. При этом полноту тестирования стараются обеспечить за счет большого количества разнообразных тестов, так как построение каждого отдельного теста требует небольших затрат.

- *Нацеленные методы*, наоборот, строят тестовые данные и последовательности целенаправленно, так, чтобы те удовлетворяли определенным свойствам, чаще всего — реализовывали ситуации из определенных классов, задаваемых моделью ситуаций. Критерий полноты тестирования при использовании таких методов удовлетворяется близким к минимальному количеством тестов, но построение каждого теста обычно требует значительных усилий человека и/или затрат вычислительных ресурсов.

- *Комбинаторные методы* строят тестовые данные и последовательности с помощью комбинирования различных типов их элементов по определенным схемам. Они занимают промежуточное положение: на создание одного теста нужно существенно меньше усилий, чем при нацеленном построении, но несколько больше, чем при вероятностном. Тестовый набор получается значительно больше, чем при нацеленном тестировании, но много меньше, чем при вероятностном, хотя в нем и достигается большее разнообразие. Кроме того, комбинаторные методы обычно гарантируют, что в полученном наборе нет совершенно идентичных тестов, которые очень часто возникают в случайно сгенерированных наборах. Некоторые комбинаторные методы, использующие эффективную фильтрацию «лишних», не вносящих собственный вклад в достижение полноты тестов, вполне сопоставимы с нацеленными методами по характеристикам получаемого тестового набора. Примером служат техники построения тестов на основе автоматных моделей, создающие тесты в виде набора путей по графу переходов автомата, минимизируя его с точки зрения покрытия и проверки всех возможных состояний и переходов.

2.1. Требования к представлениям моделей

Инструменты, поддерживающие тестирование на основе моделей, должны работать с какими-то представлениями моделей поведения и моделей ситуаций. Для полноценной поддержки этого подхода к тестированию и повышения удобства его использования в практике разработки ПО, эти представления должны обладать следующими свойствами.

- Модели поведения.

- Средства для описания требований в разных стилях, как минимум в виде чистых декларативных ограничений (утверждений о свойствах входных данных операций и их результатов), в виде контрактных спецификаций (пред- и постусловий операций), использующих модельное состояние, в виде исполнимых моделей, определяющих способ работы проверяемой системы на более высоком уровне абстракции. Это требование обусловлено необходимостью, с одной стороны, как можно более ясно отражать имеющиеся требования, сформулированные на естественном языке, и, с другой стороны, допускать возможность их переформулирования в другом стиле в ходе построения моделей, поскольку оно позволяет выявить неточности, рассогласования и неполноту в исходных требованиях и углубить их понимание [17].
- Средства для явного указания связей с документами, содержащими требования, и стандартами, если они относятся к проверяемой системе. Такие связи используются для прослеживания исходных требований и установления того, что все необходимые ограничения действительно присутствуют в модели, а лишних ограничений в ней нет. В дальнейшем они же позволяют проследить связи требований с полученными автоматически тестами.
- Возможность автоматического преобразования моделей в тестовые оракулы.
- Возможность использования структуры моделей для определения критериев тестового покрытия.
- Стоит поддерживать возможности использования таких моделей в рамках других техник верификации.
- Полезна также возможность преобразования модели поведения в однозначную и полную документацию на моделируемую систему. Однако способы реализации этой возможности остаются за рамками данной работы.

- Модели ситуаций.
  - Средства для описания ситуаций, связанных с различными аспектами поведения и структуры тестируемой системы. На практике критерии полноты тестирования используют разнообразные свойства для характеризации ситуаций: структуру входных данных и результатов, внутренние состояния системы, выполняемые инструкции кода и вызываемые внутри функции, шаблоны передачи данных внутри системы, шаблоны взаимодействия различных частей

системы при выполнении операции, характеристики ожидаемого поведения, возможные ошибки в системе и пр.
    - Средства для явного указания связей определяемых классов ситуаций с исходными требованиями к системе.
    - Возможность автоматического преобразования в компоненты, определяющие класс текущей ситуации, измеряющие достигнутое покрытие.
    - Возможность автоматического извлечения из описания классов ситуаций ограничений на входные данные и состояние тестируемой системы или моделей ее поведения, чтобы можно было их использовать для нацеленного построения соответствующих тестов.

Как видно, требования к представлению моделей делятся на три типа: достаточная для практических целей выразительность, прослеживаемость к требованиям и возможность автоматического решения различных задач построения тестов.

2.2. Инструменты модульного тестирования

Рассмотрим теперь имеющиеся инструменты тестирования с точки зрения их приближения к желаемому идеалу — инструментарию на основе компонентной технологии.

Важным классом инструментов тестирования, в большой степени обладающих свойствами модульности, являются средства *модульного тестирования* (unit testing) [5]. Наиболее известен из таких инструментов JUnit [6], написанный на Java и предназначенный для тестирования кода на этом языке, хотя исторически первым был SUnit [18,19] для программ на Smalltalk.

Для этих инструментов характерны высокая гибкость, возможность подключения совершенно независимых модулей для реализации дополнительных функций и возможность использования в рамках более сложных тестовых систем. Одним из инструментов модульного тестирования, обладающих наиболее богатой функциональностью, является TestNG [20,21]. Его основные характеристики таковы.

- Конфигурация тестов.
    - Основные элементы тестов — тестовые классы и тестовые методы, описываемые как классы и методы языка Java, имеющие аннотацию @Test.
    - Комплект тестов имеет иерархическую структуризацию: в нем выделяются тестовые наборы (test suites), состоящие из тестов (tests). Далее в этом списке характеристик эти термины используются только в узком, специфическом для TestNG смысле. Тесты и тестовые наборы определяются

конфигурацией тестов, описываемой в некотором формате на базе XML. Тест составляется из методов нескольких классов, выбираемых либо на основе их имен, либо по принадлежности к группам, указываемым в аннотациях методов.
    - Методы инициализации (set-up) и финализации (tear-down), используемые для инициализации некоторых данных перед выполнением тестов и освобождения занятых ресурсов после, могут быть определены для всех элементов иерархии: для наборов, тестов, классов и методов.
    - Возможно определение зависимостей между тестовыми методами и между методами и группами, позволяющее управлять порядком их выполнения и отменять выполнение тестового метода, если один из тестовых методов, от которых он зависит, выполнился с ошибками.

- Тестовые данные и объекты.
    - Тестовые методы в TestNG могут быть параметризованными. Набор значений параметров, используемый в рамках теста, указывается с помощью дополнительной аннотации или в конфигурационном файле и должен представляться либо в виде заданной коллекции объектов, либо как последовательность результатов, возвращаемых определенным методом при работе теста.
    - Можно создавать фабрики объектов, строящие разнообразные объекты тестовых классов. Все входящие в тест методы выполняются для каждого такого объекта.

- Проверка результатов тестирования.
    - Основной способ описания выполняемых тестовыми методами проверок, как и во многих других инструментах модульного тестирования — это использование библиотеки методов-утверждений. Каждый такой метод (обычно их названия начинаются с assert) проверяет простое свойство своих аргументов (равенство, неравенство null, вхождение символа в строку, вхождение объекта в коллекцию и пр.) и при его нарушении выдает трассировочное сообщение, также указываемое в виде аргумента.
    - Дополнительно TestNG поддерживает указание возможных исключений и ограничений на время работы тестового метода в его аннотации.

Инструменты модульного тестирования активно используют развиваемые независимо модули для решения разных более специфичных задач. Например,

dbUnit [22] — для организации работы с базами данных в модульных тестах используется, httpUnit [23] — для обработки HTTP-запросов. Для более наглядной записи выполняемых в тестах проверок (близкой к формулировкам естественных языков) можно применять библиотеки, предоставляемые инструментами разработки на основе функциональности (behavior driven development), например, JBehave [24] или NSpecify [25], для организации тестовых заглушек — библиотеки Mockito [26], EasyMock [27] и т.д.

2.3. Инструменты тестирования на основе моделей

Инструменты тестирования на основе моделей с точки зрения их приближения к желательной компонентной архитектуре можно разделить на три группы.

- Традиционные «монолитные» инструменты, использующие специфические языки для оформления моделей. Добавление новых компонентов в них может на практике осуществляться только их разработчиками, а их использование в рамках более широкого инструментария возможно, только если разработчики заранее позаботились о предоставлении подходящего набора интерфейсов.
  К инструментам такого типа относятся практически все многочисленные исследовательские прототипные средства тестирования на основе моделей и ряд более стабильных инструментов, использовавшихся во многих разных проектах. В этой второй категории находятся TorX [28,29], TGV [30,31], BZ-TT [32] и Gotcha-TCBeans [33,34]. Все они основаны на моделировании проверяемой системы в виде различных автоматных моделей. На тех же принципах в целом построены и коммерческие инструменты Conformic Qtronic [35] и Smartesting Test Designer (ранее Leirios) [36].

- Инструменты на основе расширений широко используемых языков программирования, которые сохраняют «монолитность». Примерами являются поддерживающие технологию UniTESK [37-39] инструменты CTESK и JavaTESK, а также SpecExplorer [40,41], разрабатываемый в Microsoft Research. В обоих случаях для моделирования поведения тестируемых систем используется комбинация из автоматных моделей и контрактных спецификаций.

- Инструменты, использующие для оформления моделей обычные языки программирования и обладающие рядом характеристик модульности. В частности, эти характеристики включают возможности достаточно легкой интеграции с компонентами от независимых разработчиков и использования самих таких инструментов как части более широкого инструментария.
  Такие инструменты начали появляться не так давно, около 4-5 лет назад. Два наиболее известных примера — это ModelJUnit [8,42] и NModel [43,44]. Похожий инструмент mbt.tigris.org [45] использует

для описания моделей графическую нотацию, поэтому гораздо менее приспособлен для использования в рамках чужих разработок. Иного рода пример — недавно созданная в Microsoft Research библиотека для описания и контроля декларативных ограничений на поведение .NET компонентов CodeContracts [46,47].

Возможности инструментов последнего вида стоит рассмотреть подробнее.

ModelJUnit и NModel построены как расширения простых средств модульного тестирования. Они используют для построения тестов только модели поведения, модель ситуаций явно не задается, неявно разные критерии полноты закладываются в используемые для построения тестов алгоритмы. Модель поведения преобразуется в сопряженную *модель теста*, определяющую не само требуемое поведение, а способ его исследования и проверки.

- Модель теста описывается в виде расширенного конечного автомата, представленного на языке программирования (Java или C#) в виде тестового класса, как и в средствах модульного тестирования. Такой класс либо помечается с помощью некоторой аннотации (атрибута в C#), либо реализует определенный интерфейс. Элементы автоматной модели — состояния, переходы, охранные условия переходов — задаются с помощью методов и полей этого класса.
- Состояние в NModel задается набором значений полей тестового класса (можно определять поля, не включаемые в состояние, помечая их специальными атрибутом), а в ModelJUnit — результатом метода getState().
- *Действия*, в результате выполнения которых выполняются переходы в модели теста, представляются методами, помеченными определенной аннотацией (атрибутом). В NModel действия могут быть параметризованными, причем набор значений параметров, используемый в рамках теста, указывается с помощью дополнительной аннотации в виде коллекции объектов соответствующего типа.
- Действия могут иметь *охранные условия*, которые должны быть выполнены, чтобы можно было выполнять соответствующее действие. В обоих случаях охранные условия представлены как методы, имеющие имена, построенные из имени соответствующего метода-действия с некоторым постфиксом.
- NModel дополнительно имеет следующие возможности.
    - Композиция нескольких моделей, в которых одноименные действия рассматриваются как выполняемые одновременно. Модели для композиции указываются инструменту построения тестов перечислением имен соответствующих классов.

    - Проверка моделей (model checking). Свойства безопасности проверяются за счет анализа возможности нарушения инвариантов состояний, представленных как методы с особым атрибутом. Свойства живучести могут удостоверяться за счет анализа достижимости стабильных состояний, в которых специально отмеченные характеристические методы возвращают true.

Библиотека CodeContracts предоставляет средства для описания чисто декларативных ограничений на свойства входных параметров и результатов операций. Моделирование состояния не поддерживается. Имеются следующие возможности.

- Ограничения записываются на языке C# в виде булевских выражений, передаваемых как аргументы библиотечным методам.
- Можно описывать ограничения следующих видов.
    - Предусловия операций (метод Contract.Requires).
    - Постусловия операций при нормальной работе (Contract.Ensures).
    - Постусловия операций при выдаче исключения (Contract.EnsuresOnThrow).
    - Утверждения о выполнении некоторых свойств в какой-то точке кода внутри метода (Contract.Assert).
    - Инварианты классов — оформляются в виде методов со специальной аннотацией, содержащих обращения к Contract.Invariant.
- Помимо обычных выражений на C# можно использовать следующие.
    - Кванторные выражения, оформленные в виде обращений к методам Contract.Exists и Contract.ForAll.
    - Обращение к значениям выражений до выполнения проверяемого метода в постусловиях в виде вызова метода Contract.OldValue.
    - Обращение к значению результата в постусловиях с помощью Contract.Result.
- Библиотека CodeContracts дополняется двумя инструментами: для статической проверки сформулированных ограничений на основе дедуктивного анализа и для их динамической проверки при выполнении методов, ограничения к которым описаны.

2.4. Итоги обзора существующих инструментов

Из приведенного обзора видно, что компонентные средства тестирования на основе моделей в последние годы активно развиваются, однако пока не достигли необходимой функциональности, отставая по ряду возможностей от инструментов модульного тестирования. Необходимо дополнить имеющиеся наработки следующими возможностями.

- **Явное определение модели поведения проверяемого компонента, отдельное от модели теста**. Такое разделение необходимо, прежде всего, для поддержки точности и полноты моделирования. Не менее важно и многократное использование моделей, поскольку на основе одной модели поведения компонента могут быть построены разнообразные тесты, нацеленные на достижение различных целей, как для этого компонента, так и для содержащих его подсистем. Еще одна полезная возможность — использование таких моделей в рамках других техник верификации. Пример выделения модели поведения дает библиотека CodeContracts.

- **Расширенные выразительные возможности для описания моделей поведения.** В частности, нужно обеспечить возможность использования разных стилей моделирования и сочетания разнообразных методов и техник построения тестов, а также других методов верификации. Это требование является следствием сложности и разнообразия требований к современным программным компонентам [48,49]. Библиотека CodeContracts позволяет указывать только чистые декларативные ограничения, что существенно снижает возможности ее использования для большинства практически значимых систем, поведение которых зависит от внутренних состояний их компонентов.

- **Явное определение моделей ситуаций для построения тестов.** Неявное задание модели ситуаций в имеющихся сейчас инструментах ограничивает возможности использования различных критериев полноты тестирования и затрудняет их осознанный выбор разработчиками тестов.

- **Средства явной привязки моделей к требованиям на естественном языке**. Такие возможности, необходимые для прослеживания требований, имеются в инструментах CTESK и SpecExplorer. В средствах модульного тестирования так можно использовать текстовые сообщения в методах проверки утверждений, однако подобных возможностей лишены NModel и ModelJUnit. Отметим, что добавление такой функциональности в модульный инструмент не представляет большого труда.

- **Совместное использование моделей поведения, ситуаций и тестов, связанных с различными аспектами функциональности одного компонента на основе неинвазивной композиции.** За счет этого можно существенно облегчить многократное использование моделей в тестировании и других техниках верификации. NModel частично решает эту задачу для моделей тестов за счет использования возможности расширять определения классов новыми полями и методами в C#.

3. Архитектурный каркас для тестирования на основе моделей

В данном разделе описывает предлагаемый подход к построению компонентной архитектуры инструментов тестирования на основе моделей, удовлетворяющей сформулированным выше требованиям. Однако прежде стоит сделать ряд замечаний, касающихся выбираемых средств решения поставленных задач.

- Как отмечалось выше, необходимо использовать в моделях разные техники описания ограничений на поведение тестируемых компонентов. При этом нужно отдавать себе отчет в том, что поддержка совершенно произвольных видов моделей в рамках одной технологии, скорее всего, недостижима. Поэтому нужно сразу отметить ряд практически важных походов, совместная поддержка которых реализуема.

  При моделировании программных интерфейсов с устоявшимися требованиями достаточно удобно применять *контрактные спецификации* в виде пред- и постусловий, опирающихся на модельное состояние моделируемых компонентов. Подходы на их основе продемонстрировали достаточную масштабируемость и эффективность в терминах трудозатрат на описание некоторого набора элементов интерфейса [38,50,51].

  С другой стороны, для моделирования вычислений с плавающей точкой, сложных протоколов и ряда других видов ПО, иногда более удобно использовать *операционные модели*, являющиеся, по сути, альтернативными реализациями той же функциональности. Наиболее удобными на практике моделями такого вида оказываются расширенные автоматы и системы переходов с возможностью их композиции.

  При моделировании некоторых реактивных систем, обрабатывающих большие потоки событий, или служб, предназначенных для регулярной обработки данных из большой базы, полезными оказываются *потоковые контракты*, описывающие ограничения не на конечный результат обработки, а на обработку одного структурного элемента во входном потоке данных

- Модели поведения и модели ситуаций, а также сами тесты и элементы инструментария разработки предполагается оформлять в виде компонентов или наборов компонентов (подсистем) в рамках выбранной базовой компонентной технологии, с минимальным добавлением каких-либо новых конструкций, требующих дополнительной языковой и инструментальной поддержки.

  Такой подход позволит применять для работы с этими моделями и для их интеграции с проверяемыми компонентами все инструменты, средства и техники, предлагаемые базовой компонентной

технологией. Соответственно, значительно снижаются затраты на сопровождение и развитие инструментария, поддерживающего такую технологию. Это обеспечивает возможность использования того же инструментария и созданных моделей компонентов при разработке тестов для подсистем и крупных систем. Тем самым создается основа для выполнения требований к компонентным технологиям верификации.

- Применение широко распространенных компонентных технологий и языков программирования позволяет снизить трудности обучения использованию инструментария, а также при необходимости использовать многочисленные вспомогательные библиотеки, разработанные для решения специфических задач модульного тестирования.

- Верификационные системы часто из-за более многочисленного ассортимента различных компонентов получаются сложнее систем, для проверки которых предназначены. Поэтому еще более важно снижать их сложность и трудоемкость их поддержки и развития. Для облегчения интеграции и переконфигурирования систем из многочисленных компонентов предлагается везде, где можно, использовать неинвазивные техники композиции и интеграции компонентов, избегающие внесения каких-либо изменений в их код. Это можно обеспечить за счет широкого использования образца внедрения зависимостей (dependency injection) [52] и поддерживающих его библиотек-контейнеров, а также других средств современных компонентных технологий.

3.1. Виды компонентов и их интеграция

Основой инструментария тестирования на основе моделей предлагается сделать *контейнер внедрения зависимостей* (dependency injection container), позволяющий задавать список компонентов, входящих в систему, инициализировать их состояние и определять связи между ними внешним образом, без вмешательства в код этих компонентов.

Верификационная система строится из компонентов различных типов.

- Собственно, проверяемые компоненты.
  На них накладывается только одно ограничение: возможность их внешней инициализации с помощью контейнера внедрения зависимостей. В большинстве случаев это ограничение выполняется, иначе обычно достаточно просто написать компонент-обертку, удовлетворяющий ему и предоставляющий доступ к проверяемым операциям исходного компонента.
  Проверяемые компоненты не имеют зависимостей от тестовой

системы, за исключением заглушек, подменяющих необходимые им для работы компоненты.

- Модели поведения (обобщенные контракты).
  Они оформляются на базовом языке программирования как классы с несколькими методами, выполняющими определенные роли. Например, если используется чисто декларативная спецификация, в ней должны быть определены пред- и постусловия, причем любой метод, возвращающий булевское значение, может играть эти роли. Для спецификации, использующей модельное состояние компонента, необходимо определить синхронизатор состояния, вызываемый, чтобы поддерживать в соответствии состояние модели и реальное состояние проверяемого компонента. Исполнимые спецификации должны определять предусловия и модельные операции.
  Модели поведения зависят от проверяемых компонентов или, в случае существенных различий в интерфейсах между моделью и моделируемым компонентом — от адаптеров, устраняющих такие различия.

- Модели взаимодействия.
  При описании многокомпонентных систем иногда, помимо моделей отдельных компонентов, необходимо явно вводить модель их взаимодействия, позволяющую оценить корректность сложных конгломератов воздействий и реакций, в которых задействовано несколько компонентов, каждый из которых осведомлен лишь о событиях, относящихся к его интерфейсу. Например, моделью взаимодействия является так называемая *семантика чередования* для асинхронных взаимодействий параллельно работающих компонентов, в рамках которой корректен любой набор событий, который можно линейно упорядочить так, чтобы каждое отдельное событие в таком порядке стало корректным относительно моделей компонентов, создающих или обрабатывающих его [50].
  Модели взаимодействия оформляются в виде шаблонных библиотечных модулей, привязываемых в конфигурационном файле к соответствующим им группам компонентов. Для каждого конкретного взаимодействия порождается экземпляр такого шаблона, зависящий от моделей поведения вовлеченных в него компонентов.

- Модели ситуаций.
  Модели ситуаций оформляются на базовом языке программирования в виде методов, фиксирующих наступление определенных ситуаций после проверки описывающих их ограничений. Модели ситуаций для некоторой операции могут включать как пре-ситуации, определяемые входными аргументами операции и состояниями вовлеченных компонентов до ее вызова, так и пост-ситуации, соответствующие

определенным свойствам результатов и состояний компонентов после работы операции. Модели пост-ситуаций могут иметь модельное состояние и методы-синхронизаторы, так же, как и контрактные спецификации.
Модели ситуаций зависят от проверяемых компонентов, моделей поведения или теста, сообразно тому, в каких терминах они описывают ситуации.
Модели ситуаций могут быть сгенерированы автоматически из моделей поведения, интерфейсов и кода проверяемых компонентов, поскольку критерии полноты тестирования на основе структуры кода или функциональности часто используются при построении тестов. Такие компоненты, генерируемые из других, далее будем называть *вторичными*.

- Тесты.
  Тесты, так же, как и модели ситуаций, могут создаваться разработчиками или генерироваться из моделей поведения и интерфейсов тестируемых компонентов. Каждый тест должен определять последовательность обращений к операциям тестируемого компонента (быть может, состоящую из единственного обращения) и значения параметров этих обращений, тестовые данные.
    - Для решения первой задачи можно использовать два подхода.
      - На практике более часто применяется связка из автоматной модели теста и генератора путей по графу переходов автомата. Эта техника положена в основу ModelJUnit и NModel.
        В этом случае автоматная модель теста должна определять методы, играющие роль действий, охранных условий, а также возвращающие текущее состояние автомата. Модель теста может зависеть от проверяемого компонента или от его модели поведения.
      - Для ряда случаев более эффективно применять монолитный генератор последовательностей, использующий информацию о тестируемом интерфейсе.
    - Генерация тестовых данных, особенно данных сложной структуры, может использовать большое количество компонентов, играющих различные роли.
      - Первичные генераторы, которые строят объекты некоторого типа. Такой генератор может быть устроен как итератор по некоторой коллекции.
      - Фильтры, выполняющие отсев данных, не удовлетворяющих определенным ограничениям.

- Решатели ограничений, прямым образом строящие данные, удовлетворяющие некоторым ограничениям.
- Комбинаторы, строящие данные сложного типа из простых объектов.
- Преобразователи, генерирующие данные некоторого типа по более простому кодированному их представлению.

- Адаптеры.
  Адаптеры устраняют возможные расхождения между интерфейсами моделей и моделируемых ими компонентов.
  Адаптеры зависят от проверяемых компонентов. В тех случаях, когда они отвечают за синхронизацию модельного состояния, имеется зависимость и от соответствующей модели поведения.
  Можно отметить, что во многих случаях небольшие расхождения между модельным и проверяемым интерфейсами не требует написания адаптера, поскольку могут быть устранены указанием библиотечной процедуры преобразования. Это относится к случаям, в которых различия сводятся к отсутствию ряда параметров, перестановке параметров местами или простым преобразованиям типов, например, чисел в строки и обратно. Во всех этих случаях адаптер строится не вручную, а автоматически, лишь по указанию соответствующего преобразования в конфигурационном файле тестовой системы.

- Заглушки (stubs, test doubles).
  Заглушки подменяют во время теста компоненты, от которых зависят проверяемые. Они бывают двух видов.
  - *Управляющая заглушка* (mock) передает в проверяемый компонент какие-то данные в виде возвращаемых ее методами результатов и служит дополнительным источником воздействий на тестируемый компонент.
  - *Наблюдающая заглушка* (test spy) фиксирует вызовы ее операций и их аргументы для проверки корректности сделанных тестируемым компонентом обращений.

  В принципе, одна заглушка может играть обе роли одновременно, но на практике такая потребность возникает крайне редко (это признак очень сложной организации теста, которую, возможно, имеет смысл пересмотреть).
  Если заглушки используются, проверяемый компонент зависит от них. Тест или модель поведения зависят от наблюдающей заглушки, которую они используют. И наоборот, управляющая заглушка сама зависит от теста, поскольку именно тест должен определять результаты очередного вызова ее операций.

- Вспомогательные компоненты. К вспомогательным относятся компоненты, решающие многочисленные задачи организации работы верификационной системы, интеграции ее составляющих и сбора информации о происходящих событиях.
    - Трассировщики различных видов событий. По сути, к каждому компоненту прикрепляется отдельный трассировщик событий, связанных с этим компонентом. Все трассировочные сообщения собираются одним или несколькими генераторами трасс.
    - Планировщики тестов, включающих асинхронные воздействия, отвечающие за создание отдельных процессов и нитей внутри тестовой системы и распределение воздействия по ним.
    - Диспетчеры и синхронизаторы отдельных операций в асинхронных и параллельных тестах.
    - Конфигураторы, определяющие связи внутри некоторых групп компонентов.

Рисунок 1 демонстрирует одну из возможных конфигураций тестовой системы на основе предлагаемой архитектуры. Связи между компонентами, изображенные на рисунке, представляют собой зависимости, характерные для компонентов такого типа (хотя не все возможные зависимости изображены). Связи генератора трассы и конфигуратора не показаны, поскольку все или почти все компоненты связаны с ними.

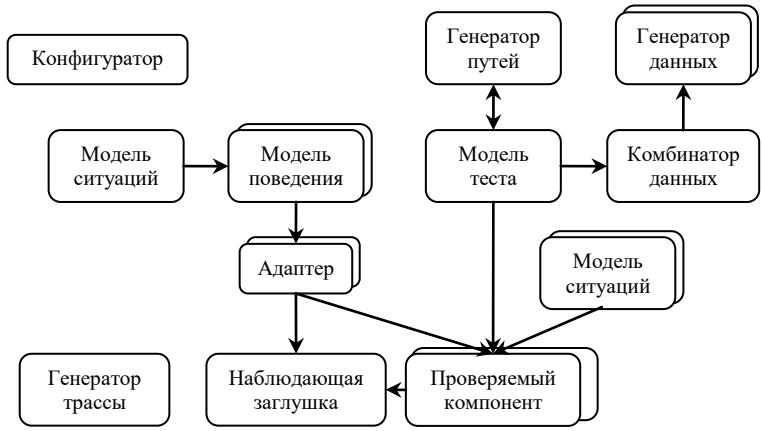


Рис. 1. Схема построения тестовой системы.

Конкретный набор компонентов и связи между ними описываются в конфигурационном файле в XML-формате, поступающем в начале работы на

вход контейнеру внедрения зависимостей, который инициализирует все компоненты и связывает их нужным образом. Такой способ задания связей позволяет строить различные конфигурации тестовой системы, не меняя кода ее компонентов, и даже не имея к нему доступа. Вместе с тем, возможно определение жестких связей в самом коде, а также более гибкое связывание с помощью аннотаций и создание специальных компонентов-конфигураторов, которые содержат явную инициализацию компонентов и связей между ними на базовом языке программирования.

3.2. Реализация предложенного подхода

Для реализации предложенной архитектуры в качестве базового языка программирования был выбран язык Java. Он обладает многими языковыми возможностями, необходимыми для описания различных ролей классов и методов, а также связей между компонентами: средствами описания декларативной информации об элементах кода в виде аннотаций и поддержкой получения в динамике информации о структуре компонентов и сигнатурах их операций (интроспекция или рефлексия).

В качестве контейнера внедрения зависимостей была выбрана открытая библиотека Spring [45,46], поддерживающая достаточно большой набор функций системы такого типа.

Для описания моделей поведения была разработана небольшая библиотека, похожая, с одной стороны, на библиотеки проверки утверждений в средствах модульного тестирования (используются разнообразные методы assert()) и, с другой стороны, на Microsoft CodeContracts (для доступа к результату операции и пре-значениям выражений используются методы result() и oldValue()). В отличие от CodeContracts поддерживается создание контрактных спецификаций, использующих модельное состояние. В разработанной библиотеке отсутствуют имеющиеся в CodeContracts кванторные выражения и поддержка статического анализа ограничений.

Для описания моделей ситуаций также создана небольшая библиотека, обеспечивающая трассировку информации о покрытии указываемых ситуаций.

Тесты оформляются в стиле, аналогичном ModelJUnit и NModel, но с некоторыми расширениями, частично заимствованными из TestNG.

- Поддерживается расширенная иерархия элементов тестов: тестовые наборы, тесты, тестовые классы, тестовые методы. Тестовый набор состоит из тестов, один тест может включать в себя несколько тестовых классов.
- Тестовый класс описывает расширенный конечный автомат.
- Состоянием этого автомата считается результат работы всех методов, помеченных аннотацией State. Такое решение делает возможным добавление новых элементов в состояние без модификации ранее

написанного кода. Состояние теста является списком состояний входящих в него классов.

- Тестовые методы определяют действия автомата, возможно параметризованные. Значения параметров извлекаются из связанного с тестовым методом провайдера данных. Провайдер может быть генератором наборов значений, определенных, например, как элементы некоторой коллекции, а может быть построен динамически из генераторов данных для разных параметров с определенной стратегией их комбинирования (выбирать все комбинации, каждое значение каждого параметра, все возможные пары значений и пр.). Провайдеры данных и способ их комбинирования задаются с помощью аннотаций метода и его отдельных параметров.
- Действия могут иметь охранные условия, оформляемые в виде методов, возвращающих булевский результат и зависящих от состояния объекта тестового класса. Охранное условие привязывается к тестовому методу при помощи аннотаций, без использования соглашений об именовании методов. Поэтому одно и то же условие может быть использовано для разных методов, и один метод может иметь несколько охранных условий. Кроме того, охранные условия могут иметь в качестве параметров любой набор, являющийся началом набора параметров соответствующего метода, в том числе пустой (в этом случае охранное условие зависит только от текущего состояния).
- Так же, как в TestNG, любой тестовый элемент — набор, тест, класс, метод — может иметь конфигурационные методы инициализации и финализации. Дополнительно можно определять конфигурационные методы, вызываемые при посещении очередного состояния.

Для построения заглушек используется свободная библиотека Mockito [26]. Она имеет достаточно богатые возможности для определения управляющих и наблюдающих заглушек и использует интуитивно понятный синтаксис при их описании. Этот пример показывает, что при наличии Java-библиотеки с необходимой функциональностью, она без особых усилий может быть использована в рамках предлагаемой архитектуры.

4. Пример построения теста

Далее описывается пример использования предложенных решений при построении тестов для простой реализации функциональности банковского счета. Интерфейс тестируемого компонента приведен ниже.

```
public interface Account
{
  int getBalance();
  int getMaxCredit();
```

```
    Validator getValidator();
    void setValidator(Validator p);

    AuditLog getLog();
    void setLog(AuditLog log);

    int transfer(int sum);
  }
```

Методы `getBalance()` и `getMaxCredit()` служат для получения текущих значений баланса и максимально возможного кредита. Баланс не может быть отрицательным и превосходящим максимально возможный кредит по абсолютной величине.

Метод **`int`** `transfer()` осуществляет перевод денег со счета или на счет, в зависимости от знака своего аргумента. Если аргумент положителен, соответствующая сумма добавляется на счет, увеличивая его текущий баланс. Если отрицателен, эта сумма списывается со счета, если при этом баланс не выходит за рамки максимального кредита. Результат этого метода — переведенная сумма или 0, если перевод не был сделан.

Данный счет позволяет использовать специализированный валидатор транзакций, `Validator`, который опрашивается при любом переводе с помощью предоставляемого им метода **`boolean`** `validateTransfer(Account a, int sum)` и может разрешить или заблокировать перевод.

Еще одна функция счета — запись данных о попытках перевода денег в трассу для последующего аудита. При этом вызываются методы интерфейса `AuditLog`: `logKind(String s)`, `logOldBalance(int b)`, `logSum(int sum)`, `logNewBalance(int b)`, записывающие, соответственно, итог транзакции (SUCCESS в случае успешного перевода, BANNED в случае его блокировки валидатором, IMPROPER в случае попытки снятия слишком большой суммы), предшествующее значение баланса, переводимую сумму и новое значение баланса.

Модель поведения для счета описана в виде двух независимых компонентов: модели основной функциональности и модели работы с трассировкой переводов. Это позволяет изменять и проверять эти две группы ограничений независимо. Описание основной функциональности выглядит так.

```
public class AccountContract
{
  int balance;
  int maxCredit;

  Account checkedObject;
```

```
  public void setCheckedObject(Account
checkedObject)
  {
    this.checkedObject = checkedObject;
    this.balance     = checkedObject.getBalance();
    this.maxCredit = checkedObject.getMaxCredit();
  }

  public boolean possibleTransfer(int sum)
  {
    if (balance + sum > maxCredit) return true;
    else                           return false;
  }

  public boolean transferPostcondition(int sum)
  {
    boolean permission =
      checkedObject.getValidator()
     .validateTransfer(checkedObject, sum);

    if (Contract.oldBooleanValue
       (possibleTransfer(sum)) && permission)
      return

Contract.assertEqualsInt(Contract.intResult(), sum
       , "Result should be equal to the argument")
      && Contract.assertEqualsInt(balance,
Contract.oldIntValue(balance)+sum
       , "Balance should be increased on the
argument")
      && Contract.assertEqualsInt(maxCredit,
Contract.oldIntValue(maxCredit)
         , "Max credit should not change");
    else
      return

Contract.assertEqualsInt(Contract.intResult(), 0
          , "Result should be 0")
      && Contract.assertEqualsInt(balance,
Contract.oldIntValue(balance)
          , "Balance should not change")
      && Contract.assertEqualsInt(maxCredit,
Contract.oldIntValue(maxCredit)
            , "Max credit should not change");
```

```
    }

    public void transferUpdate(int sum)
    {
      if(   possibleTransfer(sum)
         && checkedObject.getValidator()
         .validateTransfer(checkedObject, sum))
        balance += sum;
    }
  }
```

Здесь показаны постусловие метода `transfer()` и соответствующий синхронизатор модельного состояния.

Описание требований к работе с трассой для аудита дано ниже. Оно использует свободно распространяемую библиотеку для организации заглушек Mockito, вставляя заглушку для наблюдения за сделанными вызовами между счетом и связанным с ним трассировщиком. В ходе работы заглушка проверяет, что методы трассировщика вызывались в нужном порядке и с нужными аргументами. Поскольку построенная заглушка имеет модельное состояние, в ней также определен метод-синхронизатор этого состояния. Заглушка должна инициализироваться после каждого вызова `transfer()`, для этого в ней определен метод `initSpy()`.

```
  public class AccountLogSpy
  {
    int balance;
    int maxCredit;

    Account checkedObject;
    AuditLog logSpy;

    public void setCheckedObject(Account
  checkedObject)
    {
      this.checkedObject = checkedObject;
      this.balance   = checkedObject.getBalance();
      this.maxCredit = checkedObject.getMaxCredit();
      logSpy = Mockito.spy(checkedObject.getLog());
      checkedObject.setLog(logSpy);
    }

    int oldBalance;
    boolean wasPossible;

    public boolean possibleTransfer(int sum)
```

```
    {
      if (balance + sum > maxCredit) return true;
      else                           return false;
    }

    public void initSpy(int sum)
    {
      Mockito.reset(logSpy);
      oldBalance = balance;
    }

    public void transferLogSpy(int sum)
    {
      boolean permission =
        checkedObject.getValidator()
       .validateTransfer(checkedObject, sum);

      if (wasPossible && permission)
      {
        Mockito.verify(logSpy).logKind("SUCCESS");
        Mockito.verify(logSpy).logNewBalance(balance);
      }
      else if (!permission)
        Mockito.verify(logSpy).logKind("BANNED");
      else
        Mockito.verify(logSpy).logKind("IMPROPER");

      Mockito.verify(logSpy).logOldBalance(oldBalance);
      Mockito.verify(logSpy).logSum(sum);
    }

    public void transferUpdate(int sum)
    {
      if(    possibleTransfer(sum)
         && checkedObject.getValidator()
         .validateTransfer(checkedObject, sum))
      {
        wasPossible = true;
        balance += sum;
      }
      else
        wasPossible = false;
    }
  }
```

Описание модели ситуаций представлено ниже. В ней ситуации классифицируются по четырем характеристикам: корректность перевода, прохождение валидации, знак предшествовавшего значения баланса и знак переводимой суммы. Поскольку определение ситуации зависит от модельного состояния счета и нуждается в синхронизаторе состояния, эта модель наследует модели функциональности, используя повторно определенные в ней элементы кода.

```
public class AccountCoverage extends
AccountContract
{
  public void transferCoverage(int sum)
  {
    boolean permission =
      checkedObject.getValidator()
      .validateTransfer(checkedObject, sum);

    if (possibleTransfer(sum))
Coverage.addDescriptor("Possible transfer");
    else
Coverage.addDescriptor("Too big sum");

    if (permission)
Coverage.addDescriptor("Permitted");
    else
Coverage.addDescriptor("Not permitted");

    if(balance == 0)
Coverage.addDescriptor("Zero balance");
    else if(balance > 0)
Coverage.addDescriptor("Positive balance");
    else
Coverage.addDescriptor("Negative balance");

    if(sum == 0)
Coverage.addDescriptor("Zero sum");
    else if(sum > 0)
Coverage.addDescriptor("Positive sum");
    else
Coverage.addDescriptor("Negative sum");
  }
}
```

Модель теста для счета выглядит следующим образом.

```
@Test public class AccountTest
{
```

```
  Account account;
  boolean permission = true;

  @Mock Validator validatorStub;

  public AccountTest()
  {
    MockitoAnnotations.initMocks(this);

Mockito.when(validatorStub.validateTransfer(Mockit
o.<Account>any()
          , Mockito.anyInt())).thenReturn(true);
  }

  public void setAccount(Account account)
  {
    this.account = account;
    account.setValidator(validatorStub);
  }

  public Validator getPermitterStub() { return
validatorStub; }

  @State public int getBalance() { return
account.getBalance(); }

  @State public boolean getPermission() { return
permission; }

  @Test
  @DataProvider(name = "sumArray")
  @Guard(name = "bound")
  public void testDeposit(int x)
  {
    account.transfer(x);
  }

  @Test
  @DataProvider(name = "sumIterator")
  public void testWithdraw(int x)
  {
    account.transfer(-x);
  }

  @Test
```

```
  @Guard(name = "bound")
  public void testIncrement()
  {
    account.transfer(1);
  }

  @Test
  public void switchPermission()
  {
    permission = !permission;
    Mockito.when(validatorStub.validateTransfer
    (Mockito.<Account>any()
      , Mockito.anyInt())).thenReturn(permission);
  }

  public boolean bound()
  {
    return getBalance() < 5 || !permission;
  }

  public int[] sumArray = new int[]{1, 2};

  public Iterator<Integer> sumIterator()
  {
    return
(Utils.ArrayToTypedList(sumArray)).iterator();
  }
}
```

Состояние теста состоит из двух элементов: текущего значения баланса и значения поля `permission`, определяющего результаты работы управляющей заглушки валидатора. Тестирование снятия денег и помещения их на счет разнесено по разным тестовым методам, хотя при этом вызывается один и тот же метод тестируемого объекта. Всего имеется четыре тестовых метода, соответствующих действиям в описываемом автомате.

- Метод `testDeposit()` проверяет помещение денег на счет. Он параметризован, значения параметров при работе теста берутся из массива `sumArray`. Кроме того, этот метод имеет охранное условие, позволяющее вызывать его только в тех случаях, когда текущий баланс не превосходит 5 и валидатор-заглушка допускает выполнение операций.
- Метод `testWithdraw()` проверяет снятие денег со счета. Значения его параметра берутся из того же массива, но с использованием метода-итератора.

- Метод `testIncrement()` проверяет добавления на счет суммы, равной 1. Он имеет то же самое охранное условие, что и метод `testDeposit()`.
- Метод `switchPermission()` ничего не проверяет, он только переключает текущее значение поля `permission`, чтобы протестировать работу счета с разными балансами и разными результатами валидации переводов.

Наконец, конфигурационный файл для среды Spring, определяющий связи между всеми перечисленными компонентами, выглядит так.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans
xmlns="http://www.springframework.org/schema/beans"
       xmlns:aop="http://www.springframework.org
       /schema/aop"
       xmlns:xsi="http://www.w3.org/2001
       /XMLSchema-instance"
       xsi:schemaLocation=
       "http://www.springframework.org/schema/beans
                  http://www.springframework.org
                  /schema/beans/spring-beans-3.0.xsd
                  http://www.springframework.org
                  /schema/aop
                  http://www.springframework.org
                  /schema/aop/spring-aop-2.5.xsd">

  <bean  id="accountImpl"
  class="mbtest.tests.AccountImpl"></bean>

  <bean  id="accountTest"
  class="mbtest.tests.AccountTest">
    <property name="account" ref="accountImpl"/>
  </bean>

  <bean  id="accountContract"
  class="mbtest.tests.AccountContract">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>

  <bean  id="accountCoverage"
  class="mbtest.tests.AccountCoverage">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>
```

```
  <bean  id="accountLogSpy"
  class="mbtest.tests.AccountLogSpy">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>

  <bean  id="accountContractExecutor"
  class="mbtest.contracts.ContractExecutor">
    <property name="postcondition"
              value="mbtest.tests.AccountContract
              .transferPostcondition"/>
    <property name="updater"
value="mbtest.tests.AccountContract.transferUpdate"/>
    <property name="object" ref="accountContract"/>
  </bean>

  <bean id="accountCoverageExecutor"
class="mbtest.coverage.CoverageExecutor">
    <property name="coverage"
              value="mbtest.tests.AccountCoverage
              .transferCoverage"/>
    <property name="updater" value="mbtest.tests
    .AccountCoverage.transferUpdate"/>
    <property name="object" ref="accountCoverage"/>
  </bean>

  <bean  id="accountSpyExecutor"
  class="mbtest.contracts.SpyExecutor">
    <property  name="initialization"
    value="mbtest.tests.AccountLogSpy.initSpy"/>
    <property name="postcondition"
              value="mbtest.tests.AccountLogSpy
              .transferLogSpy"/>
    <property  name="updater" value="mbtest.tests
    .AccountLogSpy.transferUpdate"/>
    <property name="object" ref="accountLogSpy"/>
  </bean>

  <aop:config>
    <aop:aspect  id="accountContractAspect"
    ref="accountContractExecutor">
      <aop:pointcut id="accoutTransfer"
                    expression="execution(* mbtest
                    .tests.Account.transfer(..))"/>
```

```
      <aop:around pointcut-ref=
      "accoutTransfer" method="execute"/>
    </aop:aspect>

    <aop:aspect  id=
    "accountCoverageAspect" ref=
    "accountCoverageExecutor">
      <aop:pointcut id="accoutCTransfer"
                    expression="execution(* mbtest
                    .tests.Account.transfer(..))"/>
      <aop:around  pointcut-ref=
      "accoutCTransfer" method="execute"/>
    </aop:aspect>

    <aop:aspect  id="accountSpyAspect"
    ref="accountSpyExecutor">
      <aop:pointcut id="accoutSTransfer"
                    expression="execution(* mbtest
                    .tests.Account.transfer(..))"/>
      <aop:around  pointcut-ref=
      "accoutSTransfer" method="execute"/>
    </aop:aspect>
  </aop:config>
</beans>
```

В этой конфигурации указано, как инициализировать объекты всех перечисленных типов, и, кроме того, определена привязка постусловий и синхронизаторов всех моделей к методу `transfer()` с помощью поддерживаемой Spring техники привязки аспектов.

Приведенный пример демонстрирует неинвазивность использованного метода построения тестовой системы из заданных компонентов — все эти компоненты ничего не знают друг о друге, кроме типов объектов, от которых они непосредственно зависят. В данной конфигурации модель основной функциональности и модель ситуаций представлены разными объектами, однако, поскольку вторая наследует первой, можно было бы реализовать их при помощи одного и того же компонента, играющего две разные роли.

5. Заключение

В работе представлена компонентная архитектура инструментария для тестирования на основе моделей, построенная на основе компонентных технологий с использованием принципа неинвазивной композиции. Описана реализация предложенного подхода на базе среды Spring, реализующей техники внедрения зависимостей. Кроме того, приведен пример его

использования для построения теста, включающего несколько моделей разных аспектов поведения тестируемого компонента.

Имеющаяся на настоящий момент реализация описываемого подхода содержит следующие недостатки, которые нужно устранить.

- Во-первых, нужно модифицировать стандартный контекст внедрения зависимостей в Spring, чтобы он распознавал специфичные для тестовых систем виды компонентов (модель поведения, заглушка, модель ситуаций, модель теста и пр.) и требовал меньше параметров для их инициализации, а также автоматически строил их аспектную привязку к тестируемым компонентам. Это позволит значительно упростить создание и модификацию конфигурационных файлов, удалив из приведенного выше примера почти весь текст в рамках элемента `<aop:config>`.

- Во-вторых, пока не реализованы инструменты для генерации вторичных компонентов, моделей ситуаций и моделей тестов. Предполагается разработать их на основе одной из открытых библиотек для трансформации байт-кода Java. Такая реализация сделает возможной генерацию вторичных компонентов без доступа к исходному коду их прообразов.

- В-третьих, логически различные элементы каркаса для построения тестовых систем — генераторы путей по автоматной модели, библиотечные генераторы данных, комбинаторы и пр. — также нужно выделить в виде внешне определяемых и конфигурируемых компонентов.

Однако уже сейчас предложенная архитектура демонстрирует свои основные достоинства по сравнению с традиционными «монолитными» инструментами построения тестов — высокую гибкость, возможность совместного использования с разнообразными библиотеками, многочисленными инструментами, предназначенными для работы с компонентами Java (средами разработки, анализаторами кода, отладчиками и т.д.), возможность интеграции в более мощные среды.

Литература

[1] C. Szyperski. Component Software: Beyond Object-Oriented Programming. 2-nd ed. Addison-Wesley Professional, Boston, 2002.

[2] G. T. Heineman, W. T. Councill. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, 2001.

[3] D. Parnas. Information Distribution Aspects of Design Methodology. Proc. of 1971 IFIP Congress, North Holland, 1971.

[4] G. Tassey, ed. The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Report, 2002.

[5] P. Hamill. Unit Test Frameworks. Tools for High-Quality Software Development. O'Reilly Media, 2004.

[6] http://www.junit.org.
[7] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (Eds.) Model-Based Testing of Reactive Systems. Advanced Lectures. LNCS 3472, Springer-Verlag, 2005.
[8] M. Utting, B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, 2007.
[9] D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Trans. on Software Engineering, 24(3):161–173, 1998.
[10] D. Hoffman. Analysis of a Taxonomy for Test Oracles. Quality Week, 1998.
[11] L. Baresi, M. Young. Test Oracles. Tech. Report CIS-TR-01-02. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.
[12] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231-274, June 1987.
[13] D. Drusinsky. Modeling and verification using UML statecharts. Elsevier, 2006
[14] R. Alur, D. L. Dill. A Theory of Timed Automata. Journal of Theoretical Computer Science, 126(2):183-235, 1994.
[15] J. Springintveld, F. Vaandrager, P. R. D'Argenio. Testing Timed Automata. Theoretical Computer Science, 254(1-2):225-257, March 2001.
[16] H. Zhu, P. Hall, J. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427, December 1997.
[17] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт 13, ИСП РАН, Москва, 2006.
[18] K. Beck. Kent Beck's Guide to Better Smalltalk: A Sorted Collection. Cambridge University Press, 1998.
[19] http://sunit.sourceforge.net/.
[20] C. Beust, H. Suleiman. Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley Professional, 2007.
[21] http://testng.org/.
[22] http://www.dbunit.org.
[23] http://www.httpunit.org.
[24] http://jbehave.org/.
[25] http://nspecify.sourceforge.net/.
[26] http://mockito.org/.
[27] http://easymock.org/.
[28] J. Tretmans, E. Brinksma. TorX: Automated Model-Based Testing. Proc. of 1-st European Conference on Model-Driven Software Engineering, Nuremberg, Germany. pp. 31-43, December 2003.
[29] http://fmt.cs.utwente.nl/tools/torx/introduction.html.
[30] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, C. Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. Proc of 8-th International Conference on Computer-Aided Verification, LNCS 1102:348-359, Springer, 1996.
[31] http://www.inrialpes.fr/vasy/cadp/man/tgv.html.
[32] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, M. Utting. Z-TT: A tool-set for test generation from Z and B using constraint logic programming, Proc. of Formal Approaches to Testing of Software, pp. 105-119, Brno, Czech Republic, August 2002.
[33] A. Hartman, K. Nagin. TCBeans Software Test Toolkit. Proc. of 12-th International Software Quality Week, May 1999.
[34] E. Farchi, A. Hartman, S. S. Pinter. Using a model-based test generator to test for standard conformance. IBM Systems Journal, 41(1):89-110, 2002.

[35] http://www.conformiq.com/qtronic.php.
[36] http://www.smartesting.com/index.php/cms/en/explore/products.
[37] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391:77-88, Springer, 2002.
[38] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
[39] http://www.unitesk.ru.
[40] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes. Testing Concurrent Object-Oriented Systems with Spec Explorerю Proc. of Formal Methods Europe, LNCS582:542-547, Springer, 2005.
[41] http://research.microsoft.com/en-us/projects/SpecExplorer/.
[42] http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/.
[43] J. Jacky, M. Veanes, C. Campbell, W. Schulte. Model-based Software Testing and Analysis with C#. Cambridge University Press, 2007.
[44] http://nmodel.codeplex.com/.
[45] http://mbt.tigris.org/.
[46] M. Barnett, M. Fahndrich, P. de Halleux, F. Logozzo, N. Tillmann. Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract. Proc. of ICSE 2009, Vancouver, Canada, May 2009.
[47] http://research.microsoft.com/en-us/projects/contracts/.
[48] C. Kaner, J. Bach, B. Pettichord. Lessons Learned in Software Testing. John Wiley & Sons, 2002.
[49] В. В. Кулямин. Интеграция методов верификации программных систем. Программирование, 35(4):41-55, 2009.
[50] V. Kuliamin, A. Petrenko, N. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proc. of ISAS'2005, Berlin, Germany. M. Malek, E. Nett, N. Suri , eds. Service Availability. LNCS 3694, pp. 68-83, Springer-Verlag, 2005.
[51] A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov. Formal Methods in Industrial Software Standards Enforcement. Proc. of PSI'2006, Novosibirsk, Russia, 2006.
[52] M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern. 2004. http://www.martinfowler.com/articles/injection.html.
[53] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu. Professional Java Development with the Spring Framework. Wrox, 2005.
[54] http://www.springsource.org.

Технология создания гетерогенных трасс, их анализа и генерации из них отчётов

С. Г. Грошев
sgroshev@ispras.ru

Аннотация. В статье описывается архитектура модульной системы трассировки (журнализации, логгирования) и анализа трасс, поддерживающей трассировку сложноструктурированной информации, расширение видов трассируемой информации, изоляцию и выборку нужных данных при анализе полученных трасс и облегчающей связывание разнородных данных между собой. Рассматриваются методы связывания данных, в том числе разнородных, по содержанию и по времени, а также дальнейшего преобразования их в разнообразные отчёты. Описывается архитектура генератора отчётов, позволяющая собирать отчёты из независимых модулей и связывать извлечённые ими данные.

1. Введение

Для удовлетворения потребностей пользователей и поддержки стабильного развития современного общества необходимо разрабатывать всё более сложное программное обеспечение (ПО). С ростом сложности ПО увеличивается и сложность его тестирования.

До настоящего времени широко применяются методики тестирования, при которых тесты выдают в конце работы только вердикт вида «PASS» или «FAIL»; при этом предполагается, что в случае неуспешного прохождения какого-либо теста можно легко локализовать и исправить соответствующую ошибку. Однако при исчерпывающем тестировании сложных систем, особенно состоящих из множества взаимодействующих компонентов, этого обычно недостаточно. Поэтому необходимо в ходе тестирования собирать, а после завершения теста анализировать гораздо больше разнообразной информации, например следующей.

- Состояния тестируемой системы в целом или её отдельных компонентов, осуществляемые тестовой системой воздействия на неё и переданные при этом тестовые данные – для анализа ситуаций, в которых возникли ошибки, а также для оценки тестового покрытия;
- Обмен данными между компонентами тестируемой системы;

- Информация о критических секциях и синхронизации параллельно работающих компонентов;
- Сообщения об ошибках (как от самой тестируемой системы, так и от тестов, считающих её поведение некорректным);
- Разнообразные метрики тестового покрытия (по входным данным, по исходному коду, по интерфейсным вызовам, по сценариям использования и так далее) – для оценки полноты тестирования;
- Другие данные, которые могут понадобиться при анализе поведения разрабатываемой системы.

Процесс сбора информации о ходе работы программы и/или тестов для неё называется трассировкой, а собранные данные, упорядоченные по времени возникновения соответствующих событий, – трассой. Отметим сразу, что во многих русскоязычных источниках словом «трассировка» обозначается пошаговая отладка программ (в качестве отражения тенденции можно привести, например, определение слова «Трассировка» в русскоязычной википедии [1]); в настоящей статье под трассировкой понимается то же, что в англоязычных источниках: журнализация (logging) событий, происходящих при работе программы или теста для неё, высокой степени детализации, рассчитанная на то, что какая-то (иногда заранее неизвестно, какая) часть этой информации может понадобиться для анализа поведения этой программы (сравним с определением слова «Tracing» в англоязычной википедии [2]).

Во многих широко используемых сегодня технологиях разработки ПО (например, Rational Unified Process [3], eXtreme Programming [4], а также методиках разработки, известных под общим названием Agile Software Development [5]) разрабатываемая система создаётся итеративно, при этом тесты для неё разрабатываются параллельно с ней, усложняясь по мере её развития. Для оценки правильности поведения тестируемой системы необходимо собирать в ходе тестирования информацию о различных аспектах её поведения, при этом по мере её развития могут добавляться новые виды информации.

Жизненный цикл трассы состоит из двух основных фаз: создание трассы и её анализ; вспомогательные процессы этого жизненного цикла, такие как конфигурирование трассировки, хранение полученных трасс и так далее, в настоящей статье не рассматриваются. На практике фаза анализа может опускаться: в этом случае, например, логи сервера анализируются только в случае обнаружения каких-либо проблем в его работе. Описываемая технология трассировки и анализа трасс разработана для случаев, когда анализ полученных в результате работы целевого приложения (или тестов для него) трасс важен, и основная цель использования системы трассировки – получение неких отчётов в результате анализа трасс. Соответственно, мы рассматриваем нужды двух приложений, использующих описываемую

технологию: генерирующего трассы (также называемого, в зависимости от контекста, целевой системой; в случае тестирования, с точки зрения жизненного цикла трассы целевой системой будет объединение тестовой и тестируемой систем) и анализирующего их (также называемого генератором отчётов).

Как видно из вышесказанного, собираемая в виде трасс информация о поведении целевой системы может быть чрезвычайно разнообразной, причём её состав может часто меняться (в этом её отличие, например, от лога работы веб-сервера, содержащего только сообщения фиксированного формата об обработанных запросах). Такие данные и содержащие их трассы мы называем гетерогенными; они требуют особых способов генерации и анализа.

В настоящей статье рассматриваются:

1. Архитектура системы, предназначенной для создания гетерогенных трасс и облегчающей их анализ.
2. Методы анализа гетерогенных данных.
3. Архитектура генератора отчётов, позволяющего объединять результаты анализа данных, проведённого различными аналитическими модулями, и представлять их в удобном для человека виде.

2. Архитектура системы трассировки и анализа трасс

2.1. Требования

Для решения задач, встающих при работе с гетерогенными трассами, были выделены следующие требования, которым должна удовлетворять система поддержки создания и анализа гетерогенных трасс:

- Универсальность.

Решение не должно быть привязано к какой-либо предметной области, классу приложений, способу тестирования (и вообще к задачам тестирования), языку программирования, платформе или способу хранения данных. Предоставляемая функциональность должна обеспечивать применимость в широком круге задач, в которых имеются аналогичные потребности мониторинга поведения сложных систем и анализа полученных разнородных данных.

- Расширяемость и модульность.

Структура трассы и архитектура инструментов её генерации и обработки должны без существенных накладных расходов допускать добавление новых видов отслеживаемых событий. При этом:

- Инструменты, анализирующие трассу, должны иметь возможность получать только существенную для них информацию, игнорируя всё остальное эффективно (как по скорости обработки, так и по используемой памяти) и без потери согласованности данных.
- Необходимы средства, позволяющие легко добавлять функциональность анализа новых видов информации к уже существующим генераторам отчётов. Для решения этой задачи нужна инфраструктура, позволяющая собирать инструменты анализа трасс из слабо связанных анализирующих компонентов и автоматически связывать информацию, собранную этими компонентами, между собой.

- Возможность работы со сложными структурами данных.

Необходима возможность сохранять в трассе информацию о сложных объектах, с которыми работает целевая система, и потом легко восстанавливать их структуру. Эта задача особенно актуальна при тестировании программных интерфейсов, работающих с такими объектами.

- Унификация.

Для снижения трудоемкости разработки инструментов анализа и обеспечения возможности повторного использования их кода, схожие виды информации должны сбрасываться в трассу в одинаковом формате.

В современной программной индустрии уже разработано множество решений для мониторинга работы приложений (особенно хотим отметить SystemTap [6] – проект с открытым кодом, позволяющий динамически инструментировать ядро операционной системы для трассировки выбранных системных событий), однако в большинстве случаев они концентрируются на задачах сбора информации с работающей системы, а не на самой этой информации. При поиске подходящих уже существующих решений мониторинга с возможностью расширения видов собираемых данных были также найдены два проекта, имеющих одинаковое название «Расширяемый формат лога» (XLF, eXtensible Log Format). Однако несмотря на заявленную расширяемость, оба они работали с достаточно примитивными и малорасширяемыми данными: один из них [7] разрабатывался как замена для syslog, другой – для ведения логов запросов веб-серверов; то есть, степень расширяемости логов обоих этих проектов была явно недостаточной для поставленных задач.

По результатам исследований для решения поставленных задач нами была разработана система трассировки и анализа трасс Aspectrace [8]. В данном разделе описывается её архитектура и предоставляемые ею возможности.

2.2. Структура трассы

Трасса состоит из последовательных сообщений.

Сообщение – минимальная единица информации в трассе. Каждое сообщение имеет заголовок и внутренние данные. Формат заголовка фиксирован: он содержит информацию об аспекте сообщения, его типе и канале. Возможные типы сообщений определяются аспектом, а синтаксическая структура внутренних данных сообщения определяется его аспектом и типом. В качестве базовой структуры внутренних данных сообщений выбраны атрибутированные деревья, на которые аспекты накладывают собственные синтаксические и семантические ограничения.

Канал представляет собой источник последовательных сообщений, гарантирующий на них линейный порядок. Например, разные каналы могут представлять собой источники сообщений, расположенные на разных машинах в сети или в разных нитях (threads) управления. Каждый канал имеет уникальный идентификатор, отличающийся от идентификаторов других каналов. В трассе могут содержаться вперемешку сообщения из разных каналов; при этом в общем случае относительный порядок сообщений, относящихся к разным каналам, недостоверен. В трассе также может содержаться информация, позволяющая упорядочивать сообщения из разных каналов, в том числе предназначенные специально для этой цели служебные сообщения.

Аспекты представляют собой различные срезы информации о целевой системе. Каждый аспект определяет собственную модель данных, допустимые виды сообщений, их синтаксические структуры и семантические ограничения на них. Также определены несколько видов служебных сообщений; они отнесены к особому служебному аспекту. С точки зрения программиста, использующего данную технологию, каждый аспект представлен подключаемым модулем, предоставляющим интерфейсы для работы с соответствующей моделью данных, сброса в трассу предусмотренных аспектом типов сообщений и для получения соответствующих сообщений при анализе трасс.

Различные компоненты целевой системы в ходе работы могут независимо друг от друга сбрасывать в трассу разнообразную информацию, относящуюся к различным аспектам её поведения. Особенно это существенно при тестировании, когда надо одновременно следить за работой и тестируемой системы (которая может содержать отладочный код, сбрасывающий внутреннюю информацию о своей работе, как изначально, так и в результате инструментирования, осуществлённого для целей тестирования), и тестовых компонентов (генераторов тестовых данных, тестовых оракулов, вспомогательных компонентов и так далее). Как уже отмечалось выше, по мере усложнения целевой системы добавляются новые виды информации, которые необходимо учитывать; это создаёт проблемы при любом фиксированном формате трассы, так как при расширении набора собираемой информации теряется совместимость по данным между целевыми и анализирующими приложениями. В технологии Aspectrace возможность гибко

добавлять в трассу новые виды информации обеспечивается с помощью расширяемого набора независимых аспектов. При возникновении необходимости собирать и анализировать информацию какого-либо нового вида, разработчики добавляют в целевую систему трассировку относящейся к соответствующим аспектам информации, используя соответствующие готовые библиотеки аспектов или разрабатывая новые. Параллельно с этим разрабатываются специализированные инструменты, анализирующие эту информацию, или добавляются соответствующие возможности к уже имеющимся инструментам анализа. Каждый инструмент, анализирующий трассу, нацелен на обработку определённых аспектов; при этом каждый такой инструмент выбирает из трассы только информацию от тех аспектов, которые он умеет обрабатывать, и которые нужны для построения запрошенного отчёта, игнорируя всё остальное.

Поскольку информация от разных аспектов сбрасывается в трассу и извлекается из неё при анализе независимо, процессы разработки создающих трассу приложений и инструментов для их анализа могут идти в значительной степени независимо, при этом совместимость между различными версиями трасс и инструментов их анализа сохраняется в обе стороны. Так, если целевая система добавляет трассировку новой информации и обгоняет инструменты анализа, то старые инструменты анализа просто игнорируют эту информацию и анализируют только то, что могут; если же инструменты анализа обгоняют целевую систему, то они просто не получают информацию соответствующего вида и генерируют отчёт только по тем данным, которые им предоставлены.

2.3. Компонентная архитектура

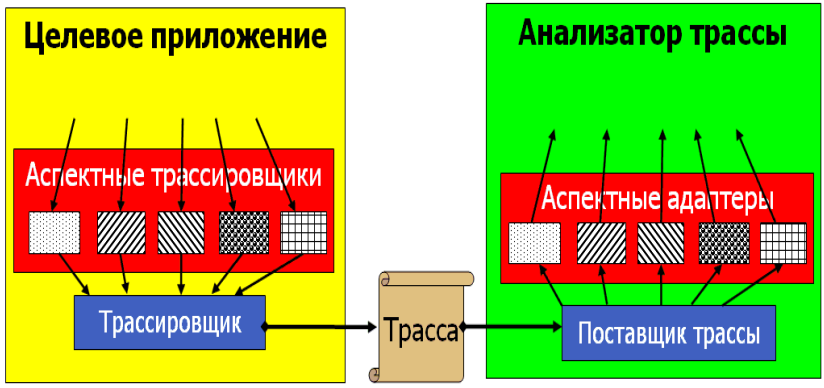


Рис. 1. Поток данных через компоненты.

На рис. 1 изображена структура компонентов системы трассировки и анализа трасс и способы её использования приложениями: как генерирующими трассы, так и анализирующими их.

Трассировщик и *поставщик трассы* – центральные инфраструктурные компоненты, поддерживающие расширяемый формат трассы и управляющие обработкой трасс. Они не поддерживают никаких моделей данных, обладающих самостоятельной ценностью для разработчика или аналитика, использующего трассы для мониторинга целевой системы; для сброса информации в трассу и выборки её оттуда приложения взаимодействуют с ними не напрямую, а через библиотеки аспектов.

Аспектные трассировщики и *аспектные адаптеры* входят в соответствующие библиотеки аспектов (на рисунке одинаковые аспекты обозначены одинаковыми текстурами) и подключаются приложениями по необходимости, независимо друг от друга. Аспектные трассировщики предоставляют приложениям, генерирующим трассы, интерфейсы для сброса сообщений в терминах модели данных своего аспекта; полученные данные преобразуются в универсальное промежуточное представление и передаются трасировщику. Аспектные адаптеры получают от поставщика трассы данные в промежуточном представлении, проверяют их на соответствие определённым в аспекте синтаксическим и семантическим ограничениям, преобразуют в модель данных аспекта и передают их анализирующим компонентам, которые для этого подписываются на получение от них соответствующих сообщений. В случае, если для какого-то аспекта не зарегистрировано ни одного подписчика, сообщения этого аспекта игнорируются на уровне поставщика трассы и не подвергаются дальнейшей обработке.

Для передачи оттрассированной информации от целевого приложения к анализатору трассы, а также для хранения трасс используются внешние представления трассы. Внешнее представление может быть, например:

- XML-файлом, хранящимся в файловой системе;
- Схемой базы данных под управлением внешней СУБД;
- Сетевым соединением, по которому передаются сообщения между брокерами объектных запросов (ORB) общего назначения с помощью имеющихся у них средств сериализации сложных объектов (COM, CORBA, Java/RMI и другие);
- Прямой связью между генератором и анализатором трассы, работающими в одном адресном пространстве, по которой ссылки в памяти на трассируемые сообщения передаются напрямую, без дополнительных преобразований (строго говоря, такое представление не является «внешним»);
- Любым другим каналом, допускающим передачу от целевой системы к анализатору трассы структурированных сообщений.

Различные внешние представления трассы поддерживаются соответствующими подключаемыми модулями, которые разработчики

системы могут подбирать в зависимости от особенностей окружения, в котором работают генераторы и анализаторы трассы.

3. Анализ трасс

В ходе работы приложения, генерирующего трассу, различные его компоненты сбрасывают информацию, относящуюся к различным аспектам его поведения. Компоненты генерирующей трассу системы могут разрабатываться и, соответственно, трассировать нужную им информацию независимо друг от друга, а могут иметь тесно связанную логику трассировки.

Получающаяся трасса похожа по структуре на медиаконтейнеры. Особо следует упомянуть контейнер Matroska («Матрёшка») [9] – открытый расширяемый формат контейнера, позволяющий добавление в медиафайл новых видов информации. В современных медиафайлах могут идти вперемешку блоки информации, описывающие, например, видео, звуковые дорожки, субтитры на разных языках, разбиение на разделы, синхронизационные метки и многое другое. Вся эта информация, представленная идущими последовательно разнородными блоками, с одной стороны, логически разделяется на идущие параллельно потоки (например, поток видео и поток звуковой дорожки), а с другой – как-то связана между этими потоками. Простейший механизм связывания информации между потоками – синхронизационные метки, но возможны и более сложные зависимости: например, звуковой поток и поток субтитров могут быть помечены одним и тем же языком, и проигрыватель может учитывать эту зависимость. Приложения, воспроизводящие информацию из медиаконтейнеров, используют соответствующие кодеки, чтобы извлекать и обрабатывать эти независимые потоки, после чего самостоятельно связывают их между собой и преобразуют всё вместе в понятный для человека вид (например, в видеоряд).

Примерно таков же принцип работы анализаторов трасс в архитектуре Aspectrace: они извлекают с помощью аспектных адаптеров нужные им поаспектные (а также поканальные) потоки сообщений, на анализ которых они нацелены, после чего анализируют их, находят связи между ними в соответствии с заложенной в эти анализаторы логикой и преобразуют всё вместе в понятный для человека вид.

Одни из этих связей могут быть определены структурой предметной области: например, сообщение в трассе об отправке сетевого пакета одним компонентом и сообщение об его получении другим должны содержать один и тот же идентификатор этого пакета. Другие связи могут быть заложены на этапе проектирования генерирующих трассу компонентов: например, если мы собираем информацию о покрытии исходного кода, то она будет коррелировать с информацией о вызовах методов, так как после вызова метода мы попадаем в соответствующий участок кода, и в этом случае имеет смысл сбрасывать в трассу сообщения о покрытии кода так, чтобы они

содержали указание на метод, в котором находится соответствующий код. Если нам известны связи этих двух видов, то в большинстве случаев разумно закладывать информацию о них в анализаторы трассы.

Третьи связи могут быть вообще не предусмотрены разработчиками приложений, генерирующих трассы, и именно их поиск будет целью трассировки и последующего анализа трасс: например, это могут быть корреляции между сообщениями об ошибках, сбрасываемыми в трассу одними компонентами целевой системы, и сообщениями о выполняемых действиях, сбрасываемыми другими компонентами.

3.1. Связывание плоских данных

Под «плоскими» здесь и далее мы понимаем данные, при анализе связей между которыми не учитывается время. Методы связывания данных на основе времени рассматриваются в следующем разделе.

Основные способы связывания и агрегирования плоских данных в целом известны нам из теории реляционных баз данных [10]. Для этих методик анализа неважно, являются ли данные однородными (относящимися к одному аспекту) или разнородными. Хотя каждый аспект предоставляет собственную модель данных, не зависящую от моделей данных других аспектов, использующее их приложение может строить над ними собственную модель, в которой присутствуют дополнительные межаспектные связи. Например, если мы оцениваем покрытие по вызовам тестируемых методов, то каждый вызов тестируемого метода (который может быть важен и для других видов анализа) также рассматривается как достижение соответствующего элемента покрытия; в этом случае при вызове метода в трассу сбрасываются (возможно, различными компонентами тестовой системы) два сообщения: сообщение о вызове и его параметрах в аспекте «вызовы методов» и сообщение о достижении элемента покрытия в аспекте «покрытия». Эти два сообщения содержат данные, которые позволяют при анализе связать их между собой; например, соответствующие методу сообщения о покрытии могут содержать указание на покрытие по вызовам методов и сигнатуру вызванного метода. В архитектуре Aspectrace нет таких жёстких связей между данными из разных аспектов, как, например, между соответствующими первичным и внешним ключами различных таблиц в реляционных БД, поэтому возможна полная изоляция данных из разных аспектов и независимый их анализ. Так, в данном примере генератор отчёта о покрытиях, ничего не знающий про методы, может, тем не менее, сгенерировать осмысленный отчёт о достигнутом покрытии. Аналогично, генератор отчёта о вызванных методах может сгенерировать осмысленный отчёт о них, игнорируя сведения о покрытиях. Однако анализатор трассы, понимающий данные обоих аспектов и знающий про эти связи, может пользоваться ими и анализировать более сложные межаспектные зависимости; в результате, например, отчёт о покрытии вызовов метода может, дополнительно к однообразной для всех отчётов о

покрытии информации, также ссылаться на дополнительную информацию об этом методе и его вызовах.

Кратко перечислим основные способы связывания плоской информации:

- Восстановление ссылок между данными сообщений (в том числе относящихся к разным аспектам). Например, описанный в примере выше в этом разделе пример со ссылками из данных аспекта «покрытие» на данные аспекта «вызовы методов», где ссылочным ключом служит сигнатура метода, размещённая в сообщении о достижении элемента покрытия так, что её могут извлечь соответствующие анализаторы трассы. Аналогом такого связывания в реляционной алгебре являются операции соединения [10].

- Простая агрегация. Например, генератор отчёта может собирать информацию о количестве вызовов указанного метода, или о времени, проведённом в его вызовах, или о наборе достигнутых и недостигнутых элементов описанной в той же трассе структуры покрытия.

- Связывание вида «класс–экземпляр». Например, в трассе может содержаться информация о вызовах метода, которые для данного связывания являются «экземплярами», а информация о самом методе является «классом». Такое связывание также является разновидностью агрегации, но имеет дополнительные важные свойства: во-первых, информация о классе может присутствовать в трассе, а может отсутствовать в ней в явном виде и восстанавливаться анализатором только на основе информации об экземплярах (по сути такое связывание аналогично запросам вида `SELECT … GROUP BY` языка SQL); во-вторых, при данном виде анализа в модели анализатора существуют и класс как целое, и отдельные его экземпляры, при этом класс и экземпляры могут ссылаться друг на друга.

3.2. Модельное время

Трасса содержит в себе не только плоские, но и темпоральные данные, которые также могут быть существенны при её анализе. Приложение, генерирующее трассу, сбрасывает в неё информацию в виде отдельных сообщений; информация из трассы поступает на вход анализатору также в виде последовательности сообщений. И отправка сообщения в трассу, и получение его оттуда анализатором являются атомарными одномоментными событиями, однако генерирующее эти сообщения приложение исходит из некоторой модели данных, сущности в которой могут иметь совсем другую структуру связей со временем и между собой, чем простая последовательность событий. Анализатор трассы, получая через трассу сообщения от анализируемого приложения, в большинстве случаев строит аналогичную

модель данных и пытается восстановить эти сущности (разумеется, если его вообще интересует информация такого вида).

В настоящем разделе мы рассматриваем темпоральные характеристики сущностей модели на примере аспекта, описывающего вызовы методов тестируемой системы. В модели данных этого аспекта могут присутствовать: методы, которые могут быть вызваны; их сигнатуры; вызовы и их параметры; возвращаемая информация и выбрасываемые исключения; стек вызовов; разнообразная дополнительная информация об этих методах.

Темпоральные шкалы каналов трассы соответствуют темпоральным шкалам потоков управления приложения, генерирующего трассу. Время, в котором работает анализатор трассы, для настоящего исследования несущественно, поэтому далее мы везде говорим о темпоральной шкале трассы, учитывая при необходимости, что может параллельно существовать несколько таких шкал, соответствующих разным каналам. Отметим также, что в трассе могут вообще отсутствовать какие-либо темпоральные метки; в этом случае в качестве таковых можно рассматривать, например, порядковые номера сообщений в канале. Такая темпоральная шкала, разумеется, не подходит для анализа производительности или временны**х** допусков на выполнение критичных действий, однако, обеспечивает анализатору линейную упорядоченность событий в рамках канала, позволяя таким образом анализировать взаимное расположение событий во времени, а также проверять условия темпоральной логики.

В результате исследований мы выделили следующие классы модельных сущностей в зависимости от их связи с модельным временем:

- Одномоментные события. К таким сущностям можно привязать единственную темпоральную метку (в качестве таковой обычно удобно использовать метку сообщения трассы, обозначающего наступление соответствующего события). Например, одномоментной сущностью может моделироваться событие вызова метода с возвратом, если продолжительность этого вызова и произошедшие внутри него события несущественны.

- Продолжительные события. Такие сущности имеют время начала и конца (в качестве которых обычно берутся темпоральные метки сообщений трассы, обозначающих соответственно начало и конец события), в результате к ним оказываются привязаны интервалы на темпоральной шкале, которые могут частично перекрываться, а также строго включать друг друга и темпоральные метки одномоментных событий. Например, продолжительными сущностями моделируются вызовы методов от момента собственно вызова до момента возврата (тем или иным образом), если в промежутке могут происходить другие вызовы, которые нас интересуют; в этом случае мы можем легко анализировать вложенность вызовов методов друг в друга.

- Глобальные. Такие сущности не имеют отношения к каким-либо моментам времени (несмотря на то, что конкретные сообщения, содержащие информацию о них, могут иметь темпоральные метки). Например, глобальной сущностью в модели является сам вызываемый метод; он может иметь сигнатуру, ссылку на исходный код и другие атрибуты, но все они существуют вне времени.

Отметим также, что существуют такие языки и платформы, в которых методы создаются и уничтожаются динамически, а также модели для них, в которой время существования методов важно – в этом случае в соответствующей модели метод перестаёт быть глобальной сущностью, а становится такой же продолжительной, как и его вызов. В общем случае темпоральная характеристика сущностей модели зависит от того, какие свойства моделируемой области учитываются в используемой при анализе модели.

3.3. Связывание данных по времени

Для анализа данных, полученных из одного канала трассы, наиболее широко применяются такие темпоральные характеристики событий трассы как последовательность событий и вложенность в продолжительные события одномоментных и других продолжительных событий. Поскольку темпоральная шкала не зависит от аспекта, возможно автоматическое построение связей между данными из разных аспектов, причём для этого не нужны компоненты анализатора, понимающие модели данных сразу от обоих аспектов, а достаточно двух независимых компонентов, каждый из которых нацелен на обработку данных от одного аспекта, и соответствующим образом настроенной связи между этими компонентами (подробнее механизмы такого связывания описаны в разделе «Генерация отчётов»).

Рассмотрим такое связывание на примере простой трассы. Допустим, у нас есть тестовые компоненты, сбрасывающие сообщения в аспектах «тест» (о выполняемых тестовых сценариях), «метод» (о запусках методов теста и их завершении), «покрытие» (о достижении элементов покрытия) и «ошибка»; а также тестируемая система, инструментированная (например, с помощью AspectJ [11]) для сбрасывания в аспекте «метод» сообщений о вызовах своих методов и возвратах из них. В таблице 1 приведён простой пример трассы, которая может получиться при работе такого теста. В колонке со скобками в заголовке обозначены точки начала и конца продолжительных событий.

№	()	Аспект: сообщение
1	<	Тест: Начало теста t1
2	<	метод: Вызов метода testf(1,2,3)
3		покрытие: Достигнут элемент покрытия «хорошие данные»
4	<	метод: Вызов метода f(4,5,6)
5	>	метод: Возврат из метода f() с возвращаемым значением «0»
6	>	метод: Возврат из метода testf() с возвращаемым значением «0»

7	>	тест: Конец теста t1
8	<	тест: Начало теста t2
9	<	метод: Вызов метода testf(7,8,9)
10		покрытие: Достигнут элемент покрытия «плохие данные»
11	<	метод: Вызов метода f(10,11,12)
12	>	метод: Возврат из метода f() с исключением «NullPointerException»
13		ошибка: Ошибка в аспекте «метод», текст = «null указатель»
14	>	метод: Возврат из метода testf() с возвращаемым значением «1»
15	>	тест: Конец теста t2

Таблица 1. Пример трассы.

В данном примере ни тестируемый компонент, ни встроенный в него код трассировки ничего не знают о тестах, которые над ними выполняются; тест также может состоять из независимых модулей, выполняющих тестовые сценарии, определяющих достижение интересующих тестировщиков тестовых ситуаций, выполняющих другие действия и сбрасывающих обо всех этих событиях сообщения в трассу. Однако за счёт вложенности событий трассы друг в друга мы можем при её анализе легко соотнести обнаруженные ошибки и достигнутые элементы покрытия с тестом, в ходе выполнения которого они произошли, а также с вызовами тестовых и тестируемых методов и параметрами этих вызовов.

Описанные в разделе 3.1 методы связывания данных действуют в пределах одного канала, когда из трассы естественным путём извлекается информация о последовательности событий. В случае наличия нескольких каналов зависимости между данными могут быть не столь однозначны. Не существует общих способов восстановления последовательностей событий в таких целевых системах, пригодных для всех случаев; однако, существуют широко используемые (в частности, в технологиях разработки и анализа телекоммуникационных систем) и пригодные для достаточно широкого класса задач способы межканального связывания данных. Поскольку данная статья посвящена работе с гетерогенными данными, а не вопросам темпорального анализа, позволяющего связывать между собой однородные данные в различных временны**х** шкалах, мы не будем рассматривать здесь методы темпорального анализа более подробно, а ограничимся перечислением простейших способов такого связывания:

- Связывание по общему таймеру. Например, такой таймер может быть доступен различным процессам, выполняющимся на одной машине.
- Связывание по общим синхронизационным меткам. Например, если несколько компонентов в сети совместно выполнили протокол синхронизации (например, MPI Barrier [12]), и сбросили в трассу соответствующие сообщения, то мы можем с уверенностью считать, что все события, сообщения о которых находятся в любом из этих

каналов канале до синхронизации, произошли раньше, чем все события, произошедшие в любом из этих каналов после неё.

- Связь по данным сообщений. Во многих случаях возможно уникально идентифицировать сообщения, передаваемые в распределённой системе, и таким образом связывать события передачи сообщений в одних каналах трассы и события их получения в других каналах; в частности, события передачи сообщений заведомо происходят раньше событий их приёма.

4. Генерация отчётов

4.1. Задачи генератора отчётов и требования к нему

Обрабатывая трассу, анализатор строит некоторую модель извлечённых из неё данных. При этом он может не только восстанавливать те модели данных, которыми пользовалось создавшее трассу приложение, но и синтезировать на их основе свои собственные; в предыдущем разделе перечислены некоторые способы такого синтеза данных, в том числе относящихся к различным предметным областям и различным временны**ы**м шкалам. Поскольку одна и та же задача извлечения определённых данных, их анализа, преобразования и связывания между собой определённым способом может вставать при разработке разнообразных отчётов, возникает потребность в инфраструктуре, позволяющей реализовывать решение таких аналитических задач с помощью переиспользуемых компонентов, а затем с невысокими накладными расходами собирать из них готовые генераторы отчётов, в которых построенные этими компонентами модели данных автоматически связывались бы между собой.

После того как данные извлечены, проанализированы и связаны, их необходимо представить в виде, понятном разработчику или аналитику целевой системы – построить отчёт. Отчёт может быть представлен в виде графиков, таблиц, гипертекстовых страниц, изображающей поведение анализируемой системы анимации и во многих других формах. В любом случае, нужна инфраструктура, поддерживающая работу с выбранной формой представления данных. Важным требованием к такой инфраструктуре является возможность автоматически переводить связи (возможно, разных типов) между элементами модели данных в связи между соответствующими элементами представления.

Ниже описана архитектура генератора отчётов, позволяющая решать обе поставленные задачи. Для примеров используется генератор HTML-отчётов, но описанные принципы применимы и к другим формам представления данных.

4.2. Компонентная архитектура генератора отчётов

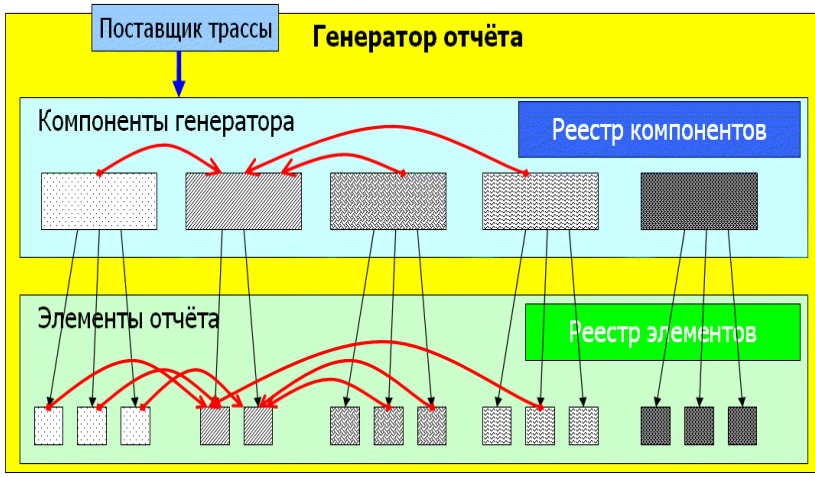


Рис. 2. Компоненты генератора и элементы отчёта.

На рисунке 2 изображена компонентная структура генератора отчётов. Ядро генератора отчётов предоставляет разработчику отчёта интерфейс для подключения к нему компонентов генератора и настройки связей определённых в нём видов между ними, а подключенным компонентам – интерфейс для добавления в отчёт элементов.

Компоненты генератора (дальше называемые также просто «компонентами») являются анализаторами трассы: они подсоединяются к общему поставщику трассы, получают из него сообщения от нужных им аспектов и анализируют их. На основе полученных данных они строят элементы отчёта соответствующих генератору видов (например, в случае генератора HTML-отчётов это будут целые HTML-страницы или их части) и добавляют их в отчёт. Компоненты могут предоставлять интерфейсы для установления связей с другими компонентами, а также для запроса контролируемых ими элементов отчёта.

Генератор отчётов ведёт реестры своих компонентов и элементов отчёта. Разные реестры могут поддерживать разные пространства имён и разные способы поиска по этим именам и связывания хранящихся в них сущностей (компонентов генератора или извлечённой ими информации). Компоненты генератора могут вести собственные реестры, хранящие информацию о созданных ими элементах отчёта, и предоставлять другим компонентам возможность осуществлять запросы к ним напрямую или подсоединять их к реестрам-посредникам ядра генератора, диспетчеризующим обращённые к ним запросы в соответствии со своей конфигурацией.

Компоненты генератора могут также регистрироваться в существующих реестрах компонентов, находить друг друга через них и запрашивать у найденных компонентов (формально говоря, у реестров, которые те ведут) нужные им элементы отчёта. Компоненты, между которыми установлены связи, с их помощью устанавливают соответствующие связи между своими элементами отчёта; на рисунке 2 такие связи обозначены дуговыми стрелками. Также ядро генератора отчётов может самостоятельно связывать между собой включённые в отчёт элементы, в соответствии со своими настройками и прикреплённой к этим элементам мета-информацией.

Некоторые способы связывания элементов отчёта описаны концептуально и на примерах ниже; подробнее ознакомиться с конкретными примерами реализации этих концепций можно на сайте проекта Aspectrace [8].

Для конструирования генератора отчёта необходимо:

1. Выбрать нужный вид отчёта (например, отчёт в виде набора HTML-документов) и поддерживающее его ядро генератора отчётов (например, входящее в поставку системы).
2. Подобрать из библиотеки готовые или разработать нужные компоненты генератора отчётов.
3. Собрать из выбранных компонентов конфигурацию генератора:
    a. Настроить получение компонентами генератора информации из нужных аспектов от поставщиков трассы.
    b. Настроить эти компоненты.
    c. Настроить нужные связи компонентов генератора отчётов друг с другом и с общими реестрами.
    d. Настроить в ядре генератора его собственные (не зависящие от подключённых компонентов) способы автоматического связывания элементов отчёта.

Когда генератор отчётов сконструирован, остаётся только указать ему входные трассы, запустить фазу анализа, а после её завершения – пост-обработку, если эта фаза предусмотрена типом генератора. В результате на выходе мы получаем экземпляр отчёта, описывающий конкретные полученные данные. Конструирование генератора отчёта можно выполнить один раз, после чего подавать на вход собранной конфигурации генератора различные входные трассы и получать на выходе соответствующие им отчёты.

4.3. Связывание элементов отчёта

В этом разделе рассматриваются вопросы, относящиеся к связыванию данных, уже извлечённых разными компонентами генератора и преобразованных в модель отчёта соответствующего вида. Вопросы связывания информации

внутри одного компонента рассмотрены выше, в разделе 3; в случае различных компонентов все эти принципы связывания также применимы, однако возникают дополнительные технические вопросы интероперабельности между компонентами.

Для связывания элементов отчёта между собой необходимо определить:

1. Как одни компоненты будут находить нужные им элементы отчёта, созданные другими компонентами. К каким реестрам они будут для этого обращаться и каким образом будут извлекать из входного потока информации нужные для поиска ключи.

2. Каким образом будут реализовываться связи между соответствующими элементами отчёта.

Существует большое разнообразие решений этих двух вопросов. В частности, элементы отчёта могут искаться в разных реестрах по ключам, относящимся к разным пространствам имён и получаемым разными способами из трассы или от других компонентов; эти реестры могут статически содержать искомые элементы отчёта или переадресовывать поисковые запросы соответствующим компонентам; элементы могут существовать в реестре независимо от запросов, создаваться в ответ на запрос или генерировать при запросе элемент-посредник. Способы связывания элементов тоже могут быть разнообразны; например, установить связь между двумя HTML-страницами можно с помощью проставления гиперссылок, включения одной страницей внутрь себя другой (так что та становится её частью, и все гиперссылки на вторую страницу перенаправляются на соответствующую метку внутри первой), её копии или некого производного текста, а также включения одной страницей другой в качестве фрейма. Другие виды отчётов могут поддерживать другие представления данных и связей между ними; при этом если соответствующие элементы отчёта поддерживают общие интерфейсы для установления связей между собой, то управляющие ими компоненты генератора отчётов могут устанавливать связи между своими элементами отчёта и элементами, созданными другими компонентами, не имея никакой дополнительной информации друг о друге.

Выбор конкретных решений не регламентируется технологией и зависит только от соглашений, принятых при разработке компонентов генератора, а также при сборке из них конкретного генератора отчётов; при этом в рамках одного генератора возможно одновременное использование разнообразных соглашений.

Наиболее простым и естественным способом связывания данных между компонентами является разметка участков трассы в сочетании с прослеживанием элементов отчёта до сообщений анализируемой трассы, на основе которых они созданы. Непрерывный участок трассы, помеченный некоторой меткой, определяет продолжительное событие (подробнее о продолжительных событиях см. в разделах 3.2–3.3), которому может

соответствовать некоторый элемент отчёта; с помощью таких меток можно естественным и технически несложным способом устанавливать связи между этим элементом отчёта и другими, которым соответствуют другие события в трассе, как вложенные в это событие, так и содержащие его.

Разметка трассы может быть явной и неявной. При явной разметке для любого сообщения трассы можно узнать набор меток, которые действуют в данной точке трассы (поскольку хранение системой анализа трасс всех сообщений и всех меток для них было бы слишком неэффективно, обработка сообщений идёт в потоковом режиме; поэтому запрос действующих на сообщение трассы пометок можно делать только в момент обработки этого сообщения), и запросить (сразу и/или после сбора всей информации из трассы, в зависимости от конкретного используемого механизма разметки, особенностей запрашиваемой информации и соглашений между компонентами генератора) в соответствующих реестрах элементы отчёта, соответствующие этим меткам.

В ядро подсистем трассировки и анализа трасс входит один механизм такого типа, называемый механизмом тэгов. Тэг представляет собой строку, квалифицированную именем аспекта. Для тэга определены операции установки и снятия в заданном канале; всё сообщения в соответствующем канале от точки установки тэга до точки его снятия считаются помеченными им. Компоненты целевого приложения могут устанавливать и снимать тэги, сбрасывая соответствующие сообщения в трассу; другие тэги устанавливаются и снимаются компонентами, анализирующими трассу, в ходе анализа. При этом первые тэги присутствуют в трассе, независимо от её формы внешнего представления, а вторые существуют только в процессе работы анализирующего трассу приложения, содержащего ставящий этот тэг компонент, но всем компонентам анализатора видны тэги обоих видов. Компонент генератора отчётов, поставивший на участке трассы собственный тэг, регистрирует в соответствующем реестре элемент отчёта, соответствующий ему. Также компоненты генератора отчёта могут на основании известной им информации о том, каким образом расставлялись тэги, пришедшие из входной трассы, отображать их с помощью соответствующего реестра в свои элементы отчёта.

Например, в трассе, приведённой в таблице 1, будут присутствовать следующие тэги:

- «тест:t1», действующий на сообщения 1–7
- «метод:testf», действующий на сообщения 2–6
- «метод:f», действующий на сообщения 4–5
- «тест:t2», действующий на сообщения 8–15
- «метод:testf», действующий на сообщения 9–14
- «метод:f», действующий на сообщения 11–12

Тэги от аспекта «тест» уникально идентифицируют тест, и компонент генератора, обрабатывающий сообщения от этого аспекта, может отображать их в элементы отчёта, описывающие соответствующие тесты, позволяя таким образом связывать с ними, например, элементы отчёта, соответствующие сообщениям №№3 и 10 (достижение элементов покрытия), а также элемент, соответствующий сообщению №13 (обнаружение ошибки), с соответствующими тестами.

Однако, как мы видим, тэги вида «метод:testf» и «метод:f» встречаются более одного раза, а значит, их однозначное отображение в элементы отчёта, соответствующие событиям вызовов методов, невозможно. Поэтому библиотека аспекта «метод» устанавливает также дополнительные тэги:

- «метод:testf:1», действующий на сообщения 2–6
- «метод:f:2», действующий на сообщения 4–5
- «метод:testf:3», действующий на сообщения 9–14
- «метод:f:4», действующий на сообщения 11–12

Уникальные тэги аналогичного вида могут устанавливать анализирующие трассу компоненты, обеспечивая таким образом уникальность ключей для поиска элементов отчёта, соответствующих вызовам методов, в рамках не только одной трассы, но и всего набора трасс, поступивших на вход генератору отчёта.

Искать в реестре с помощью тэгов элементы отчёта и связывать их между собой могут не только компоненты генератора отчётов, но и само ядро генератора по указанным ему правилам. Правила эти могут иметь, например, такой вид: «для всех элементов отчёта, помеченных как относящиеся к аспекту А и помеченных тэгами из аспекта Б, для каждого такого тэга искать в реестре элемент отчёта, соответствующий ему, и если таковой найден – устанавливать указанным образом связь между первым и вторым элементом». Связывание такого рода выполняется в фазе пост-обработки; для использования этой возможности, элементы отчёта, относящиеся к аспекту А, должны хранить тэговые метки, которыми были помечены сообщения трассы, на основе которых они созданы.

При неявной разметке трассы явных меток нет, а связывание проводится по динамическому контексту анализа входной трассы. Компоненты, анализирующие трассу, могут при получении очередного сообщения трассы запросить у заданных конфигурацией генератора реестров элементы отчёта, ассоциированные с текущей точкой трассы (такие запросы, в отличие от запросов явной разметки, не могут быть отложены до завершения обработки трассы). Поскольку все компоненты генератора отчётов подсоединены к общему источнику трассы, а сообщения от него обрабатываются строго последовательно, все компоненты имеют общий контекст обработки трассы; при этом, однако, разные компоненты могут получать сообщения от разных

аспектов и изменять набор элементов отчёта, которые они ассоциируют с текущей точкой трассы, в ответ на разные сообщения.

Например, при обработке трассы из таблицы 1 компонент, обрабатывающий сообщения от аспекта «тест» может хранить ссылку на элемент отчёта, соответствующий текущему (во времени обрабатываемой трассы) выполняемому тесту: при обработке сообщения №1 (начало теста t1) эта ссылка будет установлена на элемент отчёта, соответствующий тесту t1, при обработке сообщения №7 (конец теста t1) – сброшена, при обработке сообщения №8 установлена на элемент отчёта, соответствующий тесту t2, а при обработке сообщения №15 – опять сброшена. Соответствующий реестр, поддерживаемый этим компонентом, принимает запросы (отметим, что ключ для запросов может быть пустым), в ответ на которые возвращает ссылку на текущий тест. Компонент, обрабатывающий сообщения от аспекта «ошибка», получив сообщение №13 (ошибка), обращается к этому реестру и получает ссылку на элемент отчёта, соответствующий тесту t2, после чего устанавливает связь между ним и своим элементом отчёта, соответствующим этой ошибке.

Второй после разметки трассы основной способ связывания элементов отчёта – прямой их поиск через реестры по ключам, извлекаемым из сообщений трассы. Способы извлечения ключей зависят от известной информации о зависимостях между сообщениями, сбрасываемыми в трассу различными компонентами целевой системы; пример такого связывания приведён в разделе 3.1. Сам по себе поиск по реестрам и общие принципы работы с реестрами также уже описаны: выше в данном разделе, а также в разделе 4.2, поэтому мы не будем расписывать здесь этот механизм заново.

5. Заключение

При анализе работы сложных систем, особенно состоящих из множества взаимодействующих компонентов, в том числе при их тестировании, возникает необходимость собирать в виде трасс большое количество весьма разнообразной (гетерогенной) информации, а потом анализировать её и представлять в виде понятных разработчикам или заказчикам отчётов. В ходе разработки целевой системы набор видов трассируемой информации может часто меняться, обычно в сторону расширения. Разнообразие и частое изменение видов информации в трассе создаёт проблемы при любом фиксированном формате трассы, так как при расширении набора собираемой информации теряется совместимость по данным между целевыми и анализирующими приложениями.

Поскольку сбор одинаковых видов информации и схожие виды её анализа могут использоваться при разработке разнообразных систем, возникает потребность иметь готовые анализирующие модули, которые можно было бы переиспользовать в других проектах и в других отчётах.

В статье рассмотрены задачи, возникающие при сохранении гетерогенной информации и её анализе, и описана архитектура системы, поддерживающей создание, хранение и анализ гетерогенных трасс. Рассмотрены разнообразные методы анализа и связывания гетерогенных данных, как с использованием темпоральной информации, так и без него. Описана архитектура генератора отчётов, позволяющего собирать генераторы отчётов из готовых независимых аналитических модулей, автоматически связывать между собой результаты проведённого ими анализа и представлять извлечённые данные в виде единого отчёта, понятного человеку.

Литература

[1] Постоянная ссылка на версию статьи:
http://ru.wikipedia.org/wiki/Трассировка_(программирование)?oldid=20486892
[2] Постоянная ссылка на версию статьи:
http://en.wikipedia.org/wiki/Tracing_(software)?oldid=305164770
[3] Ф. Крачтен. Введение в Rational Unified Process // Вильямс, 2002.
[4] К. Бек. Экстремальное программирование // Питер, 2002
[5] A. Cockburn. Agile Software Development: The Cooperative Game (2nd Edition) // Addison-Wesley Professional, 2006
[6] SystemTap homepage
http://sourceware.org/systemtap/
[7] eXtensible Logfile Format (XLF) 1.9.2 Specification
http://www.graphicaldynamics.com/xlf/xlf-spec-1.9.2.html
[8] Сайт проекта Aspectrace
http://forge.ispras.ru/projects/show/aspectrace
[9] Matroska media container homepage
http://www.matroska.org/index.html
[10] С.Д. Кузнецов. Основы баз данных // Бином, 2007
http://www.citforum.ru/database/osbd/contents.shtml
[11] The AspectJ Project homepage
http://www.eclipse.org/aspectj/
[12] Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие // Издательство Московского Университета, 2004
http://parallel.ru/tech/tech_dev/MPI/

Метод зеркальной генерации ограничений для построения тестовых программ по тестовым шаблонам

Корныхин Е.В.
kornevgen@ispras.ru

Аннотация. Статья относится к области системного функционального (core-level) тестирования микропроцессоров, более точно модулей управления памяти. В статье описывается метод построения тестов (тестовой программы) для нацеленной генерации. Такая генерация предполагает систематичное построение тестов специального вида. В конце приводятся результаты апробации реализации метода для тестирования модулей управления памяти микропроцессоров архитектуры MIPS64.

1. Введение

В статье речь пойдет о функциональном тестировании микропроцессоров. Среди всевозможных способов функционального тестирования своей наглядностью и относительно невысокой трудоемкостью выделяется тестирование программами на машинном языке. Эти программы помещаются в память машины, возможно туда же помещается и некоторая дополнительная информация, затем машина запускается, микропроцессор считывает инструкции программы одну за другой и исполняет их. Тем самым микропроцессор работает в различных состояниях. После того, как машина закончила свою работу, на основе анализа ее состояния и трассы исполнения принимается решение о корректности работы микропроцессора.

Одной из подзадач тестирования является построение тестов (в данном случае, программ на машинном языке, или, по-другому, *тестовых программ*). Ручное написание таких программ (на языке ассемблера) применяется для тестирования «крайних» случаев исполнения инструкций или для тестирования «узких» мест микропроцессора, о которых сообщают разработчики. Однако разработчик теста может упустить некоторые важные случаи или посчитать их несущественными. Кроме того, объем ручной работы ограничен человеческими возможностями. Для решения этих проблем построение тестов автоматизируется. Правда, зачастую эта автоматизация не уходит дальше случайно сгенерированных программ [6] или использования программ, которые были подготовлены для совместимых микропроцессоров.

Тем не менее, даже после такого тестирования в микропроцессорах всё ещё остаются ошибки и довольно критичные [17].

В ответ на это исследователями был предложен ряд методов автоматического построения машинных программ [4,5,7,10,12,13,21]. Среди них выделяются методы *нацеленной генерации* программ. В этих методах построение теста выполняется в два этапа: на первом — формулируются цели теста, на втором — автоматически генерируются тесты, удовлетворяющие выбранным целям [21]. Цели теста фиксируют особенности теста и того, как он должен быть исполнен микропроцессором (в какие состояния он должен попасть во время теста). Цели могут задаваться тестировщиком или автоматически выделяться из модели микропроцессора. Например, в качестве цели может быть выбрана особая *ситуация* в работе микропроцессора (скажем, возникновение переполнения), соответственно эта ситуация должна произойти при исполнении построенного для этой цели теста. В том случае, когда основой для задания целей является последовательность инструкций, цель будем называть *тестовым шаблоном (test template)*.

Методы нацеленной генерации программ обладают рядом преимуществ перед остальными методами построения программ [21] (по систематичности, масштабируемости и относительно невысокой трудоемкости при применении их в условиях часто меняющейся тестируемой модели микропроцессора). Исходя из этих преимуществ, в данной статье было решено уделить большее внимание исследованию построения тестов именно для нацеленной генерации.

Надо отметить, что эта задача является сложной с алгоритмической точки зрения. Для произвольных тестовых шаблонов она становится эквивалентной известной задаче Выполнимость (SAT), поскольку задача ставится как поиск объекта, удовлетворяющего набору требований, т.е. по сути некой булевской формуле. Для решения задачи Выполнимость разрабатывается соответствующий класс инструментов [3,18], однако их применимость ограничена. Простыми, очевидными, методами не удается построить для тестового шаблона формулы, допустимые для применения таких инструментов. Тем самым остается актуальным дополнительное исследование методов построения тестов для тестовых шаблонов, возможно, с использованием сходных идей и инструментов [3,18].

Тем не менее, не все тестовые шаблоны для построения теста требуют применения действительно сложных механизмов (типа SAT) [10]. В качестве примера можно привести цели, которые состоят из заданной последовательности названий инструкций с требованиями на их аргументы. Такими требованиями могут быть *зависимости* — отношения на аргументах разных инструкций. Самый простой пример зависимостей — отношения равенства имен (тем самым одна инструкция может записывать значение в регистр, а другая этот регистр читать). Другой класс требований формулируются на значения аргументов инструкции и состояние

микропроцессора перед исполнением инструкции (такие требования суть так называемые *ситуации*). Например, известно, что микропроцессор должен генерировать исключительную ситуацию при исполнении инструкции загрузки из памяти, если в таблице страниц отсутствует нужная виртуальная страница (так называемый PageFault). Или при исполнении инструкции сохранения данных в памяти, если в кэш-памяти нет соответствующих данных по заданному инструкцией адресу (так называемая ситуация кэш-промаха).

Метод, который предлагается в данной статье, применяется для тестирования с нацеливанием на специальные подсистемы микропроцессора, а именно, модули управления памяти (MMU, memory management unit). Такие модули отвечают за выполнение операций трансляции адресов, кэширования данных оперативной памяти, защиты памяти и т.п. Микропроцессор задействует модули управления памяти для выполнения инструкций загрузки и сохранения данных памяти. Тем самым в целях тестов будут отражены особенности исполнения этих инструкций.

Для дальнейшего важными являются следующие моменты. Во-первых, в модулях управления памяти есть подсистемы, которые ведут себя как кэш-память (собственно, кэш-память в узком понимании этого термина, различные буферы, хранящие временные данные и др.). Далее такие подсистемы будут называться *кэширующими буферами*. Их можно представить как множество пар «(адрес, данные)», в которых все адреса разные (термин «данные» понимается в узком смысле, а именно как информация, которая хранится в оперативной памяти и используется в вычислениях). *Обращением* к кэширующему буферу будем называть процедуру получения данных, хранящихся по заданному адресу. Обращение будет считаться *успешным* (эта ситуация называется *кэш-попаданием*), если в кэширующем буфере есть пара с заданным адресом. В противном случае обращение будем считать *неуспешным* (и называть эту ситуацию *кэш-промахом*).

Во-вторых, во время исполнения инструкций обращения к памяти могут происходить обращения к кэширующим буферам. Например, такие обращения возможны при трансляции адреса, если в микропроцессоре хранятся последние осуществленные трансляции адресов. Или такие обращения возможны при работе с данными оперативной памяти, поскольку микропроцессор в кэш-памяти хранит те данные, к которым недавно были обращения.

В качестве целей тестов будем рассматривать заданные последовательности кэш-попаданий/кэш-промахов (в тестовом шаблоне к каждой инструкции привязана своя ситуация кэш-попадания или кэш-промаха). Кэш-попадания/кэш-промахи происходят (или не происходят) в зависимости от адресов, с которыми оперируют инструкции, и состояний кэширующих буферов. Адрес вычисляется на основе аргументов инструкции.

Сложность генерации теста для тестового шаблона определяется следующими факторами. При наличии зависимых аргументов получаются и зависимые

адреса. Каждое обращение к кэширующему буферу может изменить его состояние. Вид этого изменения зависит от успешности предыдущего обращения. А именно, при кэш-промахе некоторые данные из кэширующего буфера вытесняются и на их место помещаются данные по заданному в обращении адресу. Получается, что состояния кэширующего буфера для разных инструкций также являются зависимыми. Тем самым рассматриваемая цель теста порождает множество зависимостей между значениями аргументов инструкций и состояния микропроцессора. Если учесть, что размер состояния кэширующего буфера измеряется тысячами разных пар, а последовательность изменений состояний дает на несколько порядков большее количество пар, то применение привычных инструментов SAT становится практически бесперспективным при большом числе зависимостей. Тем не менее, отказываться от таких тестовых шаблонов тоже плохо, ибо они дают систематичный подход к построению тестов с особым исполнением (причем иными способами такие тесты построить крайне сложно или такой способ не будет систематичным).

Немаловажен вопрос представления решения. Состояние кэширующего буфера представляется набором пар чисел (адрес и данные). Однако не любой микропроцессор позволяет задать состояние своего кэширующего буфера напрямую (заполнением парами чисел). Даже если это и возможно, то изменение состояний всех нужных кэширующих буферов перед каждым тестом увеличит время проведения тестирования. Например, увеличение времени на пару секунд для 100000 тестов увеличит время тестирования более чем на 50 часов. Решением этой проблемы была бы генерация *инициализирующих* инструкций. Такие инструкции помещаются перед тестом и подготавливают микропроцессор к исполнению этого теста. Другим возможным решением могло бы быть более эффективное использование заданного начального состояния кэширующих буферов, если о нем имеется информация. В данной статье рассматривается генерация инициализирующих инструкций.

Далее будет проведен обзор существующих инструментов построения тестов для рассматриваемых целей. Затем будет рассмотрен предлагаемый метод, после чего будут даны результаты апробации реализации метода для тестирования, нацеленного на модули управления памяти микропроцессоров архитектуры MIPS64 [16].

2. Обзор

Видимо, ввиду особой сложности задачи для рассматриваемых тестовых шаблонов существует не так много инструментов ее решения.

Коммерческая американская разработка — инструмент RAVEN [13] — позволяет автоматически строить тесты нацеленным образом. В качестве целей тестов могут встречаться последовательности инструкций и типов инструкций (например, одна арифметическая инструкция, за ней две

инструкции обращения к памяти). Возможно задание вероятностей возникновения некоторых ситуаций при исполнении инструкций. В том числе в такие ситуации входят и кэш-попадания/кэш-промахи. Однако в открытой печати отсутствует изложение идей, на основе которых работает этот инструмент. Можно лишь сделать вывод, что для построения тестов используются продвинутые алгоритмы рандомизированного поиска. Кроме того, инструмент не позволяет пользователю задать архитектуру, семантику инструкций. Вместо этого используются готовые наборы инструкций (архитектуры MIPS [16], ARM [11] и x86 [1]).

Другая разработка — инструмент Genesys-Pro [12] — используется в IBM для тестирования микропроцессоров. Поддерживаются те же цели тестов, что и в RAVEN. Инструмент также является проприетарным. Для заданного тестового шаблона составляется *система ограничений* (constraints) [2]. На основе результатов ее разрешения формируется тест. Достаточно выразительный язык тестовых шаблонов (в том числе с использованием массивов — Memory, PageTable) требует нетривиальной трансляции в ограничения для возможности разрешения в адекватные временные рамки. Однако среди открытых публикаций отсутствуют подробности этой трансляции. По сравнению с RAVEN в этом инструменте есть возможность задать семантику инструкций в виде ограничений (constraints) на аргументы и результаты инструкции.

Аналогичный подход построения тестов был использован в инструменте MAATG [15]. Его разработчики преследовали цель максимально использовать существующий инструментарий и технологии. Поэтому для описания архитектуры используется один из хорошо известных инженерам язык EXPRESSION [9], а для разрешения ограничений предполагается использовать существующую библиотеку [8]. В тестовых шаблонах можно задавать последовательности инструкций, зависимости на аргументы инструкции и ситуации при исполнении инструкций. Однако в единственной статье не раскрываются подробности того, как происходит построение ограничений для заданных ситуаций.

В ИСП РАН разрабатывается инструмент MicroTESK [21] построения тестов для микропроцессоров. В нем реализован метод автоматической генерации тестовых шаблонов на основе модели микропроцессора. В тестовых шаблонах возможны заданные последовательности инструкций, зависимости аргументов инструкций, ситуации, которые происходят при исполнении инструкций (например, кэш-попадания/кэш-промахи). Особенностью инструмента является возможность автоматического построения тестовых шаблонов для заданного тестового покрытия на основе модели микропроцессора (инструкции, тестовые ситуации для инструкций). Построение же теста для тестового шаблона в этом инструменте автоматизировано лишь частично. Авторы предлагают следующую общую схему построения теста. Для каждой инструкции выбираются все зависимости с аргументами предыдущих

инструкций и для этих зависимостей на основе известных значений аргументов предыдущих инструкций и текущего состояния микропроцессора выбираются значения аргументов, для которых эти зависимости выполнены. Если же зависимости таким образом разрешить не удается (они сложные или несовместные), то всё равно выбираются некоторые допустимые значения для аргументов. Сложность выбора значений аргументов для рассматриваемых в статье тестовых шаблонов оказалась такой, что построение тестов удалось произвести только для шаблонов из 2-3 инструкций (для архитектуры MIPS [20]). Эта особенность не дает возможности проверить некоторые существенные механизмы модулей управления памяти (например, достижение некоторых «крайних» ситуаций в работе кэш-памяти [22]).

Таким образом, все инструменты либо являются закрытыми и про них неизвестны особенности методов построения тестов, либо их возможности ограничены тестами небольших размеров. В данной статье предлагается метод построения тестов для тестовых шаблонов на основе разрешения ограничений (constraints), который применим для тестовых шаблонов б*о*льших размеров.

3. Зеркальная генерация

Напомню, что рассматривается задача построения теста для тестовых шаблонов специального вида. Цель теста задается в виде последовательности обращений к кэширующему буферу с указанием успешности обращения (т.е. с указанием кэш-попадания или кэш-промаха; далее будем их называть *тестовыми ситуациями*) и некоторого дополнительного ограничения на адреса обращения: $\langle T(x_1, x_2, ..., x_n), P(x_1, x_2, ..., x_n) \rangle$, где $x_1, x_2, ..., x_n$ – адреса, с которыми происходят обращения. Последовательность тестовых ситуаций задается последовательностью $T_i = \langle S_i, x_i \rangle$, $i = 1, 2, ..., n$. Дополнительное ограничение на адреса задается с помощью предиката $P(x_1, x_2, ..., x_n)$.

Идея зеркального метода состоит в том, чтобы для каждой тестовой ситуации (для каждого обращения) обеспечить наличие предыдущей инструкции, обращающейся к тому же адресу. При этом успешность обращения в исходной инструкции (т.е. как раз тестовая ситуация) будет в этом случае определяться только той частью теста, которая расположена между предыдущим обращением и данной инструкцией. Это позволяет существенно сократить количество переменных в ограничениях и тем самым ускорить получение теста. В частности для описания тестовой ситуации не надо будет использовать начальное состояние кэширующего буфера (которое может содержать несколько тысяч констант и тем самым сильно усложнить ограничения). Конечно, какое-то начальное состояние существует всегда, и оно определяет успешность обращений в кэширующий буфер, но зеркальный метод ограничений позволит обойти явное построение начального состояния и вместо этого предлагает построение инструкций, при исполнении которых и будет получено это начальное состояние.

Возможен случай, когда «предыдущую» инструкцию невозможно выбрать среди инструкций тестового шаблона. Поэтому в число генерируемых данных зеркальным методом (кроме $x_1, x_2, ..., x_n$) добавляются *инициализирующие* инструкции. Обозначим адреса, с которыми работают инициализирующие инструкции, как $t_1, t_2, ..., t_m$. На их основе инициализирующие инструкции создаются автоматически (обращение по заданному адресу в память).

Более точно, если обращение к кэширующему буферу по некоторому адресу должно быть успешным, то перед этим обращением должно быть другое обращение по этому же адресу, причем между этими двумя обращениями данные по этому адресу не вытесняются из буфера. Если обращение к кэширующему буферу по некоторому адресу должно быть неуспешным, то перед этим обращением всё равно должно быть другое обращение по этому же адресу, причем между этими обращениями данные по этому адресу должны быть вытеснены и не помещены в буфер вновь. Получается, что у каждого адреса x_i есть свой «зеркальный» адрес среди $x_1, x_2, ..., x_{i-1}$ или $t_1, t_2, ..., t_m$. На Рис. 1 приведен псевдокод алгоритма построения ограничений (constraints) согласно зеркальному методу.

В псевдокоде использованы предикаты *NE* и *E*. Они служат для записи ограничений о вытеснении (*E*) и невытеснении (*NE*) данных из кэширующего буфера. А именно, предикат $NE(x_i; t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1})$ выполнен тогда и только тогда, когда данные по адресу x_i не вытеснены в результате обращений в кэширующий буфер по адресам $t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1}$. Предикат $E(x_i; t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1})$ выполнен тогда и только тогда, когда данные по адресу x_i вытеснены в результате обращений в кэширующий буфер по адресам $t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1}$.

```
function generate( m, ⟨S_i, x_i⟩_{i=1,2,...,n} ) : constraints
add constraint «all different t_1, t_2, ..., t_m»
foreach( S_i, x_i )
      case S_i = hit:
            add constraint « x_i ∈ {t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1}} »
            add constraint NE( x_i ; t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1} )
      case S_i = miss:
            add constraint « x_i ∈ {t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1}} »
            add constraint E( x_i ; t_1, t_2, ..., t_m, x_1, x_2, ..., x_{i-1} )
```

Рис. 1. Псевдокод алгоритма зеркальной генерации ограничений.

Для выражения предикатов *E* и *NE* в виде ограничений применяются свои методы. Эти методы не зависят от зеркального метода и поэтому в данной статье не рассматриваются. Тем не менее, в качестве иллюстрации заметим, что для их записи в виде ограничений может применяться *метод функций полезности*. Для стратегии вытеснения LRU [14], одной из наиболее часто встречающихся, выражение для *NE* будет иметь вид $\sum_{i=1}^{n+m} u_x(p_i) < w$, а для *E* вид $\sum_{i=1}^{n+m} u_x(p_i) \geq w$. Напомним еще раз смысл символов: w – ассоциативность кэширующего буфера (количество его секций), последовательность p – конкатенация последовательностей t и x: $p = \langle t_1, t_2, ..., t_m, x_1, x_2, ..., x_n \rangle$. В ограничениях используется функция полезности $u_x(p_i)$ для соответствующей инструкции (если $p_i \equiv t_j$, то этой инструкцией является *j*-я инициализирующая инструкция; если $p_i \equiv x_k$, то этой инструкцией является *k*-я инструкция тестового шаблона). Функция полезности для адреса x (он обозначен нижним индексом у символа *u*) определяет, является ли соответствующая ей инструкция *полезной* для вытеснения данных с адресом *x*. Она равна 1 для полезной инструкции и 0 для инструкции, не являющейся полезной. Вытеснение наступает в тот момент, когда количество полезных инструкций превышает некоторый константный порог. В данном случае этот порог равен *w*. Для стратегии вытеснения LRU функция полезности может иметь следующий вид (существуют и другие функции полезности для LRU) :

$$u_x(p_i) \equiv x \notin \{p_i, p_{i+1}, ..., p_{n+m}\} \wedge R(x) = R(p_i) \wedge p_i \notin \{p_{i+1}, p_{i+2}, ..., p_{n+m}\}$$

(инструкция считается полезной, если она встречается после последнего обращения в буфер по адресу *x,* в том же «сете» и среди таких же инструкций обращение по адресу p_i является последним). Выражение $R(x) = R(p_i)$ истинно в том случае, когда при обращении в кэширующий буфер по адресам x и p_i будет использован один и тот же «сет» буфера, т.е. номер строки кэширующего буфера (возможно, в разных секциях). Номер строки для адреса вычисляется с помощью выражения из операций над адресом как битовой строкой (обычно это подстрока адреса).

В [23] доказана корректность зеркального метода (т.е. тесты, построенные по зеркальному методу, действительно удовлетворяют тестовому шаблону) и его полнота для существенно вытесняющих стратегий вытеснения (т.е., если ограничения, построенные по зеркальному методу, несовместны, значит для данного тестового шаблона действительно не существует теста, и если для тестового шаблона существует тест, то ограничения, построенные по

зеркальному методу, будут совместными). Под существенно вытесняющей стратегией вытеснения понимается такая стратегия вытеснения, которая рано или поздно позволяет вытеснить любые данные. К таковым стратегиям вытеснения относятся все используемые на практике стратегии вытеснения.

Рассмотрим вопрос выбора m – длины инициализирующей программы. В [23] доказано, что она ограничена величиной, зависящей только от структуры тестового шаблона и ассоциативности кэширующего буфера. Для стратегии вытеснения LRU справедливы следующие оценки (напомним, что n – количество инструкций тестового шаблона, w – ассоциативность кэширующего буфера):

1. если кэширующий буфер является полностью ассоциативным, и в шаблоне нет кэш-промахов, то $m \le n$;
2. если кэширующий буфер является полностью ассоциативным, но в тестовом шаблоне есть кэш-промах, то $m \le w+1$;
3. для произвольного кэширующего буфера $m \le n \cdot w + M$, где M – количество кэш-промахов в тестовом шаблоне.

Зеркальный метод применим и в случае многоуровневой иерархической кэш-памяти. В этом случае составляются отдельно инициализирующие последовательности для каждого уровня кэш-памяти, а затем на основе них строится общая инициализирующая последовательность. Например, если микропроцессор допускает обращение в обход кэш-памяти второго уровня, то сначала надо инициализировать кэш-память второго уровня, а затем инициализировать кэш-память первого уровня в обход кэш-памяти второго уровня. При работе с многоуровневой кэш-памятью возникает следующая особенность: обращение к k-му уровню производится только в случае кэш-промахов при обращении к уровням 1, 2, ..., k-1. Поэтому чтобы инициализирующая последовательность для кэш-памяти второго уровня действительно достигала своего уровня, перед этой инициализирующей последовательностью надо вставить обращения в кэш-память первого уровня, вытесняющие данные с адресами, которыми надо инициализировать кэш-память второго уровня. Тем самым общая инициализирующая последовательность будет состоять из трех частей. В первой части из кэш-памяти первого уровня вытесняются данные для второй части, во второй части происходит инициализация кэш-памяти второго уровня (при этом состояние кэш-памяти первого уровня может меняться) и в третьей части происходит инициализация кэш-памяти первого уровня в обход кэш-памяти второго уровня.

В статье [24] также рассматривается построение тестов по тестовым шаблонам, нацеленным на модули управления памяти. Однако в отличие от нее в данной статье сделан ряд существенных усовершенствований. Во-первых, в [24] генерируемые ограничения содержат начальное состояние целиком, что в случае реальных микропроцессоров может серьезно затруднить разрешение таких ограничений (ввиду их огромного размера и,

поэтому, сложности). Во-вторых, в [24] предполагается, что проводится «распределение по сетам» инструкций тестового шаблона. Иными словами, с помощью некой процедуры должен быть выбран «сет» адреса, с которым будет работать инструкция. Сложность распределения заключается в цене неверного результата распределения. А именно, при неправильном распределении (обнаружится это в самом конце, когда будет получена несовместная система ограничений) придется делать распределение заново (если ещё не все распределения были задействованы). Такой «перебор с возвратом» может ухудшить скорость построения теста (тем более что в [24] не дается рецептов распределений, которые с большой вероятностью приведут в результате к совместной системе ограничений). В данной статье «распределение по сетам» не делается – вместо этого используются несколько видоизмененные ограничения (в них добавлено условие $R(x) = R(x_i)$, выражающее совпадение «сетов»). Тем самым, подводя итог, в данной статье сделан ряд улучшений по сравнению с предыдущими разработками по генерации тестов с помощью ограничений. Изменения призваны с одной стороны упростить ограничения и ускорить генерацию тестов, а с другой стороны в [24] описан конкретный случай (MIPS64 и LRU), а в данной статье описывается общий случай (достаточно провести выделение кэширующих буферов в модуле управления памяти, а поскольку все такие буферы во всех микропроцессорах ведут себя одинаково, то и методы построения ограничений будут одинаковыми).

4. Апробация

Рис. 2 и 3 показывают результаты апробации реализации зеркального метода для тестирования модулей управления памяти микропроцессоров архитектуры MIPS64 [16]. Тестовые шаблоны состояли из инструкций загрузки и сохранения данных в памяти. В каждой инструкции должна была выполняться трансляция адреса через TLB (с использованием буфера для хранения последних совершенных трансляций) и затем обращение в основную память через кэш-память первого уровня. Случайным образом выбирались зависимости аргументов инструкций и начальное состояние микропроцессора. Затем для длин тестов от 2 до 16 вычислялось среднее время работы реализации для кэш-памяти с ассоциативностью 2, 4, 8 и 16 (и бралось среднее значение по всем ассоциативностям для фиксированной длины теста). Кроме того вычислялась доля шаблонов, для которых был построен тест за время, меньшее чем 60 секунд (если время превышало 60 секунд, построение теста обрывалось). Эксперименты проходили на компьютере AMD Athlon64 3200+ 2ГГц с 1ГБ оперативной памяти. В качестве решателя ограничений использовался Z3 [19].

Рис. 3 отражает долю тестовых шаблонов, для которых удалось построить тесты. Рис. 2 отражает среднее время *продуктивного принятия решения* о тестовом шаблоне, т.е. для заданного размера теста (по оси абсцисс) среднее

время определения того, что шаблон является несовместным (для него не может быть теста вовсе) или совместным (после чего строился тест).

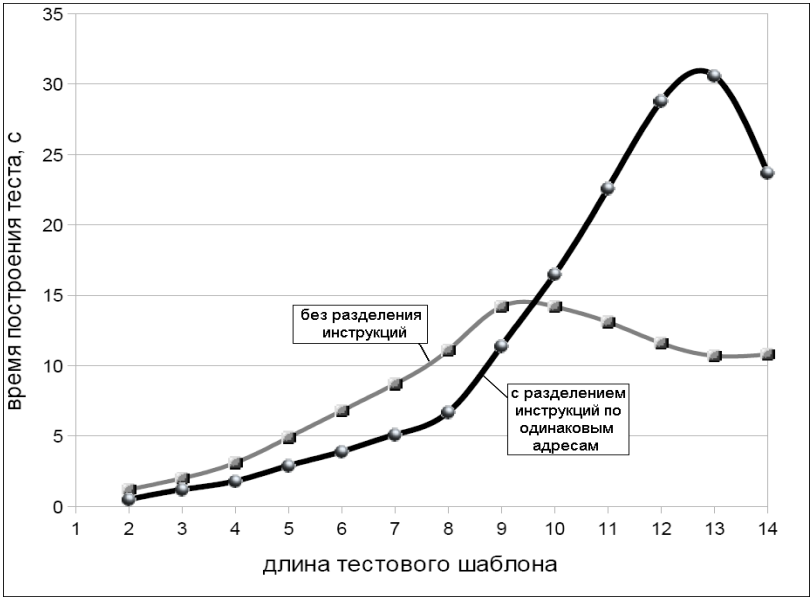


Рис. 2. Среднее время продуктивного принятия решения о тестовом шаблоне

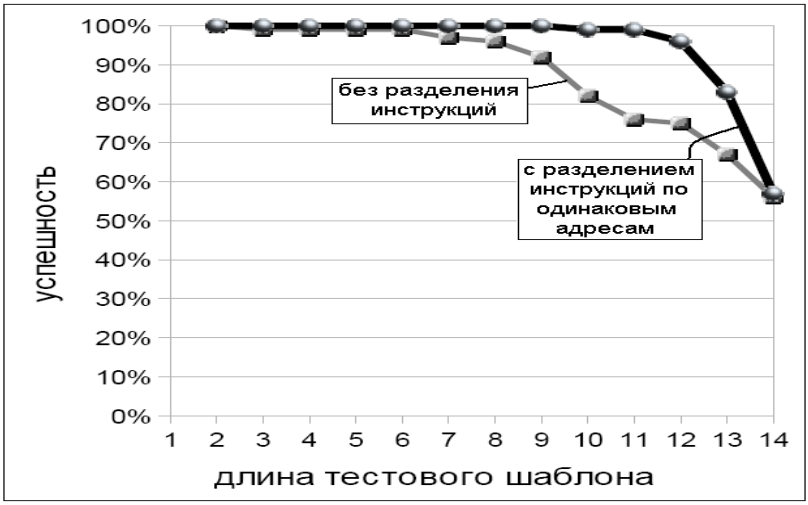


Рис. 3. Доля тестовых шаблонов, для которых удалось построить тест за 60 сек. или определить их несовместность.

В первом эксперименте (его результаты отражены на рисунках линиями с квадратиками) метод зеркальной генерации применялся для тестового шаблона целиком. Оказалось, что до некоторой длины шаблона (значит, и теста) (8-9) метод работает успешно практически для всех тестовых шаблонов (97%-100%). При дальнейшем увеличении длины теста начинает уменьшаться доля шаблонов, для которых удается построить тест (по 5-10% при увеличении длины теста). Тем самым при длине теста порядка 14-15 уже для половины шаблонов решатель не успевает за 60 секунд построить тест. В результате анализа работы реализации метода для этих шаблонов был сделан вывод о том, что для большинства из них (60-70%) тестов не может быть вовсе, так как в этих шаблонах имеются обращения по одинаковым адресам с кэш-промахами после кэш-попаданий и кэш-промахами после кэш-промахов (поскольку в результате кэш-промаха данные помещаются в кэш-память, то при повторном обращении к ним должно происходить кэш-попадание, а не кэш-промах).

Поскольку эта ситуация стала по сути следствием случайного выбора тестовых шаблонов, то в следующем эксперименте перед работой зеркального метода была вставлена проверка, отсеивающая априори несовместные тестовые шаблоны (т.е. такие, для которых не может существовать тестов вовсе). Кроме того обращения в кэш-память были поделены на группы на основе одинаковых имен аргументов и для этих групп применялся метод зеркальной генерации (обращения в разные «сеты»), обращения к буферу TLB не делились на группы, поскольку этот буфер является полностью ассоциативным (а значит в нем всего один «сет»). Результаты второго эксперимента отражены на рисунках 2 и 3 линией с кружками. Можно заметить, что примененные изменения позволили увеличить еще на несколько единиц размер тестового шаблона, для которого построение теста удастся провести достаточно эффективно.

5. Заключение

В статье предложен метод построения тестов для тестовых шаблонов специального вида. Эти шаблоны предназначены для системного функционального тестирования микропроцессоров, нацеленного на модули управления памяти. Метод состоит в особом построении систем ограничений (constraints) и в последующем их разрешении с генерацией теста. Построение ограничений основано на следующих идеях: при обращении по некоторому адресу происходит кэш-попадание, если перед этим обращением было обращение к этому же адресу и между этими обращениями данные по данному адресу не были вытеснены; и при обращении по некоторому адресу происходит кэш-промах, если перед этим обращением было обращение к тому же адресу и между этими обращениями данные по данному адресу были вытеснены и не помещены в буфер вновь.

Метод был реализован для архитектуры MIPS64 [16] и в данный момент внедряется в инструмент MicroTESK [21]. Результаты апробации показали, что с использованием предложенного метода появляется возможность строить более сложные тесты, чем это удавалось ранее. Такие тесты позволяют проверить такие особенности работы модулей управления памяти, которые до этого было сложно проверить систематичным образом (например, достижение некоторых «крайних» ситуаций в работе кэш-памяти [22]).

Литература

[1] Anderson D., Shanley T., MindShare Inc. Pentium processor system architecture / Don Anderson, Tom Shanley, MindShare Inc. Addison-Wesley, 1995.

[2] Apt K. Constraint Logic Programming using Eclipse / Krzysztof Apt, Mark Wallace. New York: Cambridge Univ. Press, 2007.

[3] Cohen B. Local Search Strategies for Satisfiability Testing / Bram Cohen, Henry Kautz, Bart Selman // Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993.

[4] Corno F. Automatic Test Program Generation from RT-Level Microprocessor Descriptions / Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, Giovanni Squillero // Proceedings of the International Symposium on Quality Electronic Design, 2002.

[5] Corno F. Fully Automatic Test Program Generation for Microprocessor Cores / Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, Giovanni Squillero // Proceedings of DATE 2003: Design, Automation and Test in Europe, 2003, pp. 1006-1011.

[6] David R. Random Testing of the Data Processing Section of a Microprocessor / Rene David, Pascale Thevenod-Fosse // Proceedings of 11th IEEE Symposium on Fault-Tolerant Computing, 1981.

[7] Dutt N. Automatic functional test program generation for pipelined processors using model checking / Nikil Dutt, Prabhat Mishra // Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.

[8] EFC Constraints Solving Library [Electronic resource] / Fahiem Bacchus, George Katsirelos. 2004. Mode access: http://www.cs.toronto.edu/~gkatsi/efc/efc.html.

[9] EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. / A. Halambi, P. Grun, V. Ganesh et al. // Proceedings of the European Conference on Design, Automation and Test, 1999, pp. 485-490.

[10] Fallah F. A new functional test program generation methodology / Farzan Fallah, Koichiro Takayama // Proceedings of 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2001, pp. 76-81.

[11] Furber S.B. ARM system-on-chip architecture / Stephen Bo Furber. Pearson Education, 2000.

[12] Genesys-pro: Innovations in test program generation for functional processor verification / A. Adir, E. Almog, L. Fournier et al. // IEEE Design and Test of Computers, 2004, vol. 21, no. 2, pp. 84–93.

[13] Hennenhoefer E. The Evolution of Processor Test Generation Technology [Electronic resource] / Eric Hennenhoefer, Melanie Typaldos. Electronic data. Obsidian Software Inc., 2008. Mode access: http://www.obsidiansoft.com/images/pdf/evolution.pdf

[14] Hennessy J. L. Computer architecture: a quantitative approach / John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau. 4 edition. Morgan Kaufmann, 2007.

[15] MAATG: A functional test program generator for microprocessor verification / Tun Li, Dan Zhu, Yang Guo, GongJie Liu, SiKun Li // Proceedings of the 8th Euromicro conference on Digital System Design (DSD'05), 2005, pp. 176-183.
[16] MIPS64TM Architecture For Programmers Volume II: The MIPS64TM Instruction Set / MIPS Technologies. 2003.
[17] Moler C. A Tale of Two Numbers [Electronic resource] / Cleve Moler. Electronic data. MATLAB News & Notes, 1995. Mode access: http://www.mathworks.com/company/newsletters/news_notes/pdf/win95cleve.pdf
[18] Moskewicz M. Chaff: Engineering an Efficient SAT Solver / Matthew W.Moskewicz, Concor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik // Proceedings of the 39th Design Automation Conference (DAC 2001), Las Vegas, 2001.
[19] de Moura L. Z3: An efficient SMT solver / Leonardo de Moura, Nikolaj Bjorner // Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337-340.
[20] Воробьев Д.Н., Камкин А.С. Генерация тестовых программ для подсистемы управления памятью микропроцессоров / Дмитрий Воробьев, Александр Камкин // Труды Института системного программирования / под ред. В.П.Иванникова. М.: ИСП РАН, 2009, Т. 17, с. 119-132.
[21] Камкин А.С. Генерация тестовых программ для микропроцессоров / Александр Камкин // Труды Института системного программирования / под ред. В.П.Иванникова. М.: ИСПРАН, 2008, Т. 14(2), с. 23-64.
[22] Камкин А.С, Корныхин Е.В. Построение тестовых программ для верификации подсистем управления памятью микропроцессоров // препринт ИСП РАН, 2010.
[23] Корныхин Е.В. Построение тестовых программ для проверки подсистем управления памяти микропроцессоров / Корныхин Евгений Валерьевич ; науч. рук. А.К. Петренко ; Мос. гос. ун-т им. М. В. Ломоносова. М., 2010.
[24] Корныхин Е.В. Генерация тестовых данных для тестирования механизмов кэширования и трансляции адресов микропроцессоров / Евгений Корныхин // Программирование, 2010, Т. 36, № 1, с. 28-35.

Создание модулей поддержки архитектур для среды TrEx с помощью специализированного языка описания процессоров

П.М. Довгалюк, М.А. Климушенкова, А.М. Мухина
{Pavel.Dovgaluk, Maria.Klimushenkova, Anna.Mukhina}@ispras.ru

Аннотация. В данной статье рассматривается предложенный авторами подход к разработке модулей поддержки архитектур для среды TrEx на основе языка описания процессорных архитектур. Проанализированы достоинства и недостатки подхода по сравнению с уже существующими методами разработки.

1. Введение

TrEx – программная среда динамического анализа бинарного кода. Возможности среды позволяют решать задачу восстановления алгоритма, преодолевая при этом комплекс средств защиты от анализа. Программные инструменты среды базируются на анализе потоков данных в трассе выполнения программы и позволяют выполнять быстрое прототипирование специфических для каждого отдельного случая алгоритмов.

Система динамического анализа программного обеспечения TrEx поддерживает возможность разработки алгоритмов анализа, не зависящих от системы команд конкретной архитектуры. Такая независимость достигается за счет трансляции анализируемых инструкций в универсальное внутреннее представление, общее для всех аппаратных платформ. Поэтому для каждой конкретной платформы необходимо реализовать модуль трансляции в универсальное внутреннее представление. Во время такой трансляции требуется описывать связи по данным между входными и выходными операндами инструкций.

Модуль поддержки архитектуры является необходимой прослойкой между трассой в сыром, необработанном виде и универсальным внутренним представлением. Он содержит функции для определения вида инструкции по ее двоичному коду, выделения операндов, генерации мнемоники и декомпозиции зависимостей.

Декомпозиция зависимостей по данным между входными и выходными операндами инструкций необходима для того, чтобы эффективно выполнять так называемый слайсинг трассы. Слайсинг – это способ фильтрации шагов трассы, позволяющий сократить ее объем на несколько порядков. Суть слайсинга состоит в том, что из трассы выделяются только те инструкции, входных операндов которых достигли начальные данные, или работа которых повлияла на результирующие значения в выходе алгоритма. Таким образом, для того, чтобы слайсинг позволял выделять именно те инструкции, которые относятся к анализируемому алгоритму, необходимо точно описывать зависимости по данным между входными и выходными операндами инструкций [[1]].

В настоящее время в TrEx реализована поддержка лишь для двух аппаратных платформ – MIPS64 и Intel x86. Для расширения сферы применения средств динамического анализа кода требуется разработать модули поддержки архитектуры для платформ Power PC и ARM, снизив, насколько это возможно, затраты ресурсов на их разработку. Основная проблема состоит в том, что модуль поддержки архитектуры содержит большое количество классов, весьма схожих по своей структуре, что делает разработку рутинной и однообразной. Необходимо сократить объем кода, написанного вручную, не усложняя общую структуру модуля и процесс поиска и исправления ошибок.

2. Описание существующих методов решения проблемы

В данный момент уже разработаны модули поддержки архитектуры для Intel и MIPS64.

Каждая из реализаций состоит из следующих частей:

- Подсистема, описывающая основные параметры архитектуры
- Модуль распознавания вида инструкции по ее двоичному коду
- Функции выделения операндов инструкций
- Функции декомпозиции зависимостей между данными

Основную часть времени разработки занимает создание функций выделения операндов и декомпозиции зависимостей между данными. Большой объем кода объясняется тем, что отдельные функции создаются для каждой из поддерживаемых инструкций, поэтому при разработке модулей поддержки архитектур MIPS64 и Intel x86 были предприняты меры, упрощающие написание этого кода.

2.1. Реализация с помощью макросов

В реализации модуля поддержки архитектуры MIPS64 проблема большого объема рутинной работы при описании функций декомпозиции была решена с помощью макросов. Макросы избавляют от необходимости писать вручную

большое количество схожих по структуре классов, перекладывая эту работу на препроцессор. Описание обработчиков инструкций, таким образом, становится более лаконичным, и общий объем кода сокращается. Однако в целом полученная структура оказалась весьма сложной: для описания разборщика инструкции используется специальный макрос, параметрами которого также являются макросы, которые, в свою очередь, принимают параметрами другие макросы, и так далее. Отследить, какие параметры на самом деле передаются в этой цепочке, весьма трудно. Кроме того, очень трудно держать в голове код, который получается после таких подстановок. Поэтому в таком коде тяжело обнаруживать ошибки, как на этапе компиляции, так и во время отладки.

2.2. Реализация с помощью шаблонов

В реализации модуля поддержки архитектуры MIPS используются шаблонные функции. Это позволило перенести часть проверок времени выполнения на этап компиляции и, таким образом, ускорить процесс разбора зависимостей данных. Кроме того, за счет того, что MIPS является RISC-процессором, в модуле поддержки описаны всевозможные способы разбора операндов и зависимостей инструкций, что делает описание разборщика для конкретной команды очень лаконичным. Недостатками данного подхода являются сложность метапрограммирования на C++ (большое количество кода, длительная компиляция, сложности с отладкой), описания команды разделено на несколько частей (что усложняет поиск ошибок и сопровождение), а также необходимость вручную заполнять таблицу выбора инструкции по ее двоичному коду.

3. Предлагаемый метод решения

Недостатки существующих методов были учтены при разработке метода автоматической генерации разборщиков инструкций. Предлагаемый метод решает проблемы сложности обнаружения ошибок в реализации функций декомпозиции и выделения операндов, поскольку позволяет многие из них находить уже на этапе компиляции.

Часть функций модуля поддержки архитектуры предлагается генерировать автоматически с помощью генератора разборщиков инструкций. Каждый разборщик инструкции представляет собой класс, выполняющий функции выделения операндов, генерации мнемоники и декомпозиции зависимостей. В зависимости от типа очередной инструкции, который определяется по ее двоичному коду, вызывается определенный разборщик, который и производит ее обработку.

Разборщики имеют во многом схожую структуру, поэтому значительную часть кода можно генерировать на основе краткого описания архитектуры и системы команд. Для описания инструкций был создан специальный язык, который транслируется генератором разборщиков в код на языке C++.

Разработка специального языка оказалась целесообразной по причине одновременной разработки модулей поддержки сразу двух архитектур. Таким образом, разработанный генератор можно эффективно использовать сразу для обоих модулей. Экономия времени будет достигаться за счет однократного тестирования классов, общих для этих архитектур, ускорения разработки таблиц выбора инструкции по их двоичным кодам, уменьшения трудоемкости кодирования функций декомпозиции зависимостей по данным, исправление части дефектов до начала этапа полноценного тестирования (за счет их обнаружения на этапе компиляции).

4. Спецификация языка описания команд и зависимостей

Генератор разборщиков инструкций получает на вход файл специального формата, и генерирует на его основе функции для выбора инструкции по двоичному коду, функции выделения операндов, генерации мнемоники и декомпозиции зависимостей.

Входной файл содержит два основных раздела: описание параметров архитектуры и описание инструкций.

В разделе описания параметров архитектуры описывается имя модели (MODEL_NAME), которое будет подставлено в имена генерируемых классов. Впоследствии этот раздел может быть дополнен и другими параметрами, если это будет необходимо для описания каких либо специфических особенностей целевой архитектуры.

За разделом описания параметров архитектуры следует основной раздел – раздел описания инструкций. Каждая инструкция описывается отдельно. На основе описания каждой инструкции генератором в результирующем файле создается определение класса, включающего в себя функции разбора двоичного кода инструкции. Описание инструкции в исходном файле включает в себя следующие параметры:

1. Имя инструкции
2. Описание маски двоичного кода инструкции
3. Описание операндов инструкции
4. Описание метода формирования мнемоники
5. Описание функции декомпозиции

Описание маски двоичного кода инструкции состоит из набора символов (для ARM и PowerPC этих символов будет по 32 [[2], [3]]), каждый из которых соответствует одному биту (от старших к младшим):

1. 1 или 0 - фиксированный бит, используемый для определения вида команды

2. Символ (латинская буква) - бит является частью поля, которое можно использовать в описании операндов, мнемоники или декомпозиции

Описание операндов представляет собой список пар (символ в описании двоичного кода, название вида операнда). Рассмотрим пример такого описания:

d = GPR

В данном случае будет создан операнд для поля кода инструкции d с помощью вызова функции, разбирающей операнд типа GPR (регистр общего назначения):

parseGPR<12,4>(instruction)

Набор функций, разбирающих операнды различного типа, должен быть создан в базовом для разборщиков инструкций классе. Шаблонные параметры функции – номер младшего бита и количество общее количество битов поля.

Описание мнемоники состоит из перечня выражений, разделенных запятыми. Каждое из выражений должно возвращать указатель на const char для использования в качестве аргумента функции sprintf.

Описание декомпозиции - это код функции декомпозиции на C++, который может содержать некоторые вспомогательные макросы:

1. $UPDATE ($KILL, $GET, $SET, $CHECK) - создать зависимость определенного вида. В скобках содержатся выходные и входные параметры зависимости, разделенные двоеточием. Число - это номер операнда команды (начиная с 0), идентификатор - имя регистра.

2. $FLAG ($ADDRESS, $CF, $STACK) – проверка, включен ли соответствующий режим декомпозиции.

Также в описании мнемоники и декомпозиции могут использоваться:

1. Значения полей, закодированные в двоичном коде инструкции. Синтаксис - $d, где d - имя поля.

2. Указатель на разбираемую инструкцию (переменная instruction).

3. Указатели на объекты-операнды инструкции. Синтаксис - $n, где n - номер операнда.

Пример описания инструкции сложения регистра с произвольным операндом:

```
ADD : cccc00I0100Sssssddddiiiiiiiiiiii
{
  isJump = $d == ARMGPR::PC;
  isConditionalJump = ($d == ARMGPR::PC) && ($c != AL);
}
( d = GPR, s = GPR, i = ShifterImm32 )
```

```
( "ADD", getConditionMnemonic(instruction), ($S ? "S" :
"") )
{
  parseCondition(instruction);
  if (instruction->getCondition())
  {
     $UPDATE(0 : 1, 2);
     if ($S)
     {
        if ($d == ARMGPR::PC)
        {
           if (ARMParserCache::hasSPSR(instruction))
               $UPDATE(CPSR : [SPSR]);
           else
               instruction->setUnpredictable();
         }
         else if ($FLAG)
           $UPDATE(CPSR_N, CPSR_Z, CPSR_C, CPSR_V : 0);
      }
  }
};
```

В результате обработки этого описания будет получен следующий код:

```
// cccc00I0101Sssssddddiiiiiiiiiiii
static class ARMInstructionADC : public
ARMInstructionBehavior
{
private:
  virtual bool isJump(const ARMInstruction *instruction)
const
  {
     Q_UNUSED(instruction);
     return ((instruction->getOpcode() >> 12) & ((1 <<
4) - 1))==ARMGPR::PC;
  }
  virtual bool isConditionalJump(const ARMInstruction
*instruction) const
  {
     Q_UNUSED(instruction);
     return (((instruction->getOpcode() >> 12) & ((1 <<
4) - 1))==ARMGPR::PC)&&(((instruction->getOpcode() >>
28) & ((1 << 4) - 1))!=AL);
  }
```

```
  virtual const char *getMnemonic(const ARMInstruction
*instruction) const
  {
     Q_UNUSED(instruction);
     static char buffer[32];
     sprintf(buffer, "%s%s%s", "ADC",
getConditionMnemonic(instruction), (((instruction-
>getOpcode() >> 20) & ((1 << 1) - 1))?"S":""));
     return buffer;
  }
  virtual void parseOperands(ARMInstruction
*instruction) const
  {
     Q_UNUSED(instruction);
     parseGPR<12,4>(instruction);
     parseGPR<16,4>(instruction);
     parseShifterImm32<0,12>(instruction);
  }
  virtual void decomposeInstruction(ARMInstruction
*instruction) const
  {
     const InstructionParseJob *job = instruction-
>getParseJob();
     Q_UNUSED(job);

  parseCondition(instruction);
  if (instruction->getCondition())
  {
     if (((job->decompositionFlags &
InstructionParseJob::DDF_FLAG_DEPS) != 0))
         {
            Dep *dep = instruction->addUpdateDep();
            instruction->operand(0)->addToOutputOf(dep);
            instruction->operand(1)->addToInputOf(dep);
            instruction->operand(2)->addToInputOf(dep);
            dep->Ins(1,
ARMParserCache::getRegister(ARMRegs::CPSR_C)->element);
         }
     else
         {
            Dep *dep = instruction->addUpdateDep();
            instruction->operand(0)->addToOutputOf(dep);
            instruction->operand(1)->addToInputOf(dep);
            instruction->operand(2)->addToInputOf(dep);
         }
```

```
     if (((instruction->getOpcode() >> 20) & ((1 << 1) -
1)))
     {
         if (((instruction->getOpcode() >> 12) & ((1 <<
4) - 1)) == ARMGPR::PC)
         {
            if (ARMParserCache::hasSPSR(instruction))
               {
                   Dep *dep = instruction-
>addUpdateDep();
                   dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR)->element);
                   dep->Ins(1,
ARMParserCache::getBankedRegister(ARMRegs::SPSR,
instruction)->element);
                }
            else
                instruction->setUnpredictable();
         }
         else if (((job->decompositionFlags &
InstructionParseJob::DDF_FLAG_DEPS) != 0))
            {
               Dep *dep = instruction->addUpdateDep();
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_N)->element);
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_Z)->element);
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_C)->element);
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_V)->element);
               instruction->operand(0)-
>addToInputOf(dep);
            }
     }
  }
  }
} instructionADC;
```

5. Преимущества и недостатки метода по сравнению с уже существующими

Данный метод достаточно универсален, поэтому его можно применять для различных процессорных архитектур.

Одним из основных преимуществ является возможность генерации дерева выбора инструкций по их двоичным кодам. Это позволяет экономить время на разработку алгоритма опознавания инструкции, так как он может быть использован повторно, а так же сравнительно легко заменен в случае, если требуется его оптимизация по производительности.

Кроме того, модули поддержки архитектуры, которые разрабатываются с помощью данного метода, значительно проще отлаживать, чем, например, написанные с помощью макросов. Некоторые ошибки, в частности, в описании маски кода инструкции, или в описании декомпозиции, могут быть обнаружены уже на этапе компиляции, что значительно упрощает их поиск. Это позволяет выявлять часть ошибок еще до начала этапа тестирования модуля.

Особенность реализации модуля поддержки архитектуры с использованием шаблонных функций (как было сделано в реализации модуля поддержки Intel x64) заключается в их инстанцировании с различными параметрами, что позволяет зафиксировать данные параметры на этапе компиляции. Это дает возможность компилятору не использовать стек для передачи данных параметров, а также оптимизировать использующий их код. Хотя предлагаемый метод не использует данную возможность в полной мере, он может быть легко расширен в случае, если понадобится оптимизировать производительность модуля поддержки архитектуры таким образом.

При разработке с использованием предлагаемого метода значительно сокращается объем исходного кода, написанного вручную. Описание архитектуры на языке команд и зависимостей гораздо более лаконично, чем код разборщиков инструкций на C++. Также, в кратком описании команды гораздо легче найти ошибку, чем в коде разборщика, так как описание команды более обозримо. Описывая инструкции и зависимости на специальном языке, разработчик может полностью сосредоточиться на особенностях целевой платформы, а не на том, как их описать на языке C++. Сложность реализации разборщиков перекладывается на генератор.

Таким образом, данный метод позволяет сократить ресурсы, затрачиваемые на разработку модуля поддержки архитектуры, а также количество допущенных во время разработки ошибок.

6. Заключение

В данной работе описан метод разработки модулей поддержки архитектуры для среды TrEx, основанный на применении промежуточного языка для описания свойств и декомпозиций инструкций. Этот метод обладает рядом преимуществ по сравнению с уже существующими, поэтому он был выбран для разработки модулей поддержки архитектуры ARM и Power PC. В дальнейшем созданный в рамках этого метода язык описания команд и

зависимостей, а также генератор разборщиков инструкций, может быть применен для разработки модулей поддержки других RISC-архитектур.

Основной сложностью при разработке модулей поддержки архитектур является тестирование функций декомпозиции данных. Поэтому одним из направлений дальнейшего развития разработанного языка является автоматизация создания сценариев тестирования модуля при помощи специальных атрибутов, встроенных в описания команд.

Другим возможным направлением развития является расширение языка для автоматизированного создания некоторых вспомогательных модулей, предоставляющих интерфейсы для взаимодействия алгоритмов анализа поведения программы и модулей поддержки архитектуры.

Литература

[1] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды института системного программирования. М.: ИСП РАН, 2009. с. 51-72.

[2] ARM Architecture Reference Manual. http://www.arm.com/miscPDFs/14128.pdf

[3] PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2/$file/6xx_pem.pdf

Генерация тестовых программ для микропроцессоров на основе шаблонов конвейерных конфликтов

Д.Н. Воробьев, А.С. Камкин
{vorobyev, kamkin}@ispras.ru

Аннотация. В работе рассматривается методика автоматизированного построения тестовых программ для верификации управляющей логики микропроцессоров. Методика основана на формальной спецификации системы команд и описании шаблонов конфликтных ситуаций возможных в работе конвейера тестируемого микропроцессора. Использование формальных спецификаций позволяет автоматизировать разработку генератора тестовых программ и систематично протестировать управляющую логику. В то же время, поскольку подход основан на высокоуровневых описаниях, не учитывающих потактовое функционирование конвейера, разработанные спецификации и шаблоны, а также сгенерированные по ним тестовые программы допускают повторное использование при изменении микроархитектуры. Это позволяет применять методику на ранних стадиях разработки микропроцессоров, когда возможна частая переработка проектных решений.

1. Введение

Работа современного микропроцессора с конвейерной архитектурой организована очень сложным образом. Одновременно он может обрабатывать несколько инструкций, которые, к тому же, могут взаимодействовать через общие ресурсы. На каждом такте микропроцессор принимает решения, связанные с разрешением конфликтов между инструкциями, изменением потока управления и обработкой исключений. Набор механизмов микропроцессора, отвечающих за управление выполнением инструкций на конвейере, называется *управляющей логикой*. Образно говоря, управляющая логика — это «нервная система» микропроцессора, без которой не возможна согласованная работа его «органов» — модулей и подсистем.

Управляющая логика является ключевым компонентом микропроцессора, и, соответственно, ее проектирование и тестирование должно проводиться с

особой тщательностью. На практике же создание тестов[1], как правило, осуществляется вручную или с использованием случайной генерации. Очевидно, что ручная разработка является непродуктивным подходом, используя который невозможно разработать качественный набор тестов за приемлемое время. Случайная генерация, хотя и позволяет быстро обнаружить многие ошибки в проекте, не дает никаких гарантий относительно полноты тестирования. В последнее время стали появляться методы автоматизации тестирования, использующие потактовые модели управляющей логики. Такие методы нацелены на детальную верификацию управляющих механизмов микропроцессора, но они не приживаются в инженерной среде из-за сложности разработки и поддержки моделей.

В настоящей работе рассматривается подход к генерации тестовых программ для проверки управляющей логики, который «расположен» между случайной генерацией и тестированием на основе точных моделей. В основе подхода лежит формальная спецификация системы команд, описывающая по отдельности инструкции микропроцессора, не вдаваясь в особенности их совместного выполнения на конвейере. Цель генерации задается с помощью тестового покрытия, определяющего шаблоны различных взаимодействий, возникающих на конвейере. Модели системы команд, безусловно, менее информативны по сравнению с точными моделями управляющей логики, однако у них есть ряд практических преимуществ. Прежде всего, такие модели существенно проще для разработки, и, кроме того, они могут быть повторно использованы даже при кардинальных изменениях микроархитектуры.

Оставшаяся часть статьи организована следующим образом. Во втором разделе говорится об управляющей логике микропроцессора, в нем также определяется понятие конвейерного конфликта. Третий раздел посвящен анализу существующих подходов к тестированию управляющей логики. В четвертом разделе рассматриваются основные концепции комбинаторной генерации тестовых программ, на которых основана предлагаемая методика, — сама методика изложена в пятом разделе. Шестой раздел описывает опыт практической апробации подхода. Наконец, в седьмом разделе делается заключение, и очерчиваются направления дальнейших работ.

2. Управляющая логика микропроцессора

Под *управляющей логикой микропроцессора* (*control logic*) понимается его внутренняя функциональность, отвечающая за управление выполнением инструкций на конвейере. Для обозначения подсистемы микропроцессора,

[1] Тесты для проверки управляющей логики микропроцессора обычно имеют форму программ в соответствующей системе команд. Программы, созданные специально для тестирования, называются *тестовыми программами*. В данной статье везде под *тестами* понимаются именно тестовые программы.

реализующей управляющую логику, обычно используют термин *устройство управления* (*control unit*). Последний термин, однако, следует трактовать достаточно широко — функции управления могут и не оформляться в отдельное устройство, а быть распределенными между разными модулями микропроцессора.

Управляющая логика имеет сложную иерархическую структуру, на разных уровнях которой решаются разные задачи управления конвейером микропроцессора. Например, на верхнем уровне может осуществляться трансляция инструкций микропроцессора во внутренние микроинструкции ядра, их распределение по каналам конвейера; на следующем уровне — декомпозиция микроинструкций на еще более мелкие операции; наконец, на уровне отдельных модулей — управление обработкой модульных операций. В рамках данной статьи нас в первую очередь интересует управляющая логика верхнего уровня.

Рассмотрим кратко устройство конвейера микропроцессора и особенности его работы. Классический конвейер состоит из пяти стадий[2]: *выборка инструкции* (*IF, Instruction Fetch*), *декодирование* (*ID, Instruction Decode*), *выполнение* (*EX, EXecute*), *доступ к памяти* (*MEM, MEMory access*) и *запись результата* (*WB, Write Back*) [1]. В идеальном случае выполнение инструкций на конвейере осуществляется, как показано на Рис. 1. На каждом такте осуществляется выборка очередной инструкции из памяти, при этом стадия *IF* инструкции с номером *N* выполняется параллельно со стадией *ID* инструкции *N*-1, стадией *EX* инструкции *N*-2, стадией *MEM* инструкции *N*-3 и стадией *WB* инструкции *N*-4.

№ инструкции (↓)	Стадии конвейера								
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB
№ такта (→)	1	2	3	4	5	6	7	8	9

Рис. 1. Выполнение инструкций на конвейере.

[2] В современных микропроцессорах длина конвейера может превышать 20-30 стадий, например, в микропроцессоре Pentium 4 (микроархитектуры Prescott или Cedar Mill) конвейер состоит из 31 стадии.

Такая ситуация возможна, только если выполняемые инструкции являются независимыми. В случае же если, например, результаты одной инструкции подаются на вход другой (то есть имеет место *зависимость по регистрам* типа «запись-чтение»), может произойти *приостановка* (*stalling*) выполнения второй инструкции (а также инструкций, следующих за ней) до тех пор, пока не будет вычислен результат первой[3]. Другим примером нарушения нормального выполнения инструкций на конвейере является обработка *переходов* — инструкций, изменяющих *поток управления*. После выборки инструкции перехода загрузка конвейера приостанавливается до тех пор, пока не будет вычислен адрес перехода и значение истинности условия (если переход условный)[4].

Ситуации, в которых происходят приостановки или сбросы конвейера, называются *конвейерными конфликтами* (*hazards*). Конфликты, в которых две инструкции пытаются обратиться для чтения или записи к одним и тем же данным, называются *конфликтами по данным* (*data hazards*). Рассмотрим их подробнее. Пусть две инструкции *i* и *j* обращаются к общим данным, причем инструкция *i* расположена перед инструкцией *j* по потоку управления. Паттерсон и Хеннесси приводят следующую классификацию конфликтов по данным [2]:

- *чтение после записи* (*RAW, Read After Write*) — инструкция *j* пытается прочитать данные до того, как они записаны инструкцией *i*;
- *запись после записи* (*WAW, Write After Write*) — инструкция *j* пытается записать данные до того, как они записаны инструкцией *i*;
- *запись после чтения* (*WAR, Write After Read*) — инструкция *j* пытается записать данные до того, как они прочитаны инструкцией *i*.

Помимо конфликтов по данным возможны и другие типы конвейерных конфликтов, например, *структурные конфликты* (*structural hazards*), связанные с использованием общих модулей микропроцессора, и *конфликты по управлению* (*control hazards*), возникающие при обработке переходов.

[3] Возможны и более сложные схемы аппаратного разрешения зависимостей, когда происходит переупорядочивание инструкций с тем, чтобы минимизировать число остановок конвейера.

[4] Возможна и другая организация конвейера, когда после выборки инструкции перехода загрузка конвейера не прекращается, но все инструкции, загруженные после нее, *сбрасываются* (*flushing*), если принимается решение о выполнении перехода. В современных микропроцессорах для постоянной загрузки конвейера (минимизации числа остановок и сбросов) применяется *прогнозирование переходов* (*branch prediction*).

Разрешение разного типа конфликтов является важной функцией управляющей логики микропроцессора[5].

Еще одной задачей, за решение которой отвечает управляющая логика, является *обработка исключений*. Исключением называется событие, сигнализирующее о возникновении нештатной ситуации во время выполнения инструкции. При возникновении исключения значение счетчика адреса (адреса текущей инструкции) сохраняется в специальном регистре микропроцессора, а управление передается на программу обработки исключения, при этом все инструкции, загруженные на конвейер после инструкции, вызвавшей исключение, сбрасываются. В широком понимании, исключение является специфической разновидностью конвейерного конфликта, поскольку препятствует нормальному выполнению инструкций на конвейере.

Настоящая работа сфокусирована на тестировании механизмов управляющей логики, отвечающих за разрешение конфликтов (включая, в частности, обработку исключений). Несмотря на то, что в статье рассматривается построение тестов для разных типов конвейерных конфликтов, в том числе конфликтов по управлению, более подробно вопросы тестирования механизмов обработки переходов (по сути, разрешения конфликтов по управлению) изложены в работе [3], опубликованной в этом же сборнике.

3. Обзор существующих работ

В работе [4] описываются две взаимодополняющие техники: генерация тестов на основе *проверки моделей* (*model checking*) и генерация тестов на основе *шаблонов* (*template-based procedures*). Исходной информацией для обоих подходов является спецификация микропроцессора на языке EXPRESSION [5]. На основе спецификации автоматически строится обобщенная структурно-событийная модель на SMV [6], для которой разработчик тестов определяет *модель ошибок* (набор свойств, описывающих классы ошибок). Для каждого элемента модели ошибок строится отрицание соответствующего свойства, а для полученного отрицания с помощью SMV генерируется контрпример. Как отмечают авторы, подход на основе проверки моделей не масштабируется на сложные микропроцессоры, поэтому в качестве дополнения к нему они предлагают технику на основе шаблонов. Шаблоны разрабатываются вручную и описывают последовательности инструкцией, приводящие к возникновению определенных ситуаций в работе конвейера (прежде всего, конфликтов). На основе шаблонов конструируются более сложные тестовые программы, покрывающие модель ошибок. Генерация осуществляется на основе графовой модели, извлеченной из спецификации микропроцессора. Подход на основе шаблонов предполагает

[5] Некоторые микропроцессорные архитектуры возлагают разрешение конфликтов на программистов или компиляторы.

больший объем ручного труда, но при этом лучше масштабируется. Заметим, что обе техники используют детальные спецификации управляющей логики микропроцессора, поэтому их лучше применять на поздних стадиях проектирования. При изменении тестируемой модели микропроцессора необходимо переделывать описание на EXPRESSION, что требует значительных затрат.

В подходе, предложенном в статье [7], формально специфицируется структура конвейера в форме автоматной модели, называемой OSM (Operation State Machine). OSM моделирует управляющую логику микропроцессора на двух уровнях: *операционном* (*operational*) и *аппаратном* (*hardware*). На операционном уровне с помощью расширенных конечных автоматов описывается «движение» инструкций через стадии конвейера (каждая операция описывается отдельным автоматом). На аппаратном уровне моделируются ресурсы микропроцессора в форме так называемых *менеджеров маркеров* (*token manager*s). Автомат, описывающий отдельную инструкцию, переходит из одного состояния в другое путем захвата и освобождения маркеров. Модель конвейера представляет собой композицию автоматов операций и автоматов ресурсов. Целью тестирования является проход по всем переходам совокупной автоматной модели. Данный подход требует разработки детальных OSM-спецификаций, что является трудоемкой работой, поэтому, как и предыдущие техники, его нецелесообразно применять на ранних этапах проектирования.

В работе [8] рассказывается об инструменте Genesys-Pro, предназначенном для генерации тестовых программ на основе *шаблонов* (*templates*). Генератор состоит из двух основных компонентов: независимого от целевого микропроцессора *ядра* (*engine*) и *модели* (*model*), описывающей тестируемый микропроцессор на уровне инструкций. Инженер-верификатор разрабатывает на специальном языке шаблоны, которые задают структуру тестовых программ и описывают свойства, которым она должна удовлетворять. Genesys-Pro транслирует каждый шаблон в систему ограничений и строит тестовую программу, используя техники *разрешения ограничений* (*constraint solving*). Преимуществами инструмента является его переносимость на разные микропроцессорные архитектуры и богатый язык описания шаблонов. Однако, поскольку разработка шаблонов осуществляется вручную, поддержка разработанных тестов достаточна сложна, а качество тестирования напрямую зависит от квалификации инженеров-верификаторов.

Статья [9] описывает метод генерации тестов на основе *обхода графа состояний* автоматной модели конвейера. В одном из своих этапов метод использует генератор Genesys (который впоследствии развился в Genesys-Pro). Суть метода заключается в следующем. Разработчик тестов создает модель микропроцессора на SMV. После этого с помощью инструмента CFSM строится множество путей (так называемых *абстрактных тестов*), покрывающих все дуги в графе состояний конечного автомата, извлеченного

из модели. Абстрактные тесты транслируются в шаблоны генератора Genesys, который на их основе генерирует тестовые программы. Метод позволяет достичь хорошего покрытия управляющей логики, но у него есть два основных недостатка. Во-первых, необходим опытный эксперт для разработки модели микропроцессора. Во-вторых, для возможности отображения абстрактных тестов в шаблоны нужно создать достаточно сложное описание в Genesys.

Рассмотренные выше подходы к тестированию управляющей логики микропроцессора можно разбить на два класса: *методы на основе точных моделей* [4,7,9] и *методы на основе шаблонов* [8]. Методы на основе *точных моделей*, то есть моделей, описывающих управляющую логику с потактовой или почти потактовой точностью, позволяют добиться очень высокого качества тестирования. Главным недостатком таких подходов является невозможность или нецелесообразность их использования на ранних этапах проектирования, когда управляющая логика микропроцессора не полностью определена или определена, но часто меняется. Методы генерации тестов на основе шаблонов лишены этого недостатка, но у них есть другой серьезный изъян — они лишены систематичности и не позволяют адекватно оценивать качество тестирования.

Подход, предлагаемый в статье, является компромиссом между этими двумя категориями методов. С одной стороны, для автоматизации построения тестов в нем используются модели. С другой стороны, используемые модели не фиксируют жестко управляющую логику микропроцессора — она описывается неявно в виде обобщенных шаблонов конвейерных конфликтов. За счет этого достигается масштабируемость подхода и возможность его применения в условиях частых изменений в проекте.

4. Основные понятия предлагаемого подхода

Подход основан на методе *комбинаторной генерации тестовых программ*, детальное описание которого доступно в работе [10]. В основе метода лежит *формальная спецификация* системы команд, описывающая отдельные инструкции микропроцессора безотносительно того, как они обрабатываются на конвейере. Описание каждой инструкции включает ее мнемонику, список операндов с указанием их типов, множество вызываемых исключений, предусловие, длительность обработки в тактах и семантику в императивной форме. Кроме того, формально в форме *тестовых ситуаций* и *зависимостей* описываются ситуации на обработку инструкций на конвейере. Генерация тестовых программ осуществляется автоматически путем комбинирования тестовых ситуаций и зависимостей для последовательностей инструкций ограниченной длины. Рассмотрим основные понятия метода комбинаторной генерации, которые важны для описания предлагаемого подхода.

4.1. Структура тестовой программы

Тестовая программа представляет собой последовательность *тестовых вариантов* (*test cases*). Каждый тестовый вариант содержит *тестовое воздействие* — специально подготовленную цепочку инструкций, предназначенную для создания определенной ситуации в работе конвейера микропроцессора. Перед тестовым воздействием помещаются *инициализирующие инструкции*, а после него — *тестовый оракул* — набор инструкций, проверяющих корректность состояния микропроцессора после выполнения тестового воздействия.

Таким образом, структуру тестовой программы можно описать с помощью формулы $Test = \{\langle Pre_i, Action_i, Post_i\rangle\}_{i=0,n-1}$, где Pre_i — это инициализирующие инструкции, $Action_i$ — тестовое воздействие, $Post_i$ — тестовый оракул. В простейшем случае каждая тестовая программа состоит из одного тестового варианта, то есть $Test = \langle Pre, Action, Post\rangle$, кроме того, тестовые оракулы часто опускаются (оценка правильности поведения осуществляется на основе сравнения с эталонной моделью) — $Test = \langle Pre, Action\rangle$.

Ниже в качестве примера приведен фрагмент тестовой программы в системе команд MIPS [11], содержащий один тестовый вариант.

```
// Инициализация инструкции sub[0]: IntegerOverflow=true
// s5[rs]=0xffffffffc1c998db, v0[rt]=0x7def4297
lui s5, 0xc1c9
ori s5, s5, 0x98db
lui v0, 0x7def
ori v0, v0, 0x4297

// Инициализация инструкции add[1]: Exception=false
// a0[rs]=0x1d922e27, a1[rt]=0x32bd66d5
lui a0, 0x1d92
ori a0, a0, 0x2e27
lui s3, 0x32bd
ori s3, s3, 0x66d5

// Инициализация инструкции div[2]: DivisionByZero=true
// a2[rs]=0x48f, a1[rt]=0x0
lui a2, 0x0
ori a2, a2, 0x48f
lui a1, 0x0

// Зависимости: div[2].rt[1]-sub[0].rd[0]

// Тестовое воздействие: 2010
sub a1, s5, v0 // IntegerOverflow=true
add t7, a0, s3 // Exception=false
div a2, a1     // DivisionByZero=true
```

4.2. Тестовый шаблон

Важным понятием подхода, которое следует рассмотреть подробнее, является понятие *тестового шаблона*. Тестовым шаблоном называется абстрактная форма представления тестового воздействия, в котором вместо конкретных значений операндов инструкций указываются ограничения (*тестовые ситуации* и *зависимости*), которым они должны удовлетворять. Кроме того, вместо определенных инструкций могут быть заданы ограничения на них (обычно в форме *классов эквивалентности* инструкций). По сути, каждый тестовый шаблон задает некоторую *цель* — ситуацию в работе конвейера, которую нужно проверить при тестировании. Задача генерации тестовых программ сводится к построению представительного множества тестовых шаблонов.

Ниже приведен один из возможных тестовых шаблонов для рассмотренного выше примера. Шаблон состоит из трех инструкций: первые две относятся к классу эквивалентности `IADDInstruction`, а третья принадлежит классу `IDIVInstruction`. Для первой инструкции задана тестовая ситуация *IntegerOverflow=true* (возникновение исключения целочисленного переполнения), для второй инструкции ситуацией является *Exception=false* (отсутствие исключений), а для третьей — *DivisionByZero=true* (деление на нуль). Кроме того, между первой и третьей инструкциями имеется зависимость (первый регистр первой инструкции должен совпадать со вторым регистром третьей инструкции).

```
IADDInstruction R, ?, ? @ IntegerOverflow=true → sub a1, s5, v0
IADDInstruction ?, ?, ? @ Exception=false      → add t7, a0, s3
IDIVInstruction ?, R    @ DivisionByZero=true  → div a2, a1
```

Выполнение описываемого этим шаблоном тестового воздействия начинается с первой инструкции (в рассматриваемом примере это `sub`). Данная инструкция вызывает исключение целочисленного переполнения. Предполагается, что программа обработки исключений устроена таким образом, что управление передается на следующую инструкцию (ей является инструкция `add`). После обработки второй инструкции выполняется третья инструкция (инструкция `div`), в которой происходит деление на нуль.

Заметим, что тестовые шаблоны допускают *параметризацию*. В качестве параметров могут выступать классы эквивалентности инструкций, тестовые ситуации и зависимости. В этом случае каждому шаблону соответствует не одна цель тестирования, а целое семейство целей, для каждой из которых строится свое тестовое воздействие. Описание семейства осуществляется с помощью *итераторов* — компонентов генератора, перебирающих допустимые комбинации значений параметров. Ниже приведен пример шаблона с четырьмя параметрами: `$FirstInstruction` (класс

эквивалентности первой инструкции тестового воздействия), `$Situation` (тестовая ситуация первой инструкции), `$ThirdInstruction` (класс эквивалентности третьей инструкции) и `$Dependency` (множество зависимостей третьей инструкций от других инструкций тестового воздействия).

```
$FirstInstruction @ $Situation
IADDInstruction   @ IntegerOverflow=false
$ThirdInstruction @ $Dependency
```

Настоящая работа сфокусирована на тестировании механизмов управляющей логики, связанных с разрешением конфликтов, поэтому нас в первую очередь интересует описание и построение *шаблонов конвейерных конфликтов* — тестовых шаблонов, вызывающих конфликтные ситуации в работе конвейера. Рассмотрим, какие тестовые ситуации и зависимости для этого используются.

4.3. Тестовые ситуации

Для тестирования управляющей логики основной интерес представляют тестовые ситуации, связанные с выполнением инструкций на конвейере. Как правило, обработка любой инструкции происходит одинаковым образом для всех значений операндов (конечно, если не брать в расчет исключения). Таким образом, для инструкции, которая может вызвать N исключений, обычно определяется N+1 тестовая ситуация: *Exception=false*, $Exception_0$=*true*, …, $Exception_{N-1}$=*true*. Для инструкций, обработка которых зависит от значений операндов, возможна дополнительная детализация тестовых ситуаций.

Отдельно рассматриваются инструкции перехода. Тестовые ситуации для них описывают положение адреса (метки) перехода в тестовом воздействии и выполнимость условий (если переход условный). В общем случае тестовая ситуация для инструкции перехода имеет вид *Target=Label*, *Trace*=$\{C_0, \ldots, C_{M-1}\}$, где *Label* — метка инструкции тестового воздействия, на которую осуществляется переход, C_i — выполнимость условия перехода при i-ой обработке инструкции. В некоторых случаях положение метки может быть обобщено до направления перехода (вперед или назад). Более подробно тестирование механизмов обработки переходов рассматривается в статье [3].

4.4. Зависимости между инструкциями

Зависимостям между инструкциями отводится ключевая роль в создании конвейерных конфликтов. Они бывают двух основных типов: *по регистрам* и *по адресам*. Регистровые зависимости выражаются через совпадение регистров, использующихся в качестве операндов двух инструкций тестового воздействия, и бывают следующих типов:

- *чтение-чтение* — обе инструкции осуществляют чтение из одного регистра;

- *чтение-запись* — первая по потоку управления инструкция осуществляет чтение из регистра, вторая — запись в него;
- *запись-чтение* — первая инструкция осуществляет запись в регистр, вторая — чтение из него;
- *запись-запись* — обе инструкции осуществляют запись в один регистр.

Зависимости по адресам имеют более сложную структуру и связаны с внутренним устройством подсистемы управления памятью (организацией буфера трансляции адресов, кэш-памяти и основной памяти). Примеры зависимостей по адресам приведены ниже:

- *VAEqual* — совпадение виртуальных адресов;
- *TLBEqual* — совпадение записей в буфере трансляции адресов;
- *PAEqual* — совпадение физических адресов;
- L_i*RowEqual* — совпадение строк в кэш-памяти L_i;

Более подробно типы зависимостей по адресам и другие вопросы, связанные с тестированием подсистем управления памятью, рассматриваются в работе [12].

5. Предлагаемый подход

Предлагаемый подход к построению тестовых программ состоит в следующем. На основе анализа документации выделяется набор «интересных» ситуаций в управляющей логике микропроцессора. Основной упор делается на разрешение конфликтов (включая обработку исключений). Для каждого типа конфликта разрабатывается его *обобщенная спецификация*[6] — параметризованный шаблон, позволяющий создать соответствующую конфликтную ситуацию. Такие шаблоны также называются *шаблонами конвейерных конфликтов* или *базовыми шаблонами*. Базовые шаблоны имеют небольшой размер, поскольку конфликты между инструкциями возможны, только если они расположены достаточно близко друг к другу. Для построения тестов задаются *итераторы* значений параметров базовых шаблонов, после чего генератор строит тестовые программы, используя разные значения параметров шаблонов и комбинируя шаблоны между собой.

[6] Спецификация называется обобщенной, поскольку она основывается на общих сведениях о микропроцессоре и не содержит информации о потактовом выполнении инструкций. Управляющая логика определяется неявно путем описания возможных взаимодействий инструкций на конвейере.

5.1. Спецификация конфликтов

Рассмотрим общую схему спецификации конфликтных ситуаций в работе конвейера. Все выделенные на этапе анализа требований ситуации классифицируются по типам. Основными типами являются *исключения*, *конфликты по данным*, *структурные конфликты* и *конфликты по управлению* (см. Рис. 2).

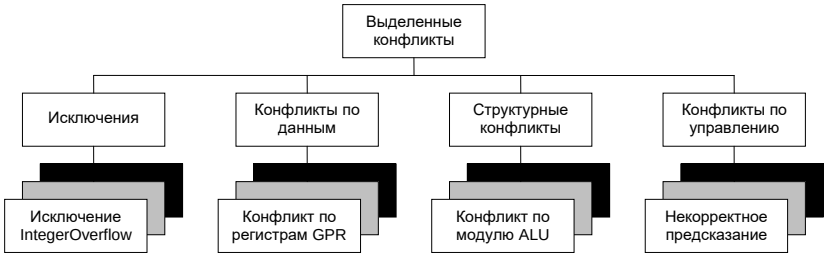


Рис. 2. Выделение конфликтных ситуаций.

Как правило, все конфликтные ситуации, относящиеся к одному типу, описываются одним базовым шаблоном. Различие в их спецификации заключается в разных ограничениях на допустимые значения параметров шаблона (см. Рис. 3). Кроме того, множества значений параметров шаблона разбиваются разработчиком тестов на классы эквивалентности, что служит основой для дальнейшего построения итераторов.

Рис. 3. Спецификация конфликтной ситуации.

Рассмотрим подробнее, как описываются конфликтные ситуации для каждого из указанных типов: исключений, конфликтов по данным, структурным конфликтов и конфликтов по управлению.

5.1.1. Спецификация исключений

Исключением называется событие, сигнализирующее о возникновении нештатной ситуации во время выполнения инструкции. При возникновении

исключения управление передается на специальный обработчик, при этом все инструкции, загруженные на конвейер после инструкции, вызвавшей исключение, сбрасываются. Типичными ошибками в управляющей логике, связанными с обработкой исключений, являются некорректная установка сигнала исключения (например, когда сигнал не устанавливается, хотя условия для исключения выполнены) и некорректная отмена инструкций, загруженных на конвейер.

В тестовых программах, в которых возможны исключения, используются две основные стратегии передачи управления после обработки исключения: управление передается на следующую инструкцию тестового воздействия или управление передается на тестовый оракул, минуя оставшиеся инструкции. Для проверки того, что при возникновении исключения, с конвейера сбрасываются все инструкции, следующие за инструкцией, вызвавшей исключение, предпочтительно использовать вторую стратегию.

Обобщенная спецификация исключения задается базовым шаблоном, представленным ниже.

```
$PreInstructions
$ExceptionInstruction @ $ExceptionType
$PostInstructions
```

В шаблоне используются следующие параметры:

- `$PreInstructions` — цепочка инструкций, предшествующих исключению, (инструкции не должны вызывать исключений);
- `$ExceptionInstruction` — инструкция, вызывающая исключение;
- `$ExceptionType` — тип вызываемого исключения;
- `$PostInstructions` — цепочка инструкций, следующих за исключением, (инструкции, которые должны быть сброшены с конвейера после возникновения исключения).

Для разных типов исключений интерес могут представлять разные типы инструкций, предшествующих исключению и следующих за ним. Ниже приведен пример конкретного тестового воздействия, соответствующего данному базовому шаблону.

```
$PreInstructions                      → dadd   r25, r30, r7
$ExceptionInstruction @ ExceptionType→ lb     r22, 0(r4) //
TLBInvalid=true
$PostInstructions                     → daddiu r5,  r18, 13457
```

Цепочка `$PreInstructions` состоит из одной инструкции `dadd`. В качестве `$ExceptionInstruction` выступает инструкция `lb`,

вызывающая исключение *TLBInvalid* (`$ExceptionType`). Цепочка `$PostInstructions` состоит из одной инструкции `daddiu`.

5.1.2. Спецификация конфликтов по данным

Конфликты по данным возникают, когда разные инструкции пытаются обратиться для чтения или записи к одним и тем же данным, причем хотя бы одна инструкция обращается к ним на запись. Таким образом, для описания конфликтов по данным основной интерес представляют зависимости типа «чтение-запись», «запись-чтение» и «запись-запись», а также их комбинации. Для разных конфликтов могут использоваться разные типы зависимостей. Типичной ошибкой, связанной с разрешением конфликтов по данным, является некорректная реализация блокировок в конвейере, в результате чего происходит нарушение целостности потока данных.

Обобщенная спецификация конфликта по данным задается базовым шаблоном, приведенным ниже.

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

В шаблоне используются следующие параметры:

- `$PreInstructions` — цепочка инструкций, предшествующих зависимости, (инструкции не должны вызывать исключений);
- `$FirstInstruction` и `$SecondInstruction` — пара инструкций, вступающих в конфликт по данным;
- `$Dependency` — зависимость, приводящая к конфликту по данным;
- `$InnerInstructions` — цепочка инструкций между зависимыми инструкциями (инструкции не должны вызывать исключений и содержать конфликты);
- `$PostInstructions` — цепочка инструкций, следующих за зависимостью, (инструкции, которые могут быть приостановлены вместе с зависимой инструкцией).

Конкретный тип конфликта по данным определяется типами зависимых инструкций и типом зависимости между ними. Ниже приведен пример тестового воздействия, соответствующего данному базовому шаблону.

```
$PreInstructions                →
$FirstInstruction               →madd.s $f18, $f6,  $f28, $f10
$InnerInstructions              →add.s  $f8,  $f17, $f3
$SecondInstruction @ $Dependency→ceil.l.s $f2,  $f18  //
Конфликт по данным

$PostInstructions               →div.s    $f23, $f13, $f24
```

В качестве пары инструкций, вступающих в конфликт, выступают `madd.s` (`$FirstInstruction`) и `ceil.l.s` (`$SecondInstruction`). Они связаны зависимостью типа «запись-чтение» по регистру `$f18`. Цепочка `$PreInstructions` является пустой, `$InnerInstructions` состоит из одной инструкции `add.s`, `$PostInstructions` — из одной инструкции `div.s`.

5.1.3. Спецификация структурных конфликтов

Структурные конфликты возникают, когда несколько инструкций пытаются одновременно обратиться к одному модулю микропроцессора, который не допускает параллельной обработки операций. Как правило, структурные конфликты связаны с близким расположением в коде двух однотипных инструкций, выполнение которых занимает более одного такта (имеется в виду длительность выполнения стадии *EX*). В некоторых случаях для создания конфликта дополнительно требуется наличие зависимости между инструкциями, например, при обращении двух инструкций к кэш-памяти можно выделить в отдельный тип структурного конфликта ситуацию, когда совпадают используемые инструкциями строки кэш-памяти. Типичные ошибки, связанные с разрешением структурных конфликтов, — это, как и для конфликтов по данным, ошибки в реализации блокировок.

Обобщенная спецификация структурного конфликта задается точно таким же базовым шаблоном, что и обобщенная спецификация конфликта по данным. Ниже приведен пример тестового воздействия, в котором создается структурный конфликт.

```
$PreInstructions                →
$FirstInstruction               →  div.s $f11, $f27, $f3
$InnerInstructions              →  add.s $f28, $f7,  $f30
$SecondInstruction @ $Dependency→  div.d $f23, $f1,  $f20 //
Структурный конфликт
$PostInstructions               →  add.d $f18, $f2,  $f25
```

В данном примере все инструкции задействуют модуль арифметики с плавающей точкой (FPU, Floating Point Unit), однако структурный конфликт возникает только между инструкциями деления `div.s` (`$FirstInstruction`) и `div.d` (`$SecondInstruction`) — их выполнение, в отличие от инструкций сложения `add.s` и `add.d`, занимает более одного такта. Цепочка `$PreInstructions` является пустой,

$InnerInstructions состоит из одной инструкции add.s, $PostInstructions — из одной инструкции add.d.

5.1.4. Спецификация конфликтов по управлению

Конфликты по управлению связаны с обработкой инструкций перехода. В зависимости от организации микропроцессора выполнение перехода может приводить к приостановке конвейера (когда после выборки инструкции перехода загрузка конвейера приостанавливается до тех пор, пока не будет принято решение о переходе) или сбросу конвейера (когда все инструкции, загруженные на конвейер после инструкции перехода, сбрасываются, если принимается решение о выполнении этого перехода). Ошибки в разрешении конфликтов по управлению могут быть связаны с блокировками конвейера, предсказанием переходов и другими механизмами обработки переходов.

Обобщенная спецификация конфликта по управления задается базовым шаблоном, приведенным ниже.

```
$PreInstructions
$BranchInstruction @ $Target, $Trace
$DelaySlots
$PostInstructions
```

В шаблоне используются следующие параметры:

- $PreInstructions — инструкции, предшествующие переходу;
- $BranchInstruction — инструкция перехода;
- $Target — адрес перехода (указывает на одну из инструкций шаблона);
- $Trace — трасса выполнения перехода (последовательность из значений истинности условия перехода);
- $DelaySlots — инструкции в слотах задержки[7];
- $PostInstructions — инструкции, следующие за переходом.

Подробнее вопросы тестирования модулей обработки переходов рассматриваются в работе [3]. Ниже приведен пример тестового воздействия, соответствующего данному базовому шаблону.

```
$PreInstructions                      →  L:addi r1, r1,  1
$BranchInstruction @ $Target, $Trace→     beq  r1, r0,  L //
Target=L, Trace={1, 0}
$DelaySlots                           →     dadd r7, r12, r23
$PostInstructions                     →
```

[7] Инструкции, следующие за инструкцией перехода, которые всегда выполняются микропроцессором при обработке перехода (не зависимо от того, выполнено условие перехода или нет). Число слотов задержки переходов является постоянной величиной для каждой микропроцессорной архитектуры, например, в MIPS имеется один слот задержки.

В качестве инструкции перехода `$BranchInstruction` выступает инструкций `beq`. Переход осуществляется на первую инструкцию тестового воздействия, помеченную меткой `L` (`$Target`). При первом выполнении перехода условие выполнено, при втором — нет (`$Trace`). Цепочка `$PreInstructions` состоит из одной инструкции `addi`. В слоте задержки перехода находится инструкция `dadd` (`$DelaySlots`). Цепочка `$PostInstructions` является пустой.

5.2. Построение тестовых программ

Теперь рассмотрим, как на основе базовых шаблонов (шаблонов конвейерных конфликтов) строятся тестовые воздействия, используемые в тестовых программах. Сразу отметим, что тестовые воздействия делятся на два типа: *простые* (соответствующие одному базовому шаблону) и *составные* (соответствующие композиции нескольких базовых шаблонов). В соответствии с этим описание методики разбито на две части.

5.2.1. Построение простых тестовых воздействий

Простые воздействия, как правило, нацелены на создание одной конфликтной ситуации в работе конвейера. Методика их построения является простой и основана на использовании *базовых шаблонов* и *итераторов* (см. Рис. 4). Для каждой выделенной ситуации, вообще говоря, строится несколько тестовых воздействий, различающихся значениями параметров (эти значения перебираются в итераторах). Множество воздействий строится путем комбинирования значений параметров (обычно используются все возможные комбинации).

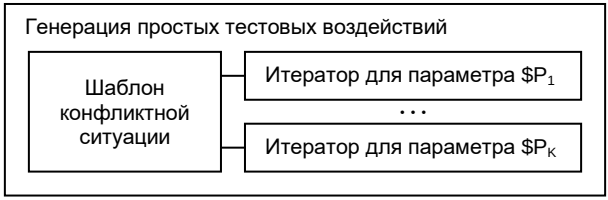


Рис. 4. Генерация простых тестовых воздействий для одной конфликтной ситуации.

Рассмотрим, например, структурный конфликт по модулю FPU, который описывается следующим базовым шаблоном:

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

Для этого конфликта определены некоторые ограничения на значения параметров. Например, конфликт возможен только между инструкциями, выполняющимися более одного такта, при этом длина последовательности `$InnerInstructions` не должна превышать уменьшенной на два такта длительности выполнения инструкции `$FirstInstruction`. Кроме того, могут быть заданы классы эквивалентности зависимых инструкций. Если, например, блок умножения реализован отдельно от блока деления, инструкции умножения и деления могут и не вызывать структурного конфликта, поэтому следует рассматривать два конфликта (между инструкциями умножения и между инструкциями деления).

Пусть для рассматриваемого конфликта определены следующие классы эквивалентности инструкций и итераторы значений параметров:

```
FMULInstruction    : {mul.s, mul.d}
FDIVInstruction    : {div.s, div.d}
IADDInstruction    : {add, sub}

$PreInstructions   : {}
$FirstInstruction  : FMULInstruction, FDIVInstruction
$SecondInstruction : FMULInstruction, FDIVInstruction
$Dependency        : class($FirstInstruction) ==
class($SecondInstruction)
$InnerInstruction  : {IADDInstruction}
$PostInstructions  : {}
```

Тогда будут построены два тестовых воздействия следующего вида:

```
$PreInstructions     →                                  →
$FirstInstruction    →   mul.d $f12, $f3,  $f21   →
div.s $f18, $f28, $f4
$InnerInstructions   →   sub   r6,   r15,  r3     →
add   r25,  r13,  r27
$SecondInstruction   →   mul.s $f9,  $f23, $f7    →
div.s $f5,  $f12, $f10
$PostInstructions    →                                  →
```

5.2.2. Построение составных тестовых воздействий

Целью составных тестовых воздействий, в отличие от простых, является создание нескольких «одновременных» конфликтов в работе конвейера. Составные воздействия позволяют проверить сложные ситуации в управляющей логике микропроцессора. К таким ситуациям относятся *параллельные конфликты* (когда, например, структурный конфликт для пары инструкций возникает одновременно с конфликтом по данным), *вложенные конфликты* (когда один конфликт происходит на фоне другого), *параллельные исключения* (когда за счет того, что исключения в разных инструкциях могут возникать на разных стадиях, они происходят одновременно) и многие другие.

Построение составных воздействий осуществляется путем композиции нескольких базовых шаблонов. Под композицией понимается как простая конкатенация шаблонов, так и более сложная их комбинация, при которой они могут иметь общие части, в частности, при которой один шаблон может быть вложен в другой.

Обозначим буквой T (возможно с индексом) шаблон произвольного типа, T_E — шаблон для исключения, T_H — шаблон для конфликта по данным или структуре, T_C — шаблон для конфликта по управлению. Основные операции композиции шаблонов приведены в следующей таблице[8].

Операция	***Обозначение***	***Семантика***
Наложение	$T=T_{H1}\|T_{H2}$	$T_H.PreInstructions = T_{H1}.PreInstructions \equiv T_{H2}.PreInstructions$ $T_H.FirstInstruction = T_{H1}.FirstInstruction \equiv T_{H2}.FirstInstruction$ $T_H.SecondInstruction = T_{H1}.SecondInstruction \equiv T_{H2}.SecondInstruction$ $T_H.Dependency = T_{H1}.Dependency \,\&\, T_{H2}.Dependency$ $T_H.InnerInstructions = T_{H1}.InnerInstructions \equiv T_{H2}.InnerInstructions$ $T_H.PostInstructions = T_{H1}.PostInstructions \equiv T_{H2}.PostInstructions$
Сдвиг	$T_H=T_{H1}\downarrow T_{H2}$	$T_H.PreInstructions = T_{H1}.PreInstructions$ $T_H.FirstInstruction = T_{H1}.FirstInstruction$ $T_H.SecondInstruction = T_{H1}.SecondInstruction$ $T_H.Dependency = T_{H1}.Dependency \,\&\, T_{H2}.Dependency$ $T_H.InnerInstructions = \{T_{H1}.InnerInstructions, T_{H2}.FirstInstruction, T_{H2}.InnerInstructions\}$ $T_H.PostInstructions = \{T_{H1}.PostInstructions, T_{H2}.SecondInstruction, T_{H2}.PostInstructions\}$ $T_{H1}.PostInstructions \equiv T_{H2}.PreInstructions$
Конкатенация	$T=T_1\rightarrow T_2$	$T.PreInstructions = T_1.PreInstructions$ $T.MainParameters = T_1.MainParameters$[9] $T.PostInstructions = T_2$ Тип шаблона T совпадает с типом шаблона T_1
Вложение	$T_H=T_{H1}[T]$	$T_H.FirstInstruction = T_{H1}.FirstInstruction$ $T_H.SecondInstruction = T_{H1}.SecondInstruction$ $T_H.Dependency = T_{H1}.Dependency$ $T_H.PreInstructions = T_{H1}.PreInstructions$ $T_H.InnerInstructions = T$ $T_H.PostInstructions = T_{H1}.PostInstructions$

Для того чтобы прояснить семантику составных шаблонов и способ построения по ним тестовых воздействий, рассмотрим примеры.

[8] Набор операций композиции является открытым и может быть расширен.

[9] Под *MainParameters* понимается набор параметров шаблона за исключением *PreInstructions* и *PostInstructions*.

Наложение шаблонов:

```
$PreInstructions1,2                    →   add.s  $f28, $f7,  $f30
$FirstInstruction1,2                   →   div.s  $f11, $f27, $f3
$InnerInstructions1,2                  →   dsub   r25,  r30,  r7
$SecondInstruction1,2 @ $Dependency1 & $Dependency2→   div.d
$f23, $f11, $f20
$PostInstruction1,2                    →   lb     r22,  0(r4)
```

Сдвиг шаблонов:

```
$PreInstructions1                      →   add.s  $f28, $f7,  $f30
$FirstInstruction1                     →   div.s  $f11, $f27, $f3
$InnerInstructions1                    →   dsub   r25,  r30,  r7
    $FirstInstruction2                 →   sub    r13,  r5,   r10
    $InnerInstructions2                →   add    r8,   r11,  r3
$SecondInstruction1 @ $Dependency1→    div.d  $f23, $f1,  $f20
    $PostInstrucions1 ≡ $PreInstructions2→ lb     r22,  0(r4)
    $SecondInstruction2 @ $Dependency2→   div    r12,  r13
    $PostInstructions2                 →   daddiu r5,   r18,  54
```

Конкатенация шаблонов:

```
$PreInstructions1                      →   add.s  $f28, $f7,  $f30
$FirstInstruction1                     →   div.s  $f11, $f27, $f3
$InnerInstructions1                    →   dsub   r25,  r30,  r7
$SecondInstruction1 @ $Dependency1→    div.d  $f23, $f1,  $f20
    $PostInstruction1 ≡ $PreInstructions2→lb    r22,  0(r4)
    $FirstInstruction2                    →sub   r13,  r5,   r10
    $InnerInstructions2                   →add    r8,   r11,  r3
    $SecondInstruction2 @ $Dependency2    →div    r12,  r13
    $PostInstructions2                    →daddiu r5,   r18,  54
```

Вложение шаблонов:

```
      $PreInstructions1                →add.s $f28, $f7, $f30
      $FirstInstruction1               →div.s $f11, $f27, $f3
          $PreInstructions2            →lb     r22,  0(r4)
          $FirstInstruction2           →sub   r13,  r5,   r10
          $InnerInstructions2          →add    r8,   r11,  r3
          $SecondInstruction2 @ $Dependency2→div    r12,  r13
          $PostInstructions2           →daddiu r5,   r18,  54
      $SecondInstruction1 @ $Dependency1→div.d $f23, $f1, $f20
      $PostInstructions1               →nop
```

Интуитивно понятно, как осуществляется итерация разных тестовых воздействий для заданного составного шаблона. Некоторые параметры разных базовых шаблонов отождествляются, после этого для полученного множества параметров задаются итераторы (как правило, используются итераторы, заданные для простых воздействий). Тонким моментом является перебор

меток переходов. Если ограничить положение меток только внутренними инструкциями базовых шаблонов получится достаточно ограниченный набор ситуаций, поэтому допускается перебор меток за пределами базовых шаблонов. Генерация составных шаблонов осуществляется путем перебора различных синтаксических структур из небольшого числа базовых шаблонов, связанных операциями композиции.

6. Практическая апробация подхода

Рассматриваемый подход был применен для верификации управляющей логики двух сопроцессоров: сопроцессора вещественной арифметики (CP1) и сопроцессора комплексной арифметики (CP2). Сопроцессоры имеют общий поток инструкций с центральным микропроцессором и используют три канала выполнения (функциональных конвейера):

- канал арифметики с плавающей точкой (вещественной и комплексной);
- канал обращения к основной памяти;
- канал обращений к накристальной памяти.

Управляющая логика микропроцессора поддерживает разрешение конфликтов разных типов. В обоих сопроцессорах могут возникать исключения при доступе к основной памяти. Кроме того, в сопроцессоре CP1 возможны исключения при выполнении арифметических инструкций. В микропроцессоре реализована возможность статического предсказания переходов и спекулятивного выполнения инструкций.

В тестовых программах использовались воздействия из четырех инструкций разных типов (применялись как простые, так и составные воздействия). Структура используемых тестовых ситуаций и зависимостей близка к описанной в статье, однако были учтены некоторые микроархитектурные особенности сопроцессоров путем введения дополнительных зависимостей по данным.

Следующая таблица показывает объем разработанных спецификаций, шаблонов и других компонентов генератора тестовых программ для разных ревизий микропроцессора[10]; число реализуемых сопроцессорами инструкций;

[10] Разработка тестов началась с ревизии №8. В таблице отражены только те ревизии, которые потребовали изменения компонентов генератора тестовых программ.

число инструкций, затронутых изменениями в соответствующей ревизии[11] и долю исходного кода генератора, который пришлось изменить[12].

№ ревизии	*Объем кода (в строках)*	*Число инструкций*	*Число затронутых инструкций*	*Объем измененного кода (в строках)*
Сопроцессор CP1				
8	28500	113	—	—
20	28650	114	94	485 (1.7%)
Сопроцессор CP2				
8	4950	15	—	—
20	11550	59	5	45 (0.9%)
29	14350	100	18	165 (1.4%)

Таб. 1. Данные по апробации предлагаемого подхода.

Как видно из таблицы, при модификации микропроцессора изменяется очень небольшой объем кода генератора. В нашем случае изменению подвергались только спецификации инструкций (мнемоника, число тактов выполнения, набор операндов и, частично, семантика). Для всех указанных ревизий правка занимала не более получаса, а генерация тестовых программ потребовала от 30 минут до 5 часов (в зависимости от детальности используемых зависимостей между инструкциями).

В результате верификации было найдено значительное число ошибок в обоих сопроцессорах, которые не были обнаружены с помощью тестовых программ, сгенерированных случайно. Ошибки были найдены как в эталонном симуляторе, так и в RTL-модели микропроцессора.

7. Заключение

Тестирование управляющей логики микропроцессоров является нетривиальной задачей, которую практически невозможно решить без применения методов автоматизации. В статье рассмотрена методика автоматизированной генерации тестовых программ на основе шаблонов конфликтных ситуаций возможных в работе конвейера. В отличие от распространенных на практике подходов (включая ручную разработку и случайную генерацию), предлагаемая методика имеет высокий уровень автоматизации и позволяет на систематичной основе проверить управляющую логику микропроцессора. В то же время, методика отличается от подходов на основе точных моделей управляющей логики тем, что она может применяться

[11] Число затронутых инструкций не включает в себя инструкции, добавленные в соответствующей ревизии.

[12] Объем измененного кода не учитывает добавление нового кода, описывающего добавленные инструкции.

на ранних этапах разработки, когда структура конвейера не является стабильной, а проект подвержен частым изменениям.

Использование потактовых моделей управляющей логики целесообразно на поздних этапах проектирования. Здесь, за счет большей информативности точных моделей, такие подходы имеют преимущества перед другими методами и потенциально позволяют обнаружить самые сложные ошибки. Кроме того, за счет использование точных моделей возможно построение более компактного набора тестов. В будущем мы планируем расширить генератор тестовых программ средствами разработки точных спецификаций конвейера и построения на их основе автоматных моделей. Таким образом, на ранних этапах разработки можно использовать генерацию на основе шаблонов, а на более поздних, когда стабилизируется управляющая логика, — тесты на основе обхода конечных автоматов.

Литература

[1] Википедия (http://en.wikipedia.org), статья *Instruction pipeline*.

[2] D. Patterson, J. Hennessy. Computer Organization and Design: The Hardware-Software Interface, 2nd edition, 1997.

[3] А.С. Камкин. Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров. Труды ИСП РАН, 2010 (этот же сборник).

[4] P. Mishra, N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems, 2008.

[5] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Technical Report 1998-29, University of California, Irvine, 1998.

[6] www.cs.cmu.edu/~modelcheck/smv.html.

[7] T.N. Dang, A. Roychoudhury, T. Mitra, P. Mishra. *Generating Test Programs to Cover Pipeline Interactions.* Design Automation Conference, 2009.

[8] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification.* Design and Test of Computers, 2004.

[9] S. Ur, Y. Yadin. Micro-Architecture Coverage Directed Generation of Test Programs. Design Automation Conference, 1999.

[10] А.С. Камкин. Генерация тестовых программ для микропроцессоров. Труды ИСП РАН, 2008.

[11] MIPS64™ Architecture For Programmers. Revision 2.0. MIPS Technologies Inc., 2003.

[12] Д.Н. Воробьев, А.С. Камкин. Генерация тестовых программ для подсистемы управления памятью микропроцессора. Труды ИСП РАН, 2009.

Автоматизация системного тестирования моделей аппаратуры на основе формальных спецификаций

М. М. Чупилко
chupilko@ispras.ru

Аннотация. В данной статье поднята проблема системного тестирования взаимосвязанных модулей аппаратуры, когда их сложность уже не позволяет применять подходы к тестированию на уровне модулей. В работе приводится краткий анализ возможностей построения тестовой системы на основе использования формальных спецификаций, а также предлагается метод верификации, являющийся расширением модульного подхода, основанного на технологии UniTESK.

1. Введение

Известно, что в проектах по разработке аппаратного обеспечения этап *верификации* является обязательным. Этот этап может быть достаточно длительным, около 70% от общего времени разработки [1]. Верификация позволяет проверить соответствие имеющейся аппаратуры (или ее модели) заявленной *спецификации*. В спецификации могут быть описаны разного рода требования: к функциональности, физическим характеристикам и др. Основным способом функциональной верификации является *тестирование* моделей аппаратуры.

В рамках данной статьи современный процесс производства компонентов аппаратуры можно условно разделить на три следующих этапа. Первый этап включает проектирование архитектуры будущего компонента, а также, возможно, создание упрощенной модели, описывающей процессы на системном уровне. Модель разрабатывается либо с использованием универсальных языков программирования (например, C/C++), либо с использованием специализированных языков системного уровня (SystemC [2], SystemVerilog [3] и др.).

Следующим этапом является создание модели на *уровне регистровых передач* (RTL[1]). На основе спецификации, иногда с привлечением общей модели системы, разрабатывается модель, содержащая детальное описание поведения компонента аппаратуры, которое может быть автоматически транслировано на *уровень вентилей*. На уровне вентилей в модели помимо функциональности начинает учитываться схемотехническая специфика интегральных схем. Имея модель на уровне вентилей, можно получать физические кристаллы, в этом заключается третий этап создания компонента, на котором функциональность создаваемого модуля не меняется, в данной статье мы его касаться не будем.

Теперь видно, что функциональные ошибки в аппаратуре могут быть выявлены уже в модели на уровне RTL, поскольку для получения физического компонента используются эквивалентные преобразования. Отсюда следует, что грамотно организованное тестирование моделей аппаратуры позволяет избежать функциональных ошибок в микросхемах. В случае сложных систем аппаратуры становится особенно актуальной тема автоматизации тестирования.

Компоненты аппаратуры различаются по своей сложности: они могут быть относительно простыми модулями, а могут представлять собой системы, в которых работает несколько функционально взаимосвязанных модулей. Для более качественного тестирования на каждом уровне «гранулированности» предлагается использовать различные подходы. Так, например, для тестирования модулей был предложен подход на основе формальных спецификаций [4]. Но для возможности его применения для более сложных компонентов требуется некоторая модификация метода.

Данная работа ориентирована на тестирование сложных моделей на уровне RTL. Статья содержит следующие разделы. В разделе «Существующие подходы к верификации систем» дается краткий обзор существующих технологий создания тестов для сложных моделей аппаратуры. Там же приводится обоснование использования формальных спецификаций. В разделе «Тестирование на модульном уровне» дается введение в существующий подход к функциональной верификации на модульном уровне, основанный на технологии UniTESK [5]. В разделе «Тестирование на системном уровне» предлагается подход, который продолжает традиции использования UniTESK, но уже применительно к многомодульным системам. В заключении приведены основные тезисы, характеризующие предлагаемый метод тестирования.

[1] RTL, Register Transfer Level, уровень регистровых передач. Для описания модели аппаратуры используются регистры, комбинационные блоки, их соединения между собой.

2. Существующие подходы к верификации систем

Подходы к верификации моделей аппаратуры можно разделить на два больших класса: формальную и имитационную верификацию. Формальные методы позволяют доказать математически, что модель работает корректно. Проверка осуществляется без учета динамики модели, с помощью анализа исходного кода. Существуют разные методы формальной верификации, однако они работают лишь для относительно простых моделей. Часто формальные методы используются для доказательства эквивалентности RTL-модели и модели на уровне вентилей. Для тестирования сложных моделей требуется слишком много ресурсов, и позволить себе такой подход могут только крупные фирмы. С увеличением вычислительной мощности компьютеров и совершенствования методик формальной верификации, возможно, они будут использоваться гораздо чаще, однако, на сегодняшний день их доля относительно мала.

Имитационная верификация подразумевает запуск модели с использованием симулятора. Для проведения верификации строится специальная тестовая система, которая генерирует стимулы для тестируемого компонента, проверяет реакции, полученные в ответ на поданные стимулы, а также оценивает полноту тестирования. Рассмотрим основные аспекты построения тестовых систем.

Тестовые последовательности могут быть описаны вручную, построены с использованием генератора случайных чисел, созданы на основе шаблонов, либо быть сгенерированы на основе обхода конечных автоматов. Ручное создание последовательностей не является технологичным, оно не применимо для тестирования сложных систем из-за больших требований к ресурсам и большой вероятности внесения ошибок в тесты: в системном тестировании необходима автоматизация создания стимулов. Одним из способов автоматизации является использование случайных стимулов: при таком подходе можно не тратить время на создание более «тонкой» методики и инструментов, но и вероятность нахождения сложных ошибок с его помощью довольно мала.

Другим часто используемым способом автоматизации является использование шаблонов. Шаблоны определяют некоторый класс эквивалентных ситуаций для проверки, создаются вручную и имеют вместо конкретных значений аргументов операций ограничения на их значения. Перед использованием шаблона в качестве тестового воздействия специальный компонент тестовой системы преобразовывает ограничения на значения параметров в систему уравнений и затем решает ее, используя техники разрешения ограничений. Если генерация случайных стимулов применяется, когда необходимо как можно раньше начать процесс тестирования, то использование техник разрешения ограничений позволяет нацелить тестирование на определенные классы функциональности. Подход к генерации стимулов на основе шаблонов используется в современных методологиях тестирования моделей

аппаратуры [6]. Необходимо учесть, что шаблоны все же создаются вручную со всеми вытекающими недостатками: ограничением на количество шаблонов, возможностью внесения в них ошибок, трудностью поддержки при изменениях в тестируемом компоненте. Поэтому провести полную верификацию компонента, а не только «крайних случаев», средствами шаблонов затруднительно.

Еще одним способом генерации тестовых воздействий является использование конечных автоматов. Конечный автомат можно представить как граф, дуги которого соответствуют подаче стимулов, а вершины представляют собой состояния тестируемой системы. Обычно для тестирования используются детерминированные автоматы, в которых для каждой последовательности входных воздействий (стимулов) существует лишь одно состояние, в которое автомат может перейти из текущего состояния. При обходе такого автомата происходит подача стимулов на тестируемую систему; процесс продолжается до покрытия всех вершин и дуг, либо до обнаружения ошибки.

Автоматы можно описывать явно с помощью графа состояний и переходов [7-9], можно описывать неявным образом, строя его на основе *формальной спецификации*, как это делается в технологии UniTESK. Явное описание автомата в реальных задачах сталкивается с трудностью поддержки его актуальности при постоянных изменениях в тестируемом проекте. В технологии UniTESK для описания каждой операции, которую реализует тестируемый компонент, задается *контракт*: *пред*- и *постусловия* операции. Контракт может накладываться не только на операцию в целом, но и на отдельные ее этапы. Неявное задание автомата осуществляется путем определения *функции вычисления текущего состояния* и описания *списка допустимых стимулов*. Построение автомата осуществляется динамически в процессе обхода. Для создания функции вычисления текущего состояния обычно используется информация о том, на каком этапе какая операция выполняется, а также информация о зависимостях между операциями. Таким образом, процедура поддержки автомата, заданного неявно с помощью формальных спецификаций, проще, чем описанного явно: при изменениях в проекте необходимо изменить спецификации, а тогда автоматически меняется и автомат.

Возвращаясь к проблеме построения тестовой системы, необходимо ответить на вопросы о *проверке правильности поведения* и *оценке полноты тестирования*. На оба вопроса позволяет ответить использование контрактных спецификаций: в первом случае это будет проверка всех необходимых постусловий на каждом этапе выполнения операций, во втором случае оценивается полнота обхода конечного автомата, заданного на основе формальных спецификаций – контрактов, и, фактически, описывающего устройство управления аппаратуры.

Альтернативными вариантами проверки правильности являются использование *эталонных моделей* и *самопроверяющихся тестов*.

Эталонные модели обычно используются для верификации в том случае, если они были разработаны на этапе архитектурного проектирования. Современным подходом к разработке эталонных моделей является использование транзакций [10] при передаче информации между модулями системы — такие модели называются *транзакционными*[2]. Отметим, что при использовании эталонных моделей есть как минимум два недостатка. Во-первых, разный уровень абстракции проверяемой и эталонной модели затрудняет проверку, а, во-вторых, для поддержки обоих моделей в актуальном состоянии при изменениях в проекте требуются значительные усилия. Конечно, формальные спецификации тоже нуждаются в поддержке, но их структура обычно проще: эталонные модели являются полноценными копиями проверяемого компонента, хоть и более абстрактными, а спецификации ограничиваются контрактами операций.

Самопроверяющиеся тесты подразумевают отсутствие какой-либо эталонной модели или спецификации, они содержат в себе проверки, которые необходимо осуществить после выполнения тестовой последовательности. Использование таких тестов вместе с автоматизированными генераторами тестовых последовательностей практически невозможно, поэтому они используются только при ручной разработке тестов, требуя значительных усилий по поддержке при изменениях в проекте.

Теперь осталось обсудить оценку полноты тестирования, которая может быть сделана двумя путями: использованием метрик на основе исходного кода модели, либо на основе спецификаций. Использование метрик покрытия кода очевидно и индикативно, но не отражает покрытие функциональности. Метрики на основе спецификации очень близки к покрытию функциональности при условии полного описания требований в спецификациях. Важным классом функциональных метрик являются так называемые *автоматные метрики*, определяемые на основе покрытия состояний и переходов конечных автоматов, построенных на основе требований. В контексте использования конечных автоматов в тестовых системах наиболее органичным видится и использование метрик на их основе для оценки полноты тестирования.

Описанные выше идеи использования формальных спецификаций и построения на их основе автоматов использовались при создании подхода к тестированию аппаратуры на модульном уровне [4], который успешно применяется для тестирования различных модулей промышленных микропроцессоров [11]. Поскольку у приведенных идей нет принципиальных

[2] Иногда говорят о модели на *уровне TLM* (*Transaction Level Modeling, моделирование на уровне транзакций*), когда имеется в виду более абстрактное представление аппаратуры по сравнению с уровнем RTL.

ограничений для использования на уровне систем (необходимо только избегать комбинаторного взрыва количества состояний конечного автомата), то их развитие для системного тестирования видится перспективным. К методу модульного тестирования и технологии UniTESK, на основе которой он разработан, обратимся в следующем разделе.

3. Тестирование на модульном уровне

Подход к модульному тестированию основан на технологии тестирования UniTESK. Как отмечалось в предыдущем разделе, при использовании данной технологии предполагается, что спецификация на тестируемый компонент записывается в виде контрактов: *пред-* и *постусловий* его операций. В случае тестирования программного обеспечения контракты накладываются на операции в целом, а в случае тестирования моделей аппаратуры каждая операция разбивается на микрооперации, для которых определяются индивидуальные контракты. На Рис. 1 изображено графическое представление спецификации на операцию: под *PRE* и *POST* понимаются пред- и постусловия микроопераций. Микрооперации могут быть записаны не только последовательно, но и с использованием ветвления потока управления операции [12].

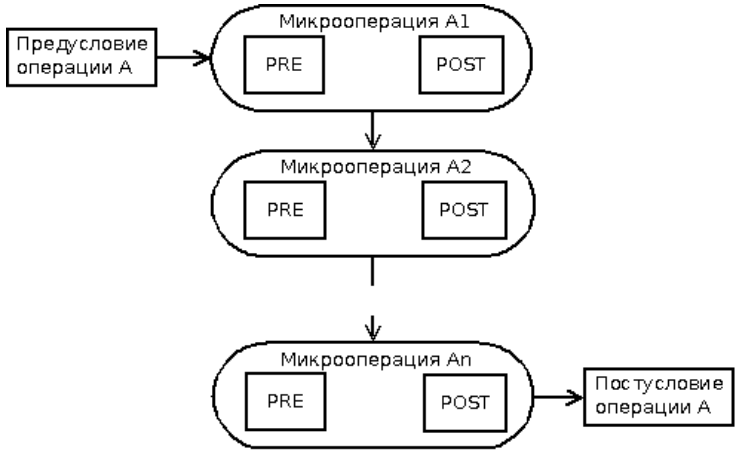


Рис. 1. Графическое изображение контракта на операцию.

Вычисление множества микроопераций, выполняемых параллельно на каждом такте (см. Рис. 2 и Рис. 3), используется как основа для задания текущего состояния автомата. Если предусловие операции выполнено, то она запускается, если выполнено предусловие микрооперации, то на следующем такте происходит проверка постусловия, а в состояние автомата добавляется информация о том, что данная микрооперация пройдена (например, *“A1*

finished"). Если микрооперация не выполнена, то в состоянии будет это отмечено (например, *"A3 delayed"*); если задержка происходит более одного такта, то состояние может отражать количество тактов задержки. После проверки постусловия осуществляется переход к следующей микрооперации рассматриваемой операции.

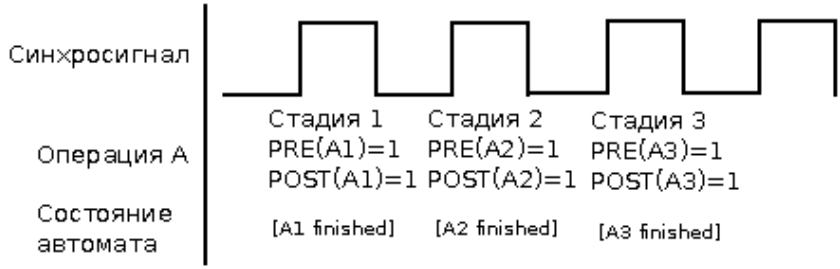


Рис. 2. Последовательное выполнение микроопераций одной операции.

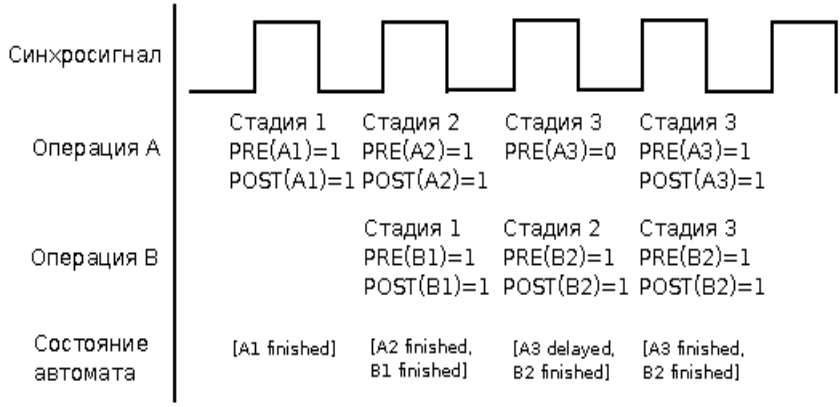


Рис. 3. Выполнение микроопераций на конвейере.

Отметим, что в случае асинхронных модулей принцип использования микроопераций остается, но меняется сигнал, к которому они привязаны: это уже не синхроимпульс *"CLK"*, а либо внешние сигналы, отвечающие за асинхронное взаимодействие, либо истечение заданного промежутка времени.

Данный метод уже несколько лет успешно применяется для тестирования моделей аппаратуры на модульном уровне. Он обеспечивает хорошее покрытие функциональности модулей с конвейерной архитектурой, находит в них ошибки после продолжительных испытаний самими разработчиками. Однако, у описанного подхода есть ограничения на сложность тестируемых модулей, поскольку их потактовая спецификация может привести к слишком большому числу состояний автомата. В следующем разделе будет предложена

модификация метода для возможности тестирования многомодульных систем, основанная на настройке точности тестирования.

4. Тестирование на системном уровне

Под *системой* мы будет понимать набор функционально зависимых модулей, имеющий единый внешний интерфейс, который служит для взаимодействия системы со своим окружением. Передаваемые через интерфейс операции для исполнения данной системой могут быть рассмотрены как на уровне всей системы, т.е. описаны посредством операций приема-передачи на уровне интерфейса системы, так и на уровне отдельных модулей, входящих в систему. Чем больше информации мы учитываем при спецификации той или иной операции, тем более тщательное тестирование становится возможным провести, тем большее количество различных ситуаций можно создать. Будем называть описание выполняемых операций, при котором не учитываются внутрисистемные особенности выполнения операции, *описанием на уровне системы*. Описание операций, в котором отведено место более детальному анализу межмодульного поведения операции, будем называть *описанием на уровне модулей*. В некоторых случаях необходимо учитывать также особенности поведения операции внутри модулей, в этом случае будем говорить о *субмодульном уровне*.

В процессе выполнения любой операции, на каком бы уровне она ни была рассмотрена, имеются этапы обмена данными между некоторыми сущностями (блоками, модулями, подсистемами), имеющими смысл на данном уровне. При рассмотрении уровня системы обмен информацией может быть только с окружением системы. При переходе на модульный уровень появляется информация об этапах обмена данными между модулями. Каждый этап такого взаимодействия легко фиксируется и лежит в основе спецификации видимого поведения операции. Будем называть этапы выполнения операции, в которых происходит какое-либо интерфейсное взаимодействие, *точками синхронизации*.

Иногда может возникнуть необходимость рассмотреть какие-либо дополнительные этапы выполнения операции, наличие которых на данном уровне абстракции (например, при рассмотрении только системного уровня) не следует из имеющегося описания интерфейсных взаимодействий. Даже на модульном уровне может возникнуть необходимость учесть некоторые особенности поведения тестируемого компонента, которые реализованы на уровне блоков. Тогда верификатору совместно с инженером-разработчиком необходимо расширить список точек синхронизации *дополнительными точками*. Описать такие точки можно либо с помощью количества тактов, которые должны пройти с момента последней точки синхронизации данной операции, либо с помощью использования интерфейсных сигналов, скрытых на данном уровне рассмотрения системы (поскольку сейчас идет речь о тестировании моделей, данный прием всегда возможен).

Основная причина, ограничивающая применение подхода к модульному тестированию для верификации многомодульных систем, заключается в комбинаторном взрыве количества состояний в конечном автомате вследствие использования слишком детальных спецификаций. Для перехода к системному тестированию необходимо уменьшить детализацию. В качестве средства настройки точности тестирования предлагается использовать точки синхронизации. Далее детально рассматриваются вопросы задания конечного автомата на основе точек синхронизации.

4.1. Построение конечного автомата для систем аппаратуры

В процессе модульного тестирования состояние конечного автомата, описывающего тестируемый компонент, обновлялось на каждом такте (см. Рис. 2 и Рис. 3). При этом учитывалась информация о выполненных микрооперациях, задержанных микрооперациях, а также о времени их задержки, и, кроме того, могла учитываться дополнительная информация, например, о зависимостях между операциями. В результате число возможных состояний автомата при тестировании сложных систем получается слишком большим.

Для уменьшения числа состояний предлагается перейти от потактового вычисления текущего состояния на основе микроопераций, к построению состояния на основе точек синхронизации. В таком случае контракты (предусловие соответствует предикату достижимости точки, постусловие – проверке результатов интерфейсного взаимодействия) накладываются не на микрооперации, а на точки синхронизации. Соответственно, формальная спецификация теперь включает в себя только ключевые аспекты важные для межмодульного взаимодействия. Автомат, который учитывает только точки синхронизации, содержит существенно меньше состояний, поскольку он не берет в расчет задержки выполнения этапов операций и сведения о субмодульном выполнении операций.

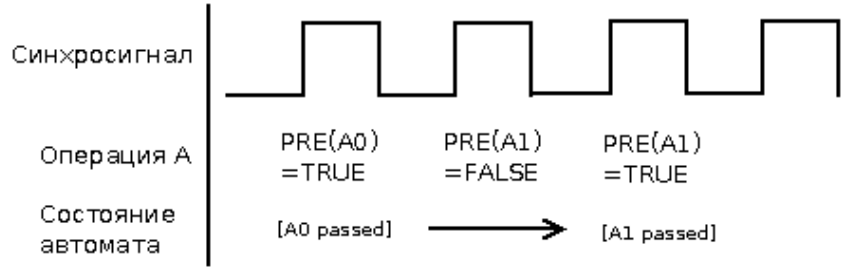


Рис. 4. Пример перехода в автомате при выполнении одной операции.

На Рис. 4 проиллюстрировано, как определяется момент времени, соответствующий переходу в очередное состояние автомата. Для наглядности рассмотрен случай, когда выполняется только одна операция. В данном

примере точка синхронизации *A0* достигается сразу, точка синхронизации *A1* задерживается на 1 такт, в котором функция вычисления текущего состояния модуля не вызывается. Следовательно, в тестовую систему вводится дополнительный компонент, который работает на каждом такте и проверяет, достигнута или нет очередная точка синхронизации. Достижение точки соответствует «окончанию» перехода в следующее состояние конечного автомата, описывающего данную операцию. По достижении точки синхронизации можно подавать новые стимулы для продолжения обхода автомата.

Если выполняется одновременно несколько операций, что обычно и наблюдается в системах аппаратуры, то появляется понятие *глобального автомата* (в дальнейшем слово *глобальный* будет опускаться, если это не будет вносить путаницы), который в общем случае содержит композицию автоматов выполняющихся операций. При построении композиции автоматов достигнутые на данном такте точки синхронизации будут помечаться как *passed*. Например, текущее состояние глобального автомата *[A0, B0 passed]* говорит о том, что сейчас работают 2 операции *A* и *B*, на данном такте достигнута точка синхронизации *B0*, а ближайшая точка синхронизации операции *A* – это *A0*.

В случае если тестируемая система допускает параллельную обработку нескольких операций, построить детерминированный автомат только на основе только точек синхронизации без учета дополнительной информации о зависимостях между операциями не всегда возможно. На Рис. 5 приведен пример, в котором операция *A* в точке синхронизации *A0* и *D* в точке *D0* влияют на время появления точки *B2*, но при этом *C1* сохраняет свое положение. В результате из состояния *[B1 passed, C1]* мы попадаем либо в *[B2 passed, C1 passed]*, либо в *[B2, C1 passed]* недетерминированным образом. В данном случае помогло бы введение дополнительной точки *D_1*, которая бы символизировала освобождение некоторого разделяемого ресурса, который не поделили операции *B* и *D*.

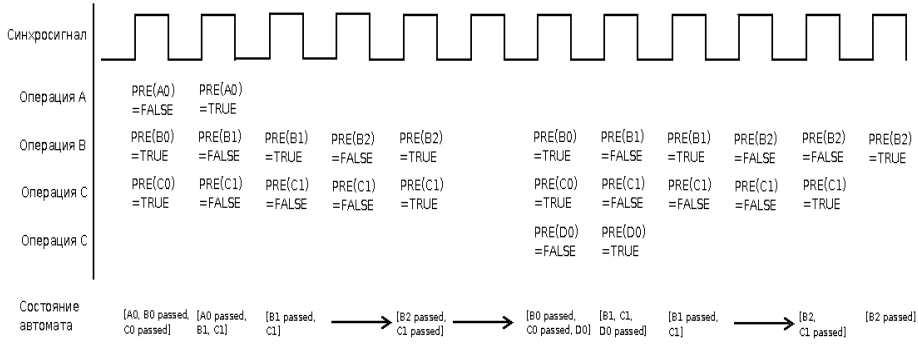


Рис. 5. Недетерминизм автомата, основанного на точках синхронизации.

Следовательно, описание глобального детерминированного автомата в случае параллельных операций должно строиться на основе точек синхронизации и, при необходимости, дополнительных точек. Оставим гипотезу о том, что в общем случае точек синхронизации и дополнительных точек достаточно для описания детерминированного автомата, без доказательства.

В примере, приведенном на рис. 5, предполагается, что каждая новая операция может быть запущена в точках синхронизации предыдущих операций или при пустом состоянии конвейера. Такой подход возможен, однако, во-первых, необходимо достаточно тонко обходить возможный недетерминизм автомата, а, во-вторых, в результате тестирования получается одна большая трасса теста, которая может быть сложна для нахождения короткого пути для повторения однажды возникшей ошибки. Поэтому в ряде случаев предлагается использовать немного модифицированные правила запуска стимулов и учета точек синхронизации работающих операций.

Так, предлагается «фиксировать» одну из выполняемых тестируемым компонентом операций, исполнение которой разрешено начинать только при пустом состоянии конвейера. Все остальные операции можно запускать только в процессе выполнения фиксированной, но как только фиксированная операция завершается, допускается только завершение всех параллельных операций и приход к пустому состоянию конвейера (Рис. 6). Также предлагается не учитывать в состоянии автомата точки синхронизации всех параллельных (по отношению к фиксированной) операций; тогда производится учет только списка операций, которые были поданы в точках синхронизации фиксированной.

Рис. 6. Использование коротких тестов.

Таким образом, тестирование разбивается на набор коротких тестов, с помощью которых в точках синхронизации фиксированной операции тестируются старты всех совокупностей допустимых стимулов. Основным преимуществом этого способа является достаточно легкое воспроизведение найденной ошибки.

В случае учета точек синхронизации операций, параллельных фиксируемой, появляются те же проблемы, связанные с недетерминизмом автомата, как и в случае «длинного» теста. Но при этом же получается набор коротких тестов, полная совокупность которых будет эквивалентна тестовой последовательности, полученной путем обхода автомата в «длинном» тесте.

Подводя итоги данного подраздела, можно сказать, что детерминированный конечный автомат для тестирования систем должен строиться не только на основе точек синхронизации, но и путем учета дополнительных точек. В состоянии автомата учитываются как достигнутые на данном такте точки, так и те, которые еще предстоит достичь во всех остальных работающих на данном такте операциях. Построенный таким образом автомат может быть единым для всего теста, а может разбиваться на набор маленьких автоматов для получения коротких тестов. В следующем подразделе попробуем расширить возможности подачи стимулов в промежутках между точками синхронизации.

4.2. Расширение возможностей подачи стимулов

В приведенном выше способе создания конечного автомата для системного тестирования предполагалось, что подача стимулов осуществляется только при достижении некоторой точки синхронизации (или дополнительной точки). В промежутках же между точками стимулы не подаются.

В некотором смысле моменты времени между двумя соседними состояниями глобального автомата можно считать *эквивалентными* и осуществлять подачу тестовых воздействий на них, а не только при достижении точек синхронизации. Такт для подачи в этом случае выбирается случайным образом. Однако при подаче стимулов в таких промежутках можно нарушить требование детерминизма автомата. Приведем пример. Пусть при выполнении операции *A* возникло два последовательных промежутка: *A0* (*x1* тактов) и *A1*. Промежуток *B0* (*x1/2* тактов) операции *B* стартует в одном случае в начале *A0*, а в другом – в конце. Тогда при одинаковом исходном состоянии *[A0 passed]* следующее состоянии автомата может быть либо *[A1 passed, B0]*, либо *[A1 passed, B0 passed]*. Очевидно, что в данном случае необходимо использовать дополнительную информацию: например, на каком конкретно такте из промежутка была запущена новая операция. Но учет такой информации может привести к росту количества состояний. Также необходимо учитывать, что точки синхронизации выбираются не случайным образом, поэтому их покрытие должно осуществляться в первую очередь. Следовательно, возможностями подачи стимула в эквивалентных промежутках следует пользоваться с осторожностью.

При повышении уровня абстракции неизбежно должны потеряться некоторые случаи, которые можно было бы охватить на модульном уровне. Рассмотрим такие случаи в следующем подразделе.

4.3. Сравнение модульного и системного подходов

Тестирование на модульном уровне в основном отличается от тестирования на системном уровне детализацией проверок и детализацией состояний. В случае модульного подхода, например, учитывается такая характеристика, как количество тактов, прошедших после блокировки микрооперации, т.е. разным

задержкам в автомате будут соответствовать разные состояния, соответственно в каждом из них (на каждом такте задержки) будет осуществляться попытка подать очередной стимул. Понятно, что некоторые из тактов ожидания можно считать эквивалентными, но если разработчик тестов не знает особенностей реализации блокировок в тестируемом модуле, такой избыточный подход оказывается предпочтительным, поскольку позволяет покрыть все ситуации.

В случае системного подхода такты ожидания не учитываются, а стимул подается либо в точке синхронизации, либо случайным образом во временном промежутке между такими точками. Поэтому, если между точками синхронизации происходила какая-либо важная скрытая активность, она не будет гарантированно проверена в сочетаниях с другими операциями (см. рис. 7). Таким образом, для повышения качества тестирования верификатору необходимо выяснять все интересные точки, в которых происходят какие-либо важные скрытые процессы в модулях в составе системы. Этими интересными точками необходимо расширять списки точек синхронизации и дополнительных точек в используемых тестах и получать возможность изменения детализации тестирования, сохраняя баланс между качеством и временем тестирования.

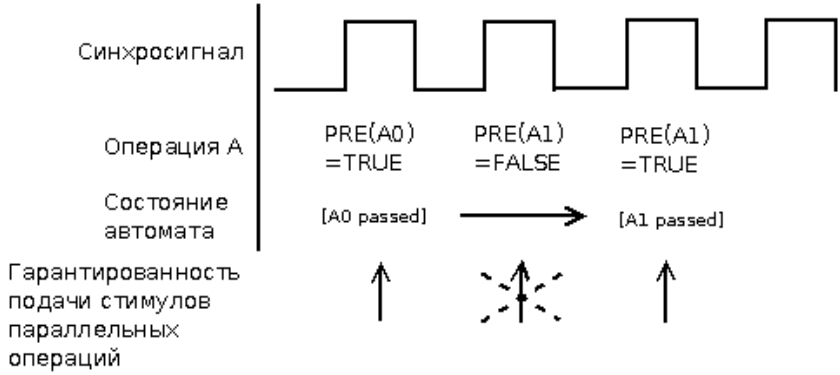


Рис. 7. Меньшая детализация тестирования в случае системного подхода.

5. Заключение

В данной работе были представлены основы метода автоматизированного системного тестирования моделей аппаратуры, использующего формальные спецификации так называемых *точек синхронизации* операций. Апробация данного метода пока не осуществлялась, но, учитывая, что в его основе лежит метод тестирования на модульном уровне, переработанный в соответствии с реалиями систем, автору видится перспективность его использования.

Инструментальная поддержка подхода может быть осуществлена с использованием инструментов технологии UniTESK.

Литература

[1] J. Bergeron. Writing Testbenches Using SystemVerilog. Springer Media, 2006.
[2] IEEE Standard SystemC Language Reference Manual. IEEE Std 1666-2005
[3] IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2009.
[4] А.С. Камкин, Метод автоматизации имитационного тестирования микропроцессоров с конвейерной архитектурой на основе формальных спецификаций, диссертация на соискание уч.ст. к.ф.-м.н., Москва, 2009.
[5] http://www.unitesk.ru
[6] S. Iman. Step-by-step Functional Verification with SystemVerilog and OVM. Hansen Brown Publishing Company, 2008.
[7] S. Ur, Y. Yadin. Micro architecture coverage direction generation of test programs. Proceedings of DAC, 1999.
[8] P. Mishra, N. Dutt. Functional coverage driven test generation for validation of pipelined processors. Proceedings of DATE, 2005.
[9] R. Ho, C. Yang, M. Horowitz, D. Dill. Architecture validation for processors. Proceedings of ISCA, 1995.
[10] TLM-2.0 Reference Manual, http://www.systemc.org/downloads/standards/
[11] M. Chupilko, A. Kamkin, D. Vorobyev. Methodology and Experience of Simulation-Based Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications. Proceedings of SYRCOSE, 2008.
[12] M. Chupilko, A. Kamkin. Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs. Proceedings of NorChip, 2009.

Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров

А.С. Камкин
kamkin@ispras.ru

Аннотация. В работе рассматриваются вопросы автоматизированного построения тестовых программ, предназначенных для функциональной верификации модулей обработки переходов микропроцессоров. Формулируются задачи, возникающие при создании таких программ, и предлагаются техники их автоматизированного решения. Статья фокусируется на общих вопросах тестирования механизмов обработки переходов и не затрагивает проблемы специфичные для конкретных микропроцессорных архитектур. Предложенные техники можно использовать в промышленных генераторах тестовых программ.

1. Введение

Во всех универсальных, вычислительно полных по Тьюрингу, компьютерах, начиная с электронного калькулятора ENIAC (1946 г.), в том или ином виде присутствуют *инструкции перехода* — инструкции, позволяющие управлять потоком выполнения программы: осуществлять переходы на заданные инструкции, организовывать ветвления и циклы. Устройство компьютера, отвечающее за обработку таких инструкций, обычно называют *модулем обработки переходов*[1].

С точки зрения программиста, возможности модуля обработки переходов типичного микропроцессора невелики и сводятся к условным и безусловным переходам, вызовам подпрограмм и возвратам из них. В то же время внутренняя, скрытая от посторонних глаз, логика работы модуля может быть очень сложной и запутанной за счет использования механизмов повышения производительности: предсказания переходов, суперскалярного выполнения и многих других.

[1] Не следует понимать этот термин буквально — физически отдельного модуля обработки переходов может и не существовать. В этом случае корректнее говорить о логике обработки переходов.

Не секрет, что сложность является основным источником ошибок, допускаемых при проектировании микропроцессоров. В случае же если сложность является скрытой, обнаружить ошибки бывает достаточно трудно из-за отсутствия средств, позволяющих явно управлять поведением тестируемой системы или хотя бы наблюдать его. Таким образом, несмотря на кажущуюся простоту, модули обработки переходов микропроцессоров могут содержать в себе нетривиальные, трудно обнаруживаемые ошибки.

От правильности обработки переходов зависит как производительность микропроцессора, так и его корректность — систематическая ошибка в предсказании переходов может снизить производительность компьютера, а, например, неправильное вычисление адреса перехода или некорректное восстановление состояния после спекулятивного выполнения неверно предсказанных инструкций являются серьезными функциональными ошибками.

Поскольку модуль обработки переходов тесно взаимодействует с другими компонентами микропроцессора, имеет смысл проверять его работу системно, в связке с окружением. Наиболее распространенным способом системной верификации микропроцессоров и других программируемых устройств является тестирование моделей уровня регистровых передач (RTL-моделей) с помощью тестовых программ. Предметом статьи является автоматизация построения таких программ. В работе рассматриваются вопросы генерации *структур переходов* (по сути, графов потока управления), *трасс выполнения* (маршрутов в графе потока управления) и *управляющего кода* (вспомогательных инструкций, обеспечивающих проход по заданному маршруту). Предлагаемые подходы основаны на комбинаторных техниках и могут использоваться для разных микропроцессорных архитектур.

Статья организована следующим образом. В разделе 2 кратко описываются возможности модулей обработки переходов микропроцессоров, и рассматриваются проблемы, возникающие при их верификации. Раздел 3 определяет основные понятия используемого подхода, а его детальное описание дается в разделе 4. В разделе 5 описывается инструментальная поддержка, и приводятся примеры тестовых программ. Раздел 6 является заключением.

2. Обработка переходов в микропроцессорах

Рассмотрим кратко, что представляют собой модули обработки переходов современных микропроцессоров, и какие проблемы возникают при их верификации.

2.1. Модули обработки переходов

Модулем обработки переходов (*branch unit*) называется подсистема микропроцессора, отвечающая за обработку *инструкций переходов* —

инструкций, осуществляющих изменение потока управления[2]. Простейшими инструкциями данного типа являются *инструкции безусловного перехода*. К более сложным относятся *инструкции условного перехода*, которые изменяют поток управления только при выполнении определенных условий на значения операндов (обычно условия переходов выражаются через сравнения вида =, ≠, <, >, ≥, ≤). Во многих микропроцессорных архитектурах также предусмотрены *инструкции вызова подпрограмм и возврата из них*. При вызове подпрограммы перед передачей управления на нее происходит сохранение адреса следующей инструкции (адреса возврата) в специальном регистре или стеке. При завершении подпрограммы этот адрес считывается и по нему делается передача управления.

Современные микропроцессоры имеют *конвейерную архитектуру*, что означает, что они параллельно обрабатывают сразу несколько инструкций (каждую на своей стадии) [1]. В зависимости от организации микропроцессора выполнение перехода может приводить к *приостановке конвейера* (когда после выборки инструкции перехода загрузка конвейера приостанавливается до тех пор, пока не будет принято решение о переходе) или *сбросу конвейера* (когда все инструкции, загруженные на конвейер после инструкции перехода, сбрасываются, если принимается решение о выполнении этого перехода).

В некоторых микропроцессорах для борьбы со сбросами конвейера имеются так называемые *слоты задержки переходов* (*branch delay slots*) [1,2]. Это означает, что перед обработкой инструкции перехода микропроцессор успевает выполнить (точнее, загрузить на конвейер) одну или несколько инструкций, следующих за ней, (для каждой архитектуры это число фиксировано — в архитектурах MIPS или SPARC имеется один слот задержки, в SHARC DSP их два [2]). Инструкции, расположенные в слотах задержки перехода, выполняются независимо от того, выполнен переход или нет. Заметим, что в слот задержки нельзя помещать инструкцию перехода.

Для постоянной загрузки конвейера (минимизации числа сбросов) также применяется прогнозирование переходов. Стратегии прогнозирования могут быть *статическими* и *динамическими*, кроме того, они могут различаться для разных типов инструкций перехода [3]. Статические методы предсказания переходов являются наиболее простыми. Их суть состоит в том, что все переходы одного типа либо всегда прогнозируются выполненными, либо всегда прогнозируются невыполненными. Например, в ранних микропроцессорах архитектуры MIPS и SPARC условные переходы «назад» предсказывались выполненными, а условные переходы «вперед» — нет. Статическое предсказание используется и в современных микропроцессорах, но только в качестве «подстраховки», когда невозможно применение динамических методов [3].

---

[2] Под изменением потока управления понимается изменение *счетчика адреса* — регистра микропроцессора, содержащего адрес текущей инструкции.

Динамические методы подразумевают анализ истории переходов. Примером метода такого типа является двухуровневый алгоритм Йеха (Yeh), используемый в микропроцессорах Intel архитектуры P6 [4]. Первый уровень алгоритма представлен таблицей, хранящей для «каждой»[3] инструкции перехода историю четырех последних переходов. История хранится в виде четырехбитного паттерна и обновляется при каждом выполнении соответствующей инструкции (осуществляется сдвиг паттерна на один разряд влево, после чего в младший разряд записывается 1 или 0 в зависимости от того, выполнен переход или нет). Второй уровень состоит из шестнадцати двухбитных счетчиков с насыщением. Паттерн первого уровня используется в качестве индекса в таблице счетчиков. При осуществлении перехода счетчик инкрементируется, в противном случае декрементируется. Переход прогнозируется тогда и только тогда, когда счетчик находится в состоянии 2 или 3.

2.2. Особенности верификации модулей обработки переходов

Модули обработки переходов современных микропроцессоров имеют сложную организацию, и при их верификации, особенно если она осуществляется в связке со смежными компонентами, необходимо учитывать большое число факторов. Некоторые из них, типичные для многих микропроцессорных архитектур, перечислены ниже:

1. *тип инструкции перехода* (условный или безусловный переход, вызов подпрограммы или возврат из нее) — разные типы инструкций обрабатываются по-разному, поскольку у них просто напросто разная семантика;
2. *направление перехода* (вперед или назад) — направление перехода обычно влияет на прогноз, даваемый статическими методами;
3. *способ передачи адреса перехода* (непосредственно или через регистр) — если адрес перехода передается непосредственно, направление перехода можно определить уже на этапе декодирования инструкции;
4. *структура переходов* (граф потока управления) — некоторые архитектуры оптимизируют выполнение кода с определенной структурой переходов, например, «коротких» циклов;
5. *трасса выполнения* (история переходов) — история переходов влияет на результаты динамического прогнозирования, которые, в свою очередь, влияют на спекулятивную загрузку инструкций на конвейер;
6. *типы инструкций в слотах задержки переходов* (если архитектура предусматривает наличие слотов задержки) — в слотах задержки

[3] Понятно, что размер таблицы конечен. Идентификация инструкций осуществляется по младшим битам адреса.

могут находиться разные инструкции, в том числе, вызывающие исключения;

7. *типы спекулятивно выполняемых инструкций* (типы инструкций, следующих за переходом или находящихся по адресу перехода) — в микропроцессорах возможны ограничения на инструкции, которые можно выполнить спекулятивно;
8. *вызовы исключений* (исключения в слотах задержки и других инструкциях) — обработка исключений, вызванных инструкциями, находящихся в слотах задержки, обычно происходит особым образом, а инструкции, вызывающие исключения, не всегда можно выполнить спекулятивно;
9. *наличие зависимостей между инструкциями* (по регистрам и через память) — зависимости, например, между инструкциями, расположенными после перехода, и инструкциями до перехода (или инструкциями в слотах задержки) может приводить к блокировкам конвейера при спекулятивном выполнении;
10. *положение инструкции перехода в памяти* (относительно буфера предвыборки, кэш-памяти, страницы виртуальной памяти) — в зависимости от положения инструкции в памяти обработка перехода по ряду причин может происходить по-разному.

Как видно, факторов, влияющих на обработку переходов, достаточно много и ручная разработка полного комплекта тестовых программ вряд ли возможна. Огромное пространство состояний делают несостоятельной и чисто случайную генерацию. Целью настоящей работы является создание методологической поддержки для автоматизированной генерации тестовых программ. Поскольку задач, возникающих при тестировании модулей обработки переходов, достаточно много, мы решили ограничиться некоторыми из них, которые не связаны с техническими деталями работы таких устройств. Рассматриваемые вопросы затрагивают (в той или иной степени) только первые семь факторов из числа перечисленных (типы инструкций, структуры переходов и трассы выполнения).

3. Основные понятия предлагаемого подхода

В предлагаемом подходе построение тестовых программ осуществляется автоматически или полуавтоматически на основе формальной спецификации системы команд и описания тестовых ситуаций и зависимостей для инструкций микропроцессора. Построение тестовых программ осуществляется путем создания различных цепочек инструкций (последовательностей из инструкций разных типов с разной структурой переходов) и комбинирования для них тестовых ситуаций (событий, связанных с выполнимостью условий переходов, исключениями и т.п.) и зависимостей (взаимосвязей между инструкциями через регистры или

память) [5]. Рассматриваемый подход можно использовать и при создании тестовых программ на основе шаблонов, когда инженер-верификатор описывает набор ограничений (тестовый шаблон), а программа генерируется автоматически на основе заданных ограничений.

3.1. Структура тестовой программы

Тестовая программа представляет собой последовательность *тестовых вариантов*. Каждый тестовый вариант содержит *тестовое воздействие* — специально подготовленную цепочку инструкций, предназначенную для создания определенной ситуации в работе микропроцессора. Тестовое воздействие предваряется *инициализирующими инструкциями*, а после него может располагаться *тестовый оракул* — набор инструкций, проверяющих корректность состояния микропроцессора после выполнения тестового воздействия[4].

Таким образом, структуру тестовой программы можно описать с помощью формулы $Test = \{\langle Pre_i, Action_i, Post_i\rangle\}_{i=0,n-1}$, где Pre_i — это инициализирующие инструкции, $Action_i$ — тестовое воздействие, $Post_i$ — тестовый оракул. В простейшем случае каждая тестовая программа состоит из одного тестового варианта, то есть $Test = \langle Pre, Action, Post\rangle$.

Ниже в качестве примера приведен фрагмент тестовой программы в системе команд MIPS [6], содержащий один тестовый вариант.

```
        /////////// Инициализирующие инструкции ///////////

        // Инициализация регистров инструкции 1:
        lui v0, 0xdead
        ori v0, v0, 0xbeaf

        lui a0, 0xdead
        ori a0, a0, 0xbeaf

        // Инициализация регистров инструкции 2:
        ori a1, zero, 0x0

        j test_action_begin
        nop // Слот задержки

        ////////////// Тестовое воздействие //////////////

backward_jump:
        j test_action_end
```

[4] Основным способом проверки правильности работы модуля обработки переходов является сравнение трасс, полученных при выполнении RTL-модели микропроцессора и его эталонного симулятора, а не встроенные в программы тестовые оракулы.

```
        addi v1, v1, 1              // Ситуация: Overflow=false
test_action_begin:
        beq  v0, a0, backward_jump // Ситуация: Condition=true
        addi v1, a1, 2009          // Ситуация: Overflow=false
test_action_end:

        ////////////////// Тестовый оракул /////////////////

        // Запись эталонного значения
        ori t1, zero, 2010

        // Ошибочное завершение при несоответствии результата
        bne v1, t1, test_fails
        nop // Слот задержки
```

Тестовое воздействие состоит из четырех инструкций: инструкции безусловного перехода `j`; инструкции `addi`, складывающей содержимое второго регистра-операнда с 16-битным непосредственным значением и сохраняющей результат в первом регистре; инструкции `beq`, осуществляющей переход в случае совпадения значений регистров; и еще одной инструкции `addi`. Точка входа в тестовое воздействие обозначена меткой `test_action_begin`. Напомним, что в архитектуре MIPS у инструкций перехода имеется один слот задержки, поэтому в данном примере при выполнении переходов (инструкций `j` и `beq`) также выполняются инструкции `addi`. Тестовый оракул сравнивает выходной регистр инструкций `addi` (он одинаковый у обеих инструкций) с эталонным значением.

Несколько слов о вспомогательных инструкциях, используемых в примере. Инструкция `lui` загружает 16-битное непосредственное значение в разряды [31:16] регистра-результата (биты [63:32] получаются путем знакового расширения). Инструкция `ori` осуществляет побитовое ИЛИ значения второго регистра с 16-битным значением, заданным в третьем операнде (результат записывается в первый регистр). Инструкция `bne` выполняет условный переход в случае неравенства значений регистров.

3.2. Тестовый шаблон

Важным понятием подхода, которое следует рассмотреть подробнее, является понятие *тестового шаблона*. Тестовым шаблоном называется абстрактная форма представления тестового воздействия или тестовой программы в целом, в котором вместо конкретных значений операндов инструкций указываются ограничения (тестовые ситуации и зависимости), которым они должны удовлетворять. По сути, каждый тестовый шаблон (если он не является противоречивым) задает некоторую цель тестирования. Разные тестовые программы, соответствующие одному и тому же тестовому шаблону, эквиваленты в том смысле, что каждая из них достигает эту цель.

Ниже приведен тестовый шаблон из рассмотренного выше примера. Шаблон состоит из инструкций `j`, `addi`, `beq` и еще одной инструкции `addi`. Для обеих инструкций `addi` задана одинаковая ситуация *Overflow=false* (отсутствие переполнения), а для инструкции `beq` ситуацией является *Condition=true* (выполнимость условия перехода). Кроме того, между двумя инструкциями `addi` имеется зависимость (первый и второй регистры первой инструкции должны совпадать с первым регистром второй инструкции).

```
backward_jump:
      j test_action_end
      addi R, R, ?              @ Overflow=false
test_action_begin:
      beq  ?, ?, backward_jump @ Condition=true
      addi R, ?, ?              @ Overflow=false
test_action_end:
```

Выполнение описываемого этим шаблоном тестового воздействия начинается с инструкции `beq`, расположенной по метке `test_action_begin`. Условие перехода выполнено (*Condition=true*), поэтому осуществляется переход на метку `backward_jump`. Выполняемая в слоте задержки инструкция `addi` не вызывает переполнения (*Overflow=false*). Далее осуществляется безусловный переход по метке `test_action_end` вместе с выполнением еще одной инструкции `addi`, которая также не вызывает переполнения (*Overflow=false*).

3.3. Структура переходов

Важным атрибутом тестового шаблона является *структура переходов*. В широком смысле, структурой переходов называется граф потока управления, задаваемый тестовым шаблоном. В более практическом понимании, структурой переходов является тестовый шаблон, который не содержит информации о тестовых ситуациях и зависимостях, но в котором все переходы снабжены метками. Ниже приведена структура переходов для рассматриваемого примера и соответствующий ей граф потока управления.

```
backward_jump:
    j test_action_end
    addi ?, ?, ?
test_action_begin:
    beq  ?, ?, backward_jump
    addi ?, ?, ?
test_action_end:
```

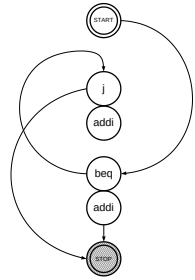


Рис. 1. Структура переходов и соответствующий граф потока управления.

Более подробно понятие структуры переходов рассматривается в разделе «*Построение структур переходов*».

3.4. Трасса выполнения

Помимо структуры переходов тестовый шаблон определяет *трассу выполнения*, задающую порядок обработки инструкций тестового воздействия (маршрут в графе потока управления). Для этого каждая инструкция условного перехода, входящая в тестовый шаблон, помечается последовательностью из значений истинности условий перехода (первое значение соответствует первому выполнению инструкции, второе — второму и т.д.; если инструкция перехода является недостижимой, она помечается пустой последовательностью). В рассматриваемом примере трасса выполнения полностью определяется истинностью условия перехода инструкции `beq` (*Condition=true*).

```
backward_jump:
    j test_action_end
    addi ?, ?, ?
test_action_begin:
    beq  ?, ?, backward_jump @ Condition=true
    addi ?, ?, ?
test_action_end:
```

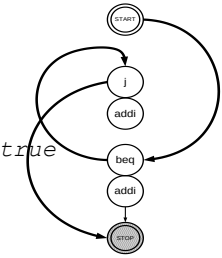


Рис. 2. Трасса выполнения и соответствующий маршрут в графе потока управления.

Более подробно трассы выполнения и метод их генерации рассматриваются в разделе «*Построение трасс выполнения*».

3.5. Управляющий код

Для того чтобы обеспечить заданную трассу выполнения, в тестовый шаблон встраивается *управляющий код*. Для рассматриваемого примера это можно и не делать, поскольку единственная инструкция условного перехода, содержащаяся в тестовом шаблоне, выполняется ровно один раз, следовательно, всю необходимую подготовку можно выполнить в инициализирующих инструкциях до выполнения тестового воздействия. В общем случае инструкции условного перехода могут выполняться по нескольку раз, причем значения истинности условий при этом могут меняться.

Рассмотрим структуру переходов, отличающуюся от рассмотренной ранее тем, что в ней отсутствует инструкция `j`, которая располагалась по метке `backward_jump`. Для этой структуры переходов рассмотрим трассу

выполнения, соответствующую последовательности {*true, false*} значений истинности условий инструкции `beq`.

```
backward_jump:
    addi ?, ?, ?
test_action_begin:
    beq  ?, ?, backward_jump @ Condition={true, false}
    addi ?, ?, ?
test_action_end:
```

Обозначим регистры инструкции `beq` как `R1` и `R2`. Для того чтобы обеспечить истинность условия при первом выполнении `beq`, инициализирующие инструкции заносят в `R1` и `R2` одинаковые значения. Еще раз подчеркнем, что инициализация не является частью управляющего кода, поскольку она выполняется вне тестового шаблона. Управляющий код должен изменить значение истинности условия с *true* на *false* после первого выполнения инструкции. Это можно сделать следующим образом.

```
backward_jump:
    addi R1, R1, 1 // управляющий код
    addi ?,  ?,  ?
test_action_begin:
    beq  R1, R2, backward_jump @ Condition={true, false}
    addi ?,  ?,  ?
test_action_end:
```

Заметим, что в данном примере изменение значения регистра `R1` (или `R2`) можно осуществить и в одной из инструкций `addi`, находящихся в слотах задержки переходов. Более подробно вопросы генерации управляющего кода рассматриваются в разделе «*Построение управляющего кода*».

4. Предлагаемый подход

В основе предлагаемого подхода к автоматизации генерации тестовых программ для модулей обработки переходов микропроцессоров лежат техники построения структур переходов, трасс выполнения и управляющего кода. Входными параметрами генератора являются классы эквивалентности инструкций переходов, *базовых блоков* (цепочек инструкций, не содержащих переходов) и инструкций в слотах задержки, а также ограничения на число переходов в тестовых шаблонах и на длину трасс выполнения. При описании предлагаемых техник будем, не ограничивая общности рассуждений, считать, что в микропроцессоре имеется один слот задержки перехода.

4.1. Построение структур переходов

Пусть инструкции переходов разбиты на классы эквивалентности, то есть задано фактор-множество $\Sigma_{jump} = \Sigma_{if} \cup \Sigma_{goto}$, где Σ_{if} — фактор-множество

условных переходов, а Σ_{goto} — безусловных[5]. Пусть, кроме того, заданы классы эквивалентности слотов задержки Σ_{delay} и базовых блоков Σ_{block}. Каждый элемент Σ_{if} имеет вид `if` *C* `goto` *i*, где *C* — это *условие*, которое может принимать значения *true* или *false*, а *i* — *метка перехода*, принимающая целое неотрицательное значение. Элементы множества Σ_{goto} имеют вид `goto` *i*. Для генерации структур переходов устройство слотов задержки и базовых блоков не важно.

Обозначим через Σ объединение $\Sigma_{jump} \cup \Sigma_{delay} \cup \Sigma_{block}$. Структурой переходов называется конечная последовательность *S* из элементов множества Σ, в которой после каждого перехода (элемента множества Σ_{jump}) следует слот задержки (элемент множества Σ_{delay})[6], а все метки переходов удовлетворяют ограничению $0 \le i < |S|$. Пример структуры переходов приведен ниже.

$$\{B_0, \texttt{if } C_1 \texttt{ goto } 6, D_2, B_3, \texttt{goto } 1, D_5, B_6, \texttt{if } C_7 \texttt{ goto } 0, D_8\}.$$

Здесь и далее мы будем обозначать слоты задержки буквой *D* с некоторым индексом, а базовые блоки — буквой *B*. Заметим, что в структурах переходов допускаются переходы на любые инструкции, в том числе на самих себя. Структура переходов, в которой нет ни одного перехода, называется *вырожденной*.

При построении структур переходов задаются такие параметры, как размер тестового шаблона и число инструкций переходов. Можно наложить и некоторые дополнительные ограничения, чтобы уменьшить число тестов, например, можно запретить подряд идущие базовые блоки, если нет переходов во внутренние блоки. Схема комбинаторной генерации структур переходов достаточна простая, поэтому мы не будем останавливаться на ней подробно. Сначала перебираются различные цепочки инструкций, включая инструкции переходов, затем — метки переходов. При использовании дополнительных ограничений осуществляется отсечение некоторых решений.

4.2. Построение трасс выполнения

Трассой выполнения структуры переходов $S = \{S[i]\}$ называется последовательность $T = \{T_j\}$ из индексов элементов *S*, которая удовлетворяет следующим свойствам:

[5] Инструкции вызова подпрограмм и возврата из них для простоты изложения не рассматриваются.

[6] Напомним, что мы рассматриваем случай, когда у инструкции перехода имеется ровно один слот задержки. Если слота задержки нет или их несколько, ограничения очевидным образом меняются.

- $T_0 = 0$[7];
- если $S[T_i] \in \Sigma_{if}$ и $S[T_i]$ = `if` C_i `goto` L_i, тогда
    - $|T| > i+1$, $T_{i+1} = T_i+1$[8] и выполняется одно из свойств:
        - если $T_{i+1} = |S|-1$, $|T| = i+2$ (C_i = *false*);
        - если $T_{i+1} < |S|-1$, $|T| > i+2$ и $T_{i+2} = T_i+2$ (C_i = *false*);
        - $|T| > i+2$ и $T_{i+2} = L_i$ (C_i = *true*);
- если $S[T_i] \in \Sigma_{goto}$ и $S[T_i]$ = `goto` L_i, тогда
    - $|T| > i+2$, $T_{i+1} = T_i+1$ и $T_{i+2} = L_i$;
- если $S[T_i] \in \Sigma_{block}$, тогда
    - если $T_i = |S|-1$, $|T| = i+1$;
    - если $T_i < |S|-1$, $|T| > i+1$ и $T_{i+1} = T_i+1$.

Примером трассы выполнения для рассмотренной в предыдущем разделе структуры переходов {B_0, `if` C_1 `goto` 6, D_2, B_3, `goto` 1, D_5, B_6, `if` C_7 `goto` 0, D_8} является последовательность:

{0, 1, 2, 6, 7, 8}.

Ей соответствует следующий *поток выполнения* (последовательность элементов структуры переходов):

{B_0, `if` C_1 `goto` 6, D_2, B_6, `if` C_7 `goto` 0, D_8}.

Заметим, что в большинстве случаев по трассе можно восстановить значения истинности условий выполняемых переходов (исключения составляют ситуации, когда метка перехода указывает на инструкцию, следующую за слотом задержки). В дальнейшем будем считать, что каждое вхождение условного перехода в трассу выполнения (или поток выполнения) помимо индекса (соответственно, элемента структуры переходов) содержит также значение истинности условия. Таким образом, трасса и соответствующий ей поток выполнения для рассматриваемого примера имеют вид:

{0, (1, *true*), 2, 6, (7, *false*), 8} и

{B_0, `if` $C_{1|true}$ `goto` 6, D_2, B_6, `if` $C_{7|false}$ `goto` 0, D_8}.

Трасса выполнения полностью определяется своей подпоследовательностью, содержащей только вхождения условных переходов. Такая подпоследовательность называется *редуцированной трассой*. Для рассматриваемого примера редуцированная трасса имеет вид {(1, *true*), (7, *false*)}.

---

[7] Для простоты считается, что выполнение начинается с первой инструкции.

[8] Выполнение слота задержки перехода.

Рассмотрим более сложную трассу выполнения той же самой структуры переходов:

{0, (1, *true*), 2, 6, (7, *true*), 8, 0, (1, *false*), 2, 3, 4, 5, (1, *true*), 2, 6, (7, *false*), 8}.

Ее редуцированное представление имеет вид:

{(1, *true*), (7, *true*), (1, *false*), (1, *true*), (7, *false*)}.

Или еще короче в форме отображения:

{(1 → {*true*, *false*, *true*}), (7 → {*true*, *false*})}.

Последовательность, состоящая из значений истинности условий перехода, называется *трассой выполнения инструкции перехода*. Пусть T — трасса выполнения структуры переходов, тогда через $T(i)$ будем обозначать трассу выполнения инструкции перехода с индексом i.

При построении трасс выполнения заданной структуры переходов указываются такие параметры, как максимальная длина трассы выполнения, максимальная длина редуцированной трассы или максимальная длина трассы одного перехода. Для перечисления трасс, удовлетворяющих такого рода ограничениям, можно использовать поиск в глубину. Ниже приводится алгоритм построения трасс выполнения, в котором используется представление редуцированных трасс в форме отображений. Кроме того, помимо трасс условных переходов в алгоритме используются трассы безусловных переходов (последовательности, состоящие только из *true*).

вход: структура переходов S;

вход: максимальная длина трассы перехода M[9];

выход: множество трасс выполнения R;

1. результат $R \leftarrow \varnothing$;
2. стек возвратов $B \leftarrow \varnothing$;
3. признак завершенности трассы $C \leftarrow$ *false*;
4. индекс текущей инструкции $i \leftarrow 0$;
5. текущая редуцированная трасса $T \leftarrow \varnothing$;
6. **цикл**: ищем первую инструкцию перехода в S, начиная с инструкции i:
    a. если инструкция перехода не найдена, то
        i. $R \leftarrow R \cup \{T\}$;
        ii. если $B = \varnothing$, то алгоритм завершает работу;
        iii. $C \leftarrow$ *true*;
        iv. $i \leftarrow top(B)$;
        v. перейти на **цикл**;
    b. если инструкция перехода найдена и j — ее индекс, то

[9] Для наглядности здесь используется ограничение на длины трасс выполнения отдельных переходов.

i. $i \leftarrow j$;

ii. если $C = false$ и $|T(i)| < M$, то

1. $T \leftarrow (T\backslash\{i\}) \cup \{i \rightarrow T(i) \cdot \{k\}\}$, где
   a. $k = false$, если $S[i] \in \Sigma_{if}$;
   b. $k = true$, если $S[i] \in \Sigma_{goto}$;
2. $push(B, i)$;
3. $i \leftarrow l$, где
   a. $l = i+1$, если $S[i] \in \Sigma_{if}$;
   b. $l = L_i$, если $S[i] \in \Sigma_{goto}$ и $S[i] =$ `goto` L_i;
4. перейти на **цикл**;

iii. если $C = true$ и $T(i)_{|T(i)|-1} = false$, то

1. $T \leftarrow (T\backslash\{i\}) \cup \{i \rightarrow T(i)_{0, \ldots, |T(i)|-2} \cdot \{true\}\}$;
2. $i \leftarrow L_i$, где $S[i] =$ `if` C_i `goto` L_i;
3. $C \leftarrow false$;
4. перейти на **цикл**;

iv. пока $B \neq \varnothing$ делать

1. $T \leftarrow (T\backslash\{i\}) \cup \{i \rightarrow T(i)_{0, \ldots, |T(i)|-2}\}$;
2. $i \leftarrow pop(B)$;
3. если $T(i) \neq \varnothing$ и $T(i)_{|T(i)|-1} = false$, то
   a. $T \leftarrow (T\backslash\{i\}) \cup \{i \rightarrow T(i)_{0, \ldots, |T(i)|-2} \cdot \{true\}\}$;
   b. $i \leftarrow L_i$, где $S[i] =$ `if` C_i `goto` L_i
   c. $C \leftarrow false$;
   d. перейти на **цикл**;

v. если $B = \varnothing$, то алгоритм завершает работу.

Рассмотрим пример построения множества трасс выполнения для структуры переходов $S = \{B_0,$ `if` C_1 `goto` $0,\ D_2\}$ при ограничении на длину трасс переходов $M = 2$.

В начале работы алгоритма $R \leftarrow \varnothing$, $B \leftarrow \varnothing$, $C \leftarrow false$, $i \leftarrow 0$ и $T \leftarrow \varnothing$. На первой итерации цикла ищется первая инструкция перехода в структуре S. Такая инструкция расположена по индексу $j = 1$ (это единственный переход в S). Делается присваивание $i \leftarrow j = 1$. Поскольку $C = false$, $|T(i)| < M$ ($|T(1)| = 0$) и $S[i] \in \Sigma_{if}$ ($S[1] =$ `if` C_1 `goto` 0), в трассу $T(i)$ добавляется $k = false$ ($T(1) = \{false\}$), в стек возвратов B добавляется индекс 1 ($B = \{1\}$), индекс i увеличивается на 1 ($i = 2$). После этого осуществляется поиск перехода, начиная с индекса $i = 2$. Переход не найден, поэтому в результат добавляется трасса $T = \{(1 \rightarrow \{false\})\}$, делается присваивание $C \leftarrow true$, индекс i получает значение верхушки стека B ($B = \{1\}$ и $i = 1$).

На следующей итерации ищется переход, начиная с $i = 1$. Находится инструкция, расположенная по индексу $j = i = 1$. Поскольку $C = true$ и в трассе $T(1)$ последний элемент равен *false*, он меняется на *true* ($T(1) = \{true\}$), после чего делаются присваивания $i \leftarrow L_1 = 0$ и $C \leftarrow false$. Снова ищется переход — на этот раз, начиная с $i = 0$. Индекс найденной инструкции $j = 1$. Делается присваивание $i \leftarrow j = 1$. Поскольку $C = false$, $|T(i)| < M$ ($|T(1)| = 1$) и $S[i] \in \Sigma_{if}$, в трассу $T(i)$ добавляется $k = false$ ($T(1) = \{true, false\}$), в стек B добавляется индекс 1 ($B = \{1, 1\}$), индекс i увеличивается на 1 ($i = 2$). Далее ищется переход, начиная с $i = 2$. Такой инструкции нет. В результат добавляется трасса $T = \{(1 \rightarrow \{true, false\})\}$, после чего делаются присваивания $C \leftarrow true$ и $i \leftarrow top(B)$ ($B = \{1, 1\}$ и $i = 1$).

На очередной итерации ищется переход, начиная с $i = 1$. Снова $j = i = 1$. Поскольку $C = true$ и в трассе $T(1)$ последний элемент равен *false*, осуществляется его замена на *true* ($T(1) = \{true, true\}$). После этого выполняются присваивания $i \leftarrow L_1 = 0$ и $C \leftarrow false$. Снова ищется переход, и находится по индексу $j = 1$. Делается присваивание $i \leftarrow j = 1$. Поскольку $|T(i)| = M$ и $B \neq \varnothing$, из трассы $T(i)$ удаляется последний элемент ($T(1) = \{true\}$), из стека извлекается индекс 1 и присваивается i ($B = \{1\}$ и $i = 1$). Последний элемент трассы $T(i)$ не равен *false*. Трасса укорачивается ($T(1) = \varnothing$), из стека извлекается индекс и присваивается i ($B = \varnothing$ и $i = 1$). Стек пуст, поэтому алгоритм завершает свою работу. Его результатом являются две трассы:

- $T = \{(1 \rightarrow \{ false\})\}$;
- $T = \{(1 \rightarrow \{true, false\})\}$.

4.3. Построение управляющего кода

Для построения управляющего кода, то есть кода отвечающего за проход по требуемой трассе, используется следующий подход. Инструкциям перехода сопоставляются уникальные регистры, называемые *управляющими*, которые не используются в других инструкциях тестового шаблона[10]. В определенные позиции тестового шаблона вставляются инструкции, называемые *управляющими*, которые изменяют значения управляющих регистров так, чтобы обеспечить требуемую трассу выполнения. Совокупность управляющих инструкций образуют *управляющий код*.

Чтобы обеспечить заданные значения истинности условий, для каждой инструкции условного перехода в тестовую программу добавляется массив значений и осуществляется загрузка одного или обоих управляющих регистров последовательными значениями этого массива. Возникает задача

[10] При наличии зависимостей по регистрам между инструкциями переходов и другими инструкциями тестового шаблона задача построения управляющего кода существенно усложняется и не всегда разрешима.

нахождения позиций в тестовом шаблоне, в которых нужно делать такую загрузку. Для наглядности рассмотрим эту задачу при условии, что управляющие инструкции можно добавлять только в базовые блоки.

Пусть дана невырожденная структура переходов S и ее трасса T. Пусть i — индекс некоторой инструкции перехода в структуре S. Будем говорить, что трасса $T(i)$ является *пустой*, если i не содержится в T. Непустые трассы, содержащие равные между собой значения истинности условий, будем называть *тривиальными*. Таким образом,

- {} — пустая трасса;
- {*true*, *true*, …, *true*} — тривиальная трасса;
- {*false*, *false*, …, *false*} — тривиальная трасса;

Напомним, что начальные значения истинности условий переходов обеспечиваются инициализирующими инструкциями тестового шаблона, поэтому управляющий код имеет смысл только для тех инструкций условных переходов, трассы которых являются непустыми и нетривиальными.

Сегментом $T_{i,j}$ трассы выполнения T называется ее фрагмент $\{T_i, T_{i+1}, \ldots, T_j\}$, заключенный между двумя соседними вхождениями одной и той же инструкции перехода. Пусть $T_i = (k, C_1)$, а $T_j = (k, C_2)$. В этом случае говорят, что $T_{i,j}$ является сегментом инструкции k. Если к тому же $C_1 = C_2$, сегмент называется *тривиальным*.

Очевидно, что управляющий код должен вызываться, по крайней мере, внутри нетривиальных сегментов. Нетрудно показать, что при достаточном числе регистров (когда каждая инструкция перехода имеет по крайней мере один управляющий регистр отличный от других) справедливо следующее утверждение.

Утверждение. Для заданной трассы выполнения структуры переходов построение управляющего кода возможно тогда и только тогда, когда внутри каждого нетривиального сегмента трассы есть хотя бы один базовый блок.

Рассмотрим структуру переходов S и ее трассу выполнения T. Пусть i — индекс некоторой инструкции условного перехода в структуре S, а I — подмножество индексов базовых блоков. Будем говорить, что множество I является *покрытием* инструкции i базовыми блоками, если для любого нетривиального сегмента T' инструкции i существует индекс $j \in I$, лежащий в T'. Базовые блоки, входящие в покрытие, будем называть *покрывающими*.

Рассмотрим следующую задачу. Для заданной структуры переходов и ее трассы выполнения требуется найти минимальное покрытие базовыми блоками каждой инструкции условного перехода. В некотором смысле, это

задача минимизации объема управляющего кода[11]. Задачи такого типа известны как *задачи о минимальном покрытии*. Для их эффективного решения используются приближенные алгоритмы, среди которых наиболее известен «жадный» алгоритм. Это простой метод, идея которого, применительно к нашей задаче, состоит в следующем. Для каждой инструкции условного перехода сначала выбирается базовый блок, который покрывает максимальное число нетривиальных сегментов рассматриваемой инструкции, затем из числа оставшихся берется блок, покрывающий максимальное число еще не покрытых сегментов и т.д. Алгоритм завершается, когда множество выбранных блоков становится покрытием.

В качестве примера рассмотрим структуру переходов

$$\{B_0, \texttt{if } C_1 \texttt{ goto } 6, D_2, B_3, \texttt{goto } 1, D_5, B_6, \texttt{if } C_7 \texttt{ goto } 0, D_8\}$$

и ее трассу выполнения

$$\{0, (1, \textit{true}), 2, 6, (7, \textit{true}), 8, 0, (1, \textit{false}), 2, 3, 4, 5, (1, \textit{true}), 2, 6, (7, \textit{false}), 8\}.$$

Для инструкции условного перехода, расположенной по индексу 1, имеется два нетривиальных сегмента:

- {(1, *true*), 2, **6**, (7, *true*), 8, **0**, (1, *false*)}, содержащий два базовых блока 0 и 6;
- {(1, *false*), 2, **3**, 4, 5, (1, *true*)}, содержащий один базовый блок 3.

Каждый базовый блок из множества {0, 3, 6} покрывает только один из двух сегментов. В качестве покрытия можно использовать {0, 3} или {3, 6}.

Пусть для каждой инструкции условного перехода построено покрытие базовыми блоками. В каждый покрывающий блок вставляются управляющие инструкции. В частности, это могут быть инструкции вида:

```
lw   R1,(P1)
addi P1, P1, 4
```

Первая инструкция загружает в один из управляющих регистров (регистр `R1`) значение из заранее подготовленного массива. Вторая инструкция увеличивает текущую позицию в массиве (регистр `P1`). При таком подходе значение второго управляющего регистра считается фиксированным.

[11] Теоретически, управляющий код может оказывать влияние на ситуацию, создаваемую тестовым шаблоном, поэтому, чтобы снизить это влияние, желательно строить управляющий код как можно меньшего объема. Для большинства практических задач влиянием управляющего кода небольшого объема можно пренебречь.

Заполнение массивов осуществляется инициализирующими инструкциями. Для каждой инструкции условного перехода берется первое значение истинности условия. Последовательно просматриваются элементы совокупной трассы выполнения. Если встречается базовый блок, принадлежащий покрытию, в массив заносится произвольное значение управляющего регистра, обеспечивающее текущее значение истинности условия. Если встречается рассматриваемая инструкция перехода, смотрится, это ее последнее вхождение или нет. Если вхождение не является последним, текущее значение истинности меняется на следующее, иначе, инициализация массива завершается.

Для трасс с «простой» структурой возможно построение компактного управляющего кода, состоящего из одной управляющей инструкции на одну инструкцию условного перехода. Это удобно, поскольку управляющую инструкцию в этом случае можно разместить в слоте задержки перехода.

Рассмотрим инструкцию, в которой в качестве условия перехода используется сравнение `R1 > R2`. Для удобства будем считать, что значение регистра `R2` равно 0. Если трасса выполнения перехода имеет вид {*true*, …, *true*, *false*, …, *false*}, можно в управляющий регистр `R1` записать положительное число (величина которого зависит от количества элементов *true* в трассе и числа выполнений покрывающих блоков), а в качестве управляющей инструкции использовать декремент `R1`. Если же трасса имеет вид {*false*, …, *false*, *true*, …, *true*}, можно записать в `R1` неположительное значение и использовать в качестве управляющей инструкции инкремент `R1`.

Сравнения вида >, ≥ и ≤ обрабатываются аналогичным образом. При сравнении на равенство или неравенство возможно аналогичное упрощение управляющего кода для трасс выполнения перехода, содержащих одно вхождение *true* или *false* соответственно, то есть трасс вида {*false*, …, *false*, *true*, *false*, …, *false*} для равенства и {*true*, …, *true*, *false*, *true*, …, *true*} для неравенства.

5. Инструментальная поддержка

В данном разделе дается краткое описание инструментальной поддержки предложенных методов, и приводятся примеры сгенерированных тестовых программ.

5.1. Генератор MicroTESK

Рассмотренные техники построения тестовых программ для модулей обработки переходов микропроцессоров интегрированы в генератор MicroTESK [7], разработанный в Институте системного программирования РАН. Генератор строит тестовые программы путем комбинирования цепочек инструкций ограниченной длины. Для каждой цепочки перебираются множества зависимостей и тестовых ситуаций (для инструкций перехода в

качестве ситуаций используются трассы выполнения). Для сокращения перебора используются различные эвристики, например, факторизация инструкций, ограничение глубины зависимостей и ограничение длины трасс выполнения.

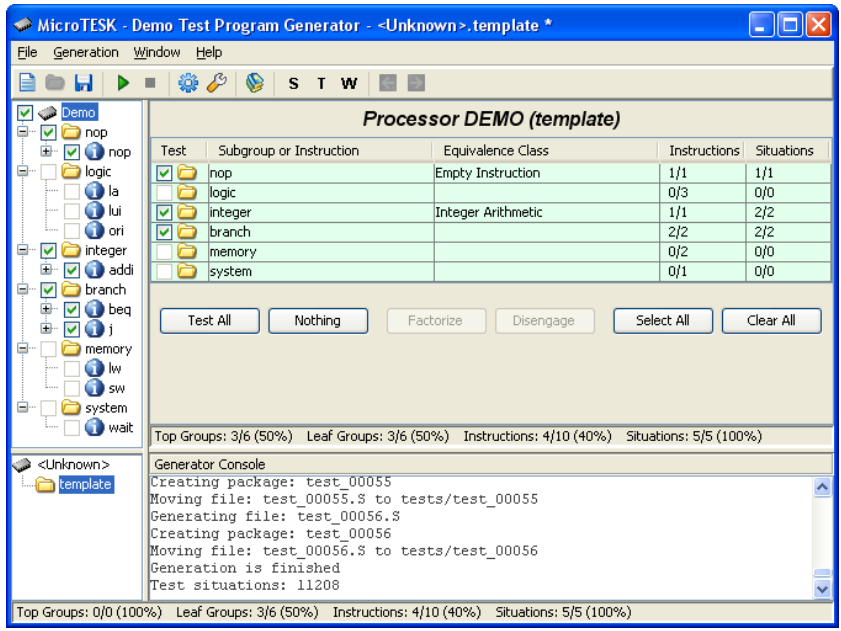


Рис. 3. Графический интерфейс демо-версии генератора MicroTESK.

Более подробная информация по методу, используемому в MicroTESK, доступна в статье [5]. Демо-версию генератора можно скачать на сайте [8].

5.2. Примеры тестовых программ

Ниже приведены фрагменты тестовых программ в системе команд MIPS, построенных автоматически генератором MicroTESK. При построении программ использовалась одна инструкция условного перехода (инструкция `beq`) и одна инструкция безусловного перехода (инструкция `j`). Для базовых блоков и слотов задержки использовались инструкции `nop` и `addi`. Рассматриваемые программы предполагают следующую обработку исключений. Если исключение возникает при выполнении инструкции базового блока, управление передается на следующую за ней инструкцию. Если исключение происходит в слоте задержки перехода, инструкция, вызвавшая исключение, заменяется на `nop`, после чего управление повторно передается на инструкцию перехода.

Пример 1

```
// Инициализация инструкции beq[3]
ori $15, $0, 0x0

// Инициализация инструкции addi[5]
lui $24, 0x8000
ori $24, $24, 0x2f56

// Тестовое воздействие
test_action_5016:

j 1f // Метка: 1, Трасса: {true}, Блоки: {}
1:
nop
beq $15, $0, 4f // Метка: 4, Трасса: {true}, Блоки: {}
4:
addi $2, $24, 0xcb01 // Переполнение
```

Пример 2

```
// Инициализация инструкции beq[1]
ori $2, $0, 0x0

// Инициализация инструкции addi[7]
lui $4, 0x7fff
ori $4, $4, 0x8f01

// Инициализация инструкции beq[8]
la   $25, beq_trace_array_1
ori  $7,  $0,  0x0
sw   $7,  0($25)
addi $25, $25, 0x4
ori  $7,  $0,  0x0
sw   $7,  0($25)
addi $25, $25, 0x4
lui  $7,  0xb566
ori  $7,  $7,  0xa7e8
sw   $7,  0($25)
addi $25, $25, 0x4
la   $25, beq_trace_array_1

// Инициализация инструкции addi[9]
lui $12, 0x7fff
ori $12, $12, 0xf916
```

```
// Тестовое воздействие
test_action_5197:

0:
beq $2, $0, 2f // Метка: 2, Трасса: {true, false,
 false}, Блоки: {2}
nop
2:
addi $2, $2, 0xffff // Управляющий код для beq[1]
lw $9, 0($25)       // Управляющий код для beq[8]
addi $25, $25, 0x4  // Управляющий код для beq[8]
addi $29, $4, 0x73a9 // Переполнение
beq $9, $0, 0b // Метка: 0, Трасса: {true, true, false},
 Блоки: {2}
addi $15, $12, 0x5835 // Переполнение
```

6. Заключение

В статье рассмотрены общие задачи, возникающие при построении тестовых программ для модулей обработки переходов микропроцессоров, и методы автоматизации их решения. Предложенные методы можно использовать в инструментах генерации тестовых программ для микропроцессоров, в частности, они реализованы в генераторе MicroTESK. Описанный в статье подход не зависит от микроархитектуры микропроцессора. С одной стороны, это является достоинством, поскольку подход позволяет быстро создавать тесты на самых ранних стадиях проектирования. С другой стороны, для детального тестирования могут потребоваться более узконаправленные тесты, учитывающие специфику механизмов обработки переходов конкретной микропроцессорной архитектуры. Методы автоматизации разработки таких тестов являются темой наших дальнейших исследований.

Литература

[1] D. Patterson, J. Henessy. *Computer Organization and Design*. 3rd Edition, Morgan Kaufmann, 2005.

[2] Википедия (http://en.wikipedia.org), статья *Branch delay slot*.

[3] Википедия (http://en.wikipedia.org), статья *Branch predictor*.

[4] T.-Y. Yeh, Y.N. Patt. *Two-Level Adaptive Training Branch Prediction*. Proceedings of International Symposium on Microarchitecture, 1991.

[5] А.С. Камкин. *Генерация тестовых программ для микропроцессоров*. Труды ИСП РАН, т. 14, ч. 2. М., 2008. С. 23–63.

[6] MIPS64™ Architecture For Programmers. Revision 2.0. MIPS Technologies Inc., June 9, 2003.

[7] A. Kamkin. *MicroTESK: Automation of Test Program Generation for Microprocessors*. Proceedings of East-West Design & Test Symposium, 2009.

[8] http://hardware.ispras.ru

Разработка тестового набора для верификации реализаций протокола безопасности IPsec v2

А.В. Никешин, Н.В. Пакулин, В.З. Шнитман

Аннотация. Статья посвящена разработке тестового набора для проверки соответствий реализаций узлов Интернет спецификациям нового протокола безопасности IPsec v2 [1-7]. Для построения тестового набора использовалась технология автоматического тестирования UniTESK [8] и программный пакет CTesK [9], реализующий эту технологию.
Работа выполнялась в Институте системного программирования РАН в рамках проекта «Верификация функций безопасности протокола нового поколения IPsec v2» при поддержке гранта РФФИ № 07-07-00243. В ходе ее выполнения были выделены требования к реализациям IPsec v2, разработаны формальные спецификации и прототип тестового набора для верификации реализаций IPsec v2, в том числе реализаций протокола автоматического создания контекстов безопасности IKEv2. В статье описаны метод формализации требований IPsec v2, процесс создания тестового набора, а также результаты тестирования существующих реализаций. Эти результаты показывают, что предложенный в данной работе метод верификации позволяет эффективно автоматизировать тестирование таких сложных протоколов, как протоколы безопасности.

1. Введение

Под семейством протоколов IPsec обычно понимается вся совокупность технических средств, обеспечивающих защиту передачи данных на уровне IP. В настоящее время эксплуатируются две несовместимые версии IPsec, которые обычно обозначаются как IPsec v1 и IPsec v2. В данной статье под IPsec мы будем понимать исключительно IPsec v2.

Семейство протоколов IPsec представляет собой инфраструктуру безопасности: спецификации IPsec v2 включают описание архитектуры обеспечения безопасности передачи данных в сетях IP, собственно протоколы защиты данных Authentication Header (AH) и Encapsulating Security Payload (ESP), протокол автоматического установления защищённого соединения Internet Key Exchange (IKEv2), а также криптографические алгоритмы аутентификации и шифрования [1-7].

Семейство IPsec предоставляет целую совокупность сервисов безопасности: контроль доступа, целостность в режиме без установления соединения (целостность дейтаграмм), аутентификацию источника данных, обнаружение и отклонение повторно воспроизводимых сообщений (вид частичной целостности последовательности сообщений), конфиденциальность данных и ограниченную конфиденциальность потоков трафика. Эти сервисы предоставляются на сетевом уровне и обеспечивают прозрачность для протоколов транспортного и верхнего уровней. IPsec предлагает единый стандартный метод защиты для всех протоколов, которые могут работать поверх IP (включая сам IP). Кроме того, IPsec v2 включает спецификацию минимальной функциональности межсетевых экранов, поскольку она является существенным аспектом организации контроля доступа на уровне IP.

Задачу тестирования соответствия можно условно разделить на две подзадачи: построение тестовых воздействий и оценка правильности наблюдаемых результатов. К первой задаче примыкает проблема оценки полноты покрытия – чем шире будет спектр тестовых воздействий, тем шире получится охват функций протокола при тестировании. Вторая задача заключается в вынесении вердикта о соответствии тестируемой системы спецификации протокола.

Разработанный метод верификации [10] основан на автоматизированном тестировании соответствия формальным спецификациям. Требования, представленные в тексте стандарта, изложены на английском языке и представляют собой неформальный текст, описывающий желаемое поведение системы на естественном языке. Для того, чтобы автоматизировано извлечь тесты для протокола, необходимо перевести его спецификацию в вид, пригодный для решений этой задачи. В разработанном подходе в этой роли выступают формальные спецификации, в которых требования задаются как логические выражения, записанные посредством математического формализма.

В основе подхода лежит представление протоколов как асинхронных автоматов, причем автомат задаётся неявно посредством контрактных спецификаций. А именно, формальное описание поведения протокола задаётся как контрактная спецификация: набор сообщений протокола рассматривается как некоторый формальный интерфейс между реализацией протокола и её окружением, поведение протокола описывается посредством пред- и постусловий. Такое задание протоколов позволяет описывать поведение сложных недетерминированных протоколов, таких как протоколы защиты передачи данных в сетях IP. Эти протоколы отличаются сложными структурами данных сообщений и состояния, недетерминированным поведением и неполными спецификациями – в спецификациях умышленно оставлены пробелы для облегчения реализации протоколов. Контрактные спецификации позволяют представлять требования к сложным протоколам в форме, пригодной для автоматизированного тестирования реализаций таких

протоколов. Для протоколов, формальная спецификация которых задана в виде контрактной спецификации, разработан метод автоматизированного построения тестовых последовательностей с полностью автоматическим вынесением вердиктов о соответствии наблюдаемого поведения реализации её спецификации. В совокупности разработанные методы позволяют автоматизировать тестирование соответствия реализаций сложных сетевых протоколов их спецификациям.

В рамках предложенного метода тест представляет собой конечный автомат. С каждым переходом автомата сопоставлено определённое тестовое воздействие. При выполнении перехода это воздействие подаётся на тестируемую реализацию, регистрируются реакции реализации и автоматически выносится вердикт о соответствии наблюдаемого поведения спецификации. Обход автомата теста совершается автоматически во время тестирования, алгоритм обхода не зависит от протокола, тестируемой реализации или конкретного теста. Последовательность переходов определяет тестовую последовательность. В силу недетерминизма протоколов и различий в поддержке необязательных функций в реализациях конкретные тестовые последовательности, получаемые при прогоне тестов на разных реализациях, могут не совпадать друг с другом.

Предложенный подход к верификации функций безопасности включает два метода: метод формализации стандартов протоколов и метод формального задания тестовых наборов [11]. Метод формализации стандартов протоколов включает анализ спецификации протокола и извлечение требований, определение формального интерфейса протокола, формализацию функциональных требований к реализации протокола, задание критериев покрытия и разработку функции реконструкции состояния. Метод формализации тестовых последовательностей состоит из определения целей тестирования, разработки проекта автомата теста для конкретной цели тестирования, задания переходов автомата теста, задания функции определения состояния автомата теста по модельному состоянию контрактной спецификации протокола, проектирования настроечной информации автомата теста и разработки формата для представления опций, а также включает прогон тестового сценария и анализ результатов тестирования. Для описания автоматов тестов в данном методе используется специальный вид задания автоматов тестов, который называется тестовым сценарием.

2. Обзор процесса формализации спецификаций IPsec

2.1. Извлечение требований

Процесс формализации стандартов начинается с извлечения требований – утверждений, определяющих наблюдаемое поведение реализации протокола в зависимости от внутреннего состояния реализации и входных воздействий, оказываемых на неё окружением. Требования к протоколам Интернета

изложены на английском языке в документах, называемых по историческим причинам «запросами комментариев» (Request for Comments, RFC)

Для построения формальной спецификации IPsec прежде всего необходимо из этих документов извлечь требования, регламентирующие поведение реализаций протоколов. Процесс извлечения требований связан с решением четырех задач: определение набора документов, регламентирующих протокол, идентификация функциональных требований в регламентирующих документах протокола, сведение требований в каталог и систематизация требований. Ниже этот процесс описан более подробно.

2.1.1. Определение набора документов, регламентирующих протокол

Регламентирующие документы IPsec разрабатываются Комитетом по стандартизации Интернета (Internet Engineering Task Force, IETF). Спецификации протоколов Интернета публикуются в формате IETF RFC [12] на английском языке. Анализ структуры документации начинается с изучения обзоров протоколов Интернета. Цель изучения – обнаружить связи между протоколами и составить список RFC для подробного изучения. В этот список должны быть занесены RFC, которые регламентируют сам протокол, его расширения и дополнения, а так же другие протоколы Интернета, с которыми протокол взаимодействует.

К настоящему моменту разработано более 50 документов, регламентирующих различные аспекты IPsec – применение криптографических алгоритмов, параметры настройки, базовые протоколы и т.д. К основным документам, регламентирующим IPsec v2 относятся спецификации [1-4].

2.1.2. Идентификация функциональных требований

Для обеспечения прослеживаемости требований, т.е для создания ссылок из разработанных формальных спецификаций в требования из регламентирующих документов, необходимо представить требования как уникальные именованные сущности. В данной работе эта задача решена составлением каталога требований.

Каталог требований – это список требований, налагаемых регламентирующими документами и общим контекстом предметной области. Каждое требование, извлечённое из документации, каталог привязывает к соответствующему месту в тексте. Одно такое требование обычно соответствует логически замкнутому фрагменту текста, выражающему одно ограничение на элемент функциональности или элемент данных. Каждое требование имеет уникальный идентификатор.

Каталог требований позволяет в дальнейшем оценивать полноту формализации и адекватность тестирования в терминах исходного текста стандарта и предметной области.

2.1.3. Систематизация требований

Систематизация требований призвана разделять требования на группы «похожих» требований. Поскольку конечной целью является разработка формальной спецификации требований, систематизация предназначена для выделения групп требований, которые формализуются сходным образом.

1. Разделение требований на **ограничения поведения и ограничения целостности**. Под ограничениями поведения мы понимаем требования к поведению реализации протоколов IPsec в определённых условиях или некоторой ситуации. Под ограничениями целостности мы понимаем требования к связям между элементами состояния протокола, между полями сообщений протокола и т.п., которые должны выполняться на протяжении всего времени функционирования системы или в большом количестве ситуаций, возникающих при работе реализации.

2. Разделение требований по **объектам требований**. Требования в спецификации предъявляются к поведению различных компонентов или объектов, входящих в архитектуру IPsec. Требования, описывающие поведение одного объекта, удобно выделять в одну группу, это облегчает использование каталога требований в последующих работах процесса.

3. Разделение по **степени обязательности**. Этот критерий отражает тот факт, что во многих протоколах выделяются функции протокола, которые должны быть представлены в каждой реализации, и необязательные для поддержки в реализации. В спецификациях протоколов Интернета выделяется три уровня обязательности [13]:
   - Обязательные – требования из этой категории должны быть представлены в каждой реализации протокола. Такие требования помечаются в тексте спецификации ключевым словом MUST (негативные требования помечаются фразой MUST NOT).
   - Рекомендуемые – реализации могут нарушить требования из этой категории, но только если есть веские причины так поступить, и авторы реализации отдают себе отчёт обо всех последствиях такого шага. Такие требования помечаются ключевыми словами SHOULD и RECOMMENDED (негативные требования помечаются фразами SHOULD NOT и NOT RECOMMENDED).
   - Необязательные – авторы реализации могут не включать поддержку требований из этой категории, если, скажем, это не требуется в тех условиях, в которых будет использоваться реализация. Впрочем, есть одно ограничение «свободы выбора» – реализации, в которых отключена поддержка некоторого требования из числа необязательных, должны (MUST) быть в состоянии взаимодействовать с реализациями, в которых поддержка такого требования включена. Аналогично, реализация, в которой включена поддержка некоторого необязательного требования, должна (MUST)

быть в состоянии взаимодействовать с реализациями, в которых поддержка требования отключена.

4. Разделение требований на **позитивные и негативные**. Под позитивными требованиями здесь понимаются требования, предписывающие реализации совершить определённые действия, например, отправить сообщение в сеть или возвратить некоторый код ответа. Под негативными требованиями понимаются требования, которые запрещают реализации совершать определённые действия.

   Например, спецификация архитектуры IPsec ЗАПРЕЩАЕТ реализации использовать одновременно нулевой алгоритм шифрования и нулевой алгоритм цифровой подписи.

   Этот критерий важен при формализации, так как позитивные и негативные требования формализуются различным образом. Данный критерий делит требования на два класса.

Указание категории, к которой относится требование, включается в каталог требований. В рамках отдельного критерия требование принадлежит только к одной категории, но часто встречается ситуация, когда одно требование подпадает под несколько критериев. В таком случае в каталог заносятся все выделенные категории.

2.1.4. Классификация требований по объекту требований

Мы вводим следующее деление требований в зависимости от объектов требований.

1. Требования к связям между полями сообщениями протокола.
2. Ограничения целостности состояния протокола.
3. Требования к обработке входящих сообщений.
4. Требования к содержимому исходящих сообщений.
5. Требования к обработке последовательностей сообщений. Фактически, это требования к отдельным переходам в автомате протокола, но у многих протоколов Интернета требования к обработке входящих пакетов и заполнению исходящих пакетов представлены без какой-либо сопутствующей модели состояния и переходов. При выделении требований из этой группы необходимо реконструировать автомат протокола.
6. Требования к взаимодействию с протоколами нижнего уровня. Сюда входят
   - правила представления сообщения протокола в пакетах нижележащего уровня,
   - правила отображения адресов протокола в адреса нижележащего уровня,
   - правила заполнения полей в сообщениях протокола в зависимости от конфигурации протокола нижележащего уровня

(например, использование протоколов шифрования и цифровой подписи в протоколе IKE).

7. Требования к взаимодействию с протоколами верхнего уровня. Сюда входят
   - правила представления сообщений протокола верхнего уровня в сообщениях рассматриваемого протокола,
   - управление состоянием протокола со стороны верхнего уровня,
   - сервисы, которые протокол предоставляет верхнему уровню
8. Требования к программному интерфейсу реализации IPsec. В спецификации протокола есть указания на требования к реализации программного интерфейса, но нет описаний типов данных и сигнатур соответствующих функций.

В данной работе из основополагающих документов IPsec v2 [1-7] было выделено в общей сложности 850 функциональных требований, из них 60% обязательных (категории MUST), 21% рекомендуемых (категории SHOULD), и 19% опциональных требований (категории MAY).

2.2. Формализация требований спецификации

2.2.1. Определение формального интерфейса протокола

Перед формализацией требований необходимо определить формальный интерфейс IPsec v2. А именно, необходимо решить следующие задачи:

1. Представить взаимодействие реализации IPsec с окружением в виде набора входных и выходных событий.
2. Определить набор внутренних событий, существенных для поведения реализации IPsec.
3. Определить сигнатуры элементов формального интерфейса, соответствующих выделенным множествам событий.

При определении элементов формального интерфейса необходимо учитывать семантику параллельного возбуждения событий, соответствующих элементам контрактной спецификации: оно должно быть эквивалентно некоторой последовательности возбуждения событий (семантика чередования или interleaving).

При составлении формального интерфейса необходимо учитывать указанное ограничение. Набор выбранных элементов формального интерфейса должен быть достаточен для построения адекватной модели поведения системы.

2.2.2. Множества входных и выходных событий для различных функций IPsec

Наиболее просто устроен формальный интерфейс для взаимодействий через синхронный программный интерфейс. Примером такого программного интерфейса служат функции настройки реализации IPsec через программный интерфейс реализации или расширения интерфейса сокетов. Так как

синхронные функции выполняются последовательно, то для синхронных функций формальный интерфейс, как правило, включает ровно по одному элементу для каждой интерфейсной функции.

Состав формального интерфейса для протоколов AH и ESP устроен следующим образом. Для каждого сообщения из алфавита сообщений в формальный интерфейс вводятся два элемента:

Прием сообщения протокола. В этом элементе формального интерфейса специфицируется обработка входящего сообщения.

Отправка сообщения. Здесь специфицируется проверка корректности исходящего сообщения.

Протокол IKEv2 расположен на прикладном уровне стека TCP/IP и использует протокол UDP для обмена сообщениями. Особенность IKEv2 заключается в том, что сообщения этого протокола обрабатываются на транспортном уровне протоколами IPsec. Несмотря на это формальный интерфейс для протокола IKE v2 устроен аналогичным образом – он включает два элемента: прием сообщения IKEv2 и отправка сообщения IKEv2.

Задание сигнатур элементов формального интерфейса

При задании сигнатур элементов формального интерфейса необходимо определить типы для представления входных и выходных значений элементов формального интерфейса. Из описания состава формальных интерфейсов следует, что элементы формального интерфейса делятся на две категории:

Элементы, представляющие программный интерфейс.

Элементы, представляющие обмены сообщениями.

Различия между этими элементами заключаются в том, как они определены в спецификации протокола. Функции программного интерфейса, как правило, описываются явным заданием сигнатуры на языке Си, с определением типов входных и выходных значений. Сообщения протоколов, напротив, заданы как размеченные наборы битов в неформальном табличном задании.

Соответственно, задачи определения типов для элементов формального интерфейса из этих категорий различаются.

2.2.3. Определение типов для элементов, представляющих функции программного интерфейса

Для задания сигнатур элементов формального интерфейса следует использовать типы, определённые в спецификации протокола, а не те типы, которые определены в заголовочных файлах реализации. Это требование к формальной спецификации призвано упростить развёртывание и сборку тестового набора, в состав которого войдёт формальная спецификация. Дело в том, что тестирование, как правило, выполняется удалённо. Для функций из программного интерфейса это означает, что спецификация будет компилироваться не на целевом узле, на котором функционирует тестируемая

реализация, а на инструментальном узле. Так как на инструментальном узле может быть установлена, вообще говоря, совсем другая операционная система, чем на целевом узле, то нельзя гарантировать, что система типов реализации будет присутствовать на инструментальном узле.

2.2.4. Определение типов для представления сообщений протоколов

В рамках разработанного метода разработка типов данных для параметров функций из программного интерфейса протокола следует общему методу разработки формальных спецификаций средствами инструмента CTesK [9]. В данном разделе рассмотрим важную особенность предлагаемого метода – представление сообщений протоколов в формальной спецификации.

Рассмотрим примерные сигнатуры спецификационных функций receive_PDU и sent_PDU, которые формализуют требования к обработке входящего сообщения протокола и требования к содержимому исходящего сообщения протокола:

specification void receive_PDU(PDU_in * pdu);
reaction PDU_out * sent_PDU(**void**);

Входное событие receive_PDU принимает входящее сообщение в модельном представлении в качестве единственного параметра. Возвращаемое значение отсутствует.

Выходное событие sent_PDU не имеет входных параметров, как и положено реакции в нотации SeC [9], а в качестве формального результата возвращает вышедшее сообщение в модельном представлении.

Для представления входящих и исходящих сообщений используются разные типы. Тип для исходящих сообщений снабжён инвариантом, в котором формализованы ограничения целостности из спецификации протокола.

Для входящих сообщений инварианты не задаются, так как реализация протокола Интернета обязана принимать на вход любую последовательность байтов, включая некорректно сформированные (нарушающие инвариант) и обрезанные сообщения.

В рамках данного метода модельное представление сообщений протоколов задаётся в нотации SeC по следующей схеме:

```
struct _PDU base {
        Object * errorDesc;
        MemBuf * actualPDU;
        Type1    field1;
        // ...
        TypeN    fieldN;
        List *   options;
```

```
        TypePayload *     payload;
};
```

Структура начинается с двух служебных полей: поля errorDesc и actualPDU. В поле errorDesc заносится информация о нарушениях формата, обнаруженных в фактическом сообщении, доставленном в тестируемую реализацию или отправленном реализацией. Если в сообщении нет нарушений формата, то значение errorDesc равно NULL. В поле actualPDU хранится фактическое сообщение, переданное по сети. Это значение используется для отладки тестов и построения подробных отчётов о тестировании.

Каждое поле заголовка сообщения, включая битовые поля, представлено в структуре отдельным полем.

Если в формате сообщения предусмотрены списки опций переменной длины, то в модели они представляются списком – объектом типа List.

Полезная нагрузка сообщения представлена полем payload.

Согласно правилам CTesK необходимо объявить спецификационные типы для использования модельных сообщений в спецификациях.

```
specification typedef struct _base_PDU PDU_in;
invariant typedef PDU_in PDU_out;
invariant (PDU_out * pdu) {
        if (pdu->errorDesc != NULL) return false;
        // проверка ограничений целостности
}
```

Тип PDU_in реализован как простой спецификационный тип на основе _PDU_base. Это позволяет использовать имена полей и описание нарушений формата в спецификациях. Тип PDU_out реализуется как подтип PDU_in, ограничение подтипа задаётся логическим условием в инварианте типа. Инвариант проверяется автоматически до и после события, соответствующего элементу формального интерфейса. Инварианты для исходящих сообщений содержат больше ограничений, так как реализации IPsec могут принимать на вход любые сообщения, а отправлять должны только корректные.

2.2.5. Разработка пред- и постусловий для формального интерфейса протоколов безопасности

Построение формальной спецификации требований сродни программированию. В обоих случаях неформальный текст на естественном языке преобразуется в текст на строго определённом формальном языке.

В последнее время методология программирования сделала значительный шаг вперёд благодаря выделению приёмов, или паттернов, проектирования программ. Методология формализации пока разработана не столь глубоко,

поэтому далее в тексте мы представим приёмы формализации только для отдельных видов требований.

2.2.6. Разработка модельного состояния

Пред- и постусловия описывают ограничения на параметры стимулов и ограничения на наблюдаемое поведение в терминах изменений модельного состояния. Для того чтобы разработать общую спецификацию, не зависящую от конкретной реализации, необходимо, чтобы модельное состояние состояло из концептуальных объектов, которые имеют смысл для всех реализаций.

Естественным образом такие структуры данных строятся на основании концептуальных моделей протокола. В протоколах безопасности Интернета концептуальные структуры данных представлены преимущественно в двух документах: спецификации архитектуры IPsec [1] и спецификации информационной базы (MIB) политик безопасности [6].

Спецификация MIB представляет собой текст на диалекте языка ASN.1. Набор основных типов и информационных объектов определён в RFC 1213 [14] и RFC 2011 [15]. Преимущества использования MIB при разработке модельного состояния заключается в том, что в MIB описаны структуры данных, не зависящие от конкретных реализаций протокола. Фактически, MIB является формальным заданием наиболее детально разработанных концептуальных структур данных.

2.2.7. Формализация семантических связей в сообщениях протоколов

Спецификации протоколов определяют синтаксис сообщений, задавая побитное разделение сообщения на поля. Для каждого поля описывается семантика и, если есть, связь между полями.

Семантические связи между полями записываются в инварианте типа, представляющего исходящие сообщения. Для входящих сообщений инварианты типа не записываются, так как для протоколов Интернета допускается получение сообщений с нарушениями требований к формату сообщений.

Для протоколов IPsec прописываются правила по обработке нарушений формата и ограничений целостности в сообщениях. Соответственно, в постусловиях событий, соответствующих обработке входящих сообщений, добавляется проверка нарушений формата сообщений и специфицируется предписанная реакция на нарушения.

2.2.8. Формализация негативных требований

Как мы уже указывали, негативные требования запрещают реализации демонстрировать определённое поведение. Негативные требования формализуются в постусловии реакции.

Предположим, что запрещаемое поведение представляется некоторым логическим выражением expr, тогда в постусловии запрещение поведения выражается следующей инструкцией:

if (expr) { **return false**; }

Другими словами, если реализация демонстрирует запрещённое поведение, то постусловие выходного события возвращает негативный вердикт.

2.2.9. Формализация требований к ожидаемым реакциям

В большинстве формальных языков, поддерживающих контрактные спецификации, включая используемое в работе расширение языка Си, нет встроенной поддержки темпоральных спецификаций, то есть спецификаций свойств цепочки событий, упорядоченных во времени. Поэтому в этих формализмах требования к ожидаемой реакции формулируются в постусловии спецификационной функции, соответствующей реакции. Постусловие выносит вердикт на основании истории взаимодействий, которая хранится в модельном состоянии.

Рассмотрим виды выходных событий, которые возникают в протоколах безопасности:

Реакции на стимулы. Это реакции, которые, согласно спецификации протокола, должны быть выданы в ответ на определённое входное событие.

Реакции на внутренние события. К этой группе относятся внешне наблюдаемые действия, связанные с ненаблюдаемыми извне событиями. Наиболее распространённый вид внутренних событий – истечение таймеров.

2.2.10. Формализация требований к обязательности реакций

Предложенные выше методы позволяют проверять корректность наблюдаемых реакций. Тем не менее, этих приёмов недостаточно для решения следующей задачи: проверить, что реализация действительно демонстрирует некоторую реакцию.

В данной работе для решения этой задачи используется следующая методика. Предполагается, что тестируемая система после приёма любого стимула за некоторое ограниченное время переходит в стационарное состояние, в котором тестируемая система ожидает прибытие стимула, и выдача реакций без предшествующего им стимула невозможна. После выдачи тестового воздействия тестовая система собирает в течение некоторого времени реакции. Наблюдаемое поведение тестируемой системы считается корректным, если корректны все реакции и конечное состояние является стационарным. В противном случае поведение реализации считается некорректным.

В формальную спецификацию вводится функция проверки стационарности состояния, которая возвращает определённые значения в зависимости от того, является ли текущее состояние спецификации стационарным или нет.

Требования к обязательности реакций формализуются в функции проверки стационарности. Функция определяет состояние как нестационарное, когда в истории взаимодействия есть стимулы, для которых не наблюдались обязательные реакции, или активные таймеры. Благодаря этому тестовая система вынесет вердикт о некорректности поведения реализации, если реализация не выдала обязательную реакцию.

2.2.11. Формализация требований к реакции на входное событие

Рассмотрим, как представляются требования к реакции на входное событие в том случае, когда на него выдаётся не более одной реакции.

Информация о полученных стимулах сохраняется в состоянии формальной спецификации. Модельное состояние может хранить историю стимулов различными способами: в виде списка объектов, набора (мультимножество, multiset), отображения, множества.

Требования, устанавливающие связь между стимулом и реакцией, формализуются в функциях со следующей сигнатурой:
bool matchReaction(Object * s, ReactionType * r);

Первый аргумент соответствует входному событию, второй – реакции. Функция возвращает **true**, если реакция соответствует стимулу. Для каждой реакции необходимо разработать такую функцию.

В постусловие реакции добавляется проверка того, что в истории принятых стимулов есть стимул, для которого вызов stimulusMatchReaction, возвращает **true** для данной реакции.

```
reaction PDU_out * sent_Message() {
 post {
  // ...
  Predicate * p=new_Predicate(matchReaction, sent_Message)
  if (!exists_Iterator(@history_iter, p)) {
   return false;
  }
  // ...
 }
}
```

Если на один стимул должно быть выдано несколько реакций, то в историю добавляется не только стимул, но и счётчик ожидаемых реакций. С каждой реакцией, соответствующей стимулу, значение счётчика уменьшается на 1. В функцию проверки стационарности добавляется проверка того, что если есть

стимул с ненулевым значением счётчика, то состояние является нестационарным.

2.2.12. Формализация требований к реакциям на истечение таймеров

В модельном состоянии есть переменные, которые представляют состояние реализации протокола на концептуальном уровне. В частности, модельное состояние должно быть спроектировано таким образом, чтобы содержать информацию об активных таймерах.

В постусловии для реакции на истечение таймера проверяется, что соответствующий таймер был активен до возбуждения реакции. Если это не верно, то постусловие выносит негативный вердикт.

Для верификации демонстрируемых реакций необходимо, чтобы информация о стимулах и текущем состоянии сохранялась в модельном состоянии. Задачу изменения модельного состояния решают функции реконструкции состояния, поэтому модификация истории взаимодействия реализуется в них.

2.2.13. Формализация требований к необязательным функциям

Под необязательными функциями мы понимаем функции протокола, которые спецификация разрешает не включать в реализации.

В рамках данного метода используется следующий приём для формализации требований к таким функциям.

Для необязательных функций в формальную спецификацию включаются параметры, которые определяют включение требования в состав спецификации. В зависимости от значения параметра логические выражения, формализующие требование, игнорируются или вычисляются при использовании формальной спецификации.

Параметры представлены в формальной спецификации как глобальные переменные. Пользователь тестового набора задаёт значения параметров в конфигурационном файле. При запуске теста автоматически производится разбор конфигурационного файла, и присваиваются значения соответствующим глобальным переменным.

Совокупность фактических значений параметров спецификации служат формальной записью *описания соответствия реализации* (Implementation Conformance Statement). Значения параметров устанавливаются экспертами на основе текстового описания соответствия реализации, исследования исходных текстов реализации или изучения поведения реализации.

2.2.14. Формализация недоопределённых функций

Под недоопределёнными функциями мы понимаем функции протокола, для которых не все ветви поведения определены в спецификации. Такие недоопределённости вскрываются при анализе концептуальных моделей поведения.

Возможны две стратегии формализации недоопределённых функций, которые условно можно обозначить как «запрещающая» и «разрешающая». В запрещающей стратегии спецификация явно запрещает подачу стимулов, которые соответствуют неопределённым ветвям функциональности, и явно запрещает использование неопределённых элементов в реакциях. А именно:

В предусловиях входных событий задаются граничные условия, которые сужают область определения постусловия только теми событиями, обработка которых полностью определена в исходных текстовых спецификациях протокола.

Для выходных событий используется следующая интерпретация недоопределённого формата: реализации должны порождать только хорошо определённые реакции. Если наблюдаемая реакция содержит недоопределённые элементы, то постусловие события выносит негативный вердикт.

При следовании «разрешающей» стратегии разработчик спецификации вносит в спецификацию параметры, значения которых соответствуют различным возможным способам разрешить неопределённость. Значения параметров устанавливаются по результатам анализа документации реализации, исходных текстов или проведения экспериментов. Соответственно, в постусловии элементов формального интерфейса вердикт выносится на основании того, соответствует ли наблюдаемое поведение реализации протокола заданному набору параметров, описывающих, как разрешена неопределённость в тестируемой реализации.

3. Формальная спецификация IPsec v2

В соответствии с приведенной выше моделью тестирования формальная спецификация IPsec v2 состоит из нескольких компонентов:

- модельного состояния, которое содержит набор структур данных, моделирующих концептуальные структуры данных из стандартов IPsec v2, таких как база данных контекстов безопасности и база данных политик безопасности;
- формального интерфейса IPsec, включающего спецификационные стимулы, формализующие требования к изменению состояния реализации IPsec v2 при внешнем воздействии на систему, и спецификационные реакции, которые

формализуют требования к реакциям реализации IPsec v2 на внешние воздействия;

- критериев покрытия, идентифицирующих различные ветви функциональности IPsec.

3.1. Модельное состояние

Модельное состояние реализации IPsec представлено тремя структурами данных: множеством контекстов безопасности (Security Association Database - SAD), упорядоченным множеством политик безопасности (Security Policy Database – SPD), и упорядоченной базой данных авторизации партнеров (Peer Authorization Database - PAD). Полное описание этих структур приведено в RFC 4301 и RFC 4807.

Рассмотрим, как эти концептуальные структуры данных представлены в формальной спецификации.

Модель контекста безопасности представлена абстрактным типом **SA** со следующими полями:

selectors селекторы трафика, включающие удаленный IP-адрес, локальный IP-адрес, протокол следующего уровня, удаленный порт, локальный порт. Каждый селектор в соответствии со спецификацией может определять несколько значений. Для протоколов, не имеющих портов, селекторы портов могут задавать другие параметры, присущие этим протоколам (для ICMP - тип и код сообщения; для Mobile IP - тип сообщения). Подробное описание селекторов трафика приведено в RFC 4301;

spi индекс параметров безопасности, используется для поиска контекста безопасности для обработки входящего трафика;

proto идентификатор протокола IPsec, используемого в данном контексте безопасности - AH или ESP;

mode режим протокола IPsec: туннельный или транспортный;

direction Направление трафика, для которого применяется данный контекст безопасности: входящий или исходящий;

sequence_number счетчик порядковых номеров, используется при формировании исходящих пакетов;

sequence_window окно порядковых номеров полученных пакетов, используется для противодействия атакам повторного воспроизведения;

lifetime время жизни данного контекста безопасности. Оно может быть выражено временем или количеством байтов, или одновременным использованием обоих типов. Согласно

спецификации преимуществом обладает первое исчерпанное время жизни;

tunnel_data IP-адреса источника и места назначения туннельного заголовка;

Algorithm_Data идентификаторы криптографических алгоритмов, используемых в данном контексте безопасности, их параметры (ключи, размеры блоков и т.п.) Содержание этих структур зависит от используемого протокола IPsec.

Модель политики безопасности представлена абстрактным типом **SP**:

selectors селекторы трафика, определяют применимость данной политики к входящим или исходящим пакетам. Это поле аналогично полю selectors структуры SA за одним отличием – селекторы для политики безопасности могут включать несколько наборов селекторов;

name дополнительный селектор, задающий список идентификаторов. В RFC 4301 определены четыре типа идентификаторов: полное имя пользователя электронной почты, полное имя DNS, различительное имя X.500, строка байтов; данный селектор используется в том случае, когда возможно установить пользователя, с чьими привилегиями исполняется процесс, порождающий защищаемые данные;

pfp флаги PFP – по одному на каждый селектор трафика. При динамическом создании контекста безопасности для данной политики каждый флаг для соответствующего селектора трафика указывает, нужно ли брать значение селектора из соответствующего поля в пакете, который инициировал создание контекста безопасности, или из значения в соответствующей политике. Заданный флаг применяется к соответствующему селектору во всех наборах селекторов, содержащихся в данной политике.

action Действие, которое требуется применить к пакету: защитить, пропустить без защиты или отбросить. Одно действие выполняется для всех наборов селекторов.

direction Направление потоков данных, для которого предназначена данная политика: входящий или исходящий

SA_spec Если в качестве требуемой обработки указывается защита, то этот элемент содержит данные, в основном повторяющие структуру контекста безопасности SA. В элементе, определяющем алгоритмы, задаются только их идентификаторы (без ключей и т.п.). При этом могут задаваться несколько однотипных алгоритмов (например для шифрования), упорядоченные по предпочтительности

использования. При согласовании с помощью протокола IKE нового контекста безопасности удаленный узел выберет из этого набора приемлемый для него алгоритм.

Модель контекста безопасности IKE представлена абстрактным типом **IKE_SA** со следующими полями:

spi индекс параметров безопасности, используется для поиска контекста безопасности IKE;

rID идентификатор партнера;

i_messageID следующий идентификатор сообщения, который должен использоваться для инициируемого данным узлом запроса;

r_messageID идентификатор сообщения, который предполагается увидеть в следующем запросе от партнера;

request_window размер окна партнера для перекрывающихся запросов;

child_ipsec_sa список ссылок на дочерние контексты безопасности IPsec;

request_list список отправленных запросов, на которые еще не получен ответ, используется для повторной передачи, если в течение заданного времени ответ не получен;

response_list список отправленных ответов, используется для повторной передачи в случае потери пакетов;

dhg группа Диффи-Хеллмана;

auth алгоритм защиты целостности;

enc алгоритм шифрования;

prf псевдослучайная функция;

keys общие ключи контекста безопасности IKE. Правила вычисления этих ключей приведены в RFC 4306;

NATDetection блок данных для пересечения NAT.

Помимо моделей данных, необходимых для описания поведения протокола IKEv2, в спецификацию включены модели состояний служебных и транспортных протоколов: UDP и ICMPv6.

Модель блока данных авторизации партнера представлена абстрактным типом **PA** со следующими полями:

selectors идентификаторы партнера, которые согласуются с типами символических имен и IP-адресами, используемыми для идентификации элементов SPD;

auth_protocol протокол для аутентификации каждого партнера;

auth_method метод для аутентификации каждого партнера;

auth_data аутентификационные данные для каждого партнера;

peer_gateway информация о местоположении шлюза партнера (используется для партнеров, о которых известно, что они находятся "за" защитным шлюзом).

Для работы с модельными структурами данных SAD, SPD, PAD реализован набор вспомогательных функций: добавление, удаление, поиск записей и другие.

3.2. Модель пакета

Для модельного представления пакетов использовалась разработанная библиотека полиморфных типов. Иерархия типов отражает устройство пакетов IPv4/IPv6, в которых заголовки протоколов разного уровня организуются в односвязный список. Полиморфизм типов позволяет для моделирования различных пакетов использовать единую структуру. Разработаны типы для преставления кадров Ethernet, пакетов IPv4 и IPv6, дополнительных заголовков IPv6, заголовков IPsec v2, сообщений ICMPv6, пакетов UDP и IKEv2.

3.3. Формальный интерфейс управления состоянием IPsec

Добавление и удаление записей в базе контекстов безопасности представлены тремя элементами формального интерфейса: append_SAD_spec для добавления новых записей, remove_SA_SAD_spec для удаления контекстов безопасности, и flush_spec для очистки базы. Спецификационные функции append_SPD_spec и remove_SP_SPD_spec предназначены для добавления и удаления политик безопасности SP. Функция flush_spec моделирует сброс баз данных SAD и SPD.

Для данных спецификационных функций разработаны медиаторные функции, которые реализуют соответствующие воздействия для конкретной целевой системы, и осуществляют трансляцию содержимого SAD и SPD из реализационного вида в модельный. Медиаторные функции существенно зависят от реализации IPsec, поскольку реализация может поддерживать не все элементы определенные для SA и SP. В пост-условии этих функций проверяется соответствие полученного после воздействия состояния SAD, SPD с их ожидаемым состоянием.

3.4. Формальный интерфейс протоколов AH и ESP

Спецификационная функция receive_IpsecPacket моделирует обработку входящих пакетов в соответствии с требованиями, определенными в RFC 4301.

Краткое описание спецификации. Если пакет содержит IPsec заголовок, то ищется соответствующий контекст безопасности SA, сравниваются алгоритмы и ключи пакета и SA, сравниваются селекторы (заметим, что здесь нет проверки целостности пакета или расшифровки шифротекста). После прохождения всех проверок из пакета удаляется обработанный IPsec

заголовок. Если пакет не содержит IPsec заголовков, то ищется соответствующая политика безопасности SP, сравниваются селекторы. На данном этапе принимается решение пропустить пакет или отбросить. Если выносится решение его пропустить, то выбирается сокет, который может получить этот пакет (в соответствии с адресом и портом), и пакет помещается к нему в очередь. Если пакет на каком-то этапе отвергается, или нет подходящего для его приема сокета, то он не помещается ни в какую очередь. Таким образом предполагается, что если пакет должен быть принят системой, он будет сохранен в очереди какого-либо сокета, иначе - просто отброшен.

Входящий пакет представлен в модельном виде как параметр спецификационной функции.

Спецификационная функция send_IpsecPacket моделирует обработку исходящих данных в IPsec v2 в соответствии с требованиями, определенными в RFC 4301.

Краткое описание спецификации. Исходящее сообщение считается корректным, если существует политика безопасности и контекст безопасности такие, что в результате обработки поданного стимула, такого, как отправка сообщения по протоколу UDP, получится сообщение, соответствующее отправленному. Данная спецификация является недетерминированной, так как точное значение шифротекста предсказать нельзя в силу включения в защищаемые данные случайного вектора.

Построение модельного исходящего пакета выполняется следующим образом. Выбирается политика безопасности, соответствующая данному пакету. На данном этапе принимается решение пропустить пакет, защитить с помощью IPsec или отбросить. Если требуется защитить пакет с помощью IPsec, то выбирается нужный контекст безопасности, в случае необходимости используется спецификация IKE v2. Если контекст безопасности не найден, фиксируется ошибка. Согласно найденному контексту безопасности в пакет добавляется модельный IPsec заголовок, в котором вместо аутентификационных данных и криптотекста присутствуют идентификаторы алгоритмов и ключи. Далее по адресу и порту из пакета выбирается сокет, который может отправить этот пакет, и пакет помещается к нему в очередь. Если пакет на каком-то этапе отвергается, или нет подходящего для его отправления сокета, то он не помещается ни в какую очередь. Таким образом предполагается, что если пакет должен быть отправлен системой, он будет сохранен в очереди какого-либо сокета, иначе - просто отброшен. В пост-условии проверяется, что если пакет отправлен, то он добавлен только в одну очередь, если пакет отброшен, то размер всех очередей не изменился.

Отправленный пакет IPsec представлен как возвращаемое значение спецификационной функции.

3.5. Формальный интерфейс протокола IKE v2

Формальный интерфейс протокола IKE v2 включает два элемента: receive_IkeMessage и send_IkeMessage. Первая спецификационная функция моделирует отправку сообщения протокола IKEv2 удаленному агенту, а вторая функция моделирует получение реализацией сообщения этого протокола.

Не смотря на то, что протокол IKE v2 использует UDP для доставки сообщений, спецификация UDP не входит в формальный интерфейс. Подуровень UDP реализуется внутренними функциями реконструкции состояния формального интерфейса IPsec.

Краткое описание спецификации. Постусловия формального интерфейса формализует требования RFC 4306 к обменам сообщениями протокола IKEv2:

IKE_SA_INIT – согласование параметров криптографических протоколов и обмен данными для генерации временных ключей по схеме Диффи-Хеллмана. Поддерживается ограниченное число алгоритмов: цифровая подпись по алгоритмам MD5 и SHA, шифрование алгоритмом 3DES.

IKE_AUTH – взаимная аутентификация и создание управляющего контекста безопасности, защищающего последующие обмены сообщениями IKE. В тестовом наборе поддерживается только один вид аутентификации – по общему секрету.

CHILD_SA – обмен сообщениями для создания дочернего контекста безопасности, обеспечивающего защиту транспортных протоколов на сетевом уровне.

Сообщения IKE в формальном интерфейсе представлены в открытом виде. Шифрование и цифровая подпись обменов реализуется медиаторами при отправке сообщения.

Структуры данных для моделирования сообщений IKE v2 основываются на форматах сообщений из RFC 4306.

4. Тестирование реализаций IPsec

4.1. Устройство тестового стенда

В состав тестового стенда входят инструментальный узел и целевой узел. На инструментальном узле исполняется основной поток управления тестовой системы. На целевом узле функционирует тестируемая реализация. Для целей тестирования на целевой узел устанавливаются тестовые агенты, которые предоставляют средства удалённого доступа к служебным функциям и протоколам верхнего уровня (UDP). Инструментальный узел и целевой узел находятся в одном сегменте локальной сети. На рис. 1 представлено распределение компонентов тестовой системы по тестовому стенду.

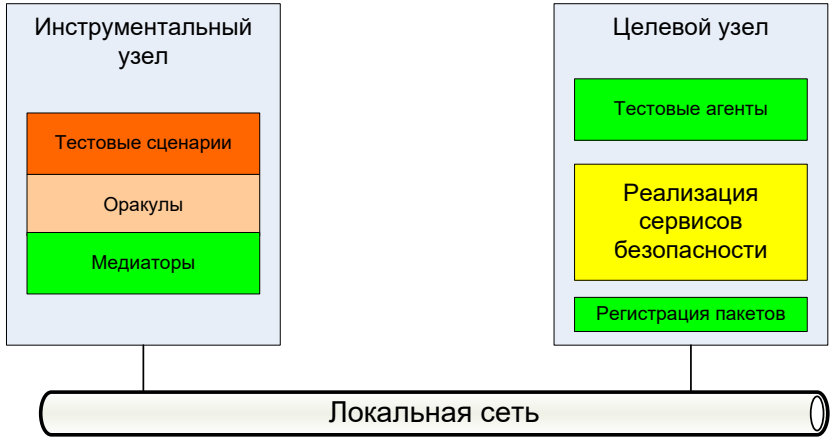


Рис. 1. Распределение компонентов тестовой системы в тестовом стенде.

В состав тестового набора входят следующие агенты:

- Агент для доступа к состоянию IPsec.
- Агент для доступа к состоянию IPv6.
- Агент для регистрации кадров канального уровня.
- Агент для управления сокетами UDP.
- Агент для управления реализацией протокола IKEv2.

Каждый агент предоставляет программный интерфейс для локальных вызовов. Для каждого агента разработан интерфейс удалённых вызовов посредством ONC RPC [16,17].

Агент для доступа к состоянию IPsec. Предоставляет тестовой системе средства для чтения и изменения настроек IPsec, прежде всего, БД политик безопасности и БД контекстов безопасности. Изменения заключаются в поддержке новых структур данных для представления контекстов безопасности и политик безопасности из IPsec v2.

Агент реализован для двух семейств операционных систем: FreeBSD 4.x и 5.x и OpenBSD. Реализации существенно зависят от платформы, так как нет стандартного программного интерфейса для операций с политиками безопасности и контекстами безопасности.

Реализации агента позволяют читать, добавлять и удалять политики безопасности, создавать, добавлять и удалять контексты безопасности.

Тестовый агент спроектирован таким образом, чтобы программный интерфейс агента не зависел от особенностей реализации. Такое решение обеспечивает переносимость удалённых вызовов агента на разных платформах.

Агент для доступа к состоянию IPv6. Предоставляет тестовой системе средства для чтения и изменения настроек протокола IPv6 на узле сети. На данный момент агент предоставляет следующие возможности:

- Получить список сетевых интерфейсов, установленных на узле,

Для любого сетевого интерфейса:

- Читать, добавлять, удалять адреса IPv6,
- Читать групповые (multicast) адреса IPv6,
- Читать адреса нижнего уровня (MAC адреса)
- Агент реализован для операционных систем FreeBSD 4.x и 5.x.

Реализация агента существенно зависит от платформы, так как нет стандартного программного интерфейса для операций с настройками IPv6.

Тестовый агент спроектирован таким образом, чтобы программный интерфейс агента не зависел от особенностей реализации. Такое решение обеспечивает переносимость удалённых вызовов агента на разных платформах.

Агент для регистрации кадров канального уровня. Обеспечивает регистрацию исходящих сообщений некоторого сетевого узла. Тестовая система использует агент для регистрации исходящих сообщений целевой реализации.

Агент реализован для операционных систем FreeBSD 4.x и 5.x, Linux с ядром версии 2.2 или 2.4. Существующая реализация агента использует фильтр пакетов BPF (Berkley Packet Filter). Реализации BPF есть на многих операционных системах, включая Microsoft Windows, поэтому предполагается, что агент может переноситься с минимальными изменениями на различные целевые платформы.

Также разработана реализация агента, использующая программный интерфейс PCAP.

Агент для управления сокетами UDP. Предоставляет тестовой системе средства для регистрации входящих сообщений UDP. Кроме того, интерфейс удалённого вызова агента предоставляет следующие средства:

- Удалённое создание, привязывание (bind) и удаление сокетов UDP.
- Удалённую отправка сообщений через сокет UDP.
- Удалённое чтение и изменение опций сокетов.

Агент реализован на операционных системах FreeBSD 4.x и 5.x, OpenBSD 3.x. Агент реализован в рамках POSIX, поэтому должен быть переносим на другие

операционные системы, поддерживающие программный интерфейс сокетов и потоков POSIX.

Агент для управления реализацией протокола IKEv2. Позволяет изменять настройки модуля IKEv2 и перегружать его.

Для корректного выполнения тестов необходимо, чтобы сетевое окружение, в котором выполняется тестирование, удовлетворяло ряду требований. В частности предполагается, что пакеты не теряются в сети (если пакет отправлен, он обязательно дойдет до адресата). Кроме того, предполагается, что пакеты не теряются внутри реализации. Для входящих пакетов это означает, что если пакет получен и прошел IPsec обработку, он будет доставлен в UDP сокет. Для исходящих пакетов: если политика безопасности разрешает такой трафик, пакет будет сформирован (возможно защищен IPsec) и отправлен в сеть.

Для обеспечения первого предположения в рамках данной работы был создан виртуальный тестовый стенд, так как исполнение тестов на виртуальных машинах имеет ряд преимуществ по сравнению с запуском тестов в реальном физическом окружении:

- Полный контроль над составом узлов в локальных сетях тестового стенда.
- Возможность создания идентичных копий тестовых стендов для проведения испытаний тестового набора разными участниками проекта.
- Возможность гибкой конфигурации состава тестового стенда путем добавления или удаления виртуальных машин.
- Экономия пространства, в частности, все средства ввода-вывода располагаются на одном физическом устройстве.

Виртуальный тестовый стенд включает в себя несколько виртуальных машин, объединенных в виртуальную локальную сеть.

Благодаря использованию виртуальных локальных сетей и виртуальных машин в тестовом стенде обеспечивается полный контроль над потоками данных. Все информационные потоки симулируются тестовой системой. В качестве виртуальной среды использовался продукт VMware Workstation 5.

4.2. Тестирование обработки исходящего трафика

В тестовом сценарии формируется модельное представление поступившего пакета, который в соответствии с ожидаемым поведением целевой системы, сохраняется (или не сохраняется) в очереди соответствующего модельного сокета. Модельный пакет преобразуется в сетевой пакет и отправляется тестируемой системе. Инструментальный узел запрашивает у тестового агента собранные пакеты, проверяет, есть ли они в модельных сокетах, и выносит

вердикт на основании логических предикатов, содержащихся в постусловиях спецификационных функций.

Первоначальный UDP-пакет с помощью дополнительного агента передается на целевой узел и отправляется через заданный сокет. При этом в сеть выходит пакет, уже прошедший IPsec обработку на целевой системе.

На целевом узле создаются UDP сокеты, привязанные к конкретным адресам имеющихся интерфейсов.

В тестовом сценарии формируется модельное представление исходящего UDP-пакета, который в соответствии с политиками безопасности проходит IPsec обработку, а затем или отбрасывается, или сохраняется в очереди соответствующего модельного сокета. Модельный UDP-пакет преобразуется в промежуточный формат и передается тестовому агенту, который через указанный сокет отправляет пакеты в сеть. При этом перед отправкой в сеть пакеты проходят IPsec обработку на целевом узле. На инструментальном узле кетчер собирает все пришедшие с целевого узла пакеты, переводит их в модельное представление с сохранением всех заголовков. После этого проверяется, есть ли такой пакет в очереди модельного сокета, и выносится вердикт.

4.3. Тестирование обработки входящего трафика

В сценарии формируется модельный пакет, который будет рассматриваться как входящий. IPsec заголовки этого пакета, если они присутствуют, содержат всю информацию, необходимую для обработки пакета (SPI, Sequence Number и т.д.), но не содержат криптографические данные (аутентификационные последовательности и шифротекст), вместо них включаются идентификаторы криптографических алгоритмов и ключи.

Затем с помощью генератора сетевых пакетов из модельного пакета создается сетевой пакет и он отправляется в сеть.

Открытые на тестируемом узле сокеты являются UDP-сокетами. Поэтому пакет, полученный таким сокетом, уже прошел IPsec обработку, т.е. для него либо был найден контекст безопасности, проверена целостность пакета, выполнена расшифровка пакета и другие проверки, либо была найдена политика безопасности с разрешением его пропустить. Таким образом вся IPsec обработка сетевого пакета перекладывается на тестируемую систему. Если какой-то сокет получил пакет, значит он был парвильно сформирован и есть соответствующая разрешающая политика, если ни один сокет неполучил пакет, значит он был отвергнут. Задача формирования сетевого пакета решается генератором сетевых пакетов, который по модельному представлению создает необходимые заголовки, а по заданным алгоритмам вычисляет аутентификационные последовательности и создает шифротекст.

Сборщик реакций тестовой системы периодически опрашивает тестового агента, и забирает все пакеты, полученные на данный момент сокетами.

Каждый полученный тестовым агентом пакет регистрируется как реакция целевой системы.

В ходе вынесения вердикта постусловие спецификационной функции реакции IpsecRecvInput вызывается для каждого такого пакета. По IP адресу и порту назначения пакета производится поиск соответствующего сокета. Если сокет не найден, фиксируется ошибка. Если сокет найден, данный пакет удаляется из его очереди. Если у сокета нет такого пакета, фиксируется ошибка.

В пост-условии проверяется, что пакет был удален из очереди только одного сокета, очереди других сокетов не изменились. После завершения всех тестовых воздействий, очереди всех модельных сокетов должны быть пусты.

4.4. Тестирование IKEv2

Стимулами в разработанном тестовом наборе являются сообщения от инструментального узла или сообщения, отправленные через UDP сокет целевого узла, с помощью агента управления сокетами, а реакциями – сообщения со стороны тестируемого узла. Основная часть требований спецификации IKEv2 проверяется в постусловиях реакций.

Разработанный тестовый набор покрывает большую часть требований спецификации [4], в том числе обеспечивает проверку правильности начальных обменов, обменов CREATE_CHILD_SA, информационных обменов, а также требований к удалению контекстов безопасности, размеру сообщений, использованию порядковых номеров, размеру окна для перекрывающихся запросов, номерам версий и совместимости, идентифицирующим цепочкам и общим требованиям к передаче сообщений IKE v2.

В тестовом стенде задействованы два узла – инструментальный узел, на котором исполняются тестовые сценарии, и целевой узел – узел, на котором работает тестируемая реализация IKE v2.

На целевом узле размещаются два тестовых агента. Один - по запросу тестовой системы отсылает сообщения через открытые UDP сокеты, привязанные к конкретным адресам имеющихся интерфейсов. Другой - регистрирует исходящие сообщения, и по запросу передает инструментальному узлу.

В начале каждого теста устанавливаются необходимые политики безопасности (в том числе разрешающие IKE трафик), и после каждого воздействия проверяется, что они не изменились.

В начале каждого теста агентом управления подсистемой IKE загружаются необходимые для данного теста настройки IKE.

4.4.1. Тестирование в режиме ответчика

В режиме ответчика реализация не генерирует запросы, а лишь поддерживает информационный обмен, инициированный другим узлом. Стимулами являются сообщения от инструментального узла.

В тестовом сценарии формируется запрос в модельном представлении, который передается спецификационной функции отправки пакетов. Если сообщение является запросом IKE_SA_INIT, создается новый контекст безопасности IKE SA. В предусловии этой функции проверяется правильность структуры тестового сообщения и его своевременность, и на основании этого делается вывод о том, должен ли на него быть ответ, или реализация должна его просто отбросить. Сообщение сохраняется в очереди запросов. В блоке медиатора из модельного представления тестового сообщения строится реализационное, которое и отправляется в сеть.

Сборщик реакций периодически опрашивает тестового агента, и забирает все пакеты, отправленные на данный момент целевой системой. Время ожидания реакции целевой системы должно устанавливаться для каждого сценария отдельно, так как некоторые ответные сообщения (например, запрос COOKIE) требуют реакции инструментального узла в течение ограниченного времени. Каждый полученный тестовым агентом пакет регистрируется как реакция целевой системы. Сборщик реакций строит из реализационного пакета его модельное представление.

В постусловии реакции сообщение проверяется на соответствие требованиям спецификации. Проверка разделена на несколько стадий. Сначала проверяется допустимость такого сообщения от реализации для текущего обмена, соответствие существующему IKE SA, соответствие ранее отправленному запросу. Далее проверяется структура самого сообщения (присутствующие блоки данных должны соответствовать данному обмену). Наконец разбирается структура каждого блока данных и проверяются значения всех его полей. После завершения проверки, соответствующий этому сообщению запрос удаляется из очереди запросов.

После проверки всех требований, сообщение передается тестовому сценарию, где в зависимости от плана сценария, принимается решение о продолжении или завершении информационного обмена. Если выявлено нарушение требований, то принимается решение, является ли это нарушение критичным для продолжения обмена. В случае продолжения обмена, формируется следующий запрос.

При отсутствии реакции реализации в течение установленного времени обмен считается завершенным.

После завершения всех воздействий, проверяется очередь запросов. Она должна быть пуста или содержать запросы, на которые не ожидался ответ.

4.4.2. Тестирование в режиме инициатора

В режиме инициатора реализация сама формирует запросы. Стимулами являются уже не сообщения от другого узла, а UDP пакеты, отправляемые агентом на тестируемом узле, через локальные UDP-сокеты. Эти пакеты перехватываются подсистемой обработки IPsec, которая передает реализации IKE запрос на создание контекстов безопасности IPsec.

В тестовом сценарии формируется модельное представление UDP пакета, который передается агенту на целевой узел. В очередь ожидаемых запросов добавляется тип запроса. Тестовый агент на целевом узле отправляет этот пакет через открытый сокет. При наличии политики безопасности, требующей IPsec обработки такого пакета, инициируется IKE обмен.

Сборщик реакций тестовой системы периодически опрашивает другого тестового агента, и забирает все пакеты, отправленные на данный момент целевой системой. Для данного режима тестирования следует учитывать время ожидания и сбора реакций, поскольку согласно спецификации, если на запрос не получен ответ в течение некоторого времени, реализация IKE должна повторно передать запрос или отказаться от соединения. Каждый полученный тестовым агентом пакет регистрируется как реакция целевой системы.

В постусловии реакции сообщение проверяется на соответствие требованиям спецификации. Проверка разделена на несколько стадий. Сначала проверяется допустимость такого сообщения от реализации для текущего обмена, соответствие существующему IKE SA (если сообщение не является запросом IKE_SA_INIT). Далее проверяется структура самого сообщения (присутствующие блоки данных должны соответствовать данному обмену). Наконец разбирается структура каждого блока данных и проверяются значения всех его полей.

Из очереди ожидаемых запросов удаляется соответствующий элемент.

После проверки всех требований, сообщение передается тестовому сценарию, где в зависимости от плана сценария, принимается решение о необходимости ответа, задержки ответа на некоторое время или завершении информационного обмена. Если выявлено нарушение требований, то принимается решение, является ли это нарушение критичным для продолжения обмена.

Если сообщение является запросом IKE_SA_INIT, создается новый контекст безопасности IKE SA.

Если принято решение отправить ответ, то формируется модельное представление сообщения. Сообщение передается спецификационной функции отправки пакетов. В предусловии этой функции проверяется правильность структуры тестового сообщения и его допустимость в текущем обмене. В очередь ожидаемых запросов добавляется тип запроса, если он

предполагается. В блоке медиатора из модельного представления тестового сообщения строится реализационное, которое отправляется в сеть.

При отсутствии реакции реализации в течение установленного времени обмен считается завершенным. После завершения всех воздействий очередь ожидаемых запросов должна быть пуста.

5. Результаты тестирования реализаций IPsec v2

5.1. Тестирование реализаций оконечных узлов и шлюзов безопасности

Тестирование реализаций оконечных узлов и шлюзов безопасности проводилось для операционной системы FreBSD 6.2. Эта реализация была выбрана по следующим причинам:

- Наличие стабильной поддержки IPsec v2. Для сравнения, на момент тестирования (2008-09 гг.) реализации IPsec для ОС Linux, Solaris не поддерживали вторую версию архитектуры IPsec. Реализация IPsec для остальных ОС семейства BSD имеют много общего с реализацией для FreeBSD, поэтому из всего семейства была выбрана наиболее разработанная и поддерживаемая ОС.
- Наличие исходных текстов утилит администрированием IPsec. Несмотря на отсутствие документации по программным интерфейсам реализации IPsec удалось реализовать удаленных агентов на основе штатных утилит управления IPsec в операционной системе. На проприетарных операционных системах, таких как MS Windows и Cisco IOS, такой возможности нет.

В ходе тестирования был обнаружен ряд отклонений от требований IPsec:

1. Неполная поддержка селекторов IPsec. Отсутствуют селекторы по портам TCP / UDP сокетов, именам процессов и некоторым другим.
2. Часть комбинаций туннелей не поддерживается. В частности, не поддерживаются вложения AH туннелей в ESP туннели.

На сообщения об обнаруженных ошибках разработчики IPsec для FreeBSD ответили, что в настоящее время IPsec в FreeBSD поддерживает только те функции, которые необходимы для функционирования IKEv2, поэтому обнаруженные ошибки не являются критичными.

5.2. Тестирование реализаций IKE v2

На момент проведения тестирования существовали четыре открытых реализации протокола IKEv2: OpenIKEv2, StrongSwan, IKEv2, Racoon2. Для

тестирования была выбрана реализация Racoon2-20090327c для операционной системы FreeBSD 6.2, что позволило использовать ряд компонентов, разработанных в рамках предыдущих проектов.

При выполнении тестового набора был выявлен ряд нарушений требований RFC 4306 и ошибок реализации:

- реализация не поддерживает диапазоны адресов и портов;
- реализация поддерживает префиксы для адресов, но не корректно с ними работает (например, когда на одном узле заданы адреса с префиксами, а на другом - точные);
- реализация игнорирует порядок заголовков в сообщениях;
- реализация не проверяет некоторые поля заголовков (во входящем запросе IKE_AUTH поле 'Exchange Type', во входящее ответе IKE_AUTH поле 'Responder's SPI');
- во входящем запросе REKEY IPsec SA не проверяется новый SPI (при совпадении с текущим, обмен успешно завершается, а IPsec SA в ядре ОС не создается);
- во входящих сообщениях обмена IKE_AUTH в блоке данных SA не проверяется ни количество Предложений, ни количество Преобразований. Если предложенный набор шире, чем требуется, он все равно может быть принят.
- при формировании запроса, содержащем в блоке данных SA несколько Предложений, начиная со второго Предложения поле SPI устанавливается 0;
- первый байт структур Предложение и Преобразование определен как не обязательный для обработки. Тем не менее, реализация опирается на него при разборе блока данных SA, и не правильные значения считает ошибкой.
- если в последней структуре Предложение первый байт ‘last’ выставлен не правильно (значение 2 вместо 0), реализация не проверяет, что блок данных SA закончился (исходя из поля ‘длина заголовка’ SA), и рассматривает следующую часть сообщения как еще одно Предложение, что приводит в лучшем к отбрасыванию пакета из-за не правильной структуры. При выполнении одного из тестов, произошло зацикливание реализации и потеря ее работоспособности, а также быстрое увеличение размера файла журнала.
- файл конфигурации racoon позволяет использовать опции esp_enc_alg и esp_auth_alg (предназначенные для ESP) для протокола AH. Причем эти опции установлены в настройках по умолчанию. При этом в обмене IKE_AUTH в структуре Предложение для протокола AH передаются преобразования Шифрование и Аутентификация. Если на другом узле такие же настройки, то обмен завершается созданием IPsec SA AH (правда только с алгоритмом аутентификации, алгоритм шифрования игнорируется ядром ОС).

Следует отметить, что в целом реализация соответствует спецификации с некоторыми ограничениями функциональности, такими как невозможность сужения селекторов трафика, не поддерживаемый протокол аутентификации EAP, невозможность одновременной обработки нескольких запросов.

Результаты тестирования доступны на сайте ipv6.ispras.ru.

6. Заключение

В ходе выполнения работы были выделены требования к реализациям IPsec v2, разработаны формальные спецификации и прототип тестового набора для верификации реализаций IPsec v2, в том числе реализаций протокола автоматического создания контекстов безопасности IKEv2. В статье описаны метод формализации требований IPsec v2, процесс создания тестового набора, а также результаты тестирования существующих реализаций.

Применение для решения задачи автоматизации тестирования таких традиционных для телекоммуникаций средств моделирования протоколов, как расширенные конечные автоматы и системы помеченных переходов (LTS), сталкивается с трудностями принципиального характера, обусловленными высокой сложностью протоколов Интернета.

Верификация функций безопасности протокола нового поколения IPsec v2 показала, что предложенный в данной работе метод верификации, основанный на контрактных спецификациях, позволяет эффективно автоматизировать тестирование таких сложных протоколов, как протоколы безопасности. При этом тестовые наборы обладают формально определенным и прослеживаемым покрытием требований, что в значительной степени улучшает качество тестирования.

Литература

[1] RFC4301 S. Kent, K. Seo. Security Architecture for the Internet Protocol December 2005

[2] RFC4302 S. Kent. IP Authentication Header S. Kent December

[3] RFC4303 S. Kent. IP Encapsulating Security Payload (ESP) December 2005

[4] RFC4306 C. Kaufman, Ed. Internet Key Exchange (IKEv2) Protocol December 2005

[5] RFC4307 J. Schiller. Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2) December 2005

[6] RFC4807 M. Baer, R. Charlet, W. Hardaker, R. Story, C. Wang. IPsec Security Policy Database Configuration MIB March 2007

[7] RFC4868 S. Kelly, S. Frankel. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec May 2007

[8] Bourdonov, I., Kossatchev, A., Kuliamin, V., Petrenko, A. UniTesK Test Suite Architecture // Proceedings of FME, LNCS 2391. Springer-Verlag, 2002. P. 77-88.

[9] CTesK 2.1: SeC Language Reference. М.: ИСП РАН, 2005. 167 с.

[10] Н.В. Пакулин. Формализация стандартов и тестовых наборов протоколов Интернета. Автореферат диссертации на соискание учёной степени кандидата физико-математических наук. Москва, 2006.

[11] Н.В. Пакулин, А.В. Хорошилов "Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов", Журнал "Программирование" № 5, 2007 г., ISSN 0132-3474, с. 1-29.

[12] IETF RFC 2223. J. Postel, J. Reynolds. Instructions to RFC Authors. IETF, 1997. 20 с.

[13] IETF BCP 14 | IETF RFC 2119. S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. IETF, 1997. 3 с.

[14] IETF RFC 1213. K. McCloghrie, M. T. Rose. Management Information Base for Network Management of TCP/IP-based internets: MIB-II. March 1991.

[15] IETF RFC 2011 K. McCloghrie, Ed. SNMPv2 Management Information Base for the Internet Protocol using SMIv2. November 1996.

[16] IETF RFC 1831. R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2, 1995.

[17] IETF RFC 1832. R. Srinivasan, XDR: External Data Representation Standard, 1995.

Тестирование конформности на основе соответствия состояний

Бурдонов И.Б., Косачев А.С.
{igor,kos}@ispras.ru

Аннотация. Статья посвящена тестированию соответствия (конформности) реализации требованиям спецификации. Идея безопасного тестирования, предложенная авторами для конформности, основанной на трассах наблюдений, распространяется на случай (слабой) симуляции – конформности, основанной на соответствии состояний реализации и спецификации. Строится теория безопасной симуляции и ее тестирования. Предлагаются общий алгоритм полного тестирования и его модификация для практического применения, опирающаяся на некоторые ограничения на реализацию и спецификацию.

1. Введение

Тестирование конформности – это проверка в процессе эксперимента соответствия (конформности) реализации требованиям, заданным в виде спецификации. Это соответствие определяется семантикой тестового взаимодействия, которая описывает возможные тестовые воздействия и возможные наблюдения ответного поведения реализации.

Если при тестировании мы можем наблюдать только (внешние) действия, выполняемые реализацией в ответ на тестовые воздействия или отсутствие таких действий (отказ), то конформность определяется на основе трасс наблюдений, то есть последовательностей действий и отказов. Спецификация в этом случае описывает те трассы, которые допускаются в реализации.

Тестирование, при котором гарантировано конечное время ожидания наблюдения после тестового воздействия, называется безопасным. Возможны две причины бесконечного ожидания: дивергенция и ненаблюдаемые отказы. Дивергенция – это бесконечное выполнение реализацией внутренних (ненаблюдаемых) действий. Ненаблюдаемый отказ – это отсутствие выполняемых реализацией внешних действий, которое тест не может определить за конечное время[1]. В обоих случаях тест не может ни продолжить

[1] Например, интервал времени между тестовым воздействием и ответным внешним действием ограничен некоторым тайм-аутом. Превышение тайм-

тестирование, ни закончить его, так как неизвестно, нужно ли ждать наблюдения или никакого наблюдения не будет. Тестирование, при котором не возникает дивергенции и ненаблюдаемых отказов, называется безопасным.

Кроме этого возможно специальное, не регулируемое тестовыми воздействиями, действие реализации, которое называется *разрушением*. Оно моделирует любое нежелательное поведение системы, в том числе и ее реальное разрушение. Семантика разрушения предполагает, что оно также не должно возникать при безопасном тестировании.

Спецификация описывает те ситуации, при которых тестовое воздействие должно быть безопасно в реализации. Соответственно, конформность основана только на безопасном поведении реализации. Конформность, учитывающая безопасность, введена авторами этой статьи в [1,2,3,5].

Кроме конформности, основанной только на трассах наблюдений, в литературе рассматриваются разные виды конформностей, основанных на соответствии состояний реализации и спецификации (обзор см. в [6]). Такие конформности называются симуляциями. Симуляция требует, чтобы правильным было не только наблюдаемое внешнее поведение реализации, но и изменение ее состояний. Все рассматриваемые в литературе симуляции либо не учитывают безопасности тестирования, предполагая отсутствие дивергенции и ненаблюдаемых отказов, либо предполагают возможность прямого наблюдения дивергенции и всех отказов. Также они не учитывают возможность разрушения.

В данной статье вводится симуляция, учитывающая безопасность, называемая безопасной симуляцией. Спецификация описывает не только класс конформных ей реализацией, но и гипотезу о безопасности, определяющую более широкий класс реализаций, которые можно безопасно тестировать для проверки конформности.

Выбор симуляции в качестве конформности наиболее естественен, когда состояния реализации доступны для их наблюдения. Тестирование, при котором в любой момент времени можно опросить текущее состояние реализации, называется тестированием с открытым состоянием. Задача тестирования – обнаружение ошибок в реализации, понимаемое как несоответствие ее поведения спецификационным требованиям. Тестирование полное, если обнаруживается любая ошибка и не фиксируются «ложные» ошибки. В данной статье рассматривается полное тестирование с открытой состоянием безопасной симуляции.

Это рассмотрение проводится как в общетеоретическом, так и в практическом планах. Теоретическое полное тестирование должно обнаруживать любую ошибку за конечное время, но при отсутствии ошибок может продолжаться бесконечно. Причины бесконечного тестирования – это бесконечность

аута при ожидании внешних действий означает наблюдение отказа. При отсутствии такого рода ограничений отказ ненаблюдаем.

реализации и/или спецификации, а также неограниченный недетерминизм поведения реализации. При некоторых ограничениях возможно построение полных тестов, в любом случае завершающих свою работу за конечное время. Такие тесты уже можно использовать на практике.

2-ой раздел статьи содержит основные положения теории конформности: семантика взаимодействия и безопасное тестирование, математическая модель реализации и спецификации, определение симуляции, гипотеза о безопасности и определение безопасной симуляции. В 3-ем разделе рассматривается связь безопасной симуляции с трассовой конформностью. 4-ый раздел посвящен теоретическому тестированию: определяется полнота тестирования, описывается общий алгоритм тестирования и вводится достаточное условие его полноты. 5-ый раздел посвящен практическому тестированию: определяются ограничения на реализацию и спецификацию, позволяющие так модифицировать общий алгоритм тестирования, чтобы он стал конечным и полным, а также предлагается более практический алгоритм, состоящий из алгоритма обхода реализации и последующей верификации симуляции, приводится пример верификации симуляции.

2. Теория конформности

2.1. Семантика взаимодействия и безопасное тестирование

Данная работа развивает теорию тестирования конформности, изложенную в [1,2,3,5]. Тестирование понимается как проверка в процессе эксперимента соответствия (конформности) реализации требованиям, заданным в виде спецификации.

Семантика тестового взаимодействия определяется в терминах *действий* и *кнопок*. Действие – это внешнее действие тестируемой системы (реализации), которое может наблюдаться при тестировании. Множество внешних действий называется алфавитом действий и обозначается $\mathbf{L}$. Кнопка – это подмножество $P \subseteq \mathbf{L}$; нажатие кнопки P моделирует тестовое воздействие на реализацию, сводящееся к разрешению выполнять любое действие из P. При нажатии кнопки P наблюдается либо действие $a \in P$, выполняемое реализацией, либо (для некоторых кнопок) отсутствие таких действий, называемое отказом P. Семантика взаимодействия задается алфавитом $\mathbf{L}$ и двумя наборами кнопок: с наблюдением соответствующих отказов – семейство $\mathbf{R} \subseteq \mathcal{P}(\mathbf{L})$ и без наблюдения отказов – семейство $\mathbf{Q} \subseteq \mathcal{P}(\mathbf{L})$. Предполагается, что $\mathbf{R} \cap \mathbf{Q} = \varnothing$ и $\cup\mathbf{R} \cup \cup\mathbf{Q} = \mathbf{L}$. Такая семантика называется **R**/**Q**-семантикой.

При нажатии кнопки $Q \in \mathbf{Q}$, вообще говоря, неизвестно, нужно ли ждать наблюдения или никакого наблюдения не будет, поскольку возник ненаблюдаемый отказ. Тем самым, нельзя ни продолжить тестирование, ни

закончить его. Поэтому при правильном взаимодействии с реализацией, в том числе при тестировании, должна быть уверенность, что при нажатии кнопки $Q \in \mathbf{Q}$ не возникает отказа.

Кроме внешних действий реализация может совершать внутренние (ненаблюдаемые) действия, обозначаемые τ. Эти действия всегда разрешены. Предполагается, что любая конечная последовательность любых действий совершается за конечное время, а бесконечная – за бесконечное время. Бесконечная последовательность τ-действий («зацикливание») называется *дивергенцией* и обозначается Δ. Дивергенция сама по себе не опасна, но при попытке выхода из нее, когда нажимается любая кнопка $P \in \mathbf{R} \cup \mathbf{Q}$, неизвестно, нужно ли ждать наблюдения или бесконечно долго будут выполняться только внутренние действия. Эта ситуация аналогична возникновению ненаблюдаемого отказа при нажатии кнопки $Q \in \mathbf{Q}$: нельзя ни продолжить тестирование, ни закончить его. Поэтому при правильном взаимодействии с реализацией, в том числе при тестировании, следует избегать тестовых воздействий, если в реализации возникла дивергенция.

Кроме этого вводится специальное, также не регулируемое кнопками, действие, которое называется *разрушением* и обозначается γ. Оно моделирует любое нежелательное поведение системы, в том числе и ее реальное разрушение. Семантика разрушения предполагает, что оно не должно возникать при правильном взаимодействии с реализацией, в том числе при тестировании.

Взаимодействие с реализацией, в том числе тестирование, при котором не возникает ненаблюдаемых отказов, попыток выхода из дивергенции и разрушения, называется безопасным.

2.2. LTS-модель

В качестве модели реализации и спецификации используется *система помеченных переходов* (LTS – Labelled Transition System) – ориентированный граф с выделенной начальной вершиной, дуги которого помечены некоторыми символами. Формально, LTS – это совокупность $\mathbf{S}=\mathrm{LTS}(V_{\mathbf{S}}, \mathbf{L}, E_{\mathbf{S}}, s_0)$, где $V_{\mathbf{S}}$ – непустое множество состояний (вершин графа), $\mathbf{L}$ – алфавит внешних действий, $E_{\mathbf{S}} \subseteq V_{\mathbf{S}} \times (\mathbf{L} \cup \{\tau, \gamma\}) \times V_{\mathbf{S}}$ – множество переходов (помеченных дуг графа), $s_0 \in V_{\mathbf{S}}$ – начальное состояние (начальная вершина графа). Переход из состояния s в состояние s' по действию z обозначается $s \text{—} z \rightarrow s'$. Обозначим $s \text{—} z \rightarrow \triangleq \exists s'\ s \text{—} z \rightarrow s'$ и $s \text{—} z \not\rightarrow \triangleq \nexists s'\ s \text{—} z \rightarrow s'$. Маршрутом LTS называется последовательность смежных переходов: начало каждого перехода, кроме первого, совпадает с концом предыдущего перехода.

Выполнение LTS сводится к выполнению того или иного перехода, определенного в текущем состоянии (начиная с начального состояния) и разрешаемого нажатой кнопкой (τ- и γ-переходы всегда разрешены)[2].

Состояние s *дивергентно* (обозначается $s\uparrow$), если в нем начинается бесконечная цепочка τ-переходов (в частности, τ-цикл); в противном случае состояние *конвергентно* ($s\downarrow$). Состояние s *стабильно*, если из него не выходят τ- и γ-переходы: $s\text{—}\tau\not\rightarrow$ & $s\text{—}\gamma\not\rightarrow$. Отказ $P\in\mathbf{R}$ порождается стабильным состоянием, из которого нет переходов по действиям из P: $\forall z\in P\cup\{\tau,\gamma\}\ \ s\text{—}z\not\rightarrow$.

Для определения трасс LTS $\mathbf{s}$ добавим в каждом ее стабильном состоянии виртуальные петли $s\text{—}P\rightarrow s$, помеченные порождаемыми отказами, и Δ-переходы из дивергентных состояний $s\text{—}\Delta\rightarrow$. В полученной LTS рассмотрим маршруты, не продолжающиеся после Δ- и γ-переходов. Трассой назовем последовательность σ пометок на переходах такого маршрута с пропуском символов τ. Будем обозначать для $s,s`\in V_{\mathbf{s}}$, $u\in\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\gamma,\Delta\}$, $\sigma=\langle u_1,\ldots,u_n\rangle\in(\mathbf{L}\cup\mathbf{R}\cup\mathbf{Q}\cup\{\gamma,\Delta\})^*$:

$s\Rightarrow s` \quad \triangleq\ s=s` \vee \exists s_1,\ldots,s_n$

$s=s_1\text{—}\tau\rightarrow s_2\text{—}\tau\rightarrow\ldots\text{—}\tau\rightarrow s_n=s`,$

$s=\langle u\rangle\Rightarrow s` \quad \triangleq\ \exists s_1,s_2\ \ s\Rightarrow s_1\text{—}u\rightarrow s_2\Rightarrow s`,$

$s=\sigma\Rightarrow s` \quad \triangleq\ \exists s_1,\ldots,s_{n+1}\ \ s=s_1=\langle u_1\rangle\Rightarrow s_2\ldots s_n=\langle u_n\rangle\Rightarrow s_{n+1}=s`,$

$s=\sigma\Rightarrow \quad \triangleq\ \exists s`\ \ s=\sigma\Rightarrow s`,$

$s=\sigma\not\Rightarrow \quad \triangleq\ \neg(s=\sigma\Rightarrow),$

s ***after*** $\sigma\ \triangleq\ \{s`\mid s=\sigma\Rightarrow s`\}.$

Запись $s=\sigma\Rightarrow s`$ (или $s\Rightarrow s`$) понимается как наличие маршрута, начинающегося в состоянии s (*пресостоянии*), заканчивающегося в состоянии $s`$ (*постсостоянии*) и имеющего трассу σ (или пустую трассу $\langle\rangle$). **R**-трассой будем называть трассу, не содержащую отказов из **Q**, а **R**-маршрутом – маршрут с **R**-трассой. Множество **R**-трасс, начинающихся в состоянии s, обозначим $\boldsymbol{T}(s)$. По умолчанию, будем считать, что трасса

[2] При параллельной композиции двух LTS, моделирующей взаимодействие, нажатой кнопке для одной LTS соответствует состояние другой LTS, в котором определены переходы по всем действиям, разрешаемым этой кнопкой (при композиции в CCS это противоположные им действия), и только они.

начинается в начальном состоянии LTS: будем обозначать $\mathbf{s}$ ***after*** σ = s_0 ***after*** σ и $\boldsymbol{T}(\mathbf{s})=\boldsymbol{T}(s_0)$.

2.3. Слабая симуляция

В наших предыдущих работах [1,2,3,5] рассматривалась конформность, основанная только на трассах наблюдений и не учитывающая состояний реализации и спецификации. В то же время в литературе рассматриваются разные виды конформностей, основанные на соответствии R состояний реализации и спецификации[3]. Такие конформности называются симуляциями (обзор см. в [6]). Симуляция требует, чтобы каждое наблюдение u, возможное в реализационном состоянии i с постсостоянием $i`$, было возможно в каждом соответствующем ему спецификационном состоянии $s$, и в спецификации для $s$ и $u$ нашлось бы постсостояние $s`$, соответствующее $i`$. Разные симуляции отличаются друг от друга, главным образом, отношением к наблюдаемости внутренних действий (τ). В данной статье мы исходим из основного допущения о принципиальной ненаблюдаемости τ-действий: при тестировании мы не можем различать наличие или отсутствие τ-действий как до, так и после внешнего действия (при наблюдении отказа τ-действия возможны, очевидно, только до отказа). Этому соответствует слабая симуляция (weak simulation), называемая также наблюдаемой симуляцией (observation simulation). Мы дадим три эквивалентных определения слабой симуляции (0).

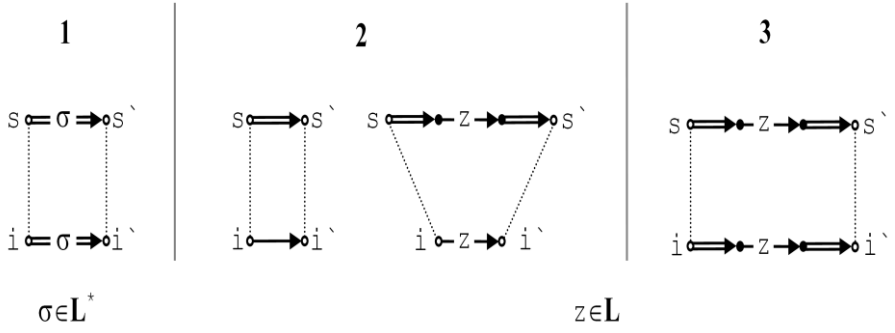


Рис. 1. Три определения слабой симуляции.

$\mathbf{I}\leq^{1}_{ws}\mathbf{S} \triangleq \exists R\subseteq V_{\mathbf{I}}\times V_{\mathbf{S}}\ (i_0,s_0)\in R\ \&\ \forall(i,s)\in R\ \forall\sigma\in\mathbf{L}^*\ \forall i`$

$(i=\sigma\Rightarrow i`\ \Rightarrow\ \exists s`\ s=\sigma\Rightarrow s`\ \&\ (i`,s`)\in R).$

$\mathbf{I}\leq^{2}_{ws}\mathbf{S} \triangleq \exists R\subseteq V_{\mathbf{I}}\times V_{\mathbf{S}}\ (i_0,s_0)\in R\ \&\ \forall(i,s)\in R\ \forall u\in\mathbf{L}\ \forall i`$

[3] Мнемоника: R – **R**elation.

$(i \longrightarrow\tau\rightarrow i` \Rightarrow \exists s` \; s\Rightarrow s` \;\&\; (i`,s`)\in R) \;\&$

$(i \longrightarrow u\rightarrow i` \Rightarrow \exists s` \; s=\langle u\rangle\Rightarrow s` \;\&\; (i`,s`)\in R).$

$\mathbf{I}\leq^{3}_{ws}\mathbf{S} \triangleq \exists R\subseteq V_{\mathbf{I}}\times V_{\mathbf{S}} \; (i_0,s_0)\in R \;\&\; \forall (i,s)\in R \; \forall u\in\mathbf{L} \; \forall i`$

$(i=\langle u\rangle\Rightarrow i` \Rightarrow \exists s` \; s=\langle u\rangle\Rightarrow s` \;\&\; (i`,s`)\in R).$

Соответствие R, для которого выполнены условия слабой симуляции $\leq^{k}_{ws}$, назовем $\leq^{k}_{ws}$-соответствием или просто *конформным соответствием*.

Первые два определения, принадлежащие Милнеру [9,10], эквивалентны.

Лемма 1: $\leq^{1}_{ws} = \leq^{2}_{ws}$.

Доказательство. Легко показать, что эти определения эквивалентны даже в более сильном смысле: если R является $\leq^{1}_{ws}$-соответствием, то оно же является $\leq^{2}_{ws}$-соответствием, и обратно.□

Лемма 2: $\leq^{1}_{ws} = \leq^{3}_{ws}$.

Доказательство. Поскольку трасса $\langle u\rangle\in\mathbf{L}^{*}$, любое $\leq^{1}_{ws}$-соответствие является $\leq^{3}_{ws}$-соответствием. Обратное, однако, не верно; зато верно следующее утверждение: из существования $\leq^{3}_{ws}$-соответствия следует существование, быть может, другого соответствия, которое является как $\leq^{3}_{ws}$-соответствием, так и $\leq^{1}_{ws}$-соответствием.

Пусть R – некоторое $\leq^{3}_{ws}$-соответствие. Пополним его каждой такой парой $(i`,s)$, для которой $(i,s)\in R$ и $i\Rightarrow i`$, и покажем, что полученное соответствие $R`$ также является $\leq^{3}_{ws}$-соответствием. Нам надо показать, что для добавленной пары $(i`,s)$ и любого маршрута $i`=\langle u\rangle\Rightarrow i``$, где $u\in\mathbf{L}$, существует $s``$ такое, что $s=\langle u\rangle\Rightarrow s``$ & $(i``,s``)\in R`$. Действительно, если $i\Rightarrow i`$ и $i`=\langle u\rangle\Rightarrow i``$, то $i=\langle u\rangle\Rightarrow i``$, а, поскольку $R$ является $\leq^{3}_{ws}$-соответствием для $\mathbf{I}$ и $\mathbf{S}$, то для $(i,s)\in R$ существует $s``$ такое, что $s=\langle u\rangle\Rightarrow s``$ & $(i``,s``)\in R\subseteq R`$, что и требуется.

Теперь докажем основное утверждение индукцией по трассе σ. Для пустой трассы из $(i,s)\in R`$ и $i\Rightarrow i`$ по построению имеем $(i`,s)\in R`$ и, очевидно, $s\Rightarrow s$. Пусть утверждение верно для трассы σ и докажем его для трассы $\sigma\cdot\langle u\rangle$, где $u\in\mathbf{L}$. Пусть $(i,s)\in R$ и $i=\sigma\cdot\langle u\rangle\Rightarrow i``$. Тогда существует $i`$ такое, что $i=\sigma\Rightarrow i`$ и $i`=\langle u\rangle\Rightarrow i``$. По предположению шага индукции, $\exists s` \; s=\sigma\Rightarrow s` \;\&\; (i`,s`)\in R`$. Поскольку $R`$ является $\leq^{3}_{ws}$-соответствием для $\mathbf{I}$ и $\mathbf{S}$, то для $(i`,s`)\in R`$ существует $s``$ такое, что $s`=\langle u\rangle\Rightarrow s``$ & $(i``,s``)\in R`$. Наконец, $s=\sigma\Rightarrow s`$ и

$s`=\langle u\rangle\Rightarrow s``$ влечет $s=\sigma\cdot\langle u\rangle\Rightarrow s``$, что и требуется: соответствие $R`$ является $\leq^1_{ws}$-соответствием для **I** и **S**. □

2.4. Отказы

В данной статье под наблюдениями понимаются не только внешние действия из **L**, но и наблюдаемые отказы из **R**. Модификация слабой симуляции с отказами выглядит так (изменения по сравнению с $\leqslant^3_{ws}$ подчеркнуты волнистой линией):

$\mathbf{I}\leq^4_{ws}\mathbf{S} \triangleq \exists R\subseteq V_{\mathbf{I}}\times V_{\mathbf{S}}\ (i_0,s_0)\in R\ \&\ \forall(i,s)\in R\ \forall u\in\mathbf{L}\underset{\sim}{\cup}\mathbf{R}\ \forall i`$

$(i=\langle u\rangle\Rightarrow i`\ \Rightarrow\ \exists s`\ s=\langle u\rangle\Rightarrow s`\ \&\ (i`,s`)\in R).$

Лемма 3: $\leq^4_{ws}\subset\leq^3_{ws}$.

Доказательство. Отношение $\leq^4_{ws}$ предъявляет больше требований к реализации: требуется верификация не только наблюдаемых действий, но и наблюдаемых отказов. □

На классе реализаций, не имеющих наблюдаемых отказов, эти соответствия совпадают: $\leq^4_{ws}=\leq^3_{ws}$.

Заметим, что после отказа $u\in\mathbf{R}$ не может быть τ-действий, то есть маршрут с трассой $\langle u\rangle$ не может заканчиваться τ-переходом, а только виртуальной петлей по отказу $i`\text{—}u\rightarrow i`$ и $s`\text{—}u\rightarrow s`$.

Лемма 4: Отношения $\leq^k_{ws}$, где $k=1\div4$, являются предпорядками (рефлексивны и транзитивны).

Доказательство. Для доказательства рефлексивности отношений достаточно взять тождественное соответствие $R=\{(s,s)\mid s\in V_{\mathbf{S}}\}$. Для доказательства транзитивности отношений достаточно взять композицию соответствий $R=R_1\circ R_2=\{(a,c)\mid\exists b\ (a,b)\in R_1\ \&\ (b,c)\in R_2\}$. Если $(a,c)\in R$, то существует b такое, что $(a,b)\in R_1$ и $(b,c)\in R_2$. Если $a=\langle u\rangle\Rightarrow a`$, то, поскольку $(a,b)\in R_1$, существует $b`$ такое, что $b=\langle u\rangle\Rightarrow b`$ & $(a`,b`)\in R_1$. Тогда, поскольку $(b,c)\in R_2$, существует $c`$ такое, что $c=\langle u\rangle\Rightarrow c`$ & $(b`,c`)\in R_2$. Тогда $(a`,c`)\in R$, что и требуется. □

2.5. Безопасность

Мы будем рассматривать проверку конформности типа слабой симуляции только при безопасном взаимодействии с реализацией.

Состояние s называется *безопасным*, если в этом состоянии не начинается γ-трасса: $s=\langle\gamma\rangle\not\Rightarrow$. При безопасном взаимодействии проходятся только безопасные состояния реализации.

Кнопка $P \in \mathbf{R} \cup \mathbf{Q}$ называется *безопасной в состоянии* s, если ее можно нажимать при безопасном взаимодействии: состояние s безопасно, конвергентно и нажатие кнопки P в состоянии s не вызывает ненаблюдаемого отказа или разрушения после действия, разрешаемого кнопкой:

$$P \ \textbf{\textit{safe}} \ s \triangleq s=\langle\gamma\rangle\not\Rightarrow \ \& \ s\downarrow \ \& \ (P\in\mathbf{Q} \Rightarrow s=\langle P\rangle\not\Rightarrow) \ \& \ \forall z\in P \ s=\langle z,\gamma\rangle\not\Rightarrow.$$

Будем называть наблюдение u *безопасным в состоянии* s, если оно разрешается некоторой кнопкой P ($u\in P\cup\{P\}$), безопасной в этом состоянии P ***safe*** s; в частности должно быть безопасно состояние s. Внутреннее действие τ безопасно в состоянии, если это состояние безопасно. Переход $s—u\rightarrow s`$ *безопасен*, если действие u (внешнее или внутреннее) безопасно в пресостоянии s этого перехода. Пустой маршрут *безопасен*, если безопасно его пресостояние, а непустой маршрут *безопасен*, если каждый его переход безопасен. Состояние *безопасно достижимо*, если оно является концом безопасного маршрута, начинающегося в начальном состоянии.

Теперь мы можем определить модификацию слабой симуляции с отказами и безопасностью. Прежде всего, заметим, что, если начальное состояние спецификации не безопасно, то есть $s_0=\langle\gamma\rangle\Rightarrow$, то это разрешает любое поведение реализации, в том числе разрушение с самого начала $i_0=\langle\gamma\rangle\Rightarrow$. В этом случае любая реализация конформна спецификации, но некоторые реализации нельзя не только тестировать, но даже запускать на выполнение, поскольку они могут разрушиться с самого начала. Если же $s_0=\langle\gamma\rangle\not\Rightarrow$, то начальные состояния должны соответствовать друг другу $(i_0,s_0)\in R$.

Далее, при безопасном взаимодействии с реализацией может нажиматься только такая кнопка P, которая безопасна в текущем состоянии реализации i. Если кнопка P безопасна в некотором соответствующем i спецификационном состоянии s, то в этом состоянии s спецификация описывает те наблюдения $u\in P\cup\{P\}$, которые допустимы в конформной реализации после нажатия кнопки P в состоянии i, и постсостояния $s`$ спецификации, хотя бы одно из которых должно соответствовать постсостоянию $i`$ реализации. Иными словами, любое наблюдение в реализации после нажатия кнопки P в состоянии i должно быть и в спецификации после нажатия той же кнопки P в соответствующем состоянии s при условии, что эта кнопка безопасна в s, и в этом случае постсостоянию $i`$ реализации должно соответствовать хотя бы одно постсостояние $s`$.

Если же кнопка P опасна в некотором соответствующем i спецификационном состоянии s_1, то такое состояние s_1 никак не регламентирует поведение реализации после нажатия кнопки P. Это, однако,

не означает, что в состоянии i могут быть любые наблюдения $u \in P \cup \{P\}$ с любыми постсостояниями $i`$, поскольку наблюдение u может разрешаться той же кнопкой P в другом спецификационном состоянии s, в котором кнопка P безопасна, или другой кнопкой, безопасной в том же или другом спецификационном состоянии, соответствующем состоянию i. Для проверки конформности сравниваются наблюдения и постсостояния в реализации и спецификации только после нажатия кнопок, безопасных в них обеих. Модификация слабой симуляции с отказами и безопасностью выглядит так (изменения по сравнению с $\leqslant^4_{ws}$ подчеркнуты волнистой линией):

$$\mathbf{I} \leq^5_{ws} \mathbf{S} \triangleq \exists R \subseteq V_{\mathbf{I}} \times V_{\mathbf{S}} \ (s_0 = \langle\gamma\rangle \not\Rightarrow \Rightarrow (i_0, s_0) \in R)$$

$$\& \ \forall (i,s) \in R \ \forall P \ \textit{safe} \ i \ \forall u \in P \cup \{P\} \ \forall i`$$

$$(P \ \textit{safe} \ s \ \& \ i = \langle u \rangle \Rightarrow i` \ \Rightarrow \exists s` \ s = \langle u \rangle \Rightarrow s` \ \& \ (i`, s`) \in R).$$

Лемма 5: $\leq^4_{ws} \subset \leq^5_{ws}$.

Доказательство. Отношение $\leq^4_{ws}$ предъявляет больше требований к реализации: требуется верификация наблюдений, разрешаемых не только безопасными кнопками. □

На классе реализаций и спецификаций, в которых нет ненаблюдаемых отказов, дивергенции и разрушения (то есть все кнопки безопасны), эти соответствия совпадают: $\leq^4_{ws} = \leq^5_{ws}$.

Лемма 6: $\leq^3_{ws} \not\subseteq \leq^5_{ws}$ и $\leq^3_{ws} \not\supseteq \leq^5_{ws}$.

Доказательство. Смотри пример на 0. □

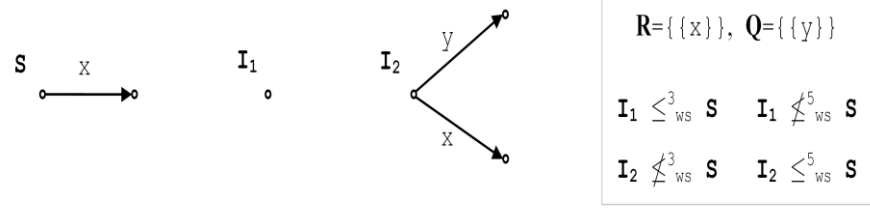


Рис. 2. Соотношение симуляций $\leq^3_{ws}$ и $\leq^5_{w}$.

Лемма 7: Симуляция $\leq^5_{ws}$ рефлексивна, но не транзитивна.

Доказательство. Для доказательства рефлексивности симуляции достаточно взять тождественное соответствие $R = \{(s,s) \mid s \in V_{\mathbf{S}}\}$. Пример нетранзитивности симуляции на 0. □

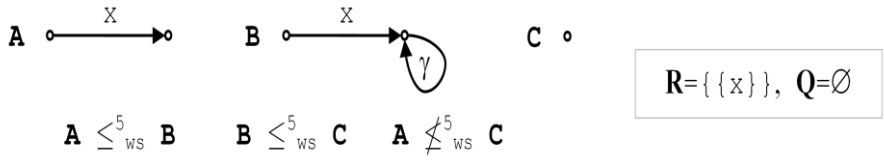


Рис. 3. Нетранзитивность отношения $\preceq^5_{ws}$.

2.6. Гипотеза о безопасности

Поскольку спецификация задана, мы можем проверять по ней условие P ***safe*** s. Но при тестировании (в отличие от аналитической верификации) реализация неизвестна, и судить о безопасности кнопок в состояниях реализации (P ***safe*** i) мы можем только на основании некоторой *гипотезы о безопасности*. В данной работе такая гипотеза основана на некотором соответствии $H \subseteq V_I \times V_S$ состояний реализации и спецификации[4]. Мы будем называть ее H-*гипотезой* о безопасности. Она предполагает 1) безопасность начального состояния i_0 реализации, если безопасно начальное состояние s_0 спецификации, 2) безопасность кнопки в состоянии реализации, если она безопасна хотя бы в одном в соответствующем по H состоянии спецификации.

Определим соответствие H рекурсивно. Начальные состояния i_0 и s_0 соответствуют друг другу, если они оба безопасны. Также соответствуют друг другу любые два состояния, достижимые из безопасных начальных состояний по пустой трассе. Два состояния $i`$ и $s`$ соответствуют друг другу, если они достижимы из соответствующих друг другу состояний i и s по одному и тому же наблюдению u, разрешаемому кнопкой P, которая безопасна в обоих состояниях i и s. Формально соответствие H определяется как минимальное соответствие, порождаемое следующими правилами вывода:

$$\forall i, i` \in V_I \ \ \forall s, s` \in V_S \ \ \forall P \in \mathbf{R} \cup \mathbf{Q} \ \ \forall u \in P \cup \{P\}$$

$$s_0 = \langle\gamma\rangle\not\Rightarrow \ \& \ i_0 = \langle\gamma\rangle\not\Rightarrow \ \& \ i_0 \Rightarrow i \ \& \ s_0 \Rightarrow s$$

$$\vdash (i, s) \in H,$$

$$(i, s) \in H \ \& \ P \textbf{ ***safe*** } i \ \& \ P \textbf{ ***safe*** } s \ \& \ i = \langle u\rangle \Rightarrow i` \ \& \ s = \langle u\rangle \Rightarrow s` \quad \vdash$$

$$(i`, s`) \in H.$$

Соответствие H похоже на соответствие R слабой симуляции $\preceq^5_{ws}$, но имеет три существенных отличия. 1) Начальные состояния i_0 и s_0 соответствуют друг другу по R, если безопасно состояние s_0, а соответствие

[4] Мнемоника: H – **H**ypothesis.

H требует безопасности обоих состояний i_0 и s_0. 2) Постсостоянию i` после наблюдения u, разрешаемому кнопкой P, которая безопасна в обоих соответствующих состояниях i и s, по R должно соответствовать хотя бы одно постсостояние s`, а по H – каждое постсостояние. 3) Соответствие H минимальное, то есть состояния соответствуют друг другу только в том случае, когда они достижимы из безопасных начальных состояний нажатием одних и тех же безопасных кнопок; соответствие R может содержать пары состояний, которые недостижимы таким образом (даже вообще недостижимы из начальных состояний).

Кнопку P будем называть H-*безопасной* в реализационном состоянии i, если она безопасна хотя бы в одном соответствующем i спецификационном состоянии s:

$P\ H\text{-}\mathit{safe}\ i \triangleq \exists s\ (i,s)\in H\ \&\ P\ \mathit{safe}\ s.$

Теперь дадим формальное определение H-гипотезы:

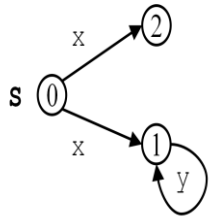
$\mathbf{I}\ H\text{-}\mathit{safe}\ \mathbf{S} \triangleq (s_0=\langle\gamma\rangle\not\Rightarrow \Rightarrow i_0=\langle\gamma\rangle\not\Rightarrow)$

$\&\quad \forall P\in \mathbf{R}\cup\mathbf{Q}\ (P\ H\text{-}\mathit{safe}\ i \Rightarrow P\ \mathit{safe}\ i).$

Реализацию, удовлетворяющую H-гипотезе, будем называть H-*безопасной*.

Лемма 8: Отношение безопасности H-***safe*** не рефлексивно и не транзитивно.

Доказательство. Пример нерефлексивности на 0. Пример нетранзитивности на 0. □



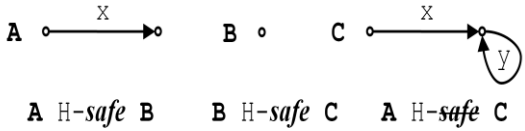
H={00, 11, 12, 21, 22}

{y} ***safe*** 1 ⇒ {y} H-***safe*** 2,

но {y} ~~***safe***~~ 2.

R={{x}}, **Q**={{y}}

Рис. 4. Нерефлексивность отношения безопасности.



R={{x}}, **Q**={{y}}

Рис. 5. Нетранзитивность отношения безопасности.

Если H-гипотеза выполнена, то либо $H=\varnothing$, если $s_0=\langle\gamma\rangle\Rightarrow$, либо $(i_0,s_0)\in H$ в противном случае.

При условии выполнения H-гипотезы определим маршруты реализации, которые могут быть пройдены при безопасном тестировании (H-безопасные маршруты), и состояния реализации, которые при этом могут достигаться (H-достижимые состояния). Маршрут реализации H-*безопасен*, если начальное состояние спецификации безопасно (тем самым по H-гипотезе начальное состояние реализации тоже безопасно) и каждый переход $i\text{—}u\rightarrow i`$ маршрута, кроме τ-переходов, помечен наблюдением u, разрешаемым кнопкой, которая H-безопасна в пресостоянии i этого перехода (то есть безопасна в некотором состоянии s спецификации, соответствующем по H состоянию i, и тем самым по H-гипотезе безопасна в состоянии i). Состояние реализации H-*достижимо*, если оно достижимо по H-безопасному маршруту, то есть является его постсостоянием.

Заметим, что пресостоянию i каждого перехода $i\text{—}u\rightarrow i`$ H-безопасного маршрута соответствует по H хотя бы одно состояние спецификации, а для постсостояния i` это не обязательно верно, если это не пресостояние следующего перехода (постсостояние маршрута). Соответствие H естественным образом индуцирует соответствие H` маршрутов реализации и спецификации. Во-первых, каждому переходу по наблюдению одного маршрута соответствует переход по тому же наблюдению другого маршрута, причем это наблюдение разрешается кнопкой, которая безопасна в пресостояниях этих переходов как в спецификации, так и в реализации. Во-вторых, каждому состоянию одного маршрута соответствует по H некоторое состояние другого маршрута таким образом, что, если пресостояния переходов маршрутов соответствуют друг другу, то постсостояния этих переходов тоже соответствуют друг другу. Все реализационные маршруты, имеющие по H` соответствующие спецификационные маршруты, являются H-безопасными. Продолжения таких реализационных маршрутов, получаемые нажатием кнопок, которые H-безопасны в постсостояниях маршрутов, могут не иметь по H` соответствующих спецификационных маршрутов, но также являются H-безопасными[5].

2.7. Безопасная симуляция

Теперь, соединяя H-гипотезу о безопасности и слабую симуляцию, мы получаем окончательный вариант слабой симуляции с отказами и

[5] Эти продолжения аналогичны безопасным продолжениям безопасных трасс (то есть тестовым трассам) в теории трассовой конформности [2,3].

безопасностью (изменения по сравнению с $\leqslant^5_{ws}$ подчеркнуты волнистой линией), которую будем называть *безопасной симуляцией* и обозначать ***ss***:[6]

I ***ss*** **S** $\triangleq$ **I** H-***safe*** **S** & $\exists R \subseteq V_I \times V_S$ $(s_0 = \langle\gamma\rangle\not\Rightarrow \Rightarrow (i_0, s_0) \in R)$
& $\forall (i,s) \in R\ \forall P$ H-***safe*** i $\forall u \in P \cup \{P\}\ \forall i`$
$(P$ ***safe*** s & $i = \langle u\rangle \Rightarrow i` \Rightarrow \exists s`\ s = \langle u\rangle \Rightarrow s`$ & $(i`, s`) \in R)$.

Если реализация задана явно, то можно аналитически проверять как H-гипотезу, так и безопасную симуляцию. Когда реализация неизвестна, верификация симуляции выполняется с помощью тестирования. В этом случае H-гипотеза является предусловием безопасного тестирования.

Если $s_0 = \langle\gamma\rangle\Rightarrow$, то $H = \varnothing$, безопасное тестирование невозможно, но и не нужно, так как любая реализация конформна (при любом R). Если $s_0 = \langle\gamma\rangle\not\Rightarrow$, то тестирование заключается в проверке *тестируемого условия* (нижние две строки определения ***ss***). Если существует кнопка P H-***safe*** i, то состояние i H-достижимо. Для каждого H-достижимого состояния i нажимается каждая H-безопасная в нем кнопка P, и полученные наблюдение u и постсостояние i` верифицируются по спецификации: наблюдение u должно быть в каждом соответствующем ему по R состоянии s, в котором кнопка P безопасна, а среди постсостояний s` хотя бы одно должно соответствовать i` по R.

Лемма 9: H-***safe*** $\cap \leq^5_{ws} \subset$ ***ss***.

Доказательство. Отношение $\leq^5_{ws}$, во-первых, не требует выполнения H-гипотезы о безопасности, и, во-вторых, требует верификации наблюдений, разрешаемых не только H-безопасными кнопками, а любыми кнопками, безопасными как в реализационном, так и в соответствующем ему по R спецификационном состояниях. □

Для класса спецификаций без ненаблюдаемых отказов, дивергенции и разрушения (когда все кнопки безопасны в спецификации), имеем: H-***safe*** $\cap \leq^5_{ws} =$ ***ss***, а на поддомене безопасных реализаций $\leq^5_{ws} =$ ***ss***.

Лемма 10: Отношение ***ss***, вообще говоря, нерефлексивно, но на классе спецификаций, удовлетворяющих собственной H-гипотезе, рефлексивно: $\forall$**S S** H-***safe*** **S** $\Rightarrow$ **S** ***ss*** **S**.

Доказательство. Нерефлексивность ***ss*** в общем случае следует из нерефлексивности отношения H-***safe*** (лемма 8). Рефлексивность ***ss*** для спецификаций, удовлетворяющих собственной гипотезе о безопасности, следует из леммы 9 и рефлексивности отношения $\leq^5_{ws}$. □

[6] Мнемоника: ***ss*** – **S**afe **S**imulation.

Лемма 11: Для конформного по ***ss*** соответствия R соответствие R∩H тоже конформно.

Доказательство. Если $s_0=\langle\gamma\rangle\Rightarrow$, то H=∅ и любая реализация конформна при любом соответствии R, в частности при R∩∅=∅. Если $s_0=\langle\gamma\rangle\not\Rightarrow$, то, выполнение H-гипотезы влечет $(i_0,s_0)\in H$, а конформность соответствия R влечет $(i_0,s_0)\in R$, следовательно, $(i_0,s_0)\in R\cap H$. Пусть для некоторых состояний, кнопки и наблюдения выполняется: (i,s)∈R∩H & P H-***safe*** i & u∈P∪{P} & P ***safe*** s & $i=\langle u\rangle\Rightarrow i`$. Тогда по H-гипотезе P H-***safe*** i влечет P ***safe*** i, а по конформности соответствия R, существует такое состояние s`, что $s=\langle u\rangle\Rightarrow s`$ & $(i`,s`)\in R$. Тем самым выполнены условия второго правила вывода в определении соответствия H. Следовательно, $(i`,s`)\in H$. В результате имеем $(i`,s`)\in R\cap H$, что и требовалось доказать. □

Эта лемма позволяет переформулировать определение безопасной симуляции следующим образом:

I *ss* S ≙ **I** H-***safe*** **S** & $\exists R\subseteq H$ $(s_0=\langle\gamma\rangle\not\Rightarrow \Rightarrow (i_0,s_0)\in R)$
& ∀(i,s)∈R ∀P ***safe*** s ∀u∈P∪{P} ∀i`
$(i=\langle u\rangle\Rightarrow i` \Rightarrow \exists s` \; s=\langle u\rangle\Rightarrow s` \;\&\; (i`,s`)\in R)$.

Лемма 12: Объединение конформных по ***ss*** соответствий конформно.

Доказательство. Тривиально.

Из последних двух лемм следует, что можно ограничиться только такими соответствиями R, которые вложены в H. Мы имеем два естественных конформных соответствия: R_1 – объединение всех конформных соответствий, и $R_2=R_1\cap H$. Ниже в разделе, посвященном практическому тестированию конформности ***ss***, мы покажем, как можно при некоторых ограничениях в процессе тестирования строить такие конформные соответствия или доказывать их несуществование для заданной спецификации и любой фиксированной, но неизвестной реализации, удовлетворяющей H-гипотезе.

Лемма 13: Отношение ***ss*** транзитивно.

Доказательство. Пусть **A *ss* B** и **B *ss* C**.

Обозначим соответствующие отношения H и конформные соответствия через H_{AB}, H_{BC}, H_{AC}, R_{AB}, R_{BC}. По лемме 11 будем считать, что $R_{AB}\subseteq H_{AB}$, $R_{BC}\subseteq H_{BC}$.

Обозначим $R_{AC}=R_{AB}\circ R_{BC}=\{(a,c)\mid \exists b\ (a,b)\in R_{AB} \;\&\; (b,c)\in R_{BC}\}$.

Если $c_0=\langle\gamma\rangle\Rightarrow$, то любая реализация конформна спецификации **C**, в частности **A** ***ss*** **C**. Далее будем считать, что $c_0=\langle\gamma\rangle\not\Rightarrow$.

Сначала покажем, что **A** H-***safe*** **C**. Так как $c_0=\langle\gamma\rangle\not\Rightarrow$, то по конформности соответствия R_{BC} имеем $b_0=\langle\gamma\rangle\not\Rightarrow$, а тогда по конформности соответствия R_{AB} имеем $a_0=\langle\gamma\rangle\not\Rightarrow$.

Теперь нам нужно доказать, что, если $(a,c)\in H_{AC}$ & P ***safe*** c, то P ***safe*** a. Мы докажем более сильное утверждение, включающее дополнительное требование: $\exists b\ (a,b)\in H_{AB}$ & $(b,c)\in H_{BC}$.

Сначала покажем, что утверждение верно для пары (a,c), полученной по 1-ому правилу вывода для H_{AC}. Действительно, если $c_0=\langle\gamma\rangle\not\Rightarrow$, $a_0=\langle\gamma\rangle\not\Rightarrow$, $a_0\Rightarrow a$ и $c_0\Rightarrow c$, то по H_{BC}-безопасности B имеем $b_0=\langle\gamma\rangle\not\Rightarrow$. 1-ое правило вывода применимо к паре (a,b_0), то есть $(a,b_0)\in H_{AB}$. Также 1-ое правило вывода применимо к паре (b_0,c), то есть $(b_0,c)\in H_{BC}$. Тогда по H_{BC}-безопасности B имеем P ***safe*** b_0, а отсюда по H_{AB}-безопасности A имеем P ***safe*** a.

Теперь предположим, что утверждение верно для пары (a,c) и выполнены условия 2-го правила вывода: $(a,c)\in H_{AC}$ & P ***safe*** a & P ***safe*** c & $a=\langle u\rangle\Rightarrow a`$ & $c=\langle u\rangle\Rightarrow c`$. Докажем утверждение для пары $(a`,c`)\in H_{AC}$ & $P`$ ***safe*** $c`$. По предположению $\exists b\ (a,b)\in H_{AB}$ & $(b,c)\in H_{BC}$. Тогда по H_{BC}-безопасности B имеем P ***safe*** b, а тогда по R_{AB}-конформности существует $b`$ такое, что $b=\langle u\rangle\Rightarrow b`$ & $(a`,b`)\in R_{AB}\subseteq H_{AB}$. Тогда по 2-ому правилу вывода для H_{BC} имеем $(b`,c`)\in H_{BC}$. Тогда по H_{BC}-безопасности B имеем $P`$ ***safe*** $b`$, а тогда по H_{AB}-безопасности A имеем $P`$ ***safe*** $a`$.

Утверждение о том, что **A** H-***safe*** **C**, доказано. Теперь докажем, что для **A** и **C** выполнено тестируемое условие конформности ***ss*** для соответствия R_{AC}. Действительно, пусть $(a,c)\in R_{AC}$, P ***safe*** c, $u\in P\cup\{P\}$ и $a=\langle u\rangle\Rightarrow a`$. Тогда существует $b$ такое, что $(a,b)\in R_{AB}$ и $(b,c)\in R_{BC}$. По $H_{BC}$-безопасности $B$ имеем $P$ ***safe*** $b$, а тогда по $R_{AB}$-конформности $A$ существует $b`$ такое, что $b=\langle u\rangle\Rightarrow b`$ и $(a`,b`)\in R_{AB}$. Но по $R_{BC}$-конформности $B$ существует $c`$ такое, что $c=\langle u\rangle\Rightarrow c`$ и $(b`,c`)\in R_{BC}$. Тем самым, $(a`,c`)\in R_{AC}$. □

Суммируя леммы 1÷13, получаем следующую теорему, описывающую соотношение различных симуляций:

Теорема 1:

1. $\leq^4_{ws} \subset \leq^3_{ws} = \leq^2_{ws} = \leq^1_{ws}$.
2. $^4_{ws} \subset \leq^5_{ws}$.
3. $\leq^3_{ws} \not\subseteq \leq^5_{ws}$ и $\leq^3_{ws} \not\supseteq \leq^5_{ws}$.
4. Симуляции $\leq^k_{ws}$, где k=1÷4, являются предпорядками.
5. Симуляция $\leq^5_{ws}$ рефлексивна, но не транзитивна.
6. Отношение H-***safe*** нерефлексивно и нетранзитивно.
7. H-***safe*** $\cap \leq^5_{ws} \subset$ ***ss***.
8. Отношение ***ss***, вообще говоря, нерефлексивно, но на классе спецификаций, удовлетворяющих собственной H-гипотезе, рефлексивно.
9. Отношение ***ss*** транзитивно и на классе спецификаций, удовлетворяющих собственной H-гипотезе, является предпорядком.

3. Связь симуляции с трассовой конформностью

3.1. Трассовые гипотеза о безопасности и конформность

В трассовой теории конформности гипотеза о безопасности основывалась на трассах реализации и спецификации [1,2,3,5] и не требовала соответствия состояний реализации и спецификации. Напомним основные определения этой теории.

Для реализации $\mathtt{I}$ определяется отношение ***safe in*** безопасности кнопки $\mathtt{P} \in \mathbf{R} \cup \mathbf{Q}$ после трассы $\sigma \in \boldsymbol{T}(\mathtt{I})$:

$\mathtt{P}$ ***safe in*** $\mathtt{I}$ ***after*** $\sigma \triangleq (\mathtt{P} \in \mathbf{R} \vee \sigma \cdot \langle \mathtt{P} \rangle \notin \boldsymbol{T}(\mathtt{I}))$

$\& \ \forall \mathtt{z} \in \mathtt{P} \ \sigma \cdot \langle \mathtt{z}, \gamma \rangle \notin \boldsymbol{T}(\mathtt{I}) \ \& \ \sigma \cdot \langle \Delta \rangle \notin \boldsymbol{T}(\mathtt{I})$.

Очевидно, что если кнопка безопасна по ***safe in*** после трассы ($\mathtt{P}$ ***safe in*** $\mathtt{I}$ ***after*** σ), то она безопасна в каждом состоянии после этой трассы: $\forall \mathtt{i} \in (\mathtt{I}$ ***after*** $\sigma)$ $\mathtt{P}$ ***safe*** $\mathtt{i}$.

Для спецификации отношение ***safe by*** безопасности кнопок после трасс определяется неоднозначно: это любое отношение, удовлетворяющее трем требованиям: $\forall \sigma \in \boldsymbol{T}(\mathbf{S}) \ \forall \mathtt{R} \in \mathbf{R} \ \forall \mathtt{z} \in \mathbf{L} \ \forall \mathtt{Q} \in \mathbf{Q}$

1) $\mathtt{R}$ ***safe by*** $\mathbf{S}$ ***after*** $\sigma \Leftrightarrow \forall \mathtt{u} \in \mathtt{R} \ \sigma \cdot \langle \mathtt{u}, \gamma \rangle \notin \boldsymbol{T}(\mathbf{S}) \ \& \ \sigma \cdot \langle \Delta \rangle \notin \boldsymbol{T}(\mathbf{S})$,

2) $\sigma \cdot \langle \mathtt{z} \rangle \in \boldsymbol{T}(\mathbf{S}) \ \& \ \exists \mathtt{T} \in \mathbf{R} \cup \mathbf{Q} \ \mathtt{z} \in \mathtt{T} \ \& \ \forall \mathtt{u} \in \mathtt{T} \ \sigma \cdot \langle \mathtt{u}, \gamma \rangle \notin \boldsymbol{T}(\mathbf{S}) \ \& \ \sigma \cdot \langle \Delta \rangle \notin \boldsymbol{T}(\mathbf{S})$
$\Rightarrow \exists \mathtt{P} \in \mathbf{R} \cup \mathbf{Q} \ \mathtt{z} \in \mathtt{P} \ \& \ \mathtt{P}$ ***safe by*** $\mathbf{S}$ ***after*** σ,

3) $\mathtt{Q}$ ***safe by*** $\mathbf{S}$ ***after*** σ
$\Rightarrow \exists \mathtt{v} \in \mathtt{Q} \ \sigma \cdot \langle \mathtt{v} \rangle \in \boldsymbol{T}(\mathbf{S}) \ \& \ \forall \mathtt{u} \in \mathtt{Q} \ \sigma \cdot \langle \mathtt{u}, \gamma \rangle \notin \boldsymbol{T}(\mathbf{S}) \ \& \ \sigma \cdot \langle \Delta \rangle \notin \boldsymbol{T}(\mathbf{S})$.

Будем считать, что вместе со спецификацией задано отношение ***safe by***, удовлетворяющее этим трем требованиям.

R-трасса σ спецификации **S** называется безопасной, если спецификация не содержит трассу $\langle\gamma\rangle$, а трасса σ не заканчивается на дивергенцию и разрушение, и каждый встречающийся в ней символ u (внешнее действие или **R**-отказ) безопасен после непосредственно предшествующего ему префикса трассы:

$$\textbf{\textit{SafeBy}}(\mathbf{S}) \triangleq \{\sigma\in \textbf{\textit{T}}(\mathbf{S}) \mid \langle\gamma\rangle\notin \textbf{\textit{T}}(\mathbf{S}) \ \&\ \forall\mu,\lambda,u\ (\sigma=\mu\cdot\langle u\rangle\cdot\lambda \Rightarrow u \ \textbf{\textit{safe by}}\ \mathbf{\Sigma}\ \textbf{\textit{after}}\ \mu)\}.$$

Для кнопок $R\in\mathbf{R}$ отношения ***safe by*** и ***safe in*** совпадают. Поэтому, если кнопка $R\in\mathbf{R}$ безопасна по ***safe by*** после трассы (R ***safe by*** **S** ***after*** σ), то она безопасна в каждом состоянии после этой трассы: $\forall s\in(\mathbf{S}$ ***after*** $\sigma)$ R ***safe*** s. Однако кнопка $Q\in\mathbf{Q}$, которая безопасна по ***safe by*** после трассы, может быть опасна в некоторых (но не всех) состояниях $s\in(\mathbf{S}$ ***after*** $\sigma)$. Тем не менее, верна следующая лемма:

Лемма 14: Если трасса безопасна по отношению ***safe by*** (в спецификации) или ***safe in*** (в реализации), то все маршруты с этой трассой безопасны (соответственно, в спецификации или в реализации). Обратное, вообще говоря, не верно.

Доказательство. Доказательство будем вести индукцией по трассе. Пустая трасса безопасна, если нет трассы $\langle\gamma\rangle$, а это означает безопасность всех состояний, достижимых по пустой трассе и, следовательно, безопасность всех маршрутов с пустой трассой. Пусть утверждение верно для трассы σ, и докажем его для трассы $\sigma\cdot\langle u\rangle$, где наблюдение u разрешается кнопкой P, которая безопасна после σ по отношению ***safe by*** или ***safe in***. Нам нужно показать, что в каждом состоянии после трассы σ, в котором имеется переход по u (включая виртуальные переходы по отказам из **R**), кнопка P безопасна.

Действительно, если кнопка $P\in\mathbf{R}$, то отношения ***safe by*** и ***safe in*** для такой кнопки совпадают, и ее безопасность по этим отношениям означает ее безопасность в каждом состоянии после трассы σ. Если кнопка $P\in\mathbf{Q}$, то ее безопасность по ***safe in*** также означает ее безопасность в каждом состоянии после трассы σ. А безопасность кнопки $P\in\mathbf{Q}$ по отношению ***safe by*** означает ее безопасность в каждом состоянии после трассы σ, в котором нет отказа $\{P\}$. Поскольку для $P\in\mathbf{Q}$ должно быть $u\neq\{P\}$, то есть наблюдение u – это внешнее действие (не отказ), кнопка P безопасна в каждом состоянии после трассы σ, в котором есть переход по u.

Из безопасности всех маршрутов с данной трассой, вообще говоря, не следует безопасность этой трассы, что показывается примером на 0. □

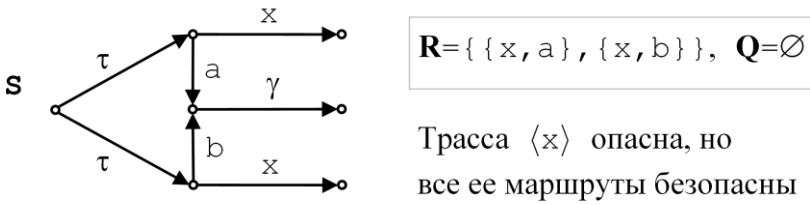


Рис. 6. Пример опасной трассы, все маршруты которой безопасны.

Трассовая гипотеза о безопасности определялась следующим образом:

I ***safe for*** **S** $\triangleq$ $(\langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{S}) \Rightarrow \langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{I}))$ & $\forall\sigma\in$***SafeBy***$(\mathbf{S})\cap \boldsymbol{T}(\mathbf{I})$

$\forall P\in \mathbf{R}\cup\mathbf{Q}$

(P ***safe by*** **S** ***after*** σ $\Rightarrow$ P ***safe in*** **I** ***after*** σ).

Трассовая конформность определялась так:

I ***saco*** **S** $\triangleq$ **I** ***safe for*** **S** & $\forall\sigma\in$***SafeBy***$(\mathbf{S})\cap \boldsymbol{T}(\mathbf{I})$ $\forall$P ***safe by*** **S** ***after*** σ $\forall i\in(\mathbf{I}$ ***after*** $\sigma)$ $\forall u\in P\cup\{P\}$ $(i=\langle u\rangle\Rightarrow \Rightarrow \exists s\in(\mathbf{S}$ ***after*** $\sigma)$ $s=\langle u\rangle\Rightarrow)$.

3.2. Связь H-гипотезы с трассовой гипотезой о безопасности

Лемма 15: H-гипотеза предъявляет к реализации более сильные требования, чем трассовая гипотеза о безопасности: H-***safe*** $\subseteq$ ***safe for***.

Доказательство. Пусть **I** H-***safe*** **S**. Покажем, что **I** ***safe for*** **S**.

Сначала покажем, что $\langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{S}) \Rightarrow \langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{I})$. Условие $\langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{S})$ эквивалентно условию $s_0=\langle\gamma\rangle\not\Rightarrow$, а условие $\langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{I})$ эквивалентно условию $i_0=\langle\gamma\rangle\not\Rightarrow$. По H-гипотезе имеем $s_0=\langle\gamma\rangle\not\Rightarrow \Rightarrow i_0=\langle\gamma\rangle\not\Rightarrow$.

Теперь нам нужно показать, что если $\sigma\in$***SafeBy***$(\mathbf{S})\cap \boldsymbol{T}(\mathbf{I})$ и кнопка P ***safe by*** **S** ***after*** σ, то P ***safe in*** **I** ***after*** σ. Условие P ***safe in*** **I** ***after*** σ эквивалентно условию $\forall i\in(\mathbf{I}$ ***after*** $\sigma)$ P ***safe*** i. Из условия P ***safe by*** **S** ***after*** σ следует, что кнопка P ***safe*** s хотя бы для одного состояния $s\in(\mathbf{S}$ ***after*** $\sigma)$. Если $(i,s)\in H$, то по H-гипотезе P ***safe*** i. Поэтому нам достаточно показать, что $\forall\sigma\in$***SafeBy***$(\mathbf{S})\cap \boldsymbol{T}(\mathbf{I})$ $\forall s\in(\mathbf{S}$ ***after*** $\sigma)$ $\forall i\in(\mathbf{I}$ ***after*** $\sigma)$ $(i,s)\in H$.

Будем вести доказательство индукцией по трассе $\sigma\in$***SafeBy***$(\mathbf{S})\cap \boldsymbol{T}(\mathbf{I})$. Пустая трасса $\sigma=\langle\rangle$ всегда есть в реализации и безопасна в спецификации, если $\langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{S})$, то есть $s_0=\langle\gamma\rangle\not\Rightarrow$, а тогда по доказанному $\langle\gamma\rangle\notin \boldsymbol{T}(\mathbf{I})$, то есть $i_0=\langle\gamma\rangle\not\Rightarrow$. А тогда по 1-ому правилу вывода для соответствия H имеем

$\forall s\in(\mathbf{S}\ \textit{after}\ \langle\rangle)\ \ \forall i\in(\mathbf{I}\ \textit{after}\ \langle\rangle)\ \ (i,s)\in H$. Пусть утверждение верно для трассы σ и докажем его для трассы $\sigma\cdot\langle u\rangle$, где наблюдение u разрешается кнопкой P ***safe by*** $\mathbf{S}$ ***after*** σ. Пусть $s`\in(\mathbf{S}\ \textit{after}\ \sigma\cdot\langle u\rangle)$ и $i`\in(\mathbf{I}\ \textit{after}\ \sigma\cdot\langle u\rangle)$. Тогда $\exists s\in(\mathbf{S}\ \textit{after}\ \sigma)\ \ P\ \textit{safe}\ s\ \&\ s=\langle u\rangle\Rightarrow s`$. Также $\exists i\in(\mathbf{I}\ \textit{after}\ \sigma)\ \ i=\langle u\rangle\Rightarrow i`$. По предположению шага индукции $(i,s)\in H$, следовательно, по H-гипотезе $P\ \textit{safe}\ i$. А тогда выполнены условия 2-го правила вывода для соответствия H, и мы имеем $(i`,s`)\in H$, что и требовалось доказать. □

Можно рассматривать симуляцию с трассовой гипотезой о безопасности. Такая симуляция определяется следующим образом (изменения по сравнению с ***ss*** подчеркнуты волнистой линией):

$\mathbf{I}\ \textit{sst}\ \mathbf{S}\ \triangleq\ \mathbf{I}\ \underline{\textit{safe for}}\ \mathbf{S}\ \&\ \exists R\subseteq V_{\mathbf{I}}\times V_{\mathbf{S}}\ \ (s_0=\langle\gamma\rangle\not\Rightarrow\ \Rightarrow\ (i_0,s_0)\in R)$
$\&\ \forall(i,s)\in R\ \ \underline{\forall\sigma\in\textit{SafeBy}(\mathbf{S})\cap\mathbf{I}\ \ \forall P\ \textit{safe by}\ \mathbf{S}\ \textit{after}\ \sigma}\ \ \forall u\in P\cup\{P\}\ \ \forall i`$
$(P\ \textit{safe}\ s\ \&\ \underline{i\in(\mathbf{I}\ \textit{after}\ \sigma)\ \&}\ i=\langle u\rangle\Rightarrow i`\ \Rightarrow\ \exists s`\ s=\langle u\rangle\Rightarrow s`\ \&\ (i`,s`)\in R)$
.

Симуляция с H-гипотезой о безопасности предъявляет более сильные требования к реализации, чем симуляция с трассовой гипотезой о безопасности.

Лемма 16: Из симуляции с H-гипотезой о безопасности следует симуляция с трассовой гипотезой о безопасности, но не наоборот: ***ss*** $\subset$ ***sst***.

Доказательство. Из леммы 15 непосредственно следует, что симуляция с H-гипотезой влечет симуляцию с трассовой гипотезой о безопасности: ***ss*** $\subseteq$ ***sst***. Это объясняется тем, что множество кнопок, безопасных в состоянии реализации по ***safe for***, вложено во множество кнопок, H-безопасных в этом состоянии. Иными словами, для ***sst*** нужно выполнять меньше проверок, чем для ***ss***. Но из ***sst***, вообще говоря, не следует ***ss***. Пример на 0. □

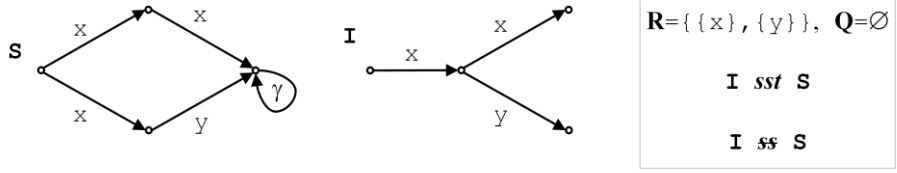


Рис. 7.Пример симуляции с трассовой гипотезой о безопасности при отсутствии симуляции с H-гипотезой о безопасности.

3.3. Симуляция с трассовой гипотезой о безопасности

Симуляция с трассовой гипотезой о безопасности, в свою очередь, предъявляет более сильные требования к реализации, чем трассовая конформность, основанная на той же гипотезе о безопасности.

Лемма 17: Из безопасной симуляции с трассовой гипотезой о безопасности следует трассовая конформность, но не наоборот: ***sst*** $\subset$ ***saco***.

Доказательство. Пусть **I** ***sst*** **S** и R – конформное соответствие. Пополним соответствие, добавив в него все пары (i, s_0), где $i_0 \Rightarrow i$, при условии безопасности начального состояния спецификации: $s_0 = \langle\gamma\rangle \not\Rightarrow$. Пополненное соответствие R` также является конформным соответствием. Действительно, нам надо показать, что если для некоторой трассы $\sigma \in$ ***SafeBy***$(\mathbf{S})$, кнопки P ***safe by*** **S** ***after*** σ и наблюдения $u \in P \cup \{P\}$ в реализации для состояния $i \in (\mathbf{I}$ ***after*** $\sigma)$ имеет место $i = \langle u\rangle \Rightarrow i`$, то в спецификации кнопка P либо опасна в $s_0$, либо найдется такое состояние s`, что $s_0 = \langle u\rangle \Rightarrow s`$ и $(i`, s`) \in R$. Поскольку $i_0 \Rightarrow i$ и $i = \langle u\rangle \Rightarrow i`$, имеем $i_0 = \langle u\rangle \Rightarrow i`$. Если P ***safe*** $s_0$, то по конформности соответствия R в спецификации найдется такое состояние s`, что $s_0 = \langle u\rangle \Rightarrow s`$ и $(i`, s`) \in R \subseteq R`$, что и требовалось доказать.

Теперь докажем вспомогательное утверждение: если трасса σ безопасна в спецификации, то есть $\sigma \in$ ***SafeBy***$(\mathbf{S})$, и заканчивается в реализации в состоянии i, то есть $i \in (\mathbf{I}$ ***after*** $\sigma)$, то $(\mathbf{S}$ ***after*** $\sigma) \cap R`(i) \neq \varnothing$. Доказательство будем вести индукцией по трассе σ.

Сначала рассмотрим случай пустой трассы $\sigma = \langle\rangle$. Тогда $\langle\rangle \in$ ***SafeBy***$(\mathbf{S})$ влечет $s_0 = \langle\gamma\rangle \not\Rightarrow$, что влечет для пополненного соответствия $(i, s_0) \in R`$, то есть $s_0 \in R`(i)$. Но также $s_0 \in (\mathbf{S}$ ***after*** $\langle\rangle)$ и, следовательно, $(\mathbf{S}$ ***after*** $\langle\rangle) \cap R`(i) \neq \varnothing$.

Пусть утверждение верно для трассы σ и докажем его для трассы $\sigma \cdot \langle u\rangle$. Если $i_1 \in (\mathbf{I}$ ***after*** $\sigma \cdot \langle u\rangle)$ и $\sigma \cdot \langle u\rangle \in$ ***SafeBy***$(\mathbf{S})$, то найдется такое состояние $i \in (\mathbf{I}$ ***after*** $\sigma)$ и такая кнопка P ***safe by*** **S** ***after*** σ, что $u \in P \cup \{P\}$ и $i = \langle u\rangle \Rightarrow i_1$. Поскольку $\sigma \cdot \langle u\rangle \in$ ***SafeBy***$(\mathbf{S})$ влечет $\sigma \in$ ***SafeBy***$(\mathbf{S})$, то по предположению шага индукции найдется состояние $s \in (\mathbf{S}$ ***after*** $\sigma) \cap R`(i)$. По конформности соответствия R` найдется такое состояние s_1, что $s = \langle u\rangle \Rightarrow s_1$ и $(i_1, s_1) \in R`$, то есть $s_1 \in R`(i_1)$. Но также $s \in (\mathbf{S}$ ***after*** $\sigma)$ и

$s=\langle u\rangle\Rightarrow s_1$ влечет $s_1\in(\mathbf{S}\ \textbf{\textit{after}}\ \sigma\cdot\langle u\rangle)$. Тем самым, $(\mathbf{S}\ \textbf{\textit{after}}\ \sigma\cdot\langle u\rangle)\cap R`(i_1)\neq\varnothing$, что и требовалось доказать.

Из доказанного вспомогательного утверждения непосредственно следует, что безопасная симуляция с трассовой гипотезой о безопасности влечет трассовую конформность. Действительно, если $\sigma\in\textbf{\textit{SafeBy}}(\mathbf{S})$ и $i\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma)$, то найдется состояние $s\in(\mathbf{S}\ \textbf{\textit{after}}\ \sigma)\cap R`(i)$. Тогда, если $P$ ***safe by*** $\mathbf{S}$ ***after*** $\sigma$, $u\in P\cup\{P\}$ и $i=\langle u\rangle\Rightarrow i`$, то по конформности соответствия $R`$ имеем $s=\langle u\rangle\Rightarrow$, что и требуется для трассовой конформности.

Мы доказали, что ***sst*** $\subseteq$ ***saco***.

Для завершения доказательства леммы заметим, что из трассовой конформности, вообще говоря, не следует симуляция. Пример на 0. □

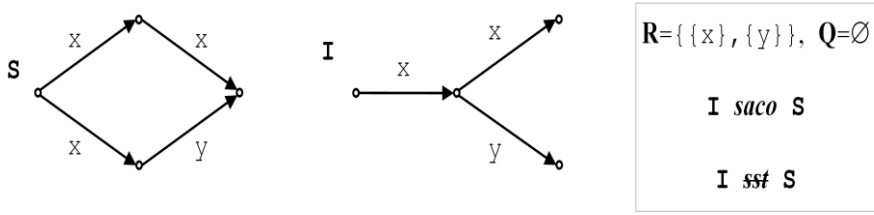


Рис. 8. Пример трассовой конформности при отсутствии симуляции с трассовой гипотезой о безопасности.

Как итог мы имеем следующую теорему, описывающую соотношение трассовой конформности и безопасных симуляций с различными гипотезами о безопасности.

Теорема 2: ss $\subset$ sst $\subset$ saco.

Усиление требований к реализации идет сначала как введение дополнительных требований к соответствию состояний реализации и спецификации для симуляции при сохранении той же гипотезы о безопасности ***sst*** $\subset$ ***saco*** и далее как усиление гипотезы о безопасности ***ss*** $\subset$ ***sst***, что ведет к большему количеству безопасных нажатий кнопок в состояниях и, тем самым, к большему числу проверок.

4. Теоретическое тестирование

4.1. Полнота тестирования

Симуляция основана на соответствии состояний реализации и спецификации. Спецификация должна быть задана и ее состояния мы видим. При тестировании реализация неизвестна, поэтому для проверки симуляции нам

нужна специальная операция опроса текущего состояния реализации (status message [8]). Тестирование с такой тестовой возможностью называется тестированием с открытым состоянием. Операция опроса состояния будет применяться в начале тестирования и после каждого наблюдения[7].

Тест – это инструкция, состоящая из пунктов, описывающих тестовые воздействия и получаемые наблюдения и постсостояния, а также указаний по выполнению следующих пунктов или вынесению вердикта (***pass*** или ***fail***). В начале опрашивается состояние реализации, и для каждого возможного состояния указывается следующий пункт. В каждом другом пункте указывается кнопка, которую нужно нажимать, и для каждого наблюдения и постсостояния, возможных после нажатия этой кнопки, – пункт инструкции, который должен выполняться следующим, или вердикт, если тестирование нужно закончить. В [1,2,3] такая инструкция соответствует формальному определению *управляемого* LTS-теста, который однозначно определяет тестирование (без лишнего недетерминизма).

Реализация *проходит* тест, если ее тестирование всегда (то есть при любом проявлении недетерминизма реализации) не заканчивается с вердиктом ***fail***. Реализация проходит набор тестов, если она проходит каждый тест из набора. Набор тестов *значимый*, если каждая конформная реализация его проходит; *исчерпывающий*, если каждая неконформная реализация его не проходит; *полный*, если он значимый и исчерпывающий. Основная задача заключается в генерации полного набора тестов по спецификации.

На практике требуется, чтобы каждый тест заканчивался за конечное время, а набор тестов был конечным. К сожалению, это возможно только при некоторых ограничениях на семантику, спецификацию, реализацию и недетерминизм ее выполнения. Иногда требуют конечности времени выполнения теста, но допускают бесконечные наборы тестов. На самом деле это не лучше и не хуже конечного набора тестов с бесконечным выполнением тестов.

Различие между тестом и набором тестов условно, в основном это вопрос удобства построения тестовой системы. Дело в том, что завершение теста и запуск другого (или того же самого) теста эквивалентно выполнению рестарта исследуемой системы в рамках одного теста. Предлагаемые в данной статье алгоритмы тестирования ориентированы на использование одного теста с рестартом системы в любой момент времени.

Прежде всего, для полноты тестирования безопасной симуляции ***ss*** требуется для каждого `H`-достижимого состояния `i` реализации и каждой `H`-безопасной в нем кнопки `P` верифицировать каждое имеющееся в

[7] Отказ возникает в стабильном состоянии. Поэтому, если после наблюдения отказа (когда реализация гарантированно оказалась в стабильном состоянии) снова наблюдается отказ (быть может, другой), то состояние можно не опрашивать, поскольку оно не изменяется.

реализации наблюдение $u\in P\cup\{P\}$ и постсостояние $i`$, то есть маршрут $i=\langle u\rangle\Rightarrow i`$. Для этого предполагается, что такой маршрут можно получить через конечное число нажатий кнопки P в состоянии i. Для того, чтобы можно было попасть из начального состояния реализации в каждое H-достижимое состояние, мы должны предположить, что некоторые маршруты $i=\langle u\rangle\Rightarrow i`$ получаются достаточное число раз (и каждый маршрут – за конечное время). Это предположение называется гипотезой о *глобальном тестировании* [6]. Без каких-либо ограничений на класс реализаций глобальное тестирование, очевидно, требует наблюдения каждого маршрута $i=\langle u\rangle\Rightarrow i`$ бесконечное число раз при бесконечной последовательности нажатий кнопки P в состоянии i. В дальнейшем будем считать, что гипотеза о глобальном тестировании выполняется.

4.2. Дерево вывода неконформности

Для конформности, основанной только на трассах наблюдений и не использующей соответствие состояний, доказано существование полного тестового набора для любой спецификации [3]. Для безопасной симуляции такой полный тестовый набор существует уже не всегда. Исследуем эту проблему.

Если $s_0=\langle\gamma\rangle\Rightarrow$, то все реализации конформны, и тестирование не требуется.

Поэтому далее будем считать, что $s_0=\langle\gamma\rangle\not\Rightarrow$. Рассмотрим минимальное множество N пар состояний (i,s), которое порождается следующими правилами вывода: $\forall(i,s)\in H\ \forall P$ H-***safe*** $i\ \forall u\in P\cup\{P\}$

1. $i=\langle u\rangle\Rightarrow$ & P ***safe*** s & $s=\langle u\rangle\not\Rightarrow$ $\vdash (i,s)\in N$,
2. $i=\langle u\rangle\Rightarrow i`$ & $P$ ***safe*** $s$ & $\{i`\}\times(s$ ***after*** $\langle u\rangle)\subseteq N\vdash (i,s)\in N$.

Пары из N будем называть неконформными.

Лемма 18: **I *ss* S** $\Leftrightarrow (i_0,s_0)\notin N$.

Доказательство. Очевидно, что неконформная пара не может принадлежать никакому конформному соответствию. Если $(i_0,s_0)\in N$, то конформного соответствия не существует, так как оно должно было бы содержать (i_0,s_0). Обратно, если $(i_0,s_0)\notin N$, то соответствие $H\setminus N$, очевидно, конформно. □

Заметим, что, если $i_0\Rightarrow i_0`$, то условие $(i_0`,s_0)\in N$, очевидно, влечет $(i_0,s_0)\in N$. Поэтому нам достаточно проверить условие $(i_0`,s_0)\in N$ для любого состояния $i_0`\in(i_0$ ***after*** $\langle\rangle)$.

Правила вывода для N определяют граф вывода. Вершинами этого графа являются пары $(i,s)\in N$. Если пара (i,s) получена применением 1-го

правила вывода, соответствующую вершину будем называть вершиной 1-го типа, в противном случае (применением только 2-го правила вывода) – вершиной 2-го типа. Помеченная дуга $(i,s)\text{—}(u,i`)\rightarrow(i`,s`)$ соответствует применению второго правила вывода для $i,u,i`,s$ и $s`\in(s\ \textbf{\textit{after}}\ \langle u\rangle)$. Набор таких дуг для всех $s`\in(s\ \textbf{\textit{after}}\ \langle u\rangle)$ проводится тогда и только тогда, когда

$i=\langle u\rangle\Rightarrow i`\ \&\ P\ \textbf{\textit{safe}}\ s\ \&\ \{i`\}\times(s\ \textbf{\textit{after}}\ \langle u\rangle)\subseteq N$.

Для каждой пары $(i,s)\in N$ в графе вывода существует (быть может, не единственное) дерево маршрутов, которое мы будем называть деревом вывода. Каждый маршрут, принадлежащий дереву, начинается в (i,s). Корень дерева – пустой маршрут. Листья дерева – маршруты, заканчивающиеся в вершинах 1-го типа. Каждый маршрут, принадлежащий дереву и заканчивающийся в графе вывода в вершине 2-го типа, продолжается в дереве теми и только теми дугами, которые соответствуют одному применению 2-го правила вывода, то есть помеченными одной и той же меткой $(u,i`)$.

Лемма 19: Дерево вывода конечно тогда и только тогда, когда оно имеет конечное ветвление: каждый маршрут продолжается в дереве конечным числом дуг, что эквивалентно конечности каждого множества $s\ \textbf{\textit{after}}\ \langle u\rangle$, участвующего в построении этого дерева.

Доказательство. Количество дуг, которыми продолжается маршрут в дереве вывода, по построению определяется применением 2-ого правила вывода и совпадает с количеством состояний в соответствующем множестве состояний $s\ \textbf{\textit{after}}\ \langle u\rangle$. Если дерево конечно, то, очевидно, оно имеет конечное ветвление и все эти множества состояний конечны. Обратно, если все эти множества состояний конечны, то дерево имеет конечное ветвление. А тогда, поскольку в дереве нет бесконечных маршрутов, оно конечно (по теореме Кёнига [7]). □

4.3. Общий алгоритм тестирования

Если $(i_0,s_0)\in N$, то для (i_0,s_0) существует (быть может, не единственное) дерево вывода. Опишем алгоритм тестирования, определяющий неконформность любой H-безопасной реализации, для которой хотя бы одно из таких деревьев конечно. Этот алгоритм будем называть *общим* алгоритмом тестирования.

Сначала рассмотрим необходимые для работы теста ограничения. Кроме глобального тестирования, нам требуется выполнение следующих условий:

1) начальное состояние спецификации s_0 известно;

2) перечислимо множество спецификационных состояний, достижимых из начального состояния по пустой трассе $S_0=(s_0\ \textit{after}\ \langle\rangle)$, и имеется итератор этого множества;
3) для каждого спецификационного состояния s перечислимо множество безопасных кнопок $P(s)=\{P\in\mathbf{R}\cup\mathbf{Q}\mid P\ \textit{safe}\ s\}$, и имеется итератор этого множества;
4) для каждого спецификационного состояния s и каждого безопасного в нем наблюдения u перечислимо множество постсостояний маршрутов с трассой $\langle u\rangle$: $S(s,u)=(s\ \textit{after}\ \langle u\rangle)$, и имеется итератор этого множества.

В каждый момент времени работы алгоритма будет построена некоторая LTS $\mathbf{I}`$, являющаяся представлением некоторой (конечной) части реализации $\mathbf{I}$. Множеством состояний $\mathbf{I}`$ будет конечное множество $V_{\mathbf{I}}`\subseteq V_{\mathbf{I}}$ «пройденных» реализационных состояний. Переход $i\text{—}u\rightarrow i`$ добавляется тогда, когда обнаруживается, что реализация «прошла» маршрут $i=\langle u\rangle\Rightarrow i`$.[8] Вместе с каждым переходом $i\text{—}u\rightarrow i`$ будем хранить *управляющую* кнопку $P(i\text{—}u\rightarrow i`)$, то есть кнопку, нажатие которой вызвало появление этого перехода. По мере построения $\mathbf{I}`$ будет постепенно формироваться для каждого пройденного состояния i множество $H(i)$ соответствующих ему по H спецификационных состояний. Это множество в каждый момент времени будет конечным, но может постепенно расти.

Тест состоит из управляющего алгоритма и нескольких итераторов перечислимых множеств. Набор итераторов динамический и в каждый момент времени конечный: какие-то итераторы появляются и работают постоянно, а некоторые из появившихся могут потом исчезнуть. Одни из итерируемых множеств могут быть бесконечными, но они не меняются в процессе работы, другие множества конечны, но могут увеличиваться. Управляющий алгоритм вызывает работающие итераторы по циклу. Мы будем описывать эти перечислимые множества и их итераторы по мере описания алгоритма тестирования.

---

[8] Мы говорим о «представлении части», а не просто о «части» реализации, поскольку $i=\langle u\rangle\Rightarrow i`$ означает лишь существование маршрута с такими пре- и пост-состояниями и с такой трассой, а не сам этот маршрут, имеющийся в реализации, который остается неизвестным и может быть, вообще говоря, не единственным. Этот маршрут мы заменяем в $\mathbf{I}`$ на переход $i\text{—}u\rightarrow i`$. Тем самым, алфавит LTS $\mathbf{I}`$ – это объединение $\mathbf{L}\cup\mathbf{R}$ множества внешних действий и $\mathbf{R}$-кнопок LTS $\mathbf{I}$.

В начале тестирования, а также после рестарта, опрашиваем состояние и получаем не обязательно начальное состояние реализации i_0, а любое состояние $i_0` \in I_0 = (i_0$ ***after*** $\langle\rangle)$. После опроса состояния $i_0`$, оно добавляется во множество I_0 (до начала тестирования пустое) и во множество $V_{\mathbf{I}}`$ и, если до этого было $i_0` \notin V_{\mathbf{I}}`$, то устанавливаем $H(i_0`) := \varnothing$. Запускается[9] итератор $S_0(i_0`)$, который, итерируя множество $S_0$, добавляет состояния из этого множества к $H(i_0`)$. Будем считать, что в первую очередь добавляется состояние s_0. Если множество S_0 конечно, итератор $S_0(i_0`)$ удаляется после завершения итерации.

Каждый раз, когда добавляется новое состояние s к множеству $H(i)$, запускается итератор $P(i,s)$, итерирующий множество $P(s)$. Итератор $P(i,s)$ перечисляет безопасные в s кнопки и для каждой кнопки P запускает итератор тестового воздействия $T(i,s,P)$ для нажатия кнопки P в состоянии i. Если множество $P(s)$ конечно, итератор $P(i,s)$ удаляется после завершения итерации.

Итератор $T(i,s,P)$ работает бесконечно следующим образом. Сначала делается рестарт, после чего по пройденной части $\mathbf{I}`$ ищется маршрут, ведущий из состояния после рестарта $i_0` \in I_0$ в состояние i. Такой маршрут, очевидно, существует и соответствует H-безопасному маршруту реализации. Итератор пытается пройти такой маршрут, нажимая управляющие кнопки. Каждый раз, когда ему не удается пройти нужный маршрут до конца (из-за недетерминизма реализации), выполняется рестарт и попытка повторяется. Глобальное тестирование гарантирует, что каждый переход $i \text{—} u \rightarrow i`$ в $\mathbf{I}`$ может быть получен за конечное время ($\mathbf{I}`$ может при этом расти, то есть появляться новые переходы), и, следовательно, итератор за конечное время попадет в состояние $i$. После этого нажимается кнопка $P$ и получается переход $i \text{—} u \rightarrow i`$, который добавляется в $\mathbf{I}`$ вместе с кнопкой P как управляющей кнопкой. Итератор $T(i,s,P)$ – это единственный итератор, который взаимодействует с реализацией. По построению для H-безопасной реализации такое взаимодействие безопасно.

Когда итератор $T(i,s,P)$ первый раз проходит переход $i \text{—} u \rightarrow i`$, запускается итератор $S(i,s,u,i`)$, который, итерируя множество $S(s,u)$, добавляет постсостояния $s`$ маршрутов $s = \langle u \rangle \Rightarrow s`$ к множеству $H(i`)$ и запускает при необходимости итератор $P(i`,s`)$. Если множество $S(s,u)$ конечно, итератор $S(i,s,u,i`)$, обнаруживая это (то есть завершение итерации), переходит во второй режим работы – режим проверки.

[9] Итератор $S_0(i_0`)$ можно не запускать, если он уже один раз запускался.

В режиме проверки итератор S(i,s,u,i`) проверяет, пусто ли множество S(s,u). Если оно пусто, пара (i,s) добавляется к множеству N по 1-ому правилу вывода, а итератор заканчивает свою работу и удаляется. В противном случае итератор опрашивает по циклу состояния из (конечного) множества S(s,u), проверяя для каждого состояния s`∈S(s,u) условие (i`,s`)∈N. Если это условие выполнено для всех состояний из S(s,u), пара (i,s) добавляется к множеству N по 2-ому правилу вывода, а итератор заканчивает свою работу и удаляется. В любом случае при удалении итератора S(i,s,u,i`) проверяется условие $i \in I_0$ & $s=s_0$. Если оно выполнено, тест заканчивает свою работу с вердиктом ***fail***. Пока итератор не удален, он работает в цикле по указанному алгоритму (после завершения итерации начинает ее сначала).

Итак, при тестировании работают итераторы: $S_0(i_0`)$, P(i,s), T(i,s,P) и S(i,s,u,i`). Число итераторов в каждый момент времени конечно, и они организованы в список, который перебирается по циклу управляющим алгоритмом, вызывающим итераторы по очереди. Однако список итераторов может расти. Поэтому для того, чтобы каждый итератор делал очередной шаг итерации через конечное время после предыдущего шага, новые итераторы добавляются в начало списка.

Из описания общего алгоритма следует, что тест, работающий по этому алгоритму, выносит вердикт ***fail*** только для таких реализаций, в которых для пары (i_0,s_0) существует конечное дерево вывода.

Теорема 3: Тест, работающий по общему алгоритму, является значимым на классе всех H-безопасных реализаций и полным на подклассе реализаций, в которых для пары (i_0,s_0) дерево вывода либо не существует (реализация конформна), либо конечно (реализация неконформна).

4.4. Достаточное условие полноты тестирования

Тест, работающий по общему алгоритму, не является полным на классе всех H-безопасных реализаций. Он работает бесконечно долго и не выносит вердикт ***fail*** для всех конформных реализаций (что правильно), но также (что неправильно) и для тех неконформных реализаций, в которых для пары (i_0,s_0) существуют деревья вывода, но все такие деревья бесконечны.

На 0 показан пример, когда реализация неконформна, но все деревья вывода бесконечны. Здесь **I ~~ss~~ S** для любой **R**/**Q**-семантики (из 22 возможных при алфавите **L**={x,y}), кроме трех семантик {{x}}/{{y}}, {{y}}/{{x}} и ∅/{{x},{y}}, то есть для всех семантик таких, что {x},{y}∈**R** или {x,y}∈**R**∪**Q**. Для этих семантик действия x и y безопасны во всех состояниях реализации и спецификации. Состояние 1 не может соответствовать состоянию s_0, так как $1=\langle y\rangle\Rightarrow$, но $s_0=\langle y\rangle\not\Rightarrow$; но оно не может соответствовать и любому другому состоянию спецификации, так как

$1=\langle x,x,\ldots\rangle\Rightarrow$ для любого числа действий x, а в спецификации такие трассы есть только в состоянии s_0. Через $\mathbf{I}(n)$ обозначим под-LTS реализации $\mathbf{I}$, содержащую состояния $0,1,\ldots,n$ и все переходы между ними. Легко видеть, что $\mathbf{I}(n)$ ***ss*** $\mathbf{S}$ для каждого n ($\mathbf{I}(n)$ совпадает с частью спецификации $\mathbf{S}$, определяемой одним переходом $s_0 \text{—} x \rightarrow s$). Любой тест за конечное время n может исследовать часть реализации, содержащую не более n состояний. Эта часть реализации, очевидно, является частью конформной LTS $\mathbf{I}(n)$, и поэтому тест не может вынести вердикт о неконформности реализации $\mathbf{I}$ на основе этой ее части, то есть через время n. Тем самым никакой тест не может за конечное время вынести вердикт ***fail***, поэтому любой набор тестов не может быть исчерпывающим.

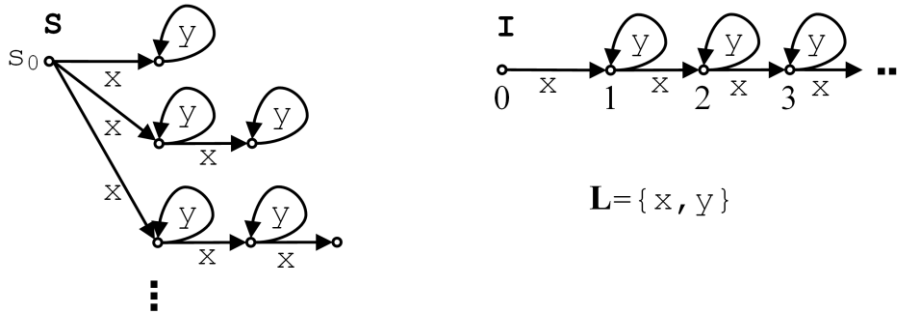


Рис. 9. Пример отсутствия полного набора тестов.

Теперь мы наложим на спецификацию ограничение, которое достаточно для существования полного теста на классе всех H-безопасных реализаций. Это ограничение *локально-конечно-ветвимости* и *τ-ограниченности* спецификации: 1) в каждом безопасно достижимом состоянии спецификации число переходов по каждому безопасному действию конечно, 2) из каждого такого состояния по τ-маршрутам достижимо конечное число состояний.

Теорема 4: На классе локально-конечно-ветвимых и τ-ограниченных спецификаций тест, работающий по общему алгоритму, полон на классе всех H-безопасных реализаций.

Доказательство. По лемме 19 и теореме 3 нам достаточно показать, что в спецификации все множества состояний s ***after*** $\langle u\rangle$, где $u\in P\cup\{P\}$ и P ***safe*** s конечны для всех безопасно-достижимых состояний s. А такие множества всегда конечны для локально-конечно-ветвящейся и τ-ограниченной LTS-спецификации. □

5. Практическое тестирование

5.1. Ограничения

Алгоритм полного тестирования, описанный в предыдущем разделе, для локально-конечно-ветвящихся и τ-ограниченных LTS-спецификаций обнаруживает ошибку за конечное время, но при отсутствии ошибок может выполняться бесконечно долго. Это не приемлемо на практике. В данном разделе мы сформулируем ограничения на семантику, спецификацию и реализацию, которые позволят выполнять полное тестирование за конечное время, и опишем алгоритм тестирования. Эти ограничения аналогичны тем, которые делают полное тестирование конечным для конформности ***saco***, основанной только на трассах наблюдений без соответствия состояний [5]. Сформулируем эти ограничения:

1) *Ограничения на семантику*: число кнопок конечно и задан алгоритм разрешения кнопки относительно всех действий. Наличие такого алгоритма не означает, что все кнопки (как множества действий) конечны (и, тем самым, конечен алфавит действий). Но для конечных кнопок алгоритм всегда существует.
2) *Ограничения на спецификацию*: LTS-спецификация конечна: конечно число состояний и переходов. В этом случае, очевидно, спецификация локально-конечно-ветвящаяся и τ-ограниченная.
3) *Ограничения на реализацию*: реализация (точнее, ее часть, порождаемая `H`-безопасными маршрутами) конечна и *ограниченно недетерминирована*.

Ограниченность недетерминизма – это более сильное предположение, чем глобальное тестирование. При глобальном тестировании требуется, чтобы каждое поведение, разрешаемое кнопкой, могло быть получено за *конечное* время, а при ограниченном недетерминизме – за *ограниченное* время. Это означает, что существует такое число `t` (степень недетерминизма), что в любом состоянии `i` реализации после `t` нажатий любой кнопки `P` будут получены все возможные пары (наблюдение `u∈P∪{P}`, постсостояние `i`). При `t=1` реализация (наблюдаемо) детерминирована.

5.2. Модификация общего алгоритма тестирования

Сначала покажем, что при этих ограничениях можно так модифицировать общий алгоритм тестирования, описанный в предыдущем разделе, чтобы он всегда заканчивался за конечное время. Поскольку при наших ограничениях спецификация локально-конечно-ветвящаяся и τ-ограниченная, общий алгоритм обнаруживает ошибку в любой неконформной реализации, удовлетворяющей `H`-гипотезе, за конечное время. Нужно, чтобы он заканчивал свое выполнение за конечное время и для конформных реализаций.

Теорема 5: При указанных в подразделе 5.1 ограничениях на семантику, спецификацию и реализацию общий алгоритм может быть так модифицирован, что тест, работающий по этому алгоритму, полон.

Доказательство. При указанных ограничениях каждый из итераторов $S_0(i_0`)$ и P(i,s) удаляется, а итератор S(i,s,u,i`) переходит в режим проверки через конечное число шагов. При t-недетерминизме реализации итератор T(i,s,P) модифицируется: теперь он выполняет не более t шагов (тестовое воздействие + получение наблюдения и постсостояния). Заметим, что если наблюдается отказ P в том же состоянии, кнопку P можно нажимать в этом состоянии не t раз, а только один раз: ничего другого, кроме отказа P, мы все равно не получим. При указанных ограничениях число всех возможных итераторов конечно. Поскольку в режиме проверки итератор S(i,s,u,i`) не создает новых итераторов, через конечное число шагов останутся только итераторы S(i,s,u,i`), работающие в режиме проверки. Очевидно, что, если за один цикл вызова всех этих итераторов не было изменения множества N (или, что эквивалентно, ни один итератор не был удален), то это множество уже больше не изменится. Управляющий алгоритм модифицируется так, чтобы в этом случае он заканчивал работу теста с вердиктом ***pass***. □

5.3. Алгоритм обхода реализации

Общий алгоритм выполняет «исследование» реализации и верификацию симуляции параллельно, поскольку для бесконечной реализации нельзя ее сначала исследовать (за конечное время), а потом провести верификацию. Теперь мы опишем частный алгоритм тестирования, работающий только при ограничениях из подраздела 5.1, но зато более быстрый, и дадим оценку его сложности. Будем использовать обозначения, введенные для общего алгоритма. Из конечности семантики и реализации следует конечность LTS **I`**, порождаемой H-безопасными маршрутами реализации **I**. Из конечности спецификации следует конечность всех множеств H(i) для состояний i из **I`**. Поэтому мы можем сначала построить LTS **I`** и множества H(i) (будем называть это *обходом реализации*), а потом провести верификацию симуляции.

С каждым пройденным состоянием i свяжем, кроме множества H(i), которое будет пополняться постепенно, множество $P(i)=\cup\{P(s)\mid s\in H(i)\}$ кнопок безопасных хотя бы в одном состоянии из H(i), а с каждой такой кнопкой P – счетчик c(i,P) числа нажатий кнопки P в состоянии i.

Будем называть кнопку $P\in P(i)$ *полной в состоянии* i, если 1) c(P,i)=1 и в **I`** получен переход $i\text{—}\langle P\rangle\to i$ или 2) c(P,i)=t. Это означает, что уже получены все возможные наблюдения и постсостояния при нажатии

кнопки P в состоянии i. В случае 1 после первого нажатия кнопки P в состоянии i наблюдался отказ P в этом же состоянии, повторное нажатие кнопки P даст тот же отказ. В случае 2 после t нажатий кнопки получены все возможные переходы. Состояние i будем называть *полным*, если каждая кнопка из P(i) полна в нем. Это означает, что все возможные переходы $i—\langle u\rangle\rightarrow i`$, где $u\in\mathbf{L}\cup\mathbf{R}$, уже получены.

Общая схема обхода реализации изображена на 0. В начале тестирования после опроса состояния $i_0`\in I_0$ имеем: $H(i_0`)=S_0$, $P(i_0`)=\cup\{P(s)\mid s\in S_0\}$, $C(i_0`,P)=\varnothing$ для каждой кнопки $P\in P(i_0`)$.

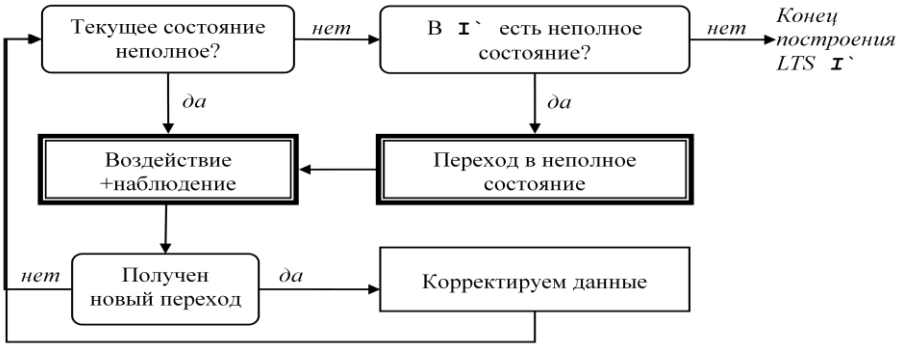


Рис. 10. Схема алгоритма обхода реализации.

Пусть тест находится в некотором текущем состоянии i из **I`**. Если текущее состояние i неполное, то некоторая кнопка $P\in P(i)$ неполна в i. В этом случае осуществляем тестовое воздействие и наблюдение: нажимаем кнопку P и получаем переход $i—\langle u\rangle\rightarrow i`$, постсостояние i` становится новым текущим состоянием. Увеличиваем счетчик c(P,i):=c(P,i)+1. Если получен новый переход $i—\langle u\rangle\rightarrow i`$, то добавляем его в **I`**, запоминаем кнопку P как управляющую кнопку $P(i—u\rightarrow i`)$ этого перехода, и корректируем $H(i`):=H(i`)\cup S(s,u)$ для каждого $s\in H(i)$ при условии P ***safe*** s.

Каждый раз, когда новое состояние s добавляется во множество H(i), корректируем $P(i):=P(i)\cup P(s)$ и для каждого полученного ранее перехода $i—\langle u\rangle\rightarrow i`$ при условии $P(i—\langle u\rangle\rightarrow i`)$ ***safe*** s корректируем $H(i`):=H(i`)\cup S(s,u)$, отмечая вновь добавленные состояния. Эта рекурсивная процедура повторяется до тех пор, пока возможно. В силу конечности реализации и спецификации процедура закончится за конечное число шагов.

Если текущее состояние i полное, то в нем все тестовые воздействия выполнены нужное число раз и получены все возможные переходы в **I`**. Для продолжения тестирования нужно перейти в любое неполное состояние. Если таких состояний нет, построение **I`** заканчивается.

Рассмотрим переход в неполное состояние. В LTS **I`** всегда существует лес деревьев, ориентированных к своим корням, которыми являются все неполные состояния. Этот лес покрывает все состояния. Выберем любой такой лес и для каждого его перехода i—u→i` обозначим связанную с ним управляющую кнопку через A(i)=P(i—u→i`). Будем двигаться, нажимая в каждом текущем состоянии i кнопку A(i). Из-за недетерминизма мы можем оказаться не в состоянии i`, а в другом состоянии i``, где будем нажимать кнопку A(i``).

Обозначим: b – число кнопок,
n – число состояний реализации,
m – число переходов реализации,
k – число состояний спецификации.

Заметим, что m=O(bnt).

Лемма 20: Алгоритм обхода реализации, описанный в данном подразделе, при указанных в 5.1 ограничениях заканчивает свою работу за конечное время и строит LTS **I`**, порождаемую H-безопасными маршрутами реализации **I**, и множества H(i) для всех состояний из **I`**. При этом число тестовых воздействий имеет оценку $O(bt^n)$ для t>1 и $O(bn^2)$ для t=1. Объем вычислений имеет оценку $O(bnt^n)+O(bnm)+O(mk)$ для t>1, для t=1 первое слагаемое заменяется на $O(bn^3)$.

Доказательство. В силу конечности семантики и реализации, а также ограниченности недетерминизма реализации, блок «Воздействие+наблюдение» вызывается конечное число раз. Поскольку каждый цикл в схеме алгоритма содержит блок «Воздействие+наблюдение», любой другой блок алгоритма также вызывается конечное число раз. При вызове каждого блока, кроме перехода в неполное состояние и коррекции данных, выполняется, очевидно, конечное число действий. Каждый переход в неполное состояние также выполняется за конечное число шагов, как показано в [5]. Рекурсивная процедура коррекции данных работает конечное время в силу конечности реализации и спецификации. Тем самым алгоритм заканчивает свою работу за конечное время.

Алгоритм заканчивается, когда все пройденные состояния стали полными. В этом случае, очевидно, все H-достижимые состояния реализации и H-безопасные переходы из них уже получены, то есть построенная LTS **I`** порождена всеми H-безопасными маршрутами реализации. Все множества H(i) для всех состояний из **I`** также построены, поскольку коррекция данных завершена.

Число тестовых воздействий определяется главным образом временем работы блока перехода в неполное состояние. Хотя один такой переход может потребовать $O(t^n)$ для $t>1$ и $O(n)$ для $t=1$ тестовых воздействий, суммарное число тестовых воздействий, как показано в [7], имеет оценку для $t>1$ не $O(nt^n)$, а $O(bt^n)$, и $O(bn^2)$ для $t=1$.

Оценка объема вычислений состоит из трех слагаемых.

1) Число тестовых воздействий умножается на n для оценки числа операций, требуемых для поиска полученного постсостояния среди уже имеющихся состояний. Суммарная оценка: $O(bnt^n)$ для $t>1$ и $O(bn^3)$ для $t=1$.
2) Построение леса деревьев. Лес деревьев строится не более bn раз, так как каждое состояние становится полным только в тот момент времени, когда в нем становится полной некоторая кнопка[10]. Как показано в [7], лес деревьев строится за $O(m)$ операций. Суммарная оценка $O(bnm)$.
3) Коррекция данных при получении нового перехода (рекурсивная процедура). Для каждого перехода $i—u→i`$ и каждого состояния $s∈H(i)$ один раз выполняется следующее корректирующее действие: состояния из $S(s,u)$ добавляются в $H(i`)$ и определяются те из них, которые ранее не принадлежали $H(i`)$. Сложность корректирующего действия зависит от представления данных. Заметим, что множества $S(s,u)$ определяются только спецификацией и могут быть подготовлены заранее до начала тестирования. Корректирующее действие может быть выполнено за $O(1)$ операций, если множества состояний $S(s,u)$ и $H(i`)$ задаются в виде битовых шкал[11]. Число пар $(i—u→i`,s)$ имеет оценку $O(mk)$. Поэтому суммарная оценка равна $O(mk)$.

Итоговая оценка объема вычислений $O(bnt^n)+O(bnm)+O(mk)$ для $t>1$, для $t=1$ первое слагаемое заменяется на $O(bn^3)$. □

5.4. Два алгоритма верификации симуляции

Мы опишем два алгоритма верификации симуляции после построения LTS $\mathbf{I}`$. Каждый из них пытается построить конформное соответствие R, выдавая вердикт ***pass***, если такое соответствие существует и построено, или вердикт

[10] Заметим, что полное состояние i может впоследствии стать неполным, если в $P(i)$ добавляется новая H-безопасная кнопка. Однако это не может случиться после того, как все пройденные состояния стали полными, и коррекция данных завершена.

[11] Если множество $S(s,u)$ задано в виде списка, а множество $H(i`)$ – в виде битовой шкалы, то потребуется $O(k)$ операций. Если оба множества заданы в виде списка, то потребуется $O(k^2)$ операций.

fail, если такого соответствия быть не может. Один из этих алгоритмов строит наибольшее конформное соответствие R_1, а другой – соответствие $R_2=R_1 \cap H$.

Начнем со второго алгоритма. Сначала строится двудольный граф. Вершины первого типа – это пары (i,s), где i – состояние $\mathbf{I}`$, а $s \in H(i)$. Вершины второго типа – это пары $(i \text{—} u \rightarrow i`,s)$, где $i \text{—} u \rightarrow i`$ – переход $\mathbf{I}`$, а $s \in H(i)$. В каждую вершину второго типа входит одна дуга (дуга первого типа) $(i,s) \rightarrow (i \text{—} u \rightarrow i`,s)$. Дуга второго типа $(i \text{—} u \rightarrow i`,s) \rightarrow (i`,s`)$ проводится тогда, когда $s` \in S(s,u)$. Одновременно составляется список терминальных вершин второго типа.

После построения двудольного графа начинается собственно верификация. Каждая терминальная вершина v_2 второго типа удаляется вместе с входящей в нее дугой $v_1 \rightarrow v_2$, начальной вершиной v_1 этой дуги – вершиной первого типа, и каждой входящей в нее дугой $v_1` \rightarrow v_1$. Одновременно для $v_1=(i,s)$ состояние $s$ удаляется из множества $H(i)$. Эти операции повторяются до тех пор, пока не будет удалена одна из вершин первого типа $(i_0`,s_0)$, где $i_0` \in I_0$, или пока не будут удалены все терминальные вершины второго типа. В первом случае алгоритм заканчивается с вердиктом ***fail***, а во втором случае – с вердиктом ***pass***.

Первый алгоритм отличается от второго тем, что при построении двудольного графа мы каждое $H(i)$ делаем равным множеству $V_{\mathbf{s}}$ всех состояний спецификации.

Лемма 21: Алгоритмы верификации, описанные в данном подразделе, при указанных в 5.1 ограничениях заканчивают свою работу за конечное время и выносят вердикт ***fail***, если реализация неконформна, и вердикт ***pass***, если реализация конформна. В последнем случае строится соответствие $R=\{(i,s) \mid i \in V_{\mathbf{I}`} \ \& \ s \in H(i)\}$, которое для первого алгоритма совпадает с R_1, а для первого – с R_2. Объем вычислений имеет оценку $O(mk^2)$.

Доказательство. Поскольку реализация и спецификация конечны, двудольный граф тоже конечен и, следовательно, строится за конечное время. Продолжение верификации связано с удалением некоторых вершин и друг из этого конечного графа, поэтому эта часть каждого из алгоритмов также завершается за конечное время.

По построению этих алгоритмов очевидно, что они выносят правильный вердикт и при вердикте ***pass*** строят требуемое соответствие.

Оценим сложность алгоритмов. Число вершин второго типа, то есть пар $(i \text{—} u \rightarrow i`,s)$, имеет оценку $O(mk)$. Поскольку в каждую вершину второго типа входит одна дуга первого типа, число таких дуг имеет оценку $O(mk)$. Поскольку из каждой вершины второго типа выходит число дуг, не

превышающее число состояний спецификации, число таких дуг имеет оценку $O(mk^2)$. Поэтому суммарно построение двудольного графа требует $O(mk^2)$ операций.

В наихудшем случае верификация требует однократного просмотра всех дуг двудольного графа, то есть имеет оценку $O(mk^2)$ операций. Эта оценка сложности верификации по двудольному графу совпадает с оценкой сложности его построения. □

Последние две леммы дают следующую теорему.

Теорема 6: Тест, основанный на алгоритме обхода реализации из подраздела 5.3 и любом из алгоритмов верификации из данного подраздела, является полным на классе семантик, спецификаций и реализаций, удовлетворяющих ограничениям из подраздела 5.1. Если тест заканчивает свою работу с вердиктом ***pass***, то строится соответствие $R=\{(i,s)\mid i\in V_{\mathbf{I}`} \ \&\ s\in H(i)\}$, которое для первого алгоритма совпадает с R_1, а для второго – с R_2. При этом число тестовых воздействий имеет оценку $O(bt^n)$ для $t>1$ и $O(bn^2)$ для $t=1$. Объем вычислений имеет оценку $O(bnt^n)+O(bnm)+O(mk^2)$ для $t>1$, для $t=1$ первое слагаемое заменяется на $O(bn^3)$.

Учитывая, что $m=O(bnt)$, оценку объема вычислений можно записать в виде $O(bnt^n)+O(b^2n^2t)+O(bntk^2)$ для $t>1$, для $t=1$ первое слагаемое заменяется на $O(bn^3)$.

5.5. Пример верификации симуляции

На 0 приведен пример верификации симуляции. Семантика содержит как **R**-, так и **Q**-кнопки. Спецификация $\mathbf{S}$ демонстрирует все виды опасности: ненаблюдаемый отказ $\{y\}$ (в состоянии 3), дивергенцию (в состоянии 4) и разрушение (в состоянии 5). Спецификация удовлетворяет собственной H-гипотезе[12] и, по лемме 10, конформна сама себе как реализация. Также приведены примеры еще одной конформной реализации $\mathbf{I_1}$ и одной неконформной реализации $\mathbf{I_2}$. После обхода реализаций будут построены LTS $\mathbf{S`}$, $\mathbf{I_1`}$ и $\mathbf{I_2`}$. Приведены двудольные графы для верификации соответствия по второму алгоритму с указанием соответствия H, в которых серым фоном отмечены терминальные вершины второго типа, а пунктиром – удаляемые дуги. Для конформных реализаций приведены соответствия R_2, а также соответствия R_1 без двудольных графов, которые строятся аналогично.

[12] Единственное проявление недетерминизма в спецификации – это два перехода $0 \text{—} y \rightarrow 1$ и $0 \text{—} y \rightarrow 3$. Но в этих состояниях безопасны одни и те же кнопки (кнопка $\{x\}$).

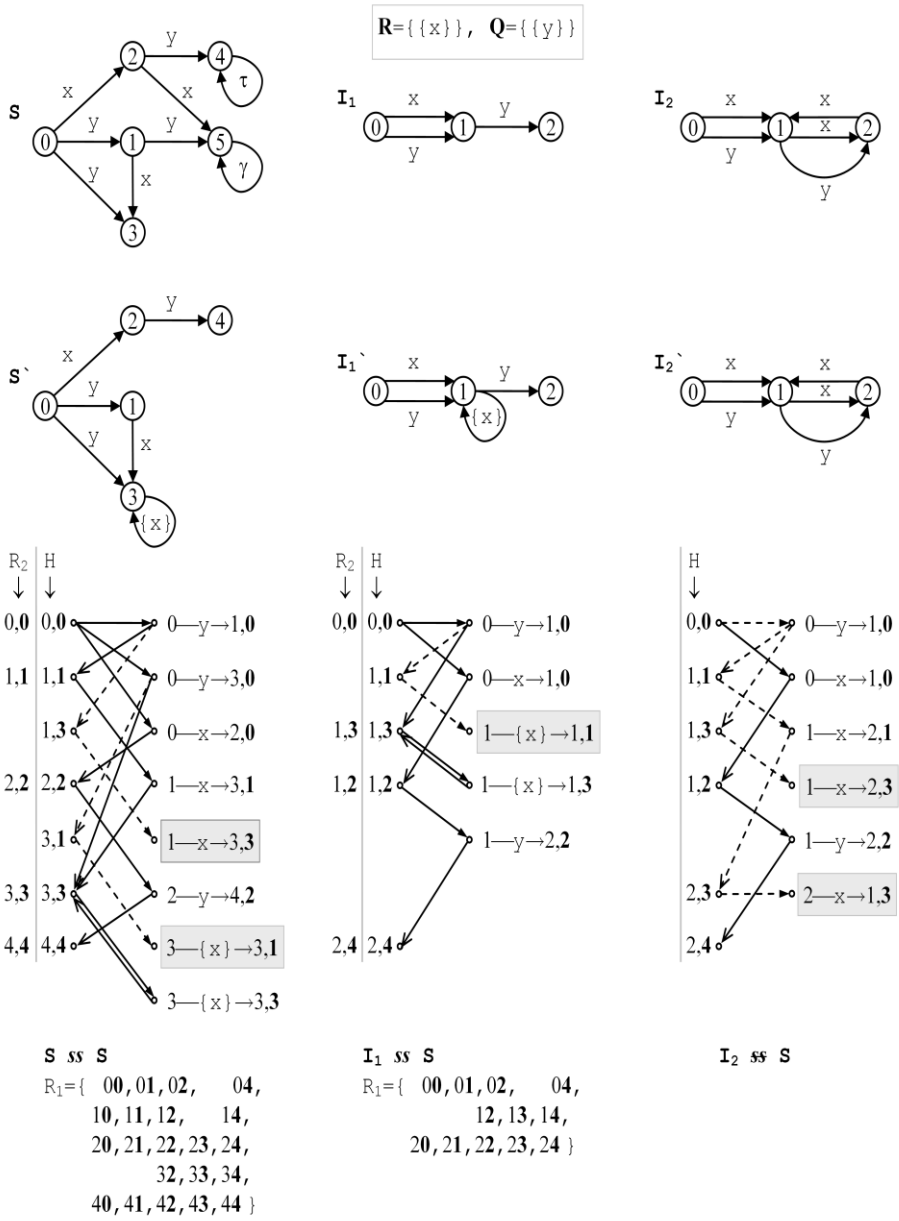


Рис. 11. Пример верификации симуляции

Литература

[1] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, No. 5.

[2] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Теория соответствия для систем с блокировками и разрушением. «Наука», 2008.

[3] Бурдонов И.Б. Теория конформности для функционального тестирования программных систем на основе формальных моделей. Диссертация на соискание ученой степени д.ф.-м.н., Москва, 2008.

[4] http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf

[5] Бурдонов И.Б., Косачев А.С. Полное тестирование с открытым состоянием ограниченно недетерминированных систем. «Программирование», 2009, No. 6.

[6] van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.

[7] König D. Über eine Schlusswaise aus dem Endlichen ins Unendliche. Acta Litt. Ac. Sci. Hung. Fran. Josep. No 3. 1927. pp. 121-130. См. Также Куратовский К., Мостовский А. Теория множеств. М.«Мир», 1970.

[8] Lee D., Yannakakis M. Principles and Methods of Testing Finite State Machines – A Survey. Proceedings of the IEEE 84, No. 8, 1090–1123, 1996.

[9] Milner R. Lectures on a calculus for communicating systems. Seminar on Concurrency, LNCS 197, Springer-Verlag, pp. 197-220.

[10] Milner R. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor: Handbook of Theoretical Computer Science, chapter 19, Elsevier Science Publishers B.V. (North_Holland), pp. 1201-1242. Alternatively see Communication and Concurrency, Prentice-Hall International, Englewood Cliffs, 1989, of which an earlier version appeared as A Calculus of Communicating Systems, LNCS 92, Springer-Verlag, 1980.

Прозрачный механизм удаленного обслуживания системных вызовов

Яковенко П.Н.
yak@ispras.ru

Аннотация. В статье рассматривается подход к удаленному обслуживанию системных вызовов, не требующий модификации кода пользовательского приложения и операционной системы. Использование технологии аппаратной виртуализации для перехвата системных вызовов, чтения их параметров и записи результатов позволяет делегировать обслуживание перехваченных системных вызовов другой системе: виртуальной, выполняющейся на этом же физическом компьютере, или даже другой машине в сети. Возможность предоставлять отдельным процессам контролируемый доступ к ресурсам, к которым сама операционная система доступа не имеет, позволяет эффективно решать некоторые задачи компьютерной безопасности.

1. Введение

Решение некоторых задач компьютерной безопасности основано на ограничении доступа программного кода к различного рода ресурсам, в частности сетевым ресурсам и файловой системе. Предоставив доступ к этим ресурсам только отдельным доверенным программам, можно гарантировать (при дополнительном контроле целостности этих программ) выполнение важных требований по безопасности, среди которых следует отметить предотвращение утечки критически важной информации через различные каналы передачи данных: сетевое соединение, переносные (USB) накопители и др.

Современные операционные системы (ОС) предоставляют широкие возможности по управлению доступом процессов к ресурсам компьютера. Однако, недостаточная надежность массовых ОС (таких как Windows, Linux и др.) [1] делает задачу разработки независимых от ОС (ортогональных к ней) программных средств защиты информации актуальной. Такие средства защиты могут быть реализованы с использованием технологии аппаратной виртуализации, когда защищаемая система выполняется в аппаратной виртуальной машине (ВМ), а система защиты размещается в теле монитора виртуальных машин (также называемого гипервизором) [2,4]. Функционирование гипервизора на более высоком аппаратном уровне привилегий позволяет полностью контролировать выполнение, как кода ОС,

так и пользовательских программ, оставаясь при этом аппаратно защищенным от вредоносного воздействия со стороны кода в ВМ, в т.ч. кода, выполняющегося в привилегированном режиме.

Пользовательский процесс в современных ОС не имеет прямого доступа к аппаратным ресурсам; операционная система представляет процессу некоторую абстрактную модель аппаратного обеспечения, взаимодействие с которой осуществляется посредством набора операций – системных вызовов (СВ). В частности, для установления сетевого соединения с удаленным компьютером и передачи ему данных процессу необходимо выполнить в заданном порядке несколько вполне определенных системных вызовов (*socket, connect, send* и т.п.).

В наших работах [2,3] было показано, как при помощи разделения полномочий по обслуживанию ресурсов между виртуальными машинами и делегирования обслуживания системных вызовов от одной виртуальной машины другой могут быть решены некоторые задачи компьютерной безопасности. Предложенные решения в целом базируются на схеме, изображенной на рисунке 1. Гипервизор обеспечивает одновременное выполнение двух, изолированных друг от друга, виртуальных машин. Обе ВМ работают под управлением одинаковой операционной системы. Первая ВМ – *вычислительная* – является основной. Пользователь может работать с ней в диалоговом режиме.

Оборудование, через которое осуществляется доступ к контролируемым ресурсам (ресурсы сети Интернет на рис. 1), физически отключается гипервизором от вычислительной ВМ. Это оборудование управляется другой – *сервисной* – виртуальной машиной, которая, вообще говоря, может выполняться скрытым для пользователя образом в фоновом режиме. Системные вызовы отдельных (доверенных) процессов перехватываются гипервизором и те из них, которые относятся к контролируемым ресурсам, передаются на обслуживание (обслуживаются удаленно) в сервисную ВМ. Заметим, что обслуживание этих вызовов внутри вычислительной ВМ неминуемо приведет к ошибке в силу отсутствия у нее возможностей (оборудования) осуществить доступ к соответствующим ресурсам. Остальные системные вызовы доверенных процессов, а также все системные вызовы других процессов, обслуживаются средствами ОС в вычислительной ВМ.

При перехвате системного вызова гипервизор может проконтролировать допустимость контекста, из которого выполняется запрос на системный вызов, т.е. разрешен ли данному процессу доступ к такой категории ресурсов, и выполнять удаленное обслуживание вызова только в случае успешного прохождения проверки. Анализируя параметры вызова, гипервизор может также осуществлять более тонкий контроль доступа к ресурсу, например, разрешать сетевой доступ только к ограниченному числу компьютеров в сети, имеющих заданные адреса.

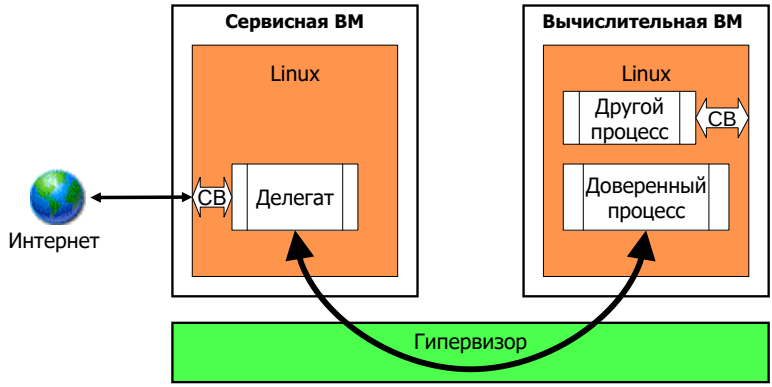


Рис. 1. Делегирование системных вызовов другой виртуальной машине.

В данной работе описывается архитектура и детали реализации механизма удаленного выполнения системных вызовов, используемого в системе безопасности, которая представлена в работе [3]. В ней контролируемыми ресурсами являются исключительно ресурсы сети Интернет. Как следствие, реализация, рассматриваемая в данной работе, допускает удаленное обслуживание только тех системных вызовов, которые могут быть использованы при сетевом взаимодействии через сокеты. Однако, учитывая общность представления сокетов и файлов для пользовательского процесса в виде дескрипторов файлов и, как следствие, общность многих системных вызовов, рассматриваемый механизм в целом пригоден также для тех сценариев, когда контролируемым ресурсом является файловая система.

Отдельные компоненты системы, рассматриваемой в данной работе, могут также использоваться для эффективного решения тех задач, в которых требуется выполнять трассировку системных вызовов. Дело в том, что штатные механизмы трассировки в ОС Linux (*ptrace*) базируются на сигналах, посылаемых ядром ОС процессу-монитору (отладчику) при выполнении отлаживаемым процессом системного вызова, что приводит к частому переключению контекста процессов (монитора и трассируемого процесса). Кроме того, чтение данных из адресного пространства трассируемого процесса возможно только порциями по 4 байта. Размещение монитора процессов на уровне гипервизора позволяет устранить указанные ограничения.

Задача удаленного выполнения системных вызовов рассматривается также в рамках проекта VirtualSquare [6]. Отличие предлагаемого нами подхода состоит в использовании технологии аппаратной виртуализации для перехвата системных вызовов. Кроме того, в работе [6] подмножество системных вызовов, исполняемых удаленно, задается жестко, в то время как в нашем подходе решение об удаленном выполнении системного вызова принимается

исходя из дескриптора ресурса, к которому обращается процесс. В ряде случаев системный вызов надо одновременно выполнять в обеих системах (локальной и удаленной) с последующим объединением результатов, и этот вопрос также рассмотрен в данной работе.

Задача удаленного выполнения системных вызовов близка задаче удаленного выполнения процедур [7], широко применяемой в распределенных программных системах, в частности, в сетевой файловой системе NFS [8]. Принципиальное отличие предлагаемого нами решения состоит в отсутствии необходимости модификации (в т.ч. перекомпиляции) кода приложения и операционной системы для обслуживания системных вызовов в другой системе.

Статья организована следующим образом. В разделе 2 представлен краткий обзор технологии аппаратной виртуализации и изложена общая архитектура системы и ее компонент. В разделе 3 детально рассмотрены принципы работы системы и обработка различных сценариев доступа к ресурсам. В разделе 4 представлены результаты анализа производительности системы как на синтетических тестах, так и на реальных приложениях. В разделе 5 подводятся итоги работы.

2. Архитектура системы

Технология виртуализации позволяет выполнять ОС в аппаратной виртуальной машине (ВМ) под управлением сравнительно небольшой по размеру системной программы – монитора виртуальных машин (гипервизора) [5]. Аппаратная виртуальная машина представляет собой эффективный изолированный дубликат реальной машины, и выполнение в ней операционной системы не требует внесения каких-либо изменений в код ОС. Гипервизор полностью контролирует взаимодействие ОС в ВМ с оборудованием и может обеспечить надежную изоляцию ВМ, опираясь на аппаратные механизмы защиты.

Функционирование гипервизора и организация выполнения виртуальной машины во многом схожи с тем, как операционная система управляет выполнением пользовательских процессов. Гипервизор инициализирует системные структуры данных, необходимые оборудованию, и выполняет специальную инструкцию VMRUN реализующую запуск ВМ и передачу управления соответствующей инструкции кода ОС. При возникновении аппаратного события (прерывания) или при выполнении операционной системой привилегированной инструкции (в т.ч. системного вызова) выполнение ВМ прерывается, и управление передается гипервизору на следующую инструкцию после VMRUN. После обработки перехваченного события гипервизор возобновляет выполнение ВМ.

Архитектура системы выглядит следующим образом. Гипервизор реализует одновременное выполнение двух виртуальных машин (Рис. 1), работающих

под управлением одинаковой операционной системы Linux. Отметим, что операционные системы в виртуальных машинах могут быть разными, если гипервизор при этом обеспечивает необходимое преобразование системных вызовов между исходной и целевой операционными системами. В вычислительной ВМ выполняются процессы пользователя, среди которых выделен набор *доверенных* процессов. Гипервизор предоставляет доверенным процессам привилегию доступа к ресурсам, обслуживаемым ОС в сервисной ВМ (контролируемым ресурсам); другие процессы доступа к этим ресурсам не имеют. Доступ доверенных процессов к контролируемым ресурсам производится посредством перехвата их системных вызовов и, при необходимости, их перенаправления для обслуживания в сервисную ВМ.

Обе виртуальные машины выполняются асинхронно, и вычислительная ВМ не блокируется на время удаленного обслуживания системного вызова. В это время процесс, инициировавший системный вызов, находится в состоянии ожидания, а все остальные процессы в вычислительной ВМ продолжают выполняться нормальным образом. Более того, система допускает одновременное обслуживание нескольких удаленных системных вызовов, поступающих от разных доверенных процессов.

При перехвате запроса процесса на выполнение системного вызова гипервизор копирует все входные параметры вызова в собственную область памяти и передает запрос на обслуживание вызова в сервисную ВМ. Системный вызов может быть блокирующим (например, *read*), и время его выполнения в общем случае не ограничено. Для избегания блокировки всей вычислительной ВМ на время обслуживания вызова процесс, инициировавший системный вызов, переводится в состояние ожидания, позволяя другим процессам продолжить свое выполнение. При поступлении из сервисной ВМ результатов обслуживания вызова гипервизор прерывает выполнение вычислительной ВМ, выводит процесс из состояния ожидания и копирует результаты в его адресное пространство. Пребывание процесса в состоянии ожидания реализуется штатными средствами ОС в ВМ, не вмешиваясь в работу механизма управления процессами в операционной системе.

Внутри сервисной ВМ выполняется набор процессов – *делегатов*, являющихся экземплярами специализированной программы. Каждый из делегатов обслуживает запросы на системные вызовы отдельного доверенного процесса из вычислительной ВМ, причем иерархия делегатов в сервисной ВМ соответствует иерархии доверенных процессов в вычислительной ВМ. Новый делегат порождается каждый раз при создании нового доверенного процесса и уничтожается при завершении работы этого процесса.

Выполнение делегата состоит в циклическом исполнении системных вызовов, поступивших от соответствующего доверенного процесса, и извещении гипервизора о результатах (делегат, в отличие от доверенного процесса, знает о существовании гипервизора). Делегаты получают запрос на выполнение

системного вызова от специализированного процесса – *диспетчера* – через механизм межпроцессного взаимодействия (очередь сообщений). Все запросы, поступающие от гипервизора, проходят через процесс-диспетчер. Диспетчер отслеживает соответствия между идентификаторами доверенных процессов и идентификаторами делегатов и, получив запрос от гипервизора на обслуживание системного вызова некоторого доверенного процесса, перенаправляет его соответствующему делегату. Дальнейшее обслуживание запроса целиком производится делегатом без участия диспетчера. В частности, делегат самостоятельно выполняет доступ к хранилищу, извещает гипервизор о результатах системного вызова и т.д.

Иерархия каждого доверенного процесса восходит к служебному процессу, являющемуся экземпляром специальной пользовательской программы – *монитора*. Первый доверенный процесс всегда порождается монитором. Монитор, в свою очередь, не является доверенной программой. Задача монитора состоит в запуске нового (в т.ч. первого) доверенного процесса и отслеживании его состояния, а также состояния всех его дочерних процессов, часть из которых могут быть доверенными, а часть – нет.

Монитор реализован на базе стандартного интерфейса отладки *ptrace*. Монитор перехватывает события порождения и завершения процессов, в т.ч. аварийного, например, при получении процессом сигнала, для которого у него не зарегистрирован обработчик. При выполнении одним из дочерних процессов системного вызова *fork* или *exec* монитор определяет, требуется ли данному процессу выполнение в доверенном режиме (т.е. будет ли новый процесс доверенным), и, в случае необходимости, запрашивает гипервизор о его включении для процесса. При завершении доверенного процесса монитор также извещает об этом гипервизор.

```
[STARTER]
/usr/local/bin/scp
[ARGUMENTS]
myfile user@host:myfile
[TRUSTED_OBJECTS]
444:/usr/local/bin/ssh
444:ssh
[CANCEL_HANDLER]
444:0x8049af3
```

Рис. 2. Паспорт доверенной задачи.

Монитор определяет, для каких процессов запрашивать доверенный режим выполнения, основываясь на специальном конфигурационном файле – *паспорте* задачи (Рис. 2). Паспорт содержит имя изначально запускаемой программы (не обязательно доверенной) и список передаваемых ей параметров командной строки. Основная часть паспорта состоит из набора

шаблонов для идентификации новых процессов, для которых следует запрашивать включение доверенного режима. Для каждого шаблона указывается уникальный идентификатор доверенной программы, зарегистрированной в гипервизоре. В гипервизоре для каждой зарегистрированной программы перечислен набор хэш-кодов, позволяющих проверить, что запускаемая программа действительно является одной из доверенных [1].

При выполнении дочерним процессом системного вызова *exec* монитор производит поиск шаблона, который может быть сопоставлен имени запускаемой программы, и, в случае успеха, делает запрос гипервизору на включение доверенного режима, сообщая ему идентификатор процесса (PID) и идентификатор соответствующей доверенной программы (например, 444 на рис. 2). Гипервизор проверяет допустимость включения доверенного режима для процесса в данной точке выполнения и, в случае соблюдения контекстных условий безопасности [1], активирует доверенный режим. При перехвате системного вызова *fork* монитор активирует доверенный режим для дочернего процесса только в том случае, если родительский процесс выполняется в доверенном режиме. Следует отметить, что гипервизор контролирует корректность выполнения запроса, т.е. что родительский процесс выполняется в доверенном режиме и действительно выполнил системный вызов *fork*.

Паспорт задачи также содержит адрес произвольной RET инструкции в коде доверенной программы. Монитор при помощи модуля (расширения) ядра в вычислительной ВМ регистрирует для доверенного процесса обработчик сигнала, не используемого процессом, по этому адресу. Посылка такого сигнала доверенному процессу приведет просто к выполнению инструкции RET. Этот сигнал используется для отмены выполнения системного вызова в вычислительной ВМ в тех случаях, когда для корректного обслуживания системного вызова его требуется выполнять в обеих виртуальных машинах (см. раздел 3.1).

2.1. Компоненты системы и их взаимодействие

Удаленное обслуживание системных вызовов реализуется гипервизором совместно с другими компонентами системы, функционирующими в обеих ВМ – вычислительной и сервисной. Компоненты функционирует как в пользовательском пространстве (монитор, диспетчер, делегаты), так и в пространстве ядра ОС (загружаемые модули ядра ОС).

В ходе инициализации системы в ядро ОС в вычислительной и сервисной ВМ динамически загружаются модули ядра. Каждый модуль выделяет непрерывное пространство физической памяти (по умолчанию 1 страницу размером 4Кб) для организации кольцевого буфера, регистрирует несколько обработчиков прерываний, при помощи которых гипервизор извещает виртуальную машину о событиях, требующих обработки и сообщает эту информацию (адрес буфера и номера прерываний) гипервизору посредством

гипервызова. В сервисной ВМ также запускается пользовательский процесс – диспетчер.

В ходе удаленного обслуживания системного вызова компоненты системы взаимодействуют между собой, причем механизмы взаимодействия реализованы по-разному (рис. 3). Реализация механизма взаимодействия некоторой пары компонент определяется уровнями привилегий, на которых они выполняются. Любая компонента может обратиться к гипервизору посредством гипервызова. Выполнение ВМ при этом прерывается, и управление передается гипервизору. Синхронный характер этого обращения позволяет передавать параметры гипервызова аналогично тому, как пользовательский процесс передает параметры ядру ОС при выполнении системного вызова: числовые параметры и адреса областей памяти передаются через регистры, при необходимости гипервизор читает область памяти виртуальной машины по указанным адресам и извлекает из нее (или записывает в нее) дополнительную информацию.

Пользовательский процесс (диспетчер или монитор) обращается за сервисом к модулю ядра посредством системных вызовов. Модуль ядра регистрирует в ОС специальное логическое устройство, видимое на уровне файловой системы как файл. Операции доступа к этому файлу (системные вызовы *read*/*write*/*ioctl*) вызывают соответствующие функции в драйвере логического устройства. Драйвер обрабатывает запрос процесса и возвращает управление ему. Если операция блокирующая, то драйвер может приостановить выполнение процесса до тех пор, пока он не сможет обслужить запрос. Модуль ядра обращается к пользовательскому процессу посредством посылки сигналов.

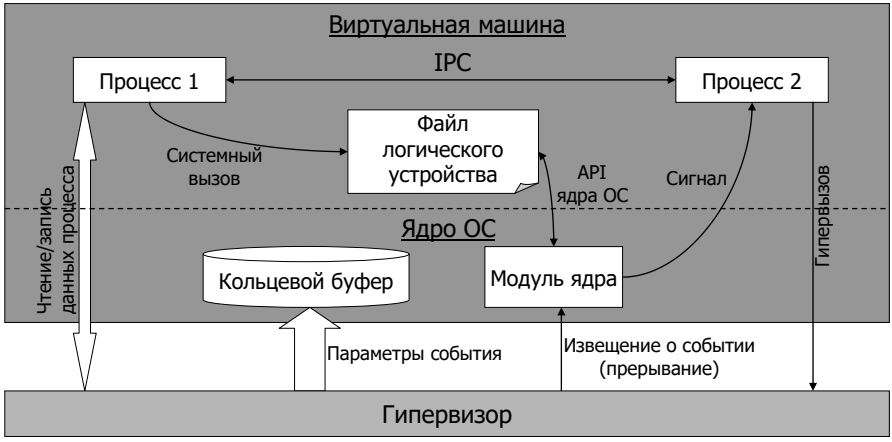


Рис. 3. Взаимодействие компонент системы.

Взаимодействие пользовательских процессов друг с другом (например, диспетчера с делегатами) осуществляется при помощи стандартных механизмов межпроцессного взаимодействия (IPC) ОС Linux – разделяемой памяти, очереди сообщений и пр. Все делегаты в сервисной ВМ являются членами одной иерархии процессов, восходящей к диспетчеру, что облегчает контроль создания разделяемых между процессами ресурсов: ресурс, создаваемый родителем, доступен потомку, а ресурс, создаваемый потомком, не доступен родителю.

Наиболее сложной является ситуация, в которой гипервизору требуется известить компоненты в виртуальной машине о некотором событии. Для этого гипервизор использует возможность, предоставляемую аппаратурой виртуализации, вбрасывать прерывания и исключительные ситуации в виртуальную машину посредством соответствующих полей в управляющей VMCB структуре ВМ. Тогда после возобновления ВМ аппаратура обеспечивает ей доставку прерывания непосредственно перед выполнением первой инструкции в ВМ. В результате вброса прерывания ОС передает управление на обработчик (вектор) данного прерывания, зарегистрированный модулем ядра в таблице обработчиков прерываний в процессе инициализации системы.

Параметры события передаются через кольцевой буфер. Буфер физически расположен в области памяти ВМ и разделяется между гипервизором и ВМ по схеме «поставщик - потребитель». Буфер представляет собой замкнутый в кольцо массив структур данных (фиксированного размера), голова которого сдвигается по мере выемки запросов из буфера, а хвост – по мере помещения запросов в буфер. Если буфер переполнен, то доставка запроса откладывается до тех пор пока в буфере не освободится место, т.е. пока ОС не обработает хотя бы одно из ранее сгенерированных событий. Запросы, ожидающие доставки в ВМ, накапливаются в очереди в памяти гипервизора.

Структура данных, представляющая собой элемент кольцевого буфера, едина для всех событий и включает поля для всех возможных параметров фиксированной длины. Параметры переменной длины передаются через отдельный буфер переменного размера, расположенный в памяти гипервизора – *хранилище*. Координаты параметра переменной длины – смещение от начала хранилища и длина, – специфицируются в структуре данных кольцевого буфера. Например, для системного вызова *write* структура включает 3 поля: идентификатор файлового дескриптора, начало (смещение) буфера в хранилище и длина буфера. Для каждого доверенного процесса гипервизор поддерживает отдельный экземпляр хранилища.

При получении запроса, содержащего параметры переменной длины, код в ВМ, которому предназначается этот запрос (например, делегат) выполняет гипервызов на доступ к хранилищу, передавая координаты запрашиваемого параметра и адрес буфера в собственной памяти, в который должны быть записаны данные из хранилища. Гипервизор обслуживает запрос и

возобновляет выполнение ВМ. При этом он контролирует, что границы запрашиваемого блока данных не выходят за пределы хранилища. Доступ к хранилищу возможен как по чтению, так и по записи.

При необходимости передать запрос одной из компонент внутри ВМ гипервизор ожидает, когда выполнение ВМ будет прервано по тому или иному событию (например, по таймеру) и анализирует, может ли он послать запрос в данной точке. Для этого в кольцевом буфере должно быть свободное место, а прерывания не должны быть маскированы в ВМ. Если это так, то гипервизор (при необходимости) заполняет хранилище параметрами переменной длины, формирует структура данных для кольцевого буфера, указывая в ней координаты параметров в хранилище, записывает сформированный запрос в буфер и вбрасывает прерывание. Вброс прерывания передает управление обработчику прерывания, который анализирует содержимое буфера и либо обслуживает его самостоятельно, либо передает запрос другой компоненте, отвечающей за обслуживание прерывания (например, диспетчеру).

Если получателем запроса гипервизора является пользовательский процесс (например, диспетчер), то доставка такого запроса производится транзитом через драйвер логического устройства. Пользовательский процесс обращается к файлу устройства и, в случае отсутствия запроса на данный момент, переходит в состояние ожидания. При поступлении запроса от гипервизора обработчик прерывания извещает об этом драйвер устройства. Тот, в свою очередь, читает запрос из кольцевого буфера, копирует его в память процесса и выводит его из состояния ожидания. Если запрос содержит параметры переменной длины, то доступ к хранилищу осуществляется тем пользовательским процессом, который непосредственно обслуживает запрос.

3. Прозрачное обслуживание системных вызовов

Механизм системных вызовов в процессорах семейства x86 может быть реализован различными способами. Исторически для этого использовались программные прерывания (инструкция INTn), в частности, ОС Linux использует 128 вектор для выполнения системного вызова. Возврат из системного вызова производился при помощи инструкции IRET. При выполнении инструкций INTn и IRET процессор проводит ряд проверок контекста выполнения инструкции и ее параметров. Частое выполнение процессом системных вызовов может оказывать существенное влияние на производительность системы. Как решение, производители процессоров предложили дополнительную пару инструкций специально предназначенную для быстрого перехода в режим ядра на заданный адрес и обратно: SYENTER/SYSEXIT от Intel и SYSCALL/SYSRET от AMD. Использование этих инструкций является предпочтительным, однако оригинальный механизм выполнения системных вызовов на базе программных прерываний по

прежнему поддерживается из соображений обратной совместимости приложений.

Рассматриваемая в этой работе система удаленного обслуживания системных вызовов требует, чтобы доверенные программы использовали механизм программных прерываний для выполнения системных вызовов. Это обусловлено возможностью перехвата инструкции программного прерывания и возврата из прерывания непосредственно при помощи аппаратуры виртуализации. Остальные процессы в вычислительной ВМ могут использовать произвольные механизмы системных вызовов. Из соображений повышения производительности перехват программного прерывания (инструкция INTn) устанавливается, только когда управление передается доверенному процессу, что позволяет предотвратить выход из ВМ, если эта инструкция выполнялась любым другим (недоверенным) процессом.

Флаг перехвата инструкции IRET, в свою очередь, устанавливается при каждом возврате управления ВМ, если в системе выполняется хотя бы один доверенный процесс. Лишь перехватывая все такие инструкции, гипервизор может отследить момент, когда ОС передает управление доверенному процессу. Это необходимо для корректного возобновления процесса после получения результатов из сервисной ВМ. Если результаты готовы и возврат управления происходит на следующую инструкцию после запроса на системный вызов, то гипервизор записывает результаты, полученные из сервисной ВМ, на регистры и в память процесса, и процесс продолжает выполнение.

При перехвате программного прерывания гипервизор проверяет, что оно было выполнено из контекста доверенного процесса, и что вектор прерывания соответствует вектору запросов на обслуживание системных вызовов (128 в ОС Linux). Далее гипервизор проверяет, требуется ли обслуживать данный системный вызов удаленно в сервисной ВМ или локально в вычислительной ВМ. Правила такого анализа системных вызовов будут рассмотрены в следующем разделе. Если вызов локальный, то гипервизор просто возобновляет управление ВМ, передавая управление ядру ОС. Если вызов удаленный, то гипервизор копирует параметры системного вызова из регистров и, возможно, из адресного пространства процесса в собственную область памяти, формирует запрос и отправляет его в сервисную ВМ. Схема механизма прозрачного обслуживания системных вызовов приведена на рисунке 4.

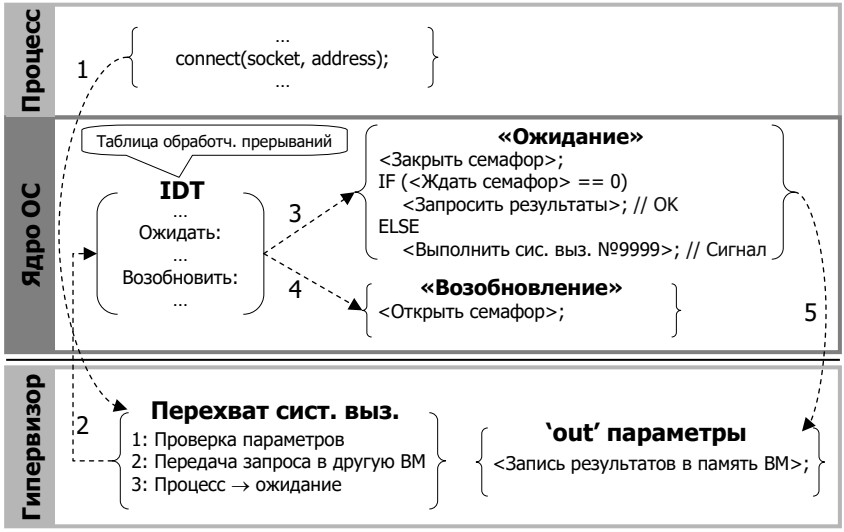


Рис. 4. Прозрачное обслуживание системного вызова в вычислительной ВМ.

Для определения адреса параметров системного вызова в физической памяти гипервизор программным образом обходит таблицы приписки процесса и вычисляет условно физический адрес в контексте ВМ. Далее, зная отображение памяти ВМ на машинную память, гипервизор вычисляет точное размещение параметров вызова в физической памяти. В процессе чтения параметров, расположенных в памяти процесса (например, в случае системного вызова *write*) возможна ситуация, когда данные расположены в странице памяти, откачанной ОС на внешнее устройство.

Гипервизор, обнаружив откачанную страницу, вбрасывает в ВМ исключение ошибка страницы с адресом, соответствующим отсутствующей странице, и возобновляет управление ВМ. ОС подкачивает страницу в память и возвращает управление процессу по адресу инструкции выполнения системного вызова. Процесс повторно выполняет системный вызов, и гипервизор заново начинает копирование параметров. Такой процесс будет повторяться до тех пор, пока все страницы памяти, занятые входными параметрами системного вызова, не окажутся в физической памяти.

После копирования входных параметров системного вызова гипервизор возобновляет выполнение вычислительной ВМ и переводит процесс в состояние ожидания. Для этого в точке выполнения системного вызова он вбрасывает синхронное прерывание, для которого модуль ядра в вычислительной ВМ зарегистрировал обработчик. В результате, вместо 128 прерывания, соответствующего системным вызовам, аппаратура доставляет другое прерывание.

Получив управление, обработчик осуществляет доступ к закрытому семафору, что переводит процесс в состояние ожидания штатными средствами ОС. Процесс ожидает открытие семафора в режиме, допускающем обработку внешних событий. В случае поступления сигнала для процесса ОС прерывает ожидание семафора. Модуль ядра при этом имитирует запрос на выполнение несуществующего системного вызова (например, с номером 9999). Ядро ОС, разумеется, не будет выполнять этот запрос, однако до того, как вернуть управление процессу, оно выполнит обработку поступивших сигналов.

После обслуживания сигнала ОС возвращает управление процессу на исходную инструкцию системного вызова, и процесс повторно выполняет запрос на системный вызов. Гипервизор перехватывает его, определяет, что в данное время системный вызов находится в процессе удаленного обслуживания, снова подменяет прерывание, и процесс во второй раз переходит в состояние ожидания. Такой циклический процесс будет повторяться до тех пор, пока из сервисной ВМ не поступят результаты выполнения системного вызова.

При получении результатов из сервисной ВМ гипервизору необходимо возобновить выполнение доверенного процесса со следующей инструкции, причем в регистр *r/eax* должен быть записан результат выполнения вызова, а в память процесса (например, в случае системного вызова *read*) по заданным адресам должны быть записаны выходные данные, если они имеются.

Гипервизор выводит процесс из состояния ожидания посредством вброса другого синхронного прерывания в вычислительную ВМ, обработчик которого освобождает семафор и, тем самым, возобновляет выполнение процесса. Процесс далее выполняет гипервызов на запрос результатов системного вызова. Заметим, что хотя гипервызов производится из контекста ядра, в точке гипервызова на аппаратуре загружены собственные таблицы приписки доверенного процесса. Гипервизор просматривает таблицы и проверяет, что виртуальные адреса, по которым должны быть записаны результаты отображены на физическую память и доступ к этим адресам открыт по записи. Наличие доступа только по чтению может означать, что данная область памяти является «копируемой при операции записи». При наличии доступа по записи ко всему необходимому диапазону адресов гипервизор копирует результаты выполнения системного вызова из хранилища в адресное пространство процесса и возобновляет выполнение ВМ. В противном случае гипервизор для всех необходимых виртуальных страниц вбрасывает в ВМ исключение «ошибка страницы», дающее ОС указание выделить память для соответствующей виртуальной страницы и отобразить ее на физическую память.

При удаленном обслуживании системного вызова делегат может получить сигнал от ОС в сервисной ВМ, например, сигнал SIGPIPE о разрыве сетевого соединения. В этом случае делегат извещает гипервизор о полученном сигнале, и гипервизор доставляет сигнал доверенному процессу. Для этого он

информирует о сигнале модуль ядра ОС, который, в свою очередь, доставляет сигнал процессу средствами API ядра ОС Linux. Доставка процессу сигнала может привести к его аварийному завершению. В этом случае монитор известит гипервизор об окончании выполнения процесса, и гипервизор освободит память, отведенную под служебные структуры данных.

3.1. Точка обслуживания системного вызова

В подавляющем большинстве случаев системный вызов может быть ассоциирован с ресурсами какой-либо одной из виртуальных машин либо на основании номера системного вызова, либо на основании его параметров. В частности, системный вызов *socket*, создающий конечную точку для сетевого взаимодействия и возвращающий файловый дескриптор для работы с ней, должен обслуживаться в сервисной ВМ. Системный вызов *write* требует дополнительного анализа файлового дескриптора, передаваемого в параметрах вызова. Если дескриптор ассоциирован с сокетом, то он должен обслуживаться в сервисной ВМ, в противном случае – в вычислительной ВМ.

ОС в обеих ВМ поддерживают наборы файловых дескрипторов соответственно для доверенного процесса и его делегата независимо друг от друга. Если гипервизор после удаленного обслуживания системного вызова вернет доверенному процессу идентификатор файлового дескриптора, полученного от делегата, без каких-либо изменений, то это может привести к наличию в памяти доверенного процесса двух идентичных дескрипторов, которые на самом деле соответствуют разным ресурсам в той и другой ВМ. В дальнейшем при выполнении доверенным процессом системного вызова для такого дескриптора гипервизор не сможет различить, относится ли вызов к локальному ресурсу в вычислительной ВМ или удаленному в сервисной ВМ.

Для решения этой проблемы гипервизор реализует дополнительный уровень абстракции файловых дескрипторов для доверенных процессов. Файловые дескрипторы, хранимые в памяти доверенного процесса (в каких-то переменных), представляют собой не реальные дескрипторы, назначенные ядром ОС в вычислительной или сервисной ВМ, а индексы в *таблице удаленных ресурсов*, поддерживаемой гипервизором. Гипервизор перехватывает и обрабатывает все системные вызовы доверенного процесса, у которых входные или выходные параметры содержать файловые дескрипторы. Если параметр входной, то гипервизор извлекает из таблицы удаленных ресурсов фактический файловый дескриптор, модифицирует параметры системного вызова и передает его на обслуживание той ВМ, которая является владельцем ресурса с этим дескриптором. Если параметр выходной, то гипервизор создает новую запись в таблице удаленных ресурсов, помечая, какая из виртуальных машин является владельцем ресурса, и модифицирует выходные параметры процесса, подставляя индекс созданной записи вместо фактического значения дескриптора. Компонента гипервизора, отвечающая за обработку файловых дескрипторов, учитывает особенности

выделения свободных дескрипторов ОС Linux, включая нюансы работы системных вызовов *dup2* и *fcntl*. Таким образом, по значению, передаваемому доверенным процессом всегда можно определить виртуальную машину – владельца ресурса и номер файлового дескриптора в ее контексте.

В ряде случаев выполнение системного вызова может задействовать ресурсы обеих ВМ. В частности, параметры системного вызова *select* и его аналогов, а именно наборы файловых дескрипторов, для которых процесс ожидает возникновения соответствующих событий, могут одновременно относиться как к ресурсам вычислительной, так и сервисной ВМ. Такой вызов должен обслуживаться одновременно в обеих ВМ, иначе программа может перестать выполняться корректно. При этом системный вызов в каждой виртуальной машине должен обслуживаться только с теми параметрами (файловыми дескрипторами), которые относятся к ресурсам данной ВМ.

При перехвате запроса на выполнение системного вызова, требующего одновременного выполнения в обеих ВМ, гипервизор, используя таблицу удаленных ресурсов, расшивает фактические параметры на два непересекающихся набора (по одному для каждой из ВМ) и выполняет вызов одновременно в обеих ВМ с соответствующими наборами параметров. Модифицированный набор параметров записывается в память доверенного процесса поверх оригинального. Перед возвратом управления процессу (после обслуживания системного вызова) гипервизор восстанавливает исходный набор параметров, возможно, корректируя его с учетом результатов системного вызова, полученных из сервисной ВМ. Итоговым результатом выполнения системного вызова является результат той ВМ, которая хронологически первой закончила выполнение своей части вызова, результаты другой ВМ отбрасываются.

По получении результатов от одной из ВМ гипервизор производит отмену выполнения системного вызова в другой ВМ. Механизм отмены выполнения системного вызова реализован в обеих виртуальных машинах по-разному. В случае вычислительной ВМ модуль ядра посылает доверенному процессу определенный сигнал (не используемый процессом). При этом модуль системы защиты непосредственно перед посылкой сигнала регистрирует для процесса специальный «пустой» обработчик сигнала, представляющий собой адрес RET инструкции в коде доверенной программы. Адрес инструкции указывается в паспорте задачи. Регистрация обработчика гарантирует, что посылка сигнала не приведет к аварийному останову процесса. В сервисной ВМ все системные вызовы, которые могут выполняться одновременно в обеих ВМ, выполняются в отдельном потоке (нити) делегата. Отмена выполнения системного вызова производится посредством принудительного завершения этого потока.

4. Производительность системы

Описанная в этой работе система реализована на базе монитора виртуальных машин KVM [9]. KVM включен в основную ветку разработки ядра ОС Linux и представляет собой модуль, динамически загружаемый в ядро базовой (хост) операционной системы Linux. Управление выполнением ВМ реализуется совместно ядром хост системы, модулем KVM и пользовательской программой QEMU. QEMU виртуализирует периферийные устройства и обеспечивает совместный доступ виртуальных машин к оборудованию, установленному на компьютере и управляемому базовой системой.

Реализация, представленная в этой работе, построена на базовой операционной системе Linux с ядром версии 2.6.31.6 и мониторе виртуальных KVM версии 88. Суммарный объем кода компонент системы составляет порядка 16 тыс. строк. Виртуальные машины выполняются под управлением ОС Linux из дистрибутива Fedora версии 9 со штатным ядром версии 2.6.27.12-78.2.8.fc9.i686. На компьютере установлен четырехядерный процессор Phenom 9750 компании AMD с тактовой частотой 2.4 Ггц и 2 Гбайта оперативной памяти. Данный процессор поддерживает технологию аппаратной виртуализации, включая виртуализацию памяти на базе вложенных (NPT) таблиц приписки виртуальной машины. Базовая операционная система использует все четыре ядра процессора (ядро базовой ОС собрано в SMP конфигурации). Каждой виртуальной машине выделяется по одному виртуальному процессору и по 512 Мбайт оперативной памяти.

Для проведения ряда тестов используется второй компьютер такой же конфигурации. В обоих компьютерах установлены 100Мбит-ные сетевые адаптеры, связанные друг с другом через сетевой концентратор (хаб). К концентратору подключены только данные две машины.

Доступ виртуальной машины к сети осуществляется посредством создания в базовой ОС штатными средствами ядра ОС программного сетевого интерфейса (TAP0). Этот интерфейс является образом сетевого интерфейса (ETH0) виртуальной машины в базовой системе. Привязка интерфейса TAP0 к физической среде производится через программный Ethernet мост (bridge) в ядре базовой ОС, также организуемый штатными средствами ОС Linux. Такая конфигурация (рис. 1) позволяет открыть ВМ для других машин в сети, в отличие от конфигурации QEMU по умолчанию, скрывающей ВМ от других машин в сети посредством механизма трансляции сетевых адресов (NAT) и разрешающей только исходящие соединения в ВМ.

Для тестирования производительности мы использовали четыре специально разработанных синтетических теста, моделирующих «тяжелые» для системы сценарии использования, и три типовых программы: утилиту тестирования производительности сети TTCP, программу удаленного доступа SSH и веб-сервер Apache.

Синтетические тесты основаны на выполнении системного вызова *select()* с одним или двумя файловыми дескрипторами. Один из дескрипторов (*локальный*), обозначим его *LocalFD*, представляет собой файл, открытый в контексте вычислительной ВМ, другой (*удаленный*), обозначим его *RemoteFD*, – сокет, созданный в сервисной ВМ. Во всех тестах, кроме первого, выполнение системного вызова требует взаимодействия между ВМ. Тесты запускаются из контекста вычислительной ВМ.

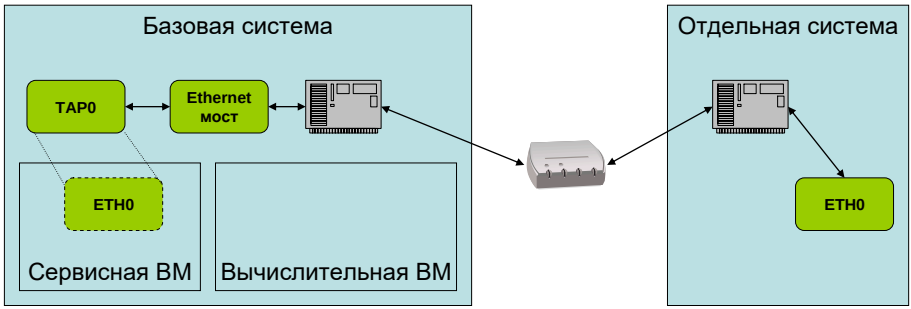


Рис. 5. Конфигурация сети для тестирования производительности.

Обрамление дескриптора квадратными скобками означает, что запрашиваемая операция не может быть выполнена для данного ресурса, и системный вызов заблокирует выполнение соответствующего пользовательского процесса в одной из ВМ. Отсутствие квадратных скобок означает возможность выполнения операции и немедленный возврат управления пользовательскому процессу. Тогда синтетические тесты проверяют следующие сценарии выполнения системного вызова:

- select(LocalFD);
- select(RemoteFD);
- select(LocalFD, [RemoteFD]);
- select([LocalFD], RemoteFD).

Выполнение системного вызова *select(LocalFD)* не требует взаимодействия между виртуальными машинами и характеризует «вычислительные» накладные расходы внутри вычислительной ВМ на перехват системных вызовов, анализ фактических параметров и пр. Выполнение системного вызова *select(RemoteFD)* показывает суммарное время, необходимое на доставку запроса делегату в сервисную ВМ, выполнение системного вызова в ней и возврат результатов процессу, пребывающему все это время в состоянии ожидания.

Выполнение системного вызова *select* с двумя дескрипторами включает в себя взаимодействие между виртуальными машинами и, кроме того, требует

выполнения отмены системного вызова в той ВМ, в которой процесс был заблокирован, а именно в той ВМ, дескриптор ресурса которой обрамлен квадратными скобками. Следует отметить принципиальное отличие третьего и четвертого теста. В тесте *select(LocalFD, [RemoteFD])* отмена системного вызова, выполняемого делегатом в сервисной ВМ, производится *асинхронно* для процесса в вычислительной ВМ, и он может продолжить свое выполнение, не дожидаясь подтверждения от сетевой ВМ. Такая оптимизация возможна, поскольку для нормального продолжения выполнения процесса достаточно результатов получаемых локально от ядра вычислительной ВМ. В свою очередь, в тесте *select([LocalFD], RemoteFD)* процесс не может продолжить выполнение, пока выполнение системного вызова не будет прервано, что приводит к дополнительным накладным расходам.

В таблице 1 указано время выполнения тестов (в секундах) в цикле из 100 тысяч итераций. Первая строка таблицы характеризуют выполнение теста *select(LocalFD)* в базовой системе. Вторая строка показывает время выполнения теста *select(LocalFD)* в ВМ со включенным механизмом отслеживания выполнения процесса. Возрастание времени выполнения на один порядок обусловлено затратами на перехват инструкции INTn, инициирующей системный вызов и, главное, инструкции IRET, реализующей возврат в пользовательский режим как из системного вызова, так из обработчиков прерываний. Мы ожидаем, что, адаптировав предлагаемую систему под механизм быстрого выполнения системных вызовов (SYSCALL/SYSRET), поддерживаемый современными процессорами семейства x86, данный показатель может быть улучшен.

Показатель в третьей строке таблицы говорит о том, что собственно вычислительные затраты системы (за исключением перехвата двух инструкций) составляют порядка 20%. Четвертая строка характеризует накладных расходы на асинхронную отмену части системного вызова, выполняемой делегатом в сервисной ВМ. Отметим, что несколько последовательных операций отмены также выполняются асинхронно, и синхронизация производится только при последующем выполнении «существенного» (не отменяемого) удаленного системного вызова. Кроме того, делегат не будет исполнять системный вызов, если обнаружит, что для него уже поступила команда на отмену. Такое возможно в силу асинхронности выполнения виртуальных машин.

В тесте select(LocalFD, [RemoteFD]) синхронизация производится только после выходе из цикла при закрытии «удаленного» сокета (системный вызов close). При наличии большого количества последовательно отмененных системных вызовов (100 тысяч в данном случае) такая операция синхронизации может занимать продолжительное время, что и отражено в таблице. Непосредственно при выходе из основного цикла время выполнения теста составляет всего 15 секунд. Последующая операция закрытия сокета,

требующая синхронизации операций отмены требует дополнительных 16 секунд.

Последние две строки таблицы характеризуют полноценное удаленное обслуживание системного вызова. Шестая строка таблицы показывает накладные расходы на отмену локальной части системного вызова в вычислительной ВМ, которая всегда выполняется синхронно для процесса.

Тест	Время (сек.)
Базовая система	1
Виртуальная машина	9
select(LocalFD)	11
select(LocalFD, [RemoteFD])	15 (31)
select(RemoteFD)	189
select([LocalFD], RemoteFD)	253

Таб. 1. Время выполнения синтетических тестов.

Синтетические тесты показывают, что при худшем сценарии время выполнения системного вызова может возрастать до 250 раз. Однако, на практике это не оказывает существенного влияния на время выполнения программы в целом. Во-первых, большинство системных вызовов, требующих удаленное выполнение, включает в себя передачу данных через периферийное устройство (в частности, по сети). Во-вторых, приложение, как правило, выполняет также другие действия, которые нивелируют данные накладные расходы, например, оно может часто находится в состоянии ожидания открытия семафора.

На рисунке 6 приведены результаты тестирования системы на утилите TTCP, выполняющей в цикле передачу пакетов между двумя машинами в сети. Первая диаграмма соответствует оригинальной TTCP утилите, вторая – модифицированной, в которую мы добавили выполнение системного вызова *select* перед каждой посылкой пакета. Системный вызов *select* в данном случае выполняется удаленно, т.е. соответствует синтетическому тесту select(RemoteFD). При выполнении оригинальной утилиты накладные расходы на удаленное обслуживание системных вызовов составили всего 2% от суммарного времени выполнения программы. Добавление системного вызова *select* увеличило накладные расходы до 31%, однако даже в этом случае они значительно меньше издержек в синтетическом тесте select(RemoteFD), где время выполнения увеличилось в 189 раз.

Мы также тестировали предлагаемую систему на утилите удаленного доступа SSH, посредством копирования файла между двумя машинами в сети, а также на веб-сервере Apache, запуская для него пакет тестов нагрузочного

тестирования Flood. В обоих случаях время выполнения тестов варьировалось в пределах 1% от их выполнения на базовой системе. Таким образом, мы считаем, что предлагаемый механизм удаленного обслуживания системных вызовов является достаточно эффективным для его использования в промышленных задачах.

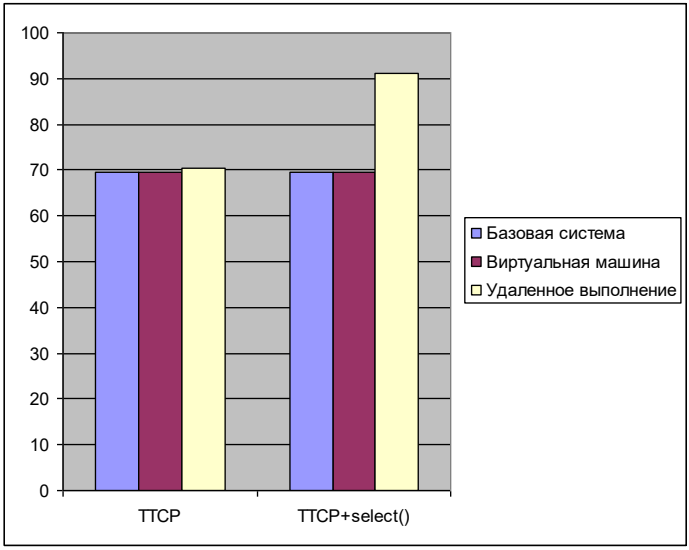


Рис. 6. Время выполнения утилиты TTCP (сек.).

5. Заключение

В данной работе представлен подход к удаленному выполнению системных вызовов пользовательского процесса, не требующий внесения каких-либо изменений (в т.ч. перекомпиляции) ни в код процесса, ни в код операционной системы. Особенностью рассматриваемого подхода является то, что процессу может быть предоставлен контролируемый доступ к таким ресурсам, к которым операционная система, под управлением которой выполняется процесс, ни прямого, ни опосредованного доступа не имеет. Это позволяет предоставлять селективный доступ к ресурсам, требующим контроля (например, сети Интернет), отдельным доверенным пользовательским процессам, в то же время оставляя эти ресурсы недоступными другим процессам и даже ядру операционной системы.

Предлагаемый подход основан на выполнении операционной системы (и управляемых ею процессов) внутри аппаратной виртуальной машины. Монитор виртуальных машин (гипервизор) перехватывает системные вызовы отдельных доверенных процессов и при необходимости передает их на

обслуживание сервисной машине. Сервисная машина может быть другой виртуальной машиной, выполняющейся параллельно, или другой физической машиной, с которой у гипервизора есть канал связи.

Рассматриваемый подход реализован на базе монитора виртуальных машин KVM. В качестве сервисной машины использовалась параллельно выполняющаяся виртуальная машина, а в качестве контролируемых ресурсов было выбраны сетевые ресурсы (в т.ч. ресурсы Интернета). Тестирование производительности системы показало, что на реальных приложениях накладные расходы на удаленное обслуживание системных вызовов укладываются в пределы 3 процентов.

Литература

[1] Tanenbaum, A. S., Herder, J. N., Bos, H. Can We Make Operating Systems Reliable and Secure?. Computer 39, 5 (May 2006), pp. 44-51.

[2] Burdonov, I., Kosachev, A., Iakovenko, P. Virtualization-based separation of privilege: working with sensitive data in untrusted environment. In Proceedings of the 1st Eurosys Workshop on Virtualization Technology for Dependable Systems, New York, NY, USA, 2009, ACM, pp. 1-6.

[3] Яковенко П.Н. Контроль доступа процессов к сетевым ресурсам на базе аппаратной виртуализации. Методы и средства обработки информации. Труды Третьей Всероссийской научной конференции, М, 2009, стр. 355-360.

[4] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., Ports, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In Proceedings of the 13th international Conference on Architectural Support For Programming Languages and Operating Systems, ACM, 2008, pp. 2-13.

[5] Adams, K., Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems, ACM, 2006, pp. 2-13.

[6] VirtualSquare: Remote System Call. http://wiki.virtualsquare.org/index.php/Remote_System_Call

[7] Sun Microsystems, Inc. RPC: Remote Procedure Call. Protocol Specification. Version 2. Network working group. RFC 1057. 1988.

[8] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. Network working group. RFC 1094. 1989.

[9] Shah. A. Deep Virtue: Kernel-based virtualization with KVM. Linux Magazine (86), 2008, pp. 37-39.

Система моделирования Grid: реализация и возможности применения

Грушин Д.А., Поспелов А.И.

Аннотация. В статье описывается разработанная в ИСП РАН система моделирования распределенных вычислительных сред Grid. С помощью этой системы проведен анализ реальной вычислительной среды Sharcnet. На основе анализа были выявлены возможные способы существенного увеличения эффективности работы среды.

1. Введение

В последнее время к вычислительным кластерам проявляется повышенный интерес со стороны науки, образования и промышленности. С доступностью кластерных технологий связан рост числа установок, которые строятся, устанавливаются и которые специалисты пытаются применять для решения своих производственных задач. Кластерные вычислительные системы становятся повседневным инструментом исследователя и инженера. Однако, любая организация, становясь обладателем вычислительного кластера, не использует его постоянно, в режиме “24/7”, более того, очень часто такой дорогостоящий вычислительный ресурс простаивает.

В связи с увеличением количества кластеров набирает все большую популярность концепция Grid [12, 16]. Grid позволяет совместно использовать вычислительные ресурсы, которые принадлежат различным организациям и которые могут быть расположены в различных административных областях. В Grid могут объединяться разнородные вычислительные ресурсы: персональные компьютеры, рабочие станции, кластеры и супер-компьютеры.

Одним из наиболее распространенных в настоящее время программных средств для реализации Grid является пакет Globus Toolkit [7]. Пакет Globus Toolkit разрабатывается, поддерживается и продвигается международным альянсом разработчиков из университетов США и Великобритании, а также научных лабораторий и вычислительных центров. Globus Toolkit является свободно распространяемым с открытым исходным кодом программным пакетом и предлагает базовые средства для создания Grid инфраструктуры: средства обеспечения безопасности в распределенной среде, средства надежной передачи больших объемов данных, средства запуска и получения

результатов выполнения задач на удаленных вычислительных ресурсах. На базе пакета Globus Toolkit создаются промышленные версии реализаций Grid инфраструктуры, например, такие как Univa [19] и Platform Globus Toolkit [17].

Однако, несмотря на то, что уже сейчас предлагаются, ставшие “де-факто” стандартными, средства создания Grid-инфраструктур, существует ряд важных научных задач, в том числе и теоретических, без решения которых полномасштабное использования возможностей Grid технологий в промышленности невозможно. Одной из актуальных задач в настоящее время является эффективное управление вычислительными ресурсами в распределенной среде. С ростом числа ресурсных центров входящих в распределенную инфраструктуру, отсутствие хорошего планировщика, обеспечивающего управление потоком задач, не только значительно снижает эффективность использования всей Grid-инфраструктуры, но может сделать бессмысленным ее создание. При этом, следует отметить, что для таких распределенных систем характерным является динамичное развитие, что делает невозможным решение задачи эффективного управления “в статике” – один раз и навсегда.

С другой стороны, оптимизация алгоритмов управления распределенной средой на непосредственно уже существующей Grid-инфраструктуре затруднено и связано со значительными издержками и простоями ресурсных центров, а часто в силу масштабности распределенной среды вообще не возможно. В связи с этим, актуальной задачей является создание системы моделирования Grid-инфраструктуры, которая позволит адекватно оценивать ее поведение при изменяющихся условиях и, на основе этого, оптимизировать стратегии управления потоками задач.

Система моделирования может быть использована для оценки эффективности распределенной вычислительной среды в различных ситуациях, например:

- при изменении нагрузки: количества поступающих задач, их размерности, приоритета, периода поступления и т.д.;
- при отключении части вычислительных ресурсов или добавлении новых ресурсов;
- при увеличении количества передаваемых данных;
- при выходе из строя части коммуникационных каналов

При этом оценка эффективности управления может проводиться по следующим наиболее популярным критериям [4]:

- минимизация среднего времени ожидания задачи в очереди;
- минимизация максимального времени выполнения группы задач (makespan);
- максимизация пропускной способности – числа завершенных задач в единицу времени;

- минимизация простоев процессоров
- и т.д.

В настоящее время существует несколько проектов по разработке систем моделирования Grid. Среди них наиболее известны: Bricks [14], MicroGrid [11], OptorSim [13], SimGrid [10] и GridSim [1]. Данные системы обладают как достоинствами, так и недостатками. Среди недостатков можно отметить узкую специализацию систем, отсутствие публично доступных версий, а также ограниченность моделируемых архитектур Grid систем. Особенности реализации некоторых из них накладывают ограничения на количество одновременно существующих элементов в Grid системе и требуют от пользователя знания специальных языков программирования, что значительно снижает эффективность работы с такими системами.

2. Система моделирования Grid

С 2007 года в ИСП РАН разрабатывается система моделирования Grid. При разработке мы старались избежать недостатков присущих существующим системам, а также реализовать некоторые новые интересные идеи.

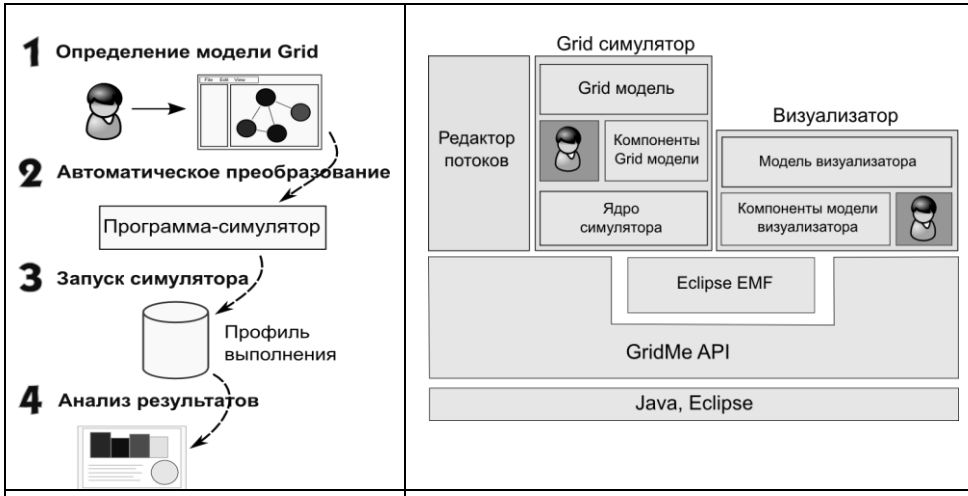
В частности, система проектировалась так, чтобы сделать работу пользователя максимально удобной и быстрой. В отличие от перечисленных выше систем в разработанной системе не нужно вручную писать программу моделирования. Пользователь работает в специальном редакторе, задавая топологию Grid системы и свойства отдельных элементов. При этом, автоматически проверяются различные виды ошибок: значения параметров, выходящие за область допустимых значений, несовместимость различных элементов, соединенных между собой, и т.п.

Сценарий работы с системой изображен на рисунке 1(а). Пользователь задает описание моделируемой среды, и указывает различные параметры. Система автоматически генерирует код программы моделирования и компилирует его. Программа-симулятор запускается и создает в результате своей работы профиль выполнения. Полученный профиль анализируется и представляется пользователю в виде HTML документа.

Система моделирования реализована на основе платформы Eclipse [2], с использованием только языка Java. Это дает возможность интеграции с другими Eclipse приложениями, например, средой разработки Java, системами контроля версий, и т.п. и позволяет использовать систему моделирования под различными операционными системами – Linux, Windows, Solaris и др.

Система расширяема и рассчитана на гибкое использование. Система позволяет моделировать различные Grid архитектуры: одно и двух-уровневые системы с одним или несколькими брокерами, добавлять хранилища данных, определять топологию сетевых соединений и т.д. Система включает в себя множество реализованных компонент, таких как брокер, кластер, поток задач

и т.д. Кроме того, пользователи могут расширять систему, реализовывая свои собственные компоненты.



(a) Сценарий работы системы моделирования Grid

(b) Основные компоненты системы моделирования Grid

Рис. 1. Система моделирования Grid.

Поведение отдельных элементов моделируется с помощью конечных автоматов, что позволяет работать с моделями больших систем – порядка тысяч процессоров и более миллиона задач.

Система предоставляет возможность для быстрого описания алгоритмов распределения задач с помощью набора правил. При моделировании распределения задач в Grid очень часто требуется проверить несколько алгоритмов, незначительно отличающихся друг от друга, например, сортировкой входного потока задач, способом выбора очередной задачи или ресурса и т.п. Описание алгоритма с помощью набора правил в такой ситуации позволяет гораздо быстрее проверить работу алгоритма, чем в случае реализации его в виде, например, Java класса, с последующей отладкой и тестированием.

В системе поддерживается возможность проведения серии экспериментов, состоящей из последовательных запусков выполняемой модели с изменением некоторых параметров при каждом следующем запуске. Например, может изменяться поток задач, конфигурация кластеров, сетевых соединений и т.п. Это позволяет в рамках одного эксперимента посмотреть динамику изменения эффективности системы и определить узкие места.

В системе реализован удобный механизм обработки результатов моделирования. Результат выполнения модели хранится в отдельном профиле и может обрабатываться независимо. Пользователь может использовать свой шаблон для выбора и визуализации только необходимой в данный момент информации. Это позволяет нескольким исследователям провести моделирование один раз, а затем независимо анализировать полученную информацию.

Система также включает в себя редактор и анализатор записей потоков задач (workload) [15, 8]. Запись потока представляет собой текстовый файл, каждая строка которого содержит характеристики отдельной задачи: время порождения, время запуска, общее время выполнения, количество занимаемых процессоров и т.д. Анализатор позволяет отобразить различные характеристики потока – количество задач, соотношение однопроцессорных и параллельных задач, график порождения задач во времени и т.п. С помощью редактора можно изменять поток – копировать и перемещать части потока, соединять несколько потоков в один, изменять характеристики группы задач и т.п. Также, редактор позволяет создавать синтетический поток по заданным параметрам.

Основные компоненты системы изображены на Рис. 1(b). Это – редактор и анализатор потоков, симулятор Grid системы, визуализатор.

2.1. Модель представления Grid системы

Модель для представления Grid инфраструктуры (мета-модель Grid) в нашей системе изображена на Рис. 2.

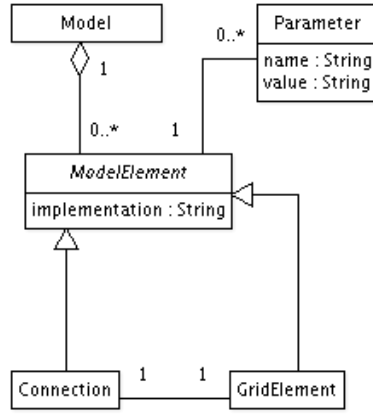


Рис. 2. Мета-модель Grid.

Модель Grid состоит из множества элементов (`ModelElement`), связанных между собой. Доступно два вида элементов – соединение (`Connection`) и Grid-элемент (`GridElement`). Grid-элементом может быть кластер, брокер, пользователь, хранилище данных и т.п. Соединения предназначены для передачи данных между Grid-элементами. У каждого элемента есть строковый атрибут реализация (implementation). Значение данного атрибута указывает на имя Java-класса, определяющего поведение элемента. Каждый элемент может быть параметризован. Параметр представляет собой пару строк (имя, значение) и может иметь дочерние параметры. Дочерние параметры используются, например, при задании свойств алгоритмов распределения задач. Предположим, для элемента мы выбираем реализацию “кластер” и для кластера определяем значение параметра “schedulerClass” как “BackfillLocal”. В данном случае реализация “BackfillLocal” может также требовать задания значений параметров. В этом случае параметр “schedulerClass” будет иметь дочерние параметры.

Таким образом, модель Grid-системы определяется в несколько этапов (Рис. 3). Сначала мы создаем Grid элементы, из которых будет состоять Grid система. Затем задаем топологию сети путем создания связей между элементами и сетевыми соединениями. И на последнем этапе выбираем реализацию каждого элемента и определяем значения параметров для конкретной реализации.

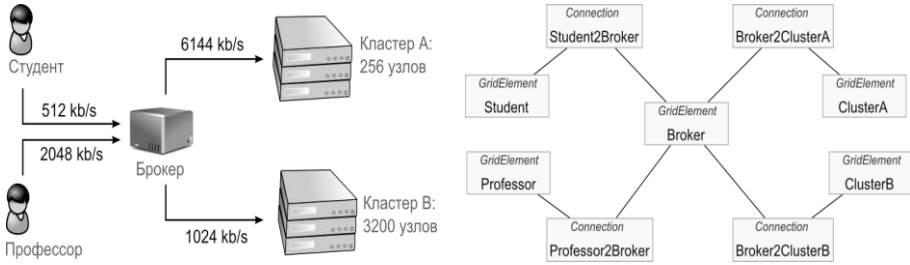
Рис. 3. Этапы определение модели Grid-системы.

В системе представлены основные элементы, необходимые для создания моделей Grid. Это – кластер, брокер, поток задач, сетевое соединение. Базовый набор реализованных алгоритмов распределения для кластера и брокера включают:

BackfillLocal	реализация алгоритма Backfill (алгоритм обратного заполнения) для кластера. Задание с меньшим приоритетом может быть запущено вне очереди, но только в том случае, если оно не будет мешать запуску более приоритетных заданий
BestFitLocal	реализация алгоритма “наилучший подходящий” для кластера. Для данного текущего количества свободных узлов подбирается задача, наиболее близкая по ширине
FirstFitLocal	реализация алгоритма “первый подходящий” для кластера. Размещается задача из начала очереди. Если узлов для запуска задачи не достаточно, то размещения не происходит
RandomFitGlobal	“случайный подходящий” для брокера. Для текущей задачи случайным образом выбирается кластер из множества подходящих

и другие.

В качестве примера рассмотрим, каким образом в системе моделирования будет определяться архитектура, изображенная на рисунке 4(a).



(a) Система (b) Модель

Рис. 4. Пример определения Grid системы.

Пользователи, брокер и кластеры представляются с помощью Grid-элементов, с именами Student, Professor, Broker, ClusterA, ClusterB соответственно – рисунок 4(b). Для задания соединений используются элементы `Connection` с именами Student2Broker, Professor2Broker, Broker2ClusterA, и Broker2ClusterB. В таблице 1 перечислены параметры и имена классов, реализующих поведение элементов.

Имя	Реализация	Параметр	Значение
ClusterA	SimpleCluster		
		nodes	256
		speedup	1
		schedulerClass	”BackfillLocal“
ClusterB	SimpleCluster		
		nodes	3200
		speedup	1
		schedulerClass	”BackfillLocal“
Student	WorkloadTaskFlow		
		wfile	”student.sfw.zip“
		startDelay	0
Professor	WorkloadTaskFlow		
		wfile	”professor.sfw.zip“
		startDelay	0
Broker	SimpleBroker		
		schedulerClass	”RandomFitGlobal“
Student2Broker	DelayedConstantConnection		
		count	512000
		period	1
Professor2Broker	DelayedConstantConnection		
		count	2048000
		period	1
Broker2ClusterA	DelayedConstantConnection		
		count	6144000
		period	1
Broker2ClusterB	DelayedConstantConnection		
		count	1024000
		period	1

Таб. 1: Параметры элементов

Для кластеров мы используем реализацию `SimpleCluster`. В качестве параметров необходимо указать количество узлов – параметр `nodes` и коэффициент ускорения – параметр `speedup`. Ускорение определяет, насколько быстрее, по сравнению с некоторым эталонным кластером, задачи будут выполняться на данном кластере. В нашем примере мы предполагаем, что задачи выполняются с одинаковой скоростью на обоих кластерах. Параметр `schedulerClass` определяет алгоритм распределения задач

локальным планировщиком. В примере мы задаем алгоритм `BackfillLocal`, представляющий собой реализацию алгоритма Backfill [6].

Для пользователей ”студент“ и ”профессор“ мы используем реализацию `WorkloadTaskFlow`. Данная реализация позволяет порождать задачи в моделируемой системе на основе собранных статистических данных использования реально существующей среды Grid. В качестве параметров задается имя файла в формате ”workload“ – параметр `wfile` и время до начала порождения первой задачи – параметр `startDelay`. Период позволяет активизировать различные потоки задач в различное время.

Брокер задается реализацией `SimpleBroker`. Единственным параметром является алгоритм распределения задач глобальным планировщиком – параметр `schedulerClass`. Мы указываем значение `RandomFitGlobal`. Это реализация алгоритма ”случайный из подходящих“ – для очередной задачи брокер выбирает множество кластеров, которые могут выполнить данную задачу, и затем случайным образом выбирает один кластер из данного множества.

Для сетевых соединений Student2Broker, Professor2Broker, Broker2ClusterA и Broker2ClusterB мы используем реализацию `DelayedConstantConnection`. Это простая реализация сетевого соединения позволяет передавать заданное количество данных с некоторой задержкой. В нашем примере это 512, 2048, 6144 и 1024 kb/sec соответственно.

После того, как описание модели завершено, трансляция данного описания, и последующая компиляция исходного кода происходит автоматически. На выходе получается выполняемая программа-симулятор, которую можно запустить и получить результат в виде профиля выполнения.

Для визуализации результатов система предоставляет готовые шаблоны, отображающие:

- загруженность системы – общую и с разбивкой по отдельным кластерам
- время ожидания задач в очереди – среднее и пиковое с разбивкой по классам задач и по отдельным кластерам
- пропускной способности – как по количеству задач, так и используя интегральную оценку

3. Эксперименты

Цель экспериментов заключалась в следующем. С использованием реализованного прототипа среды смоделировать поведение реально существующей Grid системы при различных условиях. В качестве распределенной Grid системы была выбрана сеть Sharcnet [18].

Распределенная вычислительная система Sharcnet (Shared Hierarchical Academic Research Computing Network) – это консорциум из 16 колледжей и университетов в юго-западной части провинции Канады Онтарио, вычислительные ресурсы которых объединены высокоскоростной оптической сетью (Рис. 5).

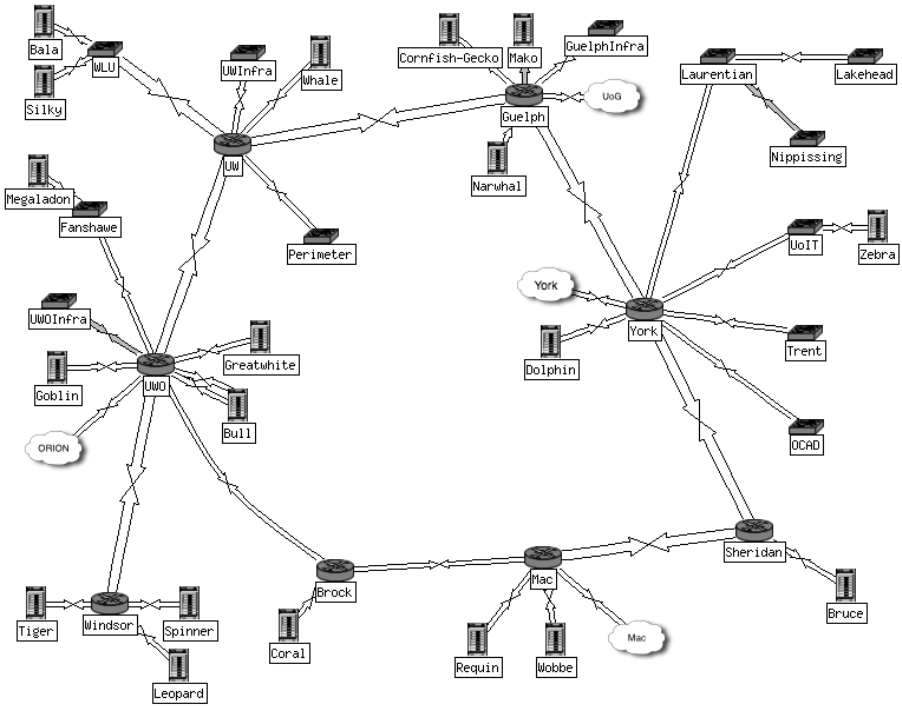


Рис. 5. Схема распределенной вычислительной системы Sharcnet.

Характеристики вычислительных ресурсов сети Sharcnet представлены в таблице 2.

Имя	Процессоры	Узлы
bruce	128	32 x 4 x Opteron
narwhal	1068	267 x 4 x Opteron dual core
tiger	128	32 x 4 x Opteron
bull	384	96 x 4 x Opteron
megaladon	128	32 x 4 x Opteron
dolphin	128	32 x 4 x Opteron
requin	1536	768 x 2 x Opteron

whale	3072	768 x 4 x Opteron
zebra	128	32 x 4 x Opteron
bala	128	32 x 4 x Opteron

Таб. 2. Характеристики вычислительных ресурсов сети Sharcnet.

В качестве входных данных была использована запись реального потока задач (“workload“ поток), выполнявшихся на кластерах с декабря 2005 по январь 2007 года [15]. Характеристики потока представлены на Рис. 6.

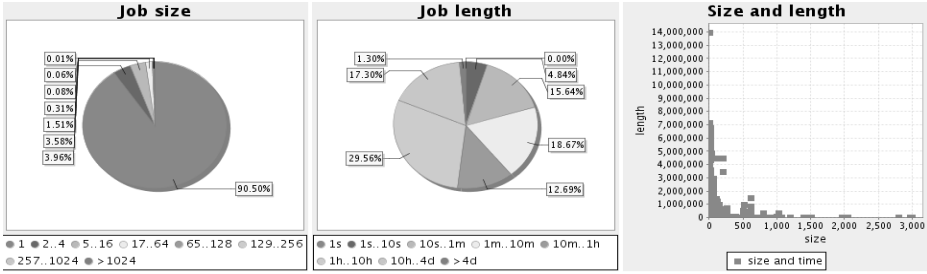


Рис. 6. Характеристики потока задач в сети Sharcnet: распределение ширины задач (количество запрашиваемых процессоров), длины задач, соотношение ширины и длины. Время показано в секундах.

Данный поток состоит по большей части из однопроцессорных задач. Однако, параллельных задач достаточно много – примерно 10%. Длина задач распределена более равномерно – большая часть, примерно 30%, имеет длину от 1 до 10 часов и чуть более половины всех задач по длительности составляют меньше одного часа, включая все большие параллельные задачи.

На Рис. 7 представлен график суммарного числа запрашиваемых процессоров в единицу времени. По оси абсцисс отложено время в секундах, по оси ординат – суммарное число запрашиваемых процессоров. Синяя горизонтальная линия показывает общее число процессоров в системе.

Мы видим, что поток не равномерный, на протяжении всего интервала присутствуют всплески запрашиваемого количества процессоров. Также, на графике можно заметить периодичность изменения нагрузки – временные интервалы 5000000, 1000000, 15000000 и т.д. Данные особенности присущи практически всем реальным потокам [3]. Примерно в середине временного интервала происходит перегрузка системы – запрашиваемая ширина становится больше доступной. Это говорит о том, что при любом распределении задач, в системе будут присутствовать очереди. Для данного потока, с 10% параллельных задач, будет происходить неполное заполнение кластеров, что еще больше увеличит размер очередей.

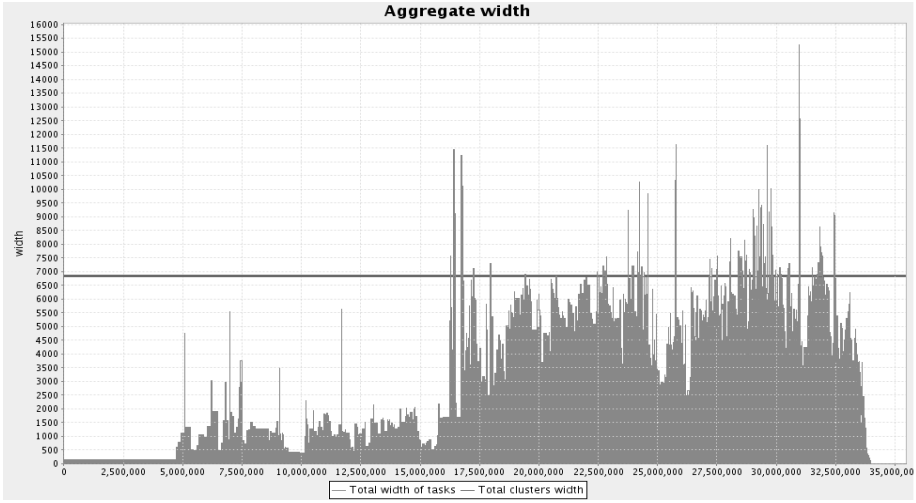
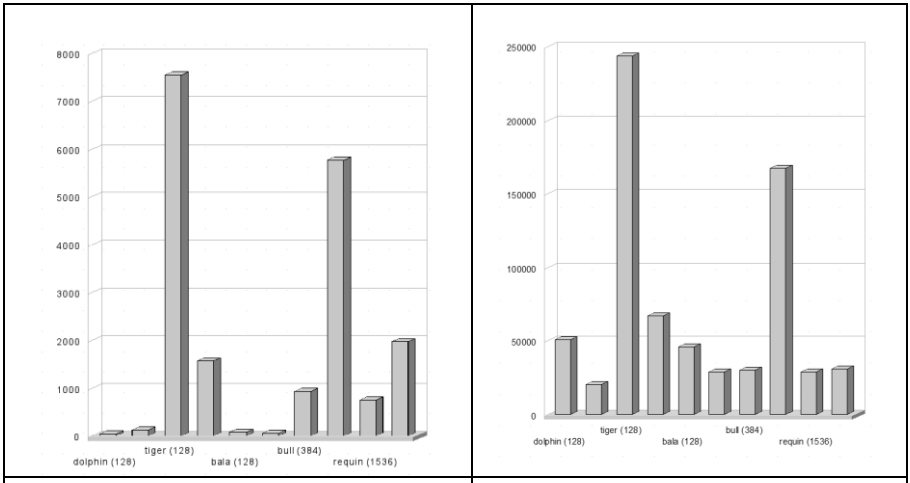


Рис. 7. Суммарное число запрашиваемых процессоров в единицу времени.

Особенность данного потока состоит также в том, что пользователи направляли задачи на кластеры непосредственно – для распределения не использовался брокер. При анализе потока оказалось, что в системе присутствует существенный дисбаланс нагрузки (Рис. 8).



Средняя длина очереди на каждом кластере	Среднее время ожидания на каждом кластере

Рис. 8. Дисбаланс в системе Sharcnet

Средние длины очередей в некоторых случаях отличаются почти в 100 раз. Это приводит к тому, что среднее время ожидания в очереди для различных кластеров отличается в несколько десятков раз.

Для проведения экспериментов, на основе данного потока мы создали несколько синтетических потоков. Необходимость использования синтетических потоков обусловлена различными факторами:

- запись оригинального потока может содержать различные несистематические особенности и всплески, связанные с конкретными событиями, что делает его неподходящим для анализа и прогнозирования поведения данной системы, но эти особенности могут быть сглажены при создании синтетических потоков;
- оригинальные потоки часто имеют большой размер и поэтому не очень удобны для выявления локальных свойств алгоритмов;
- синтетические потоки могут позволить создавать новые ситуации, которых не было в исходных данных.

Для создания синтетических потоков могут быть использованы различные подходы. Полностью синтетические потоки иногда бывают удобны для отладки, однако для анализа поведения моделируемых систем более подходящим является использование синтетических потоков, основывающихся на записях оригинальных потоков [5].

Для создания таких синтетических потоков мы использовали следующий поход. Для того чтобы определить поток задач

$$\{P_j\}_{j=1}^{M}$$

необходимо определить следующие параметры:

- . R_j - промежуток времени между поступлением . j и . $j+1$ задачами (. $j = 1, 2, \dots M-1$);
- . H_j - запрашиваемое время исполнения(. $j = 1, 2, \dots M$);
- . W_j - запрашиваемое число процессоров(. $j = 1, 2, \dots M$).

На основе оригинального потока задач для этих параметров оцениваются кумулятивные функции распределения и первые моменты. Далее, для каждого из параметров подбирается функция распределения в виде свертки нескольких распространенных функций распределения. Подбор осуществляется с помощью минимизации отклонения моментов и графиков функций по параметрам распределений в свертке и коэффициентам свертки.

Полученные таким образом распределения были использованы для генерирования нескольких синтетических потоков.

Мы применили разработанную систему моделирования и сравнили эффективность распределения задач в сети Sharcnet в оригинальном случае (без брокера) с распределением, получаемым с помощью брокера. Без брокера задачи поступали на кластеры в оригинальной последовательности, указанной

в файле загрузки. В обоих случаях на каждом кластере использовалась реализация алгоритма Backfill.

Алгоритм обратного заполнения Backfill работает по следующему принципу: размещая наиболее приоритетное задание, определяется момент времени, когда освободится достаточное количество ресурсов, занятых уже выполняющимися заданиями, затем производится резервирование этих ресурсов. Задание с меньшим приоритетом может быть запущено вне очереди, но только в том случае, если оно не будет мешать запуску более приоритетных заданий [6].

На брокере использовались различные алгоритмы распределения. В каждом случае вначале брокер выбирает множество кластеров, которые могут выполнить данную задачу – $W > W_j$, где W – число узлов кластера, а W_j – число узлов, запрашиваемых задачей, а затем выбирает один кластер исходя из заданного критерия:

N/W Выбирается кластер с наименьшим числом задач в очереди. Для данного кластера отношение N/W имеет минимальное значение, где N – число задач, стоящих в очереди, W – число процессоров кластера.

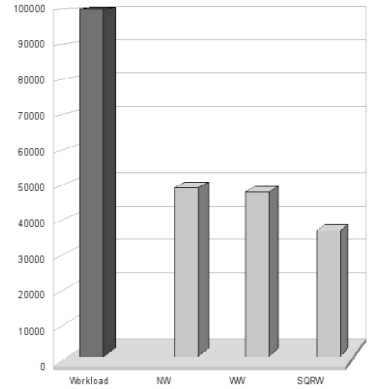
W/W Выбирается кластер с минимальной общей шириной задач в очереди. Для данного кластера отношение $(\sum_{j=1}^{N} W_j + \bar{W})/W$ имеет минимальное значение, где N – число задач, стоящих в очереди, W_j – ширина задачи, $\bar{W}$ – ширина отправляемой задачи, W – число процессоров кластера.

Sqr/W Выбирается кластер с минимальной общей площадью задач в очереди. Для данного кластера отношение $(\sum_{j=1}^{N} S_j + \bar{S})/W$ имеет минимальное значение, где N – число задач, стоящих в очереди, S_j – площадь задачи, $\bar{S}$ – площадь отправляемой задачи, W – число процессоров кластера.

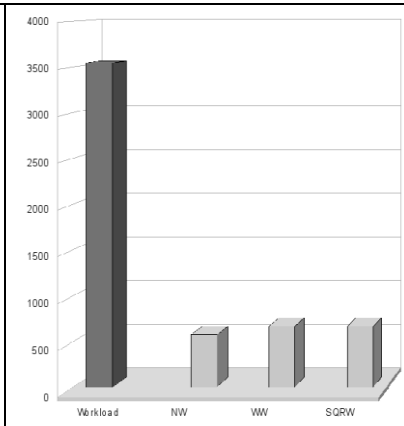
Всего было проведено 7 экспериментов:

1. задачи распределялись на кластеры согласно файлу загрузки;
2. задачи направлялись на брокер, который затем распределял их на кластеры. На брокере использовалась эвристика N/W;
3. на брокере использовалась эвристика W/W;
4. на брокере использовалась эвристика Sqr/W;
5. на брокер направлялись только однопроцессорные задачи. Параллельные задачи направлялись на кластеры согласно файлу загрузки. На брокере использовалась эвристика N/W;
6. однопроцессорные задачи, на брокере использовалась эвристика W/W;
7. однопроцессорные задачи, на брокере использовалась эвристика Sqr/W

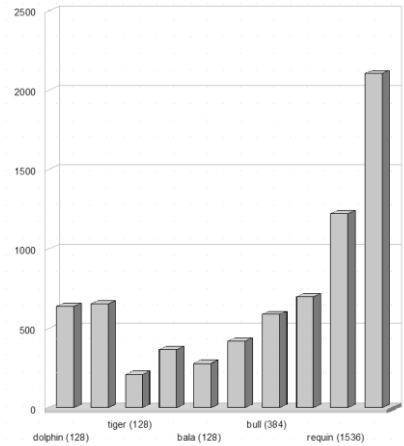
Мы сравнивали среднее время ожидания задач в очереди, а также характеристики самих очередей – длину и площадь, среднюю по всем кластерам и отдельно для каждого кластера.



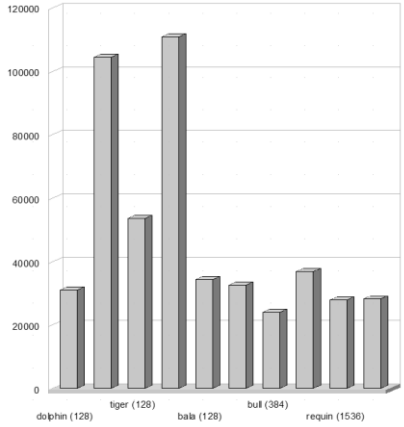
(a) Среднее время ожидания в очереди



(b) Средняя длина очереди



(c) Средняя длина очереди на каждом кластере. Распределение однопроцессорных задач, алгоритм Sqr/W



(d) Среднее время ожидания на каждом кластере. Распределение однопроцессорных задач, алгоритм Sqr/W

Рис. 9. Результаты экспериментов.

На рисунке 9 представлены результаты экспериментов при распределении только однопроцессорных задач. На рисунке 9(a) показано среднее время ожидания в очереди, определяемое как

$$T_{avg} = \frac{\sum_{j=1}^{N} T_{start} - T_{submit}}{N}$$

где . N – общее число запущенных задач, . T_{start} – время запуска задачи, . T_{submit} – время постановки задачи в очередь кластера. На рисунке 9(b) показана средняя длина очереди. Первый столбец на обеих диаграммах соответствует распределению задач без брокера. На рисунках 9(c) и 9(d) показаны средняя длина очереди и среднее время ожидания на каждом кластере в секундах.

Результаты показывают, что для данной вычислительной системы распределение потока однопроцессорных заданий через брокер дает значительный эффект: снижается среднее время ожидания заданий в очереди, а также происходит более равномерная загрузка вычислительных ресурсов.

При распределении всех задач (не только однопроцессорных) через брокер среднее время ожидания становится примерно на 5-7% меньше, чем в приведенных результатах. Однако, мы приводим результаты для распределения только однопроцессорных задач, поскольку, для проведения данных экспериментов нам была доступна информация только в виде записи потока задач. Из записи потока невозможно определить почему задача отправляется пользователем на тот или иной вычислительный ресурс. Причиной может быть архитектура системы, наличие специального программного обеспечения, личные пристрастия и т.п. Мы сделали предположение, что однопроцессорные задачи менее привязаны к конкретному кластеру, так как не зависят от среды передачи данных, которая установлена на кластере. Для системы Sharcnet такое предположение наиболее логично, так как среда передачи данных не одинакова для всех кластеров – таблица 3.

Имя	Процессоры	Архитектура
bruce	128	Myrinet 2g (gm)
narwhal	1068	Myrinet 2g (gm)
tiger	128	Myrinet 2g (gm)
bull	384	Quadrics Elan4
megaladon	128	Myrinet 2g (gm)
dolphin	128	Myrinet 2g (gm)
requin	1536	Quadrics Elan4
whale	3072	Gigabit Ethernet
zebra	128	Myrinet 2g (gm)
bala	128	Myrinet 2g (gm)

Таблица 3. Среда передачи данных кластеров сети Sharcnet.

Похожие результаты были получены в работе голландских исследователей при объединении двух Grid систем – Grid5000 и DAS2 [9]. В их работе отмечается наличие дисбаланса в обеих системах, и предлагается метод для его устранения используя глобальный планировщик. Результаты также показывают существенное уменьшение времени ожидания, примерно на 60%, и более равномерную загруженность кластеров.

В ходе проведенных нами экспериментов было замечено, что результаты распределения сильно зависят от входного потока задач. Очень трудно найти алгоритм распределения, который давал бы одинаково хорошие результаты на всех возможных потоках. Однако, зная характеристики вычислительной системы и характеристики предполагаемого потока задач, мы можем провести моделирование и определить какой алгоритм распределения показывает наиболее хороший результат. В некоторых случаях простая эвристика может давать лучшие результаты по сравнению с более сложной.

В связи с этим, актуальной представляется задача разработки такого алгоритма управления вычислительными ресурсами, при котором брокер анализирует поступающий к нему поток задач и, в зависимости от характеристик потока, выбирает эвристику, дающую оптимальное, согласно заданным критериям, распределение. Выбранная эвристика используется брокером для распределения задач по кластерам до тех пор, пока не произойдет ”переключение“ на другую эвристику.

4. Заключение

В статье представлена среда моделирования, разработанная в ИСП РАН, позволяющая оценивать поведение распределенных вычислительных систем при изменяющихся условиях и, на основе этого, оптимизировать стратегии управления потоками задач. Также представлены результаты использования реализованного прототипа данной среды на моделировании реально существующей вычислительной системы Sharcnet.

В будущем нам хотелось бы развивать данную систему как инструмент для оценки эффективности управления вычислительными ресурсами в Grid. Пользователями такой системы могут быть администраторы и исследователи, разрабатывающие новые алгоритмы управления ресурсами.

В ближайшее время предполагается провести эксперименты с задачами, требующими передачи больших объемов данных. Также, мы планируем расширить функциональность генератора синтетических потоков.

Система является свободно распространяемым с открытым исходным кодом программным пакетом и доступна по адресу http://gridme.googlecode.com.

Литература

[1] Buyya R., Murshed M. Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing // Concurrency and computation: practice and experience. "— 2002. "— Vol. 14. "— Pp. 1175–1220.

[2] Eclipse - an open development platform www.eclipse.org.

[3] Feitelson D. G. Locality of sampling and diversity in parallel system workloads // ICS '07: Proceedings of the 21st annual international conference on Supercomputing. "— ACM, 2007. "— Pp. 53–63.

[4] Feitelson D. G., Rudolph L. Metrics and benchmarking for parallel job scheduling // Lecture Notes in Computer Science. "— 1998. "— Vol. 1459. "— Pp. 1+.

[5] Feitelson D. G. Workload modeling for computer systems performance evaluation book draft. "— since 2005.

[6] Feitelson D. G., Weil A. M. Utilization and predictability in scheduling the IBM SP2 with backfilling // 12th Intl. Parallel Processing Symp. "— 1998. "— Pp. 542–546.

[7] Globus alliance. "— www.globus.org.

[8] The grid workloads archive. "— http://gwa.ewi.tudelft.nl/pmwiki/.

[9] Inter-operating grids through delegated matchmaking / A. Iosup, D. H. Epema, T. Tannenbaum et al. // Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC07). "— Reno, NV: 2007. "— November.

[10] Legrand A., Marchal L., Casanova H. Scheduling distributed applications: The simgrid simulation framework // Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid 2003 (CCGrid2003). "— 2003. "— Pp. 138–145.

[11] The microgrid: Using emulation to predict application performance in diverse grid network environments / H. Xia, H. Dail, H. Casanova, A. Chien // In Proceedings of the Workshop on Challenges of Large Applications in Distributed Environments (CLADE'04). IEEE Press. "— 2004.

[12] Tuecke S., Czajkowski K., Foster I. et al. Open grid services infrastructure (ogsi) version 1.0. "— 2003. "— June.

[13] Optorsim - a grid simulator for studying dynamic data replication strategies / W. Bell, D. Cameron, L. Capozza et al. // International Journal of High Performance Computing Applications. "— 2003. "— Vol. 17, no. 4. "— Pp. 403–416.

[14] Overview of a performance evaluation system for global computing scheduling algorithms / A. Takefusa, S. Matsuoka, K. Aida et al. // Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99). "— 1999. "— Pp. 97–104.

[15] Parallel workloads archive. "— http://www.cs.huji.ac.il/labs/parallel/workload/.

[16] Foster I., Kesselman C., Nick J., Tuecke S. The physiology of the grid an open grid services architecture for distributed systems integration. "— 2003.

[17] Platform globus toolkit. "— http://www.platform.com/.

[18] The shared hierarchical academic research computing network. "— www.sharcnet.ca.

[19] Univa. "— http://www.univa.com.