ИСП Российская Академия наук
Институт Системного Программирования

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

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том (Volume) 26

выпуск (issue) 1

Москва 2014

ИСП **Российская Академия наук**
Институт Системного Программирования

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том (Volume) 26
Выпуск (issue) 1

Под редакцией
академика РАН В.П. Иванникова
Edited by
Academician V.P. Ivannikov

Москва 2014

УДК004.45

Труды Института системного программирования: Том 26, выпуск 1. /Под ред. Академика РАН В.П. Иванникова/ – М.: ИСП РАН, 2014.

Proceedings of the Institute for System Programming: Volume 26, issue 1. /Edited by Academician V.P. Ivannikov/ – M.: ISP RAS, 2014.

Этот выпуск Трудов ИСП РАН приурочен к 20-летнему юбилею Института. Последние 15 лет Институт издает Труды ИСП РАН, в которых публикуются наиболее значительные результаты, полученные сотрудниками Института и другими российскими специалистами в области системного программирования. В статьях юбилейного выпуска Трудов ИСП РАН описываются наиболее важные направления исследований и результаты, полученные сотрудниками Института в последние годы.

This issue of the Proceedings of ISP RAS is dated for the 20th anniversary of the Institute for System Programming. The last 15 years, the Institute for System Programming publishes Proceedings of ISP RAS, which contains the most significant results of researchers of ISP RAS and other experts in the field of system programming. This anniversary issue contains articles described research results of studies carried out in last years.

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

Содержание

П р е д и с л о в и е

Этот выпуск Трудов ИСП РАН приурочен к 20-летнему юбилею Института. Последние 15 лет Институт издает Труды ИСП РАН, в которых публикуются наиболее значительные результаты, полученные сотрудниками Института и другими российскими специалистами в области системного программирования. С 2010 г. Труды ИСП РАН публикуются в виде периодического электронного издания (http://ispras.ru/ru/proceedings) с последующей публикацией небольшим тиражом в бумажной форме.

Основными задачами Института были и остаются фундаментальные исследования в области системного программирования, прикладные исследования и разработки в интересах различных областей индустрии, а также образование.

Результаты исследований и разработок публикуются в выпусках Трудов ИСП РАН, ведущем российском журнале «Программирование», авторитетных зарубежных изданиях, докладываются на признанных российских и международных конференциях. Проекты, выполняемые специалистами Института, поддерживаются грантами РФФИ, Министерства образования и науки, Президиума РАН и Отделения математики. Прикладные исследования и разработки выполняются на основе контрактов с российскими и зарубежными компаниями.

Сотрудники Института активно занимаются преподавательской деятельностью на кафедрах системного программирования факультетов ВМиК МГУ им. М.В. Ломоносова и ФУПМ МФТИ. Многие студенты этих кафедр активно участвуют в исследовательской работе отделов Института, поступают в аспирантуру своих университетов или ИСП РАН и остаются работать в ИСП РАН. В результате в ИСП РАН работает много талантливых молодых специалистов, активно участвующих в проектах Института, а зачастую и руководящих ими. Это внушает уверенность, что и в будущем ИСП РАН сможет плодотворно решать актуальные и сложные проблемы системного программирования.

В статьях юбилейного выпуска Трудов ИСП РАН описываются наиболее важные направления исследований и результаты, полученные сотрудниками Института в последние годы. Первый блок статей посвящен тематике применения формальных моделей при тестировании промышленных программных и аппаратных систем. В основе текущих исследований и разработок лежит общая технология автоматизированного создания тестов UniTESK, развиваемая в ИСП РАН на протяжении более 15 лет. В статьях

этого блока описывается процесс развития и совершенствования UniTESK, теоретические основы технологии, опыт применения UniTESK и более специализированных технологий при тестировании операционных систем, реализаций телекоммуникационных протоколов, микропроцессоров и средств авионики.

Второй блок статей связан с технологиями компиляции и оптимизации программ, статического и динамического анализа программ, запутывания и распутывания программ и т.д. Сотрудники Института активно применяют и совершенствуют известные программные системы с открытыми исходными кодами, создают собственные системы с новыми функциональными возможностями, разрабатывают новые подходы оптимизации и анализа программ. Важной областью исследований и разработок является создание и развитие Web-лаборатории, основанной на использовании облачной среды и позволяющей облегчить организацию численных экспериментов и других видов исследований, выполняемых группами специалистов.

Небольшой блок статей посвящен актуальным проблемам аналитики. Описывается архитектура и этапы развития системы поддержки интеллектуального анализа текстов на естественных языках Texterra. В основе подхода лежит использование в качестве прообраза онтологии публично развиваемой электронной энциклопедии Википедия. Во второй статье этого блока обсуждаются методы и существующие проблемы актуального направления анализа социальных сетей.

Кроме того, в юбилейном выпуске содержится статья, в которой предлагается и обсновывается новый подход к составлению расписаний организации сложных производственных проектов с учетом пространственных факторов, а также статья, в которой обобщаются ранее полученные результаты решения практически важной задачи упаковки набора прямоугольников в группу полубесконечных полос различной ширины.

Как видно, тематика исследований и разработок, проводимых в Институте, широка и разнообразна. Вполне вероятно, что в будущем эта тематика изменится. Наша цель состоит в том, чтобы в любом случае поддерживать высокое качество работы, продолжать обучать молодежь и обеспечивать поддержку высококвалифицированого научного коллектива

Академик РАН В.П. Иванников

Развитие подхода к разработке тестов UniTESK

В. В. Кулямин, А. К. Петренко
{kuliamin,petrenko}@ispras.ru

Аннотация. В статье излагаются основные принципы, на которых основана технология UniTESK, предназначенная для создания тестов на основе формальных моделей. Суммируется опыт использования UniTESK в крупных проектах по разработке тестов для промышленных программных и аппаратных систем, включающих телекоммуникационные протоколы, базовые и стандартные интерфейсы операционных систем, блоки микропроцессоров. Дается обзор возможных направлений развития технологии в целях обеспечения ее большей масштабируемости.

Ключевые слова: тестирование на основе моделей, автоматизация тестирования, формальные спецификации, программные контракты, масштабируемость тестов.

1. Введение

Развитие технологий создания программного обеспечения (ПО) привело в последние десятилетия к беспрецедентному росту сложности создаваемых программных систем. Однако столь же разительного прогресса в технологиях обеспечения качества не наблюдается. Все более настоятельно ощущается необходимость в масштабируемых промышленно применимых технологиях обеспечения и контроля качества, которые позволяли бы разрабатывать надежные системы на современном уровне сложности.

Один из перспективных подходов к решению этой проблемы — использование максимально автоматизированных техник контроля качества совместно с компонентными технологиями, позволяющими строить иерархически скомпонованные программные системы высокой сложности. Примером подобной технологии является технология автоматизированного создания тестов UniTESK [1,2], разрабатываемая в ИСП РАН уже более 15 лет.

В данной статье дан обзор основных элементов технологии UniTESK, прослеживаются основные применения с момента создания первых прототипов поддерживающих ее инструментов, а также представлено несколько направлений развития технологии, связанных с потребностью в более масштабируемых тестах и инфраструктуре для их эффективного выполнения и удобного анализа их результатов.

2. Основные элементы технологии UniTESK

Тестированием называется проверка выполнения требований к системе при помощи наблюдения за ее работой в конечном наборе специально выбранных ситуаций [3]. поэтому любая технология построения тестов должна прежде всего обеспечивать решение следующих двух задач.

- Тесты должны проверять *требования* к проверяемой системе.
- Ситуации, используемые в тестах, должны обеспечивать определенную представительность по отношению ко всем возможным вариантам поведения проверяемой системы, иначе выводы о качестве системы, сделанные на основе проведенного тестирования, будут недостоверны. Данное свойство тестового набора принято называть полнотой тестирования и характеризовать с помощью выбираемых *критериев полноты*, задающих разбиения пространства всех возможных ситуаций на классы эквивалентности точки зрения возможных ошибок — если в одной из ситуаций данного класса возникает ошибка, она с большой вероятностью проявляется и в других ситуациях этого класса, если же система работает корректно, то и в других ситуациях того же класса это, скорее всего, так.

Кроме того, эффективность технологии построения тестов определяется трудоемкостью, или, наоборот, удобством, выполнения в ее рамках следующих действий.

- Разработка элементов тестов, создание которых невозможно (или крайне сложно) автоматизировать для широких классов систем. Примерами подобных элементов являются процедуры проверки требований или процедуры формирования определенных воздействий на систему через соответствующий интерфейс — они обычно специфичны для заданной системы или достаточно узкого семейства систем.
- Развертывание набора тестов перед выполнением в рабочем окружении.
- Выполнение набора тестов, включающее мониторинг важных для последующего анализа событий, поддержание проверяемой системы в рабочем состоянии, отслеживание обнаруживаемых сбоев и адекватное реагирование на них.
- Анализ результатов тестирования на предмет аккуратного выявления характеристик обнаруженных сбоев, их возможной локализации, а также полноты проведенного тестирования.
- Сопровождение тестового набора, включающее поддержку его работоспособности и развитие в соответствии с изменяющимися требованиями к проверяемой системе, а также изменениями в ее окружении и интерфейсах.

В рамках UniTESK для достижения достаточно широкой применимости технологии и уменьшения трудозатрат на создание тестов заданного уровня качества используются следующие решения.

- Предложена *унифицированная архитектура тестового набора* [1], определяющая набор компонентов тестов с четким разделением функций и заданными интерфейсами и нацеленная на реализацию в ее рамках большого многообразия тестов различных типов.
  Основные виды компонентов тестов: *тестовый оракул* [4], осуществляющий проверку соответствия наблюдаемого поведения отдельного компонента проверяемой системы требованиям; *тестовый сценарий* (см. ниже). При необходимости используются другие виды компонентов [5]: адаптеры, заглушки, генераторы данных, и т.п.
  Чтобы достичь высокой степени автоматизации, вся информация, которая может быть предоставлена только человеком, сконцентрирована в небольшом числе компонентов. Все остальные компоненты теста генерируются автоматически или используются во всех тестах в неизменном виде.
- Требования к проверяемой системе описываются в виде *программных контрактов* [6] с состоянием, состоящих из предусловий и постусловий интерфейсных операций и инвариантов компонентов, написанных в терминах работы с некоторым *модельным состоянием* компонента (т.е., структурой данных, позволяющей описать поведение операций, но, возможно, не совпадающей с той структурой данных, которая использована при его реализации). Программные контракты позволяют зафиксировать требования в виде, достаточно близком к их исходной формулировке (в частности, как в декларативном, так и в процедурном стиле), что облегчает дальнейшее развитие тестов при возникновении изменений в требованиях. Кроме того, программные контракты легко использовать для автоматической генерации тестовых оракулов.
  В ходе развития технологии контракты (и другие, создаваемые вручную компоненты) сначала описывались на специализированном формальном языке RSL (в рамках технологии KVEST [7], предшественника UniTESK), затем на расширениях широко используемых языков программирования [8,9], позже — с помощью набора библиотечных функций специального вида [10]. При этом, похоже, основную методическую роль при описании требований и построении тестов на их основе играет не сам используемый язык, а вкладываемая в него система понятий, соответствующих элементам архитектуры теста: проверяемый компонент, его интерфейс, интерфейсная операция, ее пред- и постусловия, модельное состояние компонента, его инвариант, пре-выражение в постусловии, вычисляемое до обращения к соответствующей операции. Кроме того,

существенно облегчает оформление контрактов использование конструкций, позволяющих учитывать особенности задания интерфейсов в языке, используемом для разработки тестируемой системы. Например, в языках, использующих исключения, они являются элементом интерфейса операций, и для адекватного описания постусловий необходимы специальные конструкции или библиотеки, позволяющие задавать требования к возникновению или отсутствию исключения, а также свойства возникающего исключения.

- Для описания асинхронного поведения и параллелизма используется специфическая техника [11-13]. Взаимодействие через асинхронный интерфейс представляется наборами событий, часть которых создается окружением проверяемой системы (это вызовы интерфейсных операций и сообщения, предназначенные системе), а все остальные — самой системой (это возвраты управления из интерфейсных операций и создаваемые системой сообщения). События описываются при помощи программных контрактов, каждый вид событий имеет пред- и постусловие, в частности, постусловия определяют, как меняется модельное состояние системы. При анализе взаимодействий теста с системой по нескольким каналам используется *семантика чередования*: наблюдаемый набор событий линейно упорядочивается так, чтобы в полученной цепочке сохранялся наблюдаемый порядок (события, получаемые по одному каналу, упорядочены), а также чтобы пред- и постусловия всех событий в цепочке выполнялись (т.е. модельное состояние, определяемое постусловием предшествующего события, удовлетворяло предусловию следующего). Если такую цепочку построить из данного набора событий невозможно, значит выявлено нарушение требований, зафиксированных в контрактах.

  Сам тест при этом разбивается на последовательность шагов, в рамках которых происходят отдельные этапы взаимодействия с тестируемой системой. На каждом шаге создается необходимый для воспроизводства определенной ситуации набор событий, после чего тест приостанавливается, пока не произойдут все события, генерируемые системой в ответ, т.е. пока новые события не перестанут возникать — *состояние покоя* (*quiescence*). Обычно наступление состояния покоя фиксируется по истечении некоторого времени после наступления последнего созданного системой события. Набор событий, произошедших на данном этапе, подвергается анализу с помощью описанной выше процедуры. При обнаружении линейного порядка событий, удовлетворяющего контрактам, считается, что текущее состояние проверяемой системы соответствует модельному состоянию в конце найденной цепочки. После этого тест переходит на следующий шаг, если такой имеется. Каждый отдельный

такой шаг считается нерасчленяемым (наблюдаемым образом) асинхронным воздействием на проверяемую систему.

- Для автоматического построения последовательности тестовых воздействий в тестах используется модель тестируемой системы (или одного/нескольких тестируемых компонентов) в виде *конечного автомата* (называемого *автоматом теста*). Тестовая последовательность строится как последовательность обращений к тестируемым операциям (или асинхронных воздействий в смысле, описанном в предыдущем пункте), соответствующая определенному пути в графе переходов автомата, например, обходу всех его переходов [14,15]. Автомат теста задается не полностью, а лишь в виде процедуры вычисления текущего состояния и набора действий (каждое действие может соответствовать некоторому набору создаваемых событий или цепочке вызовов интерфейсных операций), которые можно выполнить в произвольном состоянии. Используются также охранные условия, запрещающие выполнение определенных действий в некоторых состояниях. Такое описание называется *тестовым сценарием.* Действия могут быть параметризованы, и в этом случае для построения используемых при тестировании наборов значений параметров необходимо дополнительное указание источников или генераторов данных.
- В качестве критериев полноты тестирования в UniTESK применяются, в первую очередь, критерии покрытия кода контрактов, близкие к критериям покрытия требований. Критерии, устанавливаемые в качестве целей тестирования, служат основой при формировании структуры состояния тестового сценария и используемых тестовых данных, т.е., могут быть учтены лишь человеком-разработчиком тестов. Критерии полноты, отслеживаемые во время тестирования, должны быть преобразованы в критерий покрытия какого-либо кода [16] — для их контроля при выполнении тестов создаются специальные компоненты-мониторы покрытия, покрытие кода которых измеряется с помощью обычных средств.

В целом набор решений технологии UniTESK, как это подтверждается опытом ее применения в промышленной разработке, позволяет при помощи приемлемых трудозатрат создавать тесты высокого качества для весьма сложных систем [9,17], создание аналогичных тестов для которых традиционными методами требует гораздо больше ресурсов. Особенно заметным эффект применения технологии становится при длительном сопровождении тестового набора — за распределения информации, относящейся к разным аспектам тестов по различным компонентам, достигается существенное снижение затрат на сопровождение.

3. Опыт использования UniTESK

Технология UniTESK с момента создания прототипного набора поддерживающих ее инструментов применялась для разработки тестов для различного промышленного ПО.

- Базовые библиотеки и стандартные интерфейсы операционных систем (ОС). Такие проекты включают создание регрессионного набора тестов для ядра операционной системы Nortel Networks, выполнявшую еще, для помощи прототипа UniTESK, KVEST [7], создание тестового набора для отечественной ОС реального времени ОС 2000/ОС 3000, создание тестового набора OLVER [18] для базовой части стандарта Linux Standard Base [17] (LSB), включающей POSIX. В эту же категорию попадает разработка тестового набора для проверки соответствия стандарту ARINC 653 [19].
- Библиотеки времени выполнения языков программирования.
  Значительная часть базовых библиотек языка C покрывается проектом OLVER. Кроме того, технология UniTESK использовалась при разработке тестов для одной из реализаций инфраструктуры поддержки языка Java.
- Телекоммуникационные протоколы.
  Технология использовалась при создании тестовых наборов для проверки соответствия стандартам протоколов IPv6 [9], Mobile IPv6 [20], IPsec, а также таких протоколов, как SMTP, TFTP и пр.
- Имитационное тестирование моделей аппаратного обеспечения.
  UniTESK используется также для создания тестов для Verilog-моделей отдельных блоков и компонентов микропроцессоров и шин [21].

Разработанные при помощи UniTESK тестовые наборы обладают высокой сложностью, как из-за большого числа проверяемых интерфейсов, так и потому что они позволяют проводить аккуратное тестирование сложной функциональности. Самым масштабным набором тестов является OLVER: его код составляет около 500 тыс. строк на расширении языка C, тесты нацелены на проверку более 1500 функций LSB Core, разработанные контракты покрывают требования. размещенные на 6000 страниц текста стандартов LSB, POSIX, X/Open Curses, System V Interface Definition, ISO/IEC 9899 (стандарт языка программирования C). Разработанный для OLVER набор контрактных спецификаций на момент окончания проекта являлся самым объемным реально используемым набором формальных спецификаций в мировой практике (в дальнейшем аналогичный или несколько превосходящий по объему набор формальных моделей был создан в рамках проекта Microsoft по обеспечению открытости и интероперабельности протоколов взаимодействия, используемых в ее продуктах [22]). В большинстве приведенных примеров тестовых наборов использована специфическая для UniTESK техника тестирования асинхронных интерфейсов [11-13], позволяющая обнаруживать достаточно сложные ошибки.

Остановимся на наиболее интересных примерах использования технологии UniTESK и на полученных в при этом уроках (несколько более подробно об этом можно прочитать в [31]).

Первым применением UniTESK был поддержанный Microsoft Research проект по разработке тестового набора для реализации IPv6 [20]. Проект стартовал в 2000 году. Тогда UniTESK только начинал создаваться, поэтому пришлось использовать облегченную реализацию технологии на языке C — CTESK-light. Несмотря на нестабильность инструмента удалось построить эффективный тестовый набор, который нашел дефекты, ранее не обнаруженные другими тестовыми наборами. Это был первый опыт использования контрактных спецификаций для тестирования телекоммуникационных протоколов. Было показано, что контрактные спецификации в сочетании с техникой тестирования систем с асинхронными интерфейсами, разработанной в рамках UniTESK [13,37] позволяют строить эффективные тесты. Эти тесты выявили больше ошибок и требовали меньше усилий для создания и сопровождения при заданном уровне качества тестирования, чем тесты, построенные по традиционным технологиям. Вместе с тем, опыт тестирования протоколов показал, что для эффективной генерации последовательностей тестовых воздействий при тестировании протоколов полезно иметь и исполнимые модели.

Одним из первых опытов использования UniTESK для тестирования программных компонентов Java [8] через программный интерфейс (Application Programming Interface, API) стал проект тестирования одной из реализаций стандартной библиотеки поддержки времени исполнения Java. Разработка моделей и тестов не вызывала особых проблем, так как интерфейсы были хорошо документированы. Помимо интерфейсов на Java в системе также были интерфейсы на C++, но больших проблем и это не вызвало, поскольку архитектура тестов UniTESK предусматривает слой тестовых адаптеров. Более серьезные проблемы появились, когда началось собственно тестирование, в рамках которого генератор тестовой последовательности должен был работать на базе самой тестируемой системы, еще не стабильной в это время.

Одним из значимых примеров применения UniTESK на платформе Java является проект тестирования инфраструктуры распределенной информационной системы одного из крупных операторов мобильной связи, который продолжается и сейчас. Возможность формальной и строгой фиксации интерфейсов компонентов этой чрезвычайно большой и разнородной системы стала для заказчика самым главным преимуществом UniTESK по сравнению с другими инструментами функционального тестирования. В рамках проекта были формально специфицированы и протестированы сотни компонентов. Уже к концу первого года применения технологии положительный эффект проявился в ускорении сроков интеграции новых версий распределенной системы. Вместе с тем, вскрылась серьезная проблема. Если в предыдущих проектах применения UniTESK требования к

большей части интерфейсов определялись стандартами или другими тщательно разработанными документами, здесь уровень документирования часто оказывался недостаточным для построения консистентных спецификаций. Восстановление документации или требований к интерфейсам в системах такого размера оказывается практически неразрешимой задачей, что не позволяет использовать MBT в полном объеме. О путях решения этой проблемы будет кратко сказано в Заключении.

Самым крупным примером применения UniTESK стал проект OLVER (Open Linux VERification) [18], который проводился в 2005-2007 годах при поддержке Министерства образования и науки РФ. Целью проекта была формальная спецификация интерфейсов стандарта Linux Standard Base (LSB), точнее его центральная части LSB Core. В LSB Core входят наиболее важные библиотеки операционной системы Linux, которые в значительной части реализуют стандарт POSIX. Строгое описание стандарта LSB и наличие набора тестов, который мог бы качественно проверить соответствие разных реализаций библиотек Linux требованиям стандарта, являются необходимыми условиями для обеспечения переносимости приложений для Linux с одного дистрибутива на другой. Проблема переносимости приложений под Linux является крайне острой, поскольку сейчас доступно уже несколько сотен различных дистрибутивов. Результаты проекта опубликованы [18]. Были построены контрактные спецификации более чем 1500 интерфейсов на языке C. Инструментом моделирования и генерации тестов был выбран CTESK. В ходе проекта были выявлены проблемы в самих стандартах: LSB (ISO/IEC 23360) и The Single UNIX Specification, основную часть которого составляет стандарт POSIX.1 (он же IEEE Std 1003.1, он же ISO/IEC 9945, он же The Open Group Base Specifications Issue 6). Разработанный тестовый набор включен в пакет сертификационных тестов международного консорциума The Linux Foundatuion [38].

Опыт формализации интерфейсов большого индустриального стандарта и разработки тестового набора для него дал много полезных уроков. Одним из них стало понимание важности организации информационного и методического обеспечения такого проекта. Массив документации и исходных текстов библиотек Linux, особенно с учетом многочисленности версий и вариантов, связанных с различными аппаратными платформами, является огромным. Кроме того, в разработку собственно стандарта и в разработку его реализаций вовлечены тысячи людей, распределенных по всему миру. Из этого следует, что организация «документооборота» является одной из важнейших составляющих проектов такого калибра. В организационно-методическом плане мы столкнулись с тем, что обучение новых сотрудников и контроль за качеством спецификаций и тестов требует много усилий и при этом быстро достичь необходимого уровня профессионализма невозможно. То есть, масштабируемость проектов по использованию MBT в плане расширения числа участников проекта, задействованных в самой работе по

спецификации требований и отладке тестов, — это одна из самых сложных проблем, мешающая широкому внедрению MBT.

Одной из методических проблем является выбор уровня абстракции, на котором строится модель. Более абстрактные модели или разделение моделей на два-три слоя, отличающиеся уровнем абстракции, упрощают задачу переиспользования моделей и тестов, при этом общий размер системы возрастает, и ее сложность также растет. В долгосрочном плане выгоднее иметь многослойные модели, в краткосрочном — модели, близкие по уровню детальности к реализации, конечно, если реализация уже есть. Профессиональный опытный верификатор умеет находить баланс между абстрактным описанием поведения, например, файловой системы и особенностями, деталями интерфейса конкретной реализации файловой системы. В UniTESK имеется специальная поддержка разделения уровней абстракции. В частности, специфика конкретных интерфейсов может быть скрыта в слой адаптеров. Выбор баланса во многом определяется долгосрочными планами по использованию и развитию моделей и тестового набора. То есть, такого рода работа требует достаточно широкого кругозора, чего трудно требовать то обычных инженеров-тестировщиков.

Результаты проекта OLVER впоследствии были использованы в разработке тестового набора для российской операционной системы реального времени ОС 2000/3000 [19]. Эта система поддерживает две группы интерфейсов. Первая отвечает требованиям POSIX, вторая — требованиям ARINC 653, международного стандарта для критических встроенных систем. Выделение уровня адаптеров, разделяющего модельное и реализационное представление интерфейсов, заложенное в архитектуру UniTESK, существенно упростило повторное использование OLVER в данном проекте.

Одновременно с началом работ по OLVER были развернуты работы по применению UniTESK для симуляционного (или имитационного) тестирования моделей отдельных блоков микропроцессоров [21]. Полученные тесты использовались при разработке российских микропроцессоров с архитектурой MIPS и микропроцессоров с элементами VLIM/EPIC. Размер типовых блоков в таких микропроцессорах — несколько миллионов вентилей. Для целей спецификации и генерации тестов не потребовалось больших изменений в инструментах, за основу был взят CTESK. В техническом плане привязка CTESK к API на языке C не стала проблемой, так как большинство симуляторов, работающих с языками моделирования логики микропроцессоров (High Level Design languages, HLD), например, VHDL или Verilog, предоставляют удобный интерфейс для взаимодействия с программами на C. Несколько изменилась семантика предусловий в контрактных спецификациях, они стали описывать не столько разрешенную область входных данных, сколько условия возможности выполнения микрооперации на соответствующем такте, что, кстати, характерно и для семантики предусловий асинхронных событий при тестировании

параллелизма. Так же, как и в случае моделирования протоколов, выявилась потребность в использовании наряду с постусловиями, явных моделей поведения тестируемого устройства.

Как и в проектах по верификации программных систем одной из главных проблем, мешающих внедрению MBT при верификации аппаратного обеспечения (как и большинства других методов верификации), является отсутствие четких и детальных описаний функциональных требований к компонентам. Вместе с тем, ситуация в разработке микропроцессоров несколько лучше, так как в ее ходе принято наряду с HLD моделями строить и системные или архитектурные модели, описывающие семантику набора инструкций. Элементы таких архитектурных моделей можно использовать для восполнения недостающих знаний о поведении некоторых блоков микропроцессоров [39]. Хорошей новостью оказалось достаточно простое решение задачи распараллеливания выполнения теста на кластерах. Типичные размеры конечного автомата, который генерируется при выполнении теста одного сложного блока микропроцессора — это несколько миллионов узлов и десятки миллионов переходов. Оказалось, что генерация теста с помощью обхода неизвестного конечного автомата хорошо распараллеливается на кластерах до 200 узлов, с накладными расходами всего лишь 10-15%. При этом надо помнить, что при симуляционном тестировании высокоуровневой модели устройства основное время уходит на работу симулятора HDL, т.е. такая масштабируемость связана с возможностью запустить на каждом узле отдельный симулятор и выполнять параллельно много действий в различных состояниях. Масштабируемость распараллеливания тестирования систем других типов, скорее всего, будет не такой высокой.

Важно отметить задачи верификации, которые не удалось свести к моделированию на основе контрактных спецификаций, из-за чего они дали толчок для разработки новых методов MBT. В первую очередь следует упомянуть задачу тестирования компиляторов и задачу тестирования микропроцессора в целом, так называемый «core testing». В обоих случаях это задачи системного тестирования, где требуется подавать на вход большого «черного ящика» тестовые воздействия (в нашем случае это тестовые программы, которые подаются на вход компилятору или загружаются в память симулятора микропроцессора), но при этом интересно тестировать не все подряд, а лишь некоторые заданные режимы или заданную группу модулей компилятора или процессора. В случае тестирования компиляторов был разработан инструмент OTK, который использовался для тестирования оптимизирующих компиляторов Intel и Simulink [40,41]. Он позволяет нацеливаться на заданные виды оптимизаций. Для системного тестирования микропроцессоров был разработан инструмент MicroTESK [42-44]. Основной задачей этого инструмента была проверка разнообразных ситуаций, связанных с наиболее сложными подсистемами управления памяти: блоком трансляции адресов, кэшами разных уровней или блоком управления памятью в целом.

4. Направления развития технологии

Как показывает опыт применения UniTESK в больших проектах, развитие самого процесса разработки тестов в сторону повышения степени автоматизации возможно, однако, сводится, в основном, к проблемно-ориентированным решениям. Обычно можно спроектировать набор шаблонов тестов, использование которых может существенно облегчить создание тестов для конкретной системы или узкого семейства систем, но конкретный вид этих шаблонов становится ясен тогда, когда большая часть необходимых тестов уже разработана и отлажена. Предложить же аналогичное решение, способное снизить трудоемкость создания тестов во всех описанных выше примерах, крайне трудно.

Единственным исключением здесь является задача построения тестовых данных, которая, несмотря на значительный прогресс в развитии специализированных инструментов-решателей и рост числа их приложений для генерации тестов [23-26], не поддается прямолинейным решениям — практически все опубликованные подходы пригодны для автоматического построения только входных данных тестового воздействия, если те представляют собой даже достаточно объемные, но практически однородные наборы символов или чисел, и не принимают в расчет состояние тестируемой системы и сложные структуры входных данных (по сути также означающие необходимость создавать сложные объекты в различных состояниях).

Продвижение в сторону повышения степени автоматизации при построении тестовых данных в более общем случае возможно, скорее, на пути интеграции появившихся техник смешанного, символического и реального, исполнения тестируемого фрагмента кода [23-26] и автоматического обхода графа состояний сложного объекта за счет обращений к его интерфейсным методам.

Есть, также, еще ряд проблем, решение которых способно в достаточно общем случае повысить эффективность применения технологии UniTESK. Эти проблемы так или иначе связаны с масштабируемостью выполнения тестов и анализа их результатов. Дело в том, что технология уже позволяет создавать качественные тесты для достаточно сложной функциональности, но чисто количественное повышение их сложности, увеличивающее аккуратность тестирования, но не требующее существенных усилий от разработчиков (обычно за счет использования больших значений каких-то параметров, более сложных структур состояний, большего числа одновременно тестируемых операций, большего числа параллельно оказываемых воздействий и пр.) практически всегда приводит к значительному росту ресурсов, необходимых для выполнения таких тестов и анализа получаемых трасс, в т.ч. обнаруженных ошибок. С другой стороны, успешное повышение масштабируемости тестов может позволить решать задачи, в которых сейчас нужно тщательно выбирать структуру состояния и реализовывать достаточно сложные генераторы данных, за счет «грубой силы», используя простейшие решения, что также снизит общую трудоемкость создания тестов. Видимые

уже сейчас возможности по улучшению масштабируемости тестов следующие.

- Масштабируемость выполнения тестов может быть достигнута, прежде всего, за счет его *распараллеливания*. Сейчас чаще всего в тестах используются автоматы с сотнями состояний и тысячами переходов. Возможность выполнять тесты с миллионами состояний и десятками миллионов переходов за разумное время уже реализована за счет простейшего распределения копий тестируемой системы по узлам кластера [27] и использования общего состояния теста, что оказалось пригодным для тестирования сложных блоков аппаратного обеспечения. Еще больших показателей можно добиться, применяя специализированные параллельные алгоритмы исследования автоматов, не требующие общей памяти [28].
- Другой возможный источник повышения эффективности выполнения тестов — *редукция сериализаций*, выполняемых при поиске линейного упорядочения набора наблюдаемых событий на одном шаге теста асинхронной системы. В имеющихся инструментах строятся все возможные линейные порядки на данном наборе событий (отсекаются лишь те цепочки, которые являются продолжением невозможной из-за нарушения контрактов последовательности событий). Из-за возникающего комбинаторного взрыва (нужно еще учесть, что подобную сериализацию надо проводить на каждом использующем асинхронный интерфейс переходе некоторого автомата) за разумное время можно провести тестирование с использованием двух-трех видов асинхронных воздействий на каждом шаге (они обычно пополняются еще тремя-пятью асинхронными событиями, создаваемыми тестируемой системой), при возрастании этого числа до десятка время работы теста становится неприемлемым. Перспективными подходами к эффективному выполнению таких тестов являются использование техник динамической редукции частичных порядков [29,30] или выявления симметрий набора событий с помощью их символического выполнения.
- При выполнении сложных тестов, содержащих тысячи вызовов отдельных операций в сотнях различных состояний возникает достаточно объемная трасса (в упомянутых выше проектах трассы тестов могут достигать объема в десяток гигабайт). В случае обнаружения ошибок провести аккуратный анализ такого объема информации и четко выявить характеристики ошибки, чтобы можно было передать ее описание разработчикам проверяемой системы, нелегко. Еще хуже становится ситуация, если при работе одного теста выявляется несколько различных ошибок, причем каждая проявляется еще и в разных ситуациях десятки раз. Прекращение работы теста после каждого сбоя является не слишком эффективным решением —

такой тест надо будет перезапускать много раз, пока не будут устранены все проявляющиеся в ходе его работы ошибки, а каждое выполнение требует значительного времени.

Разумным выглядит такой сценарий работы сложных тестов.

- Если тест выполняется без обнаружения сбоев, после его работы остается только отчет о достигнутом тестовом покрытии и общее заключение об успешном выполнении.
- При обнаружении сбоя запоминается весь путь, приводящий к нему, после чего тест инициализирует тестируемую систему заново и продолжает работу, учитывая ранее полученную информацию (т.е., обнаруженные состояния и пройденные переходы в автоматной модели теста).
- После окончания работы теста с обнаруженными сбоями выполняется кластеризация этих сбоев по логическим признакам, описывающим соответствующие ситуации (эти признаки включают данные состояний, в которых происходят сбои, а также все данные выполняемых на пути до сбоя операций). Кластеризация необходима при обнаружении значительного числа (десятки) сбоев, поскольку каждый из них должен подвергаться тщательному анализу. Если удается автоматически сформировать разумные гипотезы об одинаковой природе различных сбоев и свести анализ всего лишь к нескольким ситуациям, трудоемкость его существенно снижается.
- После получения набора кластеров сбоев для каждого такого кластера (с учетом гипотезы об общих характеристиках попадающих в него ситуаций) строится наиболее короткий путь по графу автомата теста, повторяющий в конце этот сбой. Поскольку автомат теста является обычно лишь достаточно абстрактной моделью проверяемой системы, не учитывающей большое количество деталей ее работы, такой путь часто не соответствует кратчайшему пути по графу переходов автомата, и для его нахождения требуется перебор различных подпоследовательностей вызовов в исходной последовательности, приводящей к данному сбою.
- Таким образом, по итогам работы теста, обнаружившего ошибки, помимо отчета о достигнутом покрытии, остается список обнаруженных сбоев с их автоматически проведенной кластеризацией (сбои одного кластера, вероятно, являются проявлением одной и той же ошибки) и, для каждого кластера автоматически формируются набор условий, как можно точнее характеризующих попадающие в него ситуации, и кратчайшая последовательность вызовов операций, демонстрирующая такой сбой.

Тестировщику должна быть предоставлена возможность провести вручную перегруппировку обнаруженных сбоев с последующим автоматическим поиском минимальных подтверждающих тестов для новых групп.

Обозначенные в данном разделе направления развития технологии UniTESK находятся пока на стадии исследований. Проведены работы по проектированию алгоритмов параллельного обхода неизвестных графов без использования общей памяти, а также предложены некоторые техники построения минимальных тестовых последовательностей, демонстрирующих обнаруженную ошибку.

5. Заключение

В статье изложены основные элементы технологии UniTESK, обеспечивающие ее применимость для создания качественных тестов сложных систем и снижение трудоемкости сопровождения сложных тестовых наборов. Также суммирован опыт использования UniTESK в промышленных проектах разработки тестов, и представлено несколько направлений дальнейшего развития технологии.

Поскольку дальнейшее повышение сложности тестируемых систем требует более аккуратного тестирования, перспективы развития технологии связаны с обеспечением более эффективного выполнения объемных и сложных тестов, основанном на распараллеливании. Другое важное направление развития — повышение уровня автоматизации при анализе результатов тестов, поскольку анализ трасс из миллионов обращений к проверяемой системе слишком трудоемок для человека.

Работы по развитию технологии послужили отправной точкой как для создания специализированных средств верификации программ, так и для работ по развитию теории верификации. Текущее состояние этих исследований и разработок представлено в серии статей, которые в совокупности представляют собой достаточно полное описание направлений развития работ по созданию UniTESK.

Так в работе [32] описываются результаты развития теории построения тестов, проверяющих соответствие требованиям, в приложении к тестированию программных систем; в работе [33] рассматриваются результаты полученые в области тестирования операционных сиситем и новые задачи, которые предстоит решить в ближайшем будущем; в работе [34] рассматриваются вопросы использования UniTESK для автоматизации тестирования соответствия для телекоммуникационных протоколов; в работе [35] рассматриваются разработанные техники тестирования микропропроцессоров; в работе [36] рассматриваются вопросы построения интегрированной системы проектирования и верификации модульной авионики, в которой решение собственно задач тестирования увязываются и с другими задачами

жизненного цикла разработки ответветственного программного обеспечения, в частности с задачами анализа требований и систематического контроля выполнения требования на всех фазах жизненного цикла программ.

Литература

[1]. I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. *UniTesK Test Suite Architecture.* Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77-88, Springer-Verlag, 2002.

[2]. В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. *Подход UniTesK к разработке тестов.* Программирование, 29(6):25-43, 2003.

[3]. ISO/IEC TR 19759 *Software Engineering — Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Geneva, Switzerland: ISO, 2005.

[4]. L. Baresi, M. Young. *Test Oracles*. Tech. Report CIS-TR-01-02. 2001, http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

[5]. В. В. Кулямин. *Организация сложных тестовых наборов*. Труды ИСП РАН, 17:9-24, 2009.

[6]. B. Meyer. *Applying Design by Contract*. IEEE Computer,25(10): 40-51, October 1992.

[7]. I. Bourdonov, A. Kossatchev, A. Petrenko, D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications*. Proceedings of FM'99, Toulouse, France, LNCS 1708:608-621, Springer-Verlag, 1999.

[8]. I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, S. V. Zelenov. *Java Specification Extension for Automated Test Development.* Proceedings of PSI'2001, Novosibirsk, Russia, LNCS 2244:301-307, Springer-Verlag, 2001.

[9]. Г. В. Ключников, А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. *Применение формальных методов для тестирования реализации IPv6.* Труды ИСП РАН, 4:121-140, 2003.

[10]. В. В. Кулямин. *Компонентная архитектура среды для тестирования на основе моделей*. Программирование, 36(5):54-75, 2010.

[11]. V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin, A. S. Kossatchev, I. B. Bourdonov. *Integration of Functional and Timed Testing of Real-time and Concurrent Systems*. Proceedings of PSI'2003, Novosibirsk, Russia, LNCS 2890:450-461, Springer-Verlag, 2003.

[12]. V. Kuliamin, A. Petrenko, N. Pakoulin. *Practical Approach to Specification and Conformance Testing of Distributed Network Applications*. Proceedings of ISAS'2005, Berlin, Germany, LNCS 3694:68-83, Springer-Verlag, 2005.

[13]. А.В.Хорошилов. *Спецификация и тестирование компонентов с асинхронным интерфейсом*. Диссертация на соискание ученой степени к.ф.-м.н., Москва, 2006.

[14]. И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. *Использование конечных автоматов для тестирования программ*. Программирование, 26(2):61-73, 2000.

[15]. И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов: детерминированный случай.* Программирование, 29(5):59-69, 2003.

[16]. H. Zhu, P. A. V. Hall, J. H. R. May. *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys, 29(4):366-427, Dec. 1997.

[17]. A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov. *Formal Methods in Industrial Software Standards Enforcement.* Proceedings of PSI'2006, Novosibirsk, Russia, LNCS 4378:459-469, 2006.

[18]. Проект OLVER, http://linuxtesting.org
[19]. A. Maksimov. *Requirements-Based Conformance Testing of ARINC 653 Real-Time Operating Systems*. Proceedings of Data Systems In Aerospace (DASIA) 2010, ESA SP-682, ISBN 978-92-9221-246-9, 2010.
[20]. Г. В. Ключников, А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. *Применение формальных методов для тестирования Mobile IPv6.* Сборник тезисов 2-й международной конференции «Интернет нового поколения», стр. 20-25, Ярославль, Россия, 2003.
[21]. В. П. Иванников, А. С. Камкин, А. С. Косачев, В. В. Кулямин, А. К. Петренко. *Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры*. Программирование, 33(5):47-61, 2007.
[22]. W. Grieskamp. *Microsoft's Protocol Documentation Program: A Success Story for Model-Based Testing*. Testing – Practice and Research Techniques. Lecture Notes in Computer Science, vol. 6303, p. 7, Springer, 2010.
[23]. P. Godefroid, N. Klarlund, K. Sen. *DART: Directed Automated Random Testing*. ACM SIGPLAN Notices — Proceedings of PLDI 2005, 40(6):213-223, 2005.
[24]. K. Sen, D. Marinov, G. Agha. *CUTE: a concolic unit testing engine for C*. Proceedings of ESES/FSE, pp. 263–272, 2005.
[25]. C. Cadar, V. Ganesh, P. Pawloski, D. Dill, D. Engler. *EXE: Automatically Generating Inputs of Death*. Proceedings of the 13-th International Conference on Computer and Communications Security CCS 2006, pp. 322-335.
[26]. C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. *Feedback-Directed Random Test Generation*. Proc. of International Conference on Software Engineering, pp. 75-84, 2007.
[27]. И. Б. Бурдонов, С. Г. Грошев, А. В. Демаков, А. С. Камкин, А. С. Косачев, А. А. Сортов. *Параллельное тестирование больших автоматных моделей.* Вестник ННГУ, № 3, 2011, стр. 187-193.
[28]. И. Бурдонов, А. Косачев. *Обход неизвестного графа коллективом автоматов*. Труды Международной суперкомпьютерной конференции "Научный сервис в сети Интернет: все грани параллелизма". 2013, стр. 228-232.
[29]. C. Flanagan, P. Godefroid. *Dynamic Partial-Order Reduction for Model Checking Software*. ACM SIGPLAN Notices— Proceedings of POPL 2005, 40(1):110-121, 2005.
[30]. Y. Yang, X. Chen, G. Gopalakrishnan, R. Kirby. *Efficient Stateful Dynamic Partial Order Reduction*. Proceedings of SPIN 2008, LNCS 5156:288-305, Springer, 2008.
[31]. В. П. Иванников, А. К. Петренко, В. В.Кулямин, А. В. Максимов. *Опыт использования UniTESK как зеркало развития технологий тестирования на основе моделей.* Труды ИСП РАН, 23:207-218, 2013.
[32]. И.Б.Бурдонов, А.С.Косачев. *Развитие теории конформности: семантики, формальные модели, алгоритмы*. Труды ИСП РАН, 2014.
[33]. Герлиц Е.А., Кулямин В.В., Максимов А.В., Петренко А.К., Хорошилов А.В., Цыварев А.В. *Тестирование операционных систем*. Труды ИСП РАН, 2014.
[34]. Н.В.Пакулин, В.З.Шнитман. *Автоматизация тестирования соответствия для телекоммуникационных протоколов*. Труды ИСП РАН, 2014.
[35]. А.С. Камкин, А.М. Коцыняк, С.А. Смолов, А.Д. Татарников, М.М. Чупилко. *Средства функциональной верификации микропроцессоров*. Труды ИСП РАН, 2014.

[36]. С.В.Зеленов, А.К.Петренко, Н.В.Пакулин, А.А.Угненко, А.А.Хорошилов. *Инструментальные средства проектирования систем интегрированной модульной авионики*. Труды ИСП РАН, 2014.
[37]. Н. В. Пакулин, А. В. Хорошилов. *Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов*. Программирование, 33 (6), 26-55 (2007).
[38]. The Linux Foundation consortium. LSB certification test suite, http://ispras.linuxbase.org/index.php/LSB_Certification_System
[39]. Chupilko, M. M. *Developing Test Systems of Multi-Modules Hardware Designs*. Programming and Computer Software, 2012, 38(1):34-42, 2012.
[40]. Zelenov, S.V., Zelenova, S.A. *Model-Based Testing of Optimizing Compilers*. In: Proc. of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems – 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007). LNCS, vol. 4581, pp. 365-377. Springer-Verlag, Berlin, 2007.
[41]. Zelenov, S.V., Silakov, D.V., Petrenko, A.K., Conrad, M., Fey I.: *Automatic Test Generation for Model-Based Code Generators*. In: IEEE ISoLA 2006 Second Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Paphos, Cyprus, pp. 68-75, 2006.
[42]. Камкин, А.С.: *Метод автоматизации имитационного тестирования микропроцессоров конвейерной архитектуры на основе формальных спецификаций.* Диссертация на степень к.ф.-м.н., Москва, 2009.
[43]. Корныхин, Е.В.: *Метод автоматизации генерации тестовых программ для верификации MMU*. Диссертация на степень к.ф.-м.н., Москва, 2010.
[44]. Kamkin, A.S., Tatarnikov, A.: *MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors*. In: Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), May 30-31, 2012, Perm, Russia, 2012.

Evolution of UniTESK Test Development Technology

V. Kuliamin, A. Petrenko
Institute for System Programming, Russain Academy of Sciences, Moscow, Russia
{kuliamin,petrenko}@ispras.ru

Abstract. The paper considers almost 20-year evolution of UniTESK test development technology in ISP RAS and its future perspectives. It presents the basic principles of UniTESK: using formal software contracts as a base of test adequacy criteria and test oracle construction, uniform test suite architecture helping to organize various kinds of testing in one framework, interleaving semantics used to specify asynchronous interactions, and using extended FSM models derived from contracts and coverage criteria to generate test sequences automatically. The paper then summarizes experience of using UniTESK in large test development projects for software and hardware systems, including telecommunication protocols, basic and standard interfaces of operating systems, microprocessor units. Several directions of future technology development are depicted, all intended for higher scalability of test suites constructed: parallelization of test execution for large test suites, using more efficient techniques of asynchronous behavior analysis, automated merging of different bugs representing the same faults, automatic extraction of compact and most substantial bug description (including short demonstration scenarios) from huge data obtained as a result of long and complex automated test execution.

Keywords: model based testing, test automation, formal specifications, software contracts, test scalability.

References

[1]. I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77-88, Springer-Verlag, 2002.

[2]. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov. The UniTesK Approach to Designing Test Suites. Programming and Computer Software, 29(6):310-322, 2003.

[3]. ISO/IEC TR 19759 Software Engineering — Guide to the Software Engineering Body of Knowledge (SWEBOK). Geneva, Switzerland: ISO, 2005.

[4]. L. Baresi, M. Young. Test Oracles. Tech. Report CIS-TR-01-02. 2001, http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

[5]. V. V. Kuliamin. Organizatsiya slozhnykh testovykh naborov [Organization of complex test suites]. Trudy ISP RAN [Proceedings of ISP RAS], 17:9-24, 2009 (in Russian).

[6]. B. Meyer. Applying Design by Contract. IEEE Computer,25(10): 40-51, October 1992.

[7]. I. Bourdonov, A. Kossatchev, A. Petrenko, D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. Proceedings of FM'99, Toulouse, France, LNCS 1708:608-621, Springer-Verlag, 1999.

[8]. I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, S. V. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'2001, Novosibirsk, Russia, LNCS 2244:301-307, Springer-Verlag, 2001.
[9]. G. V. Kluchnikov, A. S. Kossatchev, N. V. Pakulin, A. K. Petrenko, V. Z. Shnitman. Primenenie formal'nykh metodov dlya testirovaniya realizatsii IPv6 [Using formal methods for IPv6 implementation testing]. Trudy ISP RAN [Proceedings of ISP RAS], 4:121-140, 2003 (in Russian).
[10]. V. V. Kuliamin. Component architecture of model-based testing environment. Programming and Computer Software, 36(5):289-305, 2010.
[11]. V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin, A. S. Kossatchev, I. B. Bourdonov. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. Proceedings of PSI'2003, Novosibirsk, Russia, LNCS 2890:450-461, Springer-Verlag, 2003.
[12]. V. Kuliamin, A. Petrenko, N. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proceedings of ISAS'2005, Berlin, Germany, LNCS 3694:68-83, Springer-Verlag, 2005.
[13]. A. V. Khoroshilov. Spetsifikatsiya i testirovanie komponentov s asinkhronnym interfejsom [Specification and testing of components with asynchronous interface]. PhD Thesis, ISP RAS, Moscow, 2006 (in Russian).
[14]. I. B. Burdonov, A. S. Kossatchev, and V. V. Kulyamin. Application of Finite Automatons for Program Testing. Programming and Computer Software, 26(2):61-73, 2000.
[15]. I. B. Burdonov, A. S. Kossatchev, and V. V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, 29(5):245-258, 2003.
[16]. H. Zhu, P. A. V. Hall, J. H. R. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427, Dec. 1997.
[17]. A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov. Formal Methods in Industrial Software Standards Enforcement. Proceedings of PSI'2006, Novosibirsk, Russia, LNCS 4378:459-469, 2006.
[18]. OLVER project, http://linuxtesting.org
[19]. A. Maksimov. Requirements-Based Conformance Testing of ARINC 653 Real-Time Operating Systems. Proceedings of Data Systems In Aerospace (DASIA) 2010, ESA SP-682, ISBN 978-92-9221-246-9, 2010.
[20]. G. V. Kluchnikov, A. S. Kossatchev, N. V. Pakulin, A. K. Petrenko, V. Z. Shnitman. Primenenie formal'nykh metodov dlya testirovaniya Mobile IPv6 [Using formal methods for Mobile IPv6 conformance testing]. 2-nd International Conference "Next Generation Internet", pp. 20-25, Yaroslavl, Russia, 2003 (in Russian).
[21]. V. P. Ivannikov, A. S. Kamkin, A. S. Kossatchev, V. V. Kuliamin, and A. K. Petrenko. The use of contract specifications for representing requirements and for functional testing of hardware models. Programming and Computer Software, 33(5):272-282, 2007.
[22]. W. Grieskamp. Microsoft's Protocol Documentation Program: A Success Story for Model-Based Testing. Testing – Practice and Research Techniques. Lecture Notes in Computer Science, vol. 6303, p. 7, Springer, 2010.
[23]. P. Godefroid, N. Klarlund, K. Sen. DART: Directed Automated Random Testing. ACM SIGPLAN Notices — Proceedings of PLDI 2005, 40(6):213-223, 2005.

[24]. K. Sen, D. Marinov, G. Agha. CUTE: a concolic unit testing engine for C. Proceedings of ESES/FSE, pp. 263–272, 2005.
[25]. C. Cadar, V. Ganesh, P. Pawloski, D. Dill, D. Engler. EXE: Automatically Generating Inputs of Death. Proceedings of the 13-th International Conference on Computer and Communications Security CCS 2006, pp. 322-335.
[26]. C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. Feedback-Directed Random Test Generation. Proc. of International Conference on Software Engineering, pp. 75-84, 2007.
[27]. I. B. Burdonov, S. G. Groshev, A. V. Demakov, A. S. Kamkin, A. S. Kossatchev, A. A. Sortov. Parallel'noe testirovanie bol'shikh avtomatnykh modelej [Parallel testing of large FSM models]. Vestnik NNGU [Bulletin of NNSU], № 3, 2011, pp. 187-193 (in Russian).
[28]. I. B. Burdonov, A. S. Kossatchev. Obkhod neizvestnogo grafa kollektivom avtomatov [Exploration of unknown graph by a set of autonata]. Proc. of International Conference "Science Service in the Internet: all facets of parallelism ". 2013, pp. 228-232 (in Russian).
[29]. C. Flanagan, P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. ACM SIGPLAN Notices— Proceedings of POPL 2005, 40(1):110-121, 2005.
[30]. Y. Yang, X. Chen, G. Gopalakrishnan, R. Kirby. Efficient Stateful Dynamic Partial Order Reduction. Proceedings of SPIN 2008, LNCS 5156:288-305, Springer, 2008.
[31]. V. P. Ivannikov, A. K. Petrenko, V. V. Kuliamin, A. V. Maksimov. Opyt ispol'zovaniya UniTESK kak zerkalo razvitiya tekhnologij testirovaniya na osnove modelej [Experiences of UniTESK applications reflecting model based testing technology development]. Trudy ISP RAN [Proceedings of ISP RAS], 23:207-218, 2013 (in Russian).
[32]. I. B. Burdonov, A. S. Kossatchev. Razvitie teorii konformnosti: semantiki, formal'nye modeli, algoritmy [Development of conformance testing theory: models, semantics, algorithms]. Trudy ISP RAN [Proceedings of ISP RAS], 2014 (in Russian).
[33]. E. A. Gerlits, A. V. Maksimov, A. K. Petrenko, A. V. Khoroshilov. Testirovanie operatsionnykh sistem [Testing of operating systems]. Trudy ISP RAN [Proceedings of ISP RAS], 2014 (in Russian).
[34]. N. V. Pakulin, V. Z. Shnitman. Avtomatizatsiya testirovaniya sootvetstviya dlya telekommunikatsionnykh protokolov [Protocol conformance testing automation]. Trudy ISP RAN [Proceedings of ISP RAS], 2014 (in Russian).
[35]. A. S. Kamkin, A. M. Kotsyniak, S. A. Smolov, A. D. Tatarnikov, M. M. Chupilko. Sredstva funktsional'noj verifikatsii mikroprotsessorov [Microprocessor functional verification tools and methods]. Trudy ISP RAN [Proceedings of ISP RAS], 2014 (in Russian).
[36]. S. V. Zelenov, A. K. Petrenko, N. V. Pakulin, A. A. Ugnenko, A. V. Khoroshilov. Instrumental'nye sredstva proektirovaniya sistem integrirovannoj modul'noj avioniki [Integrated modular avionics system design tools]. Trudy ISP RAN [Proceedings of ISP RAS], 2014 (in Russian).
[37]. N. V. Pakulin, A. V. Khoroshilov. Development of Formal Models and Conformance Testing for Systems with Asynchronous Interfaces and Telecommunications Protocols. Programming and Computer Software, 33(6):316-335, 2007.
[38]. The Linux Foundation consortium. LSB certification test suite, http://ispras.linuxbase.org/index.php/LSB_Certification_System
[39]. M. M. Chupilko, Developing Test Systems of Multi-Modules Hardware Designs. Programming and Computer Software, 2012, 38(1):34-42, 2012.

[40]. S. V. Zelenov, S. A. Zelenova. Model-Based Testing of Optimizing Compilers. In: Proc. of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems – 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007). LNCS, vol. 4581, pp. 365-377. Springer-Verlag, Berlin, 2007.

[41]. S. V. Zelenov, D. V. Silakov, A. K. Petrenko, M. Conrad, I. Fey. Automatic Test Generation for Model-Based Code Generators. In: IEEE ISoLA 2006 Second Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Paphos, Cyprus, pp. 68-75, 2006.

[42]. A. S. Kamkin Metod avtomatizatsii imitatsionnogo testirovaniya mikroprotsessorov konvejernoj arkhitektury na osnove formal'nykh spetsifikatsij [Pipeline microprocessor simulation testing automation based on formal specifications]. PhD Thesis, ISP RAS, Moscow, 2009 (in Russian).

[43]. E. V. Kornykhin. Metod avtomatizatsii generatsii testovykh programm dlya verifikatsii MMU [Automated generation of etst programs for MMU verification]. PhD Thesis, ISP RAS, Moscow, 2010 (in Russian).

[44]. A. S. Kamkin, A. D. Tatarnikov. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. In: Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), May 30-31, 2012, Perm, Russia, 2012.

Развитие теории конформности: семантики, формальные модели, алгоритмы

Игорь Бурдонов <igor@ispras.ru>, Александр Косачев kos@ispras.ru

Аннотация. Статья посвящена теоретическим и практическим работам по тестированию конформности (conformance testing), которые выполнялись в ИСП РАН с 1994-го года и по настоящее время. Развитие теории конформности шло по нескольким направлением и в целом носило характер обобщения используемых семантик взаимодействия, моделей и конформностей. Необходимость такого обобщения диктовалась, прежде всего, требованиями практического тестирования. Это касается таких свойств систем как недетерминизм, частичная определённость, асинхронность, разнообразие тестовых воздействий и наблюдений над поведением реализации и т.п. При этом в центре внимания всегда находилась эффективность тестирования, определяемая как оптимизацией тестовых наборов, так и алгоритмами генерации тестов, в том числе on-fly. Мы рассматриваем основные вехи этого пути в кратком и неформальном обсуждении, уделяя внимание не деталям, а основным проблемам и способам их решения, пытаясь выявить общую тенденцию развития.

Ключевые слова: Семантика взаимодействия, конечные автоматы, LTS, IOLTS, трассы, конформность, приоритеты, симуляция, редукция, генерация тестов, оптимизация тестирования, пополнение спецификаций, композиция систем, медиаторы, моделирование, реализация, спецификация, безопасное тестирование, исследование графов, обход графа, параллельное тестирование.

1. Введение

Работы по тестированию конформности в ИСП РАН начались в 1994-ом году в рамках проекта по верификации ядра операционной системы реального времени для канадской телекоммуникационной компании Nortel Networks. Хотя цели этой работы были чисто практическими, ставилась задача разработки общей методологии и технологии тестирования конформности, пригодной для широкого класса приложений. Она получила название KVEST (Kernel VErification and Specification Technology) [[1]-[4]] и впоследствии легла в основу семейства методологий и технологий под общим названием UniTESK (Uniform Testing Kit) [[5]-[56]], развиваемого и широко используемого в настоящее время.

Тестирование конформности – это одно из основных направлений в области верификации систем на основе формальных моделей. Конформность – это отношение «соответствия» модели проверяемой системы (реализации) модели спецификационных требований к ней (спецификации). При тестировании конформность проверяется в процессе взаимодействия реализации с тестом, построенным по спецификации. Для этого спецификационные требования должны быть сформулированы в терминах такого взаимодействия, а семантика взаимодействия должна позволять доводить проверку до конца. Это значит, что тест должен заканчиваться за конечное время и выносить *вердикт*: ***pass*** или ***fail***. Набор тестов значимый, если он не ловит «ложных ошибок»: для конформной реализации все тесты выносят вердикт ***pass***. Набор тестов исчерпывающий, если он обнаруживает любую ошибку: для неконформной реализации хотя один тест вынесет вердикт ***fail***. Набор тестов полный, если он значимый и исчерпывающий.

Развитие теории конформности в ИСП РАН шло по нескольким направлением и в целом носило характер обобщения используемых семантик взаимодействия, моделей и конформностей. В то же время исследовались возможности оптимизации тестирования для суженных, но практически полезных, классов реализаций и спецификаций. В данной статье мы рассмотрим основные вехи пути этого развития в кратком и неформальном обсуждении.

2. Конечные автоматы

2.1. Всюду определённые детерминированные конечные автоматы

Мы начинали с модели конечного автомата[1], в котором каждый переход помечается двумя символами: стимулом (input) и реакцией (output). В 1996-ом году вышел обзор [[96]] принципов и методов тестирования таких автоматов. По этому обзору хорошо видно, что к тому времени хорошо разработанная теория тестирования была создана только для класса детерминированных и всюду определённых конечных автоматов. Детерминированность означает, что в каждом состоянии автомата имеется не более одного перехода по каждому стимулу[2]. Всюду определённость означает, что в каждом состоянии автомата имеется не менее одного перехода по каждому стимулу. Тем самым, в каждом состоянии автомата должен быть ровно один переход по каждому стимулу и какой-нибудь реакции. Если число состояний автомата обозначить через n, число переходов – через m, а число стимулов – через p, то для таких автоматов $m=np$. Тестирование заключается в подаче на автомат последовательности стимулов и наблюдения получаемых в ответ реакций.

[1] Такой автомат называется автоматом Мили.

[2] Мы называем такой детерминизм *сильным детерминизмом*.

Предполагается, что тестирование состоит из «одного сеанса» и занимает конечное время, т.е. на автомат подаётся только одна последовательность стимулов, имеющая конечную длину.

В [[96]] конформность опиралась на понятие эквивалентности состояний. Два состояния эквивалентны, если в этих состояниях любая последовательность стимулов порождает одну и ту же последовательность реакций. Иными словами, в эквивалентных состояниях определены одинаковые множества *трасс* как чередующихся последовательностей стимулов и реакций. Два автомата эквивалентны, если для каждого состояния одного автомата в другом автомате имеется эквивалентное ему состояние. Обычно имеются в виду только достижимые состояния, т.е. состояния, которых можно достигнуть по цепочке переходов из начального состояния.

На практике обычно рассматриваются автоматы с начальными состояниями, причём под начальным состоянием реализации понимается то состояние, с которого начинается её тестирование. Конформность таких автоматов подразумевает эквивалентность их начальных состояний. Для детерминированных и всюду определённых автоматов такая конформность совпадает с трассовой эквивалентностью и (слабой) бисимуляцией. Трассовая эквивалентность определяется как равенство множеств трасс, определённых в начальных состояниях автоматов. Слабая бисимуляция определяется как существование симметричного соответствия состояний, при котором начальные состояния соответствуют друг другу, и любая трасса, начинающаяся в состоянии a, начинается также в соответствующем ему состоянии $a`$, и, если состояние $b$ достижимо из $a$ по этой трассе, то хотя бы одно состояние $b`$, соответствующее b, достижимо из $a`$ по этой трассе. Трассовая эквивалентность и бисимуляция являются отношениями эквивалентности, порождаемыми соответствующими отношениями предпорядка: трассовым предпорядком (вложенность трасс) и симуляцией. Для всюду определённых детерминированных автоматов эти предпорядки совпадают с порождаемыми ими отношениями эквивалентности, но это не так в общем случае недетерминированных и частично (т.е. не всюду) определённых автоматов. Также в общем случае не рассматривают эквивалентность автоматов, а трассовая эквивалентность (предпорядок) и бисимуляция (симуляция) различаются.

Проверка эквивалентности автоматов методом тестирования в общем случае невозможна: всегда найдётся такой реализационный автомат, который не эквивалентен спецификации, но внешне ведёт себя так же, как спецификация, т.е. выдаёт те же самые реакции в ответ на те же самые стимулы. Этот «тупик» можно обойти несколькими способами: 1) наложить ограничения на класс тестируемых реализаций, 2) наложить ограничения на спецификации, 3) использовать дополнительные тестовые возможности, а также сочетанием этих способов. Прежде всего, поскольку на автомат подаётся только одна последовательность стимулов и мы должны проверить каждый стимул в

каждом состоянии, граф переходов спецификационного автомата должен быть сильно-связным.

Далее можно выделить два принципиально различных направления: тестирование с закрытым состоянием и тестирование с открытым состоянием.

При тестировании с закрытым состоянием реализация понимается как «чёрный ящик»: её состояния мы не видим. Если никаких дополнительных тестовых возможностей нет (мы можем только посылать в реализацию стимулы и наблюдать ответные реакции), то, прежде всего, налагаются ограничения на «размер» реализации. Спецификация «минимизируется», т.е. специальным алгоритмом преобразуется в эквивалентную ей спецификацию, все состояния которой попарно неэквивалентны. Тестируются только те реализации, число состояний которых не превосходит (или превосходит не более, чем на константу) числа состояний спецификации. В этом случае всегда существует алгоритм тестирования конформности, который, однако, может иметь экспоненциальную (относительно n) сложность. Для того, чтобы снизить сложность до полиномиальной (mn^2), приходится налагать ограничения на спецификацию: в ней должны существовать, так называемые, *различающие* (*distinguishing*) последовательности.

При тестировании с закрытым состоянием рассматривалась также такая тестовая возможность как *рестарт*, который гарантированно переводит реализацию из любого текущего состояния в начальное состояние. Прежде всего, при наличии рестарта граф переходов спецификации можно считать сильно-связным, если добавить в каждом состоянии переход по рестарту в начальное состояние. При наличии рестарта существует алгоритм тестирования конформности с полиномиальной сложностью (тоже mn^2).

Тестирование с открытым состоянием опирается на специальную тестовую возможность: операцию *status message*, которая позволяет узнать текущее состояние реализации без изменения его. Это самый «хороший» случай: тестирование конформности, фактически, сводится к обходу графа переходов реализационного автомата, который имеет длину $\boldsymbol{O}(mn)$, где m и n – это число переходов и состояний в реализации. Требуется сильно-связность этого графа или наличие рестарта. Заметим, что при тестировании с открытым состоянием не требуется минимизировать спецификацию. При тестировании устанавливается *естественное* соответствие состояний реализации и спецификации: проходя некоторую трассу, мы считаем соответствующими состояния реализации и спецификации после этой трассы. Для верификации конформности проверяется, что в соответствующих состояниях по стимулу получаются одинаковые реакции и соответствующие постсостояния (состояния после переходов по этим стимулу и реакции).

Развитие теории тестирования в мире поначалу шло, в основном, в направлении тестирования с закрытым состоянием, где были разработаны многочисленные алгоритмы, опирающиеся на различные последовательности стимулов в спецификации (установочные, различающие, синхронизирующие,

разделяющие, характеристические и др.), а также на те или иные гипотезы о реализации. Работы в этом направлении продолжаются и сегодня.

Для нас, однако, эти достижения оказались практически бесполезными, поскольку мы с самого начала пошли в другом направлении, положив в основу алгоритма тестирования обход графа [[58]]. При тестировании с открытым состоянием это был, как сказано выше, граф переходов реализации, а при тестировании с закрытым состоянием граф переходов спецификации. В последнем случае нам пришлось принимать следующую (довольно сильную) *гипотезу о реализационных состояниях*: любая ошибка – это выдача ошибочной реакции. Точнее, гипотеза говорит, что естественное соответствие состояний реализации и спецификации взаимно-однозначно. Если такая гипотеза выполнена, то, перебирая все стимулы в каждом состоянии спецификации, мы либо обнаружим ошибку (ошибочную реакцию), либо переберём все стимулы в каждом состоянии реализации. Эту гипотезу можно ослабить, разрешив нескольким спецификационным состояниям соответствовать одному состоянию реализации. В этом случае реализация может быть конформна только, если все состояния спецификации, соответствующие одному состоянию реализации, эквивалентны друг другу. Также гипотезу можно ослабить, разрешив нескольким состояниям реализации соответствовать одному состоянию спецификации. Но тогда может оказаться, что разные стимулы, подаваемые в одном состоянии спецификации, в реализацию поступают в разных её состояниях. Поэтому для полноты тестирования дополнительно нужно предположить, что в состояниях реализации, соответствующих одному состоянию спецификации, имеется одно и то же множество ошибок.

2.2. Частично определённые и недетерминированные конечные автоматы

Направление, в котором развивались наши исследования, определялось практическими потребностями. Мы начинали с детерминированных и всюду определённых автоматов, а потом постепенно снимали эти ограничения. Дело в том, что нам сразу нужно было тестировать системы, для которых в качестве адекватной модели приходилось выбирать недетерминированные и/или частично определённые конечные автоматы.

Частичная определённость и недетерминизм автомата имеют несколько различную прагматику для реализации и спецификации. Если стимул отсутствует в спецификации, то это значит, что не предъявляется никаких требований к поведению реализации при подаче этого стимула в соответствующем состоянии, т.е. стимул можно не подавать. Если же стимул отсутствует в состоянии реализации, то мы трактовали это как запрет на подачу стимула. Другую трактовку («блокировка стимула») мы стали рассматривать позже уже не для конечных автоматов. Недетерминизм реализации является, так сказать, её «природным» свойством: получая стимул,

реализация действительно может выдавать в данном состоянии то одну, то другую реакцию и переходить то в одно, то в другое постсостояние недетерминированным образом. Недетерминизм же спецификации является следствием неоднозначности спецификационных требований, которые не столько предписывают реализации конкретное поведение, сколько налагают на такое поведение ограничения, тем самым, оставляя на выбор реализации, какую реакцию на стимул она будет выдавать.

Это привело к тому, что саму конформность пришлось понимать не как эквивалентность автоматов.Мы понимаем конформность как состоящую из двух частей: гипотезы о реализации и проверяемого условия. Гипотеза предъявляет к реализации требования, которые позволяют её тестировать, но сами эти требования при тестировании не проверяются и являются предусловием тестирования. Проверяемое условие – это как раз те требования к реализации, которые проверяются при тестировании.

В основном мы рассматриваем трассовые конформности, которые основаны на трассах реализации и спецификации[3]. Трассовую конформность мы называем *редукцией*, если она основана на разделении всех возможных трасс на правильные (конформные) трассы и ошибочные (неконформные) трассы (ошибки). Спецификация как раз и задаёт такое разделение. Реализация конформна, если в ней нет ошибок, т.е. множество трасс реализации вложено во множество конформных трасс, определяемое по спецификации. Тем самым, редукция является частичным (нестрогим) порядком (рефлексивное, симметричное и транзитивное отношение). Не являются редукциями трассовые конформности, которые требуют обязательного наличия в реализации тех или иных наблюдений после трасс, или которые разрешают, например, иметь в реализации любую из двух трасс, но не обе сразу.

Важно отметить, что множество конформных трасс (или его дополнение – множество ошибок) обычно задаётся спецификацией неявно: множество всех конформных трасс вычисляется по множеству трасс спецификации (трасс, определяемых ею явно), но вовсе не обязательно совпадает с ним. Поэтому редукция обычно определяется как отношение множества трасс реализации не с множеством всех конформных трасс, а с множеством трасс спецификации, т.е. трасс, явно задаваемых спецификацией, и в этом случае она уже не обязательно обладает свойствами рефлексивности, симметричности и транзитивности.

Мы использовали контрактные спецификации, основанные на пре- и постусловиях [[5],[6]]. Предусловие абстрагируется в автоматной модели в наличие или отсутствие в том или ином состоянии перехода по тому или иному стимулу, иначе говоря, сразу требует частично определённых автоматов. Постусловие абстрагируется в предикат, которому должны удовлетворять реакция и постсостояние для данных стимула и пресостояния.

[3] Конформность, основанная на соответствии состояний реализации и спецификации, рассматривается ниже в разделе 5.3.

Тем самым, пресостояние и стимул, вообще говоря, неоднозначно определяют реакцию и постсостояние, что и означает недетерминизм. Кроме того, контрактная спецификация задаёт спецификационный автомат имплицитно, и он эксплицируется в процессе тестирования. Это означает, что граф спецификации заранее неизвестен.

Частично определённые детерминированные автоматы

Сначала мы исследовали частично определённые, по, по-прежнему, детерминированные автоматы. Поскольку стимул может быть определён не в каждом состоянии реализации, мы вынуждены были использовать специальную гипотезу о реализации. Она называлась *гипотезой о допустимости* (стимулов) [[58],[60]], а в дальнейшем такого рода гипотезы мы стали называть гипотезами о безопасности [[63]-[66],[70]]. Гипотеза основана на естественном соответствии состояний (состояний после общих трасс) реализации и спецификации. Стимул называется допустимым (впоследствии они стали называться безопасными) после трассы, если он определён в состоянии спецификации после трассы. Гипотеза о допустимости предполагает, что такой стимул должен быть определён в соответствующем состоянии реализации. Заметим, что в реализации в состоянии могут быть определены также недопустимые стимулы, т.е. те, которые не определены ни в каком соответствующем состоянии спецификации. Недопустимые стимулы при тестировании конформности не подаются. Допустимые стимулы определяют допустимые трассы (впоследствии они стали называться безопасными), т.е. трассы, содержащие только допустимые стимулы. В рассматриваемом случае это просто все трассы спецификации.

Частичная определённость автомата вынудила нас вместо трассовой эквивалентности автоматов использовать конформность типа редукции: во-первых, в реализации после имеющейся в ней допустимой трассы должны быть определены все допустимые после этой трассы стимулы (гипотеза о допустимости), и, во-вторых, на каждый такой стимул реализация должна выдавать такую же реакцию, как в спецификации (проверяемое условие). Для детерминированных автоматов такая конформность эквивалентна вложенности множества трасс спецификации во множество трасс реализации. Для частично определённых детерминированных автоматов тестирование по-прежнему основано на обходе графа, но подаются не любые, а только допустимые стимулы. При тестировании с открытым состоянием строится не граф реализации, а его подграф, порождаемый допустимыми трассами.

Частичная определённость впоследствии породила две идеи: идею о безопасности тестирования и идею об отказах. Это определялось выбором той или иной интерпретации отсутствия перехода по стимулу в состоянии реализации.

В одной интерпретации отсутствие в состоянии перехода по стимулу понимается как запрет на подачу стимула в состоянии. Иначе говоря, раз переход по стимулу в состоянии не определён, значит не определено поведение автомата после подачи стимула в состоянии. Обычно это понималось так, что подавать стимул нет смысла: нечего проверять, поскольку любое поведение возможно (*хаотическое* поведение [[101][102]]). Мы предложили расширенное понимание хаотического поведения, включив в него «опасное» поведение, при котором автомат может «сломаться», «разрушиться». Тем самым, стимул, не определённый в состоянии, не только нет смысла, но и нельзя подавать в состоянии. Так возникло понятие разрушения [[63]-[66]], впоследствии оно изображалось переходом по специальному символу γ. Тестирование безопасно, если не возникает разрушения, т.е. мы не оказываемся в состоянии, где есть γ-переход. Именно эта интерпретация используется в описанной выше гипотезе о допустимости: если стимул не определён в состоянии реализации, это понимается как наличие перехода по стимулу в состояние, где есть γ-переход.

В другой интерпретации отсутствие в состоянии перехода по стимулу понималось как вполне определённая ситуация «блокировки стимула» [[63]-[66]]. Если эту ситуацию нельзя распознавать при тестировании, то мы можем бесконечно долго ждать реакции, которой не будет. Поскольку мы хотим, чтобы тестирование заканчивалось за конечное время, такой стимул естественно считать тоже «опасным». Однако, если блокировка стимула наблюдаема, то это новый вид наблюдения, который порождает новую конформность. Возможность такого распознавания определяется интерфейсом между тестовой и тестируемой системой. Например, как это бывает в протоколах, подаваемый стимул – это сообщение, посылаемое в тестируемую систему, и предполагается получение ответного сообщения «стимул принят». Если такое ответное сообщение должно придти в течение заданного тайм-аута, то отсутствие ответа в течении тайм-аута как раз и означает блокировку стимула. Блокировку стимула следует отличать от игнорирования стимула, которая иногда также используется как интерпретация отсутствия перехода по стимулу. Например, в автомате, выдающем кофе, стимул игнорируется, если автомат «глотает» монету, но кофе не выдаёт, и стимул блокируется, если щель для приёма монет перекрыта. Наблюдаемая блокировка стимула не используется для конечных автоматов, мы использовали её впоследствии при переходе от автоматной модели к модели LTS, о чём речь пойдёт ниже.

Частично определённые детерминированные реализации и слабо-детерминированные спецификации

Далее мы исследовали общий случай частично определённых и недетерминированных автоматов. Поначалу мы ограничивались только детерминированными реализациями. Почему же для детерминированных реализаций спецификация может быть недетерминированной? Причина в том,

что спецификация описывает не конкретную реализацию, а требования к реализации, т.е., фактически, класс возможных (конформных) реализаций. В контрактной форме задания спецификации постусловие, как сказано выше, определяет не единственную «правильную» реакцию (и постсостояние) при заданных пресостоянии и стимуле, а множество таких «правильных» реакций (и постсостояний). В графе переходов это изображается, так называемой, Δ-переходом – множеством переходов, выходящих из одного состояния и помеченных одним стимулом [[60]].

Поскольку тестирование происходит по трассам спецификации, нам нужно в спецификации вычислять постсостояние перехода по пресостоянию, стимулу и реакции, полученной от реализации. Спецификация, в которой такое вычисление даёт однозначный результат, называется *слабо-детерминированной*[4]. Для конечных автоматов мы ограничились только такими спецификациями, чтобы не переусложнять тестовую систему.

Недетерминизм спецификации (даже слабый) вызывает изменение конформности. От реализации требуется, чтобы она выдавала на данный стимул только «правильные» реакции, но не обязательно все. Теперь уже конформность не эквивалентна вложенности множества трасс спецификации во множество трасс трасс реализации.

Сравним тестирование детерминированной реализации для сильно- и слабо-детерминированной спецификации. Гипотезы о реализации одинаковы в обоих случаях и различаются для тестирования с открытым и с закрытым состоянием.

Тестирование с открытым состоянием, по-прежнему, опирается на гипотезы о допустимости и сильно-связности (с учётом рестарта) реализации и основано на обходе подграфа реализации, порождаемого допустимыми трассами. От случая детерминированной спецификации оно отличается только тем, что изменяется проверяемое условие, поскольку меняется конформность.

Тестирование с закрытым состоянием, по-прежнему, опирается на гипотезу о допустимости и гипотезу о реализационных состояниях реализации. Мы, по-прежнему, в каждом состоянии спецификации должны попробовать каждый определённый в нём стимул. Но теперь это отличается от обхода графа, поскольку таким образом мы пройдём хотя бы по одному, а не по каждому переходу в каждом Δ-переходе. Такой обход мы назвали *обходом по стимулам*. После подачи стимула в состоянии мы получаем от реализации какую-то реакции. Если она ошибочная, тестирование заканчивается. Если реакция «правильная», мы проходим в спецификации соответствующий ей

[4] В [[99]] это называется наблюдаемым детерминизмом, но мы предпочитаем использовать этот термин для случая, когда пресостояние и стимул однозначно определяют реакцию и, быть может, неоднозначно постсостояние, т.е. когда получаемая последовательность реакций однозначно определяется последовательностью подаваемых стимулов.

переход из Δ-перехода. Чтобы гарантировать тестирование любой (удовлетворяющей принятым гипотезам) реализации, вводится понятие Δ-*маршрута*, который определяется как множество маршрутов, «ветвящееся» в каждом проходимом Δ-переходе, определяемом состоянием и стимулом: во множестве есть маршруты, соответствующие всем имеющимся в спецификации (и, тем самым, возможным в конформных реализациях) реакциям для этих состояния и стимула. Если тестирование полное, то все маршруты Δ-маршрута – это обходы по стимулам; такой Δ-маршрут называется Δ-*обходом* [[60]]. Генерация теста реализует алгоритм построения Δ-обхода.

Δ-обход существует не для всякого графа. Для слабо-детерминированной спецификации достаточным условием существования Δ-обхода является сильно-Δ-связность графа спецификации: для любых двух состояний существует Δ-маршрут, все маршруты которого начинаются в первом состоянии и заканчиваются во втором состоянии. В [[60]] предложен оригинальный алгоритм Δ-обхода, время работы которого равно $\boldsymbol{O}(mn^2)$ или $\boldsymbol{O}(mn\log_2 n)$ в зависимости от того, можно ли сравнивать идентификаторы состояний только на равенство или также на больше/меньше. Требуемая память равна $\boldsymbol{O}(mX+nI+m\log_2 m)$ бит, где X и I – размер в битах, соответственно, стимула и идентификатора состояния. В [[19]] предложен другой алгоритм Δ-обхода для подкласса графов, в которых синглетонные Δ-переходы порождают сильно-связный суграф.

Частично определённые недетерминированные реализации и слабо-детерминированные спецификации

В дальнейшем мы стали рассматривать также тестирование недетерминированных реализаций с той же конформностью. Тестирование, по-прежнему, основано на обходе графа, но нам требуется дополнительная гипотеза о реализации, которую назовём Δ-*гипотезой*. Поскольку реализация недетерминирована, она может выдавать разные реакции на один и тот же стимул в одном и том же состоянии. Для гарантии обнаружения ошибки мы предположим, что, если на данный стимул в данном состоянии может быть ошибочная реакция, то мы её получим сразу при первой подаче этого стимула в этом состоянии.

При тестировании с открытым состоянием, по-прежнему, нужно подать в каждом состоянии реализации каждый допустимый в нём стимул. Для этого мы обходим по стимулам подграф реализации, порождённый допустимыми трассами, который должен быть сильно-Δ-связным.

При тестировании с закрытым состоянием, по-прежнему, нужно совершить Δ-обход спецификации. Спецификация должна быть сильно-Δ-связна.

Факторизация спецификации

Другой подход к проблеме недетерминизма спецификации заключается в том, чтобы этот недетерминизм «устранить», тем самым сведя задачу тестирования к случаю детерминированной спецификации. Такое «устранение» возможно с помощью факторизации спецификации, основанной на классах эквивалентности состояний, переходов и/или стимулов [[57]]. Фактор-спецификация может оказаться слабо- или даже сильно-детерминированной, тогда как исходная спецификация была недетерминированной (даже не слабо-детерминированной).

Кроме того, факторизация является основным методом решения проблемы слишком «больших» графов спецификации. Например, подача стимула часто понимается как вызов той или иной процедуры с параметрами. Среди таких параметров могут быть числа (целые и вещественные), и понятно, что полный перебор всех значений таких параметров практически невозможен. Разумеется, чтобы такой подход был оправданным, нужна мотивированная *фактор-гипотеза* о том, что классы эквивалентности подобраны «правильно», т.е. все интересующие нас ошибки, которые возможны в реализации, обнаруживаются при тестировании по факторизованной спецификации. Факторизация используется на одном из этапов методологии UniTESK, а именно на этапе разработки тестовых сценариев, являющихся представлениями конечных спецификационных автоматов [[6]].

3. Модели ввода-вывода

3.1. Асинхронные автоматы

На каком-то этапе наших практических работ мы столкнулись с тестированием систем, которые не могли адекватно моделироваться конечными автоматами. В качестве первого такого примера была многопроцессная программная система, интерфейс с которой основан на обмене сообщениями. В ответ на входное сообщение система могла выдать несколько выходных сообщений, причем, если в процессе их выдачи поступало следующее входное сообщение, оно могло изменить выходной поток.

Для таких систем мы сначала придумали автоматы с отложенными реакциями (AOP), которые похожи, но не совпадают, с хорошо известными автоматами ввода-вывода (IOSM – Input/Output State Machines), называемыми также взаимодействующими конечными автоматами (CFSM - Communicating Finite State Machines). В обоих автоматах переход из одного состояния в другой происходит либо как прием стимула – принимающий переход, либо как выдача реакции – посылающий переход, либо как пустой переход, не сопровождающийся ни приемом стимула, ни выдачей реакции. Состояния, в

которых определены только принимающие переходы называются стационарными.

Дальнейшим обобщением АОР и IOSM стал *асинхронный автомат* (*АА*) [[59]]. Название «асинхронный» связано с тем, что выдача реакции происходит, вообще говоря, асинхронно с приемом стимула. Отметим, что наше понятие АА отличается от понятия асинхронно выполняющейся сети взаимодействующих автоматов.

На самом деле АА – это семейство автоматов, систематизируемое по нескольким критериям: *смешанные состояния*, *е-переходы*, *императивность* или *факультативность*, *финальная допустимость стимулов* и *приоритеты*. В смешанном состоянии имеются как принимающие, так и посылающие или пустые переходы. Е-переход – это введённый нами переход по пустому стимулу, что понимается как переход по отсутствию стимула на входе автомата. В императивном автомате (в частности, в АОР) принимающий переход по поступившему на автомат стимулу приоритетнее посылающих и пустых переходов, а в факультативном автомате (в частности, в IOSM) они равноприоритетны. Если стимул не может быть принят, то это понимается как, так называемая, ошибка неспецифицированного ввода. Императивный автомат не может принять стимул, если стимул поступает тогда, когда автомат находится в состоянии, где нет перехода по этому стимулу. Факультативный автомат не обязан сразу принимать поступивший стимул (даже если он может это сделать): автомат может совершать посылающие и пустые переходы, а стимул остаётся ждать на входе автомата. Далее возможны три варианта: 1) стимул принимается, 2) автомат переходит в стационарное состояние, где нет перехода по данному стимулу, что приводит к ошибке неспецифицированного вида, 3) автомат бесконечно совершает посылающие и пустые переходы («зацикливается»). Если вариант 2 невозможен, то стимул считается финально допустимым. Соответственно, факультативные автоматы подразделяются на те, в которых обычная и финальная допустимости стимулов совпадают, и те, в которых это не так. Кроме того, для всех типов автоматов могут быть различные варианты взаимных приоритетов е-переходов, с одной стороны, и посылающих и пустых переходов, с другой стороны.

По указанным критериям определяются 18 типов АА. Эти типы автоматов сравнивались по реализуемым ими словарным функциям: если множество словарных функций, реализуемых автоматами одного типа, вложено во множество словарных функций, реализуемых автоматами второго типа, то считалось, что второй тип моделирует первый тип. Было показано, что все АА разбиваются на три группы эквивалентных (моделирующих друг друга и, тем самым, реализующих одно и то же множество словарных функций) автоматов. В частности, АОР относится к группе, которая моделирует все АА, в то время как IOSM реализует более узкий класс словарных функций. Кроме словарных функций АА изучались, так называемые, сериализации – последовательности, в которых подпоследовательности стимулов/реакций совпадают с

входным/выходным словом. Такие сериализации впоследствии назывались трассами, как это и принято в мировой литературе. Было показано, что классификация АА по словарным функциям совпадает с классификацией АА по их трассам. Также изучались конечные трассы АА и тестирование АА по конечным трассам.

Для АА рассматривалось стационарное и нестационарное тестирование. При стационарном тестировании стимулы подаются на автомат только в стационарных состояниях; иначе говоря, после того как на автомат подаётся стимул, ожидаются все поступающие в ответ реакции, и только после этого подаётся следующий стимул. Стационарное тестирование АА использовалось ещё в KVEST, а в UniTESK ему посвящены работы [[7],[16],[17],[19],[23]]. При нестационарном тестировании стимулы также подаются в стационарных состояниях, но можно подать не один стимул, а последовательность стимулов. Тем самым, только первый стимул принимается в стационарном состоянии, а последующие стимулы могут приниматься и в нестационарных состояниях.

Основная особенность тестирования АА заключалась в том, что мы не управляли поступлением реакций от реализации: получая поток реакций и подавая стимул, мы не знали, в какой точке этого потока реализация получит стимул. По сути, мы подразумевали, что реализация снабжена двумя очередями: во входную очередь поступают стимулы, а в выходную очередь – реакции. E-переход срабатывал, если входная очередь оказывалась пустой. В качестве наблюдения мы получали входное слово и выходное слово, по которым строили все возможные сериализации, т.е. трассы. Следует отметить, что после подачи стимула (или последовательности стимулов при нестационарном тестировании) мы ожидали реакций до тех пор, пока реализация не перейдёт в стационарное состояние. По сути, это означает, что в трассы включались наблюдения такой стационарности (сравни с δ-наблюдением в следующем п. 3.2). Проверялось, что хотя бы одна такая трасса имеется в спецификации, такая конформность относится к типу редукции.

Также рассматривались АА с несколькими входными и выходными очередями, когда можно параллельно послать несколько стимулов в разные (не обязательно все) входные очереди и принимать реакции из нескольких (не обязательно всех) выходных очередей. По сути идентификация стимула (реакции) уточняется номером входной (выходной) очереди так, что очереди осуществляют разбиение алфавита уточнённых стимулов (реакций). Поскольку стимулы, по крайней мере, в разные очереди можно было подавать параллельно, фактически, выполнялось нестационарное тестирование. Проблемы сериализации для АА с несколькими входными и/или выходными очередями изучались в работах [[20],[22]]. Такого рода проблемы исследовались и в работе [[105]].

3.2. ioco и $ioco_{\beta\gamma\delta}$

С современной точки зрения тестирование АА – это, так называемое, *асинхронное* тестирование или тестирование *в контексте*, роль которого играли входные и выходные очереди. Наблюдение стационарности – это то же самое, что, так называемое, δ-наблюдение или *quiescence*, которое ввёл в 1991 г. F.Vaandrager [[93]], а в 1996 г. J.Tretmans [[97]] использовал в предложенной им конформности ***ioco*** (*Input Output Conformance*). Эта конформность (тоже типа редукции) та же самая, что при тестировании IOSM (факультативный АА без e-переходов), но предполагает *синхронное* тестирование, когда никакого контекста нет и реализация находится под полным управлением теста. Именно поэтому удаётся получить при тестировании не пару входное/выходное слово, а сразу трассу наблюдений как последовательность стимулов, реакций и δ-наблюдений. Аналогично, IOSM с несколькими входными и выходными очередями соответствуют конформности ***mioco***, которую предложил ученик Tretmans`a L.Heerink [[98]].

Ещё одно отличие тестирования АА от ***ioco*** – это предположения о допустимости стимулов в реализации. Мы применили к АА тот же подход, что и к конечным автоматам. Частично определённые реализации разрешались, но стимулы, не определённые в реализации, считались «опасными», что впоследствии породило понятие разрушения. Гипотеза о допустимости предполагала, что после общей трассы стимул, допустимый в спецификации, допустим и в реализации. В то же время Tretmans требовал от реализаций всюду определённости, но допускал частично определённые спецификации.

Интересно, что четырьмя годами ранее, в 1992 г., в своей докторской диссертации Tretmans рассматривал частично определённые реализации, в которых отсутствие стимула трактовалось как его блокировка, правда, только при тестировании в контексте (входная очередь стимулов), но потом не возвращался к этой идее. Тем не менее, в мировой литературе стали появляться работы, в которых блокировка стимулов допускалась и рассматривалась как особый вид наблюдения.

Всё это подвигло нас на исследование систем, в которых возможно как разрушение, так и блокировка стимулов. Примером тестирования в ИСП РАН систем с блокировкой стимулов может служить работа [[10]]. Мы предложили конформность типа редукции, которую назвали $\boldsymbol{ioco}_{\beta\gamma\delta}$ [[63]-[66]]. От ***ioco*** она отличалась: 1) трактовкой отсутствия перехода по стимулу в стабильном состоянии, т.е. состоянии, где нет τ-переходов (аналог пустых переходов в АА), как блокировки стимула (блокировка может входить в трассу наблюдений), 2) наличием γ-переходов, т.е. переходов по разрушению. Кроме того, Tretmans рассматривал только системы, в которых нет бесконечной последовательности (например, цикла) τ-переходов, которая называется *дивергенцией*. Мы же допускали дивергенцию, которую в то время трактовали как разрушение. Вместо всюду определённости реализации и отсутствия

дивергенции мы предложили понятия безопасного тестирования, безопасной трассы и соответствующую гипотезу о безопасности. Тестирование безопасно, если не возникают разрушение и дивергенция; гипотеза о безопасности ограничивает класс реализаций, которые могут безопасно тестироваться для данной спецификации. Отношение ***ioco*** оказывается частным случаем $\boldsymbol{ioco}_{\beta\gamma\delta}$ – для спецификаций без блокировок стимулов, разрушения и дивергенции.

Кроме введения блокировок стимулов, мы преследовали цель решить две важнейшие проблемы теории конформности: 1) нерефлексивность и нетранзитивность конформности, и 2) немонотонность конформности.

Нерефлексивность отношения ***ioco*** с практической точки зрения очень неудобна: казалось бы, если реализация написана как «калька» со спецификации, то всё должно быть правильно, но для частично определённой спецификации это не так. Кроме того, рефлексивность и транзитивность конформности необходимы для последовательного повышения уровня абстракции спецификации. В частности, такое повышение происходит, когда при тестировании в технологии UniTESK вместо спецификации используется более «грубая», так называемая, тестовая модель. Рефлексивность гарантирует, что домен спецификаций вложен в домен реализаций, что даёт возможность построить такую тестовую модель, которой конформна исходная спецификация. А транзитивность гарантирует, что при тестировании по тестовой модели не будет «ложных» ошибок с точки зрения спецификации: реализация конформная спецификации конформна тестовой модели.

Причина нерефлексивности и нетранзитивности ***ioco*** в различии допустимости стимулов в реализации и спецификации: спецификация может быть частично определенной, а реализация должна быть всюду определённой. Поэтому возникла «естественная» задача о *пополнении* спецификации – её преобразовании в эквивалентную ей всюду определенную спецификацию. Кроме того, решение проблемы немонотонности, о которой скажем ниже, опирается на пополнение спецификации.

До нас обе указанные проблемы не были решены для ***ioco***. Предлагались различные пополнения, однако все они не сохраняют отношение ***ioco***. Нам удалось решить проблему пополнения, выйдя «за пределы» ***ioco***, т.е. перейдя к более общей конформности $\boldsymbol{ioco}_{\beta\gamma\delta}$. Мы предложили пополнение спецификации [[66]], которое сохраняет отношение ***ioco***. Более того, мы расширили реализационный домен ***ioco***, допустив реализации с блокировками, разрушением и дивергенцией, если для них выполнена гипотеза о безопасности. В дальнейшем мы оптимизировали алгоритмы такого пополнения специально для ***ioco*** [[80]].

Пополнение спецификации – это первый шаг к решению проблемы немонотонности конформности. Суть этой проблемы в том, что композиция реализаций, конформных своим спецификациям, оказывается, вообще говоря, не конформна композиции этих спецификаций. Эта проблема имеет самые разные «обличья». В асинхронном тестировании она известна как

несохранение конформности: при тестировании в контексте обнаруживаются «ложные» ошибки, не обнаруживаемые при синхронном тестировании. В общем случае без решения этой проблемы невозможно «вычислить» спецификацию системы, если известны спецификации её компонентов и «схема компоновки». Соответственно, невозможна верификация декомпозиции системных требований, т.е. проверка того, что спецификация системы правильно «разложена» на спецификации её компонентов.

Мы предложили алгоритм *монотонного* преобразования пополненных спецификаций [[66]], после которого композиция реализаций, конформных своим спецификациям, конформна композиции монотонно преобразованных пополненных спецификаций компонентов. Это преобразование работает для общей конформности $\boldsymbol{ioco}_{\beta\gamma\delta}$, а для частного случая ***ioco*** (пополненные спецификации без блокировок и разрушения) предложен упрощённый алгоритм преобразования [[66]], усовершенствованный в [[80]].

4. R/Q-модель

Наши работы по ***R***/***Q***-модели см. в [[64],[67],[69]-[74],[76]-[78],[83],[85]-[86]].

4.1. LTS

Модели, лежащие в основе как AA, так и отношений ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$, относятся к классу LTS (*Labelled Transition System*). В LTS переход между состояниями помечается *действием* из алфавита действий или символом τ (в AA вместо τ--перехода пустой переход). Для AA без e-переходов и конформностей ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$ алфавит действий LTS разбивается на два подалфавита: стимулов и реакций; такая LTS называется IOLTS (Input-Output LTS). Наблюдения стационарности (δ-наблюдение) и блокировок стимулов относятся к, так называемым, отказам. Отказ возникает при deadlock`e, когда реализация не может выполнять те действия, которые ей разрешены. Такой deadlock возможен только в стабильном состоянии, поскольку τ-переходы считаются всегда разрешёнными – эти переходы как бы «самопроизвольно» меняют состояние реализации, независимо от внешнего вмешательства. Отказ – это то множество «отвергаемых» действий (*refusal set*), которые разрешены, но не могут выполняться, что и вызывает deadlock. Например, если тест при синхронном тестировании посылает стимул, а в текущем стабильном состоянии реализации нет приёма этого стимула, возникает его блокировка Если тест не посылает стимулы, но принимает все реакции, а реализация не выдаёт реакции, то возникает deadlock в стационарном состоянии.

В дальнейшем мы сосредоточились на исследовании LTS общего вида. Для LTS взаимодействие формализуется в операции параллельной композиции двух LTS. Она имеет две основные модификации: CSP и CCS, которые по-разному понимают отношение *синхронности* на алфавитах операндов. В CSP синхронные действия одинаковые, а в CCS – противоположные (на

универсуме действий задано инволюционное соответствие). Состояние композиционной LTS – это пара состояний LTS-операндов. Паре переходов в операндах по синхронным действиям в композиции соответствует в синхронный τ-переход (в CSP после операции ***hide***, превращающей часть синхронных действий в τ), меняются состояния обоих операндов. Переходы по асинхронным действиям, т.е. действиям, для которых нет парного синхронного действия, а также τ- и γ-переходы в композиции сохраняются, меняется состояние только одного операнда.

Тестовый эксперимент понимается как композиция LTS реализации с LTS теста, определённой в том же (CSP) или «противоположном» (CCS) алфавите. Для наблюдения отказа, в тесте используется θ-переход, который срабатывает при обнаружении deadlock`а (в композиции превращается в τ-переход). В композиции реализации и теста есть только τ-переходы, в терминальных состояниях теста выносится вердикт.

4.2. *RlQ*-семантика

В наших исследованиях нас интересовали конформности типа редукции, основанные на трассах, состоящих из действий и отказов. Анализируя опыт тестирования АА и отношения ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$, мы пришли к выводу, что как модель, так и отношение конформности, прежде всего, определяются тем, какие тестовые воздействия мы можем производить над реализацией и какие наблюдения можем получить в ответ, в частности, какие отказы наблюдаемы, а какие нет. Скажем, для ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$ можно посылать стимул, но блокировку стимула можно наблюдать только для $\boldsymbol{ioco}_{\beta\gamma\delta}$.

Набор тестовых воздействий и наблюдений формализуется в понятии семантики взаимодействия. Эта семантика описывается с помощью, так называемой, машины тестирования, предложенной R.Milner`ом [[91]] и van Glabbeek`ом [[92],[94]]. Машина представляет собой «чёрный ящик», внутри которого находится реализация. Управление сводится к тому, что оператор машины, выполняя тест, нажимает кнопки на клавиатуре машины, разрешая реализации выполнять те или иные действия, которые могут наблюдаться на дисплее машины. В *реактивной* машине Milner`а нажимается только одна кнопка (разрешается только одно действие), в ответ поступает ровно одно наблюдение, после чего можно нажимать следующую кнопку. В *генеративной* машине van Glabbeek`а нажимается сразу несколько кнопок (разрешается множество действий), машина выполняет последовательность разрешённых действий, пока не будут нажаты другие кнопки. Отсутствие выполняемых действий может наблюдаться как отказ.

Van Glabbeek систематизировал возможные наблюдения, что легло в основу его классификации отношений конформности. Однако не все известные в литературе и используемые на практике отношения попали в эту классификацию. В частности, там нет отношения ***ioco***. Причина в том, что, по van Glabbeek`у, оператор может разрешать любое (или, как вариант, любое

конечное) множество действий. Кроме того, наблюдаются либо все возможные отказы, либо никакой отказ не наблюдаем. В то же время для ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$ множество разрешаемых действий не любое: это либо синглетон стимула, либо множество всех реакций. Множеству всех реакций соответствует наблюдаемый отказ (δ-наблюдение), а для стимула его блокировка не наблюдаема для ***ioco*** и наблюдаема для $\boldsymbol{ioco}_{\beta\gamma\delta}$.

Это натолкнуло нас на мысль ввести машину тестирования, параметризуемую той или иной семантикой взаимодействия. Наша машина похожа на машину Milner`a, но кнопкой разрешается не одно действие, а множество, как у van Glabbeek`a. Набор кнопок определяет набор тестовых воздействий. Отказы, соответствующие кнопкам, разделяются на наблюдаемые (***R***-семейство) и ненаблюдаемые (***Q***-семейство). В LTS наблюдаемые отказы обычно изображаются дополнительными переходами-петлями, помеченными отказами. Такую семантику мы назвали ***R***/***Q***-семантикой [[64],[70]]. Трассы наблюдений могут содержать действия и ***R***-отказы.

Заметим, что факультативный АА с e-переходами, равноприоритетными с посылающими и пустыми переходами, не моделируемый IOLTS, моделируется LTS общего вида. Соответствующая ***R***/***Q***-семантика содержит для каждого стимула (включая пустой стимул) кнопку, разрешающую реализации принимать этот стимул или выдавать любую реакцию. Альтернативой этому мог бы служить θ-переход в IOLTS-реализации, который в стационарном состоянии срабатывает тогда, когда тест не посылает стимулов. Однако e-переход в нестационарном состоянии не моделируется θ-переходом. Кроме того, обычно θ-переход допускается только в тесте, но не в реализации.

Что касается АА, в которых имеются приоритеты между переходами, в частности императивных АА (например, АОР), то они не моделируется обычными LTS, в которых никаких приоритетов нет: любое разрешённое действие может выполняться независимо от того, какие ещё действия разрешены. Этот факт отмечал уже van Glabbeek. В дальнейшем мы исследовали проблему приоритетов, и нам удалось определить не только LTS с приоритетами и соответствующие конформности, но и композицию таких LTS [[71]]. Такие LTS уже могут моделировать любые АА.

4.3. Гипотеза о безопасности и конформность *saco*

Гипотеза о безопасности в ***R***/***Q***-семантике обобщает гипотезы о допустимости стимулов в конечных автоматах, АА и IOLTS (для ***ioco***) и гипотезу о безопасности для $\boldsymbol{ioco}_{\beta\gamma\delta}$. Тестовое воздействие (нажатие кнопки на клавиатуре машины тестирования) считается опасным, если оно может привести к возникновению разрушения или бесконечному ожиданию наблюдения, что препятствует продолжению тестирования.

Разрушение моделирует любое нежелательное поведение системы, в том числе и ее реальное разрушение, которого нельзя допускать при тестировании. Бесконечное ожидание наблюдений бывает при отказе и при дивергенции. Отказ возникает при deadlock`e, когда никакие действия не могут выполняться и, следовательно, наблюдаться. Если отказ тоже ненаблюдаем, то никаких наблюдений не будет. Аналогичная ситуация возникает при тестовом воздействии во время дивергенции: реализация может бесконечно долго выполнять τ-переходы, и никаких наблюдений не будет. Заметим, что в отличие от семантики $\boldsymbol{ioco}_{\beta\gamma\delta}$ опасной считается не сама дивергенция, а попытка выхода из дивергенции.

Это определение даёт отношение ***safe in***: «кнопка безопасна в реализации после трассы». Для спецификации вводится отношение ***safe by***: «кнопка безопасна в спецификации после трассы», которое отличается для ***Q***-кнопок. Если такая кнопка безопасна после трассы, то трасса должна продолжаться каким-нибудь действием, разрешаемым этой кнопкой, хотя может продолжаться и соответствующим отказом. Кроме того, если трасса продолжается действием, которое разрешается кнопкой, не вызывающей разрушения или попытки выхода из дивергенции, то этой действие должно разрешаться некоторой безопасной кнопкой. Эти условия определяют отношение ***safe by*** неоднозначно, поэтому при задании ***R***/***Q***-семантики и спецификации дополнительно указывается отношение ***safe by***.

Безопасность кнопок определяет безопасные трассы, в которых каждое наблюдение может быть разрешено безопасной (после префикса трассы) кнопкой. Только безопасные по ***safe in*** трассы должны проходиться при безопасном тестировании. Требование безопасности тестирования выделяет класс безопасных реализаций – тех, которые можно безопасно тестировать для проверки их конформности или неконформности заданной спецификации. Этот класс определяется следующей *(трассовой) гипотезой о безопасности*: реализация безопасна для спецификации, если 1) в реализации нет разрушения с самого начала (до нажатия первой кнопки), если этого нет в спецификации, 2) после общей трассы, безопасной как в реализации (по ***safe in***), так и в спецификации (по ***safe by***) любая кнопка, безопасная (по ***safe by***) в спецификации, безопасна (по ***safe in***) после этой трассы в реализации.

Следует отметить, что гипотеза о безопасности не проверяема при тестировании и является его предусловием. После этого определяется отношение безопасной конформности ***saco*** [[64]]: реализация конформна спецификации, если она безопасна и выполнено *тестируемое условие*: после общей трассы, безопасной как в реализации (по ***safe in***), так и в спецификации (по ***safe by***) любое наблюдение, возможное в реализации в ответ на нажатие безопасной в спецификации (по ***safe by***) кнопки, разрешается, т.е. имеется в спецификации.

Большинство отношений конформности типа редукции оказываются частными случаями отношения ***saco*** при выборе соответствующей семантики.

Для трассового предпорядка ***R*** пусто, а ***Q*** состоит из одной кнопки, разрешающей все действия. Для ***ioco R*** состоит из одной кнопки приёма всех реакций, а ***Q*** состоит из кнопок-синглетонов для каждого стимула. Для $\boldsymbol{ioco}_{\beta\gamma\delta}$ ***R*** состоит из кнопки приёма всех реакций и кнопок-синглетонов для каждого стимула, а ***Q*** пусто.

Важно отметить, что конформность ***saco*** определяется для данной семантики только через трассы наблюдений, которые можно получить с помощью машины тестирования, маскирующей состояния LTS-реализации. Иными словами, ***saco*** использует только трассы реализации и спецификации и не зависит от их состояний. Поэтому наряду с основной моделью LTS мы использовали в наших исследованиях также *трассовую модель* как множество трасс LTS. В [[66]] для $\boldsymbol{ioco}_{\beta\gamma\delta}$ и в [[70]] для ***R***/***Q***-семантики даны необходимые и достаточные условия того, что множество трасс является трассовой моделью. В трассовой теории LTS – это способ конечного представления регулярных трассовых моделей.

В общем случае тестирование конформности ***saco*** основано на гипотезе о *глобальном тестировании* [[94]]: любое поведение реализации можно воспроизвести в тестовом эксперименте через конечное число попыток. Иногда используются более сильные гипотезы, например, указанные выше гипотеза о реализационных состояниях (для детерминированной реализации) или Δ-гипотеза. Если гипотеза о глобальном тестировании верна, существует полный (не обязательно конечный) набор тестов. Предложены алгоритмы генерации такого набора. При наличии рестарта набор конечных тестов эквивалентен одному (быть может, бесконечному) тесту, в котором нет бесконечной части без рестарта. Иногда удаётся построить конечный полный набор тестов (или один конечный тест, быть может, с использованием рестарта).

4.4. Полное тестирование с открытым состоянием ограниченно недетерминированных реализаций

Алгоритмы тестирования с открытым состоянием, предложенные для конечных автоматов и опирающиеся на обход (под)графа реализации при соответствующих гипотезах о реализации, легко переносятся на случай LTS в ***R***/***Q***-семантике. Они опираются на гипотезу о допустимости стимулов, которой в ***R***/***Q***-семантике соответствует гипотеза о безопасности, Δ-гипотезу и сильно-Δ-связность реализации (с учетом рестарта)[5]. При этом в ***R***/***Q***-семантике обходу по стимулам соответствует обход по кнопкам: после трассы, безопасной в спецификации, в каждом состоянии реализации нужно нажать каждую кнопку, безопасную в спецификации после этой трассы. Δ-переход определяется состоянием и кнопкой и означает множество переходов

[5] Если реализация детерминирована, то для неё автоматически выполнена Δ-гипотеза, а сильно-Δ-связность эквивалентна сильно-связности.

(включая добавленные переходы-петли по наблюдаемым отказам), выходящих из этого состояния и помеченных наблюдениями, разрешаемыми этой кнопкой.

Δ-гипотеза предполагает, что, если после нажатия кнопки в состоянии возможно ошибочное наблюдение, то оно возникает при первом нажатии этой кнопки в этом состоянии. Вместо этой довольно сильной гипотезы мы предложили для LTS в ***R***/***Q***-семантике *гипотезу об ограниченном недетерминизме*: для заданной константы t первые t нажатий кнопки в состоянии дают все возможные переходы Δ-перехода[6]. Если есть ошибка, она возникнет не обязательно после первого, а после одного из t нажатий. Если эта гипотеза выполнена, вместо сильно-Δ-связности реализации требуется более слабая сильно-связность.

Предложенный в [[72],[73],[85]] алгоритм тестирования выполняет параллельно две работы: исследование (learning) реализации и верификацию конформности. Оценка числа тестовых воздействий: $\boldsymbol{O}(nbt^n)$ для $t>1$, $\boldsymbol{O}(bn^2)$ для $t=1$, где b – число кнопок, а n – число состояний реализации. При наличии рестарта в каждом состоянии реализации для $t>1$ оценка равна $\boldsymbol{O}(bt^n)$. Оценка объёма вычислений (без тестовых воздействий): $\boldsymbol{O}(bn^2t^n)+\boldsymbol{O}(b^2tn^2)+\boldsymbol{O}(btn2^k)$, где k – число состояний спецификации, или первое слагаемое заменяется на $\boldsymbol{O}(bnt^n)$, если есть рестарты, или $\boldsymbol{O}(bn^3)$ для $t=1$. Для сильно-Δ-связных реализаций предложена модификация алгоритма с оценками $\boldsymbol{O}(btn^2)$ для числа тестовых воздействий и $\boldsymbol{O}(btn^3)+\boldsymbol{O}(b^2tn^2)+\boldsymbol{O}(btn2^k)$ для объёма вычислений. Заметим, что если LTS-спецификация детерминирована (каждая трасса заканчивается в одном состоянии), то сомножитель 2^k заменяется на k. Детерминированность LTS соответствует слабо-детерминированности конечных автоматов. Любую LTS можно превратить в детерминированную LTS (правда, переходы по наблюдаемым отказам будут не обязательно петлями) с помощью процедуры детерминизации [[83],[85],[86]][7].

4.5. Спецификационные тройки и отношения на них

Гипотеза о безопасности и конформность задаются спецификационной тройкой: семантика взаимодействия ***R***/***Q***, спецификационная модель (LTS или трассовая модель), отношение безопасности ***safe by***. Семантика определяет алфавит действий как все действия, разрешаемые всеми кнопками: $\boldsymbol{L} = (\cup\boldsymbol{R})\cup(\cup\boldsymbol{Q})$. Каждая тройка определяет в этом алфавите класс безопасных (по гипотезе о безопасности) реализаций и класс конформных (по ***saco***) реализаций. На одном из направлений наших исследований мы изучали вопрос о соотношении троек через соотношение определяемых ими классов

[6] $t=1$ соответствует детерминированному случаю.

[7] Справедливости ради, нужно отметить, что детерминизация LTS с k состояниями может дать LTS с 2^k-1 состояниями, хотя обычно до этого не доходит.

безопасных и конформных реализаций. Вложенность троек определяется как равенство классов конформных реализаций и вложенность классов безопасных реализаций. Это отношение является предпорядком и индуцирует соответствующее отношение эквивалентности троек. Преобразование троек назовём вложенным (эквивалентным), если тройка-аргумент вложена в (эквивалентна) тройке-результату. Вложенное преобразование даёт тройку, которую при тестировании можно использовать вместо исходной тройки.

Эти вопросы исследовались в работе [[69]], где особое внимание было уделено эквивалентности троек, отличающихся только отношением ***safe by*** (нормализация отношения) или имеющих общую спецификационную модель. Последнее порождает отношение «не сильнее» для семантик: первая семантика не сильнее второй семантики, если для каждой тройки с первой семантикой найдётся эквивалентная ей тройка со второй семантикой и той же самой спецификационной моделью (но, возможно с другим отношением ***safe by***). Соответственно определяется эквивалентность семантик. На классе эквивалентных семантик изучались условия существования минимальных и наименьших семантик по отношению вложенности семейств (***R*** и ***Q***) кнопок.

Классы реализаций, определяемые тройкой, являются подклассами реализаций в одном алфавите. Если алфавиты спецификационных троек разные, то классы реализаций также рассматриваются в разных алфавитах и, чтобы их сравнивать, нужно привести их к «единому знаменателю». Таким «знаменателем» может служить любой «целевой» класс интересующих нас реализаций: вместо классов конформных и безопасно-тестируемых реализаций рассматриваются их пересечения с этим «целевым» классом реализаций. Для наших целей нам было достаточно в качестве такого «целевого» класса брать класс всех моделей в некотором «целевом» алфавите ***L***. Мы определили «приведение» троек к целевому алфавиту, что дало возможность рассматривать отношения троек в разных алфавитах: ***L***-вложенность (и ***L***-эквивалентность), при которых сохраняется подкласс конформных и не сужается (сохраняется) подкласс безопасных реализаций в алфавите ***L***.

Для тестирования особый интерес представляет случай, когда две разные семантики определяют одинаковые тестовые возможности для тестирования реализаций в алфавите ***L***: если в кнопках оставить только действия из алфавита ***L***, то семантики совпадут. Такие семантики мы назвали ***L***-эквивалентными. Для ***L***-эквивалентных семантик в [[83]] рассматривались алгоритмические ***L***-вложенные и ***L***-эквивалентные преобразования.

4.6. Пополнение: удаление из спецификации ненаблюдаемых отказов

Исторически конформность ***saco*** возникла как обобщение ***ioco*** на общую ***R***/***Q***-семантику. Были наследованы и две основные проблемы конформности: 1) нерефлексивность и нетранзитивность, и 2) немонотонность.

Нерефлексивность ***saco*** следует из наличия ненаблюдаемых ***Q***-отказов, так же как нерефлексивность ***ioco*** следует из ненаблюдаемости блокировок стимулов. Эта проблема решается пополнением: вложенным преобразованием спецификационной тройки. Рефлексивность гарантируется тем, что пополнение строит модель спецификации без ***Q***-отказов, а транзитивность – тем, что пополнение определяет отношение ***safe by*** = ***safe in***.

В [[70]] показано существование преобразования, которое сохраняет класс конформных реализаций, определяет отношение ***safe by*** = ***safe in***, и не меняет семантики взаимодействия. Однако класс безопасных реализаций может сузиться, то есть такое преобразование не всегда вложенное. Показано, что без изменения семантики пополнение существует не для всякой спецификационной тройки. Поэтому было предложено ***L***-вложенное пополнение, которое изменяет семантику на ***L***-эквивалентную ей. Такое пополнение определено для трассовой модели, т.е. как преобразование множества трасс. Показано, что при конечном алфавите это преобразование алгоритмизуемо, исходная спецификация может быть задана в виде LTS. Строится при этом тоже LTS, но, вообще говоря, не конечная.

В [[83]] предложен усовершенствованный алгоритм пополнения (там оно называется ~пополнением), который по конечной LTS в конечном алфавите строит конечную LTS пополненной спецификации. Правда, этот алгоритм работает не для любого, а для, так называемого, ограниченного отношения ***safe by***: после безопасных трасс, заканчивающихся в одном множестве состояний LTS-спецификации, безопасность кнопок одинакова.

Пополненная спецификация в ***R***/***Q***-семантике эквивалентна пополненной спецификации в семантике, где все отказы наблюдаемы, т.е. в (***R***∪***Q***)/∅-семантике.

4.7. Монотонное преобразование

Как и для ***ioco*** проблема ***saco*** возникает в связи с композицией системы и заключается в том, что композиция реализаций, конформных своим спецификациям, оказывается, вообще говоря, не конформна композиции этих спецификаций. Для LTS композиция задаётся оператором параллельной композиции в духе CCS или CSP.

Первым шагом решения проблемы является пополнение спецификаций. Второй шаг – это монотонное преобразование спецификаций, при котором композиция реализаций, конформных своим спецификациям, конформна композиции монотонно преобразованных пополненных спецификаций компонентов. Кроме этого, монотонное преобразование, как и пополнение, должно: 1) не изменять тестовые возможности для тестирования реализаций в исходном алфавите ***L***, 2) сохранять конформность, т.е. сохранять подкласс конформных реализаций в алфавите ***L***, 3) не сужать возможность безопасного тестирования, т.е. не сужать подкласс безопасных реализаций в алфавите ***L***.

Это означает, что монотонное преобразование должно быть ***L***-вложенным преобразованием в ***L***-эквивалентную семантику.

Монотонное преобразование позволяет решить две задачи: 1) генерация спецификации системы по спецификациям компонентов, и 2) верификация декомпозиции системных требований. Для решения первой задачи выполняется пополнение и затем монотонное преобразование спецификаций компонентов, после чего они компонуются обычным образом. Вторая задача – это верификация «согласованности» уже имеющейся спецификации системы со спецификациями компонентов. Её решение опирается на тот факт, что спецификация системы, сгенерированная для решения первой задачи, во-первых, согласована со спецификациями компонентов, а, во-вторых, среди всех таких «согласованных» спецификаций системы предъявляет к системе наибольшие требования. Поэтому (по транзитивности конформности для пополненных спецификаций) достаточно проверить, что сгенерированная спецификация конформна заданной спецификации системы. Такая проверка может выполняться аналитически[8], в том числе, моделируя тестирование сгенерированной спецификации, рассматриваемой как реализация системы, по полному набору тестов, сгенерированному из заданной спецификации системы.

Особым случаем композиции является асинхронное тестирование или тестирование в контексте, который понимается как некая среда, с которой компонуется реализация по общим правилам композиции LTS. Предполагается, что среда известна и ней нет ошибок. При асинхронном тестировании возникают две проблемы: 1) «вседозволенность» (*permissiveness*), когда асинхронные тесты не ловят ошибку, обнаруживаемую синхронными тестами, и 2) «несохранение соответствия» (*non preservation of conformance*), когда асинхронные тесты ловят «ложную» ошибку. С первой проблемой, по-видимому, приходится мириться: асинхронное (и вообще композиционное) тестирование – это более «косвенное» (через контекст) тестирование реализации. Оно может не позволить создать все те режимы взаимодействия и наблюдать всё то поведение реализации, которые возможны в синхронном тестировании, когда тест и реализация взаимодействуют непосредственно друг с другом. Вторая проблема более серьёзная – это частный случай общей проблемы монотонности. Решение – монотонное преобразование, которое применяется только к спецификации реализации, а среда остаётся неизменной.

Глубинной причиной проблемы монотонности является различие в уровнях абстракции конформности и композиции. Конформность опирается на трассовую модель, а композиция – на более детальную (учитываются ещё и состояния) модель LTS. Дело в том, что на трассах наблюдений невозможно

[8] При аналитической верификации гипотезу о безопасности можно проверять так же, как проверяемое условие конформности.

определить такую композицию, по отношению к которой композиция LTS обладала бы свойством *аддитивности*: множество трасс композиции LTS совпадает с множеством всех попарных композиций трасс LTS-операндов. Это объясняется тем, что не всякое композиционное наблюдение вычисляется по паре наблюдений в операндах: отказы так не вычисляются.

Для решения этой проблемы вводятся, так называемые, ϕ-трассы (в [[66]] для ***ioco***$_{\beta\gamma\delta}$ и в [[70]] для ***R***/***Q***-семантики). Кроме действий, в ϕ-трассы входят, так называемые, ϕ-символы, а ϕ-символ содержит информацию о всех действиях, переходы по которым определены в стабильном состоянии[9]. Модель ϕ-трасс занимает промежуточный уровень абстракции между моделью трасс наблюдений и LTS, достаточный для определения композиции ϕ-трасс со свойством аддитивности. Кроме того, ϕ-трассы обладают свойством *генеративности*: по ϕ-трассам LTS вычисляются все её трассы наблюдений (обратное неверно). На базе ϕ-трасс строилась теория монотонности и разрабатывались алгоритмы монотонного преобразования в [[80]] для ***ioco***, в [[66]] для ***ioco***$_{\beta\gamma\delta}$ и в [[70]] для общей ***R***/***Q***-семантики.

4.8. Удаление из спецификации неконформных трасс

Генерация тестов происходит по безопасным трассам спецификации. После такой трассы нажимается безопасная кнопка и верифицируются полученные наблюдения. Однако, если эта трасса *неконформна*, т.е. не встречается в конформных реализациях, то ошибку можно обнаружить раньше, как только получена сама эта трасса. Поэтому возникает задача оптимизации генерации тестов с помощью удаления из спецификации неконформных трасс. Существование безопасных неконформных трасс следует из нерефлексивности отношения ***saco***. Пример таких трасс для отношения ***ioco*** смотри в [[64],[77],[80],[83]], есть даже пример спецификации, в которой все трассы неконформны [[77],[80],[83]]. В других семантиках спецификации могут содержать *неактуальные* трассы, которые не встречаются не только в конформных, но и в безопасных реализациях [[77],[83]].

Пополнение (см. п.4.6) удаляет из спецификации ***Q***-отказы, а в такой спецификации все трассы конформны. Однако это пополнение может менять семантику и, соответственно, расширять исходный алфавит действий ***L***. Из-за этого могут появиться новые конформные реализации, хотя и не в алфавите ***L***. Назовём трассу ***L***-конформной, если она встречается в конформных реализациях в алфавите ***L***. Понятно, что ***L***-конформная трасса конформна, но обратное не верно. Следовательно, для тестирования «старых» реализаций (т.е. в алфавите ***L***) нужны только ***L***-конформные трассы, а остальные трассы, в том числе конформные, можно удалить. В [[83]] преобразование спецификации, выполняющее эту операцию, названо ∇-пополнением. Оно

[9] Это похоже на *множества готовности* (ready sets) в *трассах готовности* (ready traces) [[92][94]].

выполняется после ~пополнения (удаления ***Q***-отказов). Разработаны алгоритмы ∇-пополнения; для конечных LTS-спецификаций в конечном алфавите с ограниченным отношением ***safe by*** строится ∇-пополнение в виде конечной LTS. Для ***ioco*** в [[80]] предложен упрощённый вариант такого алгоритма.

4.9. Финальные модели спецификации

Рассмотренный выше способ задания конформности ***saco*** в общей ***R***/***Q***-семантике обладает двумя существенными недостатками, усложняющими генерацию тестов по спецификации: 1) недетерминизм LTS-спецификации, 2) необходимость кроме описания спецификационной модели давать отдельное описание отношения ***safe by***.

Мы уже говорили выше, что для конформности ***saco*** в общей ***R***/***Q***-семантике достаточной трассовой модели. LTS – это порождающий граф трассовой модели, т.е. является компактным способом представления моделей, в частности, позволяет конечным образом описывать регулярные трассовые модели. Этот способ, однако, обладает существенным недостатком: LTS, вообще говоря, недетерминирована: трасса может заканчиваться не в одном, а в нескольких состояниях. Работать с такими трассами на LTS неудобно. В то же время этот недетерминизм вовсе не является неизбежным следствием недетерминизма моделируемой системы. Причина недетерминизма LTS в том, что в ***R***/***Q***-семантике наблюдения делятся на два вида: действия и отказы, которые существенно различным образом отображаются в LTS. Если трасса продолжается как отказом R, так и действием $z \in R$, то эти два продолжения не могут быть определены в одном и том же состоянии LTS. С другой стороны, любой порождающий граф можно сделать детерминированным с помощью процедуры детерминизации. Правда, при этом переходы по отказам не обязательно будут петлями. Кроме того, при обычной детерминизации теряется информация о дивергенции (бесконечном маршруте τ-переходов).

Это натолкнуло нас на мысль использовать новую модель, которую мы назвали RTS (Refusal Transition System) и которая представляется собой детерминированную LTS в объединённом алфавите действий и ***R***-отказов. Кроме того, допускаются γ-переходы, означающие разрушение, и Δ-переходы, означающие дивергенцию (вместо бесконечного маршрута τ-переходов). LTS преобразуется в RTS модифицированным алгоритмом детерминизации, который добавляет Δ-переходы из дивергентных состояний. RTS-модель мы использовали в [[83]] как вспомогательное средство для удаления неконформных трасс.

Что касается отношения ***safe by***, то оно однозначно определяется по LTS-спецификации для ***R***-кнопок и тех заведомо опасных ***Q***-кнопок, нажатие которых приводит к разрушению или попытке выхода из дивергенции. Остальные опасные ***Q***-кнопки задаются множеством трасс вида $\sigma \cdot Q$, где σ

безопасная трасса спецификации, а Q опасная после неё кнопка. Если это множество регулярно, ***safe by*** называется регулярным.

В [[86]] предложена, так называемая, *финальная* RTS, задающая одновременно спецификационную модель и отношение ***safe by***, для чего допускаются переходы по ***Q***-отказам. Есть два выделенных состояния: терминальное состояние ω и состояние γ, из которого выходит один переход, помеченный γ и ведущий в ω. В остальных состояниях нет γ-переходов, а все переходы по Δ и по ***Q***-отказам ведут в ω. Финальная RTS обладает рядом полезных для генерации тестов свойств, выгодно отличающих ее от LTS в алфавите действий:

1. **Детерминизм**. RTS детерминирована, следовательно, каждая трасса, по которой нужно генерировать тесты, соответствует одному маршруту в графе спецификации и заканчивается в одном состоянии.
2. **Безопасные кнопки**. Кнопка P безопасна после трассы, если конечное состояние трассы отлично от ω и γ, в нём нет Δ-перехода и, если P – это ***Q***-кнопка, нет перехода по P.
3. **Безопасные трассы**. Безопасные трассы спецификации – это все ее трассы, заканчивающиеся в состояниях, отличных от ω и γ.

Любую LTS-спецификацию с любым отношением ***safe by*** можно преобразовать в эквивалентную ей (по классам безопасных и конформных реализаций) финальную RTS-спецификацию. Предложен алгоритм такого преобразования, который строит конечную RTS по конечной LTS и конечному порождающему графу регулярного отношения ***safe by***.

5. Расширения R/Q-модели

В этом разделе мы рассмотрим три важнейших расширения ***R***/***Q***-теории: медиаторы, приоритеты и симуляцию.

5.1. Медиаторы: тестирование с преобразованием семантик

Теория конформности строится в предположении, что реализация и спецификация заданы в одной семантике. Однако на практике, как правило, требуется некоторое преобразование спецификационных тестовых воздействий (кнопок) в реализационные тестовые воздействия (кнопки) и обратное преобразование реализационных наблюдений в спецификационные наблюдения. Программа, осуществляющая эти преобразования, называется медиатором. По сути, это означает, что реализация и спецификация заданы в разных семантиках, и медиатор осуществляет преобразование семантик.

В простейшем случае различие семантик только в разных способах представления одних и тех же кнопок и наблюдений в реализации и спецификации. Медиатор выполняет взаимно-однозначное преобразование алфавитов реализации и спецификации (кнопок – в кнопки, действий – в действия и отказов – в отказы). В общем случае такого преобразования

недостаточно, поскольку спецификация обычно определяется на более высоком уровне абстракции. Медиатор может оказаться довольно сложной программой, осуществляющей медиативные функции между реализацией в одной семантике и тестом, генерируемым по спецификации в другой семантике. В частности, одному нажатию спецификационной кнопки может соответствовать сеанс взаимодействия медиатора с реализацией.

Тестирование через медиатор похоже на тестирование в контексте. Там тоже тест взаимодействует с реализацией не напрямую, а через промежуточную среду взаимодействия. Отличие в том, что при тестировании в контексте спецификация задаётся в той же семантике, что и реализация. При тестировании по фактор-спецификации (см. п.0) также выполняется преобразование спецификационных кнопок в реализационные кнопки и реализационных наблюдений в спецификационные наблюдения. Но при этом используется фактор-гипотеза (эквивалентность состояний и переходов). При тестировании через медиатор, вообще говоря, фактор-гипотеза не нужна, но, если медиаторное преобразование зависит от реализационного состояния, то требуется возможность опроса реализационного состояния. Другое дело, что тестирование через медиатор обычно совмещается с факторизацией спецификации и тогда предполагает выполнение фактор-гипотезы.

В [[74]] определяется медиаторное преобразование и соответствующим образом модифицируются гипотеза о безопасности и конформность ***saco***. Для тестирования через медиатор предлагается модификация алгоритма полного тестирования с открытым состоянием ограниченно недетерминированных реализаций (см. п.4.4). Отличие в том, что число t нажатий кнопки, достаточное для получения всех наблюдений в состоянии, зависит от медиаторного преобразования, т.е. перестаёт быть константой. Вместо него вводится ответ медиатора «все наблюдения получены».

Для тестирования с открытым состоянием также рассматривается медиаторное преобразование состояний, когда по состоянию реализации «вычисляется» соответствующее ему спецификационное состояние. Это означает, что на множестве состояний реализации существует отношение эквивалентности, и состояние спецификации взаимно-однозначно соответствует классу эквивалентных состояний реализации. В отличие от факторизации это соответствие является не предусловием тестирования (реализационной гипотезой), а частью проверяемого условия. По сути, меняется сама конформность, проверяемая таким тестированием: вместо редукции – частный случай слабой симуляции (см. ниже п. 5.3).

5.2. Приоритеты

Математические модели взаимодействия, как правило, абстрагируются от приоритетов: считается, что любое разрешённое действие может выполняться независимо от того, какие ещё действия разрешены или не разрешены. Тем самым, вводится излишний недетерминизм в описание поведения систем. В то

же время на практике приоритеты широко используются. Приведём только три примера. 1) Приоритетная обработка запросов или сообщений, в том числе аппаратных прерываний. В IOLTS этому соответствуют взаимные приоритеты переходов по разным стимулам. 2) Выход из дивергенции, когда запрос, поступающий извне, прерывает внутреннюю активность системы, хотя без этого запроса она могла бы продолжаться бесконечно. В LTS этому соответствует приоритет перехода по действию над τ-переходом. 3) Прерывание цепочки действий – операция *cansel*. Этому соответствует приоритет перехода по cansel над всеми остальными переходами.

В нашей работе [[71]] предложена модель LTS с приоритетами, в которой переход помечается не только действием, но и множеством разрешаемых действий (МРД). Такой переход выполняется только тогда, когда разрешено выполнять именно это множество действий. В системах с приоритетами понятия стабильности и дивергенции становятся условными – в зависимости от нажатой кнопки. Состояние стабильно, если никакой переход не может выполняться при нажатии этой кнопки, и дивергентно, если в нём начинается бесконечный τ-маршрут, все переходы которого помечены этой кнопкой. Также появляется возможность *переключения кнопок*, т.е. нажатие следующей кнопки, не дожидаясь наблюдения. Дело в том, что такое переключение меняет МРД и, тем самым, выполняемые переходы в реализации. В частности, можно выйти из дивергенции по кнопке, нажав другую кнопку. Также можно нажимать кнопку, которая может вызвать ненаблюдаемый отказ, если после этого, не дожидаясь наблюдения (его может и не быть), нажать другую кнопку.

Если приоритетов нет, возможность наблюдения действия не зависит от того, какая именно нажимается кнопка, разрешающая это действие. При наличии приоритетов это становится важным, поскольку для одной такой кнопки действие может наблюдаться, а для другой – нет, и это может отражаться в спецификационных требованиях. Поэтому теперь нужно запоминать не только наблюдения, но также нажимаемые кнопки: результатом тестового эксперимента становится трасса как последовательность не только наблюдений (действий и отказов), но также и кнопок. Предложены основанные на таких трассах модификации гипотезы о безопасности и конформности ***saco*** для систем с приоритетами как без переключения кнопок, так и с переключением кнопок.

В композиции LTS с приоритетами переход из состояния одного операнда участвует в композиции только тогда, когда он помечен таким МРД, что его подмножество синхронных действий совпадает с множеством синхронных действий, по которым определены переходы в парном состоянии другого операнда. МРД асинхронного перехода композиции приводится к алфавиту композиционной LTS, т.е. из него удаляются синхронные действия. Синхронный переход композиции соответствует паре переходов в операндах и

помечается объединением МРД этих переходов, из которых удалены синхронные действия.

Для сокращения записи предложено также другое представление LTS с приоритетами, в котором множество кратных переходов по одному и тому же действию, но разным МРД, заменяется одним переходом по этому действию, помеченному предикатом от МРД. Тождественно ложный предикат соответствует отсутствию перехода, а тождественной истинный предикат – множеству кратных переходов по всем возможным МРД.

Для заданной **R**/**Q**-семантики LTS теста, как и раньше, определяется в том же (CSP) или «противоположном» (CCS) алфавите. Множество действий, по которым определены переходы из состояния теста, совпадает с одной из кнопок. Для обнаружения отказов используется θ-переход. Все переходы теста помечены тождественно истинным МРД.

5.3. Симуляция

Кроме конформностей, основанных на трассах наблюдений, существуют также конформности, основанные на соответствии состояний реализации и спецификации. Такие конформности называются симуляциями и требуют, чтобы правильным было не только наблюдаемое внешнее поведение реализации, но и изменение ее состояний. Выбор симуляции в качестве конформности наиболее естественен, когда состояния реализации доступны для их наблюдения, т.е. при тестировании с открытым состоянием.

Однако 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 [[75],[79],[84]] мы исследовали расширение ***R***/***Q***-теории на симуляции, выбрав наиболее практический вариант слабой или наблюдаемой симуляции, которая основана на принципиальной ненаблюдаемости τ-переходов.

Слабая симуляция определяется как существование соответствия[10] состояний реализации и спецификации, при котором начальные состояния соответствуют друг другу, и если в реализации в состоянии *i* наблюдается действие *z* (до и после которого возможны τ-переходы), а среди постсостояний есть состояние *i*`, то в спецификации в любом состоянии *s*, соответствующем *i*, также наблюдается действие *z* (до и после которого возможны τ-переходы), а среди постсостояний найдётся хотя одно состояние *s*`, соответствующее *i*`. В ***R***/***Q***-семантике наблюдаться могут не только действия, но и ***R***-отказы, поэтому в определении слабой симуляции под *z* понимается любое наблюдение: действие или ***R***-отказ.

[10] Если соответствие симметричное, то это бисимуляция.

Сложнее обстоит дело с безопасностью тестирования. Для трассовой конформности ***saco*** безопасность кнопки в реализации определялась после трассы. Это значит, что кнопка безопасна в каждом состоянии после трассы, т.е. в состоянии нет дивергенции и нажатие кнопки в этом состоянии не приводит к разрушению и ненаблюдаемому отказу. Поскольку для симуляции мы предполагаем наблюдаемость состояний реализации, мы можем нажимать кнопку, которая безопасна не в каждом состоянии после трассы, а в наблюдаемом состоянии.

Соответственно, гипотеза о безопасности строится на соответствии состояний реализации и спецификации, которое мы назвали *H-соответствием*. Начальные состояния *H*-соответствуют друг другу, если в них нет разрушения (в том числе после цепочки τ-переходов). Неначальные состояния *H*-соответствуют друг другу, если они могут быть достигнуты из начальных состояний нажатием кнопок, которые безопасны как реализации, так и в спецификации. Основанную на *H*-соответствии гипотезу о безопасности для симуляции мы назвали *H-гипотезой*. Она требует: 1) в реализации нет разрушения с самого начала (до нажатия первой кнопки), если этого нет в спецификации[11], 2) если кнопка безопасна в состоянии спецификации, то она безопасна в *H*-соответствующем ему состоянии реализации. Отметим, что *H*-гипотеза предъявляет к реализации более сильные требования, чем трассовая гипотеза о безопасности.

Конформность, которую мы назвали *безопасной симуляцией* и обозначили ***ss***, имеет в качестве предусловия тестирования *H*-гипотезу. Проверяемое при тестировании условие означает существование такого соответствия $R \subseteq H$, что 1) начальные состояния *R*-соответствуют друг другу, если в начальном состоянии спецификации нет разрушения (в том числе после цепочки τ-переходов), 2) если кнопка безопасна в состоянии *s* спецификации и в *R*-соответствующем ему состоянии *i* реализации её нажатие вызывает наблюдение *z* (до и после которого возможны τ-переходы) с постсостоянием *i`*, то в состоянии *s* тоже есть наблюдение *z* (до и после которого возможны τ-переходы) и одно из постсостояний *s`* *R*-соответствует *i`*. Можно также рассматривать безопасную симуляцию, основанную не на *H*-гипотезе, а на трассовой гипотезе о безопасности; мы обозначили такую конформность ***sst***. Показано, что $\boldsymbol{ss} \subset \boldsymbol{sst} \subset \boldsymbol{saco}$.

Тестирование безопасной симуляции, также как для ***saco***, основано на гипотезе о глобальном тестировании. Однако полное тестирование не всегда возможно: если для ***saco*** неконформность любой реализации можно определить за конечное время, то для ***ss*** это не так. Тем не менее предложен общий алгоритм значимого тестирования, которое на некотором подклассе спецификаций исчерпывающее и, следовательно, полное (определяет

[11] Первое требование совпадает с первым требованием трассовой гипотезы о безопасности.

неконформность за конечное время). В частности, этот подкласс содержит все конечные спецификации в конечном алфавите.

На этом подклассе спецификаций для конечных и ограниченно недетерминированных реализаций предложены алгоритмы полного тестирования за конечное время[12] в предположении, что рестарт возможен в любой момент времени. Один алгоритм является модификацией общего алгоритма, а другой, более быстрый, алгоритм аналогичен алгоритму тестирования таких реализаций для ***saco*** (см. п.4.4). Сначала выполняется обход (под)графа реализации, потом (без тестирования) строится *R*-соответствие и, если его удаётся построить, выносится вердикт ***pass***, иначе – вердикт ***fail***.

Для быстрого алгоритма число тестовых воздействий такое же, как в п.4.4, т.е. $\boldsymbol{O}(bt^n)$ для $t>1$, и $\boldsymbol{O}(bn^2)$ для $t=1$. Объём вычислений равен $\boldsymbol{O}(bnt^n)+\boldsymbol{O}(b^2n^2t)+\boldsymbol{O}(bntk^2)$ для $t>1$, для $t=1$ первое слагаемое заменяется на $\boldsymbol{O}(bn^3)$.

6. Модель наблюдений и модель событий

6.1. Критика *R*/*Q*-модели

R/***Q***-семантика обобщила многие известные конформности, что позволило единообразно ставить и решать многие общие проблемы, основными из которых являются проблемы оптимизации генерации тестов, композиции систем и учёта приоритетов. В то же время в решении этих проблем выявились серьёзные трудности: алгоритмы пополнения спецификации (удаления ненаблюдаемых отказов), удаления из спецификации неконформных трасс и монотонного преобразования оказались довольно сложными и громоздкими, что затрудняет их применение на практике. Анализ этих трудностей показал, что их причины лежат достаточно глубоко: в самой семантике взаимодействия и выбранных моделях реализации и спецификации. При всей их общности ***R***/***Q***-семантика и соответствующие ей модели недостаточно общи: в них существуют внутренние зависимости, которые и приводят к отмеченным усложнениям и трудностям.

Во-первых, нет «просто» наблюдений: они делятся на действия и отказы с существенно разной семантикой. Отказ – не просто «другое» по сравнению с действиями наблюдение, а именно множество разрешаемых, но не выполнимых действий. После отказа P не может наблюдаться действие $z\in P$. Кроме того, отказы возникают только в стабильных состояниях, где нет τ- и γ-переходов.

[12] Эти ограничения на спецификацию и реализацию такие же, как для ***saco*** (4.4).

Во-вторых, нет «просто» тестовых воздействий: каждая моделирующая тестовое воздействие кнопка машины тестирования соответствует множеству действий, а само тестовое воздействие (нажатие кнопки) – это разрешение реализации выполнять действия именно из этого множества. Соответственно, после тестового воздействия возможно не любое (но, конечно, зависящее от реализации) наблюдение: после нажатия кнопки P может наблюдаться только действие $z \in P$ или отказ P, если он наблюдаем. Отказы могут быть не только наблюдаемыми (***R***-отказы), но также и ненаблюдаемыми (***Q***-отказы). Тем самым, кнопки тоже подразделяются на ***R***- и ***Q***-кнопки, что влияет как на безопасность тестирования, так и на конформность, и приводит к проблеме пополнения.

В-третьих, представление спецификационных требований в виде той же модели, что и для тестируемой системы, обосновано только исторически – оно возникло из идеи об эквивалентности автоматов. Однако такое представление затрудняет генерацию тестов по спецификации. Прежде всего, спецификация может содержать неконформные трассы, которые не нужны для генерации тестов, что и вызывает необходимость их удаления из спецификации. Это, однако, не решает всей проблемы оптимизации тестов.

Возможность и необходимость оптимизации тестов объясняется наличием зависимостей между ошибками, определяемыми спецификацией. Спецификация, описывая «правильное» поведение реализации, задаёт (хотя и неявно) множество A всех ошибок: реализация конформна, если в ней нет ни одной ошибки (из A). Зависимость между ошибками означает, что существует строгое подмножество ошибок $B \subset A$ такое, что любая реализация, содержащая ошибку из A, содержит также какую-то ошибку из B. Поэтому для полного тестирования вместо тестов, которые ловят все ошибки (из A), достаточно сгенерировать тесты, которые ловят все ошибки из B.

Конформность в ***R***/***Q***-семантике основана на трассах наблюдений или, при наличии приоритетов, на более общих трассах наблюдений и кнопок. Поэтому ошибка – это трасса. Отсюда следуют две тривиальные зависимости между ошибками. Во-первых, после наблюдения трассы σ всегда можно нажать кнопку[13] P, следовательно, если σP ошибка, то σ тоже ошибка. Во-вторых, если наблюдается трасса σ, то перед этим наблюдается каждый её префикс μ, следовательно, если μ ошибка, то σ тоже ошибка. Тривиальные зависимости определяют тривиальную оптимизацию тестов, которая не создаёт проблем.

К сожалению, в ***R***/***Q***-семантике тривиальными зависимостями дело не ограничивается. Даже после удаления из спецификации ненаблюдаемых

[13] Если кнопка опасна, её нельзя нажимать при безопасном тестировании, но это не значит, что её вообще нельзя нажимать. Иными словами, ошибкой считается любая трасса, которая не наблюдается при взаимодействии с конформными реализациями, а не только безопасная.

отказов и неконформных трасс всё равно остаются нетривиальные зависимости между ошибками [[83]]. Глубинная причина наличия таких зависимостей лежит в наличии зависимостей внутри семантики и модели при том, что спецификационная модель совпадает с реализационной.

Поскольку спецификационная модель того же типа, что модель реализации, композиция спецификаций традиционно выполняется так же, как композиция любых LTS. Из-за этого возникает несохранение конформности при композиции и необходимость монотонного преобразования спецификаций-операндов, которая выполняется перед их композицией. На самом деле, нет никаких причин ни для того, чтобы спецификация задавалась моделью того же типа, что реализация, ни для того, чтобы компоновать спецификации так же, как реализации. Возникло предположение, что при другой модели спецификации и при другой композиции спецификаций проблема сохранения конформности при композиции будет решена гораздо проще.

Что касается приоритетов, то они вносятся в ***R***/***Q***-модель как чистая «добавка», что усложняет модель и соответствующие алгоритмы. Кроме того, проблема композиции до сих пор не решена для систем с приоритетами. Возникло предположение, что всё было бы проще, если бы приоритеты были естественной частью самой семантики взаимодействия.

Всё это натолкнуло нас на мысль исследовать семантику более общего вида, в которой нет внутренних зависимостей, но есть приоритеты. Первая попытка обобщения ***R***/***Q***-семантики была предпринята в работе [[68]]. Там введена, так называемая, ***P***-семантика, в которой разорвана жёсткая связь между тестовым воздействием (кнопкой) и наблюдаемым отказом: наблюдаться может не только отказ, совпадающий с множеством разрешаемых действий. Теперь кнопка – это множество наблюдений, которые разрешаются этой кнопкой: как действий, так и отказов. При нажатии кнопки P отказ $r \in P$ возникает в стабильном состоянии, если в этом состоянии не определены переходы по действиям $z \in r$ (а не $z \in P$ как в случае ***R***/***Q***-семантики). В ***P***-семантике удалось определить безопасное тестирование конформности и разработать алгоритмы генерации тестов, а также ввести приоритеты, но тоже как «добавку» аналогично ***R***/***Q***-семантике. Однако в ***P***-семантике остались как разделение наблюдений на действия и отказы, так и понимание тестового воздействия как множества разрешаемых наблюдений. Поэтому она не получила дальнейшего развития.

6.2. Модель наблюдений

В [[87],[88]] введена семантика общего вида, в которой нет внутренних зависимостей и есть приоритеты. Она задаётся двумя непересекающимися множествами: ***B*** – кнопок и ***O*** – наблюдений, без каких-либо дополнительных ограничений. Мы назвали её ***B***/***O***-семантикой. В ***B***/***O***-семантике LTS реализации содержит явные переходы как по наблюдениям, так и по кнопкам, а также, как обычно, ненаблюдаемые τ-переходы. Её можно назвать *LTS*

наблюдений – OTS (от *observation* – наблюдение) в отличие от LTS в алфавите действий, которая используется в ***R***/***Q***-семантике и которую можно назвать *LTS действий – ATS* (от *action* – действие). Предполагается, что в процессе взаимодействия с реализацией мы получаем поток наблюдений, а тестовое воздействие (нажатие кнопки) лишь регулирует этот поток. Если тестового воздействия нет (никакая кнопка не нажата), реализация может выполнить любую цепочку переходов по наблюдениям и τ-переходов, начинающуюся в её текущем состоянии. При понимании тестового воздействия в ***B***/***O***-семантике мы исходили из двух, обычно противоречащих друг другу, предположений.

Первое предположение – это приоритет тестовых воздействий над поведением реализации: не только наблюдаемым, но и ненаблюдаемым, что даёт возможность учёта приоритетов. В [[87],[88]] показано, как в ***B***/***O***-семантике моделируются различные системы с приоритетами, в том числе системы из примеров п. 5.2. В качестве курьёза можно отметить, что модель наблюдений с таким предположением больше всего похожа на императивный АА, если стимул понимать как кнопку, а реакцию – как наблюдение. Однако императивный АА при наличии стимула не мог сделать ни одного τ-перехода, а в ***B***/***O***-семантике это разрешено вторым предположением.

Второе предположение – это основное допущение о ненаблюдаемом поведении: реализация всегда может выполнять τ-переходы независимо от тестовых воздействий.

В совокупности эти два предположения определяют следующий протокол взаимодействия: нажатие кнопки блокирует наблюдения, реализация может выполнять τ-переходы, но только конечное их число, после чего должна выполнить любой переход по нажатой кнопке. Этот переход по кнопке означает, что реализация «приняла к сведению» тестовое воздействие через конечное время после нажатия кнопки. В то же время мы никак не оговариваем, что означает это «принятие к сведению», допускается и простое игнорирование тестового воздействия, что в LTS моделируется переходом-петлёй по кнопке.

Для того, чтобы реализация всегда имела возможность «принять к сведению» тестовое воздействие, отсутствие перехода по кнопке в состоянии интерпретируется как наличие перехода-петли по этой кнопке. Разрушение γ понимается как одно из наблюдений, но при безопасном тестировании его не должно быть.

В новой модели мы по-прежнему рассматриваем трассовые конформности типа редукции. Как уже было сказано в п.2.2, редукция является предпорядком, если множество трасс спецификации – это множество все конформных трасс. Для генерации тестов наиболее удобно, чтобы спецификация явно задавала множество всех ошибок, т.е. дополнение множества конформных трасс. В этом случае тесты строятся по каждой трассе

спецификации или строится один адаптивный тест, по сути, совпадающий со спецификацией. Тест выносит вердикт ***fail***, если получена ошибка, т.е. одна из трасс спецификации.

В то же время наличие тривиальных зависимостей, присущих любой трассовой конформности, позволяет задавать в спецификации не все ошибочные трассы, а только те из них, которые, во-первых, не заканчиваются кнопкой (заканчиваются наблюдением), и, во-вторых, все строгие префиксы которых не являются ошибками. Такую спецификацию мы назвали *нормализованной* и определили простейший алгоритм *нормализации* спецификации.

Для компактного представления спецификаций и тестов используются порождающие графы, которые конечны для регулярных множеств трасс. Такие графы всегда можно сделать детерминированными с помощью процедуры детерминизации. По сути, это детерминированные LTS в алфавите кнопок и наблюдений. Поскольку множество ошибок не является префикс-замкнутым (оно постфикс-замкнуто), состояния, в которых заканчиваются трассы-ошибки, помечаются как конечные.

Новая модель позволила по-новому поставить вопрос о зависимостях между ошибками. В ***B***/***O***-семантике нет нетривиальных зависимостей на классе всех реализаций, однако если ограничиться тем или иным подклассом реализаций, то на нём такие зависимости уже могут появиться просто потому, что реализации, которые содержат одну данную ошибку и не содержат остальных ошибок, могут оказаться вне этого подкласса.

Подклассы реализаций возникают, прежде всего, из-за тех или иных гипотез о безопасности, для которых тестируемые реализации ограничиваются подклассом безопасных реализаций. В то же время для ***B***/***O***-семантики без каких-либо дополнительных ограничений тоже существуют естественные гипотезы о безопасности, определяемые двумя требованиями: 1) время ожидания наблюдения конечно – λ-гипотеза, 2) при тестировании не должно возникать разрушение реализации – γ-гипотеза.

λ-гипотеза означает, что если в префикс-замыкании спецификации трасса продолжается наблюдением, то при тестировании реализации после этой трассы гарантированно должно быть какое-нибудь наблюдение. Такая гарантия имеет место, если в реализации трасса не заканчивается в дивергентных состояниях и в каждом стабильном состоянии после трассы есть переход по какому-нибудь наблюдению. λ-гипотеза вводит новую зависимость между ошибками: если для каждого наблюдения *и* трасса μ*и* является ошибкой, то трасса μ неконформна, т.е. тоже ошибка. Соответствующая процедура λ-нормализации дополнительно к обычной нормализации добавляет в спецификацию ошибку μ и удаляет все ошибки μ*и*.

γ-гипотеза означает, что при тестировании по данной спецификации в реализации не возникает разрушение. Это сужает класс тестируемых

реализаций до подкласса реализаций, удовлетворяющих γ-гипотезе, но не вводит дополнительных зависимостей между ошибками. Совокупность λ- и γ-гипотез как раз и определяют естественный для ***B***/***O***-семантики подкласс безопасных реализаций, что учитывается в относительно простой процедуре λ-нормализации спецификации.

Полезность ***B***/***O***-семантики определяется тем, что ею моделируются все другие семантики с трассовыми конформностями типа редукции, описанные в [[94]], а также класс ***R***/***Q***-семантик, включая ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$. Это моделирование определяется алгоритмом преобразования ATS в OTS, при котором класс всех ATS отображается в строгий подкласс класса всех OTS. Тем самым, в ***B***/***O***-семантике исходная конформность рассматривается не на классе всех LTS-реализаций, допускаемых ***B***/***O***-семантикой, а на подклассе преобразованных LTS-реализаций. Из-за этого и возникают дополнительные зависимости между ошибками, требующие дополнительной оптимизации тестов.

Так мы показали, что проблема оптимизации тестов для различных семантик оказывается частным случаем общей проблемы оптимизации тестов при ограничении тестируемых реализаций тем или иным подклассом. Вот лишь несколько общих примеров подклассов реализаций, которые применяются на практике и не связаны с гипотезами о безопасности: 1) подкласс LTS-реализаций с ограниченным числом состояний; 2) конечный (с точностью до изоморфизма) подкласс тестируемых реализаций; 3) конечный подкласс неконформных реализаций класса тестируемых реализаций (в [[104]] такой подкласс называется *классом неисправностей*); 4) подкласс реализаций, в котором каждая неконформная реализация содержит ошибку из некоторого заранее заданного конечного множества ошибок. В последнем случае обычно говорят, что набор тестов нацелен на поиск ошибок из указанного конечного множества, а другие ошибки могут не обнаруживаться.

6.3. Модель событий

Как было сказано в п. 4.7, проблема композиции возникает из-за неаддитивности трасс наблюдений. «Виноваты» в этом отказы, поскольку трассы действий аддитивны. Монотонное преобразование в ***R***/***Q***-семантике удалось сделать при помощи ϕ-трасс, которые обладают свойствами аддитивности и генеративности.

B/***O***-семантика тоже основана на трассах наблюдений (только общего вида), которые, вообще говоря, тоже не аддитивны. Поэтому в [[89]] предложена ***E***/***O***-семантика и основанная на ней *модель событий* (*ETS* – от event – событие). Вводится универсум событий ***E***, и вместо переходов по наблюдениям в ETS используются переходы по событиям. Генеративность обеспечивается универсальной частично-определённой однозначной функцией $\boldsymbol{f}:\boldsymbol{E}\rightarrow\boldsymbol{O}$. Разрушение γ считается как событием, так и наблюдением, и $\boldsymbol{f}(\gamma)=\gamma$. Переходы совершаются только по *наблюдаемым событиям*, т.е. событиям из домена функции ***f***. Ненаблюдаемые события, однако,

необходимы для аддитивности композиции; композиция событий, одно или оба из которых ненаблюдаемы, может быть наблюдаемым событием. Для композиции события разделяются на синхронные и асинхронные и задаётся коммутативная операция композиции событий, превращающая два события-операнды в событие-результат. Разрушение считается асинхронным, а кнопки всегда синхронны. На этом строится композиция ETS в духе CCS (с последующим сокрытием части синхронных символов операцией ***hide***), которая обладает свойством аддитивности.

Если бы спецификация задавалась множеством всех конформных (встречающихся в конформных реализациях) трасс событий, то свойство аддитивности гарантировало бы сохранение конформности при композиции. При этом множество трасс событий композиции спецификаций определялось бы просто как множество всех попарных композиций трасс событий спецификаций-операндов. Однако спецификация в новой модели – это нормализованное (или λ-нормализованное, если используется λγ-гипотеза о безопасности) множество ошибок. Поэтому перед композицией сначала 1) в OTS-операндах выполняется «разнормализация», дающая множество всех ошибок, 2) потом строится дополнение этого множества, т.е. множество всех конформных трасс наблюдений, 3) потом выполняется преобразование в OTS→ETS (задаваемой обратной функцией $\boldsymbol{f}^{-1}$). Получившиеся ETS-операнды компонуются, после чего нужно сделать обратные преобразования: 1) ETS→OTS (задаваемое функцией ***f***), 2) построить дополнение получившегося множества всех конформных трасс наблюдений композиции, т.е. построить множество всех ошибок, определяемых композицией, 3) выполнить нормализацию (или λ-нормализацию). Заметим, что эта процедура композиции спецификаций определяется на трассах, но разработан алгоритм её выполнения для регулярных спецификацией, задаваемых конечными OTS.

В [[89]] также предложен способ моделирования в ***E***/***O***-семантике семантик с трассовыми конформностями типа редукции, описанные в [[94]], включая семантику трасс готовности (ready trace semantics), а также класса ***R***/***Q***-семантик, включая ***ioco*** и $\boldsymbol{ioco}_{\beta\gamma\delta}$. Для этого предложены алгоритмы преобразования ATS-реализаций с соответствующей семантикой в ETS. Показано, что такое моделирование согласовано с композицией: композиция ETS, полученных в результате моделирования исходных ATS с исходными семантиками, эквивалентна ETS, полученной в результате моделирования композиции этих ATS. Здесь эквивалентность понимается как совпадение множеств трасс событий.

7. Параллельное тестирование

В заключении этой статьи мы хотим отметить, что почти все теоретические и все практические алгоритмы тестирования, разработанные нами для различных семантик и моделей, основаны на исследовании (обходе)

(под)графа: (под)графа спецификации при тестировании с закрытым состоянием или (под)графа реализации при тестировании с открытым состоянием. Задача исследования (learning) графа, тем самым, ключевая для полноты тестирования. Ей была посвящена серия наших работ [[57],[58],[60]-[62]].

В то же время за последние годы размер реально используемых систем и сетей и, следовательно, размер их моделей и, следовательно, размер исследуемых графов непрерывно растёт. Проблемы возникают тогда, когда исследование графа одним автоматом (компьютером) либо требует недопустимо большого времени, либо граф не помещается в памяти одного компьютера, либо и то и другое. Поэтому возникает задача параллельного и распределённого исследования графов. Эта задача формализуется как задача исследования графа коллективом автоматов, т.е. несколькими параллельно работающими компьютерами с достаточной суммарной памятью. Автоматы могут обмениваться между собой сообщениями для синхронизации их действий по исследованию графа.

В работах [[81],[82]] упор сделан на ускорении тестирования за счёт распараллеливания. При этом считается, что память каждого компьютера достаточна для хранения всего исследуемого графа. Это формализуется как обход графа коллективом автоматов, которым разрешено не только обмениваться сообщениями, но и писать/читать пометки в вершинах графа. В частности, в [[81],[82]] выполнялось функциональное тестирование различных подсистем модели процессора: кэш третьего уровня, управление прерываниями и пр. Модельные графы содержали от нескольких тысяч до нескольких миллионов узлов и несколько миллионов дуг. Тест выполнялся максимально на 150 компьютерах.

Если исследуемый граф помещается только в суммарную память всех компьютеров, но не помещается в память одного компьютера, задача резко усложняется. Она формализуется как обход графа коллективом автоматов, которым разрешено обмениваться сообщениями, но не разрешено оставлять какие-то пометки на графе (что и означает распределённое размещение графа на нескольких компьютерах). Нами предложен эффективный алгоритм такого исследования детерминированного графа, подробно изложенный в нашей работе «Обход неизвестного графа коллективом автоматов», которая в настоящее время подготовлена к печати. Более ранняя (и менее эффективная) версия этого алгоритма в сокращённом виде изложена в работе [[90]].

В дальнейшем предполагается сосредоточить усилия на решении задачи исследования коллективом автоматов 1) недетерминированных графов, 2) с использованием внешней памяти, 3) на подклассах графов, которые практически значимы и на которых можно разработать более быстрые алгоритмы.

ЛИТЕРАТУРА:

[1]. Burdonov I., Kossatchev A., Petrenko A., Cheng S., Wong H. Formal Specification and Verification of SOS Kernel. BNR/NORTEL Design Forum, June 1996.
[2]. Баранцев А.В., Бритвина Е.Н., Бурдонов И.Б., Косачев А.С., Гоманюк С.В., Демаков А.В., Иванов А.В., Максимов А.В., Петренко А.К., Сазанов Ю.Л., Сортов А.А., Стефанов В.П., Сумар Г.М. Архитектура системы генерации и пропуска тестов // Вопросы кибернетики, Москва 1998.
[3]. Bourdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications // Proceedings of Formal Method Congress, Toulouse, France, 1999, LNCS, № 1708, pp.608-621.
[4]. Бурдонов И.Б., Косачев А.С., Демаков А.В., Петренко А.К., Максимов А.В. Формальные спецификации в технологиях обратной инженерии и верификации программ // Труды ИСП РАН, № 1, 1999, стр. 31-43.
[5]. Bourdonov I.B., Kossatchev A.S., Petrenko A.K., Kuliamin V.V. UniTESK Test Suite Architecture // Proceedings of FME'2002 conference, Copenhagen, Denmark, LNCS, № 2391, 2002, pp. 77-88.
[6]. Бурдонов И.Б., Косачев А.С., Кулямин В.В., Петренко А.К. Подход UniTESK к разработке тестов // «Программирование»,-2003, №6, стр.25-43.
[7]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K., Pakoulin N.V. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems // Perspectives of System Informatics // LNCS. № 2890, Springer-Verlag, 2003, pp.450-461.
[8]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTESK: Model Based Testing in Industrial Practice // Proceedings of 1-st European Conference on Model-Driven Software Engineering, Nurnberg, December 2003, pp. 55-63.
[9]. Бурдонов И.Б., Баранцев А.В., Демаков А.В., Зеленов С.В., Косачев А.С., Кулямин В.В., Омельченко В.А., Пакулин Н.В., Петренко А.К., Хорошилов А.В. Подход UniTESK к разработке тестов: достижения и перспективы // Труды ИСП РАН, № 5, 2004, стр.121-156.
[10]. Калинов А.Я., Косачев А.С., Посыпкин М.А,, Соколов А.А. Автоматическая генерация тестов для графического пользовательского интерфейса по *UML* диаграммам действий // Труды ИСП РАН, № 6, 2004, стр.7-84.
[11]. V.Kuliamin, A.K.Petrenko. Applying Model Based Testing in Different Contexts // Proceedings of Seminar on Perspectives on Model Based Testing, Dagstuhl, Germany, September 2004.
[12]. V.Kuliamin. Multi-paradigm Models as Source for Automated Test Construction // Proceedings of Workshop on Model Based Testing, Barcelona, Spain, March 2004. Electronic Notes in Theoretical Computer Science 111:137-160, 2005, Elseveir.
[13]. А.А.Сортов, А.В.Хорошилов. Функциональное тестирование Web-приложений на основе технологии UniTESK. //Труды ИСП РАН, №8, 2004, стр.77-97.
[14]. V.Kuliamin. Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System. //Proc of ISOLA 2004, Cyprus, October 2004, pp. 311-316.
[15]. В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.К. Петренко. Применение технологии UniTESK для функционального тестирования моделей аппаратного обеспечения // Препринт ИСП РАН, 2005.
[16]. V.Kuliamin, A.Petrenko, N.Pakoulin. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. Proc. of 9-th WMSCI, Orlando, USA, July 2005, v. VII. Model Based Development and Testing, pp. 65-70.

[17]. V.Kuliamin, A.Petrenko, N.Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proc. of 2-nd ISAS 2005, Berlin, Germany, April 2005, pp. 60-73 M. Malek, E. Nett, N. Suri, eds. Service Availability. LNCS 3694, pp. 68-83, Springer-Verlag, 2005.
[18]. С.Грошев. Применение технологии UniTESK для тестирования систем с различной конфигурацией активных потоков управления. //Труды ИСП РАН, №9, 2006, стр. 67-81.
[19]. А.В.Хорошилов. Спецификация и тестирование систем с асинхронным интерфейсом. //Препринт ИСП РАН, №12, 2006.
[20]. А.И.Гриневич, В.В.Кулямин, Д.А.Марковцев, А.К.Петренко, В.В.Рубанов, А.В.Хорошилов. Использование формальных методов для обеспечения соблюдения программных стандартов.// Труды ИСП РАН, №10, 2006.
[21]. В.С. Мутилин. Паттерны проектирования тестовых сценариев. //Труды ИСП РАН, №9, 2006.
[22]. 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, June 2006.
[23]. Н.В.Пакулин, А.В.Хорошилов. Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов. Программирование, №6, 2007.
[24]. В.П.Иванников, А.С.Камкин, А.С.Косачев, В.В.Кулямин, А.К.Петренко. Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры.// Программирование, №5, 2007, стр. 47-61.
[25]. Kamkin. Coverage-Directed Verification of Microprocessor Units Based on Contract Specifications. EWDTS 2008, pp. 84-87.
[26]. А. Камкин. Генерация тестовых программ для микропроцессоров. // Труды ИСП РАН, 2008.
[27]. А.К. Петренко. Унификация в автоматизации тестирования. Позиция UniTESK // Препринт ИСП РАН, т. 14, ч. 1, 2008, стр. 7-22.
[28]. A.Petrenko. Formal Methods and Innovation Economy: Facing New Challenges. //Proceedings of the 6th IEEE International Conference on Software Enginering and Formal Methods, Cape Town, South Africa, 10-14 November 2008.
[29]. S.G.Groshev. Bug localization by constructing reduced traces.// Programming and Computer Software, Volume 35, Number 3, pp. 145-157, may 2009.
[30]. S. Frenkel, A.Kamkin. Verification Methodology Based on Algorithmic State Machines and Cycle-Accurate Contract Specifications. // East-West Design & Test Symposium, September 18-21, 2009, pp. 39-42.
[31]. В. В. Кулямин. Интеграция методов верификации программных систем. // Программирование, 35(4):41-55, 2009.
[32]. В. В. Кулямин. Перспективы интеграции методов верификации программного обеспечения. // Труды ИСП РАН, 16:73-88, 2009.
[33]. M. Chupilko. Constructing Test Sequences for Hardware Designs with Parallel Starting Operations Using Implicit FSM Models. // EWDTS-2009 (East-West Design and Test Symposium 2009). Proceedings of IEEE East-West Design & Test Symposium (EWDTS 2009), 393-396 pp.
[34]. M. Chupilko, A. Kamkin. Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs. // NorChip-2009, pp.1-4.

[35]. Я.С. Губенко, А.С. Камкин, М.М. Чупилко. Сравнительный анализ современных технологий разработки тестов для моделей аппаратного обеспечения. // Труды ИСП РАН 2009, том17, стр.133-143.
[36]. С.А. Смолов. Разработка тестового набора для функционального тестирования инфраструктурного программного обеспечения Грид с применением технологии UniTESK. // Сборник научных трудов научно-практической конференции «Актуальные проблемы программной инженерии», 2009, стр.183-189.
[37]. Н.В.Пакулин, А.Н.Тугаенко. Разработка тестовых наборов для тестирования соответствия почтовых протоколов. // Конференция АППИ-2009, стр. 154-160.
[38]. В.В. Кулямин. Архитектура среды тестирования на основе моделей, построенная на базе компонентных технологий. // Труды ИСП РАН, N 18, 2010, стр. 9-44.
[39]. С.Г. Грошев. Технология создания гетерогенных трасс, их анализа и генерации из них отчётов. // Труды ИСП РАН, N 18, 2010, стр. 45-66.
[40]. М.М. Чупилко. Автоматизация системного тестирования моделей аппаратуры на основе формальных спецификаций. // Труды ИСП РАН, N 18, 2010, стр. 115-128.
[41]. А.В. Никешин, Н.В. Пакулин, В.З. Шнитман. Разработка тестового набора для верификации реализаций протокола безопасности IPsec v2. // Труды ИСП РАН, № 18, 2010, стр. 151-182.
[42]. M. Chupilko. Models of Synchronous Hardware Designs Based on FSM at Different Abstraction Levels: Application to Functional Verification // EWDTS-2010 (East-West Design and Test Symposium 2010). Proceedings of IEEE East-West Design & Test Symposium (EWDTS 2010), 127-130 pp.
[43]. M. Chupilko, A. Kamkin. Developing cycle-accurate contract specifications for synchronous parallel-pipeline hardware: application to verification. // BEC-2010 (Baltic Electronics Conference 2010). Proceedings of the 12th Biennial Baltic Electronics Conference (BEC 2010), 185-188 pp.
[44]. N. Pakulin, A. Tugaenko. Specification Based Conformance Testing for Email Protocols. // Proceedings of ISoLA 2010, pp.371-382. Heraclion, Greece, 2010.
[45]. В. В. Кулямин. Компонентная архитектура среды для тестирования на основе моделей. // Программирование, 36(5):54-75, 2010.
[46]. Н.В. Пакулин, А.Н. Тугаенко. Тестирование протоколов электронной почты Интернета с использованием моделей. // Труды ИСП РАН, том 20, 2011 г.
[47]. Kamkin. Simulation-Based Verification with Time-Abstract Models. EWDTS, 2011.
[48]. M. Chupilko, A. Kamkin. A TLM-based approach to functional verification of hardware components at different abstraction levels. LATW, 2011.
[49]. А. Камкин, М. Чупилко. Механизмы поддержки функционального тестирования моделей аппаратуры на разных уровнях абстракции, Труды ИСП РАН, 2011.
[50]. M.Chupilko. C++TESK-SystemVerilog united approach to simulation-based verification of hardware designs. EWDTS-2011.
[51]. V. Kuliamin, N. Pakulin, A. Tugaenko. Case Studies of Summer Model-based Testing Framework, Model-based Testing User Conference, October 18-20, 2011.
[52]. В.П. Иванников, А.К. Петренко. Модели в разработке и анализе программных систем. Ломоносовские чтения, 2011.
[53]. Y. Gerlits, A. Khoroshilov. «Model-Based Testing of Safety Critical Real-Time Control Logic Software» // In Proceedings of the Seventh Workshop on Model-Based Testing (MBT 2012), Tallin, Estonia, March 25, 2012.
[54]. N. Pakulin. Integrated Modular Avionics: New Challenges for MBTB. // ETSI TTCN-3 User Conference and Model Based Testing Workshop, Bangalore, India, 11-14 June 2012.

[55]. Шнитман В.З., Никешин А.В., Пакулин Н.В. Применение моделей для тестирования протоколов безопасности. // Всероссийская конференция «Инфокоммуникационные технологии в научных исследованиях», ИКИ РАН, 14-16 ноября 2012 г.
[56]. Чупилко М.М. Разработка тестовых систем для многомодульных моделей аппаратуры. Программирование, 2012, №1, с.47-58 http://www.UniTESK.com
[57]. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Использование конечных автоматов для тестирования программ. «Программирование». 2000. № 2.
[58]. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. «Программирование». 2003. № 5.
[59]. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Асинхронные автоматы: классификация и тестирование. // Труды Института системного программирования РАН, №4, 2003, стр.7-84.
[60]. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. «Программирование». 2004. № 1.
[61]. И.Б.Бурдонов. Проблема отката по дереву при обходе неизвестного ориентированного графа конечным роботом. Программирование, №6, 2004.
[62]. И.Б.Бурдонов. Обход неизвестного ориентированного графа конечным роботом. Программирование, №4, 2004.
[63]. I.B.Bourdonov, A.S.Kossatchev, V.V.Kuliamin. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proceedings of the Workshop on Model Based Testing (MBT 2004), Elsevier, 2006.
[64]. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Формализация тестового эксперимента.// Программирование, №5, 2007, стр. 3-32.
[65]. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Безопасность, верификация и теория конформности. «Материалы второй международной научной конференции по проблемам безопасности и противодействия терроризму. МГУ 2006», М., МЦНМО, 2007, стр. 135-158.
[66]. И.Б.Бурдонов, Косачев А.С., В.В.Кулямин. Теория соответствия для систем с блокировками и разрушением. // «Физ-мат лит» Наука, Москва, 2008. 412 с.
[67]. И.Б.Бурдонов, Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. // Труды ИСП РАН, N 14.1, 2008, стр.23-54.
[68]. И.Б.Бурдонов, Косачев А.С. Обобщённые семантики тестового взаимодействии. // Труды ИСП РАН, N 15, 2008, стр.69-106.
[69]. И.Б.Бурдонов, Косачев А.С. Эквивалентные семантики взаимодействия. // Труды ИСП РАН, N 14.1, 2008, стр.55-72.
[70]. И.Б.Бурдонов. Теория конформности (функциональное тестирование программных систем на основе формальных моделей). LAP Lambert Academic Publishing, 2011, 428 стр. (докторская диссертация И.Б.Бурдонова 2008 г.)
[71]. И.Б. Бурдонов, А.С. Косачев. Системы с приоритетами: конформность, тестирование, композиция. // «Программирование», -2009, №4–стр. 24-40.
[72]. И.Б. Бурдонов, А.С. Косачев. Полное тестирование с открытым состоянием ограниченно недетерминированных систем. // Труды ИСП РАН, N 17, 2009, стр.161-192.
[73]. И.Б. Бурдонов, А.С. Косачев. Полное тестирование с открытым состоянием ограниченно недетерминированных систем. // «Программирование», -2009, №6–стр 3-18.

[74]. И.Б. Бурдонов, А.С. Косачев. Тестирование с преобразованием семантик. // Труды ИСП РАН, N 17, 2009, стр.193-208.
[75]. И.Б. Бурдонов, А.С. Косачев. Тестирование конформности на основе соответствия состояний. // Труды ИСП РАН, N 18, 2010, стр. 183-220.
[76]. Kossachev, I.Burdonov. Formal Conformance Verifcation, Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil, pp.1-6.
[77]. А.С. Косачев, И.Б.Бурдонов. Семантики взаимодействия с отказами, дивергенцией и разрушением. // «Программирование», -2010, №5–стр 3-23.
[78]. И.Б.Бурдонов, А.С.Косачев, Семантики взаимодействия с отказами, дивергенцией и разрушением. Часть 1. Гипотеза о безопасности и безопасная конформность, «Вестник Томского государственного университета. Управление, вычислительная техника и информатика», №4, 2010, стр.124-133
[79]. И.Б.Бурдонов, А.С.Косачев, Безопасное тестирование симуляции систем с отказами и разрушением, «Моделирование и анализ информационных систем», Том 17, Номер 4, 2010. стр. 27—40.
[80]. И.Б.Бурдонов, А.С.Косачев. Пополнение спецификации для ***ioco***. // Программирование, №1, 2011, с. 3-18.
[81]. Бурдонов И.Б., Грошев С.Г., Демаков А.В., Камкин А.С., Косачев А.С., Сортов А.А., Параллельное тестирование больших автоматных моделей, Вестник ННГУ, №3, 2011 г., стр. 187-193
[82]. Demakov, A. Kamkin, A. Sortov. High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration. Open Cirrus, 2011.
[83]. И.Б.Бурдонов, А.С.Косачев. Удаление из спецификации неконформных трасс // Препринт ИСП РАН, Препринт 23, 2011 г, с. 1-219.
[84]. I.B.Bourdonov, A.S.Kossatchev. Safe simulation testing of systems with refusals and destructions // Automatic Control and Computer Sciences, Vol. 45, №7, 2011, pp. 380-389.
[85]. И.Б.Бурдонов, А.С.Косачев. Семантики взаимодействия с отказами, дивергенцией и разрушением. Часть 2. Условия конечного полного тестирования // Вестник Томского Государственного Университета, № 2(15), 2011.
[86]. И.Бурдонов, А.Косачев, Финальные модели спецификации, Труды ИСП РАН, том 22, 2012 г., ISSN 2079-8156, с. 233-276
[87]. И.Бурдонов, А.Косачев, Зависимости между ошибками на классах тестируемых реализаций, Труды ИСП РАН, том 23, 2012 г., ISSN 2079-8156
[88]. И.Б.Бурдонов, А.С.Косачев. Формализация тестового эксперимента –II. // Программирование, №4, 2013, с. 3-27.
[89]. И.Б.Бурдонов, А.С.Косачев. Согласование конформности и композиции // Программирование, №6, 2013, с. 3-15.
[90]. И.Бурдонов, А.Косачев, Обход неизвестного графа коллективом автоматов, Труды Международной суперкомпьютерной конференции «Научный сервис в сети Интернет: все грани параллелизма», 2013, изд. МГУ, стр. 228-232.
[91]. Milner R. Modal characterization of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.
[92]. van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.

[93]. Vaandrager F. On the relationship between process algebra and Input/Output Automata. In Logic in Computer Science, pp. 387-398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.
[94]. 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.
[95]. De Nicola R., Segala R. A process algebraic view of Input/Output Automata. Theoretical Computer Science, 138:391-423, 1995.
[96]. D. Lee and M. Yannakakis. Principles and Methods of Testing Finite-State Machines. A survey. Proceedings of the IEEE, Vol. 84, № 8, 1996, pp. 1090–1123.
[97]. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.
[98]. Heerink L. Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
[99]. Marine Tabourier, Ana Cavalli and Melania Ionescu , A GSM-MAP Protocol Experiment Using Passive Testing, Proceeding of FM'99 (World Congress on Formal methods in development of Computing Systems), Toulouse (France), 20-24 September 1999.
[100]. Jard C., Jéron T., Tanguy L., Viho C. Remote testing can be as powerful as local testing. In Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China, J. Wu, S. Chanson, Q. Gao (eds.), pp. 25-40, October 1999.
[101]. van der Bijl M., Rensink A., Tretmans J. Compositional testing with ioco. Formal Approaches to Software Testing: Third International Workshop, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Editors: Alexandre Petrenko, Andreas Ulrich ISBN: 3-540-20894-1. LNCS volume 2931, Springer, pp. 86-100.
[102]. van der Bijl M., Rensink A., Tretmans J. Component Based Testing with ioco. CTIT Technical Report TR-CTIT-03-34, University of Twente, 2003.
[103]. Alexandre Petrenko, Nina Yevtushenko: Testing from Partial Deterministic FSM Specifications. IEEE Trans. Computers 54(9): 1154-1165 (2005).
[104]. Adenilso da Silva Simão, Alexandre Petrenko, Nina Yevtushenko: Generating Reduced Tests for FSMs with Extra States. TestCom/FATES 2009: 129-145.
[105]. Maurice P. Herllihy, Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects //ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, pp. 463-492.

Conformance theory development: semantics, formal models, algorithms

Igor Burdonov, Alexander Kossatchev

Abstract. The paper covers theoretical and practical works on conformance testing performed in ISP RAS since 1994 till now. The conformance theory development was done in various directions and, in the whole, was characterized by generalization of the interaction semantics, models and conformances in use. The necessity of such generalization was imposed, first of all, by requirements of testing practice. It is true for such system properties as nondeterminism, partial specified, asynchronous behavior, diversity of test stimuli and observations of the implementation behavior etc. It was always focused on testing effectiveness defined both by optimization of tests suites and by test generation algorithms including on-the-fly. We consider the main milestones on this way in a brief and informal discussion, paying attention not to details, but to the main problems and their solutions trying to reveal the common tendency of the development.

Keywords: Interaction semantics, finite state automata, LTS, IOLTS, traces, conformance, priorities, simulation, reduction, test generation, testing optimization, specification completion, system composition, mediators, modeling, implementation, specification, safe testing, graph learning, graph traversal, parallel testing

References

[1]. Bourdonov I., Kossatchev A., Petrenko A., Cheng S., Wong H. Formal Specification and Verification of SOS Kernel. BNR/NORTEL Design Forum, June 1996.

[2]. Barantsev A.V., Britvina E.N., Bourdonov I, Kossatchev A., Gomanyuk S.V., Demakov A.V., Ivanov A.V., Maksimov A.V., Petrenko A.K., Sazanov YU.L., Sortov A.A., Stefanov V.P., Sumar G.M. Arkhitektura sistemy generatsii i propuska testov [Architecture of the system for test generation and running]. Voprosy kibernetiki, Moscow 1998. (in Russian).

[3]. Bourdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications Proceedings of Formal Method Congress, Toulouse, France, 1999, LNCS, № 1708, pp.608-621.

[4]. Bourdonov I.B., Kossatchev A.S., Demakov A.V., Petrenko A.K., Maksimov A.V. Formal'nye spetsifikatsii v tekhnologiyakh obratnoj inzhenerii i verifikatsii programm [Formal specifications for the technology of reverse engineering fnd program verification] Trudy ISP RAN [The proceeding of ISP RAS], Vol. 1, 1999, pp. 31-43. (in Russian).

[5]. Bourdonov I.B., Kossatchev A.S., Petrenko A.K., Kuliamin V.V. UniTESK Test Suite Architecture Proceedings of FME'2002 conference, Copenhagen, Denmark, LNCS, № 2391, 2002, pp. 77-88.

[6]. Bourdonov I.B., Kossatchev A.S., Petrenko A.K., Kuliamin V.V. The UniTesK Approach to Designing Test Suites. Programming and Computer Software, Vol. 29, No. 6, 2003, pp. 310-322.

[7]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K., Pakoulin N.V. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. Perspectives of System Informatics. LNCS. № 2890, Springer-Verlag, 2003, pp.450-461.
[8]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTESK: Model Based Testing in Industrial Practice. Proceedings of 1-st European Conference on Model-Driven Software Engineering, Nurnberg, December 2003, pp. 55-63.
[9]. Bourdonov I.B., Barantsev A.V., Demakov A.V., Zelenov S.V., Kossatchev A.S., Kuliamin V.V., Omel'chenko V.A., Pakulin N.V., Petrenko A.K., KHoroshilov A.V. Podkhod UniTESK k razrabotke testov: dostizheniya i perspektivy [UniTESK approach to test development: achievements and perspectives]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 5, 2004, pp. 121-156. (in Russian)
[10]. Kalinov A. Ya., Kossatchev A.S., Posypkin M.A,, Sokolov A.A. Avtomaticheskaya generatsiya testov dlya graficheskogo pol'zovatel'skogo interfejsa po UML diagrammam dejstvij [Automated test generation for user graphical interface by UML action diagrams]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 6, 2004, pp. 7-84. (in Russian)
[11]. Kuliamin V.V., Petrenko A.K.. Applying Model Based Testing in Different Contexts Proceedings of Seminar on Perspectives on Model Based Testing, Dagstuhl, Germany, September 2004.
[12]. Kuliamin V.V. Multi-paradigm Models as Source for Automated Test Construction. Proceedings of Workshop on Model Based Testing, Barcelona, Spain, March 2004. Electronic Notes in Theoretical Computer Science 111:137-160, 2005, Elseveir.
[13]. Sortov A.A., KHoroshilov A.V.. Funktsional'noe testirovanie Web-prilozhenij na osnove tekhnologii UniTESK [Functional testing of WEB applications on UniTESK technology base]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 8, 2004, pp. 77-97. (in Russian)
[14]. Kuliamin V.. Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System. Proc of ISOLA 2004, Cyprus, October 2004, pp. 311-316.
[15]. Ivannikov V.P., Kamkin A.S., Kuliamin V.V., Petrenko A.K. Primenenie tekhnologii UniTESK dlya funktsional'nogo testirovaniya modelej apparatnogo obespecheniya [UniTESK technology application for functional testing of hardware model] Preprint № 8, ISP RAN [Preprints of the Institute for System Programming of RAS, Preprint 8], 2005. (in Russian)
[16]. Kuliamin V., Petrenko A., Pakoulin N.. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. Proc. of 9-th WMSCI, Orlando, USA, July 2005, v. VII. Model Based Development and Testing, pp. 65-70.
[17]. Kuliamin V., Petrenko A., Pakoulin N.. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proc. of 2-nd ISAS 2005, Berlin, Germany, April 2005, pp. 60-73 M. Malek, E. Nett, N. Suri, eds. Service Availability. LNCS 3694, pp. 68-83, Springer-Verlag, 2005.
[18]. Groshev S. Primenenie tekhnologii UniTESK dlya testirovaniya sistem s razlichnoj konfiguratsiej aktivnykh potokov upravleniya [UniTESK technology application for testing of the systems with various control flow configuration]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 9, 2006, pp. 67-81. (in Russian)
[19]. KHoroshilov A.V. Spetsifikatsiya i testirovanie sistem s asinkhronnym interfejsom [Specification and testing of the systems with asynchronous interfaces]. Preprint № 12,

ISP RAN [Preprints of the Institute for System Programming of RAS, Preprint 12], 2006. (in Russian)
[20]. Grinevich A.I., Kulyamin V.V., Markovtsev D.A., Petrenko A.K., Rubanov V.V., KHoroshilov A.V. Ispol'zovanie formal'nykh metodov dlya obespecheniya soblyudeniya programmnykh standartov [Formal method usage for program standards compliance support]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 10, 2006, pp. 51-68. (in Russian)
[21]. Mutilin V.S. Patterny proektirovaniya testovykh stsenariev [Project patterns of test scenarios]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 9, 2006, pp. 97-128. (in Russian)
[22]. Grinevich A., Khoroshilov A., Kuliamin V., Markovtsev D., Petrenko A., V.Rubanov. Formal Methods in Industrial Software Standards Enforcement. Proc. of PSI'2006, Novosibirsk, Russia, June 2006.
[23]. Pakulin N.V., KHoroshilov A.V. Development of formal models and conformance testing for systems with asynchronous interfaces and telecommunications protocols. Programming and Computer Software, Vol. 33, No. 6, 2007,pp. 316-335.
[24]. Ivannikov V.P., Kamkin A.S., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. The use of contract specifications for representing requirements and for functional testing of hardware models. Programming and Computer Software, Vol. 33, No. 5, 2007,pp. 272-282.
[25]. Kamkin A. Coverage-Directed Verification of Microprocessor Units Based on Contract Specifications. EWDTS 2008, pp. 84-87.
[26]. Kamkin A.. Generatsiya testovykh programm dlya mikroprotsessorov [Test program generation fo microprocessors]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 14.2, 2008, pp. 23-64. (in Russian)
[27]. Petrenko A.K. Unifikatsiya v avtomatizatsii testirovaniya. Pozitsiya UniTESK [Unification of test automatization. UniTESK position]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 14.1, 2008, pp. 7-22. (in Russian)
[28]. Petrenko A.. Formal Methods and Innovation Economy: Facing New Challenges. Proceedings of the 6th IEEE International Conference on Software Enginering and Formal Methods, Cape Town, South Africa, 10-14 November 2008.
[29]. S.G.Groshev. Bug localization by constructing reduced traces. Programming and Computer Software, Vol. 35, No 3, 2009, pp. 145-157.
[30]. Frenkel S., Kamkin A.. Verification Methodology Based on Algorithmic State Machines and Cycle-Accurate Contract Specifications. East-West Design & Test Symposium, September 18-21, 2009, pp. 39-42.
[31]. Kuliamin V.V. Integration of verification methods for program systems. Programming and Computer Software, Vol. 35, No 4, 2009, pp. 212-222.
[32]. Kuliamin V.V. Perspektivy integratsii metodov verifikatsii programmnogo obespecheniya [The perspective of software verification method integration]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 16, 2009, pp. 73-88. (in Russian)
[33]. Chupilko M.. Constructing Test Sequences for Hardware Designs with Parallel Starting Operations Using Implicit FSM Models. EWDTS-2009 (East-West Design and Test Symposium 2009). Proceedings of IEEE East-West Design & Test Symposium (EWDTS 2009), 393-396 pp.
[34]. Chupilko M., Kamkin A. Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs. NorChip-2009, pp.1-4.
[35]. Gubenko Ya.S., Kamkin A.S., Chupilko M.M. Sravnitel'nyj analiz sovremennykh tekhnologij razrabotki testov dlya modelej apparatnogo obespecheniya [Comparative

analysis of contemporary test development technologies for hardware models]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 17, 2009, pp. 133-143. (in Russian)
[36]. Smolov S.A. Razrabotka testovogo nabora dlya funktsional'nogo testirovaniya infrastrukturnogo programmnogo obespecheniya GRID s primeneniem tekhnologii UniTESK [Test case development for functional testing of GRID infrastructure software by UniTESK technology]. Sbornik nauchnykh trudov nauchno-prakticheskoj konferentsii «Aktual'nye problemy programmnoj inzhenerii» [The proceeding of conference "Actual problems of program engineering"]. 2009, pp.183-189. (in Russian)
[37]. Pakulin N.V., Tugaenko A.N. Razrabotka testovykh naborov dlya testirovaniya sootvetstviya pochtovykh protokolov [Test case development for mail protocol conformance testing]. Konferentsiya APPI-2009 [APPI-2009 conference], pp. 154-160. (in Russian)
[38]. Kuliamin V.V. Arkhitektura sredy testirovaniya na osnove modelej, postroennaya na baze komponentnykh tekhnologij [Model Based Testing Framework Using Component Technologies]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 18, 2010, pp. 9-44. (in Russian)
[39]. Groshev S.G. Tekhnologiya sozdaniya geterogennykh trass, ikh analiza i generatsii iz nikh otchyotov [Technology of heterogeneous logging, analysis and report generation]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 18, 2010, pp. 45-66. (in Russian)
[40]. Chupilko M.M. Avtomatizatsiya sistemnogo testirovaniya modelej apparatury na osnove formal'nykh spetsifikatsij [Formal specifications-based automation of system testing of hardware designs]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 18, 2010, pp. 115-12897. (in Russian)
[41]. Nikeshin A.V., Pakulin N.V., Shnitman V.Z. Razrabotka testovogo nabora dlya verifikatsii realizatsij protokola bezopasnosti IPsec v2 [Conformance test suite for implementations of the security protocol suite IPsec v2]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 18, 2010, pp. 151-182. (in Russian)
[42]. Chupilko M. Models of Synchronous Hardware Designs Based on FSM at Different Abstraction Levels: Application to Functional Verification EWDTS-2010 (East-West Design and Test Symposium 2010). Proceedings of IEEE East-West Design & Test Symposium (EWDTS 2010), 127-130 pp.
[43]. Chupilko M., Kamkin A.. Developing cycle-accurate contract specifications for synchronous parallel-pipeline hardware: application to verification. BEC-2010 (Baltic Electronics Conference 2010). Proceedings of the 12th Biennial Baltic Electronics Conference (BEC 2010), 185-188 pp.
[44]. Pakulin N., Tugaenko A. Specification Based Conformance Testing for Email Protocols. Proceedings of ISoLA 2010, pp.371-382. Heraclion, Greece, 2010.
[45]. Kuliamin V.V. Component architecture of model-based testing environment. Programming and Computer Software, Vol. 36, No 5, 2010, pp. 289-305.
[46]. Pakulin N.V, Tugaenko A. Testirovanie protokolov ehlektronnoj pochty Interneta s ispol'zovaniem modelej [Model-based testing of Internet Mail Protocols]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 20, 2011, pp. 125-141. (in Russian)
[47]. Kamkin A.. Simulation-Based Verification with Time-Abstract Models. EWDTS, 2011.
[48]. Chupilko M., Kamkin A. A TLM-based approach to functional verification of hardware components at different abstraction levels. LATW, 2011.
[49]. Kamkin A., Chupilko M. Mekhanizmy podderzhki funktsional'nogo testirovaniya modelej apparatury na raznykh urovnyakh abstraktsii [Mechanisms for functional testing of hardware models at different levels of abstraction], Trudy ISP RAN [The proceeding of ISP RAS], Vol. 20, 2011, pp. 143-160. (in Russian)

[50]. Chupilko M. C++TESK-SystemVerilog united approach to simulation-based verification of hardware designs. EWDTS-2011.
[51]. Kuliamin V., Pakulin N., Tugaenko A.. Case Studies of Summer Model-based Testing Framework, Model-based Testing User Conference, October 18-20, 2011.
[52]. Ivannikov V.P., Petrenko A.K. Modeli v razrabotke i analize programmnykh system [Models for program system development and verification]. Lomonosovskie chteniya [Lomonosov MSU conference], MSU, 2011. (in Russian)
[53]. Y. Gerlits, A. Khoroshilov. «Model-Based Testing of Safety Critical Real-Time Control Logic Software» In Proceedings of the Seventh Workshop on Model-Based Testing (MBT 2012), Tallin, Estonia, March 25, 2012.
[54]. Pakulin N. Integrated Modular Avionics: New Challenges for MBT. ETSI TTCN-3 User Conference and Model Based Testing Workshop, Bangalore, India, 11-14 June 2012.
[55]. Shnitman V.Z., Nikeshin A.V. Pakulin N.V. Primenenie modelej dlya testirovaniya protokolov bezopasnosti [Model usage for security protocol testing]. Vserossijskaya konferentsiya «Infokommunikatsionnye tekhnologii v nauchnykh issledovaniyakh» [Russian conference "Infocommunication technologies for scientific research", IKI RAN [Space Research Institute RAS], 2012. (in Russian)
[56]. Chupilko M.M. Developing test systems for multi-modules hardware designs. Programming and Computer Software, Vol. 38, No 1, 2012, pp. 34-42. http: www.UniTESK.com
[57]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Application of Finite Automatons for Program Testing. Programming and Computer Software, Vol. 26, No. 2, 2000, pp. 61-73.
[58]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, Vol. 29, No. 5, 2003, pp. 245-258.
[59]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Asinkhronnye avtomaty: klassifikatsiya i testirovanie [Asynchronous automata: classification and testing]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 4, 2003, pp. 7-84. (in Russian)
[60]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case. Programming and Computer Software, Vol. 30, No. 1, 2004, pp. 2-17.
[61]. Bourdonov I.B. Backtracking Problem in the Traversal of an Unknown Directed Graph by a Finite Robot. Programming and Computer Software, Vol. 30, No. 6, 2004, pp. 305-322.
[62]. Bourdonov I.B. Traversal of an Unknown Directed Graph by a Finite Robot. Programming and Computer Software, Vol. 30, No. 4, 2004, pp. 188-203.
[63]. I.B.Bourdonov, A.S.Kossatchev, Kuliamin V.V. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proceedings of the Workshop on Model Based Testing (MBT 2004), Elsevier, 2006.
[64]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Formalization of Test Experiments. Programming and Computer Software, Vol. 33, No. 5, 2007, pp. 239-260.
[65]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Bezopasnost', verifikatsiya i teoriya konformnosti [Safety, Verification and Conformance Theory]. Materialy Vtoroj mezhdunarodnoj nauchnoj konferentsii po problemam bezopasnosti i protivodejstviya terrorizmu [The proceeding of the Second international conference on the problems of safety and counteraction against terrorizm], Moscow, MNCMO, 2007, pp. 135-158. (in Russian)

[66]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Teoriya sootvetstviya dlya system s blokirovkami I razrusheniem [Conformance theory of the systems with Refused Inputs and Forbidden Actions]. Moscow, «Nauka», 2008, 412 p. (in Russian)
[67]. Bourdonov I.B., Kossatchev A.S. Sistemy s prioritetami: konformnost', testirovanie, kompozitsiya [Systems with priority: conformance, testing, composition]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 14.1, 2008, pp.23-54. (in Russian)
[68]. Bourdonov I.B., Kossatchev A.S. Obobshhyonnye semantiki testovogo vzaimodejstviya [Generic test interaction semantics]. Trudy ISP RAN [The proceeding of ISP RAS], v. 15, 2008, pp. 69-106. (in Russian)
[69]. Bourdonov I.B., Kossatchev A.S. Ehkvivalentnye semantiki vzaimodejstviya [Equivalent interaction semantics]. Trudy ISP RAN [The proceeding of ISP RAS], v. 14.1, 2008, pp.55-72. (in Russian)
[70]. Bourdonov I. Teoriya konformnosti (funkcional'noe testirovanie prorammny'kh system na osnove formal'ny'kh modelej [Conformance theory (functional testing on formal model base)]. LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9, 428 p. http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf) (in Russian)
[71]. Bourdonov I.B., Kossatchev A.S. Systems with Priorities: Conformance, Testing, and Composition. Programming and Computer Software, Vol. 35, No. 4, 2009, pp.198-211.
[72]. Bourdonov I.B., Kossatchev A.S. Polnoe testirovanie s otkrytym sostoyaniem ogranichenno nedeterminirovannykh system [Complete OpenState Testing of Limitedly Nondeterministic Systems]. Trudy ISP RAN [The proceeding of ISP RAS], v. 17, 2009, pp. 161-192. (in Russian)
[73]. Bourdonov I.B., Kossatchev A.S. Complete OpenState Testing of Limitedly Nondeterministic Systems. Programming and Computer Software, Vol. 35, No. 6, 2009,pp.301-313.
[74]. Bourdonov I.B., Kossatchev A.S. Testirovanie s preobrazovaniem semantic [Testing with Semantics Conversion] Trudy ISP RAN [The proceeding of ISP RAS], Vol. 17, 2009, pp. 193-208. (in Russian)
[75]. Bourdonov I.B., Kossatchev A.S. Testirovanie konformnosti na osnove sootvetstviya sostoyanij [Conformance testing based on a state relation]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 18, 2010, pp. 183-320. (in Russian)
[76]. Kossachev A., Burdonov I.. Formal Conformance Verifcation, Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil, pp.1-6.
[77]. Bourdonov I.B., Kossatchev A.S. Interaction Semantics with Refusals, Divergence, and Destruction. Programming and Computer Software, Vol. 36, No. 5, 2010, pp. 247-263.
[78]. Bourdonov I.B., Kossatchev A.S. Semantiki vzaimodejstviya s otkazami, divergentsiej i razrusheniem. CHast' 1. Gipoteza o bezopasnosti i bezopasnaya konformnost'. [Semantics of Interaction with Refused Inputs, Divergence and Forbidden Actions] «Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika» [Tomsk State University. Journal of Control and Computer Science], №4, 2010, pp. 124-133. (in Russian)
[79]. Bourdonov I.B., Kossatchev A.S. Safe Simulation Testing of Systems with Refusals and Destructions Automatic Control and Computer Sciences, 2011, Vol. 45, No. 7.
[80]. Bourdonov I.B., Kossatchev A.S. Specification Completion for IOCO. Programming and Computer Software, Vol. 37, No. 1, 2011, pp. 1-14.
[81]. Bourdonov I.B., Groshev S.G., Demakov A.V., Kamkin A.S., Kossatchev A.S, Sortov A.A. Parallel'noe testirovanie bol'shikh avtomatnykh modelej [Parallel testing of large

automata models], Vestnik NNGU [Vestnik of UNN], №3920, 2011, pp. 187-193. (in Russian)
[82]. Demakov A., Kamkin A., Sortov A. High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration. Open Cirrus, 2011.
[83]. Bourdonov I.B., Kossatchev A.S. Udalenie iz spetsifikatsii nekonformnykh trass [Nonconforming traces elimination from specification]. Preprint № 23, ISP RAN [Preprints of the Institute for System Programming of RAS, Preprint 23], 2011, pp. 1-219.
[84]. Bourdonov I.B., Kossatchev A.S.. Safe simulation testing of systems with refusals and destructions Automatic Control and Computer Sciences, Vol. 45, №7, 2011, pp. 380-389.
[85]. Bourdonov I.B., Kossatchev A.S. Semantiki vzaimodejstviya s otkazami, divergentsiej i razrusheniem. CHast' 2. Usloviya konechnogo polnogo testirovaniya. [Semantics of Interaction with Refused Inputs, Divergence and Forbidden Actions. Part 2. The condition of finite complete testing]. «Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika» [Tomsk State University. Journal of Control and Computer Science]. №2, 2011, pp. 89-98. (in Russian).
[86]. Bourdonov I.B., Kossatchev A.S. Final'nye modeli spetsifikatsii [The final models of specification]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 22, 2012, pp. 233-276.
[87]. Bourdonov I.B., Kossatchev A.S., Zavisimosti mezhdu oshibkami na klassakh testiruemykh realizatsij [Error dependencies on classes of implementations under testing], Trudy ISP RAN [The proceeding of ISP RAS], Vol. 23, 2012, pp. 323-358. (in Russian)
[88]. Bourdonov I.B., Kossatchev A.S. Formalization of a Test Experiment – II. Programming and Computer Software, Vol. 39, No. 4, 2013, pp. 163-181.
[89]. Bourdonov I.B., Kossatchev A.S. Agreement between Conformance and Composition. Programming and Computer Software, Vol. 39, No. 6, 2013, pp. 269–278.
[90]. Bourdonov I.B., Kossatchev A.S. Obkhod neizvestnogo grafa kollektivom avtomatov [Unknown graph traversing by automata group]. Trudy Mezhdunarodnoj superkomp'yuternoj konferentsii “Nauchnyj servis v seti Internet: vse grani parallelizma ”(21-26 sentyabrya 2009 g., g. Novorossijsk) [The proceeding of Russian Supercomputer conference ‘Scientific service of Internet’ (2013, Novorossijsk)] – Moscow, MSU publ., 2013, pp. 228-232. (in Russian)
[91]. Milner R. Modal characterization of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.
[92]. van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR’90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.
[93]. Vaandrager F. On the relationship between process algebra and Input/Output Automata. In Logic in Computer Science, pp. 387-398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.
[94]. 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.
[95]. De Nicola R., Segala R. A process algebraic view of Input/Output Automata. Theoretical Computer Science, 138:391-423, 1995.

[96]. D. Lee and M. Yannakakis. Principles and Methods of Testing Finite-State Machines. A survey. Proceedings of the IEEE, Vol. 84, № 8, 1996, pp. 1090–1123.
[97]. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.
[98]. Heerink L. Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
[99]. Marine Tabourier, Ana Cavalli and Melania Ionescu , A GSM-MAP Protocol Experiment Using Passive Testing, Proceeding of FM'99 (World Congress on Formal methods in development of Computing Systems), Toulouse (France), 20-24 September 1999.
[100]. Jard C., Jéron T., Tanguy L., Viho C. Remote testing can be as powerful as local testing. In Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China, J. Wu, S. Chanson, Q. Gao (eds.), pp. 25-40, October 1999.
[101]. van der Bijl M., Rensink A., Tretmans J. Compositional testing with ioco. Formal Approaches to Software Testing: Third International Workshop, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Editors: Alexandre Petrenko, Andreas Ulrich ISBN: 3-540-20894-1. LNCS volume 2931, Springer, pp. 86-100.
[102]. van der Bijl M., Rensink A., Tretmans J. Component Based Testing with ioco. CTIT Technical Report TR-CTIT-03-34, University of Twente, 2003.
[103]. Alexandre Petrenko, Nina Yevtushenko: Testing from Partial Deterministic FSM Specifications. IEEE Trans. Computers 54(9): 1154-1165 (2005).
[104]. Adenilso da Silva Simão, Alexandre Petrenko, Nina Yevtushenko: Generating Reduced Tests for FSMs with Extra States. TestCom/FATES 2009: 129-145.
[105]. Maurice P. Herllihy, Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, pp. 463-492.

Тестирование операционных систем

Герлиц Е.А., Кулямин В.В., Максимов А.В., Петренко А.К.,
Хорошилов А.В., Цыварев А.В.
{gerlits, kuliamin, andrew, petrenko, khoroshilov, tsyvarev} @ispras.ru

Аннотация. Работа операционной системы лежит в основе функционирования любой компьютерной системы. Сбои и ошибки в операционной системе сказываются на работоспособности системы в целом, поэтому к корректности и надёжности операционных систем предъявляются повышенные требования. Верификация и тестирование операционных систем осложняется целым букетом разнообразных обстоятельств — это и зависимость от аппаратуры, и массированный внутренний параллелизм, и традиционное богатство конфигурационных настроек, и вопросы устойчивости к действиям злоумышленников и к сбоям аппаратуры, и продолжительность непрерывного функционирования. В статье рассматриваются все эти особенности, описываются подходы и инструменты тестирования, разработанные в Институте системного программирования РАН, и представляется опыт их применения для тестирования операционной системы Linux, а также ряда операционных систем реального времени.

Ключевые слова: операционная система; тестирование на основе моделей; тестирование производительности.

1. Введение

Операционные системы (ОС) решают две взаимодополняющие задачи:

- организуют процесс работы многих приложений на одной ЭВМ, управляя разделением ресурсов ЭВМ между приложениями, а также защищая приложения друг от друга;
- предоставляют набор функций с целью создания удобной среды для работы пользователя и прикладных программ.

Ключевым компонентом ОС является ядро (рис. 1), к которому, как правило, относят код, выполняющийся в привилегированном режиме работы процессора. Ядро управляет всеми аппаратными ресурсами, доступными ОС, предоставляя возможность настройки политик доступа к ресурсам и разделения ресурсов, а также не позволяя приложениям нарушать эти политики. Иногда в угоду интересам оптимизации в ядро включают функциональность, которая не требует привилегированного режима

процессора. Примером такого включения может служить фильтрация сетевых пакетов в ядре ОС Linux.

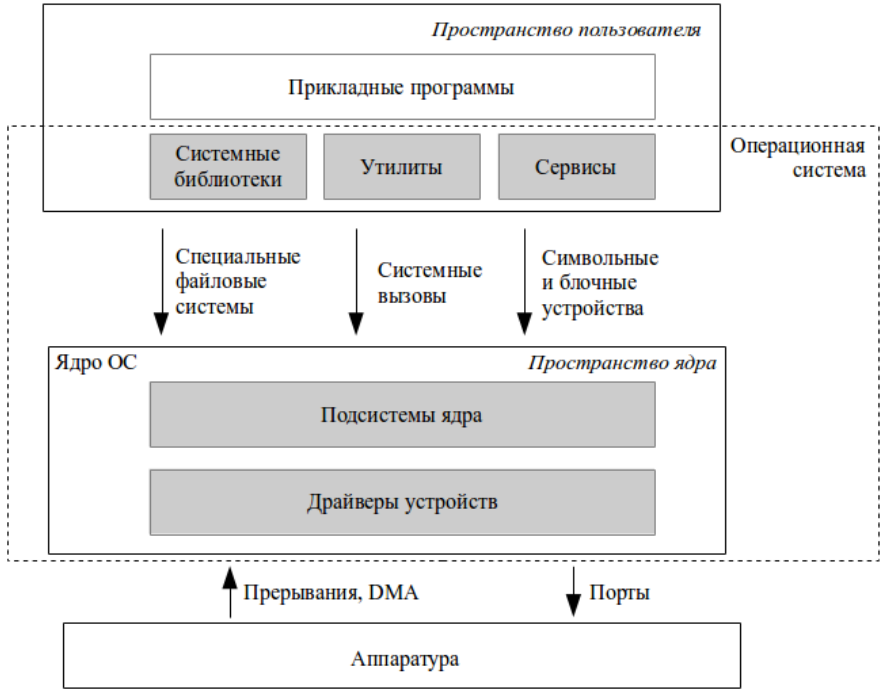


Рис. 1. Основные компоненты ОС.

Взаимодействие ядра ОС с пользовательскими приложениями обычно выполняется посредством системных вызовов, которые во многом схожи с обычным вызовом функции, но включают в себя переключение в привилегированный режим работы процессора и обратно. Часто также существуют и дополнительные механизмы взаимодействия с ядром, например, посредством чтения-записи файлов в специализированных файловых системах, таких как procfs, sysfs, debugfs в ОС Linux.

С целью создания удобной среды для работы прикладных программ ОС предоставляет системные библиотеки и утилиты, которые реализуют множество типовых функций и при необходимости обращаются к ядру ОС.

Ещё одним важным элементом ОС являются системные сервисы — активные компоненты, которые необходимы для реализации той или иной функциональности ОС. Например, для обслуживания определённых устройств, реализации сетевых протоколов, управления ресурсами и т. д.

Сервисы могут работать как внутри ядра ОС, так и в пользовательском пространстве.

Рассматривая ОС с точки зрения тестирования, необходимо учитывать следующие особенности.

- ОС как программное обеспечение, работающее непосредственно с аппаратурой, обладает:
  - значительным внутренним параллелизмом;
  - зависимостью от аппаратуры и её конфигураций;
  - внутренней активностью.
- ОС как основа компьютерной системы в целом и гарант безопасности приложений должна:
  - быть устойчивой к различным нестандартным ситуациям, таким как нехватка оперативной памяти, памяти жёсткого диска и т. д.;
  - быть устойчивой к атакам и вредоносным действиям со стороны недоверенных приложений, сетевых контрагентов, подключаемых внешних устройств и т. д.
  - не допускать утечек ресурсов с целью обеспечения продолжительного функционирования системы без перезагрузки;
  - минимизировать накладные расходы на реализацию своих функций.
- ОС как среда для работы прикладных программ должна обладать такими свойствами как:
  - соответствие стандартам на интерфейсы ОС;
  - соответствие документации на интерфейсы ОС;
  - совместимость с приложениями как на уровне бинарных интерфейсов, так и на уровне исходного кода.
- ОС как платформа, на которой работает тестовая система, является источником дополнительных требований к принципам построения тестовой системы:
  - при обнаружении ошибок, ведущих к аварийному прекращению работы ОС, информация, полученная тестом, должна сохраняться;
  - тестовая система должна вносить минимальные искажения в поведение тестируемых компонентов ОС, в том числе в их временны*е* характеристики.

Последующие разделы статьи организованы следующим образом. В разделе 2 рассматриваются вопросы функционального тестирования ОС, как тщательного, так и поверхностного. Раздел 3 посвящён тестированию обратной совместимости ОС, а раздел 4 — обнаружению специфических ошибок, таких как утечки ресурсов, гонки по данным и некорректная обработка некорректных входных данных. В разделе 5 рассматривается задача тестирования производительности ОС. В разделе 6 описаны сложности

конфигурационного тестирования ОС и необходимость комплексной системы управления тестированием ОС. В заключении подводятся итоги и формулируются направления дальнейшего развития.

2. Функциональное тестирование

Одной из ожидаемых характеристик ОС является корректная реализация своей функциональности. Это означает, что реальное поведение компонентов, доступных через публичные интерфейсы ОС, должно соответствовать декларируемым. Среди публичных интерфейсов ОС можно выделить интерфейсы системных библиотек и утилит, а также интерфейсы ядра ОС. Единичным элементом интерфейса библиотеки является функция, например, open(), sin() или fprintf(). Приведённые примеры демонстрируют различные виды реализации библиотечных функций:

- open() является примером функции-обёртки, которая сама по себе мало, что делает, за исключением трансляции запроса к компоненту более низкого уровня, в данном случае, open() обращается к соответствующему системному вызову ядра ОС;
- sin(), напротив, полностью реализуется в математической библиотеке и не требует обращения к другим компонентам;
- fprintf() представляет собой промежуточный вариант, в котором существенная часть функциональности реализуется в библиотеке (форматирование строки), но присутствует также и обращение к внешним компонентам (непосредственный вывод строки).

Таблица 1 демонстрирует количественные показатели состава интерфейсов в ОС. На одной стороне спектра фигурирует специализированная операционная система реального времени (ОСРВ), которая предназначена для работы на встраиваемых устройствах. Она включает поддержку порядка 700 функций и 80 команд, хотя при подготовке образа ОСРВ для конкретного применения в него включают только то подмножество функциональности, которое действительно необходимо для работы. В качестве другой стороны спектра представлена статистика ОС общего назначения Debian 7.0 для архитектуры x86, которая включает в себя более 10 тысяч утилит и более полутора тысяч библиотек, предоставляющих более 700 тысяч функций.

	Дата выпуска	Системные вызовы	Библиотеки	Функции	Утилиты
Debian 7.0	Май 2013	~350	~1650	~ 720 тыс.	10 тыс.
ОСРВ	Ноябрь 2013	~200	-	~700	~80

Табл. 1. Количественные характеристики размера интерфейса ОС.

Представленная статистика наглядно показывает, что задачу тщательного функционального тестирования можно ставить только для специализированных ОС и для ключевых интерфейсов ОС общего назначения. Для широкомасштабного функционального тестирования ОС общего назначения приходится использовать методы, в большей степени ориентированные не на достижение высокого качества тестирования, а на обеспечение приемлемых временны*х* и экономических показателей проектов.

2.1. Тщательное функциональное тестирование

Для тщательного функционального тестирования необходимым условием является наличие детальных функциональных требований к целевым интерфейсам, а также чёткое представление об архитектуре компонентов, реализующих целевую функциональность. При этом важно отметить, что целью тестирования является выявление дефектов не только в реализации ОС, но и в документах, описывающих функциональные требования и архитектуру.

2.1.1 Оценка качества тестирования

Поскольку исчерпывающее тестирование любой сколько-нибудь сложной системы не представляется возможным, то другим непременным атрибутом тщательного тестирования является формальная и всесторонняя оценка качества проводимого тестирования, которая позволяет оценить, насколько полно тестирование покрывает различные аспекты целевой функциональности. Для оценки качества тестирования традиционно используется два взаимодополняющих подхода:

- оценка покрытия функциональных требований;
- оценка покрытия структуры исходного кода реализации.

Для оценки покрытия требований необходимо, чтобы эти требования были структурированы, и минимальные элементы структуры представляли собой элементарные требования, то есть не подлежали дальнейшей декомпозиции. Кроме того, требования должны быть проверяемыми. Часто оказывается, что имеющиеся функциональные требования описаны в виде текста на естественном языке, допускают различные интерпретации и не обладают всеми нужными свойствами. В этом случае в процесс разработки тщательных функциональных тестов приходится включать дополнительный шаг — выделение и каталогизацию элементарных требований [1, 2]. Для более детального слежения за покрытием требований также бывает необходимо оценивать, в каких ситуациях каждое требование проверялось. Для каталогизации требований и отслеживания их покрытия в ИСП РАН был разработан специализированный инструмент управления требованиями Requality [3, 4], который позволяет построить каталог требований таким образом, чтобы каждое требование было связано с соответствующим фрагментом текста в исходном документе с функциональными требованиями.

Эта возможность оказывается наиболее востребована, когда исходные документы развиваются независимо от процесса тестирования и может потребоваться синхронизация каталога требований с новыми версиями исходных документов.

Измерение покрытия структуры исходного текста поддержано целым рядом инструментов. Основной вопрос, который возникает при применении этой метрики — это как очертить границы компонентов, за покрытием которых необходимо следить, и как относиться к непокрытым фрагментам кода. Для ответа на первый вопрос как раз и требуются знания об архитектуре компонентов, реализующих целевую функциональность. Второй вопрос подразумевает дополнительную активность, заключающуюся в анализе результатов измерения покрытия и принятии решений либо о доработке тестов, либо о внесении исправлений в исходный код, либо о признании отсутствия необходимости покрывать отдельные фрагменты.

Совместная оценка покрытия требований и покрытия исходного кода работают весьма эффективно, позволяя идентифицировать проблемы, которые могут остаться незамеченными при применении только одной из метрик, например, ввиду отсутствия в коде реализации одного из требований или отсутствия какого-то аспекта в требованиях.

Дополнительной метрикой, которой имеет смысл уделять внимание, является оценка покрытия возможных классов взаимодействия тестируемых компонентов с другими компонентами ОС [5].

2.1.2 Принципы построения тестов

Традиционный подход к построению тестов предполагает, что каждый тест представляет собой независимую единицу, состоящую из цепочки действий. Цепочка начинается с пролога, в котором происходит необходимая инициализация и подготовка системы к тестированию. Затем следует основная часть, содержащая активные воздействия на целевую систему, действия по наблюдению за поведением целевой системы и вынесение вердикта о корректности этого поведения. В заключении цепочки находится эпилог, в котором происходит освобождение ресурсов и приведение системы в исходное состояние. При построении тестов на основе функциональных требований, каждая проверка корректности обычно помечается идентификатором соответствующего элементарного требования.

Поскольку часто оказывается, что целая группа тестов выполняет приблизительно одни и те же действия, отличаясь лишь в небольшом числе деталей, то естественным развитием традиционного подхода является та или иная форма шаблонов тестов, позволяющая минимизировать дублирование кода. Примером такого развития является разработанная в ИСП РАН технология автоматизированной разработки тестов T2C (Template-to-Code) [6].

В основе T2C лежит автоматическая генерация исходных кодов тестов и других файлов, необходимых для компиляции и запуска тестов, из файлов,

содержащих шаблоны тестов. Эти файлы (далее - t2c-файлы) с шаблонами тестов создаются разработчиком тестов. Каждому тесту в t2c-файле соответствует своя секция, в которой задаётся тестовый сценарий на языке Си.

Проверки требований в тестовом сценарии выполняются с помощью специального макроса (REQ), которому в качестве параметров передаются:

- идентификатор проверяемого требования;
- булевское выражение на языке Си, невыполнение которого означает нарушение проверяемого требования;
- комментарий, который выводится в случае невыполнения проверки.

Тестовый сценарий может быть сам превращён в шаблон за счёт вынесения в параметры произвольных частей кода сценария, таких как входные данные, ожидаемые значения, идентификаторы проверяемых требований и т. д. После этого для тестового сценария можно указать один или несколько наборов значений параметров. В результате из одной секции тестового сценария в t2c-файле во время генерации кода может быть создано один или несколько тестов, каждый из которых получается подстановкой своего набора значений параметров.

Генерация кода тестов из t2c-файлов выполняется на основе специальных шаблонов генерации. За счёт выбора таких шаблонов можно получить то или иное представление тестового набора. По умолчанию генерируется набор тестов, пригодный для запуска под системой управления тестами TET [7]. Но есть также возможность генерации тестов, интегрируемых в другие системы выполнения тестов, а также так называемого автономного набора тестов, в котором каждому тесту соответствует отдельный исполняемый файл. В таком виде, например, можно сгенерировать отчуждаемый код для детального анализа конкретной ошибки.

Система T2C использовалась при разработке тестов для библиотек Linux, входящих в стандарт Linux Standard Base [8, 9]. С её помощью были подготовлены тесты для 10 библиотек. Более подробная статистика относительно тестирования части этих библиотек представлена в табл. 2. Описания найденных ошибок опубликованы на сайте [10].

Библио-тека	Версия	Протести-ровано интерфейсов	Проверено требований	Покрытие по коду	Найдено ошибок
libatk-1.0	1.19.6	222 из 222 (100%)	497 из 515 (96%)	-	11
libglib-2.0	2.14.0	832 из 847 (98%)	2290 из 2461 (93%)	12203 из 16263 (75.0%)	13
libgthread-2.0	2.14.0	2 из 2 (100%)	2 из 2 (100%)	149 из 211 (70.6%)	0
libgobject-2.0	2.16.0	313 из 314 (99%)	1014 из 1205 (84%)	5605 из 7000 (80.1%)	2
libgmodule-2.0	2.14.0	8 из 8 (100%)	17 из 21 (80%)	211 из 270 (78.1%)	2
libfonconfig	2.4.2	160 из 160 (100%)	213 из 272 (78%)	-	11
Всего		**1537 из 1553 (99%)**	**4033 из 4476 (90%)**	**18168 из 23744 (76.5%)**	**39**

Табл. 2. Результаты тестирования LSB библиотек тестами T2C.

Таким образом, система T2C продемонстрировала свои положительные стороны при достаточно масштабном применении, в условиях, когда не ставилась цель протестировать целевые библиотеки сверхтщательно и 80% покрытие считалось вполне достаточным.

В случае же, когда требуется более тщательное тестирование отдельных компонентов, наш опыт показывает [11], что наиболее эффективным подходом является разработка тестов на основе моделей. В рамках настоящей статьи мы продемонстрируем принцип построения тестов на основе моделей на примере технологии UniTESK [12], разработанной в ИСП РАН. В соответствии с технологией UniTESK функциональные требования представляются в формальном, машино-читаемом виде при помощи предусловий и постусловий модельных операций, а также инвариантов модельного состояния. Модельное состояние описывает представление тестовой системы о внутреннем состоянии целевой системы, а инварианты формулируют ограничения на его внутреннюю согласованность. Модельные операции, как правило, соответствуют интерфейсным функциям целевой системы. Предусловие описывает ограничения на состояние и аргументы функции, с которыми допустимо обращаться к ней, а постусловие формализует требования к результату работы функции, заключающемуся в

изменении состояния системы и формировании возвращаемого значения. При наличии каталога требований все проверки в инвариантах, предусловиях и постусловиях помечаются идентификатором соответствующего им элементарного требования из каталога.

Если в традиционном подходе к построению тестов проверка результата осуществляется вручную при помощи описания ожидаемого результата в явном виде, то при построении тестов на основе моделей наличие формальной модели требований позволяет выполнять такие проверки автоматически для произвольной тестовой ситуации. Как следствие, открывается возможность для автоматической генерации последовательностей тестовых воздействий на целевую систему, которые позволяют проверить целевую функциональность во множестве разнообразных ситуаций. Подготовить вручную сопоставимый массив тестов зачастую весьма непросто, а ещё сложнее такой массив сопровождать. Кроме того, наш опыт показывает, что автомат позволяет покрыть такие «угловые» случаи, которые вряд ли попадут в число тестов даже у очень изобретательного тестировщика. Достаточно часто в таких случаях находятся ошибки. В качестве примера, можно привести ситуацию с одной из реализаций очереди сообщений, в которой происходило зависание двух потоков, когда одновременно в списке ожидания на посылку сообщения в очередь и в списке ожидания на получение из очереди оказывалось более одного потока. А такое может случиться только в результате нетривиальной последовательности обращений к очереди из множества потоков с различными приоритетами.

Последовательность в рассматриваемом примере была сгенерирована при помощи метода построения нацеленных тестовых сценариев технологии UniTESK. Этот метод предлагает формировать тестовые сценарии, нацеленные на тщательную проверку определённой подобласти функциональности тестируемой системы на основе комбинации из трех элементов:

- модели требований;
- функции редукции модельного состояния;
- множества элементарных тестовых воздействий.

Модель требований описывает требования к целевой системе и не зависит от конкретного сценария. Функция редукции модельного состояния в совокупности с множеством элементарных тестовых воздействий формируют целеполагание данного тестового сценария, указывая, какие части модельного состояния являются важными, и подсказывая, из каких действий следует формировать последовательность тестовых воздействий. Далее метод предполагает, что инструменты, его реализующие, в ходе проведения тестирования автоматически сгенерируют последовательность тестовых воздействий таким образом, чтобы в каждом достигнутом редуцированном состоянии были выполнены все элементарные тестовые воздействия.

Ещё одним важным компонентом в технологии UniTESK являются медиаторы, которые устанавливают соответствие между моделью требований и тестируемой реализацией. Модельные операции отображаются на интерфейсные операции целевой системы, а модельное состояние обновляется медиатором на основе его представления о состоянии реализации.

С точки зрения тестирования ОС применение методов построения тестов на основе моделей, предполагающих достаточно много активностей в ходе самого тестирования, вызывает вопрос о возможных возмущениях, вносимых в работу ОС этими активностями — генерацией тестовых последовательностей и проверкой соответствия поведения тестируемых компонентов модели требований. Поскольку ОС управляет работой всех активностей на доверенном ей оборудовании, то для многих компонентов ОС такое возмущение может оказаться достаточно существенным.

С целью минимизации возможных возмущений мы предложили распределенную схему организации тестирования [13], согласно которой только небольшая часть тестовой системы работает на тестируемой ОС, а основная часть работает на отдельной инструментальной системе. Это достигается путём разделения медиатора на две части, которые взаимодействуют между собой через некоторый коммуникационный канал, например, через последовательный порт или сетевой интерфейс. Часть, работающая на тестируемой ОС, называется агентом. Её задача заключается в оказании тестовых воздействий в соответствии с указаниями, приходящими от тестовой системы, наблюдением за поведением целевой системы и передача собранной информации тестовой системе. Все остальные компоненты тестовой системы выполняются вне целевой ОС и никак не влияют на её работу. Таким образом, остаётся только обеспечить отсутствие влияния коммуникационного канала на целевые компоненты ОС.

Следует отметить, что многие интерфейсы ОС основаны на асинхронных принципах взаимодействия, что не вписывается в классический подход к описанию требований в виде предусловий и постусловий. Технология UniTESK была расширена для поддержки описания модели требований и построения нацеленных тестовых сценариев для систем с асинхронным интерфейсом [13].

Технология построения тестов на основе моделей для компонентов ОС применялась при тестировании ключевых компонентов, описываемых стандартами POSIX [14] и LSB [8, 9], и при тестировании ОСРВ на соответствие стандарту ARINC-653 [15]. Так, в проекте OLVER [16] целью тестирования были 1532 функции, описываемые стандартом LSB Core 3.1, в основном из библиотек glibc. В ходе анализа требований стандарта к этим функциям был составлен каталог требований, содержащий более 22 тыс. элементарных требований. Для поверки этих требований было разработано более 450 нацеленных тестовых сценариев, хотя ввиду ограничений по ресурсам далеко не для всех областей тестируемой функциональности

ставилась задача обеспечить полное покрытие требований. Поскольку тестируемый код поддерживал достаточно много расширений, не описанных в стандарте, на соответствие которому проводилось тестирование, то метрика покрытия по исходному коду не могла выступать в виде объективного критерия оценки качества тестирования на соответствие. В проекте тестирования ОСРВ [17] проверялись 54 функции, описываемые стандартом ARINC-653 часть 1, к которым предъявлялось 315 элементарных требований, и все они были покрыты тестами. Ввиду отсутствия доступа к исходному коду ОСРВ измерения покрытия по коду не проводились.

2.1.3 Тестирование обработки внутренних ошибок

Существенная часть кода многих компонентов ОС отвечает за обработку ошибок, которые могут возникнуть при выполнении различных операций, например, по причине нехватки памяти или сбоя при обращении к аппаратуре. Для того, чтобы проверить код обработки ошибок, требуется создать ситуацию, когда эти ошибки в действительности случатся. Причём создавать такие ситуации нужно целенаправленно, так как для сложных компонентов удар «по площадям» не работает. Например, попытки организовать «истощение» памяти в ядре ОС натыкается на сильное «сцепление» компонентов ядра, из-за которого чрезвычайно сложно создать ситуацию, когда нехватка памяти будет обнаружена именно тем компонентом, который тестируется, а не компонентом, который находится раньше в цепочке взаимодействия компонентов.

При тестировании кода, работающего в пользовательском пространстве, существует достаточное количество инструментов, которые позволяют решить данную проблему. Например, типичным подходом является использование специализированных библиотек выделения динамической памяти, которые реализуют стандартный интерфейс malloc/free и при запуске целевых компонентов подменяют стандартные библиотеки посредством возможностей, предоставляемых загрузчиком приложений ОС.

При тестировании компонентов ОС, работающих в пространстве ядра, задача не имеет такого простого решения. В первую очередь следует отметить, что хотя код ядра исполняется в адресном пространстве, отделенным от пространства пользователя, сценарии тестирования ядра обычно задаются с помощью программ, исполняющихся в пользовательском пространстве. Дело в том, что взаимодействие различных подсистем внутри ядра достаточно сложно, и написать тестовый сценарий, пригодный для исполнения полностью в пространстве ядра, весьма непросто. С другой стороны, большинство функциональности ядра (в том числе и драйверов), написана из расчёта их вызова при обработке того или иного системного вызова из пространства пользователя или других способов взаимодействия пользовательских программ с ядром ОС. Поэтому использование пользовательских программ в качестве сценариев тестирования ядра ОС даёт хорошие результаты.

Таким образом, для тестирования устойчивости компонентов ядра необходима возможность управлять сбоями в ядре из пользовательского пространства. Рассмотрим, какие инструменты для этого можно использовать на примере ОС Linux.

В коде ядра Linux присутствует инструмент Fault Injection [18], позволяющий симулировать сбой той или иной функции выделения памяти. За счёт различных опций можно добиться симуляции сбоя в том или ином вызове функции. Существенным для автоматического тестирования недостатком этого инструмента является ограниченность сценариев симуляции сбоев: инструмент поддерживает только вероятностный сценарий (при каждом вызове функции есть вероятность N, что вызов вернёт ошибку) и интервальный (задаётся минимальное время между последовательными симуляциями сбоев). С помощью таких сценариев очень сложно ограничить симуляцию сбоев только заранее заданным вызовом функции, из-за чего теряется воспроизводимость поведения системы при тестировании. Дополнительные сценарии можно реализовать только модифицировав код ядра.

Для преодоления обозначенных проблем в ИСП РАН была разработана платформа KEDR [19, 20], позволяющая наблюдать за модулем ядра ОС Linux и осуществлять перехват вызовов этим модулем внешних функций, таких как функции основной части ядра или функции, реализованные в другом модуле. Платформа реализована в виде модуля ядра, который за счёт инструментирования кода целевого драйвера позволяет выполнить код до, после или вместо вызова определённой внешней функции. Код для выполнения задаётся в дополнительных модулях, которые могут быть загружены в ядро при необходимости.

На платформе KEDR был реализован специализированный инструмент KEDR Fault Simulation, позволяющий симулировать сбои в функциях, вызываемых драйвером. В состав инструмента уже включены широкие возможности по организации сценариев симуляции, такие как возможность задавать номер вызова функции, который вернёт ошибку, указывать условия на значения параметров функции ("size >= 256 && flags != GFP_ATOMIC") и на адрес, откуда выполняется вызов. Это уже позволяет разрабатывать тесты с воспроизводимым поведением системы. Помимо этого, KEDR Fault Simulation позволяет реализовывать дополнительные сценарии без модификации кода ядра и самого инструмента, а также расширять список функций, для которых становится возможной симуляция сбоев.

В настоящее время KEDR Fault Simulation применяется в проекте по разработке тщательных тестов для драйверов файловых систем, работающих в пространстве ядра ОС Linux [21]. В ходе проекта инструмент продемонстрировал способность осуществлять нацеленную симуляцию ошибок при вызове функций ядра по указанию тестовых сценариев, работающих в пространстве пользователя. Помимо этого, в проекте был

предложен эффективный метод систематического внесения внутренних ошибок, который основан на следующей идее. Сначала в ходе выполнения традиционного функционального теста в специальном режиме выявляются все вызовы функций, которые могут возвращать ошибки. Затем каждый тест запускается N раз, где N – число таких вызовов функций в тесте. Каждый i-й запуск выполняется под управлением KEDR Fault Simulation со сценарием симуляции «i-й вызов функции возвращает ошибку». Этот метод продемонстрировал свою эффективность, позволив обнаружить несколько ошибок в таком зрелом драйвере файловой системе как ext4 и большое число ошибок в менее зрелых драйверах файловых систем.

2.1.4 Специализированные тестовые системы

Наш опыт разработки тщательных тестов для разнообразных компонентов ОС показывает, что часто для конкретного целевого компонента наиболее эффективным оказывается разработка специализированной тестовой системы, которая может базироваться на одном из рассмотренных принципов построения тестов, а может быть и полностью специфической.

Последний вариант, например, применялся при тестировании библиотек математических функций [22]. В этом случае каждая тестируемая функция получает одно или несколько чисел с плавающей точкой и должна вычислить число с плавающей точкой, наиболее точно представляющее результат соответствующей математической функции в текущем режиме округления и, возможно, выставить некоторые флаги, сигнализирующие об определённых событиях, таких как неточный результат, переполнение и т. д. При разработке тестов для таких функций наиболее трудоёмким является подбор множества тестовых значений, которые бы покрыли все значимые подпространства множества входных значений, интересные значения с точки зрения дилеммы составителя таблиц и т. д., а также вычисление при помощи нескольких библиотек функций, работающих с числами с повышенной точностью, корректных выходных значений для каждого входного значения. Инфраструктура же времени выполнения тестов достаточна проста: прочитать очередное входное и ожидаемое выходное значения, вызвать целевую функцию и проверить, насколько её результат отличается от ожидаемого. Нетривиальные составляющие появляются только при реализации автоматической аналитики обнаруживаемых ошибок, их кластеризации и т. д. Применение специализированного решения для тестирования математических функций продемонстрировало себя достаточно хорошо, поскольку позволило сосредоточиться на наиболее сложной подзадаче. А получившаяся тестовая система позволила обнаружить множество неожиданных результатов в распространённых библиотеках математических функций, в особенности, в нестандартных режимах округления [22].

Аналогичный математическим тестам подход к построению тестовых систем применялся и при тестировании ряда других компонентов ОС, таких как

обработка форматированного ввода-вывода. Также специализированные решения применялись при тестировании компонентов сетевого стека протоколов и драйверов файловых систем.

Если в случае тестирования протоколов за основу брался принцип построения тестов на основе моделей по технологии UniTESK [11], то при тестировании драйверов файловых систем ОС Linux в основе лежали идеи системы T2C, которые были адаптированы для тестирования компонентов ядра ОС.

Для тестирования драйверов файловых систем была создана специализированная система запуска тестов, которая поддерживает следующие возможности.

- Форматирование и монтирование тестируемой файловой системы перед началом теста. Большинство тестов реализованы как операции с файлами и директориями на уже примонтированной файловой системе.
- Опции форматирования и монтирования берутся из предварительно заполненного списка. Таким образом, каждый тест, который ориентирован на уже примонтированную файловую систему, выполняется несколько раз для разных вариантов форматирования и монтирования.
- Режим тестирования с систематической симуляцией сбоев, описанной в разделе 2.1.3.
- В случае, когда в результате одного из тестов система приходит в нестабильное состояние, в котором дальнейшее тестирование бессмысленно, есть возможность перезагрузить систему и продолжить тестирование. К таким необратимым состояниям относятся, например, срабатывания BUG_ON() в коде ядра или драйвера. Как показала практика, таких ошибок возникает немало, особенно при тестировании с симуляцией сбоев.
- Режим автоматического обнаружения утечек ресурсов с применением инструментов, описанных в разделе 4.1.

В табл. 3 представлено покрытие по исходному коду, которое достигается текущей версией тестов, и дополнительная добавка, получаемая за счёт активации метода систематической симуляции сбоев, описанного в разделе 2.1.3. В настоящий момент развитие тестового набора продолжается. Несмотря на то, что целенаправленный анализ обнаруживаемых тестами ошибок практически не проводился, в результате взаимодействия с разработчиками уже было исправлено около десятка ошибок в драйверах файловых систем, включая 4 ошибки в такой зрелой файловой системе как ext4.

Файловая система	Версия	Покрытие по строкам кода (без симуляции сбоев)	Покрытие по строкам кода (с симуляцией сбоев)
JFS	3.2	65%	71%
Ext4	3.2	60%	64%
F2FS	3.9	70%	75%

Табл. 3. Текущие показатели покрытия по строкам кода при тестировании драйверов файловых систем.

2.2. Поверхностное функциональное тестирование

Как демонстрирует статистика, приведённая в табл. 1, широкомасштабное функциональное тестирования библиотек ОС общего назначения тщательным образом не представляется возможным ввиду огромного количества имеющейся функциональности. Тщательное тестирование имеет смысл для ключевых компонентов или для компонентов, которые планируется применять для решения ответственных задач. Но это не означает, что тестировать остальные компоненты не нужно. В идеале за это должны отвечать разработчики соответствующих компонентов, но в сообществе разработчиков свободных программ это случается далеко не всегда. Большинство таких проектов просто не имеют необходимых ресурсов, чтобы уделять разработке тестов достаточное внимание, особенно на начальных стадиях. Да и по мере развития ситуация не становится проще, так как задача покрытия тестами всех функций библиотеки становится всё менее обозримой.

В данных условиях значительную помощь в исправлении ситуации могут оказать специализированные методы генерации поверхностных тестов или тестов работоспособности, под которыми обычно понимают тесты, проверяющие только то, что основные функции системы выполняются более-менее правильно, то есть, что система не разрушается и возвращает результаты, проходящие простейшие проверки на корректность (полная проверка при этом не выполняется).

Примером реализации таких методов являются инструменты Azov [23] и их дальнейшее развитие — API Sanity Autotest [24, 25]. Последний способен полностью автоматически генерировать тесты работоспособности только на основе заголовочных файлов библиотеки. При этом используется информация о сигнатуре публичных функций библиотеки, то есть о типах её входных и выходных параметров. Конечно, в большинстве случаев этой информации недостаточно для создания корректных тестов, например, из-за необходимости инициализации библиотеки или значений определённого типа неким нетривиальным образом.

Поэтому API Sanity Autotest поддерживает возможность задания дополнительной семантической информации об особенностях библиотеки, представленных в ней типах данных и специфики их использования в определённых функциях. Типичными примерами такой информации являются описания того:

- как получить корректное значение определённого типа данных;
- каким должно быть корректное значение определённого параметра функции;
- какие проверки можно сделать для возвращаемых значений определённого типа.

Может сложиться впечатление, что в этом подходе предлагается «варить суп из топора»: какая автоматизация, если требуется вручную описать код, необходимый и при обычной разработке тестов? Но тем не менее, польза от API Sanity Autotest есть как минимум по двум направлениям. Во-первых, описания пишутся в виде кода на C/C++, в котором могут присутствовать специальные конструкции, при помощи которых можно попросить инструмент сгенерировать код для получения значения определённого типа или подготовить параметры и вызвать определённую функцию. Во-вторых, семантическая информация одновременно привязывается ко многим местам, где она требуется. Например, информация о специфике инициализации библиотеки записывается один раз для всех функций. Информация о создании объектов определённого типа также записывается в одном месте и затем используется для всех функций, у которых есть параметр такого типа, и в других необходимых местах, таких как специальные конструкции, о которых шла речь выше.

Всё вместе это позволяет минимизировать дублирование кода, необходимого для подготовки описаний, что значительно сэкономит усилия при генерации тестов для больших библиотек, в которых типы данных и другие семантические элементы являются общими сразу для многих функций. Полученные в результате генерации тесты содержат минимальные проверки того, что функция работает на наиболее простом сценарии ее использования и возвращает более-менее правильный результат. Наличие таких тестов уже является большим шагом вперёд по сравнению с отсутствием тестов вовсе. И даже такие тесты позволяют обнаружить десятки ошибок.

Не менее ценным фактом является то, что сгенерированные тесты могут быть использованы в качестве хорошей стартовой точки для разработки полноценных функциональных тестов. Когда тесты работоспособности доведены до состояния корректной работы, то есть когда семантической информации достаточно для генерации корректных вызовов всех необходимых функций, полученный тестовый набор можно доработать вручную для обеспечения более качественного тестирования наиболее важных частей библиотеки. Для этих целей API Sanity Autotest поддерживает

генерацию тестов в формате T2C, который предоставляет удобные возможности для дальнейшего развития тестового набора.

Инструмент использовался для генерации тестов работоспособности для крупных библиотек, входящих в состав LSB: Qt3 (тесты для 9 792 функций), Qt4 (тесты для 10 803 функций), libxml2 (тесты для 1 284 функций). Кроме того, инструмент начал использоваться и разработчиками свободных библиотек.

3. Тестирование обратной совместимости

Современные операционные системы развиваются независимо от пользовательских приложений. Как следствие, возникает необходимость переноса приложений между различными версиями одной ОС. И так как приложений гораздо больше, чем ОС, то в большинстве случаев задача обеспечения переносимости перекладывается на плечи ОС в виде требования обеспечить обратную совместимость с приложениями. Это означает, что приложение, предназначенное для старой версии ОС, должно беспроблемно работать на новой версии ОС.

Обратная совместимость обычно рассматривается на одном из двух уровней: на уровне бинарных файлов или на уровне исходных кодов. В первом случае требуется, чтобы уже скомпилированное приложение можно было установить и использовать на новой версии ОС. Во втором случае речь идёт о возможности перекомпиляции исходного кода приложения под новую версию ОС без каких-либо изменений в коде.

Как правило, ОС обеспечивают обратную совместимость при условии, что приложения используют только специфицированные возможности ОС. Также существуют отраслевые и международные стандарты, регламентирующие интерфейс между приложениями и ОС, который позволяет обеспечить переносимость приложений между различными ОС. Примерами таких стандартов являются POSIX [14] и LSB [8, 9]. POSIX описывает интерфейс на уровне исходных кодов для семейства ОС UNIX. LSB нацелен на обеспечение переносимости на бинарном уровне между дистрибутивами ОС Linux.

Среди методов тестирования обратной совместимости можно выделить структурные и семантические.

3.1. Структурное тестирование обратной совместимости

При структурном тестировании обратной совместимости проверяется лишь наличие в новой версии ОС необходимой номенклатуры функций, которые входили в публичный интерфейс предыдущей версии, и, соответственно, не проверяется, что поведение этих функций осталось в рамках специфицированной ранее функциональности.

Примером инструментов структурного тестирования являются инструменты LSB libchk [26], который, имея список функций, входящих в состав стандарта

LSB, проверяет их наличие в соответствующих библиотеках тестируемой ОС. Поскольку в случае libchk проверка идёт на бинарном уровне ELF файлов, то для Си функций проверяется только имя функции и не отслеживаются изменения в её параметрах. Поскольку типы параметров для C++ функций кодируются в имена функций на бинарном уровне, то для них изменения в типах входных параметров могут быть обнаружены, но не могут быть обнаружены изменения в типе возвращаемого значения.

Схожим образом устроены сигнатурные тесты из Android Compatibility Test Suite [27]. Но они в исходных данных имеют список публичных интерфейсов с полной сигнатурой функции и, соответственно, проверяют неизменность сигнатуры в тестируемой версии ОС.

Следует отметить, что формирование и сопровождение списков публичных интерфейсов требует определённых усилий, и хотя оно хорошо вписывается в сертификационные программы, разработчикам библиотек такой подход не всегда удобен. Существует также ряд инструментов, позволяющих контролировать изменения в наборе функций, экспортируемых двумя версиями библиотеки, и, таким образом, избегать проблем, связанных с исключением функций. К таким инструментам относятся chkshlib [28], cmpdylib [29], cmpshlib [30], dpkg-gensymbols [31]. Все они работают по единому принципу, извлекая список функций из двух версий библиотек и сравнивая их. Различие состоит лишь в реализации этих инструментов. Другие виды несовместимостей эти инструменты находить не способны.

В качестве альтернативного варианта в ИСП РАН разработан инструмент ABI Compliance Checker [32], который в дополнение к сравнению списков экспортируемых функций позволяет находить несовместимые изменения в сигнатурах этих функций, в том числе опосредованные изменения, являющиеся следствием, например, изменений в определении соответствующих структур данных. Реализуется эта возможность за счёт дополнительного анализа деревьев синтаксического разбора заголовочных файлов библиотеки.

В качестве дополнительных входных данных инструмент получает списки внутренних функций и типов, проверку которых осуществлять не требуется. В библиотеках на языке Си эти входные данные наиболее актуальны, поскольку отсутствует контроль доступа на уровне компиляции. Выходными данными инструмента является отчёт в формате html с результатами проверки двух версий библиотеки на совместимость. В отчёте все типы проблем совместимости разделены на две группы – проблемы функций и проблемы типов данных. При этом для каждой проблемы, связанной с изменением типа данных, указывается, на какие экспортируемые функции это изменение могло повлиять. Все изменения в отчёте распределяются по трём уровням потенциального риска для пользователей библиотеки. Помимо изменений, которые могут повлиять на несовместимость на бинарном уровне, также

идентифицируются изменения, которые могут повлиять на совместимость на уровне исходных кодов.

3.2. Семантическое тестирование обратной совместимости

Под семантическим тестированием обратной совместимости подразумевается тестирование, которое нацелено на выявление несовместимостей, в том числе, на семантическом уровне, то есть на уровне поведения экспортируемых функций. Наиболее распространённым подходом к семантическому тестированию обратной совместимости является проведение обычного функционального тестирования новой версии библиотеки на соответствие требованиям.

В случае проверки обратной совместимости на бинарном уровне применяется также такой подход, как компиляция функциональных тестов со старой версией библиотеки и запуск этих тестов на новой версии.

Ещё одним логичным шагом при семантическом тестировании обратной совместимости могло бы быть использование старой версии библиотеки в качестве эталона для вынесения вердикта о корректности поведения новой версии библиотеки, но авторам о применении такого подхода на практике ничего не известно. По всей видимости, это обусловлено тем, что в случаях, когда обратной совместимости уделяется серьёзное внимание, также много внимания уделяется и качеству библиотек в целом, а значит и их функциональному тестированию, что позволяет успешно решать и задачу обеспечения обратной совместимости на семантическом уровне.

4. Обнаружение специфических видов ошибок

Ряд ошибок в программах проявляются не сразу и/или опосредованным образом, поэтому традиционные подходы функционального тестирования не эффективны для их выявления. В этом разделе мы рассмотрим два класса таких ошибок в контексте тестирования ОС: утечки ресурсов и гонки по данным. Также в этом разделе будут рассмотрены ошибки, связанные с некорректной обработкой некорректных входных данных.

В качестве интересного факта следует упомянуть, что анализ исправлений ошибок в стабильных версиях ядра ОС Linux за один календарный год [33] показал, что гонки по данным и утечки ресурсов оказались наиболее распространёнными видами исправляемых ошибок: они встречаются в 17% и 16% случаев соответственно.

4.1. Обнаружение утечек ресурсов

Под утечкой ресурсов понимаются ситуации, когда ресурс, выделенный для определённого использования, перестаёт быть нужным, но не освобождается или, другими словами, не возвращается в пул свободных ресурсов. В результате свободные ресурсы могут исчерпаться, и функциональность

системы окажется ограниченной, несмотря на то, что часть «занятых» ресурсов данного вида на самом деле никому не нужна.

Утечки наиболее неприятны для компонентов с продолжительным временем жизни, так как в этом случае они могут накапливаться и привести к нехватке ресурсов. Большинство компонентов ОС обладают длительным временем жизни или, как в случае с системными библиотеками, могут оказаться частью программы с длительным временем жизни. Но наиболее неприятны утечки для ядра ОС, так как время его жизни совпадает со временем непрерывной работы всей системы в целом. Кроме того, в отличие от пользовательских программ, где по окончании работы многие категории используемых программой ресурсов (память, файловые дескрипторы и др.) освобождаются автоматически, ядро ОС должно освобождать все ресурсы явно.

Для пользовательских приложений существует множество инструментов, помогающих обнаруживать утечки, в первую очередь утечки памяти, с помощью специализированных библиотек или более разносторонних подходов, таких как, например, реализованные в инструменте Valgrind [34]. Возможности по обнаружению утечек в ядре ОС более ограничены. Рассмотрим доступные инструменты на примере ядра ОС Linux.

В ядре Linux есть встроенный инструмент kmemleak [35], предназначенный для выявления утечек памяти. Этот инструмент поддерживает список выделенных участков памяти, используя дополнительный код в реализации функций выделения и освобождения памяти. Для выявления утечек памяти используется сканирование всей памяти, используемой ядром, на предмет наличия в них последовательностей байтов, которые могут быть указателем на ту или иную выделенную область памяти. Если при сканировании не обнаружилось возможного указателя на какой-то выделенный участок памяти, то этот участок памяти считается потерянным.

Однако, у такого метода выявления утечек памяти есть следующие недостатки.

- Сканирование всей памяти ядра — долгий процесс. Даже с использованием различных оптимизаций сканирование может выполняться несколько минут.
- Критерий утечек памяти, основанный на поиске возможного указателя на выделенный участок памяти, неточен. Возможны как ложные срабатывания (указатель на выделенную область памяти может хранится в памяти неявно), так и пропущенные ошибки (найденная последовательность байт может, в реальности, и не быть указателем).

Из-за этих недостатков применение kmemleak для автоматических тестов затруднительно, так как поиск утечек памяти будет сильно увеличивать время тестов, а результат должен дополнительно проверяться человеком на предмет

ложных срабатываний. Что до пропущенных ошибок, то их надо выявлять другими инструментами.

На основе платформы KEDR был разработан инструмент KEDR Leak Check, который может использоваться для более эффективного выявления утечек памяти в модулях ядра. Перехват вызовов функций, обеспечиваемый платформой KEDR, в этом инструменте используется для отслеживания выделенных участков памяти. Критерием же отсутствия утечек памяти в варианте использования по умолчанию является отсутствие выделенных участков памяти на момент выгрузки модуля.

KEDR Leak Check решает проблемы скорости и точности, присущие kmemleak. Это позволяет использовать KEDR Leak Check при автоматическом тестировании. Кроме того, данный инструмент выигрывает у kmemleak в плане перечисленных ниже удобств использования и расширяемости.

- Для использования kmemleak ядро должно быть собрано с соответствующей опцией, которая на большинстве дистрибутивов по умолчанию отключена. Для использования KEDR Leak Check пересборка ядра обычно не требуется, так как все необходимые для его работы опции ядра включены по умолчанию.
- KEDR Leak Check выявляет утечки только в одном модуле ядра, не перемешивая их с утечками в других модулях и в основной части ядра.
- Функциональность KEDR Leak Check может быть расширена для выявления утечек других типов ресурсов. Это расширение достигается путём написания дополнительных модулей. Для аналогичного расширения функциональности kmemleak необходимо модифицировать код ядра.

Из ограничений KEDR Leak Check можно отметить следующие.

- Для обнаружения утечек памяти необходима выгрузка модуля. kmemleak же может выявлять утечки даже при работающем драйвере.
- Инструмент не применим для обнаружения утечек в коде основной части ядра.
- Для корректной работы KEDR Leak Check необходим перехват большего количества функций, чем для kmemleak.
- Некоторые ресурсы выделяются основным кодом ядра перед вызовом некоторой функции драйвера (так называемой, callback-функции), а освобождать их должен драйвер. Случается и обратная ситуация, когда ресурс выделяется драйвером, а освобождается после того, как callback-функция возвратит управление основному коду ядра. Для корректного выявления утечек памяти в таких случаях необходимо отслеживать вызов/возврат из соответствующих callback-функций.

4.2. Обнаружение гонок по данным

Гонками по данным обычно называют ситуации, при которых два параллельно работающих потока управления «одновременно» обращаются к разделяемым данным и получают при этом некорректный результат. Например, если одна функции будет читать значение составной переменной из памяти, а другая будет записывать туда новое значение, то может оказаться, что читатель прочитает часть старого значение, после чего значение переменной обновится писателем, и вторую часть переменной читатель прочитает уже из её нового значения. В результате у читателя может оказаться значение, которое не соответствует ни старому значению переменной, ни новому. Аналогичная ситуация может сложиться и при наличии двух одновременно работающих писателей. В этом случае неконсистентное значение может оказаться записанным в переменную.

Гонки по данным могут происходить и на нескольких разделяемых переменных, связанных между собой семантическими связями. Как правило, такие гонки называются высокоуровневыми. Ещё одним типичным примером гонок является ситуация, когда одна функция работает с указателем на динамическую память, а параллельно работающая функция эту память освобождает в середине работы первой функции, после чего последующие разыменования указателя в ней могут привести к непредсказуемым последствиям.

Поскольку гонки случаются непредсказуемым образом при наложении определённых событий во времени, а последствия могут проявляться не сразу, да к тому же в коде, никак не связанном с тем кодом, где присутствует ошибка, то искать такие ошибки очень непросто, и известны случаи, когда корень таких ошибок с весьма неприятными последствиями удавалось найти лишь спустя десятки месяцев достаточно напряжённых поисков.

Для поиска гонок в пользовательских приложениях существует некоторое количество доступных инструментов, таких как Helgrind [36] и Google ThreadSanitizer [37]. Мы рассмотрим более детально, какие техники могут применяться для ядра ОС.

4.2.1 Kernel Strider и ОС2000 Data Race Detector

И Helgrind, и ThreadSanitizer построены на схожих принципах. В инструментах существует часть, ответственная за сбор информации обо всех доступах анализируемой программы к памяти и обо всех обращениях к примитивам синхронизации, и вторая часть, которая анализирует собранную информацию и сообщает о подозрительных местах. Helgrind и ThreadSanitizer первой версии собирают информацию при помощи инфраструктуры платформы Valgrind, а ThreadSanitizer второй версии — за счёт инструментации кода, выполняемой в ходе компиляции программы. Поскольку Valgrind в принципе не совместим с пространством ядра, то Google

финансировал[1] совместный с ИСП РАН проект Kernel Strider [38] по разработке компонента, собирающего информацию для ThreadSanitizer, на основе платформы KEDR. Следует отметить, что инструментация кода в компиляторе, которая используется в ThreadSanitizer второй версии, может быть адаптирована для применения в пространстве ядра, но на момент старта проекта второй версии инструмента ThreadSanitizer ещё не существовало, и по состоянию на текущий момент такой адаптации пока не реализовано.

Для реализации Kernel Strider потребовалось значительно расширить возможности платформы KEDR с целью перехвата не только вызовов функций, но и выполнения машинных операций, работающих с оперативной памятью. Целью перехвата является трасса обращений к ячейкам оперативной памяти и вызовов примитивов синхронизации, которая поступает для дальнейшего анализа на наличие состояний гонок с помощью Google ThreadSanitizer.

Использование Kernel Strider в его базовой конфигурации возможно лишь в «ручном» режиме, поскольку без дополнительной настройки на тестируемый модуль инструмент выдаёт много ложных срабатываний. Причина большинства из этих ложных срабатываний в том, что на порядок вызова callback-функций модуля ядро накладывает дополнительные ограничения, которые никак не учитываются инструментом. Один из способов настройки инструмента — аннотация тестируемого модуля. Аннотации могут описывать дополнительные отношения синхронизации, которые определяются устройством конкретного модуля.

Другой способ настройки инструмента — использование Kernel Strider как платформы для перехвата вызовов функций по аналогии с KEDR. С помощью такого перехвата можно добавлять в трассу события синхронизации, эквивалентные ограничениям, накладываемым ядром на вызов callback-функций драйвера. Такой способ настройки сложнее, но более универсальный — получившаяся настройка может использоваться не только для конкретного драйвера, а для целого семейства драйверов. В данный момент идёт настройка инструмента на семейства драйверов файловых систем и сетевых карт.

Для обнаружения гонок в ядре ОСРВ применялся схожий подход, реализованный в инструменте OC2000 Data Race Detector. Этот инструмент представляет собой специально модифицированный эмулятор центрального процессора, который в ходе своей работы собирает информацию обо всех операциях с памятью и обо всех вызываемых примитивах синхронизации. Дальнейшая обработка собранной информации также выполняется при помощи Google ThreadSanitizer.

[1] Google Research Award 2011 "Instrumentation and Data Collection Framework for Dynamic Data Race Detection in Linux Kernel Modules".

4.2.2 Race Hound

Инструмент Kernel Strider довольно сложен в реализации и требует настройки на каждое семейство драйверов. Взамен он позволяет выявлять состояния гонки, которые не произошли при данном выполнении кода драйвера, но могли бы произойти.

Инструмент Race Hound [39], реализующий идеи, предложенные в инструменте DataCollider для платформы Windows, и также разработанный в ИСП РАН, позволяет модифицировать последовательность выполнения кода и даёт шанс гонке реально произойти. Модификация времени исполнения кода осуществляется за счёт вставки дополнительных циклов ожидания после тех или иных инструкций кода. Помимо вставки дополнительных циклов ожидания, выполняется наблюдение за обращениями к участкам памяти, к которым обращается инструкция кода. В случае, если это наблюдение выявило ситуацию, подходящую под определение состояния гонки, об этом факте сообщается пользователю.

Для вставки кода после инструкций кода используется существующий механизм Kernel Probes, для наблюдения за обращениями к участкам памяти используются аппаратные точки прерывания. Количество аппаратных точек прерывания определяется архитектурой операционной системы, и их количество обычно невелико (например, x86 и x86_64 поддерживают до четырёх аппаратных точек прерывания), поэтому инструмент Race Hound может одновременно наблюдать только за ограниченным числом ячеек памяти. Для слежения за бо́льшим количеством ячеек их набор меняется с течением времени по рандомизированному алгоритму.

Такие особенности реализации делают процесс обнаружения гонок с помощью Race Hound вероятностным: чем чаще повторяется тот или иной участок кода, тем больше вероятность воспроизвести в нем состояние гонки, если, конечно, оно есть. При этом гарантированно найти все состояния гонки невозможно.

Тем не менее, Race Hound является хорошим инструментом для доказательства возможного состояния гонки, обнаруженного другим инструментом, например, Kernel Strider. В этом случае Race Hound наблюдает только за заранее заданными ячейками и вставляет дополнительные циклы ожидания только после заранее заданных инструкций. С помощью такого совместного использования Kernel Strider и Race Hound было обнаружено и доказано 3 состояния гонки в сетевых драйверах ОС Linux.

4.3. Тестирование устойчивости

Ещё одним видом специфических ошибок является некорректная обработка ОС некорректных входных данных. Этот вид ошибок весьма важен именно для ОС, поскольку к ним предъявляются повышенные требования по

устойчивости к атакам и вредоносным действиям со стороны недоверенных приложений, сетевых контрагентов, подключаемых внешних устройств и т. д.

К тестированию устойчивости существуют различные подходы. Один из подходов — рассматривать требования устойчивости как частный случай функциональных требований и, соответственно, проводить тестирование устойчивости в рамках функционального тестирования. Однако, это далеко не всегда удобно, т. к. может оказаться, что функциональные тесты, нацеленные на работу с корректными аргументами, и тесты устойчивости, нацеленные на передачу всевозможных некорректных данных, удобнее строить на основе различных подходов.

Другой подход заключается в развитии генераторов тестов работоспособности с целью поддержки перебора всевозможных некорректных данных по отдельным аргументам при условии, что остальные данные являются корректными (часто такой подход называют fuzz testing или fuzzing). Опыт генерации тестов с чисто случайными данными показывает малую эффективность, ввиду того, что большинство сгенерированных векторов входных данных не проходят далее первых тривиальных проверок в начале функции и, соответственно, просто не добираются до последующего кода, содержащего ошибки. Поэтому генерация корректных значений для части аргументов играет очень важную роль. Для этого обычно применяются аннотации, схожие с описанными выше генераторами тестов работоспособности. Наиболее успешным представителем инструментов данного класса является инструмент Trinity [40], предназначенный для тестирования устойчивости системных вызовов ОС Linux.

Дальнейшим развитием автоматического подхода являются инструменты, реализующие идею Concolic (или DART) тестирования. Идея заключается в одновременном тестировании целевого компонента и символическом выполнении его кода с целью выявления точек ветвления в программе и автоматическим подбором входных данных, которые бы позволили свернуть в этой точке ветвления не туда, куда пошло текущее выполнение, а в другую сторону. Данный подход является достаточно популярным направлением исследований в настоящее время и существует множество инструментов, его реализующих. В ходе исследования этого направления в ИСП РАН был разработан инструмент Avalanche [41], который продемонстрировал способность находить ошибки в коде, работающем в пространстве пользователя. Кроме того, в ИСП РАН ведутся работы по исследованию эффективности применения методов Concolic тестирования к коду ядра ОС Linux на основе адаптации инструментов S2E [42]. В качестве целевого компонента для тестирования используются драйвера файловых систем.

5. Тестирование производительности ОС

Тестирование производительности в инженерии программного обеспечения определяется как тестирование, которое проводится с целью определения, как

быстро работает вычислительная система или её часть под определённой нагрузкой. Такое тестирование может также служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов. Заметим, что в отличии от других видов тестирования, тестирование производительности обычно не ставит своей целью выявление нарушений заранее зафиксированных требований. Когда же речь идёт о производительности, такие требования зачастую отсутствуют. Это вызвано тем, что конкретные значения разнообразных показателей производительности для одной и той же программной системы сильно зависят от возможностей аппаратных платформ, на которых может выполняться эта программная система. Поэтому постановка задачи тестирования производительности сложных программных систем, к которым относятся и операционные системы, требует дополнительной конкретизации. Набор задач тестирования определяется как архитектурой программной системы и ее аппаратной платформы, так и целями тестирования. В достаточно общем случае к задачам тестирования производительности относится:

1. определение конкретных показателей производительности, свойственных каждой конкретной программной системе или классу систем;
2. определение зависимости показателей производительности программной системы от конфигурации системы и характеристик программно-аппаратной платформы, на которой она выполняется;
3. для каждого из показателей определение такой методики измерения значений этого показателя, которая обладает известной точностью (погрешностью) измерений;
4. создание программных средств для измерения значений показателей производительности по определённым методикам и для анализа контролируемых зависимостей между показателями производительности и влияющими на них факторами;
5. контроль изменений показателей производительности в процессе развития программной системы.

Таким образом, если разработчик программной системы предоставляет данные о значениях показателей производительности, которые должны соблюдаться в этой системе при определённых условиях, то тестирование производительности заключается в проверке соответствия измеренных значений заявленным. В противном случае тестирование производительности заключается в контроле изменений значений показателей с целью своевременного выявления случаев деградации работоспособности системы и условий, при которых такая деградация стала возможной.

Применительно к ОС и, в особенности, к ОС жёсткого реального времени задача тестирования производительности имеет особое значение. Это значение определяется ролью операционной системы в обеспечении

функционирования современных вычислительных компьютерных систем и влиянием производительности ОС не только на производительность, но и на функциональные характеристики всей системы в целом. В отличие от универсальных ОС основной задачей ОСРВ является своевременность (timeliness) выполнения обработки данных. Поэтому в качестве основного требования к ОСРВ выдвигается требование обеспечения предсказуемости (детерминированности) поведения системы в наихудших внешних условиях. Тестирование производительности помогает проверить выполнение этого требования.

5.1. Показатели производительности

Вообще говоря, номенклатура показателей производительности ОС зависит от её назначения и от стандарта (например, POSIX, ARINC-653 и т. п.), на основе которого разработана конкретная ОС и определяющего состав системных ресурсов и сервисов. Тем не менее можно сформулировать ряд низкоуровневых показателей, которые будут применимы к подавляющему большинству ОС. К таким показателям относятся:

- время обработки прерывания;
- время создания потока управления;
- время переключения между потоками управления (переключение контекстов потоков);
- время переключения процессов (переключение контекстов процессов);
- время создания ресурсов (семафоров, очередей, мьютексов, таймеров и т. п.);
- пропускная способность системы обмена сообщениями между потоками управления и/или процессами.

Этот список можно дополнить временными показателями выполнения каждого системного сервиса, в совокупности покрывающих все отдельные базовые операции, предоставляемые операционной системой приложениям. Полученная в результате детальная номенклатура показателей позволяет оценить производительность всех основных операций и сервисов ОС, которые, собственно, и влияют на работоспособность приложений.

5.2. Способы измерений показателей производительности

Основными способами измерения показателей производительности можно считать следующие:

- использование специальной внешней измерительной аппаратуры для считывания сигналов с выделенных для целей измерений портов целевого модуля, работающего под управлением целевой ОС;
- инструментирование исходного кода ОС путём вставки команд считывания текущего времени в начале и в конце участка кода,

выполняющего измеряемое действие;
- создание специальных приложений, которые вызовами интерфейсных системных сервисов заставляют тестируемую ОС выполнять (возможно, многократно) измеряемое действие.

Каждый из перечисленных способов обладает своими достоинствами и недостатками. Использование измерительной аппаратуры и инструментирование кода ОС обеспечивают самое точное измерение, т. к. позволяет исключить из последовательности действий ОС всё то, что не относится к измеряемому показателю. Однако выполнение дополнительных команд будет неизбежно вносить возмущения в работу компонентов ОС, что особенно недопустимо в случае ОСРВ. Использование внешней измерительной аппаратуры требует незначительного инструментирования кода ОС, однако возможности этого способа ограничены аппаратными возможностями целевого модуля. Кроме того, исходный код коммерческих ОС обычно недоступен. Поэтому мы считаем, что для решения задачи тестирования производительности ОС надо сосредоточиться на разработке специальных приложений-измерителей. Такой подход позволяет оценить показатели производительности «с точки зрения приложений». По этому пути идут исследователи и разработчики программных средств измерения показателей реального времени для специализированных окружений реального времени ядра ОС Linux (таких как RTAI, Xenomai) [43-47]. Главная задача при таком подходе — обеспечение предсказуемой точности измерений. Для решения этой задачи требуется разработать методики измерения каждого отдельного показателя. Эти методики зависят не только от содержания каждого показателя, но и от состава и функциональности интерфейсных сервисов ОС.

5.3. Программные средства измерения показателей производительности

В ИСП РАН разработан прототип инструмента, измеряющего показатели производительности ОСРВ, поддерживающих стандарт ARINC-653. Список показателей производительности сформирован на основе выделения ключевых функциональных блоков ARINC-653 и включает в себя:

- накладные расходы ОС, связанные со временем переключения между ARINC-разделами (процессами);
- накладные расходы ОС, связанные со временем переключения между ARINC-процессами (потоками управления);
- пропускная способность потоковых (queuing) каналов данных между ARINC-разделами;
- пропускная способность перезаписываемых (sampling) каналов данных между ARINC-разделами;

- пропускная способность потоковых (buffer) каналов данных между ARINC-процессами;
- пропускная способность перезаписываемых (blackboard) каналов данных между ARINC-процессами.

Разработанный инструмент создаёт сценарии использования ресурсов ОС и производит все измерения, используя только сервисы ОСРВ, предусмотренные стандартом ARINC-653. Таким способом эмулируется работа пользовательского приложения под управлением тестируемой ОСРВ и исключается вмешательство измерителя в работу компонентов ОС. Разработанный измерительный инструмент апробирован на ОСРВ ОС3000/ОС4000 (разработка НИИСИ РАН) и WindRiver VxWorks-653 и может быть использован для сравнительной оценки характеристик производительности разных ОСРВ на одной и той же аппаратной платформе.

6. Комплексный подход к тестированию ОС

6.1. Конфигурационное тестирование ОС

Существует ещё одна особенность ОС, которая значительно влияет на вопросы организации тестирования. А именно, ОС, как правило, обладают огромным набором конфигурационных параметров, регулирующих особенности поведения её компонентов, а также ОС предназначены для работы на разнообразном оборудовании, которое в свою очередь имеет многочисленные конфигурации. В результате возникает бесчисленное множество комбинаций, в которых может работать ОС, и встаёт вопрос о том, как можно протестировать эти комбинации адекватным образом.

Понятный ответ существует для ОС, применяемых в ответственных системах, от корректности поведения которых может зависеть жизни людей. В таких системах существует однозначно выбранная конфигурация оборудования и конфигурация ОС, которая и подлежит тщательному тестированию.

В случае ОС общего назначения разумным подходом является выделение наиболее важных конфигурационных опций и построение для них покрывающих множеств [48], чтобы сформировать ограниченное число комбинаций, в которых встретятся хотя бы все возможные пары значений каждой из пар опций. Но и число таких комбинаций всё равно оказывается достаточно велико, чтобы проводить полномасштабное тестирование для каждой из них. Тем не менее, упрощённое тестирование каждой комбинации имеет смысл провести. В качестве минимального варианта могут выступать простейшие тесты, подтверждающие возможность скомпилироваться и загрузиться в соответствующей комбинации. Для более детального тестирования возможно провести ручной отбор комбинаций, например, на основе дополнительных соображений о наиболее востребованных конфигурациях. Причём для каждого вида тестирования набор целевых конфигураций может подбираться индивидуально.

6.2. Комплексная система управления тестированием

В предыдущих разделах мы идентифицировали множество задач, возникающих при системном подходе к тестированию ОС, в том числе:

- идентификация компонентов ОС и предоставляемой ими функциональности;
- классификация компонентов ОС по их важности с целью определения целевого уровня качества их функционального тестирования:
  - тщательное тестирование;
  - среднее тестирование;
  - тестирование работоспособности;
  - отсутствие тестирования;
- выделение компонентов, для которых требуется проведение дополнительных тестов для выявления специфических видов ошибок;
- идентификация компонентов и их интерфейсов, к которым предъявляются требования по устойчивости к атакам и вредоносным действиям их контрагентов;
- определение требований к обеспечению обратной совместимости ОС и компонентов, для которых требуется проведение соответствующего тестирования;
- определение целевых показателей производительности ОС и методик их измерения;
- проектирование и разработка тестовых наборов и средств измерения показателей производительности в соответствии с целями, сформированными согласно вышеизложенным пунктам;
- выбор целевых комбинаций конфигураций ОС и аппаратного обеспечения для каждого из тестовых наборов.

Следующая задача, которая становится актуальной после появления работающих тестовых наборов, заключается в построении процесса выполнения тестов для каждой новой версии ОС. Наш опыт показывает, что с самого начала необходимо планировать разработку комплексной системы управления тестированием, которая будет решать следующие задачи:

- управление всеми тестовыми стендами (как реальными, так и виртуальными) и другими отведёнными аппаратными ресурсами;
- управление автоматическим выполнением тестовых наборов;
- сбор, обработка и визуализация информации о результатах тестирования;
- сравнительный анализ результатов тестирования по версиям ОС, по аппаратным платформам, по конфигурациям и т. д.;
- управление версиями ОС и тестовых наборов.

Первый опыт разработки такой среды в ИСП РАН воплотился в инструменте Linux Distribution Checker [49], который предназначен для проведения

тестирования дистрибутивов ОС Linux на соответствие требованиям стандарта LSB. Вследствие того, что основными объектами тестирования являлись системные библиотеки ОС и того, что инструмент предназначен для проведения тестирования в рамках сертификационной программы LSB, задачи управления тестовыми стендами и версиями ОС были не актуальными и не реализованы в Linux Distribution Checker. Более полный набор функциональности был реализован в специализированной системе управления лабораторией тестирования ОСРВ, которая включает в себя и поддержку управления аппаратными ресурсами, в том числе для проведения распределённого тестирования на нескольких машинах одновременно, и автоматическую установку новых версий ОС, и настройку аппаратуры тестовых стендов под целевую конфигурацию тестирования, и генерацию сравнительных отчётов, и визуализацию данных об измерении производительности ОС.

7. Заключение

Подводя итог, следует сделать вывод, что для решения задачи тестирования ОС в целом необходимо умело сочетать разные методы построения тестов, учитывая особенности объекта тестирования, возможности доступных техник и инструментов, а также аккуратно формируя планы в соответствии с целями проекта, имеющимися ресурсами и временны*м*и ограничениями.

В качестве ориентира можно использовать следующие усреднённые оценки трудоёмкости разработки тестов для одной условной интерфейсной функции целевого компонента. Тесты работоспособности при поддержки инструментами типа Azov или API Sanity Autotest могут разрабатываться со скоростью, позволяющей покрывать порядка 100 целевых функций в день усилиями одного инженера. Эта оценка дана в предположении однородности целевых функций. Если рассматривать целевые функции из различных малопохожих библиотек, то следует ожидать значительного падения производительности. Тесты среднего уровня качества с ожидаемым уровнем покрытия 70-80% строк исходного кода могут разрабатываться со скоростью порядка 2 функций в день. Разработка тестов высокого качества занимает порядка 5 дней на один аспект функциональности целевого компонента, что, по нашему опыту, в среднем соответствует одной функции в два дня. Также следует отметить, что разработка тестов высокого качества возможна лишь при наличии достаточно полной документации, иначе её приходится восстанавливать в ходе разработки тестов.

Дальнейшее развитие исследований в области тестирования ОС в ИСП РАН планируется по следующим направлениям:

- совмещение статических методов анализа программ и динамического тестирования, в частности:
  - исследование эффективности применения методов Concolic

тестирования к коду ядра ОС Linux на основе развития инструментов S2E, в первую очередь, на примере драйверов файловых систем;
    - поиск потенциальных гонок по данным при помощи методов статического анализа и целенаправленная проверка выдвинутых гипотез динамическими методами, такими как RaceHound;
- сравнительный анализ стратегий систематической симуляции сбоев на примере драйверов файловых систем ОС Linux;
- развитие методик и инструментов измерений производительности ключевых компонентов ОС;
- применение методов дедуктивной верификации для доказательства корректности наиболее ответственных компонентов ОС.

8. Благодарности

Авторы выражают признательность К. Власову, Р. Зыбину, А. Пономаренко, В. Рубанову, Е. Чернову и Е. Шатохину за активное участие в ряде описанных проектов. На работы по тестированию ОСРВ значительное влияние оказали соображения и советы сотрудников НИИСИ РАН и, в первую очередь, главного конструктора ОС2000/3000/4000 А.Н. Годунова. В разработке системы тестирования файловых систем ОС Linux активно участвовали сотрудники Лаборатории системного программирования Российско-Армянского Славянского университета. Кроме того, хотя описанные работы базируются на технологиях, созданных в ИСП РАН, многие работы ведутся как открытые проекты, и в развитие инструментов тестирования ОС вносят свой вклад не только сотрудники ИСП РАН, но другие участники этих проектов. Авторы также выражают им свою глубокую признательность.

Список литературы

[1]. В.В. Кулямин, Н.В. Пакулин, О.Л. Петренко, А.А. Сортов, А.В. Хорошилов. Формализация требований на практике. Препринт №13. М.: ИСП РАН, 2006.

[2]. В.В. Кулямин, А.К. Петренко, В.В. Рубанов, А.В. Хорошилов. Формализация интерфейсных стандартов и автоматическое построение тестов соответствия. «Информационные технологии», №8, 2007, С.1-8.

[3]. М.В. Екимов, И.В. Ковернинский, А.В. Хорошилов. “АРМ ПТ: создание системы проектирования тестов для критических систем на основе СПО”. Сборник докладов Всероссийской конференции “Свободное программное обеспечение – 2010”, Санкт-Петербург, 26-27 октября 2010 г.

[4]. Сайт инструмента Requality, http://forge.ispras.ru/projects/reqdb.

[5]. Д.Ю. Кичигин. Метод редукции тестового набора для регрессионного интеграционного тестирования. «Программирование», №5, 2009, С.57-69.

[6]. Alexey Khoroshilov, Vladimir Rubanov, Eugene Shatokhin. «Automated Formal Testing of C API Using T2C Framework». In Proceedings of the Third International Symposium «Leveraging Applications of Formal Methods, Verification and Validation» (ISoLA 2008), Porto Sani, Greece, October 13-15, 2008. pp.56-70. ISBN 978-3-540-88478-1.

[7]. TETWare User Guide, http://tetworks.opengroup.org/documents/3.7/uguide.pdf.
[8]. ISO/IEC 23360-1-8:2005, Linux Standard Base (LSB) Core Specification 3.1. Geneve: ISO, 2005.
[9]. А.В. Хорошилов. Linux Standard Base: история успеха?. Труды Института Системного Программирования РАН, Том 10, 2006. С.29-50.
[10]. Список ошибок, обнаруженных в ходе тестирования библиотек ОС Linux. http://linuxtesting.org/results/impl_reports.
[11]. Н.В.Пакулин, А.В.Хорошилов. Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов. Программирование, №6, 2007, С.26-55.
[12]. А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. “Подход UniTesK к разработке тестов: достижения и перспективы”. Труды Института системного программирования РАН, №5, 2004. С.121-156.
[13]. А.В.Хорошилов. Спецификация и тестирование систем с асинхронным интерфейсом. Препринт №12. М.: ИСП РАН, 2006.
[14]. IEEE 1003.1-2008. Information Technology — Portable Operating System Interface (POSIX). New York: IEEE, 2008.
[15]. ARINC. ARINC Specification 653P1-3: Avionics Application Software Standard Interface Part 1 - Required Services. Aeronautical Radio INC, Maryland, USA, 2010.
[16]. Alexey Grinevich, Alexey Khoroshilov, Victor Kuliamin, Denis Markovtsev, Alexander Petrenko, Vladimir Rubanov. “Formal Methods in Industrial Software Standards Enforcement”, PSI 2006, LNCS Vol. 4378, 2006, pp. 446-455.
[17]. A.Maksimov. Requirements-based conformance testing of ARINC 653 real-time operating systems // Proceedings of the Data Systems In Aerospace (DASIA 2010) conference, 2010, ESA SP-682.
[18]. Описание возможностей Fault Injection. https://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt.
[19]. Eugene Shatokhin. Using Dynamic Analysis To Hunt Down Problems in Kernel Modules. Presentation at LinuxCon Europe 2011, Czech Republic, Prague, 26-28 October 2011.
[20]. Сайт платформы KEDR. http://linuxtesting.org/kedr.
[21]. Проект по верификации модулей драйверов файловых систем. http://linuxtesting.org/spruce.
[22]. V. Kuliamin. Standardization and Testing of Mathematical Functions. // Proc. of PSI’2009, Novosibirsk, Russia, June 2009, LNCS 5947. pp. 257-268, Springer, 2009.
[23]. Р.С. Зыбин, В.В. Кулямин, А.В. Пономаренко, В.В. Рубанов, Е.С. Чернов. Автоматизация массового создания тестов работоспособности //Программирование, 34(6):64-80, 2008.
[24]. А.В. Пономаренко, В.В. Рубанов, А.В. Хорошилов. Автоматическая генерация тестов для C/C++ библиотек. Сборник докладов Седьмой конференции разработчиков свободных программ, Переславль, 26-27 июля 2010 г.
[25]. Сайт API Sanity Autotest. http://forge.ispras.ru/projects/api-sanity-autotest.
[26]. Инструмент LSB libchk. http://bzr.linuxfoundation.org/loggerhead/lsb/devel/misc-test/files.
[27]. Описание Android Compatibility Test Suite. http://source.android.com/compatibility.
[28]. chkshlib, http://osr507doc.sco.com/en/man/html.CP/chkshlib.CP.html.
[29]. cmpdylib, http://www.opensource.apple.com/source/cctools/cctools-795/man/cmpdylib.1.

[30]. cmpshlib, sys-admin.net/ebooks/unix3/mac/ch07_01.htm.
[31]. dpkg-gensymbols, http://man.he.net/man1/dpkg-gensymbols.
[32]. А. Пономаренко, В. Рубанов, А. Хорошилов. “Система анализа обратной бинарной совместимости библиотек Linux”. Сборник докладов международной конференции “Software Engineering Conference (Russia)”, SEC(R)-2009, сс. 25-31, Москва, 28-29 октября 2009 г.
[33]. В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. “Анализ типовых ошибок в драйверах операционной системы Linux”. Труды Института Системного Программирования, Том 22, 2012, с. 349-374. DOI: 10.15514/ISPRAS-2012-22-19.
[34]. Сайт инструмента Valgrind. valgrind.org.
[35]. Описание возможностей Kernel Memory Leak Detector. https://www.kernel.org/doc/Documentation/kmemleak.txt.
[36]. Описание возможностей Helgrind. http://valgrind.org/docs/manual/hg-manual.html.
[37]. Konstantin Serebryany, Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09). ACM, New York, NY, USA, pp.62-71.
[38]. Сайт проекта Kernel Strider. https://code.google.com/p/kernel-strider/.
[39]. Сайт проекта Race Hound. http://forge.ispras.ru/projects/race-hound.
[40]. Сайт проекта Trinity. http://codemonkey.org.uk/projects/trinity/.
[41]. И.К. Исаев, Д.В. Сидоров, А.Ю. Герасимов, М.К. Ермаков. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды Института Системного Программирования, Том 21, 2011, с. 55-70.
[42]. Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. ACM Trans. Comput. Syst. 30, 1, Article 2 (February 2012), pp.1-49.
[43]. A. Barbalace, A. Lunchetta, G. Manduchi, M. Moro, A. Soppelsa and C. Taliercio, “Performance Comparison of VxWorks, Linux, RTAI and XENOMAI in a Hard Real-time Application”, Proc. of Real-Time Conference 2007 15th IEEE-NPSS, (2007), pp. 1-5.
[44]. M. Franke, “A Quantitative Comparison of Realtime Linux Solutions”, Chemnitz University of Technology, (2007).
[45]. M. D. Marieska, A. I. Kistijantoro and M. Subair, “Analysis and Benchmarking Performance of Real Time Patch Linux and Xenomai in Serving a Real Time Application”, Proc. of International Conf. on Electrical Engineering and Informatics, (2011), pp. 1-6.
[46]. J. H. Koh and B. W. Choi, “Performance Evaluation of Real-time Mechanisms for Real-time Embedded Linux”, J. of Institute of Control, Robotics and Systems (in Korean), vol. 18, no. 4, (2012), pp. 337-342.
[47]. J. H. Koh and B. W. Choi, “Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions”, International Journal of Control and Automation, Vol. 6, No. 1, February, 2013, pp. 235-246.
[48]. В.В. Кулямин. Комбинаторная генерация программных конфигураций ОС. Труды Института Системного Программирования, Том 23, 2012, с. 359-370. DOI: 10.15514/ISPRAS-2012-23-20.
[49]. Vladimir Rubanov, Denis Silakov. Certification Infrastructure for the Linux Standard Base (LSB). //Proceedings of the second International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2008). Milan, Italy, 2008. pp. 79-88.

Testing of Operating Systems

Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V.
ISP RAS, Moscow, Russia
{gerlits, kuliamin, andrew, petrenko, khoroshilov, tsyvarev}@ispras.ru

Abstract. An operating system is a base stone of any computer system. Failures and bugs in an operating system impact the functionality of the system as a whole, that is why correctness and reliability of operating systems are so important. A variety of circumstances make verification and testing of operating systems a complicated issue. The list includes high dependence of operating systems on hardware, their massive internal concurrency, huge number of configuration options, required tolerance to aggressive actions of counteragents and hardware faults, a need for long continuous work without reboot, etc. Testing methods applied to operating systems include functional testing, backward compatibility testing, robustness testing, performance testing, configuration testing and others. Functional testing should be based on specifications of functional requirements to interfaces provided to applications and supported by test generation and coverage analysis tools. Model-based methods are very effective here. Backward compatibility testing includes both structural and semantic compatibility testing. Robustness testing includes detecting specific defects like memory leaks, data races and instability when processing incorrect data. Performance testing and benchmarking is of most importance for real-time operating systems. It includes definition of characteristics to be measured and development of benchmarking methods with predictable accuracy for each characteristic. Finally, composite approach to testing of operating systems is very important. The most appropriate testing method should be identified for each component or a subsystem of an operating system. An integrated testing control system helps making the testing process effective, especially when regression testing during operating system development and improvement.

Keywords: operating system, model-based testing, functional testing, robustness testing, performance testing

References

[1]. V.V. Kulyamin, N.V. Pakulin, O.L. Petrenko, A.A. Sortov, A.V. KHoroshilov. Formalizatsiya trebovanij na praktike [Requirements formalization in practice]. Preprinty ISP RAN [The preprints of ISP RAS], 2006, no. 13, pp. 1-70 (in Russian).

[2]. V.V. Kulyamin, A.K. Petrenko, V.V. Rubanov, A.V. KHoroshilov. Formalizatsiya interfejsnykh standartov i avtomaticheskoe postroenie testov sootvetstviya [Formalization of interface standards and automatic construction of conformance test

suites]. Informatsionnye tekhnologii [Information technologies], 2007, vol. 8, pp. 1-8 (in Russian).
[3]. M.V. Ekimov, I.V. Koverninskij, A.V. KHoroshilov. ARM PT: sozdanie sistemy proektirovaniya testov dlya kriticheskikh sistem na osnove SPO [Automated Working Place of Test Developer: developing a test design system for safety critical systems using free software]. Tezisy dokladov Vserossijskoj konferentsii "Svobodnoe programmnoe obespechenie" [The theses of All-Russian conferene "Free software – 2010"], 2010, pp. 64-65 (in Russian).
[4]. Requality tool, http://forge.ispras.ru/projects/reqdb.
[5]. D. Yu. Kichigin. A method of test-suite reduction for regression integration testing. Programming and Computer Software, 2009, vol. 35, no. 5, pp. 57-69. doi: 10.1134/S0361768809050041.
[6]. Alexey Khoroshilov, Vladimir Rubanov, Eugene Shatokhin. Automated Formal Testing of C API Using T2C Framework. The Proceedings of the Third International Symposium «Leveraging Applications of Formal Methods, Verification and Validation» (ISoLA 2008), 2008, pp. 56-70. doi: 10.1007/978-3-540-88479-8_5.
[7]. TETWare User Guide, http://tetworks.opengroup.org/documents/3.7/uguide.pdf.
[8]. ISO/IEC 23360-1-8:2005, Linux Standard Base (LSB) Core Specification 3.1. Geneve: ISO, 2005.
[9]. A.V. KHoroshilov. Linux Standard Base: istoriya uspekha? [Linux Standard Base: a success story?], Trudy ISP RAN [The Proceedings of ISP RAS], 2006, vol. 10, pp. 29-50 (in Russian).
[10]. List of errors found during testing of LINUX libraries, http://linuxtesting.org/results/impl_reports.
[11]. N.V. Pakulin, A.V. Khoroshilov. Development of formal models and conformance testing for systems with asynchronous interfaces and telecommunications protocols. Programming and Computer Software, 2007, vol. 33, no. 6, pp. 316-335.
[12]. A.V. Barantsev, I.B. Burdonov, A.V. Demakov, S.V. Zelenov, A.S. Kosachev, V.V. Kulyamin, V.A. Omel'chenko, N.V. Pakulin, A.K. Petrenko, A.V. KHoroshilov. Podkhod UniTesK k razrabotke testov: dostizheniya i perspektivy [UniTESK approach to test development: achievements and prospects]. Trudy ISP RAN [The Proceedings of ISP RAS], 2004, vol. 5, pp. 121-156 (in Russian).
[13]. A.V. KHoroshilov. Spetsifikatsiya i testirovanie sistem s asinkhronnym interfejsom [Specification and testing of systems with asynchronous interface]. Preprinty ISP RAN [The preprints of ISP RAS], 2006, no. 12, pp. 1-140 (in Russian).
[14]. IEEE 1003.1-2008, Information Technology — Portable Operating System Interface (POSIX). New York: IEEE, 2008.
[15]. ARINC Specification 653P1-3: Avionics Application Software Standard Interface Part 1 - Required Services. Aeronautical Radio INC, Maryland, USA, 2010.
[16]. Alexey Grinevich, Alexey Khoroshilov, Victor Kuliamin, Denis Markovtsev, Alexander Petrenko, Vladimir Rubanov. Formal Methods in Industrial Software Standards Enforcement. The proceedings of the 6th international Andrei Ershov memorial conference on Perspectives of systems informatics, 2006, vol. 4378 of LNCS, pp. 446-455. doi: 10.1007/978-3-540-70881-0_41.
[17]. A. Maksimov. Requirements-based conformance testing of ARINC 653 real-time operating systems. The Proceedings of the Data Systems In Aerospace conference (DASIA 2010), 2010.
[18]. Fault Injection technique, https://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt.

[19]. Eugene Shatokhin. Using Dynamic Analysis To Hunt Down Problems in Kernel Modules. Presentation at LinuxCon Europe 2011, Czech Republic, Prague, 26-28 October 2011.
[20]. KEDR framework, http://linuxtesting.org/kedr.
[21]. Linux File System Verification (Spruce) project, http://linuxtesting.org/spruce.
[22]. V. Kuliamin. Standardization and Testing of Mathematical Functions. The Proceedings of 9th international Andrei Ershov memorial conference on Perspectives of systems informatics, 2009, vol. 5947 of LNCS, pp. 257-268. doi: 10.1007/978-3-642-11486-1_22.
[23]. R.S. Zybin, Victor V. Kuliamin, Andrey V. Ponomarenko, Vladimir V. Rubanov, E.S. Chernov. Automation of broad sanity test generation. Programming and Computer Software, 2008, vol. 34, no. 6, pp. 351-363. doi: 10.1134/S0361768808060066.
[24]. A.V. KHoroshilov. Avtomaticheskaya generatsiya testov dlya C/C++ bibliotek [Automatic test suites generation for C/C++ libraries]. Tezisy dokladov Sed'moj konferentsii razrabotchikov svobodnykh programm [The theses of 7th conference of free software developers], 2010, pp. 75-78 (in Russian).
[25]. API Sanity Autotest tool, http://forge.ispras.ru/projects/api-sanity-autotest.
[26]. LSB libchk tool, http://bzr.linuxfoundation.org/loggerhead/lsb/devel/misc-test/files.
[27]. Android Compatibility Test Suite, http://source.android.com/compatibility.
[28]. chkshlib, http://osr507doc.sco.com/en/man/html.CP/chkshlib.CP.html.
[29]. cmpdylib, http://www.opensource.apple.com/source/cctools/cctools-795/man/cmpdylib.1.
[30]. cmpshlib, sys-admin.net/ebooks/unix3/mac/ch07_01.htm.
[31]. dpkg-gensymbols, http://man.he.net/man1/dpkg-gensymbols.
[32]. A. Ponomarenko, V. Rubanov, A. Khoroshilov. Sistema analiza obratnoj binarnoj sovmestimosti bibliotek Linux [System for backward compatibility analysis of LINUX libraries]. The Proceedings of International conference “Software Engineering Conference (Russia)”, SEC(R)-2009, 2009, pp. 25-31 (in Russian).
[33]. V.S. Mutilin, E.M. Novikov, A.V. KHoroshilov. Analiz tipovykh oshibok v drajverakh operatsionnoj sistemy Linux [Analysis of typical errors in LINUX drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 349-374. DOI: 10.15514/ISPRAS-2012-22-19. (in Russian).
[34]. Valgrind tool, valgrind.org.
[35]. Kernel Memory Leak Detector, https://www.kernel.org/doc/Documentation/kmemleak.txt.
[36]. Helgrind tool, http://valgrind.org/docs/manual/hg-manual.html.
[37]. Konstantin Serebryany, Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. The Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09), 2009, pp.62-71. doi: 10.1145/1791194.1791203.
[38]. Kernel Strider project, https://code.google.com/p/kernel-strider/.
[39]. Race Hound project, http://forge.ispras.ru/projects/race-hound.
[40]. Trinity project, http://codemonkey.org.uk/projects/trinity/.
[41]. I.K. Isaev, D.V. Sidorov, A.YU. Gerasimov, M.K. Ermakov. Avalanche: Primenenie dinamicheskogo analiza dlya avtomaticheskogo obnaruzheniya oshibok v programmakh ispol'zuyushhikh setevye sokety [Avalanche: Dynamic analysis for automatic error detection in programs working with network sockets]. The Proceedings of Real-Time Conference, 2007 15th IEEE-NPSS, 2011, vol. 21, pp. 55-70 (in Russian).

[42]. Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems, 2012, vol. 30, issue 1, article 2, pp.1-49. doi: 10.1145/2110356.2110358.
[43]. A. Barbalace, A. Lunchetta, G. Manduchi, M. Moro, A. Soppelsa and C. Taliercio. Performance Comparison of VxWorks, Linux, RTAI and XENOMAI in a Hard Real-time Application. The Proceedings of Real-Time Conference, 2007 15th IEEE-NPSS, 2007, pp. 1-5. doi: 10.1109/RTC.2007.4382787.
[44]. M. Franke. A Quantitative Comparison of Realtime Linux Solutions. Chemnitz University of Technology, 2007.
[45]. M. D. Marieska, A. I. Kistijantoro and M. Subair. Analysis and Benchmarking Performance of Real Time Patch Linux and Xenomai in Serving a Real Time Application. The Proceedings of International conference on Electrical Engineering and Informatics, 2011, pp. 1-6.
[46]. J. H. Koh and B. W. Choi. Performance Evaluation of Real-time Mechanisms for Real-time Embedded Linux. Journal of Institute of Control, Robotics and Systems, 2012, vol. 18, no. 4, pp. 337-342 (in Korean). doi: 10.5302/J.ICROS.2012.18.4.337.
[47]. J. H. Koh and B. W. Choi. Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions. International Journal of Control and Automation, 2013, vol. 6, no. 1, pp. 235-246.
[48]. V.V. Kulyamin. Kombinatornaya generatsiya programmnykh konfiguratsij OS [Combinatorial generation of software-based OS configurations]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 359-370. DOI: 10.15514/ISPRAS-2012-23-20. (in Russian).
[49]. Vladimir Rubanov, Denis Silakov. Certification Infrastructure for the Linux Standard Base (LSB). The Proceedings of the second International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2008), 2008. pp. 79-88.

Автоматизация тестирования соответствия для телекоммуникационных протоколов.

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

Аннотация. В данной статье обобщается опыт разработки тестовых наборов для тестирования соответствия реализаций спецификациям протоколов Интернета. Во всех проектах, представленных в статье, использовалась технология UniTESK в качестве базы для построения тестов. В ходе разработки тестовых наборов были выявлены особенности протоколов, затрудняющие тестирование реализаций с помощью технологии UniTESK, а также особенности инструментов, поддерживающих ее работу, однако все эти особенности удавалось успешно преодолеть, не выходя за рамки ее ограничений.

Ключевые слова: тестирование соответствия, UniTESK, формальные методы, автоматизация тестирования, тестирование, основанное на моделях, тестирование протоколов, формальные спецификации, моделирование протоколов

1. Введение

Создание распределенных систем, в которых различные функции, связанные с хранением и обработкой информации, взаимодействием с пользователем и т. п., распределены по различным компьютерным системам, требует организации обмена информацией между компонентами такой системы. Для успешного взаимодействия между компьютерными системами их коммуникации должны основываться на хорошо определенных правилах. Совокупность таких правил, включая представление данных для передачи, правила отправки данных одной компьютерной системой и правила приема данных другой компьютерной системой, формируют *протокол* передачи данных.

Реализация протокола — это программный (в некоторых случаях программно-аппаратный) компонент компьютерной системы, который реализует конкретные процедуры отправки и приема данных. Для того чтобы реализации протоколов разных производителей компьютерных систем одинаково интерпретировали правила необходимо, чтобы этот набор правил был зафиксирован в виде *спецификации*. Как правило, спецификация представляет собой текстовый документ на естественном языке, хотя предпринимались попытки внедрить подходы к разработке спецификаций на

языках с формализованной семантикой (SDL[1], LOTOS[2], Estelle[3], ASN.1[4], UML[5] и др.) В некоторых областях эти попытки преуспели, и были разработаны спецификации-стандарты протоколов на формальных языках, но для большинства протоколов формальные нотации не использовались. Например, спецификации протоколов Интернета публикуются исключительно в виде текстов на английском языке.

Однако, для того чтобы две системы могли успешно обмениваться информацией по некоторому протоколу, одного наличия спецификации протокола недостаточно. Необходимо, чтобы реализации протоколов, участвующих в обмене информацией, функционировали в соответствии со специфицированными правилами, то есть удовлетворяли спецификации протокола. Разумеется, это требование базируется на предположении о корректности спецификации протокола — отсутствии ошибок формата данных, блокировок, зацикливаний и иных дефектов, которые могут привести к тому, что даже реализации, удовлетворяющие спецификации на 100%, не смогут осуществлять обмен данными. В данной статье мы не будем касаться вопросов проверки собственно спецификаций на корректность. Можно указать книги [6], [7], [8], в которых представлены методы анализа непосредственно протоколов.

Для установления соответствия протокола стандарту одним из наиболее широко используемых подходов является тестирование. В общем случае, тестирование — это процесс экспериментального исследования поведения реализации. Такое исследование, как правило, проводится в специально подготовленном окружении, которое имитирует нормальные или аномальные сценарии взаимодействия с испытываемой реализацией. Тестирование, нацеленное на проверку соответствия реализации требованиям спецификации, называется *тестированием соответствия* (conformance testing). При таком тестировании специально подготовленные данные подаются на вход испытываемой *целевой* системы (system under test, SUT), результаты обработки этих данных в целевой системе собираются и выносится *вердикт* — собранные результаты проверяются на соответствие эталонным (ожидаемым) результатам, или вычисляется степень их соответствия требованиям.

Обычно терминологически верное, но слишком длинное словосочетание «тестирование соответствия реализаций протокола спецификации этого протокола» сокращают до «тестирования соответствия протокола», несмотря на то, что тестируется не протокол (как некоторый концептуальный объект), а его реализации. В данной статье в дальнейшем «*тестирование соответствия протокола*» будет пониматься именно как тестирование реализаций на соответствие спецификации. В тех случаях, когда из контекста ясно, что речь идет о протоколах, будет использоваться минимальная форма «*тестирование соответствия*».

Очевидно, что тестирование может проводиться только в течение ограниченного периода времени. Для систем, используемых на практике, это означает, что невозможно провести испытания реализации во всех мыслимых ситуациях. Поэтому для тестирования очень важной является задача систематического построения тестовых воздействий.

Традиционный подход к тестированию соответствия заключается в разработке небольших специализированных программ, проверяющих отдельные требования — *элементарных тестов* (test cases). Каждый тест выполняется независимо от остальных тестов. Все тесты начинаются в одном и том же начальном состоянии реализации, затем каждый тест, в зависимости от своей цели, оказывает воздействия на целевую систему, чтобы привести её в желаемое состояние. Если такое состояние достигнуто, тест выполняет собственно тестовую процедуру, после чего возвращает целевую систему в начальное состояние. Более подробно этот подход к тестированию соответствия рассматривается в разделе 2.1.

Разработка тестов вручную широко распространена в современной телекоммуникационной индустрии, так как, во-первых, такой подход не требует специальных знаний (формальных нотаций, конечных автоматов, темпоральных логик и т. п.), и, во-вторых, тесты появляются сразу после начала процесса разработки тестового набора, что позволяет немедленно приступить к поиску ошибок.

Однако у ручной разработки тестов есть существенные недостатки. Высокая сложность современных протоколов требует, чтобы тестовый набор состоял из большого числа испытаний. Так, тестовый набор для сетевого протокола IPv6 насчитывает более 6000 тестов. Отсутствие автоматизации делает разработку тестового набора задачей, превосходящей по сложности собственно разработку реализации. Вердикты о корректности наблюдаемого поведения не опираются на строгую модель протокола, что затрудняет проведение инспекций тестового набора. Покрытие требований оценивается эвристически, без привлечения строгих формальных процедур.

Элементарные испытания рассматриваются в академической среде как один из традиционных методов тестирования, в то время как тестирование на основе моделей рассматривается как новый метод, решающий множество неразрешимых с помощью традиционных методов проблем [9-11]. Ряд походов к автоматизации тестирования соответствия кратко рассмотрен в разделе 2.2.

В разделе 3 данной статьи представлен подход к автоматизации тестирования соответствия, развиваемый авторами. Этот подход основан на использовании технологии тестирования UniTESK[12-14] и включает в себя разработку формальной модели протокола и разработку тестов как конечных автоматов, чье состояние строится из состояния модели протокола. В разделе 4 кратко описываются проекты по созданию тестовых наборов для различных протоколов в рамках рассматриваемого подхода. В разделе 5 рассматривается

эволюция методов и средств, использовавшихся авторами при разработке тестовых наборов.

2. Обзор существующих методов тестирования соответствия

Исследования и практические проекты по установлению соответствия для протоколов стали появляться еще в 60-х годах прошлого века. Наиболее значительные результаты были получены прежде всего в теоретических исследованиях — были поставлены и решены задачи установления соответствия между двумя моделями, если модели представимы в виде конечных автоматов, разработан ряд методов построения тестов, гарантирующих обнаружение всех ошибок определенного класса [15-17].

По мере развития программных систем становилось очевидно, что конечные автоматы позволяют описывать только часть функциональности взаимодействующих систем. Были разработаны более выразительные средства для формального представления взаимодействующих программно-аппаратных систем: алгебры взаимодействующих процессов [18-20] и системы размеченных переходов[21-22]. На базе этих средств были получены новые результаты по формализации семантики протоколов, позволяющие более точно моделировать асинхронные взаимодействия.

Всплеск в области тестирования соответствия произошел в конце 80-х — начале 90-х годов XX столетия. Он был вызван разработкой и стандартизацией в середине 80-х годов стека протоколов OSI [23], который был призван объединить существующие на тот момент сетевые технологии в единую систему. В рамках процессов разработки и публикации стандартов были созданы и стандартизованы языки формальных спецификаций, предназначенные для описания протоколов на основе моделей в виде взаимодействующих автоматов или взаимодействующих процессов — SDL [1], Estelle [2] и LOTOS [3], опубликовано большое количество работ, посвященных разнообразным методам тестирования, основанным на конечных автоматах и размеченных системах переходов, например, [24-30]. Кроме того, в указанный период проводится активная систематизация исследований в этой области [31-34], разрабатываются общие схемы тестирования, и на их основе принимаются базовые стандарты, регулирующие проведение тестирования реализаций протоколов — ISO 9646 [35] ITU-T/Z.500 [36].

В начале 90-х годов созданные ранее теоретические конструкции начинают активно применять на практике для разработки и проверки корректности конкретных протоколов. Тогда же появляются регулярные конференции по данной тематике, например, Международный семинар по спецификации, тестированию и верификации протоколов (International Workshop on Protocol Specification, Testing and Verification).

2.1. Общие вопросы тестирования соответствия

Традиционно [37] разделяются два подхода к построению тестов: подход «белого ящика» и подход «черного ящика». Тестирование «белого ящика», иногда называемое *структурным* тестированием [38] ориентируется на внутреннюю структуру тестируемых программ. Цель такого тестирования — исследовать программный код, в частности, вызвать при тестировании все возможные ветвления или (как максимум) пройти все пути исполнения в программе. Тесты могут строиться на основании различных элементов исходных текстов программы (ветвлений, вызовов подпрограмм, строк кода), для построения тестов может использоваться также структура бинарного кода оттранслированной программы.

При тестировании методом «черного ящика» внутренняя структура реализации не известна, либо не используется при построении тестовых данных и вынесении вердикта. Основная цель при тестировании «черного ящика» заключается в том, чтобы установить, что целевая система корректна с точки зрения внешнего наблюдателя, который может судить об этом только на основании сопоставления внешне наблюдаемого поведения реализации и требований спецификации к этому поведению.

В подавляющем большинстве случаев тестирование соответствия протоколов спецификациям реализуется как тестирование черного ящика. Можно указать несколько причин:

- Спецификации многих протоколов явно указывают, что структуры данных для внутренних состояний, алгоритмы обработки данных и другие ненаблюдаемые извне аспекты функциональности, используются в спецификации как *концептуальные* для описания внешне наблюдаемого поведения. От реализаций не требуется использовать именно эти структуры или алгоритмы при условии, что внешне наблюдаемое поведение реализаций будет таким, как если бы реализации их использовали.

- Зачастую спецификации протоколов являются международными или общепринятыми стандартами. В таком случае тестирование соответствия приобретает качество сертификационного тестирования, при котором одни и те же тесты используются для различных реализаций. Это означает, что тесты не могут содержать зависимости от реализации, т. е. тестировать «белый ящик».

- Тесты не имеют доступа к внутреннему устройству целевой реализации, так как выполняются на отдельной *инструментальной* машине, которая физически отделена от узла сети, на котором установлена целевая реализация. Более того, при тестировании на встроенных системах нет практической возможности установить какое-либо дополнительное ПО для инспектирования внутреннего состояния целевой реализации.

Набор тестов для тестирования соответствия должен обладать рядом свойств:

1. Прослеживаемостью требований. Тесты должны соотноситься с требованиями стандарта, должно быть наглядно видно, какие требования какими тестами покрываются.

2. Многообразием настроек на особенности реализаций (MAY, SHOULD, MUST и другие). Должна быть опция для определения множества требований, поддерживаемых тестируемой реализацией. В это множество не должны попадать требования, не поддерживаемые тестируемой реализацией.

3. Полнотой тестового набора в смысле покрытия требований. Полученный тестовый набор должен покрывать как минимум все функциональные требования, помеченные в стандарте как обязательные для каждой реализации.

2.1 Неавтоматизированная разработка тестов для тестирования соответствия

Разработчики стека протоколов OSI [23], в настоящее время известного каждому студенту как «семиуровневая модель», одновременно с разработкой протоколов озаботились вопросом обеспечения совместимости реализаций. Была разработана и закреплена в стандарте ISO 9646[36] универсальная методика тестирования реализаций протоколов. Кратко рассмотрим основные идеи этой методики.

1. Тестовый набор состоит из переносимых, абстрактных, независящих от конкретной реализации протокола, абстрактных тестовых наборов. Абстрактный тестовый набор состоит из отдельных тестов, заданных в однозначной, не зависящей от конкретной реализации форме.

2. Абстрактный тестовый набор состоит из абстрактных элементарных испытаний (abstract test cases), которые записаны в нотации, не зависящей от конкретной реализации, и взаимодействуют с целевой реализацией через набор коммуникационных каналов.

a) В каждом элементарном испытании реализуется последовательность тестовых воздействий на целевую систему и вынесение вердикта о корректности наблюдаемого поведения целевой системы.

b) Для каждого элементарного испытания явно или неявно задаётся цель тестирования (test purpose), которая неформально определяет группу функциональных требований к реализации протокола. Успешное или неуспешное завершение элементарного испытания трактуется следующим

образом: реализация протокола корректно или некорректно реализует требования, соответствующие цели тестирования.

Подход, основанный на абстрактных представлениях тестов, позволил выработать строгие методы тестирования протоколов, четко определяющие различные компоненты тестового набора, их функции и степень зависимости от конкретной тестируемой реализации. Этот подход был закреплён в международном стандарте ISO/IEC 9646 [35,39].

Каждое абстрактное элементарное испытание состоит из:

- преамбулы – последовательности шагов, приводящих реализацию из начального состояния в целевое состояние,
- множества тестовых шагов, которые проверяют поведение реализации в целевом состоянии, и
- постамбулы, переводящей реализацию обратно в начальное состояние.

На каждом шаге испытания осуществляется проверка правильности поведения целевой системы. В частности, расхождение наблюдаемого поведения с ожидаемым в преамбуле делает бессмысленным дальнейшее выполнение теста и, как правило, приводит к остановке выполнения с негативным вердиктом.

Каждое элементарное испытание должно завершиться вынесением вердикта. Все действия элементарного испытания описываются абстрактно, в терминах, не зависящих от реализации. Тем самым абстрактный тестовый набор представляет собой однозначную и не зависящую от конкретной реализации спецификацию тестовой последовательности.

Для представления абстрактных тестовых наборов используются языки TTCN-1 [40], TTCN-2 [41] и TTCN-3 [42], графическая нотация UML Testing Profile [43], разрабатываются представления в виде диалектов XML [44], а также непосредственно языки программирования – Java [45], Perl [46].

В традиционных подходах, где тесты разрабатываются вручную, построение конкретных тестов из множества возможных основывается на *целях тестирования* (test purposes). Цель тестирования — это класс ситуаций, соответствующих конкретному функциональному требованию, в которых реализация должна вести себя сходным образом. Например, ответ на определенное сообщение зависит от конкретных данных в запросе, однако строится по одному алгоритму. Если таких алгоритмов несколько (например, один из них — построение сообщения об ошибке в запросе), то необходимо проверить поведение реализации в каждом из этих сценариев. Таким образом, одному функциональному требованию могут соответствовать несколько тестовых ситуаций, поэтому перед разработкой тестов для требований выписываются перечни ситуаций, которые необходимо проверить. Эти

перечни тестовых ситуаций оформляются в виде целей тестирования и позже программируются на выбранном языке описания тестов. Правда, необходимо заметить, что далеко не во всех проектах по тестированию соответствия выписывают цели тестирования явным образом, а там, где они выписываются, затруднительно проверить полноту выделенного набора тестовых ситуаций.

Достоинством указанной методологии является то, что спецификация тестовых последовательностей не зависит от конкретной реализации. В качестве недостатков можно отметить:

- методология абстрактных тестов, формализованная в серии стандартов ISO 9646, не предполагает автоматизированного построения элементарных испытаний;
- в методологии нет формальной процедуры для оценки полноты тестирования;
- методология не может служить надёжным базисом для анализа протокола, так как не включает разработку формальной модели целевого протокола;
- все элементарные испытания должны разрабатываться экспертами в целевой области, так как в теле каждого элементарного испытания должен выноситься вердикт о корректности поведения целевой системы.

Вообще говоря, подход создания тестового набора, включающего отдельные, разрабатываемые вручную небольшие тесты, позволяет строить тестовые наборы для любых протоколов. Однако, оборотной стороной является значительные затраты ресурсов на разработку такого тестового набора. При разработке абстрактных элементарных тестов для сложных протоколов рано или поздно тестировщики сталкиваются с существенными трудностями:

1. Вердикты о соответствии поведения реализации спецификации протокола выносятся в каждом тесте независимо, поэтому проверки корректности в тестах часто дублируются, что усложняет поддержку тестового набора, его расширение и модификацию.

2. Вердикты выносятся на основе ожидаемого поведения. Обработка «неожиданных сообщений», характерных для недетерминированных систем и параллельно взаимодействующих компонентов, требует дополнительных усилий при разработке тестового набора.

Распространённые индустриальные технологии тестирования, основанные на методологии элементарных тестов, не поддерживают использование формальных моделей при разработке и прогоне тестов. Цикл разработки тестового набора не пересекается с разработкой формальной модели системы. Формальные модели не включаются в состав тестового набора. Проверки корректности поведения реализации в тестах должны основываться на требованиях, изложенных в регламентирующих документах, а формальные спецификации, как правило, не входят в нормативные разделы стандартов, поэтому создатели тестовых наборов в процессе разработки обращаются

непосредственно к тексту стандарта, а не к формальным моделям. Тесты не опираются на строгую модель целевой системы при проведении проверок и определении достигнутого уровня тестирования.

2.2. Методы автоматизации тестирования соответствия

Высокая сложность современных протоколов требует, чтобы тестовый набор состоял из большого числа испытаний. При разработке тестового набора без формальной модели возникают следующие трудности:

1. Отсутствие автоматизации делает разработку тестового набора задачей, превосходящей по сложности собственно разработку реализации. Наличие формальной спецификации позволяет автоматизировать процедуру генерации тестовых воздействий и минимизировать ручной труд при разработке тестов.

2. Вердикты о корректности наблюдаемого поведения не опираются на строгую модель протокола, что затрудняет проведение инспекций тестового набора. Наличие модели, во-первых, дает возможность обеспечить валидацию корректности тестового набора, так как для этого достаточно проверить корректность модели, а во-вторых, позволяет многократно использовать модель для проверки правильности поведения реализаций, уменьшая тем самым размер тестового набора.

3. Из-за отсутствия формальной модели протокола нет строгой процедуры для оценки покрытия функций протокола тестовым набором. Наличие формальной модели протокола позволяет однозначно задавать связи между требованиями и тестами, автоматически отслеживать качество тестирования в терминах покрытия спецификации.

Для формального описания протоколов было разработано множество нотаций. Некоторые из них получили широкое распространение в индустрии, и для них были изданы стандарты международных организаций. Разработка формальных спецификаций телекоммуникационных протоколов позволяет дать ясное, однозначное и полное описание протокола, выявить ошибки и неполноту текстовой спецификации протокола. Формальная спецификация включена как нормативная или информативная часть в ряд стандартов телекоммуникационных протоколов, например GSM Handover procedures [47] или Harmonized Programmable Communication Interface (HPCI) for ISDN [48].

Задачу тестирования соответствия спецификации протокола можно рассматривать как задачу сравнения двух расширенных конечных автоматов. В обзорной работе Д.Ли и М.Янакакиса[32], посвящённой вопросам тестирования конечных автоматов, представлены алгоритмы и подходы к построению тестовых последовательностей для проверки соответствия двух конечных автоматов. Главная проблема проверки соответствия двух автоматов заключается в том, что с ростом числа состояний в автоматах и/или числа переходов длина тестовых последовательностей растёт очень быстро.

Для уменьшения длины тестовых последовательностей используется целенаправленная генерация тестов, при которой обход автомата ограничивается специальными условиями. Примеры реализации целенаправленной генерации есть в инструментах TGV [49], Gotcha-TCBEANS[50,51], SpecExplorer[52]. Но и при целенаправленной генерации тестов остаётся проблема недетерминизма протокола или реализации. Суть проблемы заключается в том, что если тесты генерируются заранее, до исполнения, то для недетерминированного протокола необходимо генерировать все возможные цепочки допустимых переходов, что может привести к значительному увеличению числа тестовых последовательностей.

Для решения этой проблемы был предложен подход к генерации тестов, получивший название «динамической генерации» (on the fly), при котором тестовые воздействия строятся непосредственно во время исполнения тестового набора. Как правило, тестовые воздействия генерируются при обходе исполнимой и, возможно, недетерминированной модели протокола. При этом тестовые воздействия подаются одновременно на целевую реализацию и на модель протокола, а реакции целевой реализации передаются в модель. Это необходимо для того, чтобы состояние модели обновлялось в соответствии с правилами протокола. Передача реакций позволяет обновлять состояние модели в случае недетерминированных протоколов. Например, спецификация SMTP - протокола отправки электронной почты — допускает при обработке запроса клиента вернуть код ответа, обозначающий внутреннюю ошибку сервера или нехватку ресурсов и разорвать соединение. Соответственно, для корректного построения следующего состояния модели необходимо знать код ответа, который вернул сервер.

Основные трудности, которые возникают при практическом использовании динамической генерации, связаны с определением подмножества модели, подлежащего обходу. Для протоколов с большим числом состояний полный обход модели за приемлемое время невозможен, поэтому были разработаны различные методы сокращения обхода. В следующем разделе мы кратко обсудим методы сокращения обхода, применявшиеся в проектах по тестированию соответствия протоколов с использованием технологии UniTESK.

Для тестирования протоколов с большим числом состояний и переходов (исчисляемых тысячами), отдельные авторы предлагают использовать случайный обход явной модели протокола. Этот подход был разработан и реализован под руководством Я.Третманса в инструменте тестирования TorX [53,54] совместно с динамической генерацией тестов. Было показано [55], что если реализация некорректна, то вероятность обнаружить ошибку стремится к 1 при стремлении числа тестов к бесконечности. Оценки числа тестов, которые с заданной вероятностью находят ошибку, не приводятся. Инструмент TorX использовался для тестирования ряда простых протоколов, но к настоящему времени результаты применения к сложным промышленным протоколам или протоколам Интернета не известны.

3. Применение технологии UniTESK к тестированию соответствия

В данной статье обобщается опыт разработки тестовых наборов для тестирования соответствия протоколов Интернета. Все проекты, представленные в статье, использовали технологию UniTESK [12-14] в качестве базы для построения тестов. В этом разделе мы обсудим общие вопросы тестирования соответствия с использованием UniTESK, а сами проекты и их результаты представим в следующем разделе.

Технология UniTESK поддерживает нотацию формальных спецификаций, автоматическую динамическую генерацию тестовых воздействий (on-the-fly) и автоматический анализ результатов. UniTESK предоставляет средства для формальной спецификации протоколов [56] в виде контрактных спецификаций переходов расширенного конечного автомата [57-59].

Как уже отмечалось, спецификации многих современных протоколов публикуются на естественном языке без использования формальных нотаций. Для применения модели в тестировании соответствия необходимо трансформировать *неформальную* спецификацию (на естественном языке) в *формальную*, основанную на машино-читаемой нотации. Однако, одной лишь машино-читаемой формальной нотации недостаточно. Формальная спецификация может использоваться для тестирования соответствия при выполнении ряда требований:

1. однозначность представления спецификации: эксперты в данной предметной области одинаковым образом трактуют спецификацию;
2. проверяемость требований: для каждого требования в спецификации эксперты в предметной области могут предложить процедуру проверки;
3. адекватность исходным текстовым спецификациям: требования в разработанной спецификации соответствуют требованиям, представленным в исходной спецификации;
4. полнота представления исходной спецификации: все требования исходной текстовой спецификации представлены в разработанной спецификации;
5. прослеживаемость исходной спецификации: есть строго определѐнная процедура, при помощи которой можно для каждого требования из исходной спецификации определить, как оно представлено в разработанной спецификации и наоборот, для каждого логического или исполнимого выражения модели найти исходные требования, представленные им;
6. Пригодность для тестирования соответствия: спецификацию можно использовать для автоматизации тестирования соответствия. Это, в свою очередь, подразумевает:

a) возможность генерировать оракулы из спецификации;

b) возможность генерировать тестовые данные из спецификации;

c) наличие инструментальной поддержки генерации оракулов и

тестовых данных из спецификации.

В технологии UniTESK требования к поведению формализуются в виде спецификаций контракта между участниками протокола. Контрактные спецификации развивают идеи Т.Хора[60] о спецификации поведения при помощи логических ограничений (пред- и постусловий). При описании внешне наблюдаемого поведения системы контрактными спецификациями взаимодействие системы с окружением представляется как набор операций в некотором формальном интерфейсе. С каждой операцией формального интерфейса связаны предусловие и постусловие. Предусловие операции накладывает ограничения на ситуации, в которых эта операция может быть вызвана, постусловие операции накладывает ограничение на результаты операции и изменение состояния системы после выполнения операции.

Контрактные спецификации UniTESK удовлетворяют приведенным выше требованиям к формальной спецификации. Действительно, контрактные спецификации являются однозначными, так как записываются средствами строго определённой нотации. Далее, пригодность контрактных спецификаций для тестирования определяется тем, что из контрактных спецификаций можно построить оракул для проверки корректности внешне наблюдаемого поведения, и есть инструментальная поддержка UniTESK для тестирования на основе контрактных спецификаций.

Рассмотрим достоинства контрактных спецификаций в контексте анализа и формализации функциональных требований и тестирования протоколов:

1. Контрактные спецификации позволяют сравнительно легко описывать различные виды недетерминированного поведения, в том числе вызванные неполнотой исходной спецификации или особенностями реализации.

2. Существуют эффективные способы автоматического построения тестовых оракулов из контрактных спецификаций [12]. Это позволяет автоматизировать процесс проверки соответствия реализации требованиям.

3. Создание контрактных спецификаций требует от разработчика мышления в стиле требований к результату, что отличается от характерных стилей мышления разработчиков программных систем. Это позволяет выявить особенности требований, которые остались бы незамеченными при анализе требований средствами прототипирования или разработки экспериментальных реализаций. В частности, мы считаем, что контрактная спецификация позволяет лучше анализировать требования к недетерминированному поведению, чем явные исполнимые спецификации или прототипы.

Условимся называть элементарные акты взаимодействия реализации протокола с окружением *событиями*. Примерами событий, связанных с протоколами, могут служить

- операции, входящие в программный интерфейс реализации (как правило, операции по отправке или получению данных протоколов более высокого уровня, настройка параметров протокола, аудит внутренних событий); возникновение события состоит в приеме сообщения реализацией

протокола;

- обработка входящих сообщений протокола; возникновение события состоит в приеме сообщения реализацией протокола;
- или отправка сообщений; возникновение события – выход сообщения в сеть или передача его протоколу нижнего уровня.

Обращение к реализации протокола со стороны вышележащего уровня или приложения естественно представляется как процедурный вызов в некотором программном интерфейсе. Зачастую именно так события этого вида и реализуются. Но наличие событий, связанных с отправкой или приемом пакетов существенно отличает тестирование соответствия от других приложений технологии UniTESK. Для интеграции событий, связанных с сообщениями, была введения концепция *формального интерфейса*, который объединяет взаимодействия, построенные на вызове процедур, с взаимодействиями, основанными на обмене сообщениями.

Элемент формального интерфейса описывается при помощи сигнатуры, состоящей из его имени и набора типов данных, передаваемых этим событием, и контракта события. Напомним, что в UniTESK контракты состоят из двух компонентов — *предусловия* и *постусловия*. Предусловие является предикатом, зависящим от данных события и состояния протокола, предшествовавшего возникновения события. Постусловие является предикатом, зависящим от предшествовавшего возникновению события состояния реализации протокола, еѐ же состояния после наступления этого события и данных возбужденного события.

Предусловие определяет ответственность другого компонента, возбуждающего это событие. Событие можно возбуждать только тогда, когда его предусловие выполнено, иначе корректная работа реализации протокола не гарантируется (её поведение не определено). Постусловие описывает ответственность реализации протокола — она гарантирует его выполнение после возникновения данного события, если непосредственно перед его возбуждением было выполнено предусловие. Элементы формального интерфейса, представляющие обработку запросов от верхнего уровня или обработку входящих сообщений протокола, оснащаются и пред-, и пост-условием. Практика использования UniTESK, накопленная к моменту первых приложений UniTESK к протоколам, позволила прямо обобщить существовавшую на тот момент семантику контрактных спецификаций для программных интерфейсов на обработку входящих сообщений.

Контракт события, представляющего отправку сообщения, не содержит предусловия — он состоит из одного постусловия, определяющего ответственность реализации протокола. Она должна гарантировать, что после возбуждения данного события постусловие этого события выполнено. Отсутствие предусловия в контракте таких событий обосновывается тем, что выходные события возбуждаются самой реализацией протокола, и она отвечает за их корректность.

В ряде проектов в состав формального интерфейса вводились элементы для описания внутренних событий: тайм-аутов и передачи данных внутри реализации по стеку в случае многоуровневых сетевых систем. В настоящее время нет единой методики, как формировать контракт для таких элементов, так как и вызывающая сторона, и обработчик находятся внутри целевой реализации.

Общие части предусловий и постусловий, определяющие ограничения целостности для элементов состояния протокола, выносятся в инварианты состояния. Реализация протокола должна обеспечивать выполнение инвариантов компонента между возникновением событий, но сохранение инвариантов во время обработки входных событий не определено. Последнее ограничение имеет существенное значение для тестирования протоколов, так как большинство современных протоколов спроектированы для параллельной обработки нескольких сообщений (для ускорения обработки и минимизации задержек реализации могут обрабатывать события в нескольких физически параллельных потоках). Соответственно, для построения модели протокола важно представлять семантику параллельной обработки сообщений.

Параллельная обработка событий, соответствующих элементам формального интерфейса, интерпретируется в рамках семантики чередований, а именно:

- События, входящие в формальный интерфейс, являются атомарным, т.е. их возникновение и обработка являются единым процессом, не прерываемым ничем другим, в том числе никакими другими событиями.
- Входные и выходные события составляют частично упорядоченное множество; порядок на этом множестве означает, что предшествующее событие гарантированно завершилось до начала последующего. События несравнимы, если такой информации нет.
- Выходные события, которые порождает целевая система, и её конечное состояние соответствует некоторому полному упорядочению такого частично упорядоченного множества событий.

3.1. Построение формальной модели из текстовой спецификации

Спецификации современных протоколов, как правило, описывают требуемое поведение реализаций на естественном языке. Для построения тестового набора средствами UniTESK необходимо текстовую спецификацию *формализовать* — то есть преобразовать функциональные требования в текст на языке с более строгой семантикой, чем естественный.

Процедура формализации состоит из нескольких шагов. Разделение на шаги обусловлено природой задач, которые возникают при формализации функциональных требований протоколов:

1. Выделение набора функциональных требований к реализации протокола. В текстовых спецификациях требования, как правило, перемежаются пояснениями и обоснованиями, которые не несут нормативной

нагрузки и призваны облегчить понимание требований читателям спецификации, поэтому перед началом разработки формальной модели протокола необходимо отделить требования от всего остального текста. Результатом этого шага является *каталог требований*.

2. Разработка формального интерфейса протокола. А именно, необходимо определить множества входных, выходных и внутренних событий протокола и разработать их представление в виде сигнатур элементов формального интерфейса. Результатом этого шага является представление операций протокола в виде сигнатур функций.

3. Разработка контракта. Текстовые требования, выделенные на первом шаге, переводятся в логические формулы пред- и постусловий элементов формального интерфейса. Результатом этого шага является контрактная спецификация, в которой сигнатуры функций оснащаются пред- и постусловиями.

Предикаты, составляющие контракт, представляются в виде булевских функций. Так как обработка сообщений в общем случае зависит от текущего состояния протокола, в предикатах используется *абстрактное состояние* — набор переменных, моделирующих состояние протокола. Прилагательное «абстрактный» подчеркивает отличие этого состояния от тех переменных, которые используются внутри конкретной реализации. Как правило, абстрактное состояние повторяет концептуальные структуры данных и переменные, которые используются в текстовой спецификации для описания требуемого поведения реализации.

Определение критериев покрытия. Критерий покрытия задаёт разбиение множества переходов в модели протокола и связывает с каждым элементом разбиения уникальную метку. Назначение критерия покрытия – предоставить классификацию переходов, произошедших в ходе тестирования. Переходы, принадлежащие одному и тому же элементу разбиения, считаются эквивалентными. Так как возможны разные виды эквивалентностей, то с одним элементом формального интерфейса может быть связано несколько критериев покрытия.

Выше мы уже упоминали понятие *цели тестирования*. В UniTESK цель тестирования получает строгое определение — при выполнении тестов должны быть осуществлены переходы, соответствующие всем элементам покрытия. Благодаря тому, что спецификация является формальным объектом, отслеживание полноты тестового набора полностью автоматизируется.

Здесь необходимо сделать одно важное замечание. Формальная спецификация в UniTESK используется для тестирования. Как уже упоминалось выше, модель протокола должна включать в себя модель состояния. Для того чтобы в процессе тестирования модель оставалась адекватной реализации, необходимо после каждого наблюдаемого события обновлять значения переменных состояния модели в соответствии со спецификацией. С одной

стороны, постусловие можно рассматривать как некое уравнение, связывающее состояние до возникновения события, параметры события и состояние после события. Если решить это уравнение, зная начальное состояние и параметры события, то можно вычислить новое состояние. На практике, однако, этот подход реализовать не удалось из-за высокой сложности постусловий, моделирующих реальные протоколы. Поэтому, для успешного использования в тестах контракты необходимо дополнить ещё одним компонентом — *функциями реконструкции состояния.* Эти функции нельзя рассматривать как явную спецификацию протокола, так как они не конструируют результат операции по входным данным, а реконструируют состояние реализации, зная её ответ на входное воздействие. Функции реконструкции состояния всегда детерминированы, в отличие от спецификации протокола.

3.2. Построение тестовых сценариев для тестирования соответствия

В рамках UniTESK тестовый сценарий задает некоторый конечный автомат, обход графа состояний которого порождает тестовую последовательность. Состояние автомата вычисляется из состояния модели, символы на дугах автомата определяют тестовые воздействия, которые необходимо оказать на реализацию. Обходчик автомата стремится подать все тестовые воздействия во всех состояниях автомата, достижимых из начального. Обходчик является библиотечным компонентом и не зависит от конкретного протокола или тестового набора.

Необходимо отметить, что описание автомата в тестовом сценарии отличается от обычного конечного автомата: в описаниях дуг отсутствует конечное состояние. Состояние автомата после перехода вычисляется по состоянию модели, а то, в свою очередь, строится функцией реконструкции состояния из реакций, продемонстрированных целевой системой в ответ на тестовое воздействие. Таким образом, конечное состояние перехода полностью определяется моделью протокола.

При проектировании тестовых сценариев для тестирования соответствия протоколов разработчики тестового набора решают следующие задачи:

1. Создание алфавита входных символов автомата теста: разработчики теста составляют множество ситуаций, которые они считают нужным проверить в данном тестовом сценарии. Каждой тестовой ситуации соответствует отдельный символ алфавита (более строго: автомат теста является расширенным конечным автоматом, поэтому символ алфавита ещё дополнен n-кой параметров, что позволяет варьировать аспекты тестовой ситуации — например, перебирать возможные адреса сообщений или значения каких-то полей, — поэтому помимо символов входного алфавита автомата задаются сигнатуры наборов параметров для каждого символа).

Этим алфавитом пользуется обходчик при построении и хранении графа переходов автоматов во время выполнения теста. О том, как входные символы автомата превращаются в конкретные тестовые воздействия см. ниже.

2. Определение состояния автомата теста: необходимо задать структуры данных для представления состояния теста и разработать функцию, которая строит конкретное состояние автомата теста из модельного состояния.

3. Разработка переходов тестового автомата: необходимо определить функции, которые по абстрактному символу (и набору параметров) построят конкретное тестовое воздействия — вызовут функцию из программного интерфейса реализации или сконструируют и отправят в реализацию тестовый пакет.

Важно, что функции, реализующие переход в автомате теста, не обращаются к реализации напрямую. Они вызывают методы из формального интерфейса: UniTESK предоставляет средства для отображения формального интерфейса на конкретные операции с реализацией, такие как вызов функции из программного интерфейса, отправку сообщения в реализацию или ожидание ответа.

4. Разработка средства настройки теста – параметры теста и механизм передачи параметров в тест при его запуске.

Очевидно, что для протоколов возможно практически бесконечное число тестовых ситуаций. Поэтому тест не может перебрать все возможные сообщения во всех возможных состояниях. Необходимо из каких-то соображений уменьшать размерность задачи, проводить факторизацию пространства входных воздействий. В UniTESK для факторизации используются критерии покрытия, введенные в модели. Тесты разрабатываются таким образом, чтобы тестовые воздействия привели к задействованию того или иного элемента покрытия.

В общем случае даже факторизация по элементам покрытия приводит к слишком большому автомату теста. Поэтому тестовый набор, как правило, состоит из нескольких тестовых сценариев, которые дополняют друг друга и в совокупности обеспечивают полное покрытие функциональных требований.

При тестировании протоколов целевая система может демонстрировать реакции спустя некоторое время после того, как на неё было оказано воздействие. Для облегчения моделирования систем с отложенными реакциями в UniTesK введено представление об *автоматах с отложенными реакциями*. Такие автоматы отличаются от обычных конечных автоматов тем, что переходы между состояниями представляют собой цепочку реакций.

Как правило, реализация протокола генерирует реакции по нескольким каналам. Например, у реализации сетевого протокола есть как минимум два канала реакций – пакеты, которые отправляется в сеть, и данные, которые передаются на верхний уровень. Реакции могут регистрироваться с

запозданием, причем для реакций различных типов величина задержки может быть различной. Тестовая система соотносит полученные реакции с элементами формального интерфейса, которые их моделируют. Данная операция кардинально отличается от ситуации, когда тест оказывает воздействие на реализацию: в случае стимула тест явным образом «говорит», какому элементу формального интерфейса соответствует воздействие. В ситуации с реакциями необходимо найти правильный элемент интерфейса, так как от этого зависит и корректность вердикта, и правильность обновления состояния модели. После того, как нужный элемент формального интерфейса найден, с реакцией связывается его постусловие, которое проверяет, допустима ли зарегистрированная реакция в текущем состоянии.

Напомним, что в UniTESK одновременность событий описывается в семантике чередования. Соответственно, тестовой системе необходимо определить допустимый порядок реакций различных видов. В процессе сериализации тестовая система строит различные цепочки реакций и проверяет их допустимость. Каждая реакция рассматривается как переход между промежуточными состояниями, конечное состояние последней реакции рассматривается как последнее состоянии в цепочке реакций и принадлежит множеству состояний конечного автомата. Если в ходе этой операции не удалось найти ни одной допустимой последовательности реакций, то тестовая система выносит вердикт о нарушении семантика чередования: нет такой линейно упорядоченной последовательности событий, которая была бы эквивалентна частично упорядоченному множеству реакций. Такая ситуация трактуется как рассогласование модели и целевой системы – набор зарегистрированных реакций не соответствуют спецификации.

Важная особенность тестирования протоколов заключается в том, что ошибочное поведение может проявляться не только в виде ошибок в полях сообщений или данных, но и в отсутствии ответа на запрос, или наоборот, генерации сообщения в ситуации, когда передача запрещена спецификацией. Нарушение запрета на передачу выявляется в тестах как ошибка перебора цепочек реакций — какую бы линейную последовательность событий ни построила тестовая система, все они окажутся недопустимыми: постусловие для «лишней» реакции будет нарушено в любом случае.

Для выявления ситуации с «отсутствующими» реакциями на тестовый сценарий накладывается ограничение на *стационарность начальных и конечных состояний*, то есть в начале перехода и в конце перехода целевая система не демонстрирует спонтанных (т.е. без воздействия извне) реакций. В частности, это означает, что существует такой интервал времени $\mathbf{T_0}$, что в течение $\mathbf{T_0}$ с момента оказания воздействия все реакции целевой системы на воздействие будут собраны. Стационарность состояния должна определяется по состоянию модели (что логично — состояние реализации недоступно для наблюдения), поэтому разработчики спецификации должны заранее предусмотреть удобные средства для оценки стационарности тестом.

Результатом работы теста является трасса тестовых событий – выбор абстрактного символа, построение конкретного тестового воздействия, вызов модели, вызов реализации и т. д. По набору трасс, порожденных сценариями из тестового набора, строится отчет, в котором важное место занимает отчет о покрытии.

3.3. Устройство тестового стенда

Стенд для тестирования протоколов включает целевое устройство, инструментальный узел и вспомогательные устройства. На целевом устройстве функционирует тестируемая реализация, а на инструментальном узле — тестовая система. В некоторых случаях в состав тестового стенда добавляются вспомогательные устройства, которые используются в ходе тестирования для оказания воздействий на целевое устройство и регистрации сообщений, высылаемых целевым устройством.

Инструментальный узел и целевое устройство подключены к одному или нескольким общим сегментам локальной сети. Несмотря на то, что тестовый сценарий и модель выполняются на инструментальном узле, часть тестовой системы может быть развернута на целевой системе. Именно, на целевом устройстве, могут быть размещены тестовые агенты для оказания воздействия на реализацию или сбора реакций системы, направленных на приложение.

Такой распределенный тестовый стенд может быть реализован как набор виртуальных машин, подключенных к общему сегменту виртуальной сети. Исполнение тестов на виртуальных машинах имеет ряд преимуществ по сравнению с запуском тестов в реальном физическом окружении:

1. Полный контроль над составом узлов в локальных сетях тестового стенда.

2. Возможность создания идентичных копий тестовых стендов для проведения испытаний тестового набора разными участниками проекта.

3. Возможность гибкой конфигурации состава тестового стенда путем добавления или удаления виртуальных машин.

4. Экономия пространства, в частности, все средства ввода-вывода располагаются на одном физическом устройстве.

Благодаря использованию виртуальных локальных сетей и виртуальных машин в тестовом стенде обеспечивается полный контроль над потоками данных. Все информационные потоки эмулируются тестовой системой.

Разумеется, виртуальные тестовые стенды применимы лишь для протоколов, реализованных в типовых операционных системах, работающих на распространенных архитектурах компьютеров (x86, ARM, PowerPC). Тестирование встроенных систем производится только на реальном оборудовании.

4. Опыт применения UniTESK для тестирования соответствия для различных протоколов

4.1. Тестирование сетевого протокола IPv6

Первым приложением UniTESK к тестированию протоколов был проект по разработке тестового набора для нового на тот момент сетевого протокола IPv6 в операционной системе MS Windows 2000 [61-65]. В качестве объекта тестирования выступала реализация IPv6, созданная в исследовательском подразделении корпорации Microsoft [61].

IPv6 – это сетевой протокол нового поколения, призванный заменить прежний протокол сетевого уровня IPv4. IPv6 содержит ряд усовершенствований по сравнению с IPv4 и разрешает проблемы, которые ограничивают дальнейшее развитие сетей на IPv4.

Протокол IPv6 относится к протоколам сетевого уровня. На этом уровне происходит маршрутизация пакетов на основе преобразования сетевых адресов в адреса канального уровня. Сетевой уровень обеспечивает прозрачную передачу данных между транспортным уровнем и канальным уровнем.

Были разработаны спецификации и тестовые сценарии для следующих базовых протоколов стека IPv6:

- протокол IPv6, базовые функции оконечного узла (IETF RFC 2460, 2461, 2462, 2463, 2464, 2710);
- протокол UDP в сетях IPv6 (IETF RFC 768, 2460)
- программный интерфейс стека протоколов (IETF RFC 2292, 2553).

В ходе тестирования были обнаружены отклонения от стандартов IPv6 и ошибки программирования. Отклонения от стандартов заключаются в том, что в ряде случаев поведение MSR IPv6 отличается от требований, изложенных в стандартах на IPv6. При тестировании были также выявлены дефекты, которые можно охарактеризовать как ошибки, допущенные при программировании. В частности, был выявлен дефект обработки некоторых входящих сообщений, при которых выполнение ядра ОС аварийно завершается («синий экран смерти») [62,63].

Кроме того, в процессе разработки формальных спецификаций были обнаружены пробелы в RFC2460 в описании алгоритма сборки фрагментов.

Помимо базовой функциональности IPv6 тестовый набор включал модель и спецификации тестов для функции мобильности IPv6 (Mobile IPv6) [64,65]. Протокол Mobile IPv6 призван обеспечить надежный и эффективный способ поддержания соединений узла IPv6 при перемещении между различными сегментами локальной сети. Одна из важнейших задач Mobile IPv6 заключается в том, чтобы мобильный узел всегда оставался достижим по

своим «постоянным» адресам, даже если он перешёл в сеть с другими диапазонами адресов.

Тестовый набор для мобильных функций разрабатывался на базе существующего тестового набора для MSR IPv6, который содержит тесты для базовых функций оконечного узла IPv6 (host). К тестовому набору для MSR IPv6 были добавлены спецификации и тесты для функций узла-корреспондента и мобильного узла. Тестирование функций домашнего агента не проводилось, так как для этого требуется большой объем дополнительных работ по добавлению спецификаций и тестов на базовые функции маршрутизаторов.

Тестовый набор для IPv6 разрабатывался средствами реализации технологии UniTESK для языка C — CTESK. В технологии CTESK формальные спецификации записываются на спецификационном расширения языка C – SEC (Specification Extension of the C language) [66].

SEC представляет собой ANSI C, пополненный несколькими ключевыми словами и синтаксическими конструкциями. SEC поддерживает такие элементы академических языков формальных спецификаций, как предусловия, постусловия, инварианты типов, инварианты глобальных переменных и описания доступа (access descriptors).

Помимо разработанного нами тестового набора существуют и другие наборы тестов для проверки соответствия требованиям стандартов IPv6. Сравнение с этими тестовыми наборами показало, что автоматизация средствами UniTESK позволяет сократить объем ручного кода в 1,5-5 раз, при этом обеспечивает более высокое покрытие функциональности тестами. В частности, ни один из тестовых наборов не нашел критическую ошибку в MSR IPv6, приводящую к перезагрузке компьютера. Тестирование соответствия Mobile IPv6 позволило выявить ряд нарушений стандартной спецификации и ошибки целевой реализации, которые не были обнаружены разработчиками традиционными методами [62-65].

4.2. Тестирование протокола безопасности сетевого уровня IPsec

Обеспечение безопасности передачи данных на сетевом уровне осуществляется сервисом IPsec. Средствами IPsec можно обеспечить аутентификацию отправителя, целостность данных, конфиденциальность данных, защиту от повторов.

Спецификация IPsec предусматривает гибкий механизм настройки политик безопасности (security policy) и контекстов безопасности (security association). Политики безопасности определяют, какие виды защиты должны быть применены к данным, а в контекстах безопасности хранятся ключи и другие параметры защиты. Для упрощения администрирования IPsec содержит протокол автоматической настройки контекстов безопасности IKE (Internet Key Exchange).

После публикации в 1998 году первой версии спецификаций IPsec они были подвергнуты всестороннему анализу. Выявленные в процессе критического анализа недостатки способствовали лучшему пониманию их особенностей и послужили причиной продолжения исследований в этой области. В результате в 2005 году была опубликована новая версия спецификаций (RFC 4301-4309). Архитектура IPsec (RFC 4301) сохранила основные черты первой версии, однако был внесен ряд существенных изменений. По умолчанию автоматическим протоколом управления ключами выбран IKEv2. IKEv2 представляет собой новый протокол, созданный на основе IKE первой версии (RFC 2407, 2408, 2409), с целью его упрощения и оптимизации. Спецификации протоколов безопасности AH и ESP (RFC 4302, 4303) незначительно отличаются от предыдущей версии (RFC 2402, 2406):

Далее в текущем разделе мы не будем углубляться в подробности отличий между версиями протокола безопасности, и будем обозначать обе версии как IPsec.

В рамках архитектуры IPsec различаются две роли, которые могут выполнять узлы сети:

1. оконечный узел (host) является источником или узлом назначения данных, защищённых посредством IPsec;

2. защитный шлюз (security gateway) осуществляет защиту передаваемых через него данных. В частности, функции защитного шлюза может выполнять маршуртизатор или сетевой экран.

В соответствии с приведенной выше методикой тестирования формальная спецификация IPsec состоит из нескольких компонентов:

- модельного состояния, которое содержит набор структур данных, моделирующих концептуальные структуры данных из стандартов IPsec , таких как база данных контекстов безопасности и база данных политик безопасности;

- формального интерфейса IPsec, включающего спецификационные стимулы, формализующие требования к изменению состояния реализации IPsec при внешнем воздействии на систему, и спецификационные реакции, которые формализуют требования к реакциям реализации IPsec на внешние воздействия;

- критериев покрытия, идентифицирующих различные ветви функциональности IPsec.

Были разработаны тестовые сценарии для оконечного узла, проверяющие требования к обработке входящих и исходящих пакетов, и протокола управления ключами IKE (обеих версий) [67-71].

Тесты для генерации исходящих сообщений использовали статически настроенные политики и контексты безопасности для транспортного протокола UDP. Тестовое воздействие для создания исходящего сообщения

оказывается через специализированного агента, который по команде тестовой системы отправлял заданное сообщение UDP на заданный адрес. Благодаря тому, что настройки безопасности задаются статически, тестовая система может расшифровать результат IPsec обработки на целевой машине и удостовериться в корректности исходящей IPsec дейтаграммы.

Тесты для обработки входящих сообщений использовали статически настроенные политики и контексты безопасности для транспортного протокола UDP. Для проверки того, что сообщение успешно прошло обработку IPsec и данные были доставлены приложению, на целевой узел устанавливался агент, задача которого была получать данные из UDP сокета и передавать их в тестовую систему на инструментальной машине. Тестовые сценарии для входящих сообщений генерируют различные (в том числе и некорректные) защищенные сообщения, а целевая система должна обработать сообщение и либо доставить получателю, либо обработать ошибку в соответствии с требованиями.

Фактически, в тестовых сценариях для IPsec была разработана частичная реализация IPsec, необходимая для корректной генерации и разбора защищенных сообщений.

Для тестирования протокола управления ключами тестовый сценарий эмулировал вторую сторону обменов сообщениями IKE — инициатора установления соединения или ответчика. Это необходимо для установления корректных параметров алгоритмов защиты и т. п. Правильность работы IKE устанавливается посредством тестирования входящих и исходящих сообщений — если реализация IPsec получается обмениваться UDP сообщениями с тестовым сценарием, значит IKE правильно настроил контексты безопасности.

Для тестирования реализаций IPsec использовались два вида тестовых стендов — стенд на реальном оборудовании и вируальный тестовый стенд. В результате апробации тестовых наборов остановились на виртуальном тестовом стенде.

Тестовый стенд включает 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).

В рамках проектов по тестированию соответствия IPsec были получены следующие результаты[67-71]:

- Был проведен анализ различных видов спецификаций функций безопасности телекоммуникационных протоколов.

- Были разработаны тестовые наборы для верификации реализаций IPsec и IPsec v2. Для этого были извлечены функциональных требований из стандартов IPsec первой и второй версии, составлены каталоги требований.

Разработаны формальные спецификации IPsec первой и второй версий, включая спецификацию IKEv2. Разработан набор тестов, проверяющих соответствие реализаций формальной спецификации IPsec.

- Был разработан тестовый стенд с использованием виртуальных машин и развернут стенд для испытаний реализаций в системе виртуализации VMware.

- Проведена апробация разработанного тестового набора на реализациях IPsec v2.

Тестовые наборы для IPsec разрабатывались средствами CTESK. В ходе тестирования был обнаружен ряд отклонений от требований спецификаций и ошибок реализации [ ].

4.3. Тестирование сервиса безопасности транспортного уровня

Протокол IPsec используется преимущественно для прозрачной защиты трафика на сетевом уровне, прежде всего для создания защищенных туннелей между компьютерными системами. Защита обменов данными по транспортным протоколам между приложениями управляется протоколом Transport Layer Security, TLS.

Среди всех современных протоколов, предназначенных для защиты передачи информации в открытых сетях, именно TLS используется наиболее широко. Протокол TLS обеспечивает криптографическую защиту соединения между двумя участниками прикладного протокола. Протокол TLS применяется для защиты обменов между клиентом и сервером в различных прикладных сценариях:

- для защиты обмена данными между веб-сервером и браузером (протокол HTTPS),

- для защиты передачи почтовых сообщений между почтовыми агентами (протоколы SMTP, IMAP, POP3),

- при организации виртуальных защищенных сетей (OpenVPN),

- с протоколом запуска сессий (SIP: Session Initiation Protocol) и основанными на нем приложениями (например, VoIP),

- а также во многих других прикладных сценариях.

На настоящий момент широко используются более десятка различных реализаций TLS. По этой причине обеспечение совместимости реализаций TLS является актуальной задачей.

В данном проекте для записи моделей и тестовых сценариев использовался формализм JavaTESK [72]. JavaTESK использует язык программирования Java с набором расширений для записи формальных спецификаций и задания

тестов. Спецификация на языке JavaTESK обычно состоит из одного или нескольких спецификационных классов, которые описывают состояния и переходы моделируемого протокола. Переходы протокола представляются как методы класса специального вида (спецификационные методы), кроме того поддерживается возможность задать ограничения на множество допустимых состояний посредством инвариантов типов (ограничений на значения типов) и инвариантов переменных состояния. Автомат теста задается в сценарном классе, который содержит процедуру определения текущего состояния автомата теста и итераторы тестовых воздействий. Инструмент JavaTESK предоставляет набор обходчиков, которые строят цепочки тестовых воздействий из описания автомата теста.

Спецификация протокола TLS написана на естественном языке. Для тестирования реализации на соответствие было необходимо выделить из стандарта отдельные требования и затем их формализовать. В результате анализа текста стандарта был составлен полный список требований (около 300 требований).

Для модельного представления TLS-сообщений разработана библиотека соответствующих спецификационных типов, позволяющая моделировать различные варианты сообщений. В спецификации каждое входящее TLS-сообщение рассматривается как последовательность стимулов. Каждый стимул в этой последовательности соответствует обработке отдельного блока данных в TLS-сообщении (TLS-сообщение может содержать несколько структур данных конкретного типа).

Разработан тестовый набор для тестирования соответствия серверных реализаций TLS [73]. Тестовые воздействия проверяют корректность ответа сервера на запросы клиента. Особое внимание в тестах уделено тестированию аномального поведения клиента: спецификация TLS насчитывает более 40 различных ситуаций, при которых сервер должен прекратить обмен сообщениями и разорвать соединение.

В отличие от протокола IPsec в данном тестовом наборе не используется приложение-агент на целевой машине, который бы отвечал за установление соединения с клиентом. Вместо него используется TLS в реальных серверных приложениях — веб-сервере или почтовом сервере, - с которым тест устанавливает связь по соответствующему порту.

Для тестирования на соответствие стандарту были несколько известных открытых реализаций TLS. В ходе тестирования во всех реализациях были выявлены отклонения от стандартов, причем в одном случае реализация нарушает критическое требование[73].

Тестовый набор для TLS является новым результатом. Мы не обнаружили других открытых тестовых наборов, проверяющих соответствие реализаций TLS стандарту. Существуют отдельные тестовые наборы, созданные разработчиками реализаций TLS (например, openssl, Java SSE), но эти

тестовые наборы сфокусированы на тестировании внутренних аспектов работы реализации и непригодны для тестирования других реализаций.

4.4. Тестирование почтовых протоколов

В данном разделе приводится краткий обзор проектов [74-79] по тестированию реализаций электронной почты.

Электронные письма являются основой современного взаимодействия между людьми. Миллионы писем перемещаются ежедневно в сети Интернет. Надежность и корректность инфраструктуры почтовых сообщений чрезвычайно важна для современного информационного общества. В этой статье мы коснемся двух аспектов этих вопросов: надежности (1) передачи почты в сети Интернет и (2) доставки писем конечным адресатам.

Несмотря на более чем двадцатилетнюю историю почтовых протоколов и существования десятков реализаций протоколов SMTP, POP3 и IMAP4, до сих пор нет открытого и независимого от реализации тестового набора для проверки соответствия стандартам. Несмотря на кажущуюся простоту, почтовые протоколы:

1. недоспецифицированы, существенная часть функциональности оставлена на усмотрение разработчика, в спецификации описаны несколько вариантов возможного дальнейшего поведения системы;

2. недетерминированы: стандарт допускает различные варианты поведения системы, включая отказ в доставке почтовых сообщений или разрыв соединений;

3. расширяемы: реализация протокола может использовать различные расширения, как дополняющие функциональность протокола, так и изменяющие её;

4. функции почтовых протоколов различаются по степени обязательности (MUST, SHOULD, MAY и прочие).

Перечисленные особенности определяют фактическую сложность разработки тестовых наборов для тестирования реализаций протоколов на соответствие спецификациям.

В проекте по тестированию почтовых протоколов был предложен новый метод [78] создания тестов для тестирования соответствия. Кратко опишем ключевые моменты.

Выше были представлена методика разработки тестовых наборов, в которой разработка начинается с выделения требований, затем требования формализуются и только на заключительном этапе появляются тесты. Такой подход к разработке тестовых наборов имеет много общего с известной каскадной моделью разработки ПО, когда разработка начинается с детального анализа требований и ведется путем последовательного проектирования, реализации и тестирования.

Метод разработки модели и тестов, опробованный в проекте по тестированию почтовых протоколов, напротив, больше перекликается с новыми, популярными подходами к разработке ПО из семейства «гибкого программирования» (Agile Programming, AP). Один из ключевых моментов в AP — первая работающая версия продукта появляется достаточно рано в ходе проекта, новые работающие версии продукта появляются достаточно часто. Разумеется, ранние сроки и частота обновления версий объясняются неполнотой реализаций — на ранних этапах клиент видит скорее прототип, чем полную реализацию всех его требований и пожеланий.

В проекте по тестированию почтовых протоколов был опробован аналогичный подход — первые тесты появились достаточно рано по времени проекта, и далее они уточнялись и развивались. Так как разработка модели протокола — это трудоемкий и затратный по времени процесс, который нельзя сделать «наполовину». Модель должна быть полной и точной. Поэтому в описываемом методе модель разрабатывается ближе к концу проекта.

Рассмотрим шаги метода. Разработка тестового набора начинается с изучения и анализа требований. Необходимо выделить набор функций, составляющих протокол, и расставить приоритеты — в какой последовательности тестировать эти функции.

На следующем шаге для самых приоритетных функции разрабатываются элементарные тесты (test cases) без использования модели. Если сравнивать с Agile методами, то этот тестовый набор аналогичен «первому релизу» – он не полон, но функционален и уже работает.

Важно отметить, что в данном методе все разрабатываемые тесты пропускаются на реализации. Они отлажены и в конце каждого этапа гарантированно корректны.

Так как наша цель заключается в том, чтобы в конечном итоге получить тестовый набор, основанный на моделях, последующие шаги направлены на то, чтобы приблизить разработку к этой цели.

Первый тестовый набор, как правило, «бесструктурен», он представляет собой набор независимых тестовых программ. Поэтому на втором шаге проводится структурирование — выделяются интерфейсы, разграничивающие тесты и реализацию. Так, если исходно тест взаимодействовал с реализацией протокола SMTP непосредственно через отправку и получение сообщений, то после второго шага появляется интерфейс, абстрагирующий отдельные команды протокола SMTP: helo, ehlo, maifrom, rcpt и т. д. Это ещё не модель, так как внутри этого интерфейса нет состояний и переходов, это скорее формальный интерфейс протокола (понятие формального интерфейса обсуждается в разделах 3 и 3.1). Подготовка тестовых данных, отслеживание состояния модели и вынесение вердикта на этом шаге метода остаются в тестах. Параллельно с выделением формального интерфейса могут разрабатываться дополнительные тесты, но уже в терминах операций формального интерфейса.

На третьем шаге метода вводится первый элемент автоматизации. Тестовый набор, полученный к настоящему моменту, представляет собой набор линейных программ. Тестирование соответствия требует большого количества тестов, так как полный тестовый набор должен содержать тесты для каждого пути в графе переходов автомата, моделирующего протокол. Разработка такого тестового набора вручную — это монотонный, однообразный труд, причем большинство тестов будут иметь значительные пересечения. Автоматизация заключается в том, что тесты перестают быть линейными, они трансформируются в конечные автоматы. Тесты становятся тестовыми сценариями UniTESK. Количество тестовых программ может остаться тем же самым или даже уменьшиться, но количество тестовых ситуаций, которое они покрываю, значительно возрастет благодаря обходу автомата.

В отличие от канонического подхода, изложенного в разделе 3, в этих тестах структура состояния и обновления состояния реализованы непосредственно в тестовых методах. Благодаря тому, что про эти тесты пропускаются на реализации, мы можем быть уверены, что разработанная модель состояния и структура переходов корректны.

Следующий шаг метода заключается в трансформации формального интерфейса в модель протокола. На одном из предыдущих шагов был выделен компонент, реализующий формальный интерфейс. На этом шаге он разделяется на два: в один компонент отправляется код, реализующий абстрактные операции helo, mailfrom и т. п. через сообщения протокола, и этот компонент становится адаптером (медиатором), а во второй компонент переносятся из тестов состояние и функции обновления состояния. Фактически, появляется модель протокола, включающую модель состояния протокола и функции обновления этого состояния в зависимости от ответов реализации. Соответственно, обновляются тесты: проверки корректности, которые были в них реализованы, переписываются в терминах изменений состояния компонента модели. Параллельно с разработкой модели могут добавляться новые тесты, но эти тесты должны быть устроены как автоматы и выносить вердикты на основе выделенной модели. Следующая трансформация заключается в переносе кода для вынесения вердикта из тестов в модель. В тестах остаются только подготовка данных и вызовы методов формального интерфейса. Так как к текущему моменту все тесты были уже отлажены, то мы можем быть уверены, что проверки, добавленные в модель — корректны.

После того, как модель протокола выделена в виде исполнимого компонента, можно трансформировать её в спецификацию UniTESK. Проверки переносятся в пред- и постусловия, операции с состоянием становятся функциями реконструкции состояния. Все последующие тесты разрабатываются как тестовые автоматы с использованием полученной модели.

Предложенный подход включает несколько этапов разработки тестов, причем этапы сконструированы таким образом, что каждый этап может быть

завершающим. Новизна разработанного метода тестирования соответствия заключается в последовательной трансформации модели тестирования, при которой на каждом шаге получаются отлаженные тесты и/или модель. При выполнении этих шагов получается тот же по функциональности тестовый набор, что был после первого шага. Но на данном этапе уже намного проще расширять тестовый набор путем расширения формальной спецификации тестируемого протокола.

Причины, побудившие разработать изложенный подход:

1. В проекте по созданию тестовых наборов для почтовых протоколов участвовали разработчики, незнакомые с технологией UniTESK. Предложенный метод позволил вводить концепции тестирования, основанного на моделях, постепенно и на практике. Разработка началась с простых и понятных элементарных тестов и постепенно включила в себя автомат теста, модель протокола, спецификацию протокола.

2. Разработчики тестов для почтовых протоколов ранее не были знакомы с этими протоколами. Начав с исполнимых тестов, они на практике познакомились с операциями протокола. Как показывает опыт предыдущих проектов по тестированию соответствия, немногие люди обладают достаточным развитым абстрактным мышлением, чтобы сразу после прочтения стандарта суметь написать модель протокола. Большинству нужно сначала попробовать протокол на практике, например, написав программы, которые используют протокол для решения каких-то несложных задач, вроде отправки письма по SMTP. Первый тестовый набор, состоящий из линейных тестов, как раз и направлен на решение задачи знакомства с протоколом.

3. Опять-таки, как показывает практика, первые прогоны тестов, разработанных с использованием моделей по технологии UniTESK, завершаются с негативными вердиктами из-за ошибок в спецификации. В данном подходе спецификация получается через последовательность трансформаций отлаженных и работающих программ, поэтому степень доверия такой спецификации выше, чем той, которая разработана без запуска тестов.

Разумеется, помимо достоинств у предложенного метода есть недостатки. Во-первых, для его успешного применения требуется наличие корректной реализации. Если реализации нет, или она некорректна, то предложенный метод «не работает»: нет возможности запускать и отлаживать тесты, что является критически важным для успешного применения данного метода. Во-вторых, полученная спецификация, очевидно, неполна, так как она получается из изначально неполного тестового набора. Если бы тестовый набор сразу был полон, то не было бы необходимости разрабатывать модель. Тем не менее, даже такая неполная спецификация обладает определенной ценностью — разработчики спецификации освоили предметную область, и развитие спецификации будет проще, чем разработка «с нуля».

В качестве примера применения предложенного метода с его помощью был разработан открытый тестовый набор для проверки соответствия основной функциональности протоколов SMTP, POP3 и IMAP4 их стандартам. Также был разработан тест, использующий композицию спецификаций протоколов SMTP и POP3. Данный тест используется для проверки корректности взаимодействия реализаций различных почтовых серверов.

Спецификации и тесты разрабатывались средствами технологии JavaTESK.

Тесты применялись для почтовых серверов, разрабатываемых по модели открытого исходного кода – Apache James, hMail Server, Postfix и Dovecot. Сгенерированные тесты обнаружили несколько несоответствий между реализациями протоколов и стандартами, в том числе одну ошибку в реализации, которая приводила к бесконечному зацикливанию письма внутри сервера при определенной конфигурации окружения.

Тестовый набор для почтовых протоколов является новым результатом. Разработчики реализаций почтовых протоколов разрабатывают также и тесты, но, как оказалось, тесты сильно связаны с конкретными реализациями и непереносимы для тестирования серверов от других разработчиков, так как используют особенности реализации: настройки параметров, доступ к внутреннему состоянию, выполнение в одном процессе с реализацией. Использование тестов, разработанных для одной реализации, не предоставляется возможным для проверки других реализаций. Более того, эти тесты не предназначены для тестирования соответствия, они проверяют прежде всего корректность реализации многочисленных настроек почтового сервера, не относящихся напрямую к стандартам SMTP, POP3 и IMAP4. Проблема заключается в том, что такой подход к тестированию не гарантирует обеспечения совместимости сервера со стандартом. В частности, именно тестирование соответствия позволило выявить в одном из серверов серьезную функциональную ошибку.

4.5. Тестирование протоколов авионики

Использование CTESK или JavaTESK для разработки моделей и тестов требует от авторов тестового набора значительных навыков программирования. Мало того, что сами по себе языки C и Java сложны в изучении для неспециалиста, требуется дополнительно освоить спецификационные расширения и научиться пользоваться соответствующими инструментами.

Поэтому была проведена работа по реализации технологии UniTESK в более простом языке и без использования расширений. В качестве базового языка был выбран динамический язык Python. Это простой язык с хорошо определенной семантикой, ясным синтаксисом, мощными и содержательными конструкциями. Язык Python, безусловно, проще в обучении, чем C или Java.

На языке Python были разработаны базовые компоненты UniTESK, прежде всего обходчики автомата теста. Разработаны специальные декораторы, упрощающие спецификацию автомата теста (тестового сценария),

позволившие отказаться от расширения языка дополнительными синтаксическими конструкциями.

Благодаря тому, что Python является динамическим языком, связывание модели и реализации происходит легко и непринужденно. Не требуется сложная трансляция из расширения в базовый язык, как в CTESK или JavaTESK, или динамическая генерация байт-кода Java, как это делается в некоторых фреймворках на Java.

Средствами PyTESK были разработаны тестовые наборы для трех протоколов, использующихся для обмена файлами между бортом самолета и наземными службами.

Тестирование доступной реализации выявило существенные ошибки в реализации, как отклонения от стандарта, так и программные ошибки, которые препятствовали успешному выполнению операций протокола.

Использование динамического языка для автоматизации тестирования является новой тенденцией в Software Engineering. Существуют несколько проектов, нацеленных на автоматизацию тестирования с использованием динамических языков, например PyModel на Python [80] и errfix на Ruby [81].

PyTESK отличается от этих проектов тем, что автомат теста не извлекается из модели, а частично задается разработчиком.

5. Обсуждение применимости инструментов CTESK, JavaTESK и PyTESK для тестирования соответствия

Изначально UniTESK «вырос» из проекта KVEST [82] по верификации ядра операционной системы. В этом проекте модель системы и тесты разрабатывались на академическом языке формальных спецификаций RSL. Проект KVEST завершился успешно — был разработан тестовый набор и выявлен ряд критических ошибок в ядре ОС. Однако, в компании-заказчике не смогли перенять и поддержать разработанный тестовый набор, так как нотация языка и концепции, лежащие в его основе, были трудны для восприятия инженерами-программистами.

В результате, UniTESK стал развиваться в направлении сближения формальных методов и повседневной практики программирования. Ключевую роль в этом сближении играл перенос концепций контрактной спецификации и автомата теста в популярный язык программирования.

5.1. CTESK

Первым языком, для которого была разработана технология автоматизации тестирования с использованием моделей, стал язык C. Это очень популярный и сравнительно простой язык, хорошо знакомый подавляющему большинству программистов. Однако, встроенные средства языка не позволяют добавлять в него новые конструкции, в частности пред- и пост-условия. Поэтому было

разработано *расширение* языка C, которое представляет собой ANSI C с несколькими дополнительными конструкциями [66].

С использованием CTESK были разработаны ряд тестовых наборов для программных интерфейсов ОС Linux, включая программный интерфейс POSIX [83,84]. В этих проектах CTESK хорошо себя зарекомендовал — благодаря близости языка моделирования к языку реализации, профессиональный программист на Си без труда мог понять модели и тесты, даже не зная подробности семантики расширении языка.

Однако применение CTESK для тестирования протоколов оказалось менее успешным. Средствами CTESK были разработаны тестовые наборы для IPv6 и IPsec. В силу того, что язык Си не содержит высокоуровневых конструкций, таких как списки, итераторы, ООП, полиморфизм, и т. п., модели и тесты на Си содержат большое количество «технического» кода, необходимого для компиляции и выполнения тестового набора.

Концепции технологии UniTESK естественным образом ложатся на парадигму объектно-ориентированного программирования. Но в языке Си нет поддержки ООП, поэтому аналоги виртуальных функций приходилось создавать вручную (одна из частей «технического») кода.

Универсальный обходчик, который не зависит от модели и теста, требует нетривиального управления памятью. Аллокация и деаллокация памяти могут отстоять друг от друга на сотни шагов по тесту, поэтому в CTESK был реализован свой сборщик мусора. Что только прибавило сложностей при разработке состояния и тестов. Этот сборщик предоставил специальный API для объектов времени выполнения. API позволяет создавать объекты, модифицировать и удалять, когда они становятся ненужными. Такое ручное управление данными в модели и тестовом сценарии неоднократно приводили к трудновоспроизводимым ошибкам.

Как правило, первые расхождения со спецификацией, которые находит тестовый набор, вызваны ошибками в модели или тесте. Трансляция дополнительных конструкций в базовый язык (Си) порождала сложный код, для отладки которого требовалось хорошее знание особенностей трансляции расширения.

Несмотря на все трудности, обусловленные использованием CTESK, тестовые наборы были разработаны и успешно использованы для тестирования реализации и обнаружения ошибок.

5.2. JavaTESK

Следующий крупный шаг в развитии UniTESK был связан с реализацией технологии на базе объектно-ориентированного языка с автоматической сборкой мусора. Было разработано спецификационное расширение языка Java, получившее название JavaTESK[72].

Переход от Си к Java позволил значительно упростить модели и тестовые сценарии, освободить их от многих вынужденных технических «костылей»,

сделать более понятными. Тем не менее, JavaTESK не сумел разрешить ряд ключевых проблем, связанных с использованием расширения языка программирования:

1 Расширение отличается от базового языка, поэтому среды разработки считают программы, написанные с использованием расширения, некорректными. Требуется разрабатывать специализированные плагины для сред разработки, которые добавляют поддержку расширения в среду разработки. Такие специализированные решения, как правило, отстают по набору функций от поддержки базового языка. Это, безусловно, мешает разработчикам адаптироваться к использованию расширения языка.

- Расширение транслируется в базовый язык нетривиальным образом. Запутанная схема генерации кода порождает трудности с отладкой тестов и поиском ошибок в тестах, если что-то пошло не так.
- Язык Java содержит больше высокоуровневых конструкций, чем Си, но тем не менее, разработка моделей и тестов на JavaTESK требует много технического кода (меньше чем в C, но тем не менее).
- Переход от базовых протоколов к расширяемым показал, что для JavaTESK нетривиально реализуется метод моделирования расширений протоколов. Требуются «фокусы» с динамической генерацией байткода, чтобы собирать модель реализации по набору расширений, которые она поддерживает.

Тем не менее, на JavaTESK были успешно разработаны тесты для протокола TLS и почтовых протоколов.

5.3. PyTESK

Применение современного динамического языка в качестве базы для автоматизации тестирования с использованием моделей открывает новые перспективы. Динамический язык позволяет гибко описывать модели и тесты с минимумом вспомогательного кода, не содержащего логики протокола или теста. Динамический язык позволяет легко строить композиционные модели и тесты. Отсутствие строгой типизации, мощные высокоуровневые конструкции, сравнительно простой синтаксис облегчает освоение этого языка неспециалистам, в частности, занятых тестированием гибридных систем и интеграции программно-аппаратных комплексов.

К настоящему моменту на PyTESK реализованы несколько экспериментальных тестовых наборов, включая протокол безопасности транспортно уровня TLS. Ведутся работы по созданию максимально простого метода разработки тестов.

При всех своих плюсах динамический язык имеет существенный минус — большинство ошибок программирования выявляются на этапе исполнения.

Разработка тестового набора требует наличия реализации, для которой можно запускать тесты и отлаживать их.

6. Заключение

В статье обобщается опыт коллектива в исследовании и разработке формальных методов верификации и тестирования реализаций протоколов Интернета. Необходимая методологическая основа подобных исследований была разработана и внедрена в рамках работ ИСП РАН, поддержанных грантами РФФИ, результатом которых стала широко применяемая в настоящее время оригинальная технология тестирования на основе формальных спецификаций UniTESK.

В результате этих исследований был разработан новый, основанный на контрактных спецификациях, метод верификации, позволяющий эффективно автоматизировать тестирование очень сложных протоколов Интернета. При этом созданные тестовые наборы обладали формально определенным и прослеживаемым покрытием требований спецификаций, что в значительной степени улучшило качество тестирования и позволило выявить целый ряд ошибок в существующих реализациях соответствующих протоколов.

В ходе разработки тестовых наборов были выявлены и некоторые особенности протоколов, затрудняющие тестирование реализаций с помощью технологии UniTESK, а также особенности инструментов, поддерживающих ее работу, однако все эти особенности удавалось успешно преодолеть, не выходя за рамки ее ограничений.

Список литературы

[1] CCITT Recommendation Z.100. Specification and Description Language (SDL). Geneve, Switzerland: ITU, 1993. 245 с.

[2] ISO/IEC 9074. Information Processing Systems — Open Systems Interconnection. Estelle — A Formal Description Technique based on an Extended State Transition Model. Geneve, Switzerland: ISO, 1989 1 я редакция,1997 2-я редакция. Отозван 06.05.1999.

[3] ISO/IEC 8807. Information Processing Systems — Open Systems Interconnection. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Geneve, Switzerland: ISO, 1989. 142 с.

[4] ITU-T X.680 (11/08) Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T, 2008. 194 с.

[5] OMG Unified Modeling LanguageTM (OMG UML), Infrastructure . OMG, 2011. 230 с.

[6] Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ. // Изд-во Моск. центра непрерыв. мат. образования, 2002. 416 с.

[7] Карпов Ю.Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем. // БХВ-Петербург, 2010г. 560 с.

[8] M. Diaz. Petri Nets: Fundamental Models, Verification and Applications. // Willey, 2013г. 656с.

[9] Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007).
[10] Blackburn, M., Busser, R., Nauman, A.: Why Model-Based Test Automation is Different and What You Should Know to Get Started. Software Productivity Consortium, NFP (2004).
[11] Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-Based Testing in Practice. In: Proceedings of the ICSE 1999 (May 1999).
[12] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. *UniTesK Test Suite Architecture.* Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77-88, Springer-Verlag, 2002.
[13] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. *Подход UniTesK к разработке тестов.* Программирование, 29(6):25-43, 2003.
[14] В. В. Кулямин, А. К. Петренко. Развитие подхода к разработке тестов UniTESK. Труды ИСП РАН, 26(1), 2014. DOI: 10.15514/ISPRAS-2014-26(1)-1.
[15] F. C. Hennie. Fault detecting experiments for sequential circuits. // Proc. 5-th Ann. Symp. Switching Circuit Theory and Logical Design, 1964. С. 95-110.
[16] [Василевский] М. П. Василевский. Диагностика ошибок в автоматах. // Кибернетика и системный анализ, т. 9, № 4, 1973. С. 98-108.
[17] T. S. Chow. Testing software design modeled by finite-state machines. // IEEE Trans. on Software Engineering, vol. 4, no. 3, 1978. С. 178-187.
[18] J. A. Bergstra, J. W. Klop. Algebra of Communicating Processes with Absraction. Theoretical Computer Science, 37(1), 1985. С. 77-121.
[19] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985; электронное издание, 2004. 260 с. [PDF] (http://www.usingcsp.com/cspbook.pdf).
[20] R. Milner. Communication and Concurrency. Prentice-Hall, 1989. 260 с.
[21] N.A. Lynch, M.R. Tuttle, "An Introduction to Input/Output Automata" // CWI-Quaterly 3, 1989, P. 219-246. [PDF] http://www.markrtuttle.com/papers/lt89-cwi.pdf.
[22] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. // CONCUR'92 Proceedings, LNCS 630. Springer-Verlag, 1992. [PS] http://www-verimag.imag.fr/~maraninx/ArgosCONCUR92.ps.gz
[23] ISO/IEC 7498. Information technology – Open Systems Interconnection – Basic Reference Model. Geneva, Switzerland: ISO, 1994.
[24] Ed. Brinksma. A theory for the derivation of tests. Proc. IFIP WG6.1 8th Intl. Symp. on Protocol Specification, Testing, and Verification, North-Holland, S. Aggarwal and K. Sabnani Ed. pp. 63-74, 1988.
[25] K. K. Sabnani and A. T. Dahbura. A protocol test generation procedure. Computer Networks and ISDN Systems, vol. 15, no. 4, pp. 285-297, 1988.
[26] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. IEEE Trans. on Software Eng., vol. 17, pp. 591-603,1991.
[27] G. Luo, A. Petrenko, and G. v. Bochmann. Selecting test sequences for partially specified nondeterministic finite state machines, Proceedings of the IFIP Seventh International Workshop on Protocol Test Systems, Japan, 1994, pp. 95-110.
[28] Н. В. Евтушенко, А. В. Лебедев, А. Ф. Петренко. Построение проверяющего множества для компоненты последовательной автоматной сети. Автоматика и телемеханика, № 8. стр. 145-153, 1994.
[29] J. C. Fernandez, C. Jard, T. Jeron, C. Viho. Using on-the-Fly Verification Techniques for the Generation of Test Suites // Proceedings of the 8th International Conference on Computer Aided Verification, LNCS 1102, Springer-Verlag, 1996, P. 348-359.

[30] J. Tretmans. Test Generation with Inputs, Outputs, and Repetitive Quiescence. Software — Concepts and Tools, 17(3):103-120, 1996.
[31] D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: a detailed study. IEEE Trans. Soft. Eng., vol. 15, no. 4, pp. 413-426, 1989.
[32] D. Lee, M. Yannakakis. Principles and methods of testing finite state machines — a survey. Proc. IEEE, 84(8):1090-1123, 1996.
[33] G. v. Bochmann, A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. Proc. of ACM SIGSOFT ISSTA'1994, Software Engineering Notes, Special Issue, pp. 109—124.
[34] A. Petrenko. Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In F. Cassez, C. Jard, B. Rozov, M. Dermot, eds. Modeling and Verification of Parallel Processes: 4-th Summer School, Nantes, France, LNCS 2067, pp. 196-200, Springer-Verlag, 2000.
[35] ISO/IEC 9646. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts. Geneva: ISO, 1994. 46 c.
[36] ITU-T Recommendation Z.500. Framework on formal methods in conformance testing. Geneve, Switzerland: ITU, 1997. 49 c.
[37] Glenford J. Myers, Corey Sandler, Tom Badgett. The Art of Software Testing, 3rd Edition. Wiley, 2011. C. 240.
[38] J. Tretmans, An Overview of OSI Conformance Testing. Translated and adapted from: J. Tretmans and J. van de Lagemaat, Conformance Testen, in Handboek Telematica, Vol. II, pages 1--19. Samson, 1991.
[PDF] http://people.cs.aau.dk/~kgl/TOV03/iso9646.pdf
[39] ISO/IEC 9646. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 2: Abstract Test Suite specification. Geneva: ISO, 1994. 33 c.
[40] Information technology – Open systems interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN). 1-е издание. Geneva, Switzerland: ISO, 1992.
[41] Information technology – Open systems interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN). 2-е издание. Geneva, Switzerland: ISO, 1998.
[42] ETSI ES 201 873-1 V3.1.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia-Antipolis, France: ETSI, 2005. 210 c.
[43] OMG formal/05-07-07. UML Testing Profile. Version 1.0. Needham, USA: Open Management Group, 2005. [PDF, PostScript] (http://www.omg.org/cgi-bin/doc?formal/05-07-07).
[44] L. Ebrecht, M. Schacher, C. Bühler. Test Specification in XML – the most important Element for Test Automation. // ARTiSAN Benutzerforum D.A.CH, 2005.
[45] Программный комплекс для разработки тестов. [URL] http://www.junit.org.
[46] Проект TAHI по разработке тестового набора для стека протоколов IPv6. [HTML] (http://www.tahi.org/).
[47] Digital cellular telecommunications system (Phase 2+) (GSM); Handover procedures (GSM 03.09 version 5.1.0) // ETSI, Sophia-Antipolis, France, 1997. 81 c.
[48] ETSI ETS 300 838. Integrated Services Digital Network (ISDN); Harmonized Programmable Communication Interface (HPCI) for ISDN. // ETSI, Sophia-Antipolis, France, 1998. 546 c.

[49] C. Jard, T. Jéron. TGV: Theory, principles and algorithms. // International Journal on Software Tools for Technology Transfer (STTT), vol. 7(4). Berlin: Springer, 2005. С. 297 – 315
[50] E. Farchi, A. Hartman, S.S. Pinter. Using a Model-based Test Generator to Test for Standard Conformance. // IBM System Journal - special issue on Software Testing. Volume 41(1), 2002. С. 89 - 110..
[51] G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected State Machine Coverage for Software Testing. // Proceedings of ISSTA 2002 International Symposium on Software Testing and Analysis. New York, USA: ACM Press, 2002. С. 134 – 143.
[52] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, in Formal Methods and Testing, vol. 4949, pp. 39-76, Springer Verlag, 2008
[53] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink. Formal test automation: A simple experiment. // G. Csopaki, S. Dibuz, K. Tarnay, editors. 12th Int. Workshop on Testing of Communicating Systems. Budapest, Hungary: Kluwer Academic Publishers, 1999. С. 179-196.
[54] Axel Belinfante. JTorX: a Tool for On-Line Model-Driven Test Derivation and Execution. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010. LNCS vol. 6015, pp. 266-270. Springer.
[55] N. Goga. A probabilistic coverage for on-the-fly test generation algorithms. // Automated Verification of Critical Systems (AVoCS '03), 2003.
[56] 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.
[57] V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin, A. S. Kossatchev, I. B. Bourdonov. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. // Proceedings of the 5-th International Conference on Perspectives of System Informatics, July 9-12, 2003, Novosibirsk, Russia; LNCS 2890, Springer, 2003, pp. 450-461.
[58] V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. // Proceedings of the 2-nd International Service Availability Symposium, April 25-26, 2005, Berlin, Germany; LNCS 3694, Springer, 2005, pp. 68-83.
[59] V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. // Proceedings of the 9-th World Multiconference on Systemics, Cybernetics, and Informatics, Model Based Development and Testing Workshop, July 10-13, 2005, Orlando, Florida, USA, pp. 65-70
[60] C. A. R. Hoare. An axiomatic basis for computer programming. // Communications of the ACM, volume 12 №10, 1969. С. 576-580.
[61] R. P. Draves, A. Mankin, B. D. Zill. Implementing IPv6 for Windows NT. Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA, August 3–4, 1998
[62] Агамирзян И., Грошев С.Г., Хорошилов А.В., Ключников Г.В., Косачев А.С., Омельченко В.А., Пакулин Н.В., Петренко А.К., Шнитман В.З. Применение формальных методов для тестирования MSR IPv6. // Интернет нового поколения. Сборник тезисов международной конференции. Ярославль, 2002. С. 29-33.
[63] Ключников Г.В., Косачев А.К., Пакулин Н.В., Петренко А.К., Шнитман В.З. Применение формальных методов для тестирования реализации IPv6. // Труды ИСП РАН, Том 4. М., 2003. С. 121-140.

[64] Ключников Г.В., Косачев А.С., Пакулин Н.В., Петренко А.К., Шнитман В.З. Применение формальных методов для тестирования Mobile IPv6. // Интернет нового поколения. Сборник тезисов II международной конференции. Ярославль, 2003. С. 20-25.
[65] Зацепин Д.В., Шнитман В.З. Особенности применения технологии UniTESK для тестирования функций мобильности в протоколе IPv6. // Труды ИСП РАН 13 (1). М. , 2007. С. 143-170
[66] CTesK 2.1: SeC Language Reference. М.: ИСП РАН, 2005. 167 с.
[67] Ключников Г.В., Пакулин Н.В., Шнитман В.З. Автоматизированное тестирование сетевых сервисов Интернет-протокола // Труды Всероссийской научной конференции «Научный сервис в сети ИНТЕРНЕТ: технологии распределенных вычислений», г. Новороссийск, 19-24 сентября 2005 г. С. 168-170.
[68] Н.В. Пакулин "Применение формальных методов для тестирования реализаций сложных современных протоколов", Сборник трудов международного семинара "Go4IT - шаг к новым технологиям Интернета", ИСП РАН, М., 2007 г., с. 11-18.
[69] А.В. Никешин, Н.В. Пакулин, В.З. Шнитман. Особенности тестирования сервисов безопасности сетевого уровня IPsec второй версии. // Научный сервис в сети Интернет: решение больших задач: Труды Всероссийской научной конференции (22-27 сентября 2008 г., г. Новороссийск). - М.: Изд-во МГУ, 2008. - 468 с. ISBN 978-5-211-05616-9.
[70] А.В. Никешин, Н.В. Пакулин, В.З. Шнитман. Разработка тестового набора для верификации реализаций протокола безопасности IPsec v2. // Труды Института системного программирования РАН, том 18, 2010 г. Стр. 151-182.
[71] А.В. Никешин, Н.В. Пакулин, В.З. Шнитман. Верификация функций безопасности протокола IPsec v2. // Программирование, том 37 № 1. М., 2011. С. 36-56.
[72] Bourdonov I.B., Demakov A.V., Jarov A.A., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. and Zelenov S.V. Java Specification Extension for Automated Test Development // Proceedings of PSI'2001. Novosibirsk, Russia July 2-6 2001, LNCS 2244:301-307. Springer-Verlag, 2001.
[73] А.В. Никешин, Н.В. Пакулин, В.З. Шнитман. Разработка тестового набора для верификации реализаций протокола безопасности TLS. // Труды ИСП РАН, том 23, 2012 г. Стр. 387-404. DOI: 10.15514/ISPRAS-2012-23-22.
[74] А.Н. Тугаенко. Тестирование соответствия почтовых протоколов сети Интернет. // Материалы Международного молодежного научного форума «ЛОМОНОСОВ-2010» (тезисы), стр.25-27.
[75] Н.В.Пакулин, А.Н.Тугаенко. Разработка тестовых наборов для тестирования соответствия почтовых протоколов. // Конференция АППИ-2009, стр. 154-160.
[76] А.Н. Тугаенко. Метод тестирования соответствия для расширяемых протоколов сети Интернет. // Материалы Международного молодежного научного форума «ЛОМОНОСОВ-2011».
[77] N. Pakulin, A. Tugaenko. Model Based Conformance Testing for Extensible Internet Protocols // Proceedings of SYRCoSE 2011.Н.В. Пакулин, А.Н. Тугаенко.
[78] Пакулин Н.В., Тугаенко А.Н., Шнитман В.З. Тестирование протоколов электронной почты Интернета с использованием моделей. // Труды ИСП РАН, том 20, 2011 г. Стр. 125-141.
[79] Пакулин Н.В., Тугаенко А.Н., Шнитман В.З. Тестирование протоколов электронной почты Интернета с использованием моделей. // Программирование. т. 37 №5 - Москва: МАИК "Наука/Интерпериодика", 2012.
[80] J. Jacky, "PyModel: Model-based testing in Python", Northwest Python Day 2010.

[81] Errfix: библиотека классов для тестирования с использованием моделей на Ruby. https://code.google.com/p/errfix/
[82] I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS, volume 1708, Springer-Verlag, 1999, pp. 608–621.
[83] А.И.Гриневич, В.В.Кулямин, Д.А.Марковцев, А.К.Петренко, В.В.Рубанов, А.В.Хорошилов. Использование формальных методов для обеспечения соблюдения программных стандартов.// Труды Института системного программирования РАН, №10, 2006.
[84] В.В.Кулямин, А.К.Петренко, В.В.Рубанов, А.В.Хорошилов. Формализация интерфейсных стандартов и автоматическое построение тестов соответствия. Информационные технологии, 8:2-7, М. Новые технологии, 2007.

Automation of conformance testing for communication protocols

Nikeshin A.V., Pakulin N.V., Shnitman V.Z.
alexn@ispras.ru, npak@ispras.ru, vzs@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This article summarizes the experience gained by the UniTESK team in development of model-based test suites for conformance testing of Internet protocols. Here we call “Internet protocols” the protocols of TCP/IP stack starting network level. We performed testing of network-level protocols IPv6 and Mobile IPv6, security protocols , IPsec v2 and TLS/SSL, mail protocols SMTP, POP3 and IMAP. The projects described in this article used the UniTESK technology as a base for test construction: we modeling protocol under test as a state machine with implicitly defined transitions, and the tests are constructed as traversal of test state machine. During the development of test suites we identified certain features of the protocols that make it difficult to apply UniTESK unmodified to protocol conformance testing, as well as some limitations of UniTESK toolkits. Nevertheless all these obstacles were successfully overcome without going beyond UniTESK concepts.

Keywords: conformance testing, UniTESK, formal methods, test automation, Model-based testing, protocol testing, formal specification, protocol models.

References

[1]. CCITT Recommendation Z.100. Specification and Description Language (SDL). Geneve, Switzerland: ITU, 1993. 245 p.
[2]. ISO/IEC 9074. Information Processing Systems — Open Systems Interconnection. Estelle — A Formal Description Technique based on an Extended State Transition Model. Geneve, Switzerland: ISO, 1989 1 я редакция,1997 2-я редакция. Отозван 06.05.1999.
[3]. ISO/IEC 8807. Information Processing Systems — Open Systems Interconnection. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Geneve, Switzerland: ISO, 1989. 142 p.
[4]. ITU-T X.680 (11/08) Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T, 2008. 194 p.
[5]. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure . OMG, 2011. 230 p.
[6]. Edmund M. Clarke, Orna Grumberg and Doron Peled. Model checking. MIT Press, 1999. 325 pp.

[7]. Karpov Yu. MODEL CHECKING. BHV-Peterburg, 2010, 560 pp.
[8]. M. Diaz. Petri Nets: Fundamental Models, Verification and Applications. Willey, 2013г. 656с.
[9]. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007).
[10]. Blackburn, M., Busser, R., Nauman, A.: Why Model-Based Test Automation is Different and What You Should Know to Get Started. Software Productivity Consortium, NFP (2004).
[11]. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, p.M., Patton, G.C., Horowitz, B.M.: Model-Based Testing in Practice. In: Proceedings of the ICSE 1999 (May 1999).
[12]. I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77-88, Springer-Verlag, 2002.
[13]. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov. 2003. The UniTesK Approach to Designing Test Suites. Program. Comput. Softw. 29, 6 (November 2003), 310-322.
[14]. V. Kuliamin, A. Petrenko. Evolution of UniTESK Test Development Technology. ISP RAS Proceedings, vol. 26, issue 1, 2014. pp. 9-26. DOI: 10.15514/ISPRAS-2014-26(1)-1. (in Russian).
[15]. F. p. Hennie. Fault detecting experiments for sequential circuits. Proc. 5-th Ann. Symp. Switching Circuit Theory and Logical Design, 1964. p. 95-110.
[16]. M. Vasilevsky. Error diagnostics in state machines. Cybernetics and system analysis, vol. 9, № 4, 1973. pp. 98-108 (in Russian).
[17]. T. S. Chow. Testing software design modeled by finite-state machines. IEEE Trans. on Software Engineering, vol. 4, no. 3, 1978. p. 178-187.
[18]. J. A. Bergstra, J. W. Klop. Algebra of Communicating Processes with Absraction. Theoretical Computer Science, 37(1), 1985. pp. 77-121.
[19]. p. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985; e-book http://www.usingcsp.com/cspbook.pdf, 2004. 260 p.
[20]. R. Milner. Communication and Concurrency. Prentice-Hall, 1989. 260 p.
[21]. N.A. Lynch, M.R. Tuttle, "An Introduction to Input/Output Automata" CWI-Quaterly 3, 1989, P. 219-246. [PDF] http://www.markrtuttle.com/papers/lt89-cwi.pdf.
[22]. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. CONCUR'92 Proceedings, LNCS 630. Springer-Verlag, 1992.
[23]. ISO/IEC 7498. Information technology – Open Systems Interconnection – Basic Reference Model. Geneva, Switzerland: ISO, 1994.
[24]. Ed. Brinksma. A theory for the derivation of tests. Proc. IFIP WG6.1 8th Intl. Symp. on Protocol Specification, Testing, and Verification, North-Holland, S. Aggarwal and K. Sabnani Ed. pp. 63-74, 1988.
[25]. K. K. Sabnani and A. T. Dahbura. A protocol test generation procedure. Computer Networks and ISDN Systems, vol. 15, no. 4, pp. 285-297, 1988.
[26]. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. IEEE Trans. on Software Eng., vol. 17, pp. 591-603,1991.
[27]. G. Luo, A. Petrenko, and G. v. Bochmann. Selecting test sequences for partially specified nondeterministic finite state machines, Proceedings of the IFIP Seventh International Workshop on Protocol Test Systems, Japan, 1994, pp. 95-110.
[28]. N. V. Evtushenko, A. V. Lebedev, A. F. Petrento, "Development of the checking set for a component of a sequential automaton network". Automation and Remote Control, 1994, 55:8, 1203–1210

[29]. J. C. Fernandez, C. Jard, T. Jeron, C. Viho. Using on-the-Fly Verification Techniques for the Generation of Test Suites Proceedings of the 8th International Conference on Computer Aided Verification, LNCS 1102, Springer-Verlag, 1996, P. 348-359.
[30]. J. Tretmans. Test Generation with Inputs, Outputs, and Repetitive Quiescence. Software — Concepts and Tools, 17(3):103-120, 1996.
[31]. D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: a detailed study. IEEE Trans. Soft. Eng., vol. 15, no. 4, pp. 413-426, 1989.
[32]. D. Lee, M. Yannakakis. Principles and methods of testing finite state machines — a survey. Proc. IEEE, 84(8):1090-1123, 1996.
[33]. G. v. Bochmann, A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. Proc. of ACM SIGSOFT ISSTA'1994, Software Engineering Notes, Special Issue, pp. 109—124.
[34]. A. Petrenko. Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In F. Cassez, p. Jard, B. Rozov, M. Dermot, eds. Modeling and Verification of Parallel Processes: 4-th Summer School, Nantes, France, LNCS 2067, pp. 196-200, Springer-Verlag, 2000.
[35]. ISO/IEC 9646. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts. Geneva: ISO, 1994. 46 p.
[36]. ITU-T Recommendation Z.500. Framework on formal methods in conformance testing. Geneve, Switzerland: ITU, 1997. 49 p.
[37]. Glenford J. Myers, Corey Sandler, Tom Badgett. The Art of Software Testing, 3rd Edition. Wiley, 2011. p. 240.
[38]. J. Tretmans, An Overview of OSI Conformance Testing. Translated and adapted from: J. Tretmans and J. van de Lagemaat, Conformance Testen, in Handboek Telematica, Vol. II, pages 1--19. Samson, 1991.
[39]. ISO/IEC 9646. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 2: Abstract Test Suite specification. Geneva: ISO, 1994. 33 p.
[40]. Information technology – Open systems interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN). 1-е издание. Geneva, Switzerland: ISO, 1992.
[41]. Information technology – Open systems interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN). 2-е издание. Geneva, Switzerland: ISO, 1998.
[42]. ETSI ES 201 873-1 V3.1.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia-Antipolis, France: ETSI, 2005. 210 p.
[43]. OMG formal/05-07-07. UML Testing Profile. Version 1.0. Needham, USA: Open Management Group, 2005. [PDF, PostScript] (http://www.omg.org/cgi-bin/doc?formal/05-07-07).
[44]. L. Ebrecht, M. Schacher, p. Bühler. Test Specification in XML – the most important Element for Test Automation. ARTiSAN Benutzerforum D.A.CH, 2005.
[45]. JUnit testing framework. http://www.junit.org.
[46]. IPv6 Test Suite TAHI. http://www.tahi.org/.
[47]. Digital cellular telecommunications system (Phase 2+) (GSM); Handover procedures (GSM 03.09 version 5.1.0) ETSI, Sophia-Antipolis, France, 1997. 81 p.

[48]. ETSI ETS 300 838. Integrated Services Digital Network (ISDN); Harmonized Programmable Communication Interface (HPCI) for ISDN. ETSI, Sophia-Antipolis, France, 1998. 546 p.
[49]. C. Jard, T. Jéron. TGV: Theory, principles and algorithms. International Journal on Software Tools for Technology Transfer (STTT), vol. 7(4). Berlin: Springer, 2005. p. 297 – 315
[50]. E. Farchi, A. Hartman, S.S. Pinter. Using a Model-based Test Generator to Test for Standard Conformance. IBM System Journal - special issue on Software Testing. Volume 41(1), 2002. p. 89 - 110.
[51]. G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected State Machine Coverage for Software Testing. Proceedings of ISSTA 2002 International Symposium on Software Testing and Analysis. New York, USA: ACM Press, 2002. p. 134 – 143.
[52]. Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, in Formal Methods and Testing, vol. 4949, pp. 39-76, Springer Verlag, 2008
[53]. A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink. Formal test automation: A simple experiment. G. Csopaki, S. Dibuz, K. Tarnay, editors. 12th Int. Workshop on Testing of Communicating Systems. Budapest, Hungary: Kluwer Academic Publishers, 1999. p. 179-196.
[54]. Axel Belinfante. JTorX: a Tool for On-Line Model-Driven Test Derivation and Execution. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010. LNCS vol. 6015, pp. 266-270. Springer.
[55]. N. Goga. A probabilistic coverage for on-the-fly test generation algorithms. Automated Verification of Critical Systems (AVoCS '03), 2003.
[56]. N. V. Pakulin and A. V. Khoroshilov. 2007. Development of formal models and conformance testing for systems with asynchronous interfaces and telecommunications protocols. Program. Comput. Softw. 33, 6 (November 2007), 316-335.
[57]. V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin, A. S. Kossatchev, I. B. Bourdonov. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. Proceedings of the 5-th International Conference on Perspectives of System Informatics, July 9-12, 2003, Novosibirsk, Russia; LNCS 2890, Springer, 2003, pp. 450-461.
[58]. V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proceedings of the 2-nd International Service Availability Symposium, April 25-26, 2005, Berlin, Germany; LNCS 3694, Springer, 2005, pp. 68-83.
[59]. V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. Proceedings of the 9-th World Multiconference on Systemics, Cybernetics, and Informatics, Model Based Development and Testing Workshop, July 10-13, 2005, Orlando, Florida, USA, pp. 65-70
[60]. C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, volume 12 №10, 1969. p. 576-580.
[61]. R. P. Draves, A. Mankin, B. D. Zill. Implementing IPv6 for Windows NT. Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA, August 3–4, 1998
[62]. I. Agamirzian, S. Groshev, A. Khoroshilov, G. Kluchnikov, A. Kossatchev, V. Omeltchenko, N. Pakulin, A. Petrenko, V. Shnitman. Using formal methods for MSR IPv6 conformance testing. Proc. of Next Generation Internet, Yaroslavl, 2002. Pp. 29-33 (in Russian)

[63]. G. Kluchnikov, A. Kossatchev, A. Petrenko, N. Pakulin, V. Shnitman. Using formal methods in testing of an IPv6 implmentation. ISPRAS Proceedings, vol. 4, pp. 121-140, 2003 (in Russian).
[64]. G. Kluchnikov, A. Kossatchev, A. Petrenko, N. Pakulin, V. Shnitman. Using formal methods in testing of Mobile IPv6 . Proc. of Next Generation Internet-2, Yaroslavl, pp. 20-25, 2003 (in Russian).
[65]. D. Zatsepin, V. Shnitman. Aspects of applications of UniTESK to testing of mobility functions of IPv6 protocol. ISPRAS Proceedings, vol. 13(1), pp. 143-170, 2007 (in Russian)
[66]. CTesK 2.1: SeC Language Reference. ISPRAS, 2005. 167 p.
[67]. G. Kluchnikov, N. Pakulin, V. Shnitman. Automated testing of network services of Internet protocols. Proc. of Scientific services in Internet — 2005, pp. 168-170 (in Russian).
[68]. N. Pakulin. Applying formal methods to testing of implementations of complex modern protocols. Proc. of int. workshop «Go4IT — towards new Internet technologies» 2007, pp. 11-18 (in Russian).
[69]. A. Nikeshin, N. Pakulin, V. Shnitman. Aspects of testing of network-level security services of IPsec version 2. Proc. of Scientific services in Internet — 2008 (in Russian).
[70]. A. Nikeshin, N. Pakulin, V. Shnitman. Development of a test suite for verification of IPsec v2 implementations. ISPRAS Proceedings, vol. 18, 2010, pp. 151-182 (in Russian).
[71]. A. Nikeshin, N. Pakulin, V. Shnitman. Verification of security functions of IPsec v2. Programmirovanie, vol. 37 № 1. 2011. pp. 36-56 (in Russian)
[72]. Bourdonov I.B., Demakov A.V., Jarov A.A., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. and Zelenov S.V. Java Specification Extension for Automated Test Development Proceedings of PSI'2001. Novosibirsk, Russia July 2-6 2001, LNCS 2244:301-307. Springer-Verlag, 2001.
[73]. A. Nikeshin, N. Pakulin, V. Shnitman. Test Suite development for verification of TLS security protocol. IPSRAS Proceedings, vol. 23, pp. 387-404. 2012. DOI: 10.15514/ISPRAS-2012-23-22. (in Russian).
[74]. A. Tugaenko. Conformance testing of extensible mail Internet protocols. Proceedings of Young Researchers Forum «Lomonosov 2010». (in Russian)
[75]. N. Pakulin, A. Tugaenko. Development of conformance test suites for email protocols. Proceedings of APPI-2209, pp 154-160 (in Russian)
[76]. A. Tugaenko. Method for conformance testing of extensible Internet protocols. Proceedings of Young Researchers Forum «Lomonosov 2011». (in Russian)
[77]. N. Pakulin, A. Tugaenko. Model Based Conformance Testing for Extensible Internet Protocols Proceedings of SYRCoSE 2011.
[78]. N. Pakulin, A. Tugaenko, V. Shnitman. Model-based testing of internet e-mail protocols. Proceedings of ISPRAS, vol 20, 2011. p. 125-141. (in Russian)
[79]. N. Pakulin, A. Tugaenko, V. Shnitman. Model-based testing of internet e-mail protocols. Programming and Computer Software, vol. 38, №5, 268-275. 2012
[80]. J. Jacky, "PyModel: Model-based testing in Python", Northwest Python Day 2010.
[81]. Errfix: model-based testing in Ruby. https://code.google.com/p/errfix/
[82]. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS, volume 1708, Springer-Verlag, 1999, pp. 608–621.

[83]. A.Grinevich, V.Kuliamin, D.Markovcev, A.Petrenko, V.Rubanov, A.Khoroshilov. Using formal methods to ensure conformance for programming standards. // Proceedings of ISPRAS, №10, 2006.
[84]. V.Kuliamin, A.Petrenko, V.Rubanov, A.Khoroshilov. Formalization of interface standards and automated construction of conformance tests. Information technologies. 8:2-7, Moscow, New Technologies, 2007.

Средства функциональной верификации микропроцессоров

А.С. Камкин, А.М. Коцыняк, С.А. Смолов, А.А. Сортов, А.Д. Татарников, М.М. Чупилко
{kamkin,kotsynyak,ssedai,sortov,andrewt,chupilko}@ispras.ru

Аннотация. Обеспечение корректности микропроцессоров и другой микроэлектронной аппаратуры является фундаментальной проблемой, для решения которой применяют разнообразные средства функциональной верификации. В отличие от программ, ошибки в которых исправляются сравнительно просто, дефекты в интегральных схемах (конструктивные и производственные) не могут быть устранены. Несмотря на то, что постоянно совершенствуются системы автоматизированного проектирования (САПР), инструменты генерации тестов и методы анализа схем, верификация остается самым узким местом процесса разработки (на нее тратится около 70% всех ресурсов проектирования). В работе делается краткий обзор средств верификации микропроцессоров, рассматриваются проблемы, возникающие в промышленной практике, анализируются возможные пути их решения. Значительная часть статьи посвящена исследованиям по верификации аппаратуры, проводимым в ИСП РАН: подводятся итоги выполненных работ, описываются текущие разработки, формулируются направления дальнейших исследований.

Ключевые слова: микропроцессоры; цифровая аппаратура; верификация; валидация; тестирование; генерация тестов; моделирование; языки описания архитектуры; распараллеливание.

1. Введение

Верификацией называется проверка соответствия результатов, полученных на отдельных этапах проектирования (разработки) программных и аппаратных систем, требованиям и ограничениям, установленным для них на предыдущих этапах (на начальном этапе проверяется соответствие исходным требованиям — техническому заданию) [1]. Основной задачей верификации является контроль качества проектирования, включая такие его аспекты, как корректность, надежность, производительность, энергопотребление, эргономика и многие другие. В рамках настоящей статьи рассматривается лишь один из них — *функциональная корректность*. Комплекс мер, нацеленный на обеспечение корректности разрабатываемой системы (прежде всего, на обнаружение и исправление ошибок проектирования), называется

функциональной верификацией (в дальнейшем под верификацией будет пониматься именно функциональная верификация).

Год	***Микропроцессор***	***Транзисторы***
1971	4004	2 300
1974	8080	5 000
1978	8086	29 000
1982	186	55 000
1982	286	134 000
1985	386	275 000
1989	486	1 180 235
1993	Pentium	3 100 000
1997	Pentium II	7 500 000
1999	Pentium III	24 000 000
2000	Pentium 4	42 000 000
2001	Itanium	25 000 000
2002	Itanium 2	220 000 000
2004	Itanium 2 9M	592 000 000
2006	Core 2 Duo (2 ядра)	291 000 000
2008	Core i7 (4 ядер)	731 000 000
2011	Core i7 (6 ядер)	2 270 000 000
2012	Xeon Westmere-EX (8 ядер)	2 600 000 000
2012	Xeon Phi (62)	5 000 000 000

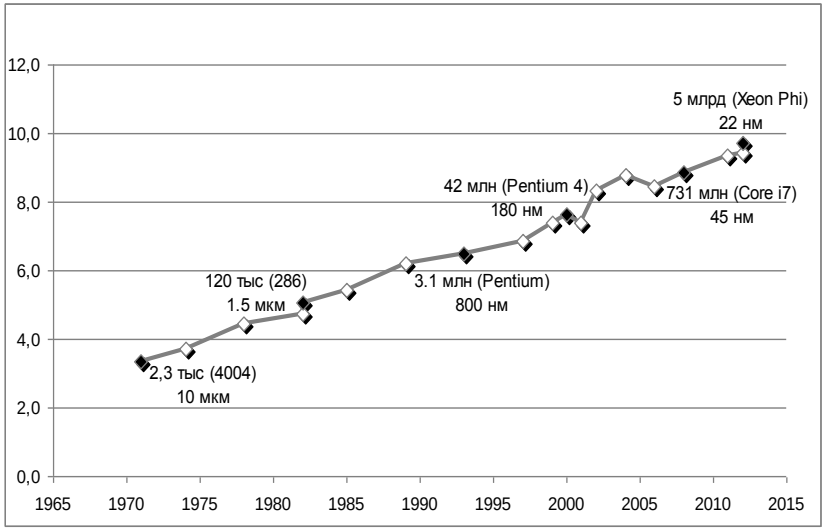


Рис. 1. Рост числа транзисторов (на графике — десятичный логарифм) в микропроцессорах фирмы Intel [2]

Тема статьи ограничена не только видом верификации, но и типом рассматриваемых систем — *микропроцессорами*, программно управляемыми устройствами, предназначенными для цифровой обработки данных. Поскольку функциональность микропроцессора определяется реализуемой им системой команд, то, в самых общих словах, задача верификации состоит в проверке того, что микропроцессор (точнее, его проектная модель или схема) корректно реализует все указанные в техническом задании команды (инструкции): результат выполнения каждой команды во всех возможных ситуациях соответствует ее спецификации [3]. Верификация микропроцессора — чрезвычайно трудоемкая задача. По некоторым оценкам, затраты на нее составляют порядка 70% от всех затрат на проектирование, число инженеров-верификаторов примерно вдвое превосходит число инженеров-разработчиков, а исходный код тестовых систем составляет до 80% от общего объема кода [4]. С ростом сложности микропроцессоров (закон Мура [5] работает до сих пор — см. рис. 1) ситуация только ухудшается — возможности методов верификации отстают от развития микропроцессоров; соответственно, проверка корректности (и без того являющаяся самым узким местом процесса проектирования) вовлекает в себя все бóльшие объемы ресурсов. К примеру, над верификацией микропроцессора Pentium 4 (2000 г.) работала команда, состоявшая приблизительно из 70 человек, а для прогона тестов использовалось около 6 тысяч компьютеров, работавших в круглосуточном режиме [6, 7].

Почему верификация микропроцессоров так важна? Потому что никто не хочет доверять свою жизнь, здоровье и благополучие системам, содержащим ошибки, которые при некоторых обстоятельствах могут повести себя непредсказуемым образом, а микропроцессоры — это основа основ всех компьютерных систем. Неслучайно пользователи компьютеров (которых с каждым годом становится все больше и больше) очень бурно реагируют на ошибки, обнаруживаемые в микропроцессорах. Показательна в этом плане ошибка в реализации команды деления в микропроцессоре Pentium компании Intel, обнаруженная в 1994 г. [8]. Несмотря на то, что большинства пользователей эта проблема не касалась, и вероятность ее возникновения крайне мала, для сохранения имиджа компании Intel пришлось организовать замену микросхем, что обошлось ей в 475 миллионов долларов. В 2007 г. много шума наделала ошибка в реализации механизмов кэширования 4-ядерного микропроцессора AMD Phenom (ошибка #298 [9]), которая может приводить к зависанию системы или порче данных. В 2008 г. в Сети обсуждалась близкая проблема в Intel Core i7 (Nehalem), но, как выяснилось, тревога оказалась ложной [10]. С другой стороны, в Core i7 хватает и других проблем — в спецификации [11] (2011 г.) перечислены 153 ошибки, из которых только 16 имеют статус «исправлена» и 2 «исправление запланировано». Следует понимать, что это лишь известные проблемы — общее число ошибок может быть существенно больше.

Трудоемкость и актуальность верификации стимулируют научные исследования в этой области. Основная тенденция в развитии средств верификации состоит в усилении роли *формальных методов* — методов, основанных на анализе математических (формальных) моделей систем, модулей и протоколов их взаимодействия [12]. В рамках формальной верификации используются специфические техники, такие как *проверка моделей*, *дедуктивный анализ*, *проверка эквивалентности* и другие [1]. Использование формальных методов требует значительных усилий на построение моделей, однако если модели построены, то их анализ в значительной мере может быть выполнен автоматически. Лидером в области формальной верификации микропроцессоров является компания Intel, которая формально проверяет широкий класс устройств: от модулей арифметики с плавающей точкой до протоколов обеспечения когерентности памяти [13]. К 2015 г. Intel планирует увеличить использование формальных методов в модульной верификации микропроцессоров до 50% [14, 15]. Методы тестирования по-прежнему доминируют, но следует отметить, что за последние 20 лет они претерпели значительную трансформацию: современные подходы сочетают в себе как эвристические, так и формальные техники [16].

Свой вклад в развитие средств функциональной верификации микропроцессоров вносит и ИСП РАН, занимающийся этой тематикой с 2005 г. Сфера интересов Института включает технологии промышленной верификации микропроцессоров на модульном и системном уровнях, а также формальные методы проектирования и верификации микропроцессоров. Помимо выполнения теоретических исследований и разработки инструментальных средств Институт сотрудничает с ведущими отечественными производителями микропроцессоров, НИИСИ РАН и ЗАО «МЦСТ», выполняя для них проекты по верификации с использованием разработанных средств. В статье описываются результаты, полученные Институтом в области верификации микропроцессоров, рассматриваются текущие разработки, обрисовываются направления дальнейших исследований.

Статья организована следующим образом. Раздел 2 представляет собой краткое введение в предметную область. В разделе 3 описываются выполненные исследования и разработки. Раздел разбит на два подраздела, посвященных модульной и системной верификации соответственно. Каждый подраздел содержит несколько частей, отражающих основные вехи проделанных работ. Сравнение предложенных методов и реализованных инструментов с существующими подходами осуществляется непосредственно в той части, которая описывает предлагаемое решение. Раздел 4 рассматривает текущие и перспективные разработки Института. Как и раздел 3, он разбит на два подраздела, посвященных модульной и системной верификации. Раздел 5 завершает статью.

2. Средства верификации микропроцессоров

В общих словах, процесс проектирования микропроцессора состоит из четырех основных этапов, на каждом из которых создается его модель определенного уровня абстракции: (1) *архитектурное проектирование*, (2) *детальное проектирование*, (3) *логическое проектирование*, (4) *физическое проектирование* [3]. Рассмотрим первые два этапа более подробно (этапы логического и физического синтеза автоматизированы средствами современных САПР и в данной статье не рассматриваются).

На этапе архитектурного проектирования разрабатывается система команд микропроцессора (*макроархитектура*) и уточняется его внутренняя структура (*микроархитектура*). Основными средствами, используемыми на данном этапе, являются (1) *языки программирования общего назначения*, (2) *языки системного проектирования* (*SLDL, System-Level Design Languages*) и (3) *языки описания архитектуры* (*ADL, Architecture Description Languages*) [17, 18, 19]. Примеры языков указанных типов представлены в табл. 1. Результатом архитектурного проектирования является *симулятор микропроцессора* — программная модель, позволяющая интерпретировать программы, написанные в соответствующей системе команд. Симуляторы микропроцессоров используются для кросс-разработки ПО, а также для верификации, где они выступают в качестве эталонных моделей (см. раздел 2.3).

Табл. 1. Языки, используемые при проектировании микропроцессоров

Тип языка	***Примеры***
Языки программирования общего назначения	C, C++, Perl, Python
Языки описания архитектуры (ADL)	LISA, EXPRESSION, ISDL, nML
Языки системного проектирования (SLDL)	SystemC, SystemVerilog, Bluespec
Языки описания аппаратуры (HDL)	Verilog, VHDL

На этапе детального проектирования применяются *языки описания аппаратуры* (*HDL, Hardware Description Languages*), такие как VHDL и Verilog, позволяющие предельно точно описывать структуру и поведение микропроцессора [17]. Результатом этапа является *модель уровня регистровых передач* (*RTL, Register Transfer Level*), которая с потактовой точностью определяет пересылки данных, возникающие при работе устройства. RTL-модель (называемая также *HDL-моделью* или *HDL-описанием*) преобразуется (посредством логического и физического синтеза) в представление (как правило, основанное на *фотошаблонах*), используемое при производстве интегральных схем. Хотя верификация присутствует на всех этапах разработки, особенно она актуальна при создании HDL-модели, поскольку функциональность, описанная на этом этапе, впоследствии не изменяется [3].

2.1. Методы верификации микропроцессоров

Существующие методы верификации микропроцессоров можно разбить на три основных класса: (1) *экспертиза*, (2) *имитационная верификация* (*simulation-based verification*), также называемая *динамической верификацией* или *тестированием*, и (3) *формальная верификация*. Кроме того, существуют так называемые *гибридные методы* [16] (другие названия — *синтетические методы* [20] и *полуформальные методы* [21]), которые используют комбинации указанных подходов (прежде всего, комбинации имитационных и формальных методов).

К *экспертизе* относятся методы верификации, в которых оценка результатов проектирования выполняется людьми путем умозрительного анализа (инспекция кода, визуальный анализ схем и т.п.). Отличительной чертой экспертизы является возможность ее выполнения с использованием только результатов проектирования, а не их формальных моделей (как в формальной верификации) или результатов работы (как в имитационной верификации). Гипотетически, экспертиза позволяет выявлять практически любые виды ошибок, причем на самых ранних стадиях. В то же время она не может быть автоматизирована и ее эффективность существенно зависит от опыта и мотивации ее участников [1].

Под *имитационной верификацией* обычно понимается тестирование HDL-моделей аппаратуры, выполняемое в специальной среде имитационного моделирования — *HDL-симуляторе* [4]. Для применения этого подхода необходимо иметь модель микропроцессора, что невозможно на ранних этапах проектирования. Создание набора тестов, позволяющих адекватно проверить такое сложное устройство, как микропроцессор, является чрезвычайно трудоемкой задачей. Однако наличие простых методик (например, *случайная генерация тестов*), а также возможность проверить реальное поведение на реальных примерах являются основными причинами, по которым имитационная верификация широко распространена (более подробно этот класс методов рассмотрен в разделах 2.2 и 2.3).

Формальная верификация основана на построении математической (формальной) модели системы и ее анализе на предмет выполнимости свойств, также выраженных формально (например, свойств *безопасности* (*safety*) — недостижимости ошибочных состояний — и *живости* (*liveness*) — отсутствия зависаний и зацикливаний) [22]. Модель может разрабатываться вручную или извлекаться из HDL-описания устройства [23]. Преимущество формальной верификации состоит в том, что проверка осуществляется для всех возможных вариантов поведения модели, что позволяет считать верификацию исчерпывающей (для заданных модели и свойств) [24]. Недостатком является высокая трудоемкость, связанная с необходимостью разработки формальной модели (если модель не извлекается из исходного кода) и обоснованием ее адекватности (эквивалентности исходному HDL-описанию) [25].

В табл. 2 представлено распределение найденных ошибок в зависимости от используемого метода верификации в проекте Pentium 4 компании Intel (2000 г.) [6]. Согласно представленным данным, 74% ошибок находится с помощью имитационной верификации, причем бóльшая их часть приходится на модульную верификацию (автономную проверку модулей микропроцессора). Формальные методы применялись ограниченно, поэтому процент ошибок, найденных таким способом, невелик (около 6%), однако это такие ошибки, которые практически невозможно обнаружить с помощью других средств. Несмотря на то, что статистике, приведенной в табл. 2, более 10 лет, она все еще сохраняет свою актуальность (особенно для отечественных компаний, которые уступают ведущим мировым компаниям по уровню развития технологий верификации[1]).

Табл. 2. Метод верификации — число найденных ошибок (Pentium 4, 2000 г.) [6]

Метод верификации	***Число ошибок***	***Процент ошибок***
Имитационная верификация (модульный уровень)	3411	43.5%
Имитационная верификация (системный уровень)	2398	30.5%
Экспертиза (инспекция кода)	1554	20.0%
Формальная верификация	492	6.0%
Итого	**7855**	**100%**

Доминирующим подходом (даже в таких компаниях, как Intel, AMD и IBM) по-прежнему является имитационная верификация. Чтобы повысить ее эффективность микропроцессор декомпозируется на множество относительно простых модулей (устройств, блоков, подсистем, кластеров), каждый из которых проверяется автономно. Таким образом, помимо обязательной *системной верификации*, оценивающей работоспособность микропроцессора в целом, применяют еще и *модульную верификацию* для более тщательной проверки отдельных устройств. Ниже кратко рассматривается, как имитационная верификация реализуется на модульном и системном уровнях.

2.2. Средства модульной верификации

Имитационная верификация на *модульном уровне* осуществляется с помощью *тестовых систем* (*testbenches*) — специализированных программ, которые в автоматическом режиме подают на верифицируемую HDL-модель *тестовые воздействия* (*стимулы*) и проверяют корректность выдаваемых ею *ответов*

[1] В условиях ограниченного финансирования обычно экономят на верификации.

(*реакций*) [4]. Типичная архитектура тестовой системы представлена на рис. 2. Тестовая система выполняется в HDL-симуляторе и эмулирует *окружение* тестируемой HDL-модели. В общих словах, она решает три задачи: (1) генерация тестовой последовательности, (2) проверка корректности поведения HDL-модели и (3) оценка полноты тестирования. За решение каждой задачи отвечает свой компонент тестовой системы: соответственно *генератор стимулов*, *тестовый оракул* и *сборщик тестового покрытия*.

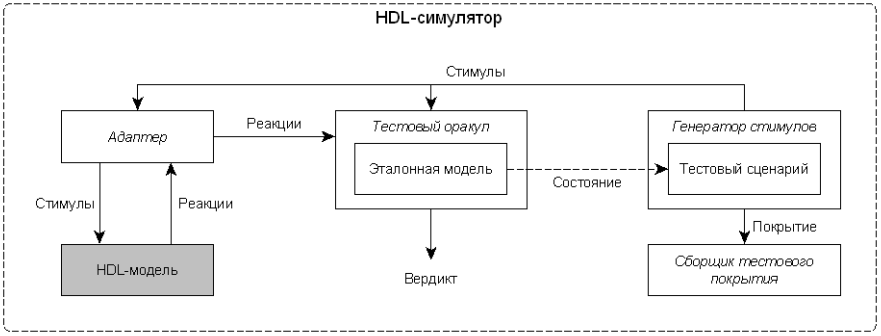


Рис. 2. Архитектура тестовой системы (системы имитационной верификации)

Генератор стимулов на основе *тестовых сценариев* или иных *параметров генерации* создает поток стимулов на тестируемую HDL-модель. Цель генератора — реализовать максимально возможное число ситуаций в работе устройства. В зависимости от используемой технологии сценарии (или непосредственно генераторы) описываются разными способами: используя техники *случайной генерации*, в том числе *случайной генерации на основе ограничений* [26] или на основе *конечно-автоматных моделей* (*графов состояний*). Первый подход используется в технологии UVM (Universal Verification Methodology) [27], продвигаемой организацией Accellera Systems Initiative; второй — в технологии UniTESK (Unified TEsting and Specification Kit) [28], разрабатываемой в ИСП РАН (см. раздел 3.1). Перспективным направлением является генерация тестов на основе статического анализа HDL-описания (см. раздел 4.1). Дополнительное представление о методах построения тестов для микропроцессоров дает раздел 2.3, посвященный системной верификации.

Тестовый оракул проверяет корректность поведения HDL-модели и выносит *вердикт* о его соответствии (или несоответствии) требованиям. Как правило, требования задаются посредством *эталонной модели* — упрощенной реализации устройства на языке программирования (C или C++) или SLDL-языке (SystemC или SystemVerilog). Также используются спецификации в декларативной форме: *расширенные регулярные выражения* [29],

контрактные спецификации [30], *системы правил* [31] и *темпоральные утверждения* [32]. Самым популярным формализмом, используемым для описания свойств аппаратуры, является *темпоральная логика линейного времени* (*LTL, Linear Temporal Logic*) [33], расширения которой используются в *языках верификации аппаратуры* (*HVL, Hardware Verification Languages*): ForSpec [34], OpenVera [35], Property Specification Language (PSL) [36], *e* [37], SystemVerilog[2] [38].

Сборщик тестового покрытия предназначен для оценки полноты тестирования и принятия решения о его завершении. Для этого используются количественные показатели, называемые *метриками* [39]. Метрики могут определяться на основе разных артефактов проектирования (HDL-описания, эталонной модели, формальной спецификации или документации). Основная идея состоит в следующем. Некоторым систематическим образом определяется набор *тестовых ситуаций*, составляющих в совокупности *тестовое покрытие*. Для заданного выполнения тестовой системы (и HDL-модели) метрика возвращает число реализованных ситуаций. Цель тестирования — покрыть все ситуации в рамках выбранной модели покрытия. Метрики на основе реализации называются *структурными,* а метрики на основе спецификации — *функциональными*. Широко используются метрики на основе кода (HDL-описания и эталонной модели): покрытие строк кода, ветвей и путей в графе потока управления, условий. Для аппаратуры большую роль играют также метрики на основе автоматных моделей: покрытие состояний и переходов.

Как правило, тестовые системы разрабатываются не на уровне RTL, а на более высоком уровне абстракции — *уровне транзакций* (*TLM, Transaction Level Modeling*) [40, 25]. TLM-модели основаны на парадигме *передачи сообщений*: компоненты модели соединяются *каналами* и взаимодействуют друг с другом посредством *транзакций* (посылки и приема сообщений). Соответственно, стимулы и реакции внутри тестовой системы имеют форму сообщений, а их передача на *входные интерфейсы* HDL-модели (логически связанные группы сигналов) и прием с *выходных интерфейсов* осуществляется с помощью специализированных каналов, *адаптеров* (*транзакторов*), инкапсулирующих детали преобразования данных между разными уровнями. Для разработки TLM-моделей используются SLDL-языки, а также языки программирования общего назначения (см. табл. 1).

---

[2] Язык SystemVerilog может использоваться как для проектирования аппаратуры, так и для верификации. Такие языки называются *языками описания и верификации аппаратуры* (*HDVL, Hardware Description and Verification Languages*).

2.3. Средства системной верификации

Верификация на *системном уровне*[3] осуществляется путем создания *тестовых программ* и анализа результатов их выполнения на HDL-модели микропроцессора [41, 42]. Как правило, тестовые программы разрабатываются на *языке ассемблера*, а их целью является создание разнообразных ситуаций в работе микропроцессора (особые случаи выполнения арифметических операций, возникновение прерываний, заполнение буферов, вытеснение данных из кэш-памяти, сложные взаимодействия между модулями и т.п.). Ввиду высокой трудоемкости верификации тестовые программы обычно генерируются автоматически. Тесты, разработанные вручную, используются для проверки сложно формализуемых и маловероятных ситуаций (такие тесты создаются с привлечением экспертного знания об особенностях реализации той или иной подсистемы микропроцессора). Для верификации также применяется код, полученный с помощью компиляции существующих программ на языках высокого уровня: известных библиотек (вроде glibc), архитектурно-независимых тестов (типа арифметического теста PARANOIA [43]), тестов на оптимизирующий компилятор (создаваемый для разрабатываемого микропроцессора) и других.

Для автоматического построения тестовых программ применяются следующие способы [44]: (1) *случайная генерация*, (2) *комбинаторная генерация*, (3) *генерация на основе шаблонов* и (4) *генерация на основе моделей*. *Случайная генерация* является самым распространенным методом создания тестовых программ. Примерами генераторов случайных тестов являются RAVEN (Random Architecture Verification Engine) [45], разработанный в Obsidian Software и в настоящее время используемый в ARM, и INTEG [46], созданный в НИИСИ РАН. Идея *комбинаторной генерации* состоит в систематическом переборе тестовых программ небольшого размера [42]. Различные варианты этого подхода реализованы в инструменте MicroTESK (Microprocessor TEsting and Specification Kit) [47], разрабатываемом в ИСП РАН (см. раздел 3.2.1). Более общий подход к генерации тестовых программ основан на использовании *шаблонов* — абстрактных символических представлений тестовых программ. Построение программ по шаблонам базируется на *случайной генерации на основе ограничений* [26]. Примерами инструментов такого типа являются Genesys-Pro [41, 48] от IBM Research и MicroTESK (см. раздел 3.2.2). Для генерации высококачественных тестовых программ могут использоваться *модели уровня микроархитектуры* [49, 50]. Коммерческие инструменты такого типа нам не известны, однако исследования на эту тему ведутся (см. раздел 4.2.1).

[3] Для многоядерных микропроцессоров также применяется *верификация уровня ядра*. Сказанное в этом разделе относится как к системной верификации, так и к верификации уровня ядра.

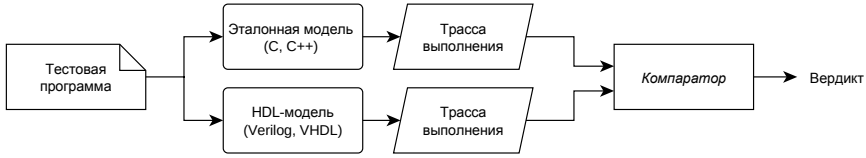


Рис. 3. Системная верификация посредством сравнения трасс

Для проверки корректности поведения HDL-модели микропроцессора применяются два основных метода: (1) сравнение *трасс выполнения* тестовых программ с эталонными трассами и (2) использование тестовых программ со *встроенными проверками* (*self-checking test programs*). В первом подходе (см. рис. 3) при выполнении программы на HDL-модели создается трасса выполнения, отражающая события, возникающие в микропроцессоре. Полученная трасса сравнивается с эталонной трассой — трассой, полученной при выполнении той же программы на *симуляторе микропроцессора*. Сравнение трасс осуществляется с помощью специальной программы, называемой *компаратором*. Во втором методе в код тестовой программы (вручную или автоматически) включаются проверки, которые необходимо осуществить в процессе выполнения программы. Программы со встроенными проверками можно использовать не только для тестирования проекта в HDL-симуляторе, но и для его верификации с помощью аппаратных ускорителей и опытных образцов интегральных схем (*post-silicon verification*). Точность проверок при этом снижается (программно можно наблюдать лишь малую часть событий, возникающих в микропроцессоре: запись данных в регистр или память, прерывание и другие), однако производительность значительно возрастает (год круглосуточного тестирования с использованием тысяч компьютеров соответствует нескольким минутам реальной работы микропроцессора [7]).

Для принятия решения о завершении верификации, как и на модульном уровне, используются метрики тестового покрытия, однако они задают необходимое, но не достаточное условие сдачи проекта в производство (на системном уровне практически невозможно создать адекватную модель тестового покрытия). Большинство производителей микропроцессоров используют подход, основанный на измерении *частоты обнаружения ошибок*. Этот метод предполагает постоянный прогон случайно генерируемых тестов: когда частота обнаружения ошибок становится устойчиво низкой, и в течение длительного времени ошибки не проявляются, тестирование прекращается (при условии, что достигнуты показатели, заданные метриками). В начале 2000-х гг. в компании Motorola одним из необходимых условий сдачи проекта в производство была безошибочная работа HDL-модели в течение 40 миллиардов тактов на случайных тестах [51].

3. Выполненные исследования и разработки

Исследования по верификации аппаратуры ведутся в ИСП РАН с 2005 г., когда была показана применимость технологии UniTESK [28] к модульному тестированию HDL-моделей, и были разработаны средства интеграции тестовых систем, создаваемых с помощью инструмента CTESK [52], с HDL-симуляторами [53]. Немного позже, в 2006 г., начались совместные работы с НИИСИ РАН — организацией, занимающейся промышленной разработкой микропроцессоров. Темой первой из них стала генерация тестовых программ для подсистемы управления памятью (MMU, Memory Management Unit) MIPS-совместимого микропроцессора [54]. Вторая работа (которая тоже проводилась в 2006 г.) была посвящена модульной верификации буфера преобразования адресов (TLB, Translation Lookaside Buffer) [55]. Эти работы стали отправными точками дальнейших исследований и разработок в области верификации микропроцессоров, о которых рассказывается в этом разделе (следуя структуре статьи, сначала описываются средства модульной верификации, а потом — системной).

3.1. Разработки в области модульной верификации

При создании средств модульной верификации микропроцессоров и другой аппаратуры (на модульном уровне микропроцессорная специфика не заметна) нами использовался имеющийся задел в области тестирования программного обеспечения (ПО) — прежде всего, технология UniTESK [28], развиваемая в ИСП РАН с момента его основания в 1994 г. (технология является преемницей подхода KVEST [56]). Технология базируется на *контрактных спецификациях* в форме *пред-* и *постусловий* интерфейсных операций и на уникальных средствах генерации тестовых последовательностей, основанных на факторизации (обобщении) состояний тестируемой системы и обходе графа состояний с помощью *неизбыточных алгоритмов* [57, 58].

Контрактные спецификации и основанный на них процесс проектирования (Design by Contract) [59] были предложены Мейером в 1986 г. в результате прикладного развития идей Флойда, Хоара и Дейкстры 1960-х – 1970-х гг. о формализации программирования [60, 61, 62]. Компонент, предоставляя окружению некоторую операцию, указывает требования, которые должны быть выполнены окружением перед ее вызовом (*предусловие*); при выполнении заданных требований компонент, в свою очередь, гарантирует достижение определенного результата (*постусловие*). Спецификации такого типа широко используются в практике тестирования ПО, поскольку они привычны для разработчиков и позволяют автоматически строить тестовые оракулы.

3.1.1 Контрактные спецификации конвейера

Используя опыт тестирования ПО, прежде всего, реализаций коммуникационных протоколов (для которых, как и для аппаратуры,

характерны параллелизм и событийный характер поведения) [63], подход UniTESK был адаптирован для верификации HDL-моделей аппаратуры. Это потребовало уточнения понятия контракта. Классический контракт — это пара логических формул $\Phi = \langle \varphi, \psi \rangle$, определенных над множеством видимых переменных (параметров вызова операции, возвращаемого ей результата, глобальных переменных программы). Первая формула, φ, называется *предусловием*, а вторая, ψ, — *постусловием*. Контракт интерпретируется следующим образом: если непосредственно перед вызовом операции было выполнено предусловие φ, то непосредственно после выполнения операции должно быть выполнено постусловие ψ. Нарушение контракта трактуется как *ошибка*: нарушение предусловия — ошибка окружения, нарушение постусловия при выполненном предусловии — ошибка компонента.

В общем случае реализуемые аппаратурой операции не являются *атомарными* — это процессы, функционирующие в дискретном времени, которые в определенные моменты могут вступать во взаимодействие с окружением, выдавая наружу результаты выполнения элементарных действий — *микроопераций*. Следует обратить внимание на то, что процессы выполнения разных операций могут перекрываться по времени: для аппаратуры, ввиду физических особенностей ее реализации, характерен высокий уровень параллелизма, что на логическом уровне выражается в конвейерно-параллельной организации (см. рис. 4). Последнее обстоятельство определило выбор названия для предлагаемого метода формальной спецификации — *контрактные спецификации конвейера* [64] (другое название — *контрактные спецификации потактовой точности* [65]).

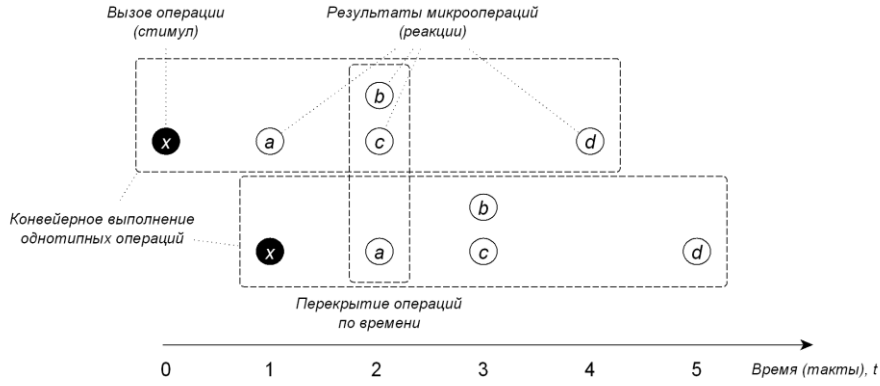


Рис. 4. Конвейерное выполнение двух однотипных операций

Для аппаратуры без внутренних блокировок контракт операции имеет вид $\Phi = \langle \varphi, \{(\psi_i, t_i)\}_{i=1,n} \rangle$, где φ — предусловие операции, ψ_i — постусловие i-ой микрооперации, t_i — время завершения выполнения i-ой микрооперации

(относительно времени вызова операции), n — общее число микроопераций [30]. Например, контракт операции x, представленной на рис. 4, имеет следующую структуру $\Phi_x = \langle \varphi_x, \{(\psi_a, 1), (\psi_b, 2), (\psi_c, 2), (\psi_d, 4)\}\rangle$. Семантику подобных контрактов можно определить в терминах логики LTL [33]. Так, контракт, приведенный выше, эквивалентен следующей формуле[4]: $\mathbf{G}(\phi_x \rightarrow \varphi_x) \wedge \mathbf{G}(\phi_x \rightarrow (\mathbf{X}\psi_a \wedge \mathbf{XX}(\psi_b \wedge \psi_c) \wedge \mathbf{XXXX}\psi_d))$, где ϕ_x обозначает факт вызова операции x. Другой способ определить семантику спецификаций — описать на их основе тестовый оракул. Пусть X — алфавит операций. Для $x \in X$ введем следующие обозначения: $\psi_x(t)$ — конъюнкция постусловий микроопераций, помеченных временно́й меткой t, T_x — время выполнения операции (время завершения последней микрооперации). Тестовый оракул работает синхронно с проверяемой системой и вызывается на каждом такте. Возможная организация выполняемой процедуры приведена в примере 1 (в начале работы $S = \varnothing$ и $v = true$; в случае ошибки в v заносится $false$).

Пример 1. Организация тестового оракула для операций без блокировок

```
01:  for x ∈ X do                                  проверка предусловий операций
02:      if φ_x ∧ ¬φ_x then v ← false
03:  for x ∈ X do                                  эмуляция вызова операций
04:      if φ_x then S ← S ∪ {(x, 0)}
05:  for (x, t) ∈ S do                             проверка постусловий микроопераций
06:      if ¬ψ_x(t) then v ← false
07:  S ← {(x, t+1) | (x, t) ∈ S ∧ t < T_x}         эмуляция такта работы системы
```

Если в аппаратуре возможны внутренние блокировки, формализация контракта операции немного усложняется. Предположим, что в рассматриваемом примере одноименные микрооперации, а также микрооперации a и b не могут выполняться одновременно (например, они используют общий ресурс), причем микрооперация a имеет больший приоритет по сравнению с b, а при других конфликтах приоритет получает та микрооперация, родительская операция которой была вызвана раньше. В такой ситуации конвейерное выполнение операций могло бы выглядеть, как показано на рис. 5. Формально блокировки конвейера могут быть описаны с помощью предусловий микроопераций, называемых также *охранными условиями* (*guards*) [64, 66]. Соответственно, контракт операции имеет вид

[4] Темпоральный оператор **G** (**G**lobally) требует, чтобы формула выполнялась всегда (начиная с текущего момента времени), **X** (ne**X**t time) — чтобы формула выполнялась в следующий момент времени.

$\Phi = \langle \varphi, \{(\gamma_i, \psi_i, t_i)\}_{i=1,n} \rangle$, где γ_i — охранное условие i-ой микрооперации. Охранные условия интерпретируются следующим образом: если для некоторой микрооперации завершились все предшествующие ей действия и выполнено охранное условие, в этот момент должны быть видимы результаты ее работы[5].

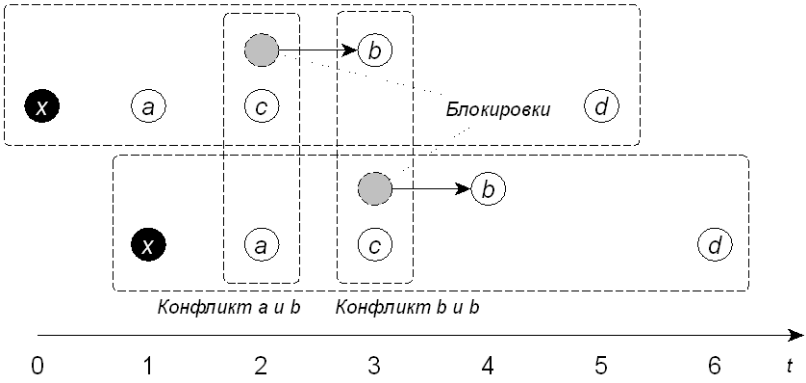


Рис. 5. Конфликты и блокировки при конвейерном выполнении операций

Выразить такие свойства с помощью LTL проблематично. Возможным вариантом видится замена подформул вида $\mathbf{X}\ldots\mathbf{X}\psi_i$ (используемых при описании операций без блокировок) на формулы[6] $\neg\gamma_i \mathbf{U} (\gamma_i \wedge \psi_i)$, однако и этот вариант не вполне адекватен — в нем не принимается во внимание то обстоятельство, что охранное условие микрооперации должно учитываться только после того, как завершились все предшествующие действия (вследствие этого возможны ложные обнаружения ошибок). В предположении, что у каждой выполняемой операции в любой момент времени в обработке находится не более одной микрооперации, семантику спецификаций можно выразить следующим алгоритмом (см. пример 2) работы тестового оракула (первые четыре строки полностью совпадают с описанием алгоритма для случая операций без блокировок).

---

[5] Возможна и другая интерпретация — результаты работы микрооперации становятся видимыми через такт после того, как охранное условие стало истинным.

[6] Бинарный темпоральный оператор **U** (**U**ntil) требует, чтобы первая формула была выполнена до тех пор, пока не станет истинной вторая формула (такой момент обязательно должен наступить).

Пример 2. Организация тестового оракула для операций с блокировками

…	…	…
05:	$E \leftarrow \{(x, t) \mid (x, t) \in S \wedge \gamma_x(t)\}$	вычисление множества активных микроопераций
06:	**for** $(x, t) \in E$ **do**	проверка постусловий активных микроопераций
07:	**if** $\neg\psi_x(t)$ **then** $v \leftarrow$ *false*	
08:	$S \leftarrow \{(x, t+1) \mid (x, t) \in E \wedge t < T_x\} \cup (S \backslash E)$	эмуляция такта работы системы

Контрактные спецификации могут быть расширены для операций со сложной структурой потока управления [67]. Поток управления описывается ориентированным графом, в котором возможны следующие типы вершин: начало и конец (*start* и *stop*), микрооперация (*stage*), ветвление и соединение потока управления (*switch* и *merge*), создание параллельных процессов и их соединение (*fork* и *join*). Семантика подобных блок-схем может быть естественным образом формализована. Не останавливаясь на этом вопросе подробно, отметим следующие моменты: (1) как и прежде, каждая микрооперация описывается парой $\langle\gamma, \psi\rangle$; (2) обработка всех типов вершин, за исключением *stage* и *join* (когда есть незавершившиеся процессы), осуществляется мгновенно (сразу после обработки микрооперации вычисляется множество следующих микроопераций); (3) для моделирования задержек можно использовать вершины типа *delay*, в которых указана величина задержки (вершина *delay* Δt раскрывается в цепочку из Δt микроопераций с тривиальным контрактом $\langle true, true\rangle$). Описание графа потока управления операции из рассматриваемого примера (см. рис. 4) приведено на рис. 6.

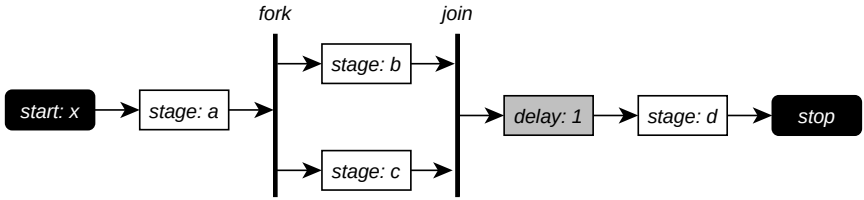


Рис. 6. Описание графа потока управления операции

Остановимся на важном вопросе — проверке выполнимости условий спецификации в процессе верификации. Вообще говоря, формулы, используемые в спецификациях, задают ограничения не только на интерфейсные сигналы, но и на состояние устройства. HDL-симуляторы предоставляют средства для доступа к внутренним переменным HDL-модели,

что позволяет определять условия на состояние и проверять их в процессе верификации. Однако сильная привязка спецификаций (и тестовой системы в целом) к реализации нежелательна с технологической точки зрения — изменения реализации (по крайней мере, незначительные) не должны приводить к изменению спецификации. Как правило, спецификация имеет свой набор переменных, значения которых либо синхронизируются с состоянием реализации (с помощью адаптера — см. раздел 2.1), либо изменяются в самой спецификации (для этого в нее добавляется исполнимая часть).

Чаще используется второй подход — каждая микрооперация описывается тройкой $\langle \gamma, \alpha, \psi \rangle$, где α — это *действие* (элементарная программа) [68]. Отметим также, что точное определение охранных условий γ (по существу, управляющей логики устройства) не всегда требуется — часто неважно, как устроен арбитр, управляющий доступом к некоторому ресурсу, главное, чтобы он был *безопасным* (*safe*) и *справедливым* (*fair*) — обеспечивал взаимное исключение процессов и рано или поздно давал доступ каждому желающему процессу[7]: $\mathbf{G}(|\{\gamma \in \Gamma_R \,|\, \gamma\}| \le 1)$ и $\mathbf{F}\gamma$ для всех $\gamma \in \Gamma_R$, где Γ_R — множество охранных условий микроопераций, желающих получить доступ к ресурсу R. Когда управляющая логика не специфицируется детально, охранные условия определяются через переменные HDL-модели (выходные или внутренние сигналы) — такие спецификации называются *адаптивными*, поскольку, используя обратную связь от реализации, они подстраиваются под ее поведение [69].

Контрактные спецификации конвейера можно использовать не только для проверки корректности поведения HDL-модели, но и для построения тестовой последовательности [65]. В подходе, описанном в [66], тестовая последовательность генерируется *на лету* (*on the fly*) на основе обхода состояний управления (множеств параллельно выполняемых микроопераций). Для этого используются *неизбыточные алгоритмы обхода графов*, позволяющие пошагово обходить сильно связные графы, располагая в каждый момент времени только информацией о текущей вершине (состоянии) и множестве исходящих из нее дуг (допустимых стимулов) [57, 58]. В большинстве проектов по верификации аппаратуры использовалась оптимизированная версия алгоритма α_{ndfsm}, ориентированного на графы, в которых имеется детерминированный (по стимулам) сильно-связный полный остовный подграф [70].

Заметим, что граф, состояниями которого являются множества параллельно выполняемых микроопераций, не обязан быть детерминированным (даже в смысле наличия детерминированного остовного подграфа). Для

---

[7] Темпоральный оператор **F** (in the **F**uture) требует, чтобы формула когда-нибудь выполнилась (начиная с текущего момента времени): $\mathbf{F}\gamma \equiv \mathit{true}\ \mathbf{U}\ \gamma$.

детерминизации обходимого графа был предложен следующий подход: в состояние обработки операции (часть состояния управления) добавляется информация о *ситуации*, возникающей при выполнении операции, и *зависимостях* операции от других обрабатываемых операций [65]. Ситуация задает путь в графе потока управления операции (между вершинами *start* и *stop*) и детерминизирует выбор очередной микрооперации, зависимости определяют конфликты между операциями и детерминизируют установку блокировок. Понятно, что для сложного устройства граф может быть огромным, а его обобщение может снизить качество верификации. Для решения этой проблемы был предложен метод параллельного обхода графа на компьютерном кластере (см. раздел 3.1.3).

Инструментальная поддержка разработки контрактных спецификаций конвейера и тестов на их основе реализована в виде библиотечного расширения PIPE (от англ. pipeline — конвейер) инструмента CTESK [52]. Библиотека разработана на языке SeC (Specification extension of C), используемом в инструменте CTESK, и не привязана к конкретному языку описания аппаратуры. Основные возможности библиотеки PIPE можно разбить на две группы: (1) описание структуры конвейера — композиция операций из множества микроопераций с использованием описанных выше примитивов (*switch*, *merge*, *fork*, *join* и других); (2) эмуляция поведения конвейера — потактовое моделирование конвейера, основанное на отслеживании множества параллельно выполняемых микроопераций, и выполнение проверок, определенных в постусловиях микроопераций. Для разработки тестовых сценариев и генерации тестов использовались стандартные средства инструмента CTESK. В примере 3 приведено описание графа потока управления операции, рассмотренной ранее (см. рис. 6), средствами библиотеки PIPE.

Пример 3. Описание операции с использованием примитивов PIPE

```
01:  specification void x(...) {                    описание операции x:
02:    pre { ... }                                  описание предусловия операции
03:    coverage { ... }                             описание тестового покрытия
04:  }
05:  reaction Process* a(void) {                    описание микрооперации a:
06:    pre { ... }                                  описание охранного условия микрооперации
07:    post { ... }                                 описание постусловия микрооперации
08:  }
09:  Operation *op = create_Operation(x);           описание графа потока управления операции x:
10:  register_Stage (op, 0, a);                     вершина stage (метка 0, спецификация a)
```

11:	register_Fork	(op, 1, 3);	вершина *fork* (метка 1, следующие вершины 2 и 3)
12:	register_Stage	(op, 2, *b*);	вершина *stage* (метка 2, спецификация *b*)
13:	register_Edge	(op, 2, 4);	дуга между вершинами 2 и 4
14:	register_Stage	(op, 3, *c*);	вершина *stage* (метка 3, спецификация *c*)
15:	register_Join	(op, 4);	вершина *join* (метка 4)
16:	register_Delay	(op, 5, 1);	вершина *delay* (метка 5, задержка 1)
17:	register_Stage	(op, 6, *d*);	вершина *stage* (метка 6, спецификация *d*)
18:	register_Stop	(op, 7);	вершина *stop* (метка 7)

Предложенный метод и поддерживающий его инструмент (CTESK вместе с библиотекой PIPE) в 2007-2010 гг. использовались в нескольких проектах по верификации модулей микропроцессоров: буфера преобразования адресов TLB [55], модуля арифметики с плавающей точкой (FPU, Floating Point Unit)[8] [71], кэш-памяти второго уровня (L2) [72] и коммутатора северного моста (Databox) [67]. Во всех модулях были обнаружены ошибки (включая критические), которые были пропущены другими методами верификации (это не удивительно, поскольку детальность спецификаций и систематичность перебора состояний делают подход сравнимым по полноте верификации с формальными методами). Наиболее сложный модуль, проверенный описанным методом, — коммутатор северного моста (описание модуля составляет около 6 тысяч строк на языке Verilog; модуль реализует около 60 различных типов операций[9]; выполнение некоторых операций занимает около 40 тактов). Метод хорошо себя зарекомендовал для сравнительно небольших и детально задокументированных устройств; для сложных проектов со скудной документацией затраты на разработку и отладку потактовых спецификаций трудно оценить [73]. Трудоемкость верификации устройств средней сложности (3-6 тысяч строк кода) составляют 0.7-1.3 человеко-месяца на 1 тысячу строк кода [72].

Сравнивая контрактные спецификации конвейера с другими формализмами, используемыми для описания поведения аппаратуры, следует еще раз упомянуть темпоральную логику LTL и ее варианты. Как было показано ранее, логика LTL не способна адекватно описать некоторые свойства, которые простым образом описываются с помощью контрактных

[8] При верификации модуля арифметики с плавающей точкой основное внимание уделялось не описанию структуры конвейера, а генерации тестовых данных.

[9] Под типами операций здесь понимаются не только коды операций, но и различные способы выполнения (ветви функциональности).

спецификаций. В языках верификации аппаратуры, поддерживающих темпоральные утверждения (см. раздел 2.2), выразительные возможности LTL существенно расширены: можно обращаться к значениям переменных в прошлом и будущем, задавать временны́е интервалы наступления событий, описывать регулярные события, создавать параметризованные шаблоны утверждений, использовать в утверждениях присваивания [34]. Темпоральная логика такого вида позволяет описывать широкий диапазон свойств и активно используется в имитационной верификации, однако, как показывает опыт, инженеры не используют всех ее возможностей, а ограничиваются описанием простых причинно-следственных связей, которые, на наш взгляд, более естественно описываются в терминах контрактов и действий: $\langle \gamma, \alpha, \psi \rangle$ — «если γ, выполнить α и проверить ψ».

Заметим, что добавление в контракты действий приводит нас к классическому формализму — *охраняемым действиям* (*guarded actions*), введенному Дейкстрой в 1975 г. [74], который в последнее время активно используется для высокоуровнего описания аппаратуры с последующим логическим синтезом [75] (в этой связи следует упомянуть язык Bluespec [76]). Охраняемыми действиями называются пары $\langle \gamma, \alpha \rangle$ (в другой форме — $\gamma \rightarrow \alpha$), интерпретируемые следующим образом: когда истинно условие γ, выполняется действие α. Учитывая то, что в контрактных спецификациях конвейера условия и действия структурированы в графы (см. рис. 6), рассматриваемая форма спецификаций имеет много общего с *алгоритмическими машинами состояний* (*ASM, Algorithmic State Machines*) — диаграммами, применяемыми для описания и синтеза аппаратуры [77, 78]. В ASM, однако, нет проверок (постусловий микроопераций) и не поддерживается параллельное выполнение микроопераций (вершины типа *fork* и *join*).

3.1.2 Событийные спецификации аппаратуры

При применении метода верификации на основе потактово точных контрактных спецификаций в промышленной практике были выявлены его ограничения и слабые места [40]: (1) *высокие требования к качеству документации* — как правило, описание модулей не отличается полнотой (акцент делается на их структуре, а не на требованиях к поведению); тонкости работы устройства, необходимые для создания потактовых спецификаций, описываются разработчиками с трудом (часто происходит апелляция к исходному коду); (2) *медленный старт верификации* — сроки проведения верификации ограничены, а результаты (единственным значимым результатом на практике являются найденные ошибки) требуется получить как можно скорее; детальные спецификации позволяют провести тщательную проверку, но результаты могут быть получены слишком поздно; (3) *сложность интеграции в процессы проектирования* — разработка модулей микропроцессоров часто сопровождается созданием их обобщенных моделей

на языках высокого уровня, таких как C++ и SystemC; вместо разработки спецификаций с нуля целесообразно использовать имеющиеся наработки.

В результате анализа опыта промышленной верификации используемый подход был расширен (2011 г.). В теоретическом плане, расширение подхода связано с переходом от потактовой парадигмы к *событийной*, что позволяет использовать спецификации с неточной моделью времени (с упрощенной или отсутствующей управляющей логикой). Используемая техника проверки корректности поведения HDL-моделей основана на динамическом сопоставлении трасс (частично упорядоченных множеств событий) [79]: одна трасса порождается реализацией, вторая — спецификацией (эталонной моделью). Практическая сторона работ связана с переходом от языка SeC (используемом в CTESK расширения ANSI C) к языку программирования C++, что дает возможность использовать в составе тестовых систем модели устройств (компоненты симулятора), создаваемые в процессе архитектурного проектирования (см. раздел 2).

Наблюдаемое поведение HDL-модели описывается *временны́м словом* — последовательностью вида $\{(a_i, t_i)\}_{i \geq 0}$, где a_i — событие, а t_i — время его наступления. События, происходящие в одно и то же время, допустимы, но должны возникать на разных интерфейсах устройства. Поведение эталонной модели задается *временно́й трассой* (или просто *трассой*) — частично упорядоченным множеством пар (a, t). Частичный порядок описывает причинно-следственные связи между событиями и другие ограничения, связанные с упорядоченностью. Заметим, что временны́е слова являются частной разновидностью трасс — трассами с тривиальным порядком (по сути, это означает отсутствие информации о взаимосвязях между событиями). При использовании неточных эталонных моделей трассы обобщаются — вместо временны́х меток указываются *временны́е интервалы*, задающие ограничения на времена возникновения событий в реализации [80]. Пример показан на рис. 7: наблюдаемое поведение верифицируемой HDL-модели описывается трассой $\{(b, 1), (a, 2), (c, 3), (d, 5)\}$, поведение эталонной модели — $\{(a, 1), (b, 2), (c, 2), (d, 3)\}$; при обобщении эталонной трассы временны́е метки расширяются до интервалов, но, вместе с тем, в трассу добавляется информация о зависимостях между событиями.

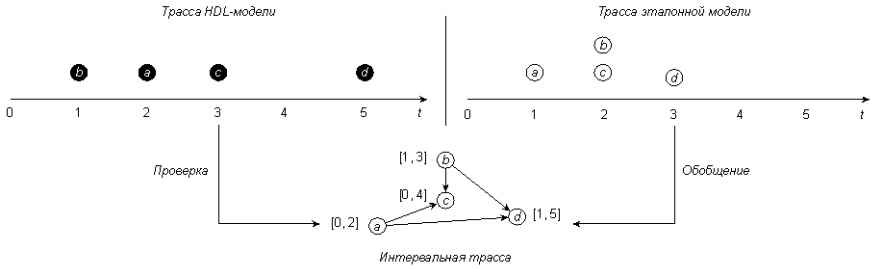


Рис. 7. Схема проверки соответствия между HDL-моделью и эталонной моделью

В общих словах, суть выполняемой проверки поведения следующая: трасса реализации (HDL-модели) должна быть линеаризацией трассы спецификации (эталонной модели), а временны́е метки событий реализации не должны выходить за рамки интервалов, заданных в соответствующих событиях спецификации (формальное определение можно найти в работе [79]). Важно отметить, что проверка соответствия трасс выполняется в динамике, а не после завершения верификации. Рис. 8 иллюстрирует проверку корректности поведения на примере трасс, показанных на рис. 7 (рассматривается момент времени $t = 4$). В верхней части изображены реакции HDL-модели, в нижней — реакции эталонной модели; отношение порядка между реакциями эталонной модели показано стрелками; пунктирные линии соединяют сопоставленные реакции.

Тестовый оракул, осуществляющий сопоставление трасс, имеет два типа входов: (1) реакции HDL-модели и (2) реакции эталонной модели. На каждом такте тестовый оракул выполняет следующие действия: (1) *прием реакций* — полученные реакции спецификации и реализации, X' и Y', заносятся в соответствующие частично упорядоченные множества событий, X и Y: $X \leftarrow X \cup X'$, $Y \leftarrow Y \cup Y'$; из обоих множеств удаляется пересечение множеств минимальных (по отношению порядка) элементов[10]:

$$X \leftarrow X \setminus (min(X) \cap min(Y)),\ Y \leftarrow Y \setminus (min(X) \cap min(Y))$$

(2) *проверка превышения лимита времени* — если время нахождения реакции в соответствующем множестве превышает заданную величину, фиксируется ошибка: *пропущенная реакция* (для реакции спецификации) или *неожиданная реакция* (для реакции реализации).

[10] Как отмечалось выше, поведение реализации (как правило) моделируется трассой с тривиальным порядком (порядком, задаваемым отношением равенства), для которого все события являются минимальными.

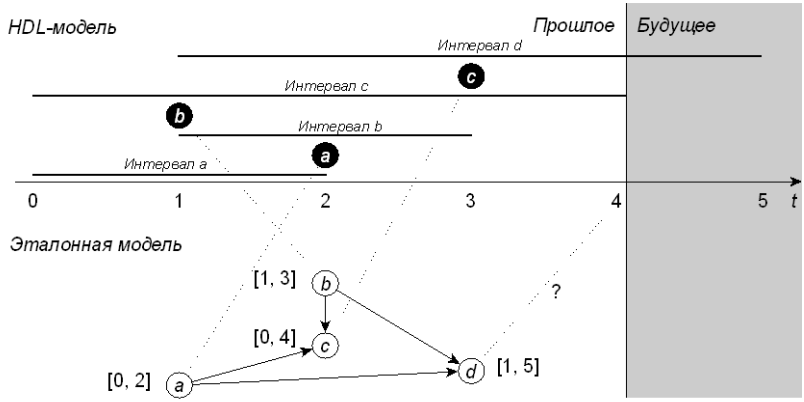


Рис. 8. Соответствие между HDL-моделью и эталонной моделью

Упрощенное описание алгоритма работы оракула на каждом такте верификации приведено в примере 4 (в начале работы $X = Y = \varnothing$ и $v = true$; в случае ошибки в v заносится *false*). Более детальное и формализованное описание можно найти в работе [79].

Пример 4. Примерный алгоритм работы тестового оракула

01:	**receive** *X', Y'*	прием множеств реакций на текущем такте
02:	**for** $(x, t) \in X'$ **do**	сопоставление реакции эталонной модели
03:	**if** $(x, t) \notin Y$ **then** $X \leftarrow (x, t)$	(несопоставленные реакции откладываются)
04:	**for** $(y, t) \in Y'$ **do**	сопоставление реакции HDL-модели
05:	**if** $(y, t) \notin X$ **then** $Y \leftarrow (y, t)$	(несопоставленные реакции откладываются)
06:	**for** $(x, t) \in X$ **do**	если для реакции эталонной модели превышен лимит времени,
07:	**if** timeout(x, t) **then** $v \leftarrow$ *false*	фиксируется ошибка: *пропущенная реакция*
08:	**for** $(y, t) \in Y$ **do**	если для реакции HDL-модели превышен лимит времени,
09:	**if** timeout(y, t) **then** $v \leftarrow$ *false*	фиксируется ошибка: *неожиданная реакция*

Рассмотренный подход к проверке корректности поведения HDL-моделей был реализован в инструменте C++TESK [81]. Отношение порядка между реакциями эталонной модели может быть задано двумя способами: (1) непосредственно в эталонной модели и (2) с помощью *арбитров реакций* [40]. Арбитр реакций задается для каждого выходного интерфейса и по запросу тестовой системы выдает очередную реакцию эталонной модели. Арбитры делятся на три типа: (1) *модельные арбитры* (которые при выборе реакции ориентируются только на порядок их выдачи эталонной моделью), (2) *адаптивные арбитры* (которые осуществляют выбор реакции эталонной модели путем сопоставления с заданной реакцией реализации) и (3) *двухуровневые арбитры* (которые работают в два этапа: сначала строят множество возможных реакций эталонной модели, используя модельный арбитр (множество минимальных по отношению порядка реакций), а затем применяют к полученному множеству адаптивный арбитр).

В основе C++TESK лежит библиотека классов, обеспечивающая разработку тестовых систем для HDL-моделей аппаратуры на уровне TLM (см. раздел 2.2). Средства инструмента позволяют создавать такие компоненты, как эталонные модели, адаптеры, тестовые сценарии, сборщики тестового покрытия и другие (см. рис. 2). Использование для создания тестовых систем языка программирования C++ позволяет без труда встраивать в них сторонние эталонные модели, разработанные на этом языке. В примере 5 показано описание операции, изображенной на рис. 6, средствами C++TESK. Обратим внимание, что процессы могут быть запущены как параллельным (*PARALLEL*), так и последовательным образом (*SEQUENTIAL*). При запуске процесса указывается интерфейс ($iface_a$, $iface_b$, $iface_c$, $iface_d$) и входное сообщение для обработки (msg_a, msg_b, msg_c, msg_d).

Пример 5. Описание операции с использованием примитивов C++TESK

01:	**DEFINE_STIMULUS** (*x*) {	описание операции *x*:
02:	**START_STIMULUS**();	начало операции
03:	**CALL_PROCESS**(**SEQUENTIAL**, *a*, $iface_a$, msg_a);	запуск микрооперации *a*
04:	**CALL_PROCESS**(**SEQUENTIAL**, *b*, $iface_b$, msg_b);	запуск микрооперации *b*
05:	**CALL_PROCESS**(**PARALLEL**, *c*, $iface_c$, msg_c);	запуск микрооперации *c* (параллельно *b*)
06:	**CYCLE**();	задержка в один такт
07:	**CALL_PROCESS**(**SEQUENTIAL**, *d*, $iface_d$, msg_d);	запуск микрооперации *d*
08:	**STOP_STIMULUS**();	завершение операции
09:	}	описание операции *x*:

Инструмент C++TESK в 2011-2013 гг. использовался в ряде проектов по верификации модулей микропроцессоров [40, 79, 73]: межпроцессорного коммутатора данных (MAU Hub, Memory Access Unit's Hub), системного контроллера прерываний (SAPIC, System Advanced Programmable Interrupt Controller), модуля поиска по таблице страниц (TLU, Table Lookup Unit), коммутатора северного моста (Databox), буфера команд (IB, Instruction Buffer), подсистем кэширования 2-ого и 3-ого уровней (L2 и L3). Предложенный метод позволил создать эталонные модели для модулей средней сложности без наличия детальной документации и найти большое количество сложных ошибок. Трудоемкость разработки тестовой системы в среднем оценивается в 1 человеко-месяц на 1 тысячу строк кода HDL-описания (причем работа по созданию эталонной модели трудно распараллеливается). Метод применим для верификации модулей средней сложности (до 10 тысяч строк кода) и модулей большой сложности (10-50 тысяч строк кода), но допускающих абстракцию управляющей логики. Для более сложных подсистем целесообразнее использовать верификацию с помощью тестовых программ (см. раздел 3.2).

Рассмотрим подходы, близкие к предлагаемому методу. В работе [82] используется модель *автомата с частично упорядоченными входными/выходными событиями* (*POIOA, Partial Order Input/Output Automaton*) для представления поведения эталонной модели и реализации. Если (1) реализация сообщает о приеме неподдерживаемых стимулов, (2) можно установить порядок выдачи реакций реализацией, (3) время ответа реализации ограничено, и (4) каждый единичный переход в эталонной модели соответствует единичному переходу в реализации, можно установить соответствие между реализацией и эталонной моделью: они соответствуют друг другу, если реализация принимает допустимые эталонной моделью стимулы и выдает описываемые ею реакции в допустимом порядке. Данный подход к определению соответствия схож с предложенным нами, но отличается отсутствием контроля временны́х интервалов. В работе [63] рассматривается подход к тестированию параллельных систем с помощью неявно заданных *асинхронных конечных автоматов* (*AFSM, Asynchronous Finite State Machines*). Поведение реализации проверяется только в стационарных состояниях, в которых не ожидается выдача реакций: (1) собираются все события, и определяется их частичный порядок; (2) строятся и проверяются все возможные линеаризации событий. Считается, что реализация соответствует эталонной модели, если допускается хотя бы одна линеаризация. Данный подход, в отличие от предложенного метода, не может использоваться с произвольными генераторами стимулов, поскольку в нем требуется, чтобы в процессе тестирования регулярно возникали стационарные состояния, но, с другой стороны, он применим в том случае, когда не известен точный порядок выдачи реакций реализацией.

3.1.3 Распараллеливание процесса верификации

Способ генерации тестовых последовательностей, используемый в технологии UniTESK [28], основан на обходе графа состояний тестируемой системы. Отличительной чертой технологии является то, что граф состояний задается неявно (с помощью функции построения текущего состояния и множества допустимых стимулов) и строится по мере его обхода [57, 58]. При проходе по дуге графа осуществляется подача стимула на тестируемую систему. Обход графа завершается после прохождения всех дуг (когда обнаружено, что ни из одной известной вершины не выходит непройденная дуга) или обнаружения ошибки (при невыполнении одного из постусловий). Для тестирования HDL-моделей аппаратуры пространство состояний задавалось множествами параллельно выполняемых микроопераций (см. раздел 3.1.1). Такой способ позволяет добиться высоких показателей тестового покрытия (см. раздел 2.2), однако получаемые графы могут быть очень большими. Так, для системного контроллера прерываний SAPIC (простого устройства, HDL-описание которого содержит около 2 тысяч строк кода) граф состояний (для одного из тестовых сценариев[11]) содержал около 100 тысяч вершин и 500 тысяч дуг, а для модели коммутатора северного моста Databox (около 6 тысяч строк кода) — около 500 тысяч состояний и 2 миллионов дуг. Размер графов, используемых для генерации тестовых последовательностей, может быть уменьшен с помощью техник сокращения [83], однако это может снизить качество верификации.

Чтобы ускорить процесс тестирования (обход больших графов состояний) были использованы возможности, предоставляемые распределенными вычислительными системами (компьютерными кластерами). Был разработан метод, позволяющий динамически распараллеливать обход графа, не располагая при этом никакой информацией о его структуре до начала тестирования [84]. Реализация метода потребовала расширения классической архитектуры тестовой системы. Это было сделано следующим образом. На каждом компьютере кластера выполняется свой процесс тестовой системы. Процесс осуществляет обход графа, а полученной информацией о пройденной части графа обменивается с процессами на других компьютерах кластера (заметим, что разные процессы могут использовать разные алгоритмы обхода). Архитектура процесса распределенной тестовой системы представлена на рис. 9 (для наглядности такие компоненты, как тестовый оракул и адаптер, опущены).

Процесс включает в себя следующие компоненты: (1) *реализация алгоритма обхода* (*обходчик*), (2) *представление графа состояний* (*хранилище*),

[11] Каждый тестовый сценарий предназначен для проверки определенного аспекта тестируемой системы; получаемый в результате граф состояний обычно существенно меньше совокупного графа, моделирующего систему.

(3) *синхронизатор* и (3) *генератор стимулов*. Обходчик — это библиотечный компонент, который по запросу строит маршрут из текущей вершины графа в вершину, из которой выходят еще не пройденные дуги, и выбирает одну из них. Хранилище отвечает за представление пройденной части графа (посещенных вершин, допустимых в них стимулов, пройденных дуг). Синхронизатор получает от обходчика информацию о новых пройденных дугах и рассылает ее синхронизаторам других процессов; он также получает информацию о дугах, пройденных другими процессами, и передает ее своему обходчику (который добавляет информацию в хранилище). Генератор стимулов является активной частью тестовой системы, использующей обходчик для построения путей в графе и тестовый сценарий для вычисления текущего состояния и определения допустимых в нем стимулов. Более подробно архитектура тестовой системы и протокол синхронизации процессов описаны в работе [84].

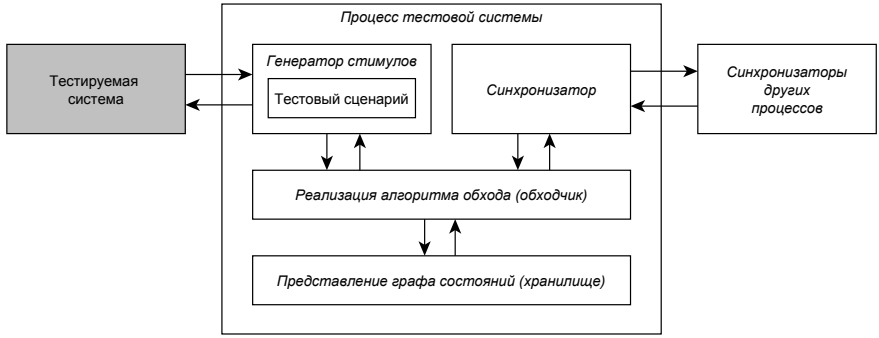


Рис. 9. Архитектура процесса распределенной тестовой системы

Предложенный метод реализован в рамках проекта CTESK [52] (в настоящее время разработанные средства также доступны в инструменте C++TESK [81]). Для управления процессом тестирования разработан *координатор*, позволяющий через Web-интерфейс задавать такие параметры, как используемые для тестирования компьютеры, топологию связей между ними (граф обмена информацией) и другие. Координатор запускает процессы тестовой системы и собирает информацию о состоянии тестирования (размере пройденного графа, числе найденных ошибках и достигнутом уровне тестового покрытия). Так как обход графа может занимать продолжительное время, система поддерживает возможность сохранения и последующего восстановления пройденной части графа. Это позволяет при необходимости прервать тестирование или же продолжить его после сбоя без потери полученных результатов.

В 2010-2012 гг. подход применялся при верификации нескольких модулей микропроцессоров: коммутатора северного моста Databox, системного

контроллера прерываний SAPIC и межпроцессорного коммутатора данных MAU Hub. Размеры графов доходили до нескольких миллионов вершин. Использование описанного подхода позволило существенно снизить время выполнения тестов, коэффициент эффективности распараллеливания превосходил 0.8 (то есть при использовании, например, 100 компьютеров время выполнения тестов уменьшалось более чем в 80 раз) [85]. Существующая реализация имеет ограничение — каждый процесс тестовой системы хранит совокупный граф состояний, между тем, графы могут быть настолько велики, что не помещаются в оперативной памяти компьютера (а при использовании внешней памяти сильно возрастает время построения путей). Одно из возможных решений этой проблемы предлагается в статье [86] — способ параллельного обхода, позволяющий избежать дублирования информации о пройденной части графа в процессах, распределяя ее между ними.

3.2. Разработки в области системной верификации

Если при создании методологии модульной верификации микропроцессоров мы отталкивались от опыта разработки и применения технологии тестирования UniTESK [28], то, столкнувшись в 2006 г. с новой для нас задачей системной верификации (с задачей генерации тестовых программ на языке ассемблера для одноядерного MIPS-совместимого микропроцессора), выбор метода решения был неочевиден. Наиболее близкими подходами, разработанными в ИСП РАН, представлялись методы автоматического построения тестов (программ на языках высокого уровня) для проверки синтаксических анализаторов [87] и оптимизирующих компиляторов [88]. Эти методы основаны на синтаксических моделях: в первом из них используется грамматика языка, во втором — редуцированная грамматика, построенная с учетом алгоритма оптимизации.

В отличие от языков высокого уровня язык ассемблера имеет бедный, нерекурсивный синтаксис, что снижает значимость синтаксически-ориентированных методов (хотя в некоторой форме они используются для генерации ассемблерных программ [89] и могут быть полезны для построения графов потока управления программ [90] и композиции сложных программ из простых [91]). Следует также отметить, что методы, описанные в [87, 88] не учитывают семантику грамматических конструкций — между тем, это важно для генерации *тестовых программ со встроенными проверками* (см. раздел 2.3). На основе анализа большого числа методов и инструментов генерации тестовых программ для микропроцессоров (см., например, [42]) в качестве спецификаций были выбраны *модели уровня системы команд*.

3.2.1 Комбинаторная генерация тестовых программ

Модели уровня системы команд описывают микропроцессор в форме переменных (регистров и памяти) и команд — атомарных действий над

переменными (для которых также указываются предусловия). В отличие от детальных моделей, используемых для модульной верификации микропроцессоров, такие модели абстрагируются от структуры конвейера и совместного выполнения команд на нем. Основное назначение моделей уровня системы команд — формализация семантики команд и формирование программистского взгляда на микропроцессор (программист, составляя программу, считает команды атомарными — выполнение последовательности команд равносильно их последовательному выполнению одна за другой). Описание семантики команд на псевдокоде можно встретить в большинстве руководств по архитектуре микропроцессоров. В примере 6 приведено описание команды *ADD* (целочисленное сложение) из руководства по системе команд MIPS [92]. Команда имеет формат *ADD rd, rs, rt*, где *rs*, *rt* и *rd* — это номера регистров общего назначения (GPR, General-Purpose Registers).

Пример 6. Описание команды ADD архитектуры MIPS [92]

01:	**if** NotWordValue(GPR[rs]) **or** NotWordValue(GPR[rt]) **then**	проверка предусловия команды:
02:	**UNPREDICTABLE**	если предусловие не выполнено,
03:	**endif**	результат команды непредсказуем
04:	temp ← ($GPR[rs]_{31}$‖$GPR[rs]_{31..0}$) + ($GPR[rt]_{31}$‖$GPR[rt]_{31..0}$)	описание производимых действий:
05:	**if** $temp_{32} \neq temp_{31}$ **then**	если возникает переполнение,
06:	SignalException(IntegerOverflow)	Генерируется исключение,
07:	**else**	в противном случае
08:	GPR[rd] ← sign_extend($temp_{31..0}$)	записывается в регистр *r*
09:	**endif**	проверка предусловия команды:

На основе анализа спецификации команды можно выделить набор *тестовых ситуаций* — ограничений на значения операндов команды и состояние микропроцессора, характеризующих различные способы выполнения этой команды. Как правило, тестовая ситуация соответствует одному из возможных путей в графе потока управления спецификации. В приведенном выше примере можно идентифицировать две тестовые ситуации: *целочисленное переполнение* ($temp_{32} \neq temp_{31}$) и *нормальное выполнение* ($temp_{32} = temp_{31}$), где $temp = (GPR[rs]_{31} \,||\, GPR[rs]_{31..0}) + (GPR[rt]_{31} \,||\, GPR[rt]_{31..0})$ (символ || обозначает конкатенацию битовых векторов, а нижние индексы используются для выделения подмассива). Покрытие всех тестовых ситуаций реализуемых микропроцессором команд является необходимым, но не достаточным условием качественной системной верификации.

Другим типом информации, полезным для верификации микропроцессора, является информация о *зависимостях* между командами, характеризующая совместное выполнение команд на конвейере и, прежде всего, конфликты использования ресурсов (см. рис. 5). Обычно подобные сведения (за исключением очевидных зависимостей по регистрам) не отражаются в моделях уровня системы команд (и руководствах по системам команд) — их источником выступают спецификации конкретных микропроцессоров. В качестве примера рассмотрим следующую зависимость между командами обращения в память — попадание в разные строки кэш-памяти L1, расположенные по одному индексу (в одном множестве). Зависимость определяется организацией кэширования. В микропроцессорах семейства RM7000 кэширование осуществляется по физическим 36-битным адресам: биты 2..0 адреса определяют позицию байта внутри двойного слова, биты 4..3 — позицию двойного слова в строке кэш-памяти, биты 11..5 — индекс строки и, наконец, биты 35..12 — тэг, используемый для проверки попадания данных в кэш-память [93] (см. пример 7).

Пример 7. Формат физического адреса в микропроцессоре RM7000 [93]

Тэг данных	***Индекс строки L1***	***Номер двойного слова***	***Номер байта***
35 12	11 5	4 3	2 0

Таким образом, две команды связаны зависимостью данного типа, если для физических адресов, по которым происходит реальное обращение в память (обозначим их x и y), выполнено ограничение $(x_{11..5} = y_{11..5}) \wedge (x_{35..12} \neq y_{35..12})$. Виртуальные 40-битные адреса (обозначим их a и b) должны быть подобраны соответствующим образом — если размер страницы виртуальной памяти составляет 4 килобайта (в этом случае биты 11..0 виртуального адреса задают смещение, а биты 39..12 — номер страницы), должно выполняться условие $(a_{11..5} = b_{11..5}) \wedge (a_{39..12} \neq b_{39..12})$, причем $map(a_{39..12}) \neq map(b_{39..12})$, где map — отображение страниц виртуальной памяти в страницы физической памяти, осуществляемое с помощью буфера трансляции адресов. Более детально различные типы зависимостей по физическим и виртуальным адресам рассмотрены в статье [54].

Идея предложенного подхода к построению тестовых программ, названного *комбинаторной генерацией на основе моделей* [94], основана на двух предположениях: (1) поведение микропроцессора определяется множеством параллельно выполняемых команд (загрузкой конвейера), зависимостями между командами (характером обращений к ресурсам микропроцессора), а также ситуациями, возникающими при выполнении команд (путями выполнения команд в конвейере); (2) ошибки (в RISC-процессорах) могут быть обнаружены с помощью небольших тестовых примеров (состоящих из 2-5 команд) [42, 94]. В простейшем случае генерация тестовых программ

осуществляется путем систематического перебора коротких последовательностей команд (включая перебор тестовых ситуаций для отдельных команд и зависимостей между парами команд). Для сокращения перебора могут использоваться эвристики (объединение команд в классы эквивалентности, ограничение числа зависимостей и другие).

Подход также позволяет комбинировать простые тестовые шаблоны (последовательности команд с заданными тестовыми ситуациями и зависимостями без указания конкретных значений операндов) в сложные [91]. Целью комбинирования является создание нетривиальных ситуаций в работе микропроцессора (одновременных исключений, параллельных конфликтов и других). Составление сложных тестовых шаблонов из простых может осуществляться с помощью следующих операций: *наложение* ($T_1 \parallel T_2$ — объединение множеств тестовых ситуаций и зависимостей у однотипных шаблонов), *сдвиг* ($T_1 >> T_2$ — соединение шаблонов без отождествления команд со сдвигом одного относительно другого), *конкатенация* ($T_1 \cdot T_2$ — последовательное расположение шаблонов друг за другом) и *вложение* ($T_1[T_2]$ — помещение одного шаблона внутрь другого). Генерация тестовых программ для фиксированного набора базовых шаблонов осуществляется путем перебора синтаксических деревьев ограниченного размера, описывающих составные шаблоны. В примере 8 приведен результат комбинирования простых тестовых шаблонов с помощью указанных операций ($T_1[T_2 >> T_3] \cdot T_4$) [95].

Пример 8. Результат комбинирования тестовых шаблонов

01:	LD ?, ?(?) @ hit<TLB>(PFN1)	шаблон T_1 (зависимость по адресам)
02:	ADD r, ?, ?	шаблон T_2 (зависимость по регистрам)
03:	DIV.S ?, ?, ?	шаблон T_3 (конфликт между командами)
04:	SUB ?, r, ?	шаблон T_2
05:	DIV.D ?, ?, ?	шаблон T_3
06:	SD ?, ?(?) @ hit<TLB>(PFN2) && PFN1 = PFN2	шаблон T_1
07:	FPU ?, ?, ? @ Overflow	шаблон T_4 (исключение

Отдельного упоминания заслуживает подход к генерации тестовых программ, содержащих команды ветвления. Как минимум, такие программы не должны зацикливаться. Построение тестовых программ осуществляется перебором разнообразных графов потока управления (*структур переходов*) и маршрутов в них (*трасс выполнения*) [90]. Перебор трасс выполнения для заданной структуры переходов основан на поиске в глубину в графе потока управления. Чтобы программа выполнялась согласно заданной построенной трассе, в нее

встраиваются вспомогательные команды, называемые *управляющими*. Представленный ниже пример 9 содержит команду перехода *BEQ* (переход по равенству), которая выполняется два раза (когда она выполняется первый раз, условие перехода истинно; затем условие делается ложным, и выполнение теста завершается). Управляющая команда (*ADDI*) инкрементирует один из регистров, используемых в команде перехода (вначале значения регистров равны).

Пример 9. Тестовый шаблон, содержащий команды ветвления

01:	INIT:	инициализация регистров
02:	ORI r1, r0, 2014	r1 = 2014
03:	ORI r2, r0, 2014	r2 = 2014
04:	...	
05:	J START	переход на начальную метку
06:	...	
07:	BACK:	промежуточная метка
08:	ADDI r1, r1, 1	управляющая команда
09:	FPU ?, ?, ?	команда тестового шаблона
10:	START:	начальная метка
11:	BEQ r1, r2, BACK @ trace = {true, false}	условный переход по равенству
12:	LD ?, ?(?)	команда тестового шаблона
13:	STOP:	инициализация регистров

Предложенный метод был реализован в генераторе тестовых программ для микропроцессоров MicroTESK[12] [47] (первоначально инструмент был назван TestFusion 4M [94]). Генератор реализован на языке программирования Java и состоит из трех основных компонентов: (1) *библиотека моделирования* — средства высокоуровневого описания микропроцессоров и реализуемых ими команд; (2) *библиотека тестирования* — средства разработки генераторов тестовых данных и тестовых последовательностей; (3) *графическая оболочка* — пользовательский интерфейс, позволяющий задавать параметры генерации тестовых программ. Следует отметить, что в MicroTESK тестовые ситуации и

[12] В этом разделе речь идет о генераторе MicroTESK версии 1.0 (2006-2008 гг.); в текущей версии инструмента (версии 2.0) реализованы дополнительные возможности, которые рассматриваются в разделе 3.2.2.

зависимости описываются конструктивно — для каждого типа тестовой ситуации (и зависимости) разрабатывается генератор, конструирующий случайные тестовые данные, удовлетворяющие заданной ситуации. В примере 10 приведен фрагмент описания команды *ADD* архитектуры MIPS (см. пример 6).

Пример 10. Описание команды ADD с использованием примитивов MicroTESK

```
01:  public class ADD extends Instruction {                  класс, описывающий команду ADD
02:    public ADD() {                                        конструктор класса команды:
03:      rd = add(this, OUT, GPR, WORD);                     добавление операндов
04:      …
05:    }
06:    public void execute(Processor processor) {            метод, эмулирующий выполнение команды:
07:      Data result = add(rs.getWord(), rt.getWord());      код соответствует спецификации команды (см. пример 6)
08:      if(isOverflow32(result))
09:        { setException(new IntegerOverflow()); }
10:      else
11:        { rd.setWord(result); }
12:    }
13:    public String toString() {                            метод вывода команды в формате
14:      return format("ADD %s, %s, %s", rd, rs, rt);        языка ассемблера
15:  }}
```

Предложенный метод и генератор тестовых программ MicroTESK в 2006-2010 гг. использовались в нескольких проектах по верификации микропроцессоров и их компонентов: подсистема управления памятью (MMU, Memory Management Unit) [42, 54], MIPS-совместимый микропроцессор [42], два арифметических сопроцессора (CP1, CP2) [91]. Во всех указанных проектах были обнаружены ошибки, которые были пропущены другими

методами верификации. Наиболее сложное устройство, проверенное описанным методом, — MIPS-совместимый микропроцессор (одноядерный микропроцессор, реализующий более 200 команд; имеет в своем составе буфер трансляции адресов и двухуровневую кэш-память). При использовании метода не было выявлено ограничений на сложность верифицируемых систем (в классе одноядерных микропроцессоров). Трудоемкость разработки генератора тестовых программ определяется числом команд микропроцессора и составляет приблизительно 0.8 человеко-дня на одну команду (судя по всему, этот показатель может быть снижен за счет использования специализированных ADL-языков). Другие подходы к генерации тестовых программ рассмотрены в разделе 2.3.

3.2.2 Генерация тестовых программ на основе шаблонов

Одна из проблем, которая возникает при разработке нового микропроцессора, состоит в том, что имеющиеся генераторы тестовых программ сложно адаптировать к его архитектуре. Для этого приходится вручную модифицировать логику генерации, что приводит к дополнительным затратам ресурсов и чревато внесением ошибок. Генераторы, основанные на моделях, такие как Genesys-Pro [41], решают эту проблему, выделяя функции генерации в архитектурно-независимые компоненты. Адаптация генератора осуществляется путем изменения модели, инкапсулирующей архитектурно-зависимую информацию. Для того чтобы облегчить создание модели нами был предложен подход [96], состоящий в использовании ADL-языков [19].

По характеру описываемых свойств ADL-языки принято разделять на две группы: *структурные* и *поведенческие*. Языки первой группы ориентированы на спецификацию микроархитектуры (примером такого языка является MIMOLA). Вторая группа концентрируется на описании команд микропроцессора (к этой категории можно отнести ISDL и nML). Кроме того, существуют *смешанные языки*, сочетающие возможности языков обеих групп (например, LISA и EXPRESSION). Для спецификации макроархитектуры наиболее подходят поведенческие ADL-языки, позволяющие явно описывать синтаксис и семантику команд. На основе этой информации может быть построена обобщенная модель микропроцессора, которую можно использовать для настройки генератора под конкретную архитектуру. Из нескольких поведенческих и смешанных ADL-языков, таких как LISA, EXPRESSION, ISDL и nML [19], был выбран последний. Данный язык обладает интуитивно понятным синтаксисом, имеет доступную документацию и используется в нескольких зарубежных университетах и коммерческих компаниях.

Язык nML [97] был разработан в начале 1990-х гг. в Берлинском техническом университете (Technische Universität Berlin) и изначально предназначался для создания симуляторов микропроцессоров и настраиваемых компиляторов. С конца 1990-х гг. nML активно используется Target Compiler Technologies [98]

для решения таких задач, как генерация симуляторов, компиляторов и HDL-моделей. Эта компания расширила язык nML средствами моделирования конвейера [98, 19]. Другая версия языка nML была разработана в Индийском технологическом институте Каппур (Indian Institute of Technology Kanpur) и получила название Sim-nML [99]. В язык были добавлены средства описания ресурсов микропроцессора. В Исследовательском институте информатики в Тулузе (Institut de Recherche en Informatique de Toulouse) для создания симуляторов микропроцессоров применяется язык nMP, основанный на Sim-nML [100, 101]. Среда генерации тестовых программ MicroTESK (версии 2.0) [47], разрабатываемая в ИСП РАН с 2010 г., использует nML для спецификации архитектуры верифицируемого микропроцессора.

Пример 11. Описание команды ADD на языке nML

```
01:  op ADD (rd : index, rs :  REG, rt : REG)                    команда ADD:
02:     syntax = format("ADD %d, %s, %s"                         ассемблерный
     rd, rs.syntax, rt.syntax)                                   формат
03:     image = format("000000%s%s%5b00000100000",               бинарный
     rs.image, rt.image, rd)                                     формат
04:     action = {                                               описание
                                                                 семантики
05:        temp = rs<31..31>::rs<31..0> + rt<31..31>::rt<31..0>;
06:        if (temp<32..32> != temp<31..31>) then
07:           SignalException("Integer Overflow Exception");
08:        else
09:           GPR[rd] = temp<31..0>;
10:        endif;
11:     }
12:  mode REG (r : index) = if (r == 0) then 0 else              режим
     GPR[r] endif                                                адресации
                                                                 REG:
13:     syntax = format("%d", r)                                 ассемблерный
                                                                 формат
14:     image = format("%5b", r                                  бинарный
                                                                 формат
```

Спецификация на языке nML включает описание следующих сущностей: (1) *элементов хранения данных* (памяти, регистров, внутренних переменных), определяющих состояние микропроцессора; (2) *команд*, выполнение которых изменяет это состояние; (3) *режимов адресации*, реализующих абстракцию

доступа к элементам хранения данных. Пример 11 содержит спецификацию команды *ADD* архитектуры MIPS (см. пример 6) и спецификацию режима адресации *REG*, который она использует для чтения данных из регистров общего назначения.

Архитектура среды генерации тестовых программ MicroTESK показана на рис. 10. Среда включает в себя *транслятор*, который преобразует спецификацию на языке nML в исполнимую *модель микропроцессора* и *модель покрытия*. Модель микропроцессора строится на основе *библиотеки моделирования* и реализует следующие функции: (1) предоставление *мета-информации,* описывающей элементы микропроцессора (регистры, память, команды и т.д.); (2) *симуляция* команд микропроцессора; (3) *мониторинг* состояния элементов памяти модели. Модель покрытия содержит информацию о ситуациях, которые могут возникнуть в процессе выполнения команд [102].

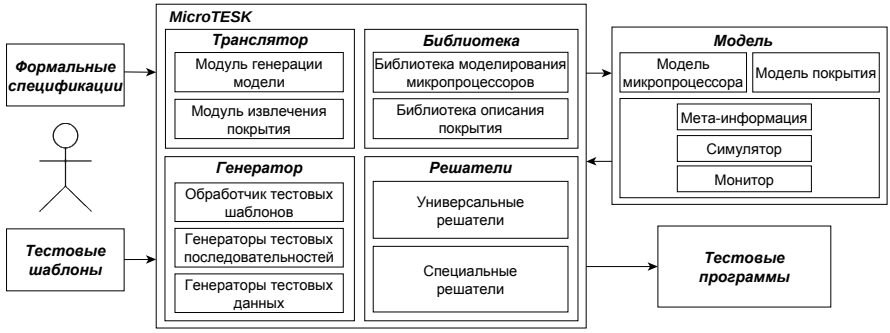


Рис. 10. Архитектура среды генерации тестовых программ MicroTESK

MicroTESK генерирует тесты на основе *тестовых шаблонов*, представляющих собой абстрактные описания тестовых программ. Для их разработки используется *язык описания тестовых шаблонов*, реализованный в виде набора библиотек на языке Ruby [103], что отличает его от узкоспециализированных языков, используемых в инструментах Genesys-Pro [41] и RAVEN [45]. Средства языка описания тестовых шаблонов можно разделить на две группы: (1) встроенные конструкции Ruby (условные операторы, операторы циклов и т.д.) и (2) средства библиотек MicroTESK (базовые классы тестовых шаблонов, конструкции описания блоков построения тестовых последовательностей и другие). Пример 12 демонстрирует формат тестового шаблона.

Пример 12. Структура тестового шаблона на языке Ruby

```
01:  class Template2014 < Template      класс тестового примера
02:    def pre ... end                  инициализирующий код
03:    def run ... end                  основной код
04:    def post ... end                 завершающий код
05:  end
```

Основная секция тестового шаблона представляет собой иерархическую систему блоков, каждый из которых описывает способ генерации тестовой последовательности. Объединяющие блоки заданным образом комбинируют тестовые последовательности, описанные во вложенных блоках (см. раздел 3.2.1). Такой подход позволяет применять различные методы построения тестов в одном тестовом шаблоне (см. пример 13).

Пример 13. Фрагмент тестового шаблона с вложенными блоками

```
01:  block (:combine => "product",          объединяющий блок
                                            случайно смешивает все
                                            возможные комбинации
                                            результатов вложенных
                                            блоков
02:          :compose => "random") {
03:    block (:engine => "random",          вложенный блок
                                            генерирует 2 случайные
                                            последовательности
                                            длины 3
04:            :length => 3, :count => 2) {
05:      ADD reg(8), reg(16), reg(17)
06:      SUB reg(9), reg(18), reg(19)
07:      AND reg(10), reg(16), reg(17)
08:      OR reg(11), reg(18), reg(19)
09:    }
10:    block (:engine => "permutate") {     вложенный блок гене-
                                            рирует последова-
                                            тельности из всех воз-
                                            можных перестановок
11:      LW reg(12), reg(20), imm16(0)      команд
12:      SW reg(19), reg(21), imm16(4)
13:    }
14:  }
```

За генерацию тестовых программ по шаблонам в среде MicroTESK отвечает *обработчик тестовых шаблонов*, который формирует тестовые

последовательности, вызывая *генераторы*, указанные в блоках. Результатом являются *символические тестовые программы*, в которых операнды команд задаются с помощью *тестовых ситуаций* (ограничений, описывающих условия возникновения определенных событий в работе микропроцессора). Для конструирования значений операндов используются *генераторы тестовых данных*, основанные на *решателях ограничений*. После генерации значений операндов в тестовую программу добавляется *инициализирующий код*, в котором полученные значения записываются в соответствующие элементы памяти. Среда MicroTESK предоставляет набор *универсальных* и *специальных* решателей. Первые применимы для ограничений в широком классе теорий (булевой алгебры, целочисленной арифметики, теории битовых векторов и многих других). Вторые же предназначены для решения узкоспециализированных задач (например, для воспроизведения событий, связанных с подсистемой управления памятью [104, 105]).

4. Текущие работы и перспективы

При выборе тем для своих исследований ИСП РАН отталкивается от потребностей индустрии. Партнерские отношения с ведущими отечественными разработчиками микропроцессоров — НИИСИ РАН и ЗАО «МЦСТ» — позволяют изнутри увидеть проблемы, возникающие при промышленной верификации, и поставить актуальные задачи перед научными сотрудниками, аспирантами и студентами. Описанные выше подходы к проверке микропроцессоров были испытаны в реальных проектах, и, надо сказать, не всегда результаты их использования были положительными [73]. Мы тщательно анализируем негативный опыт, глубже проникаем в проблематику и стараемся найти решения возникающих задач. В этом разделе делается краткий обзор тем, над которыми мы работаем в настоящее время, и которые, так или иначе, имеют индустриальные корни.

4.1. Развитие средств модульной верификации

В промышленных технологиях верификации микропроцессоров возрастает роль формальных методов (проверки моделей, дедуктивного анализа, проверки эквивалентности). Одной из приоритетных задач модульной верификации является автоматическое построение формальной модели по HDL-описанию и генерация тестов на ее основе. Естественным формализмом для описания таких моделей является *расширенный конечный автомат* (*EFSM, Extended Finite State Machine*). В отличие от классического автомата, EFSM обладает переменными, а его переходы имеют особый вид. Каждый переход содержит охранное условие и действие над переменными: переход может сработать, только когда истинно охранное условие; при этом выполняется действие. Таким образом, EFSM — это множество *охраняемых действий* (см. раздел 3.1.1) над общими переменными, структурированное в систему переходов. Формализм охраняемых действий применяется для

описания сложных аппаратных систем, и для него разработаны эффективные алгоритмы синтеза [106].

Для извлечения EFSM-модели (автомата) из HDL-описания предлагается следующий подход [23]. Сначала из исходного кода извлекаются охраняемые действия. Затем из множества входных сигналов HDL-описания выделяется сигнал, по изменению которого меняется внутреннее состояние описываемого устройства (синхросигнал). После чего идентифицируются внутренние переменные HDL-описания, значения которых кодируют состояния устройства. Множество состояний автомата строится путем декомпозиции охранных условий, содержащих выделенные переменные состояния, на набор попарно несовместных ограничений (каждое из полученных ограничений соответствует одному состоянию EFSM-модели). Отношение переходов конструируется следующим образом. Для каждого действия проверяется совместимость его охранного условия с ограничениями, задающими состояния автомата. Таким образом определяется, из каких состояний есть переходы, помеченные данным охраняемым действием. Если имеется несколько состояний автомата, с которыми совместимо охраняемое условие, осуществляется «расщепление» перехода. Для того чтобы определить, в какое состояние ведет переход, проверяется совместимость результата символического выполнения действия с ограничениями состояний.

В качестве направлений дальнейших исследований рассматриваются следующие задачи: (1) генерация тестовых последовательностей на основе EFSM-моделей [107] и (2) формальный анализ EFSM-моделей на предмет достижимости ошибочных ситуаций (гонок, зависаний, конфликтов использования ресурсов) [108].

4.2. Развитие средств системной верификации

В рамках системной верификации микропроцессоров имеются две важные и взаимосвязанные темы, практически не затронутые нашими разработками: (1) верификация микропроцессоров на основе *моделей уровня микроархитектуры* и (2) верификация *многоядерных* микропроцессоров и *многопроцессорных* вычислительных комплексов. В настоящее время в ИСП РАН ведутся исследования по обоим направлениям. Работы по первой теме проводятся в контексте создания среды генерации тестовых программ MicroTESK (см. раздел 3.2). Исследования по второй теме выполняются совместно с компанией ЗАО «МЦСТ», занимающейся разработкой многоядерных микропроцессоров и систем на их основе: МЦСТ-R500S (2 ядра, 2008 г.), Эльбрус-2С+ (2+4 ядра, 2011 г.), МЦСТ-R1000 (4 ядра, 2012 г.) и других [109]. Рассмотрим кратко содержание обозначенных работ.

4.2.1 Моделирование на уровне микроархитектуры

Модели, используемые в настоящее время в MicroTESK, игнорируют устройство конвейера и динамику совместного выполнения команд. С одной

стороны, это делает инструмент проще и облегчает его изучение и использование. С другой стороны, это сказывается на качестве генерируемых тестовых программ — на некотором этапе увеличение уровня тестового покрытия микропроцессора становится возможным только при существенном увеличении объема тестов. Обычно эта проблема решается с помощью ручного создания *направленных тестов*, точечно покрывающих ситуации, оставшиеся неохваченными, однако такой подход имеет два очевидных недостатка: (1) структурные метрики тестового покрытия (см. раздел 2.2) не позволяют выразить ситуации, связанные с взаимодействиями параллельных процессов; (2) ручная разработка тестов требует много ресурсов.

Модели уровня микроархитектуры позволяют задавать более адекватные метрики тестового покрытия, а также автоматизировать генерацию тестов, направленных на возникновение определенных событий в работе микропроцессора (особых случаев выполнения операций, прерываний, ситуаций заполнения буферов и т.п.). Кроме того, они делают возможной формальную верификацию проекта и могут использоваться для синтеза HDL-описания и генерации таких инструментов, как симулятор и компилятор (как правило, чем детальнее модель, тем точнее симулятор и эффективнее компилятор). В свою очередь, наличие средств кросс-разработки позволяет исследовать альтернативы проектных решений (выбор той или иной микроархитектуры) с использованием ПО (типовых задач), предназначенного для целевого микропроцессора [19].

Спецификация микропроцессора — это очень сложная задача, и выбор адекватного формализма имеет первостепенное значение. Важными требованиями к формализму, используемому для описания микроархитектуры, являются: (1) возможность частичной спецификации проекта (возможность инкрементального описания подсистем и связей между ними), (2) возможность спецификации на разных уровнях абстракции (от обобщенных протоколов взаимодействия подсистем до детально описанных коммуникационных сетей), (3) возможность повторного использования элементов моделей (возможность составления моделей из имеющихся частей). Скорее всего, решение представляет собой комбинацию различных подходов: сетей Петри, конечных автоматов, алгебр и исчислений процессов.

Среди возможных вариантов рассматриваются следующие нотации: (1) графические языки типа UML (Unified Modeling Language) [110]; (2) Sim-nML [101] — расширение ADL-языка nML, используемого в генераторе тестовых программ MicroTESK (см. раздел 3.2.2); (3) EXPRESSION [19] — ADL-язык смешанного типа (позволяющий описывать не только систему команд микропроцессора, но и его структуру); (4) xMAS (Executable MicroArchitecture Specifications) [111] — язык описания коммуникационных сетей; (5) Promela (Process meta language) [112] — язык описания процессов и протоколов, используемый в инструменте проверки моделей SPIN;

(6) Bluespec [76] — язык, основанный на парадигме атомарных охраняемых действий.

4.2.2 Верификация реализаций протоколов когерентности

Вторая задача (верификация многоядерных микропроцессоров и многопроцессорных комплексов) связана с проверкой механизмов обеспечения *когерентности памяти*. Каждый узел системы (вычислительное ядро или микропроцессор) имеет в своем составе кэш-память, из-за чего в системе могут существовать несколько копий одних данных – одна копия в основной памяти и несколько копий в кэш-памяти узлов. При изменении копии данных в одном из узлов другие совладельцы этих данных должны согласованным образом изменить свои копии (или удалить их). Взаимодействие между узлами системы осуществляется согласно *протоколу когерентности*, за реализацию которого отвечают соответствующие механизмы подсистемы управления памятью. Их разработка осуществляется в два этапа: (1) проектирование протокола когерентности (и создание его формальной модели); (2) реализация протокола в аппаратуре. На первом этапе осуществляется верификация *модели* протокола; для этого используются формальные методы (прежде всего, проверка моделей и дедуктивный анализ). Для проверки *реализации* протокола (HDL-модели подсистемы управления памятью) обычно используется случайная генерация тестов, что не гарантирует выявления всех ошибок (пример – ошибка #298 в AMD Phenom [9]).

Представляется целесообразным использовать модели, применяемые для верификации протоколов когерентности, для тестирования HDL-моделей подсистем управления памятью. При этом возникают две задачи: (1) генерация тестов по модели и (2) детерминированное (сохраняющее порядок событий) воспроизведение построенных тестов на реализации. Первая задача решается методами тестирования на основе моделей (в том числе методами обхода графов состояний). Исследования, проводимые совместно с ЗАО «МЦСТ», сфокусированы на второй задаче и нацелены на создание средств, позволяющих тестировать реализации протоколов когерентности на основе *трасс* выполнения моделей. Разрабатываемые средства позволяют воспроизводить трассы на реализации, проверять корректность поведения реализации и оценивать уровень тестового покрытия. Кроме того, средства являются универсальными — применимыми к системам, основанным на разных архитектурах и протоколах когерентности [113]. Прототип инструмента позволил обнаружить ошибку в эталонной модели подсистемы управления памятью, которая не была обнаружена другими способами верификации.

4.3. Унификация и интеграция средств верификации

Среди задач, решаемых разными средствами верификации микропроцессоров (как модульного, так и системного уровней), есть много похожих. Унификация интерфейсов *механизмов* (*engines*) решения общих подзадач и создание библиотек переиспользуемых компонентов снижает трудоемкость разработки инструментов верификации, упрощает их интеграцию друг с другом и создает предпосылки для создания расширяемых сред [20, 44, 114]. Так, во многих средствах верификации (формальной и имитационной) применяются техники разрешения ограничений и реализующие их инструменты — *решатели* (*solvers*), например, Yices [115], Z3 [116] и другие. Сфера применения решателей широка: генерация тестовых данных [26], статический анализ HDL-описаний [23], символическая проверка моделей [117]. Ввиду многочисленности таких инструментов (каждый решатель лучше справляется с ограничениями того или иного вида) имеется потребность в унификации интерфейсов работы с ними.

В ИСП РАН была разработана библиотека JCS API (Java Constraint Solver API) [118], позволяющая описывать ограничения в форме объектов (синтаксических деревьев) на языке Java (на котором разработаны многие наши инструменты) и вызывать внешние решатели. В качестве основного средства взаимодействия JCS API с решателями используется язык SMT-LIB (Satisfiability Modulo Theories Library) [119], поддерживаемый многими инструментами. Библиотека, изначально разработанная в рамках проекта MicroTESK [47], где она применялась для генерации тестовых данных (см. раздел 3.2.2), в настоящее время активно используется в наших разработках в области анализа HDL-описаний (см. раздел 4.1). Развитие JCS API осуществляется по трем направлениям: (1) расширение множества поддерживаемых операций и типов данных; (2) разработка средств трансформации ограничений (и других формул); (3) создание эффективных решателей, ориентированных на типичные виды ограничений, встречающиеся при верификации аппаратуры.

Другими работами общего характера, выполняемыми в ИСП РАН и связанными с верификацией микропроцессоров, являются: (1) эффективная реализация операций над битовыми векторами (такие операции могут быть полезны при построении симуляторов HDL- и ADL-моделей, а также при создании генераторов тестовых данных); (2) разработка библиотеки для представления моделей аппаратуры (автоматов, сетей и т.п.) и их анализа (библиотека может применяться в составе разных инструментов верификации, включая генераторы тестов и инструменты проверки моделей); (3) разработка средств интеграции разных инструментов верификации (преобразователей моделей, представленных в формате одного инструмента, в формат другого). В качестве примера интеграции разных инструментов верификации можно привести следующий. Для статического анализа HDL-описаний (см. раздел 4.1) мы используем открытую платформу ZamiaCAD [120], которая по

описанию аппаратуры на языке VHDL строит *граф экземпляров модулей* (*IG*, *Instantiation Graph*), преобразуемый с помощью средств интеграции во внутреннее представление наших инструментов.

5. Заключение

Несмотря на развитие методов функциональной верификации, их возможности не справляются со все возрастающей сложностью микропроцессоров [3]. Если первые устройства мог проверить буквально один человек путем непосредственного анализа (экспертизы) схемы из логических вентилей или даже транзисторов (микросхема 4004, выпущенная в 1971 г., состояла менее чем из 2.5 тысяч транзисторов, соединения между которыми можно изобразить на листе ватмана), то сейчас штат инженеров-верификаторов достигает внушительных размеров (последние образцы микропроцессоров содержат в 2 миллиона раз большее число транзисторов по сравнению с микросхемой 4004 — см. рис. 1 [2]). Костяк команды, работающей над проверкой Pentium Pro (5.5 миллиона транзисторов), состоял из 10 человек, в проекте Pentium 4 (42 миллиона транзисторов) к ним присоединилось еще 60 [6]. Скорее всего, сейчас штат верификаторов в таких компаниях, как Intel, насчитывает несколько сотен специалистов (с тех пор прошло более 10 лет, а число транзисторов выросло более чем в 100 раз). Трудоемкость верификации огромна — сотни человеко-лет и миллионы часов компьютерного времени (только на формальную проверку Pentium 4 было затрачено 60 человеко-лет [51]).

Мечтой всех производителей микропроцессоров является автоматическое доказательство корректности проектирования. Некоторые компании, университеты и исследовательские институты проводят большую работу, чтобы приблизиться к этой мечте. По заявлению Боба Бентли (Bob Bentley), к 2015 г. Intel планирует увеличить долю использования формальных методов в модульной верификации микропроцессоров до 50% — это серьезный вызов, который потребует кардинальной перестройки процессов проектирования [14]. Если говорить о ситуации в настоящий момент времени, то основным подходом по-прежнему остается тестирование; но, нужно отметить, методы тестирования не стоят на месте и многое заимствуют из формальных методов. Скорее всего, взаимное проникновение разных методов верификации будет продолжаться, что приведет к появлению новых гибридных методов. Именно таким подходам, находящимся на стыке разных методов, посвящены исследования, выполняемые в ИСП РАН: формальная спецификация (на уровнях микро- и макроархитектуры), генерация тестов по моделям (потактовым, событийным, обобщенным), статический анализ HDL-описаний и другие. Результаты этих исследований нашли воплощение в инструментах CTESK [52], C++TESK [81], MicroTESK [47] и JCS API [118], которые успешно применялись в промышленных проектах.

Отдельно следует упомянуть проблему кадров. Кен Элбин (Ken Albin) из компании Motorola справедливо утверждает, что инженеры-верификаторы должны иметь совершенно другой набор знаний и навыков, нежели инженеры-проектировщики [51]. Среди качеств, желательных для верификатора, можно отметить следующие: умение изменять уровень абстракции модели, опыт программирования, знание основ проектирования аппаратуры, умение работать с огромными массивами данных, хорошие коммуникационные навыки и внутренняя мотивация. Кроме того, для работы в области формальной верификации требуется серьезная математическая подготовка (дискретная математика, математическая логика, теория трансляции и преобразования программ). Многие коммерческие компании осознают необходимость в подготовке высококлассных верификаторов и сотрудничают с ведущими университетами. Для развития отрасли крайне важно, чтобы университеты и исследовательские институты вели образовательную деятельность в области верификации и готовили квалифицированные кадры. На протяжении нескольких лет в ИСП РАН читаются лекции по методам верификации для студентов старших курсов МФТИ. Мы рассчитываем, что работа Института будет способствовать росту надежности микропроцессоров.

Список литературы

[1] В.В. Кулямин. *Методы верификации программного обеспечения*, 2008. 111 с. (http://www.ispras.ru/~kuliamin/docs/VerMethods-2008-ru.pdf)

[2] *Статистика числа транзисторов в микропроцессорах* — http://en.wikipedia.org/wiki/Transistor_count

[3] А.С. Камкин. *Верификация микропроцессоров: борьба с ошибками и управление качеством*. Электроника: НТБ, №3, 2010. С. 98-104.

[4] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000. 354 p.

[5] G.E. Moore. *Cramming More Components onto Integrated Circuits*. Electronics Magazine, 86(1), 1965. P. 82-85. (http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf)

[6] B. Bentley. *Validating the Intel® Pentium 4® Microprocessor*. Design Automation Conference (DAC), 2001. P. 244-248.

[7] B. Bentley. *Validating a Modern Microprocessor*. International Conference on Computer Aided Verification (CAV), 2005. P. 2-4. (http://www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt)

[8] *FDIV Replacement Program: Description of the Flaw* — http://www.intel.com/support/processors/pentium/sb/CS-013007.htm

[9] *Revision Guide for AMD Family 10h Processors*. Revision 3.84, August 2011. (http://developer.amd.com/wordpress/media/2012/10/41322.pdf)

[10] S. Barak. *Intel Denies Core i7 TLB Bug*. The Inquirer, December 02 2008. (http://www.theinquirer.net/inquirer/news/1049427/intel-denies-core-i7-tlb-bug)

[11] *Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series*. Specification Update, May 2011. (http://download.intel.com/design/processor/specupdt/320836.pdf)

[12] E.M. Clarke, J.M. Wing. *Formal Methods: State of the Art and Future Directions.* ACM Computing Surveys, 28(4), 1996. P. 626-643.

[13] J. Harrison. *Formal Methods at Intel — An Overview*. Second NASA Formal Methods Symposium (NFM), 2010. (http://www.cl.cam.ac.uk/~jrh13/slides/nasa-14apr10/slides.pdf)

[14] P. McLellan. *History of Formal Verification at Intel*. DAC Blog, December 05 2012. (http://blog.dac.com/post/2012/12/05/History-of-Formal-Verification-at-Intel.aspx)

[15] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, A. Naik. *Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation*. International Conference on Computer Aided Verification (CAV), 2009. P. 414-429.

[16] J. Bhadra, M. Abadir, S. Ray, L. Wang. *A Survey of Hybrid Techniques for Functional Verification*. IEEE Design & Test of Computers, 24(22), 2007. P. 112-122.

[17] Z. Navabi. *Languages for Design and Implementation of Hardware*. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[18] S. Mikhani, Z. Navabi. *System Level Design Languages*. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[19] P. Mishra, N. Dutt (Eds.). *Processor Description Languages. Systems on Silicon*. Morgan Kaufmann, 2008. 432 p.

[20] В. Кулямин. *Перспективы интеграции методов верификации программного обеспечения*. Труды ИСП РАН, 16, 2009. С. 73-88.

[21] S. Agbaria, D. Carmi, O. Cohen, D. Korchemny, M. Lifshits, A. Nadel. *SAT-Based Semiformal Verification of Hardware*. Formal Methods in Computer-Aided Design (FMCAD), 2010. P. 25-32.

[22] H. Jain, D. Kroening, N. Sharygina, E. Clarke. *VCEGAR: Verilog Counterexample Guided Abstraction Refinement* Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2007. P. 583-586.

[23] A. Kamkin, S. Smolov, I. Melnichenko. *Static Analysis of HDL Descriptions: Extracting Models for Verification.* East-West Design and Test Symposium (EWDTS), 2013. P. 184-187.

[24] W.K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005. 624 p.

[25] N. Bombieri, F. Fummi, G. Pravadelli, J. Marques-Silva. *Towards Equivalence Checking between TLM and RTL Models.* IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2007. P. 113-122.

[26] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification*. AI Magazine, 28(3), 2007. P. 13-30.

[27] *Технология верификации UVM —* http://www.accellera.org/community/uvm

[28] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. *Подход UniTesK к разработке тестов: достижения и перспективы*. Труды ИСП РАН, 5, 2004. С. 151-156.

[29] K. Sen, G. Rosu. *Generating Optimal Monitors for Extended Regular Expressions*. Electronic Notes in Theoretical Computer Science, 89(2), 2003. P. 162-181.

[30] В.П. Иванников, А.С. Камкин, А.С. Косачев, В.В. Кулямин, А.К. Петренко. *Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры*. Программирование, № 5, 2007. С. 47-61.

[31] H. Barringer, D. Rydeheard, K. Havelund. *Rule Systems for Run-Time Monitoring: From Eagle to RuleR*. International Workshop on Runtime Verification, 2007. P. 111-125.

[32] A. Bauer, M. Leucker, C. Schallhart. *Runtime Verification for LTL and TLTL*. ACM Transactions on Software Engineering and Methodology, 20(4), 2011. P. 14:1-14:64.

[33] A. Pnueli. *Temporal Logic of Programs*. Symposium on Foundation of Computer Science (SFCS), 1977. P. 46-57.

[34] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, Y. Zbar. *The ForSpec Temporal Logic: A New Temporal Property-Specification Language*. Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2002. P. 296-311.

[35] *OpenVera® Language Reference Manual: Assertions*. Version 1.4.1, November 2004.

[36] *1850-2010 — IEEE Standard for Property Specification Language (PSL)*, 2010.

[37] *1647-2011 — IEEE Standard for the Functional Verification Language e*, 2011.

[38] *1800-2012 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, 2013.

[39] A. Pizialli. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004. 216 p.

[40] M. Chupilko, A. Kamkin. *A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels.* Latin American Test Workshop (LATW), 2011. P. 1-6.

[41] 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.* IEEE Design & Test of Computers, 21(2), 2004. P. 84-93.

[42] А.С. Камкин. *Генерация тестовых программ для микропроцессоров*. Труды ИСП РАН, 14(2), 2008. С. 23-63.

[43] *Тестовая программа PARANOIA* — http://people.sc.fsu.edu/~%20jburkardt/c_src/paranoia/paranoia.html

[44] А.С. Камкин, Т.И. Сергеева, С.А. Смолов, А.Д. Татарников, М.М. Чупилко. *Расширяемая среда генерации тестовых программ для микропроцессоров*. Программирование, № 1, 2014. (*в печати*)

[45] *Слайды по генератору тестовых программ RAVEN* — http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc

[46] I.V. Gribkov, A.V. Zakharov, P.P. Koltsov, N.V. Kotovich, A.A. Kravchenko, A.S. Koutsaev, A.S. Osipov, I.S Khisambeev. *INTEG: A Stochastic Testing System for Microprocessor Verification*. WSEAS International Conference on Circuits, Systems, Signal and Telecommunications (CSST), 2007. P. 55-59.

[47] *Страница инструмента MicroTESK* — http://forge.ispras.ru/projects/microtesk

[48] *Страница инструмента Genesys-Pro* — https://www.research.ibm.com/haifa/projects/verification/genesys_pro/index.shtml

[49] P. Mishra, N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors.* ACM Transactions on Design Automation of Electronic Systems, 13(3), 2008. P. 1-36.

[50] T.N. Dang, A. Roychoudhury, T. Mitra, P. Mishra. *Generating Test Programs to Cover Pipeline Interactions.* Design Automation Conference (DAC), 2009. P. 142-147.

[51] N. Mokhoff. *Intel, Motorola Report Formal Verification Gains.* EE Times, June 21 2001. (http://www.eetimes.com/document.asp?doc_id=1215957)

[52] *Страница инструмента CTESK* — http://forge.ispras.ru/projects/ctesk

[53] В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.К. Петренко. *Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения*. Препринт ИСП РАН, 2005. 16 с.

[54] Д.Н. Воробьев, А.С. Камкин. *Генерация тестовых программ для подсистемы управления памятью микропроцессора*. Труды ИСП РАН, 17, 2009. С. 119-132.

[55] С.И. Аряшев, А.С. Камкин, Б.Ю. Рогаткин. *Тестирование RTL-моделей аппаратуры с помощью технологии UniTESK на примере блока преобразования адресов микропроцессора*. Электроника, микро- и наноэлектроника, 2007. С. 183-187.

[56] I. Bourdonov, A. Kossatchev, A. Petrenko, D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications. Formal Methods.* World Congress on Formal Methods in the Development of Computing Systems (FM), 1, 1999. P. 608-621.

[57] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай*. Программирование, № 5, 2003. С. 11-30.

[58] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай*. Программирование, № 1, 2004. С. 4-24.

[59] B. Meyer. *Design by Contract*. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc, 1986.

[60] R.W. Floyd. *Assigning Meaning to Programs*. Symposium on Applied Mathematics, 1967. P. 19-32.

[61] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10), 1969. P. 576-585.

[62] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 217 p.

[63] V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev, I. Bourdonov. *Integration of Functional and Timed Testing of Real-Time and Concurrent Systems*. Perspectives of System Informatics, 2003. P. 450-461.

[64] A. Kamkin. *Contract Specification of Pipelined Designs: Application to Testbench Automation*. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2007. P. 7-13.

[65] A. Kamkin. *Coverage-Directed Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications.* East-West Design & Test Symposium (EWDTS), 2008. P. 84-87.

[66] А.С. Камкин. *Метод формальной спецификации аппаратуры с конвейерной организацией и его приложение к задачам функционального тестирования*. Труды ИСП РАН, 16, 2009. С. 107-128.

[67] M. Chupilko, A. Kamkin. *Developing Cycle-Accurate Contract Specifications for Synchronous Parallel-Pipeline Hardware: Application to Verification*. Baltic Electronics Conference (BEC), 2010. P. 185-188.

[68] M. Chupilko, A. Kamkin. *Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs*. NORCHIP, 2009. P. 1-4.

[69] M. Chupilko, A. Kamkin. *Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification*. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2010. P. 125-129.

[70] А.В. Хорошилов. *Спецификация и тестирование систем с асинхронным интерфейсом*. Препринт ИСП РАН, 2006. 139 с.

[71] А.С. Камкин, М.М. Чупилко. *Тестирование модулей арифметики с плавающей точкой микропроцессоров на соответствие стандарту IEEE 754*. Труды ИСП РАН, 14(2), 2008. С. 7-22.

[72] M. Chupilko, A. Kamkin, D. Vorobyev. *Methodology and Experience of Simulation-Based Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications.* Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2008. P. 25-31.

[73] Р.А. Баратов, А.С. Камкин, В.М. Майорова, А.Н. Мешков, А.А. Сортов, М.А. Якушева. *Трудности модульной верификации аппаратуры на примере буфера команд микропроцессора «Эльбрус-2S».* Вопросы радиоэлектроники, № 3, 2013. С. 84-96.

[74] E. Dijkstra. *Guarded Commands, Non-determinacy and Formal Derivation of Programs*. Communications of the ACM, 18(8), 1975. P. 453-457.

[75] D.L. Rosenband, Arvind. *Hardware Synthesis from Guarded Atomic Actions with Performance Specifications*. International Conference on Computer-Aided Design (ICCAD), 2005. P. 784-791.

[76] *Bluespec^{TM} SystemVerilog Reference Guide*, 2012.

[77] C. Clare. *Designing Logic Systems Using State Machines*. McGraw-Hill, 1973. 150 p.

[78] S. Baranov. *Logic and System Design of Digital Systems*. TUT Press, 2008. 266 p.

[79] В.П. Иванников, А.С. Камкин, М.М. Чупилко. *Проверка корректности поведения HDL-моделей цифровой аппаратуры на основе динамического сопоставления трасс.* Tools & Methods of Program Analysis (TMPA), 2013. С. 71-82.

[80] M. Chupilko, A. Kamkin. *Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces.* Model-Based Testing Workshop (MBT), 2013. P. 67-81.

[81] *Страница инструмента C++TESK* — http://forge.ispras.ru/cpptesk

[82] G. von Bochmann, S. Haar, C. Jard, G.V. Jourdan. *Testing Systems Specified as Partial Order Input/Output Automata*. International Conference on Testing of Software and Communicating Systems (TestCom), 2008. P. 169-183.

[83] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Использование конечных автоматов для тестирования программ.* Программирование, 26(2), 2000. С. 61-73.

[84] И.Б. Бурдонов, С.Г. Грошев, А.В. Демаков, А.С. Камкин, А.С. Косачев, А.А. Сортов. *Параллельное тестирование больших автоматных моделей*. Вестник ННГУ, № 3, 2011. С. 187-193.

[85] A. Demakov, A. Kamkin, A. Sortov. *High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration*. Open Cirrus Summit, 2011.

[86] И.Б. Бурдонов, А.С. Косачев. *Обход неизвестного графа коллективом автоматов.* Научный сервис в сети Интернет: все грани параллелизма, 2013. С. 228-232.

[87] А.В. Демаков, С.А. Зеленова, С.В. Зеленов. *Тестирование парсеров текстов на формальных языках.* Программные системы и инструменты: Тематический сборник факультета ВМиК МГУ, № 2, 2001. С. 150-156.

[88] С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. *Применение модельного подхода для автоматического тестирования оптимизирующих компиляторов*, 2003.(http://citforum.ru/SE/testing/compilers)

[89] L.-M. Wu, K. Wang, C.-Y. Chiu. *A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification.* ACM Transactions on Design Automation of Electronic Systems, 9(1), 2004. P. 105-132.

[90] А.С. Камкин. *Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров*. Труды ИСП РАН, 18, 2010. С. 129-149.

[91] Д.Н. Воробьев, А.С. Камкин. *Генерация тестовых программ для микропроцессоров на основе шаблонов конвейерных конфликтов*. Труды ИСП РАН, 18, 2010. С. 91-113.

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

[93] *RM7000 Family User Manual*. Issue 1, May 2001.

[94] А.С. Камкин. *Комбинаторная генерация тестовых программ для микропроцессоров на основе моделей.* Препринт ИСП РАН, 2008. 18 с.

[95] A. Kamkin, E. Kornykhin, D. Vorobyev. *Reconfigurable Model-Based Test Program Generator for Microprocessors*. International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011. P. 47-54.

[96] A. Kamkin, A. Tatarnikov. *MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors.* Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012. P. 64-69.

[97] M. Freericks. *The nML Machine Description Formalism. Techical Report*. TU Berlin, FB20, Bericht, 1991/15. 47 p.

[98] *Страница компании Target Compiler Technologies* — http://www.retarget.com

[99] S. Chandra, R. Moona. *Retargetable Functional Simulator using High Level Processor Models*. VLSI Design, 2000. P. 424-429.

[100] *Страница инструмента GLISS* — http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=54

[101] H. Cassé, J. Barre, R. Vaillant, P. Sainrat. *Fast Instruction-Accurate Simulation with SimNML*. Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), 2011. P. 8-12.

[102] A. Kamkin, T. Sergeeva, A. Tatarnikov, A. Utekhin. *MicroTESK: An Extendable Framework for Test Program Generation*. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2013. P. 51-57.

[103] *Язык программирования Ruby* — http://www.ruby-lang.org

[104] E. Kornykhin. *SMT-Based Test Program Generation for Cache-Memory Testing.* East-West Design & Test Symposium (EWDTS), 2009. P. 124-127.

[105] E. Kornykhin. *Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors.* Programming and Computing Software, 36(1), 2010. P. 28-35.

[106] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, J.-P. Talpin. *Integrating System Descriptions by Clocked Guarded Actions.* Forum on Design Languages, 2011. P. 1-8.

[107] G. Guglielmo, L. Guglielmo, F. Fummi, G. Pravadelli. *Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs.* Journal of Electronic Testing, 27(2), 2011. P. 37-162.

[108] B. Karaçali, K.-C. Tai, M.A. Vouk. *Deadlock Detection of EFSMs using Simultaneous Reachability Analysis*. Dependable Systems and Networks (DSN), 2000. P. 315-324.

[109] А.К. Ким, В.И. Перекатов, С.Г. Ермаков. *Микропроцессоры и вычислительные комплексы семейства «Эльбрус»*. СПб.: Питер, 2013. 272 с.

[110] *Сайт ресурсов UML* — http://www.uml.org

[111] S. Chatterjee, M. Kishinevsky, U. Ogras. *xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification*. IEEE Design & Test of Computers, 2011. P. 80-88.

[112] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003. 608 p.

[113] А.С. Камкин, М.В. Петроченков. *Система поддержки верификации реализаций протоколов когерентности с использованием формальных методов*. Вопросы радиоэлектроники, серия ЭВТ, 2014. (*в печати*)

[114] K. Schneider, T. Kropf. *A Unified Approach for Combining Different Formalisms for Hardware Verification.* International Conference on Formal Methods in Computer Aided Design (FMCAD), 1996. P. 202-217.

[115] B. Dutertre, L. Moura. *The YICES SMT Solver*, 2006. (http://yices.csl.sri.com/tool-paper.pdf).

[116] L. Moura, N. Bjørner. *Z3: An Efficient SMT Solver.* Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008. P. 337-340.

[117] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993. 194 p.

[118] *Библиотека Java Constraint Solver API* — http://forge.ispras.ru/projects/solver-api

[119] D.R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial.* GrammaTech, Inc., Version 1.1, 2011.

[120] *Страница инструмента ZamiaCAD* — http://zamiacad.sourceforge.net

Tools for Functional Verification of Microprocessors

A. Kamkin, A. Kotsynyak, S. Smolov, A. Sortov, A. Tatarnikov, M. Chupilko
ISP RAS, Moscow, Russia
{kamkin,kotsynyak,ssedai,sortov,andrewt,chupilko}@ispras.ru

Abstract. Ensuring the correctness of microprocessors and other microelectronic equipment is a fundamental problem. To deal with it, various tools for functional verification are used. Unlike bugs in software programs which are relatively easy to fix (it does not apply to their consequences), defects in integrated circuits (both design and manufacturing ones) cannot be removed. In spite of continuous development of computer-aided design (CAD) systems, test generation tools and approaches to analysis of circuits, verification remains the bottleneck of the microprocessor design cycle (it accounts for approximately 70 percent of total design resources). The article gives a brief overview of microprocessor verification tools, describes issues that commonly occur in industrial practice and analyzes possible ways to solve them. The main part of the article is dedicated to research in the field of unit- and system-level hardware verification conducted at ISPRAS. It describes such approaches as contract specification of pipeline, event-driven hardware specification, parallel/distributed testing, combinatorial test program generation and template-based test program generation. The article also summarizes the outcomes of accomplished projects, describes the present works and formulates the directions of further research.

Keywords: microprocessors, hardware, verification, validation, testing, test generation, modeling, architecture description languages, parallelization.

References

[1]. Kuliamin V.V. Metody verifikatsii programmnogo obespecheniya [Methods for Software Verification]. 2008, 111 p. (in Russian). (http://www.ispras.ru/~kuliamin/docs/VerMethods-2008-ru.pdf)
[2]. Transistor count in microprocessors — http://en.wikipedia.org/wiki/Transistor_count
[3]. Kamkin A.S. Verifikatsiya mikroprotsessorov: bor'ba s oshibkami i upravlenie kachestvom [Microprocessor Verification. Combating Errors and Quality Control]. EHlektronika: NTB [Electronics: Science, Technology, Busincess], no. 3, 2010. pp. 98-104 (in Russian).
[4]. Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Kluwer Academic Publishers, 2000. 354 p. doi:10.1007/978-1-4615-0302-6
[5]. Moore G.E. Cramming More Components onto Integrated Circuits. Electronics Magazine, vol. 86, no. 1, 1965. P. 82-85. (http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf) doi:10.1109/JPROC.1998.658762
[6]. Bentley B. Validating the Intel® Pentium 4® Microprocessor. Proc. Design Automation Conference (DAC), 2001. pp. 244-248. doi:10.1145/378239.378473
[7]. Bentley B. Validating a Modern Microprocessor. Proc. International Conference on Computer Aided Verification (CAV), 2005. pp. 2-4.

(http://www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt) (DOI 10.1007/11513988_2)
[8]. FDIV Replacement Program: Description of the Flaw — http://www.intel.com/support/processors/pentium/sb/CS-013007.htm
[9]. Revision Guide for AMD Family 10h Processors. Revision 3.84, August 2011. (http://developer.amd.com/wordpress/media/2012/10/41322.pdf)
[10]. Barak S. Intel Denies Core i7 TLB Bug. The Inquirer, December 02 2008. (http://www.theinquirer.net/inquirer/news/1049427/intel-denies-core-i7-tlb-bug)
[11]. Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series. Specification Update, May 2011. (http://download.intel.com/design/processor/specupdt/320836.pdf)
[12]. Clarke E.M., Wing J.M. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, vol. 28, no. 4, 1996. pp. 626-643.
[13]. Harrison J. Formal Methods at Intel — An Overview. Second NASA Formal Methods Symposium (NFM), 2010. (http://www.cl.cam.ac.uk/~jrh13/slides/nasa-14apr10/slides.pdf)
[14]. McLellan P. History of Formal Verification at Intel. DAC Blog, December 05 2012. (http://blog.dac.com/post/2012/12/05/History-of-Formal-Verification-at-Intel.aspx)
[15]. Kaivola R, Ghughal R, Narasimhan N, Telfer A, Whittemore J, Pandav S, Slobodová A, Taylor C, Frolov V, Reeber E, Naik A. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. Proc. International Conference on Computer Aided Verification (CAV), 2009. pp. 414-429. doi:10.1007/978-3-642-02658-4_32
[16]. Bhadra J, Abadir M, Ray S, Wang L. A Survey of Hybrid Techniques for Functional Verification. IEEE Design & Test of Computers, vol. 24, no. 22, 2007. pp. 112-122. doi:10.1109/MDT.2007.30
[17]. Navabi Z. Languages for Design and Implementation of Hardware. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.
[18]. Mikhani S, Navabi Z. System Level Design Languages. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.
[19]. Mishra P, Dutt N (Eds.). Processor Description Languages. Systems on Silicon. Morgan Kaufmann, 2008. 432 p.
[20]. Kuliamin V.V. Perspektivy integratsii metodov verifikatsii programmnogo obespecheniya [Prospects for Integration of Software Verification Methods]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009. pp. 73-88 (in Russian).
[21]. Agbaria S, Carmi D, Cohen O, Korchemny D, Lifshits M, Nadel A. SAT-Based Semiformal Verification of Hardware. Proc. Formal Methods in Computer-Aided Design (FMCAD), 2010. pp. 25-32.
[22]. Jain H, Kroening D, Sharygina N, Clarke E. VCEGAR: Verilog Counterexample Guided Abstraction Refinement. Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2007. pp. 583-586. doi:10.1007/978-3-540-71209-1_45
[23]. Kamkin A, Smolov S, Melnichenko I. Static Analysis of HDL Descriptions: Extracting Models for Verification. Proc. East-West Design and Test Symposium (EWDTS), 2013. pp. 184-187. doi:10.1109/EWDTS.2013.6673126
[24]. Lam W.K. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall, 2005. 624 p.

[25]. Bombieri N, Fummi F, Pravadelli G, Marques-Silva J. Towards Equivalence Checking between TLM and RTL Models. Proc. IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2007. pp. 113-122. doi:10.1109/MEMCOD.2007.371236
[26]. Naveh Y, Rimon M, Jaeger I, Katz Y, Vinov M, Marcus E, Shurek G. Constraint-Based Random Stimuli Generation for Hardware Verification. AI Magazine, vol. 28, no. 3, 2007. pp. 13-30. doi:10.1609/aimag.v28i3.2052
[27]. UVM verification methodology — http://www.accellera.org/community/uvm
[28]. Barantsev A.V., Bourdonov I.B., Demakov A.V., Zelenov S.V., Kossatchev A.S., Kuliamin V.V., Omeltchenko V.A., Pakoulin N.V., Petrenko A.K., Khoroshilov A.V. Podkhod UniTesK k razrabotke testov: dostizheniya i perspektivy [UniTesK Approach to Test Development: Achievements and Prospects]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 5, 2004. pp. 151-156 (in Russian).
[29]. Sen K, Rosu G. Generating Optimal Monitors for Extended Regular Expressions. Electronic Notes in Theoretical Computer Science, vol. 89, no. 2, 2003. pp. 162-181. doi:10.1016/S1571-0661(04)81051-X
[30]. Ivannikov V.P., Kamkin A.S., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. The Use of Contract Specifications for Representing Requirements and for Functional Testing of Hardware Models. Programming and Computer Software, vol. 33, no. 5, 2007. pp. 272-282. doi:10.1134/S0361768807050039
[31]. Barringer H, Rydeheard D, Havelund K. Rule Systems for Run-Time Monitoring: From Eagle to RuleR. Proc. International Workshop on Runtime Verification, 2007. pp. 111-125. doi:10.1007/978-3-540-77395-5_10
[32]. Bauer A, Leucker M, Schallhart C. Runtime Verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology, vol. 20, no. 4, 2011. pp. 14:1-14:64. doi:10.1145/2000799.2000800
[33]. Pnueli A. Temporal Logic of Programs. Proc. Symposium on Foundation of Computer Science (SFCS), 1977. pp. 46-57. doi:10.1109/SFCS.1977.32
[34]. Armoni R, Fix L, Flaisher A, Gerth R, Ginsburg B, Kanza T, Landver A, Mador-Haim S, Singerman E, Tiemeyer E, Vardi M, Zbar Y. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2002. pp. 296-311. doi:10.1007/3-540-46002-0_21
[35]. OpenVera® Language Reference Manual: Assertions. Version 1.4.1, November 2004.
[36]. 1850-2010 — IEEE Standard for Property Specification Language (PSL), 2010. doi:10.1109/IEEESTD.2005.97780
[37]. 1647-2011 — IEEE Standard for the Functional Verification Language e, 2011. doi:10.1109/IEEESTD.2011.6006495
[38]. 1800-2012 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, 2013. doi:10.1109/IEEESTD.2013.6469140
[39]. Pizialli A. Functional Verification Coverage Measurement and Analysis. Kluwer Academic Publishers, 2004. 216 p. doi:10.1007/b117979
[40]. Chupilko M, Kamkin A. A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels. Proc. Latin American Test Workshop (LATW), 2011. pp. 1-6. doi:10.1109/LATW.2011.5985902
[41]. Adir A, Almog E, Fournier L, Marcus E, Rimon M, Vinov M, Ziv A. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. IEEE

Design & Test of Computers, vol. 21, no. 2, 2004. pp. 84-93. doi:10.1109/MDT.2004.1277900
[42]. Kamkin A.S. Generatsiya testovykh programm dlya mikroprotsessorov [Test Program Generation for Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 14, no. 2, 2008. pp. 23-63 (in Russian).
[43]. Test program PARANOIA — http://people.sc.fsu.edu/~%20jburkardt/c_src/paranoia/paranoia.html
[44]. Kamkin A.S., Sergeeva T.I., Smolov S.A., Tatarnikov A.D., Chupilko M.M. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, vol. 40, no. 1, 2014. pp. 1-9. doi:10.1134/S0361768814010046
[45]. RAVEN test program generator — http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc
[46]. Gribkov I.V., Zakharov A.V., Koltsov P.P., Kotovich N.V., Kravchenko A.A., Koutsaev A.S., Osipov A.S., Khisambeev I.S. INTEG: A Stochastic Testing System for Microprocessor Verification. Proc. WSEAS International Conference on Circuits, Systems, Signal and Telecommunications (CSST), 2007. pp. 55-59.
[47]. MicroTESK tool — http://forge.ispras.ru/projects/microtesk
[48]. Genesys-Pro tool — https://www.research.ibm.com/haifa/projects/verification/genesys_pro/index.shtml
[49]. Mishra P., Dutt N. Specification-Driven Directed Test Generation for Validation of Pipelined Processors. ACM Transactions on Design Automation of Electronic Systems, vol. 13, no. 3, 2008. pp. 1-36. doi:10.1145/1367045.1367051
[50]. Dang T.N., Roychoudhury A., Mitra T., Mishra P. Generating Test Programs to Cover Pipeline Interactions. Proc. Design Automation Conference (DAC), 2009. pp. 142-147. doi:10.1145/1629911.1629953
[51]. Mokhoff N. Intel, Motorola Report Formal Verification Gains. EE Times, June 21 2001. (http://www.eetimes.com/document.asp?doc_id=1215957)
[52]. CTESK tool — http://forge.ispras.ru/projects/ctesk
[53]. Ivannikov V.P., Kamkin A.S., Kuliamin V.V., Petrenko A.K. Primenenie tekhnologii UniTesK dlya funktsional'nogo testirovaniya modelej apparatnogo obespecheniya [Application of the UniTESK Technology for Functional Testing of Hardware Models]. Preprint ISP RAN [Preprint of ISP RAS], 2005. 16 p. (in Russian).
[54]. Vorobyev D.N., Kamkin A.S. Generatsiya testovykh programm dlya podsistemy upravleniya pamyat'yu mikroprotsessora [Test Program Generation for Memory Management Units of Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 17, 2009. pp. 119-132 (in Russian).
[55]. Aryashev S.I., Kamkin A.S., Rogatkin B.Yu. Testirovanie RTL-modelej apparatury s pomoshh'yu tekhnologii UniTESK na primere bloka preobrazovaniya adresov mikroprotsessora [Using the UniTESK Technology for Testing RTL Hardware Models by the Example of the Microprocessor Translation Lookaside Buffer]. Proc. EHlektronika, mikro- i nanoehlektronika [Electronics, micro- and nanoelectronics], 2007. pp. 183-187 (in Russian).
[56]. Bourdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications. Proc. World Congress on Formal Methods in the Development of Computing Systems (FM), vol. 1, 1999. pp. 608-621. doi:10.1007/3-540-48119-2_34
[57]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, vol. 29, no. 5, 2003. pp. 245-258. doi:10.1023/A:1025733107700

[58]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case. Programming and Computer Software, vol. 30, no. 1, 2004. pp. 2-17. doi:10.1023/B:PACS.0000013436.72070.95
[59]. Meyer B. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc, 1986. doi:10.1109/2.161279
[60]. Floyd R.W. Assigning Meaning to Programs. Proc. Symposium on Applied Mathematics, 1967. pp. 19-32.
[61]. Hoare C.A.R. An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, no. 10, 1969. pp. 576-585. doi:10.1145/363235.363259
[62]. Dijkstra E.W. A Discipline of Programming. Prentice-Hall, 1976. 217 p.
[63]. Kuliamin V, Petrenko A, Pakoulin N, Kossatchev A, Bourdonov I. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. Proc. Perspectives of System Informatics, 2003. pp. 450-461. doi:10.1007/978-3-540-39866-0_45
[64]. Kamkin A. Contract Specification of Pipelined Designs: Application to Testbench Automation. Proc. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2007. pp. 7-13.
[65]. Kamkin A. Coverage-Directed Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications. Proc. East-West Design & Test Symposium (EWDTS), 2008. pp. 84-87. doi:10.1109/EWDTS.2008.5580153
[66]. Kamkin A.S. Metod formal'noj spetsifikatsii apparatury s konvejernoj organizatsiej i ego prilozhenie k zadacham funktsional'nogo testirovaniya [Method for Formal Specification of Pipelined Designs and Its Application to Functional Testing]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009. pp. 107-128 (in Russian).
[67]. Chupilko M, Kamkin A. Developing Cycle-Accurate Contract Specifications for Synchronous Parallel-Pipeline Hardware: Application to Verification. Proc. Baltic Electronics Conference (BEC), 2010. pp. 185-188. doi:10.1109/BEC.2010.5631143
[68]. Chupilko M, Kamkin A. Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs. Proc. NORCHIP, 2009. pp. 1-4. doi:10.1109/NORCHP.2009.5397808
[69]. Chupilko M, Kamkin A. Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification. Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2010. pp. 125-129.
[70]. Khoroshilov A.V. Spetsifikatsiya i testirovanie sistem s asinkhronnym interfejsom [Specification and Testing of Systems with Asynchronous Interfaces]. Preprint ISP RAN [Preprint of ISP RAS], 2006. 139 p (in Russian).
[71]. Kamkin A.S., Chupilko M.M. Testirovanie modulej arifmetiki s plavayushhej tochkoj mikroprotsessorov na sootvetstvie standartu IEEE 754 [Testing Microprocessor Floating Point Arithmetic Modules for Conformity to the IEEE 754 Standard]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 14, no. 2, 2008. pp. 7-22 (in Russian).
[72]. Chupilko M, Kamkin A, Vorobyev D. Methodology and Experience of Simulation-Based Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications. Proc. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2008. pp. 25-31.
[73]. Baratov R.A., Kamkin A.S., Mayorova V.M., Meshkov A.N., Sortov A.A., Yakusheva M.A. Trudnosti modul'noj verifikatsii apparatury na primere bufera komand mikroprotsessora Elbrus-2S [Difficulties of the Unit-Level Hardware Verification on the Example of the Instruction Buffer of the Elbrus-2S Microprocessor]. Voprosy radioehlektroniki [Issues of Radio Electronics], no. 3, 2013. pp. 84-96 (in Russian).

[74]. Dijkstra E. Guarded Commands, Non-determinacy and Formal Derivation of Programs. Communications of the ACM, vol. 18, no. 8, 1975. pp. 453-457. doi:10.1145/360933.360975
[75]. Rosenband D.L., Arvind. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. Proc. International Conference on Computer-Aided Design (ICCAD), 2005. pp. 784-791. doi:10.1109/ICCAD.2005.1560170
[76]. BluespecTM SystemVerilog Reference Guide, 2012.
[77]. Clare C. Designing Logic Systems Using State Machines. McGraw-Hill, 1973. 150 p.
[78]. Baranov S. Logic and System Design of Digital Systems. TUT Press, 2008. 266 p.
[79]. Ivannikov V.P., Kamkin A.S., Chupilko M.M. Proverka korrektnosti povedeniya HDL-modelej tsifrovoj apparatury na osnove dinamicheskogo sopostavleniya trass [Checking Behavior Correctness of HDL Models of Digital Hardware By On-the-Fly Trace Matching]. Proc. Tools & Methods of Program Analysis (TMPA), 2013. pp. 71-82 (in Russian).
[80]. Chupilko M, Kamkin A. Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces. Proc. Model-Based Testing Workshop (MBT), 2013. pp. 67-81. doi: 10.4204/EPTCS.111.6
[81]. C++TESK tool — http://forge.ispras.ru/cpptesk
[82]. von Bochmann G, Haar S, Jard C, Jourdan G.V. Testing Systems Specified as Partial Order Input/Output Automata. Proc. International Conference on Testing of Software and Communicating Systems (TestCom), 2008. pp. 169-183. doi:10.1007/978-3-540-68524-1_13
[83]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Using Finite State Machines in Program Testing. Programming and Computer Software, vol. 26, no. 2, 2000, pp. 61–73.
[84]. Bourdonov I.B., Groshev S.G., Demakov A.V., Kamkin A.S., Kossatchev A.S., Sortov A.A. Parallel'noe testirovanie bol'shikh avtomatnykh modelej [Parallel Testing of Large Automata Models]. Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo [Vestnik of Lobachevsky State University of Nizhny Novgorod], no. 3, 2011. pp. 187-193 (in Russian).
[85]. Demakov A, Kamkin A, Sortov A. High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration. Proc. Open Cirrus Summit, 2011.
[86]. Bourdonov I.B., Kossatchev A.S. Obkhod neizvestnogo grafa kollektivom avtomatov [Traversal of an Unknown Graph by a Collective of Automata]. Proc. Nauchnyj servis v seti Internet: vse grani parallelizma [Scientific Service in Internet: All Facets of Parallelism], 2013. pp. 228-232 (in Russian).
[87]. Demakov A.V., Zelenova S.A., Zelenov S.V. Testirovanie parserov tekstov na formal'nykh yazykakh [Testing Parsers of Formal Languages]. Programmnye sistemy i instrumenty: Tematicheskij sbornik fakul'teta VMiK MGU [Software Systems and Tools: Proceedings of CMC Department of MSU], no. 2, 2001. pp. 150-156 (in Russian).
[88]. Zelenov S.V., Zelenova S.A., Kossatchev A.S., Petrenko A.K. Primenenie model'nogo podkhoda dlya avtomaticheskogo testirovaniya optimiziruyushhikh kompilyatorov [Application of the Model-Based Approach to Automated Testing of Optimized Compilers], 2003 (in Russian). (http://citforum.ru/SE/testing/compilers)
[89]. Wu L.-M., Wang K, Chiu C.-Y. A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification. ACM Transactions on Design Automation of Electronic Systems, vol. 9, no. 1, 2004. pp. 105-132. doi:10.1145/966137.966142

[90]. Kamkin A.S. Nekotorye voprosy avtomatizatsii postroeniya testovykh programm dlya modulej obrabotki perekhodov mikroprotsessorov [Some Issues of Automation of Test Program Generation for Branch Units of Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 18, 2010. pp. 129-149 (in Russian).
[91]. Vorobyev D.N., Kamkin A.S. Generatsiya testovykh programm dlya mikroprotsessorov na osnove shablonov konvejernykh konfliktov [Test Program Generation for Microprocessors Based on Pipeline Hazards Templates]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 18, 2010. pp. 91-113 (in Russian).
[92]. MIPS64™ Architecture For Programmers, Revision 2.0. MIPS Technologies Inc, June 9 2003.
[93]. RM7000 Family User Manual. Issue 1, May 2001.
[94]. Kamkin A.S. Kombinatornaya generatsiya testovykh programm dlya mikroprotsessorov na osnove modelej [Combinatorial Model-Based Test Program Generation for Microprocessors]. Preprint ISP RAN [Preprint of ISP RAS], 2008. 18 p. (in Russian).
[95]. Kamkin A, Kornykhin E, Vorobyev D. Reconfigurable Model-Based Test Program Generator for Microprocessors. Proc. International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011. pp. 47-54. doi:10.1109/ICSTW.2011.35
[96]. Kamkin A, Tatarnikov A. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012. pp. 64-69.
[97]. Freericks M. The nML Machine Description Formalism. Techical Report. TU Berlin, FB20, Bericht, 1991/15. 47 p.
[98]. Target Compiler Technologies — http://www.retarget.com
[99]. Chandra S, Moona R. Retargetable Functional Simulator using High Level Processor Models. VLSI Design, 2000. pp. 424-429. doi:10.1109/ICVD.2000.812644
[100]. GLISS tool — http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=54
[101]. Cassé H, Barre J, Vaillant R, Sainrat P. Fast Instruction-Accurate Simulation with SimNML. Proc. Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), 2011. pp. 8-12.
[102]. Kamkin A, Sergeeva T, Tatarnikov A, Utekhin A. MicroTESK: An Extendable Framework for Test Program Generation. Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2013. pp. 51-57.
[103]. Ruby programming language — http://www.ruby-lang.org
[104]. Kornykhin E. SMT-Based Test Program Generation for Cache-Memory Testing. Proc. East-West Design & Test Symposium (EWDTS), 2009. pp. 124-127.
[105]. Kornykhin E. Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors. Programming and Computing Software, vol. 36, no. 1, 2010. pp. 28-35. doi:10.1134/S0361768810010056
[106]. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin J.-P. Integrating System Descriptions by Clocked Guarded Actions. Proc. Forum on Design Languages (FDL), 2011. pp. 1-8.
[107]. Guglielmo G, Guglielmo L, Fummi F, Pravadelli G. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. Journal of Electronic Testing, vol. 27, no. 2, 2011. pp. 37-162. doi:10.1007/s10836-011-5209-8
[108]. Karaçali B, Tai K.-C., Vouk M.A. Deadlock Detection of EFSMs using Simultaneous Reachability Analysis. Proc. Dependable Systems and Networks (DSN), 2000. pp. 315-324. doi:10.1109/ICDSN.2000.857555

[109]. Kim A.K., Perekatov V.I., Ermakov S.G. Mikroprotsessory i vychislitel'nye kompleksy semejstva Elbrus [Microprocessors and Computer Systems of the Elbrus Family]. Saint-Petersburg, Piter, 2013. 272 p. (in Russian).
[110]. UML language — http://www.uml.org
[111]. Chatterjee S, Kishinevsky M, Ogras U. xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification. IEEE Design & Test of Computers, vol. 29, no. 3, 2011.pp. 80-88. doi:10.1109/MDT.2011.72
[112]. Holzmann G.J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2003. 608 p.
[113]. Kamkin A.S., Petrochenkov M.V. Sistema podderzhki verifikatsii realizatsij protokolov kogerentnosti s ispol'zovaniem formal'nykh metodov [A System to Support Formal Methods-Based Verification of Coherence Protocol Implementations]. Voprosy radioehlektroniki [Issues of Radio Electronics], no. 3, 2014. pp. 27-38 (in Russian).
[114]. Schneider K, Kropf T. A Unified Approach for Combining Different Formalisms for Hardware Verification. Proc. International Conference on Formal Methods in Computer Aided Design (FMCAD), 1996. pp. 202-217. doi:10.1007/BFb0031809
[115]. Dutertre B, Moura L. The YICES SMT Solver, 2006. (http://yices.csl.sri.com/tool-paper.pdf).
[116]. Moura L, Bjørner N. Z3: An Efficient SMT Solver. Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008. pp. 337-340. doi:10.1007/978-3-540-78800-3_24
[117]. McMillan K.L. Symbolic Model Checking. Kluwer Academic, 1993. 194 p.
[118]. Java Constraint Solver API — http://forge.ispras.ru/projects/solver-api
[119]. Cok D.R. The SMT-LIBv2 Language and Tools: A Tutorial. GrammaTech, Inc., Version 1.1, 2011.
[120]. ZamiaCAD tool — http://zamiacad.sourceforge.net

Инструментальные средства проектирования систем интегрированной модульной авионики

Буздалов Д.В., Зеленов С.В., Корныхин Е.В., Петренко А.К., Страх А.В., Угненко А.А., Хорошилов А.В.
{buzdalov, zelenov, kornevgen, petrenko, strakh, ugnenko, khoroshilov} @ispras.ru

Аннотация. Масштабы современных комплексов бортового авиационного оборудования таковы, что их проектирование становится невозможным без привлечения средств автоматизации. В настоящее время в мире в этой области имеются с одной стороны закрытые разработки крупных авиакомпаний, таких как Boeing и Airbus, а с другой стороны ряд открытых международных проектов с разной степенью зрелости, доступности исходного кода и документации. В настоящей статье представлена отечественная разработка открытой системы поддержки проектирования и верификации комплексов бортового авиационного оборудования осуществляемая в ИСП РАН совместно с ГосНИИАС в рамках государственной программы по развитию Интегрированной Модульной Авионики.

Ключевые слова: интегрированная модульная авионика; ИМА; моделирование систем; AADL.

1. Введение

Разработка современных систем авионики и других критических по безопасности систем управления требует развитой методической и инструментальной поддержки. За рубежом имеются соответствующие инструменты, но развитие таких высокотехнологичных отечественных отраслей, как авиастроение, не может опираться только на них в силу, по крайней мере, двух причин. Во-первых, такие инструментальные средства достаточно дороги, во-вторых, что, вероятно, более важно, они «закрыты» для развития и адаптации силами отечественных исследователей и инженеров, что ведет к еще большему отставанию наших технологий в данной области.

Средства проектирования, разработки, верификации и валидации систем типа авионики традиционно поддерживают подход разработки систем на основе моделей (Model Driven Engineering – MDE и Model Driven System Engineering - MDSE), поскольку методы моделирования в разных его видах: натурного, полунатурного, математического, - всегда культивировались в авиастроении и смежных отраслях [1]. В последние 20-30 лет в области разработки

программного обеспечения появился новый вид моделирования, связанный с исследованиями по формальным спецификациям программ и использованием так называемых формальных методов для анализа, в частности, для верификации программных систем. Системы авионики сейчас – это комплексы программно-аппаратных средств, поэтому методы и подходы, наработанные в области проектирования и анализа авионики и программных систем, естественно, должны обогащать друг друга. По этой причине опыт ИСП РАН в области использования формальных методов верификации сложных программных и аппаратных систем, таких как операционные системы и микропроцессоры, позволил достаточно быстро освоить еще одно направление – разработку средств проектирования и интеграции систем авионики, поскольку многие задачи в этой новой области могут быть решены на основе технологий моделирования и верификации, созданных в ИСП РАН ранее. Решающее значение в достижении полученных в этом направлении результатов оказала поддержка академика Е. А. Федосова, Г. А. Чуянова, И. В. Ковернинского и других сотрудников ГосНИИАС, работающих в рамках государственной программы по развитию Интегрированной Модульной Авионики (ИМА). В настоящей статье рассматривается опыт развития методов моделирования, синтеза и верификации сложных систем применительно к ИМА воздушных судов, но сфера потенциального применения технологии существенно шире.

Последующие разделы статьи построены следующим образом. В разделе 2 рассматриваются основные идеи подхода ИМА, новые задачи в проектировании и интеграции программного и аппаратного обеспечения, возникающие при его внедрении, а также возможности применения архитектурных моделей для автоматизации этих задач. Раздел 3 содержит сравнение распространенных языков описания архитектурных моделей. В 4-м разделе представлен краткий обзор инструментальных средств MASIW, разработанных в ИСП РАН, с целью автоматизации проектирования систем ИМА. Следующие два раздела посвящены отдельным исследовательским задачам, возникшим при разработке MASIW. В разделе 5 рассматриваются задачи анализа архитектурных моделей, а в разделе 6 — задачи синтеза отдельных частей модели. В разделе 7 представлено сравнение инструментальных средств MASIW с другими инструментами, построенными на основе языка описания архитектурных моделей AADL. В заключении обсуждаются перспективы развития методов и инструментов моделирования, синтеза и верификации сложных систем на основе архитектурных моделей.

2. Интегрированная модульная авионика

Долгое время доминирующим подходом к построению бортовых электронных систем воздушных судов была так называемая федеративная архитектура авионики, согласно которой каждая подсистема самолета единолично владела полным набором собственного программного и аппаратного обеспечения,

включающего датчики, исполнительные устройства, управляющие контроллеры и провода их соединяющие. Однако к началу 90-х этот подход начал исчерпывать свои ресурсы, поскольку суммарный вес, энергопотребление и стоимость авионики достигли критических величин (рис. 1) [2], что потребовало внедрения принципиально новых подходов к построению бортовых электронных систем.

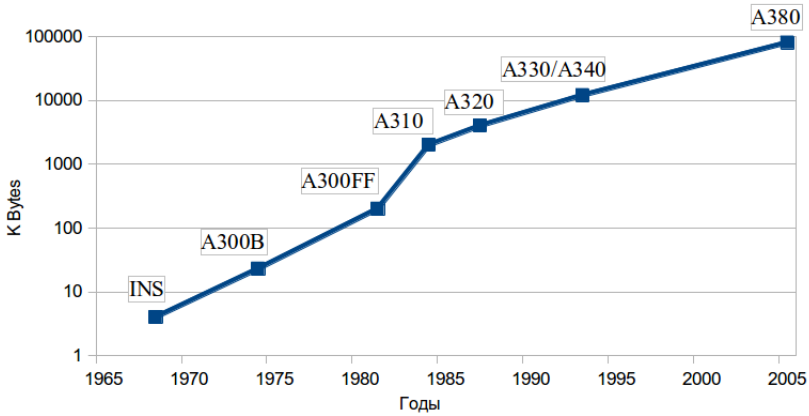


Рисунок 1. Рост размера исходного кода ПО самолетов AIRBUS

В настоящее время основным подходом к проектированию и разработке бортовых систем гражданских воздушных судов является подход интегрированной модульной авионики. Согласно этому подходу, специализированные контроллеры заменяются на процессорные модули общего назначения, на которых обеспечивается независимая работа различных авиационных систем, собственные провода каждой авиационной подсистемы заменяются на виртуальные соединения внутри коммутируемой сетевой инфраструктуры, основанной на таких технологиях, как AFDX (Avionics Full DupleX Switched Ethernet) [3][4][5] и CAN (Controller area network) [6][7]. Это позволяет снизить необоснованное дублирование аппаратного обеспечения, приводящее к неприемлемому уровню энергопотребления и сложности системы бортового оборудования. Но, с другой стороны, такой подход в значительной мере усложняет процесс разработки программного и аппаратного обеспечения, ставит новые задачи в проектировании и интеграции программного и аппаратного обеспечения.

При внедрении подхода ИМА в комплексе бортового оборудования воздушного судна появляется новая подсистема, предоставляющая аппаратную платформу для работы программного обеспечения других бортовых систем. Эта подсистема получила название платформы ИМА и кодовое обозначение ATA-42. Команда, ответственная за проектирование, конфигурирование и верификацию платформы ИМА, обычно называется

Группой системной интеграции, поскольку её задачей является не только разработка отдельно стоящей подсистемы, но и согласование потребностей всех пользователей платформы и конечная интеграция всего комплекса программных и аппаратных компонентов, работающих с использованием платформы ИМА.

В число задач Группы системной интеграции также входит:

- уточнение/согласование требований/потребностей с разработчиками программного и аппаратного обеспечения;
- проектирование платформы ИМА исходя из потребностей функциональных приложений в аппаратных ресурсах, в том числе:
  a. распределение функциональных приложений по вычислительными модулям (Core Processing Module – CPM) с учетом потребностей приложений (количество процессорного времени, распределение процессорного времени между строго периодическими приложениями, объем памяти ОЗУ/ПЗУ, пропускная способность сетевых интерфейсов и т. п.);
  b. определение состава сетевых компонентов (топологии сети) с учетом требований надежности, времени доставки сообщений от отправителя к получателю и т. п.
- проверка разрабатываемого комплекса бортового оборудования (КБО) на соответствие требованиям, изложенных в проектной документации к самолету, КБО и его отдельным компонентам;
- подготовка конфигурационных таблиц для компонентов платформы ИМА.

Для решения этих задач требуется точное понимание всех деталей разрабатываемого комплекса как на высоком, так и на низком уровне детализации, а также предельная внимательность при анализе последствий, в случае внесения изменений. При этом размер КБО современных воздушных судов и количество существенных деталей таково, что их невозможно удержать в голове одного человека. В таких условиях применение специалистами традиционных способов разработки, основанных на аккуратном описании всех требований, архитектурных решений и т. п. в текстовых документах, становится чрезмерно трудоемким и подверженным ошибкам. Возможность подключить к разрешению этих проблем программные средства автоматизации наталкивается на разнородность и неструктурированность информации. Естественным шагом по преодолению этой проблемы является формализация информации, переведение её в унифицированный машино-читаемый вид, что позволяет автоматизировать её обработку.

Рисунок 2. Место ранней валидации в процессе проектирования и разработки платформы ИМА

В контексте проектирования комплексных программно-аппаратных систем, таких как платформа ИМА, основным стержнем является архитектура комплекса, вокруг которой выстраиваются требования к системе в целом и к её отдельным компонентам, проектные компромиссы, работы по анализу и верификации и т. д. Поэтому неудивительно, что именно архитектурные модели, описывающие компоненты системы и связи между ними, становятся основой для формирования новых технологий и инструментов автоматизации проектирования. Они позволяют описывать различные аспекты архитектуры в единой формализованной модели, которая может обрабатываться различными инструментами с целью проверки внутренней согласованности архитектуры, удовлетворения системой разнообразных требований к ней, автоматизации принятия проектных решений, генерации конфигурационных данных/файлов, исходных текстов и т. п. При этом инструменты анализа модели могут быть применимы на разных уровнях абстракции, в том числе на самых ранних стадиях проекта в условиях наличия только частичной и оценочной информации. Среди специалистов данная практика получила название «ранняя валидация» («Early Validation»), а наборы соответствующих инструментов – «инструменты ранней валидации» («Early Validation Tools») [8].

Место применения таких инструментов в процессе проектирования и разработки платформы ИМА показано на рис. 2. Использование архитектурных моделей в этой области позволяет решать следующие задачи:

1. Проверка ограничений/требований предъявляемых к компонентам разрабатываемого комплекса:
   a) проверка достаточности аппаратных ресурсов, например, что потребности всех функциональных приложений в процессорном времени и объеме памяти соответствуют аппаратным характеристикам вычислительного модуля, на котором данные функциональные приложения будут выполнятся;
   b) проверка временны*х* характеристик взаимодействия функциональных приложений или вычислительных модулей, например, что время доставки сообщения от одного функционального приложения до другого не превышает заданного требованиями;
   c) проверка возможности выделения того или иного аппаратного ресурса в соответствии с определенными ограничениями, например, возможности выделения процессорного времени набору строго периодических задач с учетом того, что каждая задача должна запускаться в определенные моменты времени в соответствии с заданным периодом;
   d) проверка безопасности и устойчивости к отказам отдельных компонентов КБО (safety analysis).
2. Автоматизация распределения аппаратных ресурсов между функциональными приложениями с учетом заданных ограничений, например, распределение функциональных приложений по вычислительным модулям с учетом достаточности пропускной способности сетевых интерфейсов и возможности построения расписания для строго периодических задач.
3. Генерация элементов платформы КБО: конфигурационных данных/файлов, исходных кодов отдельных компонентов платформы и т. п.

3. Языки описания архитектурных моделей

За время исследований в области проектирования программно-аппаратных систем на основе моделей сформировалось несколько подходов к описанию архитектурных моделей. Наиболее широкое распространение получили подходы, основанные на языках AADL [9], EAST-ADL [10] и UML [11]. Язык EAST-ADL в настоящей работе не рассматривается, так как его область применения ограничена автомобильными системами, основанными на архитектурных решениях AUTOSAR. Язык AADL унаследовал основные черты от языка Meta-H [12], разработанного для описания бортовых систем авионики в конце 90-х, а в настоящее время является распространенным

языком описания архитектурных моделей программно-аппаратных систем в различных прикладных областях. Язык UML чаще всего применяется для описания программно-аппаратных систем в виде одного из своих профилей, наиболее востребованными среди которых являются SysML [13] и MARTE [14]. Ниже представлены основные особенности данных языков [15].

UML	**AADL**
Нотации	
• предоставляет набор диаграмм для представления структуры программного обеспечения; при этом отдельные диаграммы, описывающие те или иные компоненты программно-аппаратного комплекса, не могут быть полностью связаны друг с другом, т. е. объединение моделей разработанных разными группами разработчиков крайне затруднительно	• разработан более в традициях языков программирования, нежели описания диаграмм; он оперирует декларациями типов и реализаций компонентов модели, которые могут быть переиспользованы в декларациях других компонентов.
Расширяемость	
• может быть расширен путем использования следующих механизмов: ○ стереотипов (stereotypes), которые позволяют расширять словарь UML для создания новых элементов моделирования; ○ картами соответствия идентификаторов и значений (tagged values); ○ переопределение модельных элементов при помощи дополнительных ограничений (constraints); ○ эти механизмы обычно используются тем или иным профилем, который представляет из себя диалект описания моделей (например, SysML и MARTE).	• может быть расширен путем определения: ○ определяемых пользователем наборов свойств, которые могут добавлять новые типы и определения свойств или расширять уже имеющиеся типы и свойства; ○ Annex-спецификаций, которые позволяют описывать дополнительные характеристики элементов модели в произвольном синтаксисе и с произвольной семантикой, которая обрабатываются специализированными инструментами;

Аспекты моделирования	
• используется в основном для описания структуры программного обеспечения; при этом базируется на трех аспектах: данные, взаимодействие и состояния; данные описываются диаграммами классов, взаимодействие описывается диаграммами соединений или диаграммами последовательностей, состояния описываются диаграммами состояний. Наиболее используемые профили SysML и MARTE расширяют UML следующим образом: ○ SysML добавляет два вида диаграмм – диаграмму требований и параметрическую диаграмму; диаграмма требований используется для описаний требований и связи требований с элементами модели; параметрическая диаграмма используется для описания взаимосвязей компонентов модели программного обеспечения с компонентами модели аппаратного обеспечения. ○ MARTE расширяет UML посредством введения следующих стереотипов: модель программного обеспечения, модель аппаратного обеспечения, взаимосвязь моделей программного и аппаратного обеспечения.	• используется для описания «архитектуры исполнения». «Архитектура исполнения» неявно подразделяется на две части: набор программных компонентов и взаимодействие между ними, набор компонентов аппаратного обеспечения и взаимодействие между ними; также описывается взаимосвязи между программными компонентами и компонентами аппаратного обеспечения.

Исходя из вышеперечисленного, можно сделать вывод, что и UML (в виде профилей SysML и MARTE), и AADL предоставляют примерно одинаковые возможности по описанию программно-аппаратной модели КБО. В то же время AADL обладает рядом преимуществ:

- AADL помимо графической нотации имеет текстовое представление, которое позволит специалисту создавать и редактировать модели, а так же анализировать семантику существующих моделей без наличия

специализированных редакторов, в то время как «чтение» моделей на базе UML без использования специальных редакторов диаграмм может стать неразрешимой задачей;

- AADL ограничивает разработчика конкретным набором типов деклараций (типов элементов модели), обладающих определенной семантикой, которые разработчик может использовать для описания модели программно-аппаратного комплекса, что позволяет без дополнительных затрат переиспользовать существующие модели, разработанные независимыми командами. В то же время UML из-за своей универсальности не накладывает строгих ограничений на типы и семантику используемых элементов, что затрудняет понимание моделей, разработанных сторонними специалистами.

4. MASIW – рабочее место системного интегратора

Ввиду рассмотренных выше особенностей язык AADL был выбран в качестве формализма для описания архитектурных моделей в рамках исследований в области автоматизации проектирования программно-аппаратных систем. Одновременно с этими исследованиями в сотрудничестве с коллективом ГосНИИАС ведется разработка инструментальных средств поддержки проектирования ИМА, которые получили название MASIW (Modular Avionics System Integrator Workplace).

В рамках исследований преследуется двуединая цель, состоящая из исследовательской составляющей – развития методов моделирования и верификации комплексных программно-аппаратных систем, и инженерной составляющей – разработки рабочего инструментария для проектировщиков и интеграторов систем авионики.

Основные принципы, на которых построены исследования и инструменты, заключаются в следующем:

- открытость – как необходимое условие сотрудничества с международным исследовательским сообществом;
- опора на международные стандарты;
- сочетание математической строгости в выборе предложенных решений и обеспечении доступности этих решений для инженеров;
- нацеленность на поддержку и интеграцию различных процессов жизненного цикла работки систем: определения и анализа требований, проектирования, интеграции и верификации программных и программно-аппаратных систем.

На текущий момент разработанные инструментальные средства MASIW позволяют решать следующие задачи.

1. Создание, редактирование и управление моделями на языке AADL:
   a) создание/редактирование моделей посредством текстового или графического редактора;

b) поддержка командной разработки с возможностью отслеживания и внесения изменений для отдельных элементов модели;
c) поддержка переиспользования AADL моделей сторонних разработчиков.

2. Анализ моделей:
   a) анализ структуры программно-аппаратного комплекса – достаточности аппаратных ресурсов, согласованности интерфейсов и т. п.;
   b) анализ характеристик передачи данных в сети AFDX – времени доставки сообщений от отправителя к получателю, глубины очередей передающих портов и т. п.;
   c) симуляцию модели программно-аппаратного комплекса с генерацией пользовательских отчетов по результатам работы симулятора.
3. Синтез моделей:
   a) распределение функциональных приложений по вычислительным модулям с учетом ограничений ресурсов аппаратной платформы и с учетом дополнительных ограничений касающихся вопросов надежности и безопасности программно-аппаратного комплекса;
   b) генерация распределения вычислительного времени процессора между функциональными приложениями (циклограмма расписания запуска приложений для ARINC-653 совместимых операционных систем реального времени);
4. Генерация исходного кода/конфигурационных данных:
   a) разработка специализированных инструментов генерации кода/конфигурационных данных, на основе предоставляемого программного интерфейса (API);
   b) генерация конфигурационных файлов для ОС РВ VxWorks653 и оконечных устройств сети AFDX.

Создание, редактирование и управление моделями, а также генерация кода и конфигурационных данных реализованы с использованием широко распространенных расширений среды Eclipse, такими как Eclipse Modeling Framework, Graphical Editing Framework, Eclipse Team Providing, SVN Team Provider, GIT Team Provider. При реализации этих возможностей в основном приходилось решать инженерные задачи, поэтому в последующих разделах мы более подробно остановимся на реализациях поддержки анализа и синтеза моделей, где были сконцентрированы основные исследовательские задачи.

5. Анализ моделей

Когда заходит речь об анализе моделей, то под этим понимают выведение новых свойств модели в результате рассуждений об ее уже известных свойствах. Например, результатом анализа может быть оценка наибольшего времени между отправкой сообщения и его доставкой на основании анализа

пути следования сообщения и характеристик компонентов, встречающихся на этом пути. Наиболее важным видом анализа модели является ее верификация, то есть проверка выполнения моделью требований, которые к ней предъявляются. Другие виды анализа, как правило, используются в качестве промежуточного шага в процессе верификации.

Требования к архитектуре КБО возникают из разнообразных источников.

- Это могут быть проектные требования к самолёту и архитектуре КБО — эти требования в процессе анализа уточняются и декомпозируются на требования к отдельным компонентам системы.
- Часто в рамках определенного проекта регламентируются требования на оформление и организацию архитектурных моделей, которые описываются в так называемом стандарте на проектирование моделей.
- Еще одним источником требований являются ограничения на область допустимого использования или на допустимые конфигурации моделируемых компонентов (usage domain rules).
- Автор библиотечного модельного компонента может накладывать требования по консистентному использованию этого компонента.
- Также встречаются требования, накладываемые инструментами или средствами анализа моделей и необходимые для возможности проведения соответствующего анализа.

Так как при моделировании системы есть потребность обнаруживать ошибки как можно раньше, то стоит задача анализа модели, в которой есть недоспецифицированные компоненты или компоненты с еще неизвестной структурой. Иногда в таких случаях для проведения некоторого вида анализа достаточно предположений о неразработанном компоненте. Например, в системе есть процесс A с неизвестной реализацией. Однако, предполагается или известно, что в среднем он раз в 100 мс генерирует пакет данных размером в среднем 100 байт, предназначенный процессу B. При этом компоненты, обеспечивающие сетевое взаимодействие, описаны в модели достаточно детально. Тогда такую неполную модель можно анализировать в аспекте сетевого взаимодействия, задержек при связи процессов, заполненности буферов сетевых компонентов и пр.

5.1. Типы анализа моделей

Типы анализа различаются по *способу* его выполнения (**статический** или **динамический**) и по *аспектам* исследуемого объекта (самое грубое деление это аспекты **структуры** или архитектуры системы и аспекты **поведения**, функционирования системы).

Динамический анализ подразумевает, что явным образом заданы некоторые закономерности, по которым происходит изменение модели (внутреннего состояния компонентов, связей между ними) и взаимодействие с окружающей средой. Во время проведения такого анализа происходит выполнение

действий по заданным закономерностям, получение новых состояний компонентов, новых связей, обеспечение взаимодействия с окружающей средой. В дальнейшем (в зависимости от проверяемых требований) происходит анализ полученного состояния или последовательности состояний и, например, оценка их корректности.

При **статическом анализе** используется математическое описание компонентов модели, которое сопоставляется с описанием требований на них. В процессе анализа производится сопоставление требований, вычисление характеристик компонентов, по которым в дальнейшем делаются выводы о корректности или некорректности анализируемой модели.

При анализе **поведения** модели рассматриваются характеристики, возникающие только при рассмотрении того, как себя ведут компоненты во времени, как они взаимодействуют с окружающим миром, на какие события и как реагируют, какие события и данные генерируют, как изменяют своё внутреннее состояние. При анализе **структуры** рассматриваются характеристики того, как соединены компоненты, какими свойствами обладают эти связи, какие есть возможности и направления передачи данных и событий, у каких компонентов есть доступ до тех или иных ресурсов и пр.

Способы и аспекты моделирования могут сочетаться произвольным образом, далее, в разделах 5.3-5.6, рассматриваются все четыре возможные комбинации.

5.2. Входные данные для анализа

Входными данными для анализа является модель программно-аппаратного комплекса, которая описывает структуру и характеристики/свойства элементов комплекса. Рассмотренные ранее языки описания моделей (UML и AADL) позволяют детально описывать структуру разрабатываемого комплекса, вплоть до описания каждого датчика, кнопки и т.п. Как показала практика, такая модель является избыточной для большинства видов анализа. Значительная часть модели игнорируется специализированными инструментами анализа, и в результате инструменту приходится выполнять лишнюю работу по выборке существенных данных из модели комплекса. Кроме того, языки описания моделей предоставляют разработчику моделей определенную свободу в выборе того, какими сущностями будут описаны те или иные компоненты разрабатываемого комплекса (UML в большей степени, AADL – в меньшей). В то же время при разработке инструмента анализа можно точно определить структуру входных данных, которая не зависит от того, какими конкретными сущностями описана модель программно-аппаратного комплекса. Поэтому при разработке инструмента MASIW была предложена концепция так называемых представлений. **Представление** (англ. view) – специализированная модель всего программно-аппаратного комплекса или определенной его части, которая представляет набор существенных данных в виде удобном для обработки, как это происходит с представлениями

в системах управления базами данных. Для создания специализированного представления применяется набор адаптеров – правил, по которым исходная модель трансформируется в специализированную и, при необходимости, наоборот. Такой подход позволяет абстрагироваться от того, каким образом разработчик опишет модель комплекса или его части (какие сущности будут применены и даже какой язык описания моделей будет использован). Благодаря этому разрабатываемые инструменты анализа могут быть переиспользованы в других инструментах работы с моделями.

5.3. Статический структурный анализ

Структуру модели можно понимать как граф, узлами которого являются компоненты модели, а дуги - связи между компонентами. Виды связей могут отличаться в разных языках моделирования. Например, связь между двумя компонентами может означать, что один компонент является частью другого, или что один компонент является аппаратным ресурсом, на котором выполняется другой, программный компонент.

Структура модели может содержать информацию о составе компонентов модели (из каких компонентов-частей он состоит), о размещении компонентов, о степени связности компонентов и т.п. Зачастую языки моделирования позволяют сопоставить компонентам модели атрибуты, т.е. некоторые скалярные значения. В этом случае они также становятся частью структуры модели и могут быть использованы в структурном анализе. Например, можно проанализировать модель на предмет того, *принадлежат ли значения атрибута с некоторым заданным именем во всех компонентах, у которых он определен, некоторому заданному множеству допустимых значений*.

Одним из возможных атрибутов компонентов может быть тип компонента. Примерами использования типа в структурном анализе могут быть следующие задачи: *узнать, содержат ли все компоненты некоторого типа атрибут с некоторым заданным именем*. Или другой пример: *выяснить, все ли компоненты типа A являются частью какого-либо из компонентов типа B*.

В тех языках моделирования, которые нацелены на описание структуры модели (в отличие от нацеленных на поведение), структурный анализ модели удобнее, чем поведенческий. Причина этого в том, что при использовании таких языков моделирования структура модели уже может присутствовать, в то время как поведение еще до конца не определено или не описано. Как следствие, ряд анализов можно провести заранее, до более трудоемкой операции определения поведенческой составляющей модели.

Для организации и автоматизации статического структурного анализа необходимо решать следующие задачи:

1. как задать то, *что* надо проанализировать (в частности, какое условие корректности структуры модели необходимо проверить и какой язык

следует выбрать для описания этого условия);

2. как задать *контекст* анализа (для какой части структуры модели необходимо проводить анализ) - чаще всего, анализ следует проводить на всей модели, хотя вполне возможно, например, что анализ надо проводить лишь на компонентах, являющихся частью данного компонента модели, или на компонентах лишь некоторого заданного типа;
3. как *выполнять* заданный анализ для заданной части модели;
4. в каком *виде* представить разработчику результат анализа.

Все эти задачи возникли и при разработке инструмента статической проверки правильности структуры моделей в среде MASIW. В этой среде в качестве языка моделирования используется AADL. Структура модели в этом языке является иерархической по отношению включения одних компонентов в другие. Также модель на языке AADL содержит иерархию типов, описывающих классы компонентов. Среда MASIW строит экземпляр модели, генерируя экземпляры всех компонентов и соединяя эти компоненты так, как того требует семантика языка AADL с учетом всех наследуемых и переопределяемых атрибутов. В результате инструменты анализа работают с уже подготовленным экземпляром модели и для них нет необходимости знать о сложностях трансформации декларативной AADL модели в экземпляр архитектурной модели программно-аппаратного комплекса.

В соответствии с контекстом проведения анализа обычно выделяют глобальные условия корректности и компонентные условия корректности. Глобальные условия корректности представляют ограничения на модель в целом. Компонентные условия корректности описывают ограничения на определенные виды компонентов. Иногда про компонентные условия говорят как об “инвариантных свойствах компонентов”.

Чтобы иметь возможность автоматически проводить анализ модели на выполнение требований, требования должны быть формализованы и описаны в машинно-читаемом виде. Для этого необходимо иметь язык для записи формализованных требований. В настоящее время в MASIW поддерживается описание и проверка условий корректности архитектурной модели на языке REAL (Requirements Enforcement Analysis Language) [16].

Язык REAL был предложен в 2010 году в Telecom ParisTech (Франция) и был поддержан сразу несколькими исследовательскими лабораториями по всему миру. Этот язык основан на аппарате теории множеств. Автор языка пытался сделать язык как можно более удобным для инженеров, занимающихся моделированием и владеющих базовыми навыками императивного программирования. Однако этот язык оказался непригодным для практического использования ввиду своей ограниченной функциональности, а в открытой печати так и не появилась качественная документация по этому языку.

После этого в Rockwell Collins был разработан язык Lute [17] и инструмент по проверке моделей на выполнение требований на Lute. Язык Lute является наследником языка REAL, незначительно расширяя его синтаксис. Кроме того, вместе с инструментом поставляется библиотека теорем, разработанная в Rockwell Collins в рамках проекта META. Однако качественная документация по языку Lute и библиотеке теорем не доступна, что не дает возможности оценить выразительные возможности этого языка.

Как показала практика язык REAL содержит ряд серьезных недостатков. Во-первых, он поддерживает не все типы данных и компонентов языка AADL (в частности, он совсем не поддерживает значения с единицами измерения). Во-вторых, он не поддерживает компонентные условия корректности. В-третьих, в языке отсутствуют средства переиспользования части одних условий в других (что особенно актуально, если проверка нескольких условий требует одних и тех же вычислений). Кроме того, не все условия корректности удобно представлять в императивном виде, а язык REAL не содержит средств для неимперативного описания условий корректности.

Для частичного решения этих проблем мы внесли ряд изменений в язык REAL. Более того, мы поставили цель сделать статический анализ не столько средством демонстрации правильности модели на AADL, сколько средством поиска ошибок в модели. Для этого мы, формально не меняя язык, предложили способ аннотирования текста утверждения на REAL с целью документирования его семантики. Эта документация используется нашим инструментом при построении отчёта о проведенной верификации, чтобы продемонстрировать, какие условия корректности проверялись, на каких компонентах были проверены условия корректности, каков статус проверки на компонентах (на каких компонентах обнаружено нарушение условий корректности, а на каких - нет), причина нарушения (если она была задана в комментариях-аннотациях).

5.4. Статический поведенческий анализ

Целью статического анализа поведения модели системы является получение математическими методами оценок предельных величин различных характеристик поведения и взаимодействия компонентов системы.

Одной из важнейших поведенческих характеристик является время реакции системы на поступающие извне события. На время реакции влияет как скорость обработки событий, так и время доставки информации между компонентами. В основе архитектуры ИМА лежит идея разделения аппаратных ресурсов между множеством авиационных функций с обеспечением отсутствия непреднамеренного влияния одной функции на другую. Первым шагом в этом направлении было разделение вычислительных ресурсов. Следующим шагом стала виртуализация шин данных, которая начиная с Airbus-380 базируется на технологии AFDX [5].

AFDX построена на основе обычного Ethternet, но доработана для обеспечения детерминированности, устойчивости, безопасности, надежности, необходимых для удовлетворения требований сертификации. Ключевым элементом AFDX в этом плане является понятие виртуального канала (Virtual Link), которое по сути представляет собой виртуальный провод, эквивалентный физическому проводу между отправителем и получателем сообщений. Виртуальность проводов сокращает вес, энергопотребление, сложность прокладки и, что самое главное, стоимость сопровождения и развития сети, так как прокладка физического кабеля заменяется на модификацию конфигурационных таблиц в коммутаторах.

Собственно компоненты, генерирующие и получающие сообщения, могут находится вне сети AFDX. Это либо специализированные управляющие функции, располагающиеся в датчиках и актуаторах, либо приложения, выполняющиеся на вычислительных модулях. Во всех случаях эти компоненты соединены с сетью AFDX посредством одного или нескольких промежуточных шлюзов, сообщение с которыми может происходить по нескольким разным протоколам передачи данных.

Анализ передачи данных в сети AFDX

Технология AFDX построена на основе полнодуплескного коммутируемого Ethernet, поэтому конфликтов и задержек на линиях при передаче данных не возникает. Общее время доставки пакета равно сумме времен передачи пакета на линиях плюс задержки в коммутаторах.

На выходных портах коммутаторов установлены очереди. Таким образом, задержка в коммутаторах может быть очень изменчивой из-за слияния различных виртуальных каналов, конкурирующих за каждый выходной порт. Следовательно, для определения верхней границы общего времени передачи пакета необходимо анализировать задержки в каждом выходном порту коммутатора.

Еще одной важной поведенческой характеристикой сети AFDX является гарантия того, что в каждой очереди имеется достаточно места для хранения всех прибывающих пакетов. На практике для каждой очереди в сети оценивается верхняя граница количества данных в этой очереди.

Существует несколько аналитических методов для вычисления оценки верхних границ времени доставки пакета и размеров очередей: Model Checking, Trajectory, Network Calculus. Каждый имеет свои преимущества и недостатки. Главным недостатком большинства методов является так называемый пессимизм — то есть получение заведомо большей оценки из-за некоторых предположений или грубых вычислений. Кроме того, существенным недостатком метода Model Checking является то, что при увеличении размера сети он очень быстро приводит к комбинаторному взрыву. Для промышленного использования подходят лишь методы Trajectory и Network Calculus.

Метод Trajectory

Метод Trajectory [18] базируется на анализе сценария худшего случая, который может произойти с пакетом на его траектории. *Интервалом занятости* для пакета f в выходном порту p называется временной интервал, в течение которого f может обрабатываться в p. Метод Trajectory предполагает наибольший интервал занятости в каждом порту. Для каждого канала-конкурента оценивается максимальное количество его пакетов, которые могут задержать отправку пакета f из порта p во время интервала занятости.

При вычислении верхней границы времени доставки пакета для данного виртуального канала, предполагается, что пакет f становится на выходном порту p в очередь, в которой уже стоят по максимальному количеству пакетов всех остальных виртуальных каналов, которые могут задержать отправку пакета f.

Оценка верхних границ размеров очередей методом Trajectory производится так. Необходимо для каждого выходного порта найти максимум из размеров очередей среди интервалов занятости всех виртуальных каналов, отправляемых через этот порт. Это значение и будет верхней границей размера очереди данного порта.

Метод Network Calculus

В методе Network Calculus [19] поток информации через определенный узел сети представляется функцией потока. *Функцией потока* R(t) от времени называется функция, значением которой в момент t является суммарное количество бит, вошедшее в этот узел начиная с момента $t_0=0$.

Поскольку на характер функции потока в данном узле, вообще говоря, влияют много разных факторов, точное ее определение довольно затруднительно. Для анализа потоков информации в методе Network Calculus используются так называемые входящие кривые (arrival curve), которые мажорируют функцию потока “равномерно” начиная с любого момента времени: *входящая кривая* $\alpha(t)$ потока R(t) — это такая неубывающая функция, что для любых $0 \le s \le t$ верно неравенство $R(t) - R(s) \le \alpha(t-s)$.

Если поток информации представляет из себя периодическую последовательность пакетов ограниченной длины, функция R(t) является “ступенчатой” функцией. Входящей кривой для функции такого рода является функция $\gamma_{r,b}(t) = rt + b$, где r – это средняя скорость потока, b – максимальный размер пакета.

Кроме входящей кривой, которая является “представителем” функции потока, в методе Network Calculus в каждом узле сети рассматривается *обслуживающая кривая* (service curve), которая описывает количество обработанной информации в узле к данному моменту времени.

Рассмотрим ситуацию, при которой узел обрабатывает информацию с постоянной скоростью R (как правило, эта скорость соответствует пропускной

способности линии связи на выходе из узла), однако перед обработкой вносит некоторую задержку, ограниченную временем T (как правило, это время соответствует максимальной технологической задержке по доставке информации внутри узла от входа к выходу). В таком случае обслуживающая кривая в этом узле равна $\beta_{R,T}(t) = \max\{ 0, R(t-T) \}$.

Максимальная задержка, которую информация из потока с входящей кривой α получает в узле, который обеспечивает обслуживающую кривую σ, оценивается сверху величиной максимального горизонтального (т. е. по оси «время») отклонения от α до σ. Для случая $\alpha = \gamma_{r,b}$ и $\sigma = \beta_{R,T}$ размер максимальной задержки равен $T + b/R$.

Максимальное количество необработанной информации из потока с входящей кривой α в узле, который обеспечивает обслуживающую кривую σ, оценивается сверху величиной максимального вертикального (т. е. по оси «информация») отклонения от α до σ. Для случая $\alpha = \gamma_{r,b}$ и $\sigma = \beta_{R,T}$ максимальное количество необработанной информации равно $\gamma_{r,b}(T) = b + rT$.

Использование огрубления в виде входящих кривых вместо функций потока приводит к тому, что по мере прохождения пакета по узлам сети входящая кривая становится все более «грубой». В частности, для случая, когда в узле входящая кривая для входного потока равна $\alpha = \gamma_{r,b}$, а обслуживающая кривая равна $\sigma = \beta_{R,T}$, входящая кривая для выходного потока (она же будет являться входящей кривой для входного потока следующего узла на пути данного пакета) равна $\gamma_{r,b+rT}$, т.е. по мере прохождения пакета узлам сети происходит рост второго параметра функции γ, так что на последующих узлах соответственно растут оценки времени задержки и количества необработанной информации.

Если сравнивать Network Calculus и Trajectory между собой, то нельзя сказать об однозначном превосходстве одного из методов. Хотя во многих случаях Trajectory дает более точные оценки наихудшего времени, чем Network Calculus, существуют такие конфигурации сети, на которых наблюдается обратная ситуация.

5.5. Динамический поведенческий анализ

Динамический анализ поведения позволяет получать менее пессимистичные оценки поведенческих характеристик, по сравнению со статическим анализом. Помимо этого, иногда применение динамического анализа позволяет провести анализ в тех случаях, когда статически его провести не получается или это требует слишком много ресурсов (слишком большая сложность модели, трудносоставимое и трудноанализируемое математическое описание компонентов, комбинаторный взрыв и пр.). Однако, надо учитывать, что динамический анализ проводится в конкретном выполнении, а не в худшем случае, и требует задания конкретного контекста работы: входных данных и воздействий.

Для проведения динамического поведенческого анализа требуется, чтобы в модели некоторым исполнимым образом было задано поведение компонентов. Для корректного использования исполнимых моделей важно организовать работу с модельным временем, с передачей информации и событий внутри модели.

Эта задача хорошо решается при использовании дискретно-событийного подхода к моделированию поведений [20]. В этом подходе работа компонента представляется как набор “событий” – актов действий по изменению состояния объектов и взаимодействию с другими компонентами и внешним миром в определённые моменты времени.

Этот подход зарекомендовал себя как наиболее подходящий при моделировании поведений компьютерных систем [21], какими являются системы КБО ИМА. С одной стороны, он является достаточно мощным для моделирования таких систем, с другой стороны, такой подход гораздо более легко применим (в отличие от ещё более мощного подхода непрерывной симуляции [22], сталкивающегося с решением нелинейных дифференциальных уравнений и используемый при моделировании физических процессов).

Для поддержки дискретно-событийной парадигмы описания поведения, в инструменте MASIW была реализована библиотека, поддерживающая симуляционное время и обеспечивающая обработку событий, возникающих с моделируемой системе, и синхронную и асинхронную передачу данных. Для обеспечения этих возможностей был использован подход в стиле продолжений (continuations programming) [23], для поддержки которого была использована библиотека Matthias Mann’s continuations [24].

Язык AADL имеет собственные средства описания поведенческой семантики для некоторых элементов модели. Однако этих средств недостаточно для описания поведения приложений, устройств и других сложных компонентов модели. Существуют стандартизованные расширения AADL — Behavioral Model Annex [25] и BLESS [26], — позволяющие описывать поведение компонентов модели на основе конечных автоматов, работающих со временем и событиями.

На данный момент в инструменте MASIW исполнимая модель поведений задаётся на языке Java. Это позволило довольно быстро реализовать поддержку динамического поведенческого анализа AADL-моделей и в данный момент инструмент уже можно использовать для такого анализа. Однако приходится задавать поведения компонентов нестандартным способом. С другой стороны, использование Java позволит в будущем реализовать трансляторы со стандартных языков задания поведения без необходимости переделывания части, отвечающей собственно за динамический анализ.

Как было сказано ранее, часто целесообразно проводить ранний анализ системы на неполных моделях. В таких моделях некоторые компоненты ещё не имеют детального описания, а часто есть только некоторые предположения

о том, как они себя ведут в этой системе. В таком случае анализ можно производить, записав предположения в виде того или иного поведения компонентов. В инструменте MASIW поддерживается специальный способ параметризации модели, упрощающий описание различных предположений о компонентах для проведения различных вариантов анализа модели.

5.6. Динамический структурный анализ

Этот тип анализа требуется для реконфигурируемых систем, то есть для систем, структура которых может меняться во время работы. В общем случае, может существовать зависимость структуры модели от входных воздействий или окружения моделируемой системы, поэтому не всегда можно статически проанализировать выполненность всех требуемых структурных ограничений.

В чистом виде динамический структурный анализ предполагает проверку того, чтобы все достижимые состояния архитектурной модели являются структурно-корректными, то есть удовлетворяют требованиям на корректность структуры модели. Для проверки свойств структуры модели, полученной в какой-либо момент выполнения модели, требуется получать изменённую структуру и запускать проверки над новой моделью. Такая проверка подобна статической структурной проверке, на вход которой подаётся модель, полученная в динамике. Сложность здесь состоит в том, что не всегда нужно проверять все свойства во всех состояниях, но требуется некоторым образом задавать, какие проверки в какой момент осуществлять.

Динамический структурный анализ может также проверять свойства, которые задаются не над одним состоянием структуры модели, а над последовательностью таких состояний, хотя такой анализ скорее является анализом поведенческого аспекта функциональности по реконфигурированию системы.

Традиционно среди свойств объектов во времени выделяют свойства *безопасности* и свойства *живучести* [27]. Первые требуют, чтобы что-то никогда не случилось, в то время как вторые — чтобы что-то когда-нибудь обязательно случилось. Примером свойства безопасности является требование, чтобы сразу после возникновения события X у заданного компонента A появится подкомпонент B, а свойства живучести — требование о том, что после возникновения события X у заданного компонента A рано или поздно появится подкомпонент B.

Уже рассмотренное требование структурной корректности всех достижимых состояний является простейшим примером свойства безопасности. Динамическая проверка более сложных свойств безопасности может быть реализована по схожим принципам, с учетом того, что часть информации используемой при статическом структурном анализе фиксированного состояния архитектурной модели должна вычисляться на основании свойств предыдущих состояний модели.

Проверка свойств живучести – существенно иная задача. Основной особенностью такого рода свойств является то, что они нарушаются только на бесконечности. Поэтому задача проверки таких свойств не может быть решена чистым динамическим анализом и требует разработки специальных средств.

6. Автоматический синтез моделей

Перед проектировщиком системы ИМА стоит задача построения архитектуры, которая должна удовлетворять требованиям различных видов: по достаточности аппаратных ресурсов, по устойчивости к сбоям, надежности, безопасности системы в целом, ограничениям на максимально допустимое время доставки сообщений между компонентами, требованиям на своевременное выполнение функциями своих действий и т. д.

До определенного уровня решить такую задачу позволяет искусство опытных специалистов, вооруженных кроме того инструментами верификации строящейся архитектурной модели. Однако этот подход имеет ограниченную масштабируемость и высокую субъективность. Средства автоматизации проектирования систем, удовлетворяющих заданным наборам требований и ограничений, могут сделать работу проектировщиков намного эффективнее.

Во многих случаях отдельные части модели могут быть автоматически синтезированы на основании информации, содержащейся в другой («исходной») части модели, которая описывает основные логические связи между компонентами и требования к результирующей архитектуре. При этом, разработка исходной части модели гораздо проще, чем разработка соответствующей синтезируемой части. Кроме того, исходная часть в любом случае должна быть описана в процессе проектирования. Например, на основании исходной информации об имеющемся наборе приложений и их потребностях в аппаратных ресурсах, а также информации об архитектуре и возможностях вычислительных модулей возможно автоматически синтезировать привязку приложений к этим модулям, так чтобы удовлетворить всем требованиям по достаточности ресурсов и возможности построения расписаний.

В среде проектирования MASIW предлагается следующий сценарий работы для разработки модели проектируемой системы. Проектировщик разрабатывает необходимую исходную часть модели, после чего запускает алгоритм автоматического синтеза, который на основании имеющейся информации, содержащейся в исходной части модели, достраивает модель архитектуры новыми частями, которые при необходимости могут быть скорректированы вручную или перегенерированы в случае обновления исходной части модели.

6.1. Автоматический синтез расписаний выполнения строго периодических задач

При разделении аппаратных ресурсов между несколькими приложениями одним из важнейших аспектов является своевременное обеспечение процессорными ресурсами всех выполняющихся в системе задач. Этим аспектом обычно занимается специальная подсистема планирования задач операционной системы, которая выделяет процессорное время функциональным приложениям на основании заранее подготовленного расписания.

В качестве исходных данных в задаче построения расписания для каждой из периодических задач заданы:

- период запуска задачи;
- время выполнения задачи на одном периоде запуска.

Классические алгоритмы планирования периодических задач работают лишь в случае, когда время запуска задачи внутри периода разрешается варьировать на разных периодах ее выполнения. Однако, в настоящее время имеется потребность в составлении таких расписаний, в которых время между соседними запусками одной периодической задачи было бы фиксировано и равнялось бы длине периода. Такое дополнительное требование *строгой периодичности* не позволяет использовать в планировщике классические алгоритмы планирования.

Основной сложностью алгоритма планирования строго периодических задач является поиск стартовых точек для всех задач, так чтобы было возможно построение собственно расписания. Этот поиск является NP-полной задачей.

Предложенный нами алгоритм [28] позволяет сократить поиск, отсекая на ранних стадиях те ветви перебора, которые заведомо ведут к отрицательному результату. Алгоритм перебора основан на теоретико-числовых соображениях. Основной идеей является перебор для каждой следующей рассматриваемой задачи лишь тех точек, которые заведомо не приведут к совпадению стартовых точек данной и никакой из ранее рассмотренных задач. А именно, если уже выбраны стартовые точки r_1, ..., r_k для первых k задач с периодами p_1, ..., p_k соответственно, то при поиске стартовой точки r для очередной задачи с периодом p надо перебирать лишь те точки, которые не удовлетворяют ни одному из сравнений

$$r - r_i \equiv 0 \pmod{\text{НОД}(p_i, p)}$$

для всех i от 1 до k.

Кроме того, используемая нами стратегия поиска стартовых точек подразумевает перебор в первую очередь таких вариантов, которые обеспечивают как можно более длительное непрерывное выполнение первых тиков после запуска каждой задачи.

В целом, такой подход позволяет быстро получать решение задачи построения расписания для строго-периодичных задач.

6.2. Автоматический синтез архитектуры системы ИМА

В качестве исходных данных в задаче синтеза архитектуры ИМА заданы:

- функциональные приложения и логические потоки данных между ними, а также между приложениями и датчиками/актуаторами;
- набор потребностей функциональных приложений к аппаратным ресурсам (память, вычислительная мощность и т.д.);
- набор требований на максимальное время доставки/обработки сообщений в логических потоках данных;
- набор имеющихся аппаратных компонентов (вычислительные модули, коммутаторы и пр.) в совокупности с описанием предоставляемых ими возможностями и ограничениями на область их допустимого использования (usage domain rules).

Требуется автоматически построить архитектуру системы ИМА, включающую в себя:

- состав и связи аппаратных компонентов;
- размещение функций по вычислительных модулям;
- детали организации соединений в AFDX-сети;
- расписание работы прикладных и системных разделов ARINC-653 совместимых операционных систем;

При этом архитектура системы должна удовлетворять всем требованиям по безопасности и производительности.

Задача синтеза разбивается на две крупные подзадачи:

1. размещение приложений по вычислительным модулям так, чтобы на каждом модуле было возможно построение расписания;
2. назначение виртуальных каналов между вычислительными модулями и распределение коммутаторов по виртуальным каналам так, чтобы удовлетворить требованиям по времени доставки сообщений.

Решение первой задачи основано на рассмотрении множества периодов запуска приложений и на применении теоретико-числовых соображений, которые позволяют разбить множества периодов на такие подмножества, что для каждого полученного подмножества гарантированно отыщутся стартовые точки соответствующих приложений.

Решение второй задачи основано на применении генетических алгоритмов. На каждом шаге генетического алгоритма строится популяция, состоящая из N корректных топологий AFDX-сети. Каждая топология новой популяции получается либо в результате небольшой модификации (мутации) некоторой топологии предыдущей популяции, либо в результате скрещивания некоторых

двух топологий предыдущей популяции. При скрещивании результирующая топология получает максимальное количество общих свойств (в смысле соединения компонентов между собой), имеющихся в обеих исходных топологиях.

После построения очередной популяции производится ранжирование входящих в нее топологий с тем расчетом, чтобы для последующего построения выбрать N топологий, наиболее удовлетворяющих требованиям по времени доставки сообщений. Для оценки получившихся в данной топологии времен доставки используются статические методы (Trajectory, Network Calculus), и основная составляющая функции ранжирования выглядит так:

$$\Sigma\, e^{T-\tau},$$

где суммирование ведется по всем каналам, для которых задано ограничение по времени доставки, T — получившееся время доставки для данного канала в данной топологии, τ — заданное максимальное время доставки для данного канала.

7. Сравнение возможностей MASIW с другими инструментами

Язык AADL поддерживается такими инструментами как OSATE (и производными), Ocarina, STOOD/AADL Inspector, а также инструментом MASIW. Возможности этих инструментов представлены в таблице 1.

	MASIW	**OSATE**	**RC_META**	**Adele**	**Ocarina**	**AADL Inspector**
AADL процессор	MASIW		OSATE		Ocarina	STOOD
Наличие графической среды IDE	+	+	+	+	–	+
Текстовый редактор AADL-моделей	+	+	+	+	–	+
Графический редактор AADL-моделей	+	–	∓(1)	+(2)	–	+
Статический структурный анализ AADL-моделей	REAL+	Lute	Lute	–	REAL	Prolog
Статический поведенческий анализ AADL-моделей	см. п. 5.4	–	AGREE(3)	–	–	Cheddar(4)
Динамический структурный анализ AADL-моделей	–	–	–	–	–	–

	MASIW	OSATE	RC_META	Adele	Ocarina	AADL Inspector
Динамический поведенческий анализ	см. п. 5.5	–	–	–	–	∓(5)
Анализ безопасности (Safety Analysis, FMEA)	–	+	–	–	–	–
Возможность переиспользования AADL-библиотек сторонних разработчиков	+	–	–	–	+	+
Специализированные задачи для систем ИМА:						
Генерация кода/конфигурационных файлов	±(6)	±(7)	–	–	±(8)	–
Генерация распределения процессорного времени приложений	+	–	–	–	+	+

(1) RC_META предполагает использование коммерческого инструмента Enterprise Architect для редактирования SysML диаграмм, которые настроены для однозначного представления AADL моделей и автоматически трансформируются в AADL.

(2) Графический редактор Adele поддерживает только старую версию стандарта AADL 1.0.

(3) AGREE (Assume Guarantee Reasoning Environment) позволяет описывать предположения о значениях входных данных компонентов и требования к значениям выходных данных, а также проверять их согласованность [29].

(4) Cheddar — инструмент анализа возможности построения расписаний для различных стратегий планирования [30].

(5) Поддерживается только симуляция распределения процессорного времени при помощи инструмента Marzhin.

(6) Генерация набора конфигурационных файлов для ОС VxWorks653, программного обеспечения промежуточного слоя и оконечных систем сети AFDX.

(7) Проект RAMSES поддерживает генерацию кода на основе подмножества AADL-BA и предварительной трансформации абстрактной архитектурной модели в более детальную [31].

(8) Ocarina поддерживает генерацию кода на языке Си для операционных систем RT-POSIX, Xenomai, RTEMS и на языке Ада для профиля Ravenscar.

Следует отметить, что благодаря плагину, реализующему интеграцию инструмента Ocarina в среду OSATE, в ней появилась поддержка возможностей Ocarina для тех AADL моделей, которые не используют конструкции не поддерживаемые Ocarina.

Основными отличиями инструмента MASIW является то, что, с одной стороны, он представляет удобную интегрированную среду разработки с возможностью создания/редактирования моделей КБО как в текстовом, так и в графическом представлении и базовыми возможностями для проведения статического и динамического анализа, а, с другой стороны, его внутренняя архитектура позволяет расширять его функциональность при помощи компонентов (плагинов к специализированному «рабочему пространству»), созданных сторонними разработчиками.

8. Перспективы развития

На текущий момент времени инструмент MASIW позволяет выполнять лишь часть задач возложенных на группу системной интеграции и в дальнейшем планируется расширять функциональность его функциональность по многим направлениям.

В контексте *статического структурного анализа моделей* основным направлением развития является разработка полнофункционального языка описания ограничений на структуру архитектурной модели удобного для компактного описания как глобальных, так и компонентных ограничений. По нашему мнению, этот язык должен базироваться на одном из хорошо известных существующих языков программирования, чтобы получить возможность переиспользовать уже готовые библиотеки с разнообразной функциональностью и упростить задачу подготовки инженерных кадров. Хорошим претендентом на роль такого языка является язык Python, который за счет концепции декораторов предоставляет возможность сформировать на своей основе специализированный язык, оставаясь в рамках стандартного синтаксиса, что означает возможность использования в неизменном виде для нового языка уже существующего интерпретатора и других инструментов. Другими перспективными направлениями являются развитие библиотек готовых частей кода для их переиспользования при проверке условий корректности и реализация статического структурного анализа реконфигурируемых систем.

В контексте *статического поведенческого анализа моделей* перспективным направлением развития поддерживаемых методов анализа является анализ передачи данных в системе в целом, а не только в рамках сети AFDX. Основную сложность здесь составляет учет особенностей поведения всех

компонентов-шлюзов, располагающихся между отправителем/получателем сообщения и сетью AFDX.

В контексте *динамического поведенческого анализа моделей* основным направлением развития является поддержка стандартных способов задания поведения для компонентов модели (Behavioral Model Annex, BLESS). Другим очень важным направлением развития этого типа анализа является реализация возможности использования симулятора в комплексе со стендом полунатурного моделирования и в комплексе с внешними эмуляторами аппаратных платформ. Это позволит не тратить лишние силы на разработку детальных моделей для уже имеющихся в наличии компонентов системы, которые доступны для использования на стенде или в виртуальной среде, что сокращает общее время и стоимость подготовки к проверке модели.

В направлении *динамического статического анализа моделей* проведены лишь исследовательские работы, поэтому реализация и проведение экспериментов с этим методом анализа является еще одной задачей для будущего развития функциональности инструмента.

В контексте *автоматического синтеза моделей* перспективными направлениями развития являются поддержка новых видов ограничений на синтезируемую модель, исследование методов инкрементального синтеза архитектуры и автоматического обновления модели при изменении исходных требований с учетом ручных модификаций предыдущих синтезированных моделей, а также синтез отказоустойчивых архитектур, которая бы с учетом информации о степени критичности каждой функции обеспечивала бесперебойную работу всей системы при условии возможности отказов отдельных компонентов.

Еще одним направлением для развития инструментов MASIW является генерация документации с описанием архитектуры системы КБО, а также генерация шаблонов проектов и исходного кода функциональных приложений, которые бы уже включали типовые функции, такие как обработка сообщений структура которых уже описана в архитектурной модели.

9. Заключение

Сложность современных авиационных систем и высокие требования к их надёжности приводят к необходимости использования разделяемых ресурсов (архитектура ИМА). При создании систем ИМА разработчики (в частности, системные интеграторы) сталкиваются с рядом задач и проблем, с которыми они не сталкивались ранее. Для решения этих задач приходят на помощь различные средства автоматизации и компьютерной поддержки разработки. Развитие этого направления в первую очередь связано с использованием разнообразных моделей, в том числе, архитектурных моделей программно-аппаратных систем. Соответствующая группа технологий получила название

разработки систем на основе моделей (Model Driven System Engineering - MDSE).

Внедрение в практику технологий MDSE требует серьезных исследований и продуманных инженерных решений. Одним из источников сложности развития и внедрения MDSE является необходимость учета потребностей и предпочтений различных групп специалистов, поскольку модели одновременно используются как исходные данные для синтеза и для верификации, как инструмент дизайна и как средство коммуникации и кооперации большого числа участников разработки. Методам и инструментам решения этих задач собственно и посвящена данная статья. Особое внимание в статье уделяется вопросам интеграции методов формальной спецификации и формального анализа моделей авионики с методами проектирования, реализации и интеграции систем авионики, которые были наработаны в этой отрасли ранее.

Инструмент MASIW, разрабатываемый в ИСП РАН в сотрудничестве с ГосНИИАС, упрощает решение ряда задач, связанных с разработкой авиационных систем. Он позволяет удобно и наглядно создавать и редактировать модели таких систем на языке AADL, а также проводить **анализ** таких моделей на соответствие разнообразным требованиям, связанным как со структурой, так и с поведением модели (вычислять разнообразные временны́е характеристики, прогнозировать поведение моделируемой системы в различных ситуациях, в том числе при нестандартном поведении компонентов и при отказах внутри системы).

Помимо этого, MASIW облегчает проектирование архитектуры за счет реализации ряда алгоритмов **синтеза** моделей. Это позволяет, в частности, распределить задачи по вычислительным блокам так, чтобы каждой задаче было выделено достаточно процессорного времени, и сгенерировать модель бортовой сети и схему распределения ресурсов сети в соответствии с потребностями компонентов системы.

Инструмент MASIW постоянно развивается. Это развитие опирается на тесную кооперацию с заказчиками, потенциальными пользователями и с международным сообществом разработчиков открытых стандартов и открытых инструментов для поддержки разработки, интеграции и верификации ответственных систем на основе использования средств моделирования.

Список литературы

[1] Е.А. Федосов. Авиация в России конца XX - начала XXI веков в статьях и интервью академика, М.: НИЦ ГосНИИАС, 2013.

[2] Gilbert Edelin. Embedded Systems at Thales: the Artemis challenges from an industrial perspective. ARTIST Summer School, 2009.

[3] Aircraft Data Network, Part 1: Systems Concepts and Overview, 2002.

[4] Aircraft Data Network, Part 2: Ethernet Physical and Data Link Layer Specification, 2002.

[5] Aircraft Data Network, Part 7: Deterministic Networks, 2003.
[6] Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signaling.
[7] Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit.
[8] Brett Murphy, Amory Wakefield. Early verification and validation using model-based design. The MathWorks, 2009.
[9] SAE International. “Architecture Analysis & Design Language (AADL)”, SAE International Standards document AS5506B, Nov 2004, Revised Mar 2012.
[10] The ATESST Consortium. East-adl 2.0 specification (November 2010), http://www.atesst.org
[11] ISO/IEC 19505-1:2012 Object Management Group Unified Modeling Language (OMG UML)
[12] Steve Vestal. MetaH Avionics Architecture Description Language. Honeywell Labs, 2001.
[13] Object Management Group (OMG). “Systems Modeling Language SysML”, Version 1.3.
[14] Object Management Group (OMG). “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems.” Version 1.1.
[15] Dionisio de Niz, Diagrams and Languages for Model-Based Software Engineering of Embedded Systems: UML and AADL, SEI, 2007.
[16] O. Gilles, J. Hugues. Expressing and Enforcing User-Defined Constraints of AADL Models, Engineering of Complex Computer Systems (ICECCS), 2010.
[17] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2_Lute
[18] S. Martin and P. Minet. Schedulability analysis of flows scheduled with FIFO: application to the expedited forwarding class. Parallel and Distributed Processing Symposium, 2006.
[19] Jean-Yves Le Boudec, Patrick Thiran. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Lecture Notes in Computer Science, vol. 2050, 2001.
[20] Jerry Banks, John S. Carson II. Discrete-Event System Simulation. Englewood Cliffs: Prentice-Hall, Englewood Cliffs: Prentice-Hall, 1984.
[21] S. Robinson, Discrete-Event Simulation: From the Pioneers to the Present, What Next? Journal of the Operational Research Society, 2005
[22] R.J. Ord-Smith, J.Stephenson, Computer, Simulation of Continuous Systems. Cambridge, 1975.
[23] Haynes, C. T., Friedman, D. P., and Wand, M., Continuations and coroutines. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84. ACM, New York, 1984.
[24] http://www.matthiasmann.de/content/view/24/26/
[25] AS5506/2 SAE Architecture Analysis and Design Language (AADL) Annex Volume 2.
[26] Brian R. Larson, Patrice Chalin, John Hatcliff, BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software, 10.1007/978-3-642-38088-4_19.
[27] L . Lamport. Proving the Correctness of Multiprocess Programs, 1977.
[28] С. Зеленов, Планирование строго периодических задач в системах реального времени, Труды Института системного программирования РАН, т. 20, 2011.
[29] Abdullah Al-Nayeem, Lui Sha, Darren D. Cofer, Steven M. Miller. Pattern-Based Composition and Analysis of Virtually Synchronized Real-Time Distributed Systems.

Proceedings of the Third International Conference on Cyber-Physical Systems – April 2012.
[30] P. Dissaux, F. Singhoff. Stood and Cheddar: AADL as a Pivot Language for Analysing Performances of Real Time Architectures, Proceedings of 4th International Congress ERTS-2008.
[31] F.Cadoret, E.Borde, S.Gardoll and L.Pautet. Design Patterns for Rule-based Refinement of Safety Critical Embedded Systems Models. International Conference on Engineering of Complex Computer Systems (ICECCS'12), Paris (FRANCE), 2012.

Tools for System Design of Integrated Modular Avionics

Buzdalov D.V., Zelenov S.V., Kornykhin E.V., Petrenko A.K., Strakh A.V., Ugnenko A.A., Khoroshilov A.V.
{buzdalov, zelenov, kornevgen, petrenko, strakh, ugnenko, khoroshilov} @ispras.ru

Abstract. Growth of modern avionics systems makes design of such systems impossible without involvement of automation. Nowadays an area of such tools is represented by both proprietary tools developed by the major aircraft manufacturers like Boeing and Airbus, and a number of open or partially open international projects varying in maturity, availability of source code and documentation. All the tools are based on architecture models of a system under design.

The paper considers languages available to describe architecture models of avionics systems and shows that AADL is the most appropriate one because of availability of textual notation and build-in concepts well suited to represent most elements of embedded systems.

Then the paper presents a toolset for design of modern avionics systems developed by ISPRAS in collaboration with GosNIIAS. The toolset named MASIW provides both a generic platform for design and analysis of architecture models and a specialized solution for the particular domain of avionics systems. It supports creation, editing and management of AADL models in both textual and graphical notation. Also MASIW provides various features for analysis and synthesis of AADL models.

Analysis capabilities include

- a checker of static structural constraints such as resource sufficiency, interface consistency, usage domain rules, etc.
- specialized analyzer of AFDX networks aimed to statically estimate latencies, buffer usage, etc.
- a simulator of AADL models augmented with behaviour specification in AADL Behaviour Annex notation or in Java.

Synthesis capabilities include schedule generation for a particular processor module as well as automatic building of assignment of hardware platform resources to software components in accordance with all requirements formalized in the architecture model.

Finally, MASIW provides a framework for generation of configuration data from architecture models. Currently it is used for generation of configuration tables for ARINC-653 operating systems and for AFDX switches and endpoints.

Keywords: Integrated Modular Avionics; IMA; Early Validation; model of platform; AADL.

References

[1]. E.A. Fedosov. Aviacija v Rossii konca XX - nachala XXI vekov v stat'jah i interv'ju akademika [Aviation in Russia in the late XX - early XXI centuries in articles and interviews of the Academician], M.: NIC GosNIIAS, 2013.
[2]. Gilbert Edelin. Embedded Systems at Thales: the Artemis challenges from an industrial perspective. ARTIST Summer School, 2009.
[3]. Aircraft Data Network, Part 1: Systems Concepts and Overview, 2002.
[4]. Aircraft Data Network, Part 2: Ethernet Physical and Data Link Layer Specification, 2002.
[5]. Aircraft Data Network, Part 7: Deterministic Networks, 2003.
[6]. Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signaling.
[7]. Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit.
[8]. Brett Murphy, Amory Wakefield. Early verification and validation using model-based design. The MathWorks, 2009.
[9]. SAE International. "Architecture Analysis & Design Language (AADL)", SAE International Standards document AS5506B, Nov 2004, Revised Mar 2012.
[10]. The ATESST Consortium. East-adl 2.0 specification (November 2010), http://www.atesst.org
[11]. ISO/IEC 19505-1:2012 Object Management Group Unified Modeling Language (OMG UML)
[12]. Steve Vestal. MetaH Avionics Architecture Description Language. Honeywell Labs, 2001.
[13]. Object Management Group (OMG). "Systems Modeling Language SysML", Version 1.3.
[14]. Object Management Group (OMG). "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems." Version 1.1.
[15]. Dionisio de Niz, Diagrams and Languages for Model-Based Software Engineering of Embedded Systems: UML and AADL, SEI, 2007.
[16]. O. Gilles, J. Hugues. Expressing and Enforcing User-Defined Constraints of AADL Models, Engineering of Complex Computer Systems (ICECCS), 2010.
[17]. https://wiki.sei.cmu.edu/aadl/index.php/Osate_2_Lute
[18]. S. Martin and P. Minet. Schedulability analysis of flows scheduled with FIFO: application to the expedited forwarding class. Parallel and Distributed Processing Symposium, 2006.
[19]. Jean-Yves Le Boudec, Patrick Thiran. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Lecture Notes in Computer Science, vol. 2050, 2001.
[20]. Jerry Banks, John S. Carson II. Discrete-Event System Simulation. Englewood Cliffs: Prentice-Hall, Englewood Cliffs: Prentice-Hall, 1984.
[21]. S. Robinson, Discrete-Event Simulation: From the Pioneers to the Present, What Next? Journal of the Operational Research Society, 2005

[22]. R.J. Ord-Smith, J.Stephenson, Computer, Simulation of Continuous Systems. Cambridge, 1975.
[23]. Haynes, C. T., Friedman, D. P., and Wand, M., Continuations and coroutines. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84. ACM, New York, 1984.
[24]. http://www.matthiasmann.de/content/view/24/26/
[25]. AS5506/2 SAE Architecture Analysis and Design Language (AADL) Annex Volume 2.
[26]. Brian R. Larson, Patrice Chalin, John Hatcliff, BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software, 10.1007/978-3-642-38088-4_19.
[27]. L. Lamport. Proving the Correctness of Multiprocess Programs, 1977.
[28]. S. Zelenov. Planirovanie strogo periodicheskih zadach v sistemah real'nogo vremeni [Scheduling of Strictly Periodic Tasks in Real-Time Systems]. Trudy ISP RAN [The Proceedings of ISP RAS], t. 20, 2011.
[29]. Abdullah Al-Nayeem, Lui Sha, Darren D. Cofer, Steven M. Miller. Pattern-Based Composition and Analysis of Virtually Synchronized Real-Time Distributed Systems. Proceedings of the Third International Conference on Cyber-Physical Systems – April 2012.
[30]. P. Dissaux, F. Singhoff. Stood and Cheddar: AADL as a Pivot Language for Analysing Performances of Real Time Architectures, Proceedings of 4th International Congress ERTS-2008.
[31]. F.Cadoret, E.Borde, S.Gardoll and L.Pautet. Design Patterns for Rule-based Refinement of Safety Critical Embedded Systems Models. International Conference on Engineering of Complex Computer Systems (ICECCS'12), Paris (FRANCE), 2012.

Статический анализатор Svace для поиска дефектов в исходном коде программ

В.П. Иванников, А.А. Белеванцев, А.Е. Бородин,
В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов
{ivan, abel, alexey.borodin, valery.ignatyev, zhur, arut, maksim.leonov}@ispras.ru

Аннотация. В работе описывается разрабатываемый в ИСП РАН инструмент автоматического статического анализа Svace. Инструмент позволяет находить ошибки и потенциальные уязвимости в исходном коде программ на языках Си/Си++. Особенностью инструмента являются простота использования, широкий набор поддерживаемых типов предупреждений, масштабируемость до программ в миллионы строк кода и приемлемое качество анализа (30-80% истинных предупреждений).

Ключевые слова: статический анализ; анализ потока данных; уязвимости; межпроцедурный анализ; анализ на основе аннотаций.

1. Введение

Высокая сложность программ делает практически невозможным создание программного продукта без дефектов. Причём с увеличением размера программного обеспечения возрастает не только количество дефектов, но и их плотность [1]. Поэтому растет необходимость в инструментах и методах поиска дефектов. Одним из таких методов является статический анализ текстов программ. Анализ программ осуществляется без их реального выполнения. При этом происходит исследование всего кода программы, в том числе редко достигаемых участков кода, что позволяет найти ошибки, которые сложно воспроизвести, и которые обычно остаются незамеченными в ходе тестирования.

В статье описывается инструмент статического анализа Svace, разрабатываемый в ИСП РАН для анализа программ, написанных на языках C и C++. В настоящее время ведется добавление поддержки анализа программ, написанных на языке Java. Более ранние описания инструмента можно найти в статьях [2] и [3].

2. Постановка задачи

При разработке инструмента учитываются следующие основные требования:

- От пользователя требуются минимальные действия для интеграции Svace в систему сборки программы. Не нужно как-либо менять либо аннотировать исходный код программы, либо вмешиваться в процессе анализа.
- Выполняется глубокий межпроцедурный анализ, т.е учитывается влияние разных функций на поведение программы при поиске заданных ситуаций.
- Анализ должен быть масштабируемым. Размер анализируемых программ может достигать миллионов строк кода и сотен тысяч функций.
- Инфраструктура анализа должна быть расширяемой. В настоящее время анализатор позволяет находить довольно широкий класс ошибок: разыменование нулевого указателя, использование неинициализированных данных, переполнение буфера, уязвимости форматной строки, утечки памяти, некорректная работа с динамической памятью и др. Этот список постоянно дополняется.
- Доступ к полному исходному коду программы не является необходимым. Для стандартных библиотечных функций Svace имеет внутренние спецификации, описывающие их действия.
- Значительная часть выдаваемых предупреждений должна быть истинной, т.е. необходимо находить ситуации, действительно подпадающие под описание выданного типа предупреждения как ошибочные или опасные; это не означает, что такие ситуации обязательно будут уязвимостями или проявятся во время выполнения программы.

Дополнительными практическими задачами являются обеспечение удобного пользовательского интерфейса для просмотра результатов анализа и настройки инструмента, а также наличие дополнительных возможностей, таких как удаленный анализ и ведение истории результатов анализа.

Естественным ограничением, вытекающим из поставленных требований, является то, что не гарантируется нахождение всех ошибок и допускается наличие ложных срабатываний. Все известные коммерческие инструменты для автоматического статического анализа, не требующего трудоемкой подготовки анализируемых программ или требований к ним (SAVE компании Coverity [4] и Insight компании Klocwork [5]), обладают таким же ограничением.

3. Схема проведения анализа

Анализ с помощью Svace проходит в несколько этапов, как показано на рис. 1. Прежде всего необходимо подготовить внутреннее представление для исходного кода анализируемой программы. Для этого в процессе сборки анализируемого приложения перехватываются запуски компилятора и других утилит, влияющих на процесс сборки. При этом перехват проводится прозрачно, то есть не требует менять файлы системы сборки. С помощью перехваченных команд генерируются новые командные строки для запуска внутреннего компилятора Svace, который генерирует внутреннее представление для исходного кода приложения. Этот компилятор основан на модифицированном компиляторе Clang с открытым исходным кодом. В качестве внутреннего представления, соответственно, используется биткод LLVM. Он содержит описания всех используемых типов, глобальных и локальных переменных модуля компиляции, а также текст функций кода для абстрактной регистровой машины.

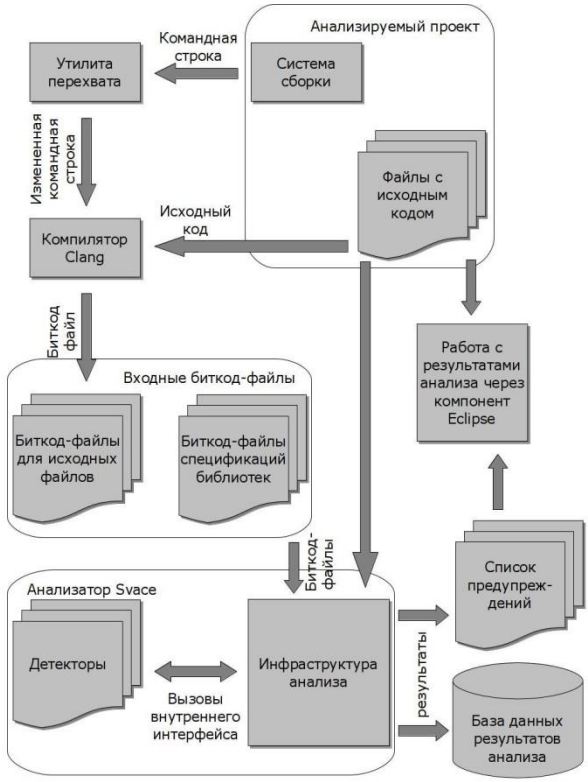


Рис. 1. Схема анализа

Файлы внутреннего представления исходного кода анализируемой программы сохраняются на диск для последующего анализа. При запуске анализа они считываются с диска вместе с файлами биткода, содержащими спецификации стандартных библиотечных функций. Инфраструктура анализа проводит анализ внутреннего представления, используя набор правил нахождения определенных видов дефектов, предоставляемых подключенными детекторами, и в результате выдает список предупреждений. Этот список может быть импортирован в базу данных истории результатов предупреждений для отслеживания предупреждений об исправленных или внесенных дефектах с предыдущего анализа. Также этот список может быть загружен в среду разработки Eclipse с возможностью навигации по участкам кода, соответствующим предупреждениям.

4. Межпроцедурный анализ на основе аннотаций

В основе Svace находится анализ “снизу-вверх”, основанный на параметризованных аннотациях. Во время анализа производится обход функций программы в соответствии с графом вызовов таким образом, чтобы вызываемая функция анализировалась перед вызывающей[1]. Каждая функция анализируется только один раз и независимо от других функций, что позволяет добиться почти линейной масштабируемости от количества функций в анализируемой программе. В результате анализа функции (помимо предупреждений о найденных дефектах) создаётся аннотация — структура данных ограниченного размера, приближенно описывающая наиболее важные для анализа эффекты от вызова этой функции в произвольном контексте. Различные детекторы могут сохранять в аннотациях необходимые им данные. Аннотации являются параметризованными — все свойства описываются относительно входных параметров и их отношений. В терминах анализа потока данных аннотация является структурой, достаточной для создания передаточной функции вызова процедуры.

Во время анализа кода каждой функции при обработке инструкции вызова функции используется её аннотация. Производится процесс трансляции аннотации: формальные параметры сопоставляются фактическим, побочные эффекты аннотации отражаются в контексте точки вызова. При этом анализ не меняет аннотацию вызываемой функции.

При анализе функции предполагается, что входные параметры указывают на непересекающиеся области памяти. Подобное допущение соответствует семантике большинства функций, и позволяет резко сократить сложность анализа указателей и повысить точность, а также скорость анализа [6].

1 Если подобное невозможно (например, при использовании рекурсивных вызовов), из графа вызовов удаляется одно из ребер.

Все аннотации функций по возможности хранятся в оперативной памяти, но при ее нехватке сохраняются на диск и считываются при необходимости. Сохранение на диск позволяет снять ограничение на размер анализируемого кода и выполнять анализ программ размером во много миллионов строк кода. Тем не менее, обычного современного объема оперативной памяти (1-4 Гб) хватает для анализа программ из сотен тысяч строк кода без использования сохранения на диск.

5. Внутрипроцедурный анализ

Во время анализа одной функции анализатору доступно внутреннее представление анализируемой функции, а также аннотации всех непосредственно вызываемых ей функций. Для функции строится граф потока управления, в узлах которого лежат инструкции анализируемой функции, а ребра отражают возможный переход управления между ними. Затем этот граф потока управления обходится во время символьного выполнения в топологическом порядке. После нескольких прямых проходов, производится дополнительный обратный проход.

С каждым ребром графа потока управления ассоциируется контекст — информация о потоке данных, установленная для путей исполнения, проходящих через данное ребро. Контекст описывает взаимосвязь между ячейками памяти, идентификаторами значений и атрибутами. Абстрактные ячейки памяти моделируют ячейки памяти, к которым происходит обращение в программе на различных путях исполнения. Идентификаторы значений обозначают значения, разделяемые различными ячейками памяти без изменения (схожей цели служат поколения переменных в представлении с единственным присваиванием, SSA). Например, для значения, скопированного из одной переменной в другую, поддерживается один идентификатор значения, так что этот идентификатор становится общим для нескольких абстрактных ячеек памяти. Атрибуты описывают анализируемые свойства абстрактных ячеек памяти и идентификаторов значений, например, интервал возможных целочисленных значений или зависимость от данных, полученных из сети.

Больше всего Svace оперирует с атрибутами идентификаторов значений, т.к. наиболее интересные свойства являются свойствами значений. Например, при поиске ошибок разыменования нулевого указателя необходимо проверить, что значение переменной, которую разыменовывают, равно нулю. Для этого можно создать специальный атрибут, который будет обозначать свойство, что значение равно нулю (противоположное свойство - “значение не обязательно равно нулю”), и помечать им идентификаторы значений переменных, которые заведомо равны нулю. После этого для выявления дефекта достаточно при обработке инструкции разыменования проверить значение такого атрибута у идентификатора значений, соответствующего разыменовываемой абстрактной ячейке памяти.

Для каждого типа дефектов определяются свои атрибуты, необходимые для обнаружения искомых ситуаций, и правила их распространения. В основную задачу инфраструктуры анализа входит распространение атрибутов между контекстами на ребрах (в зависимости от инструкций в узлах) графа потока управления в пределах одной функции во время внутрипроцедурного анализа и между различными функциями с помощью аннотаций.

Несмотря на то, что аннотация функции создается без учета вызываемого контекста, обработка вызова функции производится контекстно-чувствительным образом. Инфраструктура анализа производит трансляцию аннотации в контекст вызова, при этом элементы вызывающего контекста изменяются в зависимости от содержания аннотации.

На рис. 3 схематично показан пример аннотации для функции, изображённой на рис. 2. На схеме p и x - формальные параметры функции, относительно которых параметризуется аннотация; ret - возвращаемое значение функции; Vp, Vx - идентификаторы значений для формальных параметров; *p - абстрактная ячейка памяти, обозначающая область памяти, на которую указывает формальный параметр p. В аннотации хранится информация о том, что функция возвращает значение своего формального параметра Vx, в ячейку памяти *p записано значение Vx. Дополнительно сохраняются атрибуты deref и pt-to, которые означают, что значение Vp было разыменовано и указывает на абстрактную ячейку памяти *p.

```
int set_p(int *p, int x) {
    *p = x;
    return x;
}
```

Рис. 2. Код модельной функции

Идентификаторы значений		Ячейки памяти	
p	Vp	*p	Vx
x	Vx	Атрибуты	
ret	Vx	Vp	deref, pt-to { *p }

Рис. 3. Аннотация модельной функции

6. Чувствительность к путям

Одним из полезных качеств анализа является чувствительность к путям, когда каждый возможный путь выполнения кода функции анализируется независимо. Однако из-за практических требований такой анализ нельзя проводить полноценно. В некоторых случаях это может приводить к ложным срабатываниям.

```
1: if(!x)
2:     y = 0;
3: if(x)
4:     *y;
```

Рис. 4. Зависимые условия.

В примере на рис. 4 из 4 путей выполнения программы (1→2→3→4, 1→3→4, 1→3→4, 1→3) два являются невыполнимыми (1→2→3→4 и 1→3), так как условия x и !x взаимно исключают друг друга. Поэтому, если выполнение достигло строки 4, то инструкция присваивания нулевого значения в строке 2 не могла быть выполнена. Для повышения точности анализа при сохранении масштабируемости в Svace используется два механизма: зависимые атрибуты и обратный анализ.

Зависимые атрибуты — это атрибуты, имеющие ссылки на идентификаторы значений. Можно создать атрибут IsNull, означающий, что переменная имеет нулевое значение, если другая переменная имеет такое значение. В примере y имеет нулевое значение, если x равно нулю. Зависимые атрибуты позволяют добавить чувствительность к путям для отдельных типов дефектов, для которых чувствительность к путям особенно критична. Svace использует зависимые атрибуты, например, для поиска ошибок двойных блокировок. Это обусловлено тем, что, во-первых, блокировки довольно часто находятся под некоторым флагом, а во-вторых, блокировки используются довольно редко, поэтому их более детальное рассмотрение не замедлит анализ в целом.

Другим механизмом подавления ложных срабатываний на несуществующих путях является обратный анализ. Цель обратного анализа — выделить ситуации, в которых, если происходит одно событие, то другое будет происходить для всех возможных путей выполнения. В примере выше после выполнения инструкции присваивания в строке 2, сравнение в строке 3 выполнится в любом случае, а разыменование в строке 4 только, если условие будет истинным.

На рис. 5 приведены некоторые возможные варианты использования присваивания нулю и разыменования. Инструкции обозначены прямоугольниками, пустые прямоугольники обозначают любые другие

инструкции, которые не изменяют переменную x и не разыменовывают её. На рисунках А, Б, Г и Д после присваивания переменной нуля всегда происходит разыменование нулевого указателя. При этом на рисунке Г разыменование происходит в разных местах, а на рисунке Д дважды встречается присваивание нулю. На рисунке В, если управление дошло до инструкции присваивания, то разыменование выполнится в любом случае. А на рисунке Е наоборот, если управление попало в инструкцию разыменования, то присваивание обязательно было выполнено. Во всех выше перечисленных случаях выдается ошибка разыменования нулевого указателя. На рисунке Ж и присваивание и разыменование выполняются условно, если условия зависимые, то без дополнительного анализа нельзя сказать, будет ли разыменование нулевого указателя в этом случае.

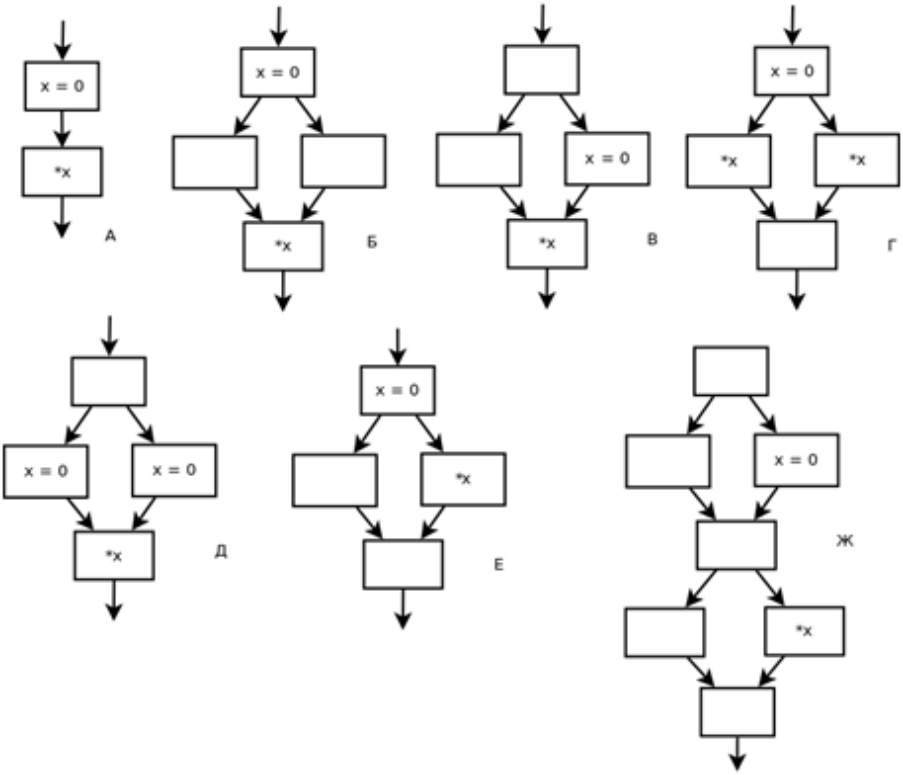


Рис. 5. Использование обратного анализа

Обратный анализ позволяет различить ситуации А-Е от Ж. Общая идея для рассматриваемого типа дефекта заключается в использовании двух булевых

атрибутов IsNull и IsDeref. Во всех контекстах, соответствующих инструкциям программы, которые выполняются после присваивания переменной нулевого значения, атрибут IsNull для этого идентификатора значений будет иметь истинное значение. Этот атрибут вычисляется только при прямом анализе. На обратном проходе атрибут IsDeref получает истинное значение для тех идентификаторов значений, которые были разыменованы на все путях ниже по графу потока управления. В точке, где после обратного анализа идентификатор значения переменной имеет одновременно истинные атрибуты IsDeref и IsNull, выдаётся ошибка разыменования нулевого указателя, т.к. на всех путях выполнения, проходящих через данную точку, было присваивание переменной нулю, и в то же время будет разыменование этой переменной. Таким образом, благодаря обратному анализу можно эффективно и с минимальными накладными расходами получить частичную чувствительность к путям.

7. Пример поиска предупреждения DEREF_AFTER_NULL

Опишем детектор, осуществляющий поиск следующего дефекта: «переменная была разыменована после положительного сравнения с нулём». Приведём пример работы детектора для фрагмента кода на рис. 6. В строке 4 переменная x сравнивается с нулём, а затем разыменовывается либо в строке 8, либо в строке 10. Таким образом пример содержит либо лишнее сравнение с нулём в строке 4, либо разыменование нулевого указателя.

```
1: func(int*x, int* p, int m) {
2:     if(p==0) {}
3:     else {
4:         if(x==0) {}
5:         else *p = 0;
6:     }
7:     if(m==0)
8:         *x = 1;
9:     else
10:       *x = 2;
11: }
```

Рис. 6. Пример для DEREF_AFTER_NULL

Для поиска ошибки мы используем два булевых атрибута идентификаторов значений: IsNullCompared, имеющий значение истина, если значение переменной было положительно сравнено с нулём на всех путях выше по графу потока управления, и ложь иначе (отсутствие информации); и атрибут IsDeref, имеющий значение истина, когда значение было разыменовано без проверки на ноль, и ложь во всех остальных случаях. Предупреждение

выдаётся, если после положительной проверки на нуль на всех достижимых путях выполнения происходит разыменование нулевого указателя. Нам потребуется обратный анализ, чтобы распространить атрибут IsDeref до точек, где значение указателя всегда равно нулю (IsNullCompared имеет значение истина).

Выполнение анализа показано на рис. 7. Для удобства рисунок содержит точки функции, соответствующие пустым веткам условия в строках 2 и 4, а также две вершины Join1 и Join2, обозначающие слияние путей выполнения. На рисунке показаны детали анализа только для переменных x, p и m. Для обратного анализа стрелки между инструкциями показывают направление анализа, а не выполнение программы. Предупреждение DEREF_AFTER_NULL выдаётся, т. к. в строке 4 идентификатор значения имеет истинными одновременно атрибуты IsNullCompared и IsDeref.

Рис. 7. Прямой и обратный анализ функции func

8. Расширение анализа

Для расширения возможностей анализатора нужно уметь добавлять новые виды обнаруживаемых дефектов. При разработке Svace требовалось, чтобы добавление нового типа предупреждения не влекло серьёзной переработки всей системы. Для этого в составе Svace выделяется инфраструктура анализа, содержащая общие части для всех видов анализов, а конкретный анализ реализуется с помощью детекторов для каждого типа дефекта или группы схожих дефектов [7].

Инфраструктура анализа создаёт идентификаторы значений и абстрактные ячейки памяти, оповещает детекторы о важных операциях в программе, а детекторы могут подписываться на интересующие их операции, просматривать состояние программы перед выполнением операций, и модифицировать атрибуты выходного состояния. При этом для большинства случаев используются стандартные типы атрибутов: булевы атрибуты, тернарные атрибуты, интервалы, двойные интервалы, или атрибуты, параметризованные идентификаторами значений.

Такая структура позволяет достаточно легко добавлять новые типы предупреждений. Для большинства видов потенциальных ошибок необходимо понять, в чём состоит нежелательная ситуация, какими свойствами должны обладать переменные и их значения для возникновения ошибки. После этого, чтобы реализовать детектор, достаточно описать то, как нужно помечать переменные или их значения некоторыми атрибутами, обозначающими интересующие свойства, а также указать правила продвижения атрибутов по контексту и условия выдачи предупреждения.

Ещё одним преимуществом разделения анализа на общую инфраструктуру и детекторы является низкая стоимость включения отдельного детектора. Большинство необходимых действий производится инфраструктурой анализа, поэтому добавление нового детектора незначительно изменяет общее время анализа.

9. Скорость и масштабируемость

Высокая скорость позволяет чаще производить поиск ошибок, в том числе для вновь сделанных изменений. Медленный анализ может стать причиной отказа от инструмента. Другой важной характеристикой является масштабируемость. Многие инструменты не способны проанализировать большие программы из-за нелинейного роста требований к памяти и процессорному времени. Это сильно ограничивает сферу их применения. При разработке Svace скорость и масштабируемость были одними из главных требований, которые были успешно учтены.

В табл. 1 приведены времена анализа инструментом Svace некоторых проектов различного размера. Для проведения анализа использовались две машины: персональный компьютер с процессором Intel Core i7 (4 ядра) и 12ГБ

оперативной памяти и двухпроцессорный сервер на основе Intel Xeon с 8 ядрами и 70ГБ оперативной памяти, на обоих системах реализована технология гиперпоточность (hyper-threading). Учитывалось только время анализа, время требуемое для компиляции проектов не приведено. При этом были включены все реализованные детекторы. Для операционной системы Android анализ производился только на Intel Xeon. Ядро операционной системы Linux на обычном компьютере было проанализировано быстрее, чем за полчаса. Существует довольно мало проектов большего размера чем ядро Linux и операционная система Android. Поэтому можно считать, что представленный анализ масштабируется для любых существующих проектов.

Пакет	**Размер, тыс. строк кода**	**Количество функций**	**Скорость анализа для Intel Core i7, 4 ядра, 12 Гб**	**Скорость анализа для Intel Xeon, 8 ядер, 70 Гб**
pcre3-7.8	19	506	2 м. 59 с.	40 с.
openssh-5.3p1	65	1549	1 м. 15 с.	51 с.
openssl-0.9.8k	240	5030	3 м. 53 с.	2 м. 25 с.
ffmpeg-0.5.1	300	7119	6 м. 21 с.	3 м. 33 с.
postgresql-8.4	501	13237	11 м. 35 с.	6 м. 10 с.
binutils-2.20.1	1086	24271	19 м. 20 с.	5 м. 35 с.
linux-kernel-3.10.9	6219	53114	28 м. 12 с.	9 м. 58 с.
android-4.2.1	12250	584743	-	2 ч. 43 м.

Табл. 1. Время анализа проектов с помощью анализатора Svace

В настоящее время широкое распространение получили вычислительные машины с несколькими процессорами или ядрами. Алгоритм анализа на основе аннотаций, применяемый в Svace, довольно хорошо распараллеливается и позволяет эффективно использовать многоядерные и многопроцессорные машины. В качестве атомарной единицы анализа используется анализ функции, поэтому потенциал распараллеливания зависит

от ширины графа вызовов. В табл. 2 приведено время анализа приложения ffmpeg-0.5.1 на упоминавшемся выше сервере Intel Xeon при использовании разного количества потоков.

Количество используемых потоков	**Время анализа, сек.**
1	1275
2	652
3	445
4	343
6	252
8	225
16	213

Табл. 2. Анализ с разным количеством используемых потоков.

10. Легковесный анализ лексических, синтаксических, семантических и ряда ситуационных ограничений

Для проверки языково-зависимых правил языков C и C++ может использоваться легковесный анализатор на основе Clang [8]. Он позволяет быстро анализировать лексические, синтаксические, семантические и ряд ситуационных правил. В отличие от Svace, доступна полная информация об исходном тексте программы, в том числе, не влияющая на семантику программы, например, комментарии, переводы строк, пробелы и т.д. В качестве внутреннего представления используется аннотированный абстрактный семантический граф (ААСГ), строящийся на основе абстрактного синтаксического дерева, дополненного помеченными ребрами и аннотациями узлов. ААСГ позволяет моделировать не только семантику языка, но и граф потока управления, граф вызовов и используется для анализа потоков данных. Каждый узел графа имеет тип, определяющий его семантику. Множество всех типов ограничено и состоит из подмножеств для каждого этапа компиляции: *Lex={Keyword, Identifier, ...}, AST={IfStmt, BinaryOperator, ...}* и т.д. Для анализа работы с памятью используется понятие *область памяти* $A = StorageClass \times N_0$, где $N_0 = N \bigcup \{0\}$ – декартово произведение класса памяти и размера в битах, что позволяет учитывать битовые операции и структуры с битовыми полями. Всего определено 10 классов памяти, разделенных на 4 подкласса:

1. стековый (автоматические переменные, параметры функции, выделенная alloca);
2. код (указатели на функцию, метки);
3. данные (глобальная системная память, глобальная статическая, глобальная);
4. динамическая память (выделенная malloc, new).

Для поддержки элементов массива и полей структур используется *подобласть памяти* $A_{\subset} = A_{base} \times N_0$ – декартово произведение базовой области памяти и размера. Состояние памяти задается как $Stmt \times Env \times Mem$, где Stmt – идентификатор оператора программы, $Env = \{\alpha : Var \mapsto A, \alpha^f : A_{base} \times Field \mapsto A_{\subset}, \alpha^e : A_{base} \times N_0 \mapsto A_{\subset}\}$ – состояние переменных, $Mem = \{\lambda : A \mapsto \{0,1\}^*, \hat{\lambda} : A \mapsto A_{\subset}\}$ – состояние памяти. Контекст функции не учитывается для ускорения анализа. В табл. 3 приведены примеры правил вычисления областей памяти для выражений C и C++.

Выражение(E)	l(E)	r(E)
константа C	Ø	C
переменная V	A^V	$\lambda(A^V)$
s.f	$A^f_{\subset}(A^s)$	$\lambda(A^f_{\subset}(A^s))$
*E	$\hat{\lambda}(l(E))$	$\lambda(l(E))$

Табл. 3. Пример вычисления l-value и r-value выражений

Для ускорения анализа необходимо избегать полной символьной интерпретации и вычислять только необходимую информацию. Поэтому лишь минимальное количество часто используемых атрибутов вычисляется заранее. На начальных этапах работа анализатора аналогична компилятору. Основное отличие заключается в вычислении дополнительных атрибутов и проверке ограничений. Правила не модифицируют ААСГ, а только активируют функции вычисления «редких» атрибутов, которые могут быть сохранены.

Поэтому они обрабатываются параллельно. Для ряда типов правил, например, для проверки корректности обработки исключений, разработаны межмодульные алгоритмы анализа, экспортирующие необходимые данные в процессе работы и выполняющие проверку на этапе линковки с помощью интеграции в систему сборки программ. Система состоит из трёх основных компонентов: подсистема определения правил, сбора информации и планировщик правил. Ограничения задаются в виде классов, унаследованных от базового класса правил и содержат список типов необходимых узлов ААСГ, используемый для планирования порядка их проверки. Доступ к атрибутам узлов ААСГ осуществляется с помощью методов соответствующего класса, что позволяет получать доступ к данным по мере вычисления их анализатором или запускать “ленивые” алгоритмы их вычисления. Реализованная подсистема способна проверять более 50 различных правил. Среднее замедление сборки проекта при включенном анализаторе составляет 22 %, что допускает его применение при каждой компиляции и позволяет обнаруживать ошибки на самых ранних стадиях. Результаты работы легковесного анализатора могут быть импортированы в Svace позднее во время его анализа и показаны наряду с предупреждениями, выданными самим Svace.

11. Просмотр результатов в среде Eclipse

После окончания анализа его результаты могут быть просмотрены пользователем через компонент в среде разработки Eclipse. Для каждого предупреждения показывается сообщение и релевантные места в исходном коде программы (таких мест может быть несколько, например, может быть указана полная межпроцедурная трасса по функциям программы, приводящая к ошибке). Пример просмотра результатов анализа в Eclipse приведен на рис. 8.

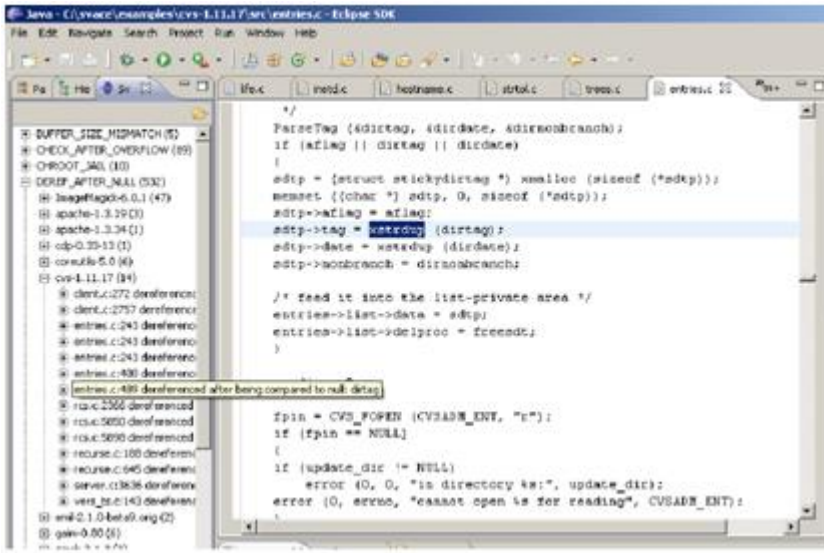


Рис. 8. Графический интерфейс компонента Svace в среде разработки Eclipse

12. База данных результатов анализа

Результаты анализа, представляющие из себя набор предупреждений, могут сохраняться как в специальных файлах, хранящих в себе результаты отдельного анализа, так и в базе данных. База данных позволяет сохранять результаты множества последовательных анализов одного проекта (между которыми исходный код проекта может меняться), и сопоставляет предупреждения с этих различных анализов, что позволяет отслеживать историю изменения отдельных предупреждений. Также база данных позволяет пользователю устанавливать комментарии и различные метки для предупреждений (например, размечать истинные и ложные срабатывания анализатора). Они автоматически переносятся между последовательными результатами анализа. Сопоставление предупреждений на различных результатах анализа производится путём выявления для каждого сопоставляемого нового предупреждения наиболее похожих предупреждений из уже существующих в базе данных. Схожесть предупреждений вычисляется на базе различных параметров, таких как тип предупреждений, контекст положения предупреждения в исходном коде, наборе имён участвующих в проблемной ситуации переменных.

Система контроля истории результатов анализа также позволяет производить синхронизацию хранилищ результатов анализа по сети. Это позволяет получать новые результаты анализа (например, с централизованного сервера анализа) и производить синхронизацию комментариев и меток, созданных различными пользователями.

13. Примеры находимых дефектов

В данном разделе приведено несколько типичных ошибок, найденных в популярных проектах с исходным открытым кодом.

Многие ошибки являются результатом вставки кода из другого места и долгое время остаются незамеченными. На рис. 9 показан пример подобной ошибки. Функция fopen в случае неудачного выполнения возвращает нулевой указатель. В этом случае приведённый код сообщает об ошибке, очищает ресурсы и зачем-то вызывает функцию fclose. Передача нулевого указателя в функцию fclose является неопределённым поведением. Подобную ошибку легко допустить и относительно сложно найти с помощью тестирования, т. к. она происходит на редком пути исполнения. При этом для статических анализаторов реализация детектора для нахождения подобных дефектов не является сложной задачей.

```
outfile = fopen(argv[2], "wb");
if (!outfile) {
      perror(argv[2]);
      fclose(outfile);
      free(moov_atom);
      return 1;
}
```

Рис. 9. Ошибка DEREF_OF_NULL в ffmpeg-0.5.1/tools/qt-faststart.c

Некоторые библиотечные функции могут возвращать нулевой код возврата в качестве индикации ошибки. Например, функция malloc стандартной библиотеки языка Си возвращает нуль, если не удалось выделить память. Программисты часто не проверяют код возврата функции malloc, т.к. нехватка памяти довольно редкое явление, и даже в случае недостатка памяти, непонятно что делать с такой ситуацией. Тем не менее во многих проектах приняты более жёсткие стандарты, в которых необходимо проверять результат malloc. Типичный пример подобной ошибки, найденной в busybox, приведён на рис. 10.

```
sym->user.val = val = malloc(size);
*val++ = '0';
*val++ = 'x';
```

Рис. 10. Ошибка DEREF_OF_NULL.ALLOC в busybox-1.13.3/scripts/kconfig/symbol.c

Ошибки, допущенные в программе, могут приводить не только к падению программы или некорректной работе, но и создавать уязвимости системы. Svace имеет группу предупреждений, проверяющих, что полученные из внешней среды данные не используются в критичных операциях. Пример подобной уязвимости показан на рис. 11, где переменная env указывает на массив, полученный из переменной окружения с помощью функции getenv, которая затем без проверок используется в качестве аргумента форматной строки в функции sprintf.

```
env = getenv(SRCTREE);
if (env) {
 sprintf(fullname, "%s/%s", env, name);
}
```

Рис. 11. Ошибка TAINTED_PTR в busybox-1.13.3/scripts/kconfig/confdata.c

Поиск утечек памяти является сложной задачей, тем не менее, многие ошибки подобного вида могут быть найдены с помощью статического анализа. На рис. 12 показан фрагмент кода, где с помощью функции strdup выделяется память, которая затем присваивается указателю ‘type’ и никогда не освобождается.

```
case 't': {
          char *type;

          type = strdup(optstate->value);
          crlType = atoi (type);
          if (crlType != SEC_CRL_TYPE && crlType
!= SEC_KRL_TYPE) {
            PR_fprintf(PR_STDERR, "%s: invalid crl
type\n", progName);
            PL_DestroyOptState(optstate);
            return -1;
          }
          break;
```

Рис. 12. Ошибка MEMORY_LEAK.STRDUP в nss-3.12.6/mozilla/security/nss/cmd/crlutil/crlutil.c

14. Заключение

В статье описывается инструмент статического анализа Svace, нацеленный на практическое использование для автоматического поиска дефектов и уязвимостей в программах на языках Си и Си++. В основе анализа лежит межпроцедурный анализ потока данных, что позволяет обнаруживать дефекты, причинами которых являются сложные взаимодействия между функциями разных модулей компиляции. При этом анализ обладает хорошей скоростью, в том числе за счет параллелизации, и масштабируется до проектов из миллионов строк кода. Качество анализа при этом сравнимо с лучшими коммерческими аналогами. Svace прост в использовании и обладает расширенным набором пользовательских возможностей, в том числе, графическим интерфейсом и поддержкой истории результатов анализов. Инструмент находится в развитии и в ближайшее время получит возможность анализировать программы на языке Java.

Список литературы

[1]. S.C. Misra, V.C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality, in: Proceedings of the International Conference on Computational Science and its Applications, ICCSA, in: Lecture Notes in Computer Science, vol. 2667, Springer, Montreal, Canada, 2003, pp. 724–732.

[2]. В.С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции «РусКрипто’2009».

[3]. А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, 2011, с. 23–38.

[4]. Инструмент Coverity SAVE. http://www.coverity.com/products/coverity-save.html
[5]. Инструмент статического анализа компании Klocwork. http://www.klocwork.com/products/insight/klocwork-truepath
[6]. V. Benjamin Livshits, Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs, 2003.
[7]. А. Аветисян, А. Бородин. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН, том 21, 2011, с. 39–54.
[8]. Игнатьев В.Н. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования. Труды ИСП РАН, том 22, 2012, с. 169–188. DOI: 10.15514/ISPRAS-2012-22-11.

Svace: static analyzer for detecting of defects in program source code

Viktor Ivannikov <ivan@ispras.ru>, ISP RAS, Moscow, Russia
Andrey Belevantsev <abel@ispras.ru>, ISP RAS, Moscow, Russia
Alexey Borodin <alexey.borodin@ispras.ru>, ISP RAS, Moscow, Russia
Valery Ignatyev <rook@ispras.ru>, ISP RAS, Moscow, Russia
Dmitry Zhurikhin <zhur@ispras.ru>, ISP RAS, Moscow, Russia
Arutyun Avetisyan <arut@ispras.ru>, ISP RAS, Moscow, Russia
Maksim Leonov <maksim.leonov@ispras.ru>, ISP RAS, Moscow, Russia

Abstract. High complexity of present-day programs makes it nigh impossible to write a program without a defect. Thus it is increasingly necessary to use tools for defects detection. This article presents Svace, a tool for static program analysis developed in ISP RAS. This instrument allows to automatically find defects and potential vulnerabilities in programs written in C and C++ languages. Main features of the tool are simplicity of usage, deep interprocedural analysis, wide variety of supported warning types, scalability up to programs of millions lines of code and acceptable quality of analysis (30-80% of true positive warnings).

In the core of the Svace tool lies an engine for interprocedural data-flow analysis based on function annotations. Each function is analyzed once and independently of the other functions which allows to achieve almost linear scalability (Linux kernel can be analyzed within 10 minutes on a relatively powerful machine and analysis of the whole Android source code takes less than 3 hours). Intraprocedural analysis is performed on source code internal representation derived from LLVM bitcode. It operates with value identifiers that are shared between memory locations with same values (similarly to generations in SSA representation). Special attributes of these value identifiers are calculated over the control-flow graph of the function. When specific combination of attributes is observed a defect warning is issued. Svace analysis engine is accompanied by Clang compiler-based lightweight analysis tool for checking of language-dependent rules which allows to quickly check a number of syntactic, semantic and situational rules. Analysis results can be presented to the user with the help of Eclipse IDE plugin. They can also be imported into analysis results database to trace history of program defects over time.

Keywords: static analysis; data-flow analysis; vulnerabilities; interprocedural analysis; annotation-based analysis.

References

[1]. S.C. Misra, V.C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality, in: Proceedings of the International Conference on Computational Science and its Applications, ICCSA, in: Lecture Notes in Computer Science, vol. 2667, Springer, Montreal, Canada, 2003, pp. 724–732.

[2]. V.S. Nesov. Automatic defect detection with the help of interprocedural static analysis of source code, in: Proceedings of International Conference Ruscrypto'2009.

[3]. Coverity SAVE, http://www.coverity.com/products/coverity-save.html
[4]. Klocwork Insight, http://www.klocwork.com/products/insight/klocwork-truepath
[5]. V. Benjamin Livshits, Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs, 2003.
[6]. A. Avetisyan, A. Borodin. Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors, in: Proceedings of the Institute for System Programming of RAS, volume 21, 2011, pp. 39-54.
[7]. V.N. Ignatyev. Using static analysis for checking configurable semantic restrictions on a programming language, in: Proceedings of the Institute for System Programming of RAS, volume 22, 2012, pp. 169-188. DOI: 10.15514/ISPRAS-2012-22-11.

Методы и программные средства, поддерживающие комбинированный анализ бинарного кода ⋆

В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко
{ vartan, thorin, eyescream, bakulinm, helendile, korpse, il, ustas, spanasenko}@ispras.ru

Аннотация. В статье рассматриваются разработанные в ИСП РАН методы и инструменты анализа бинарного кода и их применение к задачам восстановления алгоритмов и форматов данных. Предметом анализа выступает исполняемый код различных процессорных архитектур общего назначения в отсутствии исходных кодов, отладочной информации и привязки к определенным версиям операционных систем. Подход состоит из сбора детальной трассы выполнения уровня машинных команд; метода последовательного повышения уровня представления; выделения кода алгоритма и последующей структуризации как кода, так и форматов обрабатываемых данных. Были достигнуты важные результаты: разработано промежуточное представление, позволяющее проводить большую часть предварительных обработок и выделение кода алгоритма без привязки к особенностям определенной машины; разработан метод и инструмент автоматизированного восстановления форматов сетевых сообщений и файлов. Разработанные инструменты были интегрированы в единую среду анализа, поддерживающую совместное их использование; архитектура среды также описана в статье. Приводятся примеры применения к реальным программам.

Ключевые слова: бинарный код, статический анализ, динамический анализ, интегрированная среда.

1. Введение

На протяжении нескольких лет в ИСП РАН ведутся работы по разработке интегрированной среды анализа бинарного кода. Потребность в такой среде возникает у любого специалиста, пытающегося провести анализ исполняемого (бинарного) кода достаточно большой программы с целью достичь понимания ее устройства. Последующее применение достигнутого понимания зависит от того, в какой прикладной области работает специалист. Достаточно типичными ситуациями, когда требуется понять устройство программы по ее

⋆ Работа поддержана грантом РФФИ 11-07-00450-а

бинарному коду, являются: изучение вредоносного кода разработчиками антивирусов, оценка свойств алгоритмов во время сертификационных испытаний, восстановление закрытых протоколов для их последующей реализации в программах с открытым исходным кодом, выявление ошибок и выделение среди них уязвимостей, и т.п.

Распространенная ранее практика предполагала применение интерактивного дизассемблера (фактически безальтернативным инструментом до сих пор является IDA Pro) и интегрированного отладчика. Из-за легкости обнаружения отладчиков, работающих в одном с исследуемой программой окружении, предпочитают отладчики, работающие в виртуальных машинах. Связка из IDA Pro и отладчика позволяет справляться с некоторыми базовыми проблемами анализа бинарного кода, например, различать в рамках отдельного выполнения код и данные, определять целевые адреса в случае косвенной адресации. Не решив последнюю задачу, невозможно гарантировать корректное представление изучаемой программы даже в виде декодированных машинных команд.

Приведенный способ анализа имеет принципиальное ограничение: он позволяет изучать только небольшие фрагменты кода, автоматизация работы в этом случае крайне затруднительна. В результате проведенных ранее исследований был предложен метод динамического анализа [1, 2], позволяющий преодолеть указанное ограничение, а также и некоторые другие, характерные для интерактивной отладки. Идея метода заключается в получении детальной трассы выполнения на уровне машинных команд и ее последующем глубоко автоматизированном анализе. В последующих работах метод был развит [3, 4] за счет соединения динамического и статического анализа с целью взаимной компенсации слабых сторон этих подходов, когда они применяются по отдельности.

Метод был реализован в интегрированной среде анализа (СА) бинарного кода (рис. 1). В качестве средства получения трассы, как правило, используется виртуальная машина QEMU, поскольку в ней был реализован механизм детерминированного воспроизведения [5], позволяющий расширить классы анализируемых программ. Реализованные в СА алгоритмы поэтапно поднимают уровень представления, восстанавливая события уровня ОС и системы программирования, и сохраняя полученные результаты в виде разметки трассы. На основе собранных и размеченных трасс строится статико-динамическое представление, отражающее изменение кода с течением времени. Имеется возможность экспортировать это представление в IDA Pro, но и сама СА обладает широкими возможностями, опирающимися на результаты трассировки. К их числу относятся: навигация в трассах и статическом коде, выделение кода отдельных алгоритмов, восстановление структуры этих алгоритмов и форматов обрабатываемых данных, описание функций и построение модельных примеров, содержащих извлеченные из бинарного кода реализации алгоритмов.

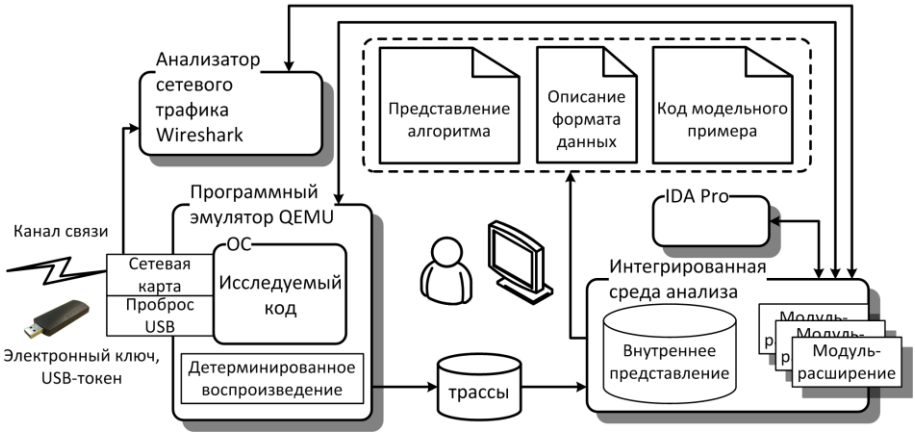


Рис. 1. Внешняя архитектура среды анализа бинарного кода.

Основные представленные в статье результаты затрагивают следующие вопросы: поэтапное повышение уровня представления (раздел 2), которое выполняется в автоматическом режиме при импорте в CA только что записанной трассы; выделение алгоритма на основе динамического слайсинга бинарного кода (раздел 3); и восстановление форматов данных (раздел 4). В Разделе 5 описана внутренняя архитектура CA.

2. Предварительное повышение уровня представления

Типовой ситуацией является задача анализа бинарного кода приложения, работающего под управлением известной ОС (Windows, Linux, BSD и т.п.). Как уже упоминалось выше, инструментами анализа будут выступать IDA Pro и отладчик. Неизбежные затруднения в данном случае будут обусловлены косвенной адресацией и размером анализируемого кода. Поскольку предполагается, что анализ происходит не на этапе разработки ПО, исполняемый файл не содержит никакой отладочной информации, доступен только объектный (машинный) код и, возможно, таблицы экспортируемых функций. Источником знаний о том, что делается программой, является семантика машинных команд, вызовы библиотечных функций, в том числе API ОС и библиотек поддержки времени выполнения. Используется неявное предположение, что вся функциональность заключена в одном пользовательском процессе.

Расширение класса анализируемых программ сразу же вызывает нарастающие технические трудности. Простая задача – анализ в IDA Pro кода программы, состоящей из нескольких модулей, – требует либо получения снимка

адресного пространства процесса в некоторый момент времени, причем необходимо определить, в какой именно, либо дополнения базового функционала IDA Pro. Более тяжелые задачи возникают при анализе поведения вредоносного кода, например, когда приходится отслеживать его распространение в адресных пространствах нескольких процессов, памяти ОС.

Следующим направлением для расширения класса задач является анализ кода BIOS, гипервизоров, систем, перехватывающих обращения ОС к аппаратуре, например, систем полнодискового шифрования, а также самих ОС, когда они представляют собой модифицированную известную или неизвестную закрытую ОС. В этом случае нет гарантий корректной интерпретации изучаемого кода по вызовам известных библиотечных функций, поскольку этих функций либо нет, либо они сами являются предметом анализа.

Следует еще упомянуть, что исследуемый код, как правило, не является «замкнутым». Например, вредоносное ПО обычно взаимодействует с управляющими серверами, от которых получает команды, код «полезной» нагрузки, а также передает на эти сервера данные со скомпрометированных компьютеров.

Все эти, а также и другие особенности различных классов программ учитывались при разработке подхода к анализу. Была поставлена цель создать максимально общий подход, подходящий для разных процессорных архитектур, разных типов устройств и разных классов программ. Такие требования привели к тому, что разработанный подход способен довольствоваться одной только семантикой выполняющихся машинных команд (см. рис. 2): происходит постепенное повышение уровня представления до уровня, когда восстановленный алгоритм может быть передан для последующего анализа прикладному специалисту, не владеющему знаниями об особенностях реализации. Типовым выходным представлением является двудольный граф, где один класс вершин – обрабатываемые данные, второй класс – функции (блоки кода) эту обработку выполняющие.

Максимально низкий начальный уровень анализа – главное отличие предложенного подхода от известных аналогов, например, BitBlaze [6] или DroidScope [7], где предметом анализа выступают трассы, содержащие не только машинные команды, либо анализ ограничен адресным пространством только одного процесса. При полносистемном анализе в трассирующей виртуальной машине, как правило, разработанной на основе QEMU [7, 8], работает зафиксированная версия известной ОС, и знание о ее внутреннем устройстве активно используется.

Рис. 2. Различные уровни представления и диапазоны применения методов анализа.

Платой за общность является необходимость выполнять предварительное повышение уровня представления (рис. 3), заканчивающееся восстановлением потоков данных и управления. Метод предварительного повышения не требует каких-либо априорных знаний об устройстве ОС, работающей в анализируемой системе, и пригоден для большинства процессорных архитектур общего назначения. Более того, метод применим в случае модифицированной версии известной ОС, и даже в случае анализа кода неизвестной ОС.

Устранение артефактов требуется вследствие особенностей применяемых средств трассировки. Примерами дефектов могут служить отсутствие части строковых инструкций (инструкции с префиксом REP) для архитектуры x86 и неверное указание адресов инструкций из слотов задержки для архитектуры MIPS64.

Первичная разметка трассы выделяет в ней команды, относящиеся к различным процессам, нитям и зонам. Под зоной понимается период действия адресного пространства виртуальной памяти. Отображение адресов в современных процессорных архитектурах поддерживается аппаратно и не всегда соотносится с понятием процесса: например, ОС при копировании содержимого буфера памяти между различными процессами переключает адресные пространства.

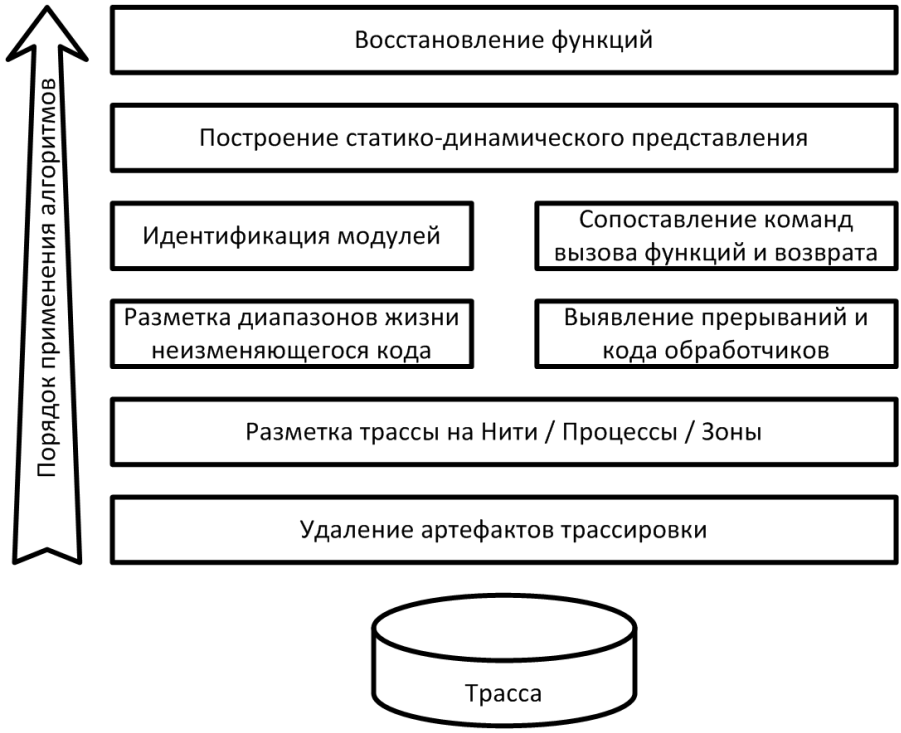


Рис. 3. Последовательное повышение уровня представления трассы.

Следующие четыре алгоритма разбиваются на две пары, которые могут выполняться параллельно.

Восстановление точек возникновения программных и аппаратных прерываний, исключений с выявлением диапазона шагов трассы, в которых выполнялся код обработчика прерывания.

После выявления обработчиков прерываний в трассе происходит сопоставление машинных команд вызовов функций и соответствующих возвратов. Одним из результатов этого сопоставления является то, что для каждой нити восстанавливается стек вызовов для каждого момента времени.

Крайне важным алгоритмом является выявление диапазонов времени, когда не происходило никаких модификаций кода. Различение кода и данных в общем случае – не решаемая задача. В данном случае различение происходит на основе трассы: поддерживаются три множества адресов памяти, используемых для чтения данных, записи, и извлечения выполняемых команд. Выявление факта модификации исполняемого кода приводит к созданию

нового поколения, характеризуемого номером. Для каждого поколения определен период жизни в терминах номеров шагов трассы.

В периодах постоянства кода можно выполнять поиск загруженных в память и выполнявшихся модулей, что впоследствии позволяет заменять адреса памяти символьными метками, извлеченными из бинарных файлов соответствующих модулей.

Наиболее важный этап – построение статико-динамического представления. Для каждого поколения кода строится граф, описывающий поток управления. Среди ребер выделяются ребра вызова функций и возврата в них, ребра, передающие управление между различными поколениями.

На основе полученного статико-динамического представления происходит выявление в коде функций. Следует отметить, что на практике регулярно встречается весьма неожиданное устройство тел функций. Например, точка входа может быть расположена после команд возврата, несколько функций могут иметь общие фрагменты кода и т.п.

После применения перечисленных алгоритмов анализа, большая часть которых работает в полностью автономном режиме, пользователь имеет возможность изучать построенное высокоуровневое представление, извлекать из него слайсы на основе анализа зависимостей по данным и управлению, формально описывать восстановленные функции и т.п. Начиная с уровня потоков данных и управления, появляется возможность проводить анализ (дальнейшее повышение уровня представления) полностью изолировав все особенности целевой процессорной архитектуры. Таким образом, добавление поддержки новой архитектуры сводится к разработке декодера и модификации части этапов предварительной обработки.

3. Выделение кода алгоритма

Базовой задачей при восстановлении алгоритма является выделение относящегося к нему кода. Так как любой алгоритм является последовательностью действий по преобразованию входных данных в выходные, для выделения применяются алгоритмы слайсинга (построения срезов) [9]. В общем виде на выходе выделения кода алгоритма должен быть получен подграф графа потока управления программы, в котором оставлены только те ребра, базовые блоки и команды в них, которые относятся к реализации алгоритма.

3.1. Выделение кода алгоритма по трассе

Для определения шагов трассы, относящихся к алгоритму, выполняется динамический слайсинг: либо прямой по входным данным, либо обратный по выходным данным, либо строится чоп как пересечение прямого и обратного слайсов. Входные и выходные данные должны быть заданы пользователем. Задача идентификации входов и выходов алгоритма частично

автоматизируется при помощи различных компонентов среды анализа, рассмотрение которых находится за рамками данной статьи.

Проведение слайсинга по трассе требует последовательного рассмотрения зависимостей в машинных командах и поддержания множества помеченных регистров и ячеек памяти. Один из способов организации такого слайсинга был предложен в [10].

Основная проблема динамического анализа – потенциальная неполнота покрытия программы – оказывает свое серьезное влияние на слайсинг по трассе. В связи с этим важно было решить задачи объединения результатов слайсинга по нескольким трассам и подготовки входных данных для снятия трасс, гарантированно улучшающих покрытие кода алгоритма. Первая задача решается на базе восстанавливаемого из трассы межпроцедурного графа потока управления программы: по результатам выделения алгоритма в каждой трассе помечаются отдельные ребра, базовые блоки и команды в них. В итоге помеченные хотя бы один раз элементы графа считаются относящимися к алгоритму.

В некоторых случаях, кроме того, требуется проведение статического слайсинга как обеспечивающего консервативную полноту выделения кода алгоритма. Кроме того, разработанный метод анализа позволяет получать дополнительные базовые блоки и ребра не только из трасс, но и в результате взаимодействия с дизассемблером IDA Pro или симулятором QEMU. В этих случаях подход к описанию машинных команд в виде списков зависимостей, предложенный в [10], не может быть применен: списки зависимостей строятся не для команды вообще, а для команды в данной точке трассы. Тем самым рассматриваются конкретные адреса ячеек памяти и для команд с нетривиальным внутренним потоком управления конкретный вид данного потока. В то же время в статическом слайсинге необходимо рассматривать всю совокупность возможных зависимостей, обусловленных выполнением команды.

Для того чтобы задачи проведения статического слайсинга и подбора входных данных для улучшения покрытия могли быть решены в общем виде для всех целевых процессорных архитектур, в рамках среды анализа было разработано промежуточное представление, удобное для описания поведения машинных команд. Представление обеспечивает возможность отслеживания потоков данных и выполнения компиляторных преобразований (таких как продвижение и свертка констант), что необходимо для корректного рассмотрения ячеек памяти при статическом слайсинге и не может быть достигнуто без знания операционной семантики операций.

3.2. Промежуточное представление Pivot

Промежуточное представление Pivot [11] обеспечивает возможность единообразного описания операционной семантики машинных команд, что

позволяет абстрагироваться от особенностей набора команд целевой машины в различных видах анализа потоков данных и семантики программы.

В представлении Pivot каждая машинная команда представлена в виде последовательности простых операторов, которые взаимодействуют с тремя видами переменных:

- временными переменными, находящимися в форме с единичным статическим присваиванием (SSA) и локальными в рамках набора операторов одной машинной команды;
- элементами адресных пространств, единообразно описывающими регистры и диапазоны памяти в виде троек (S, a, s), где S – адресное пространство, a – адрес в нем, а s – размер переменной;
- битами слова состояния Pivot, такими как флаги переноса, переполнения, нуля, знака и т.п.

Основными операторами являются:

- оператор инициализации временной переменной константным значением;
- оператор применения операции;
- оператор ветвления, в т.ч. условного в зависимости от битов слова состояния;
- операторы загрузки и выгрузки, выполняющие пересылки между временными переменными и элементами адресных пространств.

Каждая машина может иметь свой набор операций, однако на практике многие операции оказываются общими для всех машин (арифметико-логические операции, операции с плавающей точкой и т.д.). Операция принимает в качестве параметров набор временных переменных и формирует результат также во временной переменной. Кроме того, операция может читать и записывать некоторые биты слова состояния. Для каждой операции набор таких битов фиксирован и не зависит от конкретных значений параметров. В сочетании с минимальным набором операторов и SSA-формой для временных переменных это проектировочное решение позволило соблюсти баланс между простотой анализа потоков управления и данных поверх представления Pivot с одной стороны и его относительной компактности (особенно для архитектур с большим количеством побочных эффектов, таких как x86) с другой стороны. Одновременное достижение этих свойств является отличительной особенностью представления Pivot по сравнению с решающими близкие задачи промежуточными представлениями в других средах анализа бинарного кода, в частности Vine [6] и BAP [12].

Описания операций, адресных пространств и операционной семантики команд каждой машины в компактном виде представляются в виде бинарных файлов, называемых архивами машин. Эти файлы подгружаются средой анализа при работе с трассами соответствующих машин.

3.3. Применение промежуточного представления Pivot

Наличие единого промежуточного представления позволило решить в рамках среды следующие задачи.

1. Выполнение инструментируемой конкретной интерпретации машинных инструкций, при помощи чего может быть реализован, в том числе, слайсинг по трассе. При этом одновременно вычисляются конкретные адреса, необходимые для корректной обработки косвенной адресации, и поддерживаются множества помеченных элементов в каждом адресном пространстве.
2. Проведение статического слайсинга на межпроцедурном графе потока управления для более полного выделения алгоритма. Свойства промежуточного представления позволяют применить известные алгоритмы статического слайсинга без существенных изменений.
3. Проведение символьных вычислений, что позволяет для не реализованного в трассе перехода подбирать входные данные, обеспечивающие его выполнение. При этом строится система уравнений, состоящая из условий вдоль соответствующего пути в программе: рассматриваются отдельные операторы, составляющие инструкции, влияющие на поток управления, и переводятся в уравнения. Полученная система подается на вход SMT-решателю Z3, а результат работы последнего отображается обратно на модель адресных пространств в начальной точке. На основе этих данных может быть снята новая трасса, улучшающая покрытие кода программы.

4. Восстановление форматов данных

Внутри программы данные хранятся и передаются в виде переменных, имеющих некоторые типы в системе типов языка программирования, на котором написана программа. В процессе компиляции операции с переменными этих типов преобразуются в конструкции на языке ассемблера соответствующей процессорной архитектуры. Задача восстановления типов, то есть получение высокоуровневых типов данных отдельных переменных по ассемблерной программе, является частью задачи декомпиляции, равно как и идентификация самих переменных. Подходы к решению этих задач зависят как от процессорной архитектуры, так и от особенностей компилятора. С другой стороны, в ходе обмена данными с другими системами программа может воспользоваться либо одним из видов внешней памяти (например, жестким диском), либо каналом связи, таким как сеть Ethernet. При этом данные принимают форму файлов и сетевых пакетов, формат которых должен поддерживаться другими системами и, таким образом, в общем случае не связан с особенностями конкретной архитектуры и компилятора. Например, поддержка одних и тех же протоколов стека TCP/IP реализуется в самых разных ОС и архитектурах, а формат исполняемых файлов PE может

применяться в процессорных архитектурах x86, MIPS, ARM, PowerPC и многих других.

В работах [13, 14] приведено описание того, что обычно вкладывается в понятие «формат данных»:

- границы отдельных полей базовых типов в сообщении (в случае работы с файлом весь файл рассматривается как сообщение);
- группировка полей базовых типов в поля-последовательности и поля-записи с учетом вложенности;
- семантика отдельных полей, в частности поля длины; поля-разделители, определяющие длины последовательностей; поля-указатели, хранящие смещения других полей; ключевые поля, которые могут содержать предопределенный набор констант, специфичных для данного формата; поля-флаги, состоящие из групп отдельных битов;
- дополнительная информация для полей специальных типов: возможные значения ключевых полей (константы протокола); связь между полями-указателями и полями, на которые они указывают; связь между полями-последовательностями и полями, задающими их длину.

Особенность большинства современных вычислительных систем заключается в том, что программа может обрабатывать только данные, которые лежат в оперативной памяти и на регистрах. Поэтому в процессе обработки любому объекту, такому как файл или сетевое сообщение, соответствует буфер в оперативной памяти, в котором в некоторый момент времени находятся данные этого объекта. Следовательно, задачу восстановления формата данных можно свести к восстановлению формата буфера памяти, содержащего этот объект. Вообще говоря, может не существовать единого буфера, содержащего весь объект в один момент времени. Сложности, возникающие в таких ситуациях, будут описаны ниже. Далее рассматривается частный случай, когда все сетевое сообщение или файл считываются за одну операцию чтения и, соответственно, все данные исследуемого объекта попадают в один буфер в памяти. После того, как буфер определен, схема анализа может быть представлена в виде последовательности следующих шагов.

- Выделение алгоритма обработки данных, содержащихся в буфере, с привязкой конкретных частей буфера к машинным командам, в которых они обрабатываются.
- Восстановление плоского формата сообщения: определение границ отдельных полей на основе анализа отдельных команд доступа к данным буфера.

- Восстановление иерархического формата сообщения путем группировки отдельных полей в записи и последовательности на основе структурного анализа графа потока управления выделенного алгоритма.
- Восстановление семантической информации полей.

При выделении алгоритма отличают случаи разбора полученных извне программы данных и генерации данных для отправки. В первом случае для выделения алгоритма применяется классический анализ распространения помеченных данных [15]. Во втором случае может быть использован, например, алгоритм, описанный в [16].

Восстановление полей производится в соответствии с алгоритмом анализа диапазонов, использующихся отдельными инструкциями при доступе к буферу [17]. Для разрешения возникающих конфликтов применяется жадный алгоритм.

При группировке полей в виде записей и последовательностей используется эвристическое предположение о том, что поля, доступ к которым осуществляется в цикле, монотонно и непрерывно, от меньших смещений к большим, образуют последовательность, причем на каждой итерации цикла обрабатываются поля, образующие одну запись. Для осуществления этого анализа предварительно производится выделение циклов (в том числе развернутых) в CFG и поиск границ их выполнения в трассе.

Идея *выведения семантики* основана на алгоритме выведения типов в задаче декомпиляции. Вводится понятие *источник семантики*, то есть место в программе, в котором семантика используемых данных точно известна, после чего восстанавливаются зависимости по данным между этими источниками и данными в буфере. Список источников семантики приведен в работе [18]. Его можно дополнить специфическими конструкциями в CFG программы, например условными переходами, прерывающими выполнение цикла по условию, по которым можно определять поля длины и разделители.

Дополнительными трудностями, возникающими при восстановлении форматов являются:

- неполное восстановление формата из-за ограничений динамического анализа;
- шифрование/дешифрование данных;
- работа программы с несколькими источниками/потребителями и форматами данных;
- поступление данных по частям (считывание файла, сообщение из двух пакетов).

Для преодоления неполноты восстановления используется объединение форматов, полученных при анализе нескольких трасс, с помощью модифицированного алгоритма Нидлмана-Вунша [19], изначально применявшегося в биоинформатике. В ходе этого алгоритма выполняется обобщение каждого формата, то есть получение частичной грамматики, и

выравнивание грамматик, то есть сопоставление элементов двух грамматик и их слияние.

Для обработки шифрованного трафика используются *модели функций* шифрования и дешифрования. При выделении отдельных источников и потребителей данных кроме моделей функций ввода-вывода используется понятие *идентификатор источника*. Для обработки данных, поступающих по частям, вводится понятие *виртуальный буфер* и анализ параметров функций ввода-вывода. Более подробно подход к обработке шифрованного трафика описан в [14].

Одной из целей восстановления форматов данных, помимо описания алгоритма, является автоматическая генерация разборщиков сетевых сообщений для систем DPI, IDS и IPS. Примером системы разбора сетевых сообщений общего назначения с открытым исходным кодом является Wireshark. Разборщики в данной системе пишутся на языке Си, что делает затруднительной их автоматическую генерацию. Поэтому для реализации прототипа системы автоматического создания разборщиков по восстановленным форматам данных была выбрана система IDS Bro, для которой доступен компилятор binpac, создающий библиотеку разбора по описанию формата в виде грамматики. Пример с восстановлением формата и генерацией разборщика для него приводится в разделе 6.

Дальнейшим развитием рассмотренного подхода в случае анализа сетевых сообщений является автоматическое получение автомата протокола по коду реализации, что позволит решить общую задачу восстановления протоколов. На данный момент известно несколько работ в этом направлении [12, 20, 21].

5. Архитектура среды анализа

При проектировании архитектуры принимаемые решения базировались на требованиях, обусловленных исследовательским характером среды и ее достаточно быстрым развитием, большим объемом анализируемых трасс и вспомогательных данных, а также практикой использования среды. Среди таких требований можно выделить три основных.

1. Среда должна быть построена на модульной архитектуре: добавление поддержки новой процессорной архитектуры, формата хранения, алгоритма анализа или компонента отображения информации не должно требовать изменения инфраструктуры.
2. Среда должна в полной мере задействовать возможности современной аппаратуры: использование нескольких вычислительных ядер должно быть поддержано как на уровне архитектуры в целом, так и на уровне отдельных алгоритмов анализа.
3. Среда должна проводить анализ в фоновом режиме, позволяя пользователю продолжать интерактивную работу с графическим интерфейсом. Интерфейс должен сохранять высокий уровень

отзывчивости вне зависимости от вычислительной нагрузки, обусловленной выполняемым анализом.

5.1. Архитектура потоков выполнения

При работе среда использует несколько потоков выполнения, распределяемых между вычислительными ядрами (рис. 4). Два потока с повышенным приоритетом существуют постоянно и выполняют два цикла обработки сообщений: первый поток обслуживает графический интерфейс, а второй – неграфические объекты среды, доставляя сообщения от одних к другим. Оба эти потока не нагружаются вычислительными задачами, за счет чего обеспечивается должный уровень отзывчивости.

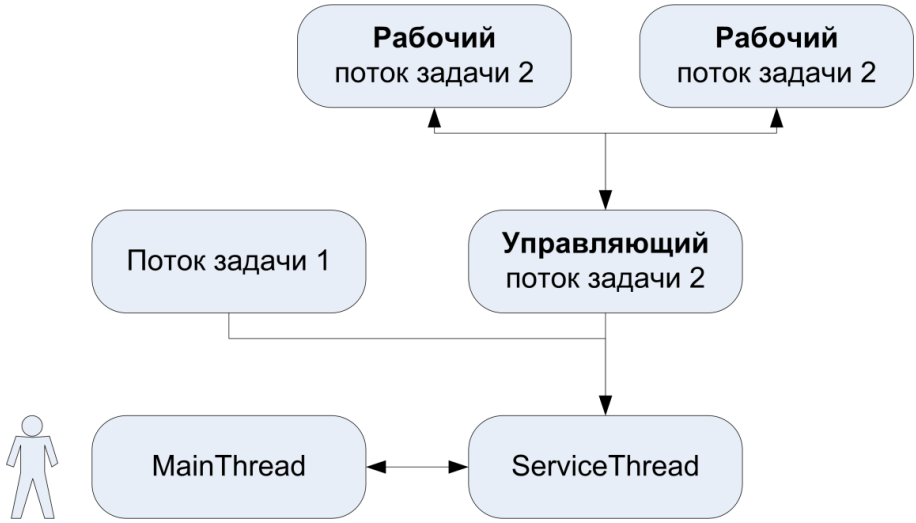


Рис. 4 – Потоки выполнения.

Алгоритмы анализа работают в дополнительных потоках, создаваемых по требованию. При запуске вычислительной задачи для нее создается управляющий поток. В зависимости от природы задачи вычисления могут проводиться либо в этом единственном потоке, либо могут быть созданы дополнительные рабочие потоки. В последнем случае управляющий поток организует взаимодействие рабочих потоков и формирование общего результата.

5.2. Рабочее пространство

Рабочее пространство организуется в виде дерева, пример которого приведен на рис. 5, со следующими вершинами:

- вершина рабочего пространства: корень дерева, отвечающий исследуемой программной системе в целом;
- вершина гнезда трасс, соответствующая набору трасс, снятых с одного и того же начального состояния машины, и хранящая результаты статического анализа этого набора;
- вершина трассы, хранящая результаты ее динамического анализа.

При этом вершины трасс могут образовывать поддеревья произвольной высоты за счет возможности работы с подтрассами – подпоследовательностями шагов в родительской трассе или подтрассе.

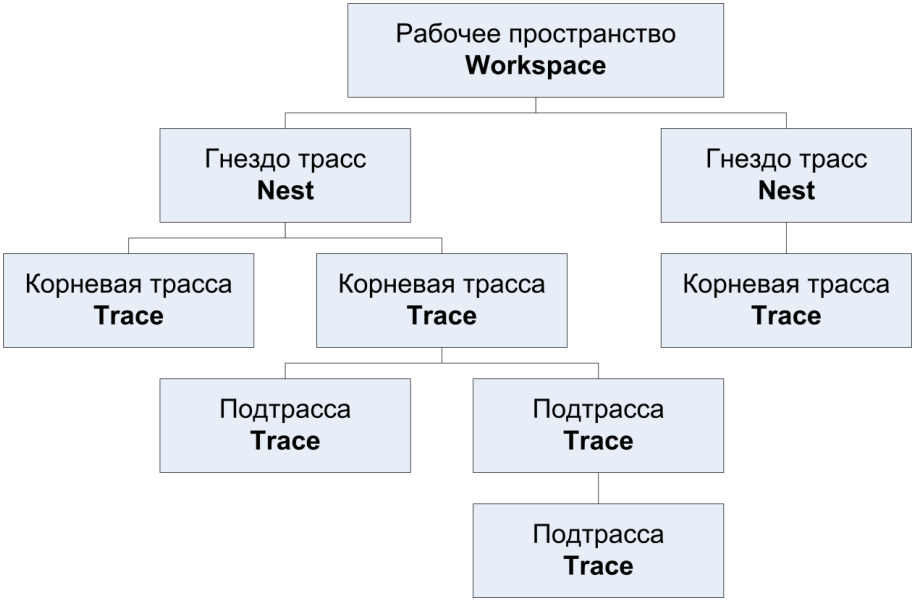


Рис. 5. Дерево рабочего пространства.

Вершины в дереве рабочего пространства называются *контекстами* и предоставляют возможность единообразной работы с данными, соответствующими различным уровням анализа. В рамках каждого контекста поддерживается база данных типа ключ-значение, файловое хранилище и реестр ресурсов, более подробно описываемый в следующем подразделе.

Гибкая структура дерева рабочего пространства позволяет совместно применять как динамические, так и статические методы анализа, проводить анализ по нескольким трассам, а также агрегировать в рамках одного рабочего пространства несколько физически обособленных компонентов исследуемой системы. Так, возможно создать, например, рабочее пространство, где одному

гнезду будет соответствовать клиент, а другому – сервер клиент-серверного приложения. При этом среда анализа позволяет гнездам одного рабочего пространства относиться к разным процессорным архитектурам: например, клиентское приложение может работать на мобильной платформе с процессором ARM, а серверная часть – на машине x86.

5.3. Управление ресурсами и задачами

Для управления набором построенных алгоритмами анализа данных используются *реестры ресурсов* в каждом из контекстов дерева рабочего пространства. *Ресурсом* называется объект, соответствующий некоторому виду данных и обладающий определенным интерфейсом, например, база данных процессов, потоков выполнения и зон в гнезде или разметка вызовов в трассе.

Объекты-ресурсы снабжены счетчиками ссылок, что позволяет отслеживать количество их пользователей и автоматически управлять их освобождением. Кроме того, ко всем ресурсам предъявляется требование потоковой безопасности, и каждый из них снабжается синхронизационным примитивом типа read-write lock, который захватывается всякий раз при вызове публичного метода ресурса (рис. 6).

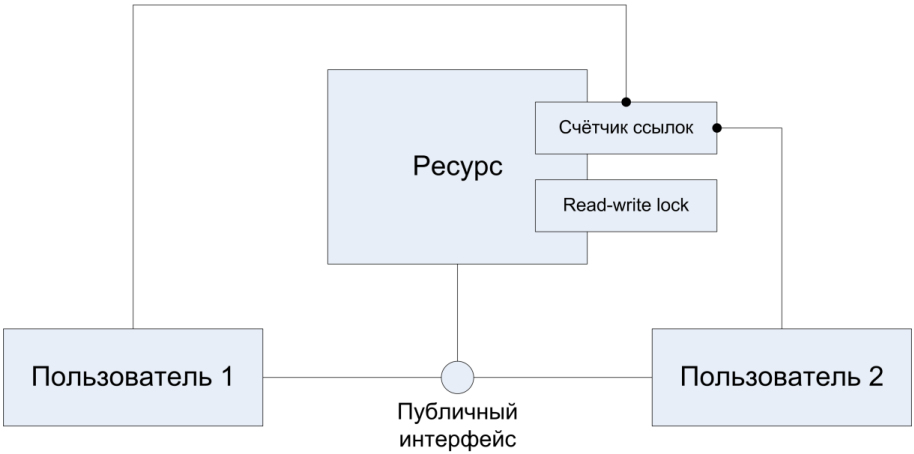


Рис. 6 – Ресурс и его пользователи.

Ресурсы могут в произвольный момент быть *отозваны*. Это означает, что данные, предоставляемые этим ресурсом, более не являются актуальными. При отзыве ресурса необходимо обеспечить решение двух задач. Во-первых, все текущие пользователи ресурса должны быть уведомлены о том, что данный ресурс больше не пригоден к использованию. Во-вторых, необходимо

освободить объект ресурса (здесь важно напомнить о том, что среда работает в большом количестве потоков выполнения, не во все из которых возможно доставлять сообщения асинхронно).

Для решения этих задач реализован следующий механизм. При осуществлении каждого доступа к ресурсу, как было указано выше, захватывается синхронизационный примитив. Непосредственно перед таким захватом среда автоматически проверяет признак, отозван ли данный ресурс. Если да, то выбрасывается исключение, обрабатываемое вызывающим: уменьшается счетчик ссылок ресурса, а вызывающий либо заканчивает текущую обработку с ошибкой, либо действует иным уместным способом. Когда все ссылки на отозванный ресурс будут удалены, среда автоматически освободит его.

Реестры ресурсов, привязанные ко всем контекстам в дереве рабочего пространства, устанавливают соответствие между строковыми идентификаторами интерфейсов (*точками монтирования*) и ресурсами, реализующими эти интерфейсы (рис. 7). Один ресурс может реализовывать несколько интерфейс, и один интерфейс может быть реализован в нескольких ресурсах.

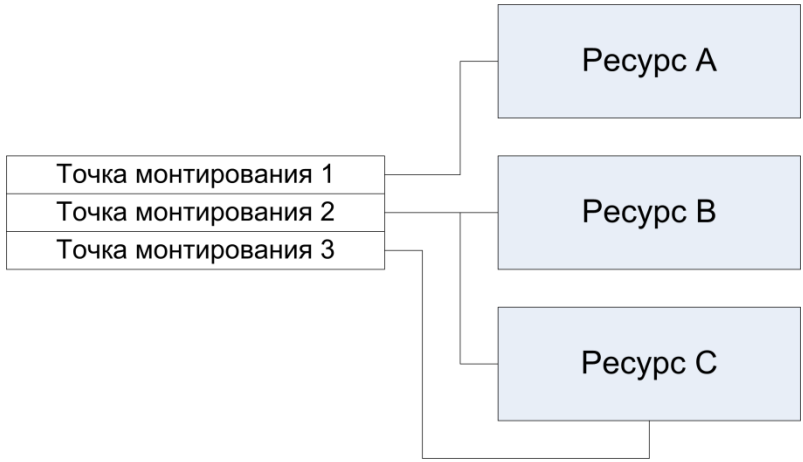


Рис. 7 – Реестр ресурсов.

Построение данных для ресурсов, как и все остальные виды длительных обработок, в среде реализуются в виде *задач*. Задача представляет собой обособленный алгоритм, работающий в отдельном потоке выполнения с возможностью порождения дополнительных рабочих потоков. Задача при запуске связывается с контекстом дерева рабочего пространства, в котором она работает. Контекст поддерживает список работающих в нем задач и не

может быть закрыт или удален до тех пор, пока все они не завершат работу. Задачи обладают механизмом для печати информационных сообщений и передачи информации о текущем прогрессе и состоянии, что отображается в графическом интерфейсе среды. По желанию пользователя задача может быть приостановлена или отменена.

Задача получает на вход набор параметров и после окончания работы формирует результат. Все параметры и результаты представляются единообразно в виде объектов *параметров задачи*. Кроме того, в процессе работы задача может асинхронно выдавать промежуточные результаты. Например, алгоритм поиска в трассе всех обращений к ячейке памяти может выдавать шаги по мере их нахождения, что позволяет пользователю начать анализ результатов, не дожидаясь полного завершения задачи.

5.4. Модули расширения

Модули расширения среды анализа представляют собой динамически подгружаемые библиотеки и разбиты на следующие два класса.

1. Инфраструктурные модули расширения (ИМР) предоставляют реализации базовых интерфейсов среды. ИМР обеспечивают поддержку процессорных архитектур, форматов исполнимых файлов, форматов трасс, интеграцию с трассирующими симуляторами и т.п.
2. Функциональные модули расширения (ФМР) обогащают среду поддержкой новых видов анализа, а также связанными с этим компонентами хранения и отображения данных. ФМР реагируют на основные события среды, в первую очередь события загрузки или выгрузки одного из контекстов дерева рабочего пространства. В ответ на такое событие ФМР добавляет или удаляет предоставляемые ресурсы в реестре этого контекста. Для хранения данных используются база данных типа ключ-значение и файловое хранилище контекста. Такая схема позволяет реализовывать произвольные виды анализа и обеспечивает возможность взаимодействия между модулями расширения без необходимости внесения изменений в ядро среды.

6. Примеры применения среды

6.1. Пример выделения и построения модели алгоритма

Данный пример показывает работу методов выделения алгоритмов. Был выбран алгоритм генерации лицензии в программе управления лицензиями. Данная программа получала на вход файл, содержащий идентификатор машины, а на выходе генерировала файл, содержащий лицензию в бинарном виде. Для начала анализа была снята с процесса генерации выходного файла. В полученной трассе были восстановлены модули и подключена символьная информация. Далее был найден вызов функции WriteFile, соответствующий выводу лицензии в файл. Были восстановлены значения параметров вызова и,

соответственно, данные лицензии. По характерным адресам был определена точка вызова функции main. Для извлечения алгоритма генерации ключа был применен обратный слайсинг с учетом адресных зависимостей для буфера лицензии, от точки вызова функции WriteFile до точки вызова функции main. Полученный слайс был далее отфильтрован для удаления кода ядра ОС. Результат был представлен в виде дерева вызовов – последовательности вызовов функций, реализующих алгоритм, с учетом их вложенности. Результат представлен на рис. 8.

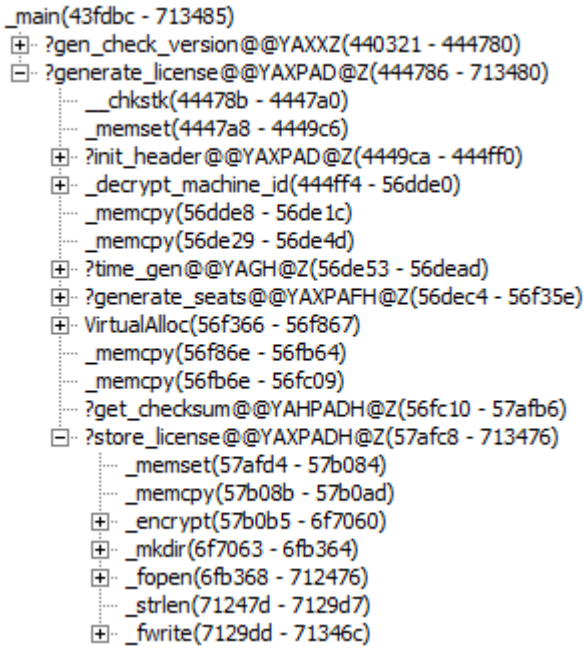


Рис. 8 – Последовательность вызовов функций, реализующих алгоритм, с учетом их вложенности.

Как видно из рисунка, алгоритм состоит из двух основных частей – функций gen_check_version для извлечения из реестра версии программы и проверки ее соответствия версии кода и generate_license, отвечающей за непосредственную генерацию лицензии. Для описания этих частей были проанализированы все перечисленные функции, определены и описаны их параметры. В частности, функция get_checksum получает на вход два параметра – адрес (addr) и размер (size) буфера данных, по которому считается контрольная сумма. Они передаются через стек и могут быть описаны, как Mem(ESP + 4, 4) и Mem(ESP +8, 4), где Mem(*адрес*, *размер*) – набор ячеек памяти, начиная с адреса *адрес*

размером *размер*, а ESP – значение указателя стека в момент начала исполнения функции. В ячейке Mem(ESP, 4) лежит адрес возврата. В свою очередь, сам буфер данных (buf) может быть описан как Mem(addr, size). Выходной параметр checksum является целым числом и в соответствии с используемым соглашением о связях cdecl находится в регистре EAX.

По результатам анализа были построены графы зависимостей на уровне вызовов функций для каждой из двух частей. Граф для основного алгоритма генерации лицензии приведен на рис. 9.

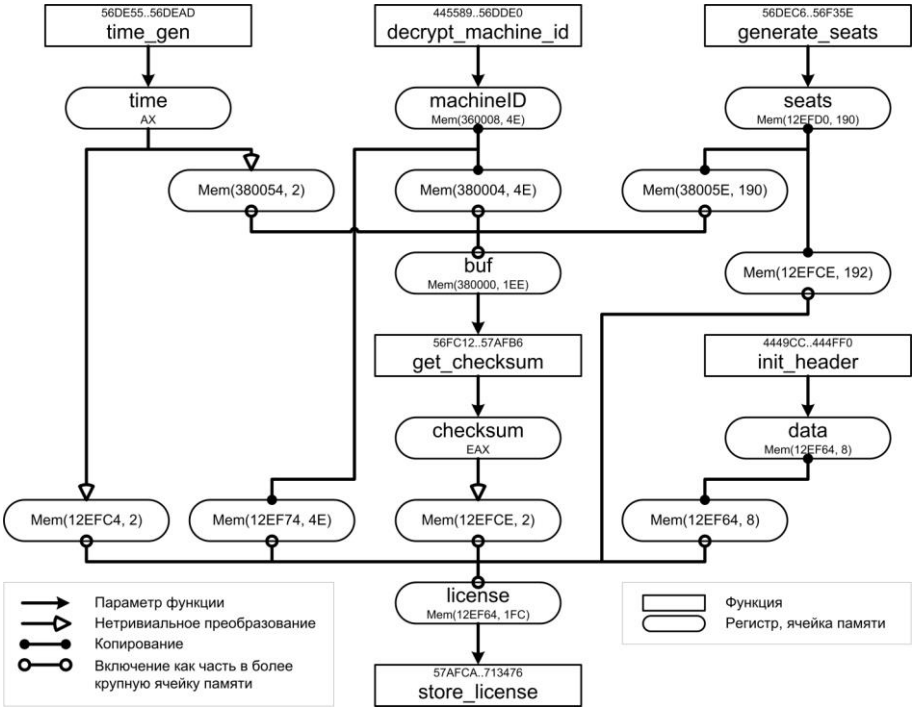


Рис. 9 – Схема алгоритма генерации лицензии.

Как видно из двух рисунков, в лицензию в зашифрованном виде (функция encrypt) входят следующие данные, вычисляемые соответствующими функциями:

- идентификатор машины (функция decrypt_machine_id, берущая данные из входного файла);
- время создания лицензии (функция time_gen);
- количество рабочих мест (функция generate_seats);

- контрольная сумма по перечисленным выше данным (функция get_checksum);
- заголовок лицензии (функция init_header).

6.2. Пример восстановления формата данных

В качестве примера для демонстрации восстановления форматов данных выбран протокол DNS, описание которого содержится в документах RFC 1350 и RFC 3425. Описание формата приведено кратко на рис. 10 (а). Выбор обусловлен тем, что этот формат имеет достаточно сложную структуру и большое количество полей с различной семантикой. Анализ протокола DNS был выполнен по программе nslookup, входящей в стандартную поставку Windows 2000. Трасса снималась в процессе отправки прямого DNS-запроса до получения ответа от DNS-сервера, его разбора и вывода на экран в текстовом виде. Размер буфера сообщения составил 46 байт, время анализа – 3 с. Фрагмент восстановленного формата приведен на рис. 10 (б). По восстановленному формату была получена частичная грамматика, листинг которой приведен на рис. 11 для трансляции формата в библиотеку разборки пакетов с помощью компилятора binpac. На вход полученной программе-разборщику был подан набор сообщений-ответов на прямые DNS-запросы, отличные от исходного, которые были полностью и без ошибок разобраны. Для дополнения грамматики путем анализа нескольких трасс, помимо прямого DNS запроса были проанализированы также обратный и SOA DNS запросы, что позволило расширить применимость разборщика и на эти варианты сообщений.

Рис. 10 – (а) Формат DNS сообщения согласно RFC 1350, (б) восстановленный формат ответа на прямой DNS-запрос.

```
0:buffer;
{
      1:field:uint16:simple:empty;
      2:field:uint8:flag(1+1+1+4+1):empty;
      3:field:uint8:flag(4+2+1+1):empty;
      4:field:uint16:length:empty;
      5:field:uint16:length:empty;
      6:field:uint16:simple:empty;
      7:field:uint16:simple:empty;
      8:sequence:?:10:4;
      9:sequence:?:11:5;
}
```

Рис. 11 – Описание формата для генерации разборщика компилятором binpac.

7. Обзор близких работ

По открытым публикациям хорошо известны работы, в которых предлагались расширяемые среды анализа бинарного кода. Большинство публикаций с результатами высокого уровня принадлежат исследовательским коллективам ведущих университетов США.

В 2005 году была опубликована работа [22] (Университет Висконсина), в которой представлена платформа CodeSurfer/x86, поддерживающая статический анализ бинарного кода. Для более точного восстановления потока управления авторами был предложен алгоритм VSA [23], строящий верхнее приближение множества значений для регистров и ячеек памяти во всех точках программы. Работа с восстановленным графом потока управления производилась инструментами системы CodeSurfer, рассчитанной на статический анализ Си-программ.

В проекте BitBlaze [6] (Университет Беркли) были получены наиболее близкие результаты по сравнению с теми, что представлены в данной статье. В рамках данного проекта была разработана инфраструктура (промежуточное представление Vine [6], эмулятор TEMU [8]), облегчающая анализ бинарного кода, а также значительное количество программных инструментов, этой инфраструктурой пользующихся. Основная направленность работ – поддержка восстановления алгоритмов из бинарного кода, в том числе вредоносного. Авторы BitBlaze предлагают получать по трассам выполнения статическое представление [24] и повышать его уровень до блок-схем [25].

Блок-схемы представляют собой двудольные графы, совмещающие в себе потоки данных, управления и отношение производитель-потребитель. Помимо того, коллективом BitBlaze были разработаны инструменты поиска ошибок в бинарном коде [26, 27] на основе инфраструктурных компонент, реализующих символьную интерпретацию.

Ответвлением проекта BitBlaze стал проект BAP, Binary Analysis Platform [12], ведущийся в Университете Карнеги-Меллон. Базовым компонентом инфраструктуры является промежуточное представление BIL, и средства трансляции в него. Основным направлением работ является автоматический поиск ошибок на основе символьной интерпретации [28].

С развитием рынка мобильных устройств возник интерес к анализу бинарного кода прошивок и прикладных программ смартфонов. Система DroidScope [7] переназначена для восстановления алгоритмов реализованных на платформе Android. Модифицированная версия эмулятора QEMU используется для сбора трасс различного уровня: машинные команды (ARM или x86), события ОС, команды байт-кода Dalvik. Совмещение трасс и автоматический анализ помеченных данных позволяют в итоге восстанавливать алгоритм в виде двудольного графа.

8. Заключение

В статье представлена среда анализа, позволяющая проводить исследования бинарного кода, опираясь только на знание о работе машинных команд целевой процессорной архитектуры. Это позволяет анализировать код широкого класса программ – прикладных, код драйверов и ядра ОС, код BIOS. Модульная архитектура среды поддерживает в настоящий момент такие архитектуры, как x86, x86-64 и ARM, и, начиная с уровня потоков данных и управления, может быть расширена на любую другую архитектуру без изменения самих алгоритмов анализа.

В составе среды реализованы методы восстановления алгоритмов и форматов данных, которые этими алгоритмами обрабатываются. Разработано промежуточное представление Pivot, используемое для статического слайсинга и организации символьных вычислений.

Архитектура среды, в рамках которой реализованы алгоритмы анализа, позволяют эффективно задействовать ресурсы современных многоядерных рабочих станций.

Дальнейшие работы в данной области предполагают развитие методов автоматического выявления ошибок и их последующей классификации.

Список литературы

[1]. Андрей Тихонов, Арутюн Аветисян, Вартан Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. №3, 2008. Стр. 66-71

[2]. А.И. Аветисян, В.А. Падарян, А.И. Гетьман, М.А. Соловьев. О некоторых методах повышения уровня представления при анализе защищенного бинарного кода. // Материалы XIX Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации», 2010. Стр. 97-98.
[3]. А.Ю.Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. // Труды Института системного программирования РАН, том 22, 2012 г. стр. 131-152. DOI: 10.15514/ISPRAS-2012-22-9.
[4]. Alexander Getman, Vartan Padaryan, and Mikhail Solovyev. Combined approach to solving problems in binary code analysis. // Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT'2013), pp. 295-297.
[5]. К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян. Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU. // Труды Института системного программирования РАН, том 22, 2012 г. Стр. 77-94. DOI: 10.15514/ISPRAS-2012-22-6.
[6]. Dawn Song, David Brumley, HengYin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. // International Conference on Information Systems Security, 2008, LNCS 5352, pp. 1-25.
[7]. Lok Kwong Yan and Heng Yin. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. // In Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA, pp. 29-29.
[8]. Heng Yin and Dawn Song. TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution. / EECS Department University of California, Berkeley, Technical Report No. UCB/EECS-2010-3, January 11, 2010, p. 14.
[9]. M. Harman, S. Danicic, Y. Sivagurunathan, D. Simpson. The next 700 slicing criteria. // Second UK Workshop on Program Comprehension, 1996.
[10]. Падарян В. А., Гетьман А. И., Соловьев М. А. Программная среда для динамического анализа бинарного кода. // Труды Института системного программирования РАН, том 16, 2009 г. Стр. 51-72.
[11]. Падарян В. А., Соловьев М. А., Кононов А. И. Моделирование операционной семантики машинных инструкций. // Программирование, № 3, 2011 г. Стр. 50-64.
[12]. David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: a binary analysis platform. // In Proceedings of the 23rd international conference on Computer aided verification (CAV'11), Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 463-469.
[13]. А. И. Гетьман, Ю. В. Маркин, В. А. Падарян, Е. И. Щетинин. Восстановление формата данных. // Труды Института системного программирования РАН, том 19, 2010 г. Стр. 195-214.
[14]. А. И. Аветисян, А. И. Гетьман. Восстановление структуры бинарных данных по трассам программ. Труды Института системного программирования РАН, том 22, 2012 г. Стр. 95-118. DOI: 10.15514/ISPRAS-2012-22-7.
[15]. J. Newsome, D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. // In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2005.
[16]. J. Caballero, P. Poosankam, C. Kreibich, D. Song. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. // In Proceedings of the 16th ACM conference on Computer and communications security (CCS), 2009, pp. 621-634.

[17]. W. Cui, M. Peinado, K. Chen, H. J. Wang, L. Irun-Briz. Tupni: automatic reverse engineering of input formats. // In CCS'08: Proceedings of the15th ACM conference on Computer and communications security, 2008.
[18]. Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. // In Proceedings of the Network and Distributed System Security Symposium, 2010.
[19]. S. B. Needleman and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. // Journal of Molecular Biology, 48(3):443–453, 1970.
[20]. Y. Wang, Z. Zhang , D. Yao, B. Qu, L. Guo. Inferring protocol state machine from network traces: a probabilistic approach. // In Proceeding of the 9th international conference on Applied cryptography and network security (ACNS), 2011, pp. 1-18.
[21]. P.M. Comparetti, G. Wondracek, C. Kruegel, E. Kirda. Prospex: Protocol Specification Extraction. // In Proceedings of the 30th IEEE Symposium on Security and Privacy, 2009, pp. 110-125.
[22]. Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86—A platform for analyzing x86 executables. // In Proceedings of the 14th international conference on Compiler Construction (CC'05), Springer-Verlag, Berlin, Heidelberg, pp. 250-254.
[23]. Gogul Balakrishnan and Thomas Reps. Analyzing Memory Accesses in x86 Executables. // In Proceedings of Compiler Construction, Springer-Verlag, New York, 2004, pp. 5-23.
[24]. Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. // In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, New York, USA, pp. 12-22.
[25]. Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant and Dawn Song. HI-CFG: Construction by Binary Analysis, and Application to Attack Polymorphism. // In Proceedings of 18th European Symposium on Research in Computer Security, Egham, UK, 2013. LNCS 8134, pp. 164-181.
[26]. Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. // In Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09). ACM, New York, USA, pp. 225-236.
[27]. Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babić, and Dawn Song. Input generation via decomposition and re-stitching: finding bugs in Malware. In Proceedings of the 17th ACM conference on Computer and communications security (CCS '10). ACM, New York, USA, pp. 413-425.
[28]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. // In Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12). IEEE Computer Society, Washington, USA, pp. 380-394.

Methods and software tools for combined binary code analysis ⋆

V. A. Padaryan, A. I. Getman, M. A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko
{ vartan, thorin, eyescream, bakulinm, helendile, korpse, il, ustas, spanasenko}@ispras.ru

Abstract. This paper presents methods and tools for binary code analysis that have been developed in ISP RAS and their applications in fields of algorithm and data format recovery. The analysis subject is executable code of various general purpose CPU architectures. The analysis is carried out in lack of source code, debug records, and without specific OS version requirements. The approach consists of collecting a detailed machine instruction level execution trace; method for successive presentation level increase; extraction of code belonging to the algorithm followed by structuring of both code and data formats it processes. Important results have been achieved: an intermediate representation has been developed, that allows for carrying out most of the preliminary processing tasks and algorithm code extraction without having to focus on specifics of a given machine; and a method and software tool have been developed for automated recovery of network message and file formats. The tools have been incorporated into a unified analysis platform that supports their combined use. The architecture behind the platform is also described in the paper. Examples of its application to real programs are given.

Keywords: binary code, static analysis, dynamic analysis, integrated software platform.

References

[1]. Tikhonov A.YU., Avetisyan A.I., Padaryan V.A., Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of exploring of an algorithm from binary code by dynamic analysis]. Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy. 2008, №3. pp. 66-71 (in Russian)

[2]. Avetisyan A.I., Padaryan V.A., Getman A.I., Solov'ev M.A. O nekotorykh metodakh povysheniya urovnya predstavleniya pri analize zashhishhennogo binarnogo koda [Some Approaches To Raising Representation Level In Analysis Of Protected Binary Code]. Materialy Obshherossijskoj nauchno-tekhnicheskoj konferentsii «Metody i tekhnicheskie sredstva obespecheniya bezopasnosti informatsii», 2010. pp. 97-98. (in Russian)

[3]. Tikhonov A.YU., Avetisyan A.I. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda. [Combined (static and dynamic) analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, 2012, pp. 131-152. DOI: 10.15514/ISPRAS-2012-22-9. (in Russian)

⋆ The paper is supported by RFBR grant 11-07-00450-a

[4]. Getman A.I., Padaryan V.A., Solov'ev M.A. Combined approach to solving problems in binary code analysis. Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT'2013), pp. 295-297.
[5]. Batuzov K.A., Dovgalyuk P., Koshelev V.K., Padaryan V.A. Dva sposoba organizatsii mekhanizma polnosistemnogo determinirovannogo vosproizvedeniya v simulyatore QEMU [Two Approaches To Full-System Deterministic Replay QEMU]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, 2012, pp. 77-94. DOI: 10.15514/ISPRAS-2012-22-6. (in Russian)
[6]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: A New Approach to Computer Security via Binary Analysis. International Conference on Information Systems Security, 2008, LNCS 5352, pp. 1-25.
[7]. Yan L.K., Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA, pp. 29-29.
[8]. Yin H., Song D. TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution. EECS Department University of California, Berkeley, Technical Report No. UCB/EECS-2010-3, January 11, 2010, p. 14.
[9]. Harman M., Danicic S., Sivagurunathan Y., Simpson D.. The next 700 slicing criteria. Second UK Workshop on Program Comprehension, 1996.
[10]. Padaryan V.A., Getman A.I., Solov'ev M.A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 16, 2009, pp. 51-72 (in Russian).
[11]. Padaryan V.A., Solov'ev M.A., Kononov A.I.. Simulation of operational semantics of machine instructions. Programming and Computer Software, May 2011, Volume 37, Issue 3, pp 161-170, DOI 10.1134/S0361768811030030
[12]. Brumley D., Jager I., Avgerinos T., Schwartz E. J. BAP: a binary analysis platform. Proceedings of the 23rd international conference on Computer aided verification (CAV'11), pp. 463-469.
[13]. Getman A.I., Markin YU.V., Padaryan V.A., Shhetinin E.I. Vosstanovlenie formata dannykh [Data format recovery]. Trudy ISP RAN [The Proceedings of ISP RAS], 2010, vol. 19, pp. 195-214 (in Russian)
[14]. Avetisyan A.I., Getman A.I. Vosstanovlenie struktury binarnykh dannykh po trassam program [Recovery the structure of binary data structures from program traces]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 95-118. DOI: 10.15514/ISPRAS-2012-22-7. (in Russian)
[15]. Newsome J., Song D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Proceedings of the Network and Distributed System Security Symposium (NDSS), 2005.
[16]. Caballero J., Poosankam P., Kreibich C., Song D. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. Proceedings of the 16th ACM conference on Computer and communications security (CCS), 2009, pp. 621-634.
[17]. Cui W., Peinado M., Chen K., Wang H. J., Irun-Briz L. Tupni: automatic reverse engineering of input formats. Proceedings of the15th ACM conference on Computer and communications security, 2008.
[18]. Lin Z., Zhang X., Xu D. Automatic reverse engineering of data structures from binary execution. Proceedings of the Network and Distributed System Security Symposium, 2010.

[19]. Needleman S. B., Wunsch C. D. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology, 48(3):443–453, 1970.
[20]. Wang Y., Zhang Z., Yao D., Qu B., Guo L. Inferring protocol state machine from network traces: a probabilistic approach. Proceeding of the 9th international conference on Applied cryptography and network security (ACNS), 2011, pp. 1-18.
[21]. Comparetti P.M., Wondracek G., Kruegel C., Kirda E. Prospex: Protocol Specification Extraction. Proceedings of the 30th IEEE Symposium on Security and Privacy, 2009, pp. 110-125.
[22]. Balakrishnan G., Gruian R., Reps T., Teitelbaum T. CodeSurfer/x86—A platform for analyzing x86 executables. Proceedings of the 14th international conference on Compiler Construction (CC'05), Springer-Verlag, Berlin, Heidelberg, pp. 250-254.
[23]. Balakrishnan G., Reps T. Analyzing Memory Accesses in x86 Executables. Proceedings of Compiler Construction, Springer-Verlag, New York, 2004, pp. 5-23.
[24]. Babić D., Martignoni L., McCamant S., Song D. Statically-directed dynamic automated test generation. Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, New York, USA, pp. 12-22.
[25]. Caselden D., Bazhanyuk A., Payer M., McCamant S., Song D. HI-CFG: Construction by Binary Analysis, and Application to Attack Polymorphism. Proceedings of 18th European Symposium on Research in Computer Security, Egham, UK, 2013. LNCS 8134, pp. 164-181.
[26]. Saxena P., Poosankam P., McCamant S., Song D. Loop-extended symbolic execution on binary programs. Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09). ACM, New York, USA, pp. 225-236.
[27]. Caballero J., Poosankam P., McCamant S., Babić D., Song D. Input generation via decomposition and re-stitching: finding bugs in Malware. Proceedings of the 17th ACM conference on Computer and communications security (CCS '10). ACM, New York, USA, pp. 413-425.
[28]. Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing Mayhem on Binary Code. Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12). IEEE Computer Society, Washington, USA, pp. 380-394.

Применение программных эмуляторов в задачах анализа бинарного кода ⋆

Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И.
{Pavel.Dovgaluk, vladimir_makarov, vartan, melon, Natalia.Fursova}@ispras.ru

Аннотация. В статье приводится опыт применения программных эмуляторов в качестве средства динамического анализа бинарного кода: как трассировщика уровня машинных команд и как развитого инструмента интерактивной отладки. Описывается механизм детерминированного воспроизведения, реализованный в эмуляторе QEMU, позволивший обеспечить указанные функциональности.

Ключевые слова: эмулятор, динамический анализ, детерминированное воспроизведение, обратная отладка

1. Введение

Программные эмуляторы успешно применяются в различных областях: как средство консолидации серверов, во время кросс-платформенной разработки, при замещении устаревшей аппаратуры. В последние годы эмуляторы активно начали применять при анализе бинарного кода. Например, изучение вредоносного кода, как правило, ведется в виртуальной машине, а не на реальной аппаратуре. Помимо вопроса защиты окружения от влияния изучаемого кода, решается и обратная задача – разграничивается предмет изучения и инструмент анализа. Факт работы отладчика в общем окружении легко выявляется, после чего изучаемая программа может либо целенаправленно изменить свое поведение, либо попытаться нарушить процесс отладки.

Помимо интерактивной отладки, программные эмуляторы позволяют собирать детальные трассы выполнения уровня машинных команд, которые затем можно анализировать с высокой степенью автоматизации. Разработанная в ИСП РАН среда анализа [1] использует именно такой подход. Возможность выполнения анализа базируется на том, что для целевой процессорной архитектуры существует трассирующий полносистемный эмулятор, т.е. эмулятор, в котором работает операционная система, драйвера устройств, пользовательские приложения. В данной статье приводится опыт работы с такими эмуляторами, а также описываются ключевые

⋆ Работа поддержана грантом РФФИ 12-01-31417

усовершенствования, внесенные в эмулятор с открытым исходом QEMU, ставший основным средством сбора трасс.

2. Опыт сбора трасс в программных эмуляторах

Основной способ добиться понимания, как устроена программа, базируется на анализе потоков управления и данных, включая знания о том, какие значения имеют обрабатываемые данные. Запись одной только последовательности выполнившихся команд не может обеспечить успешность анализа. Для вычисления исполнительных адресов в случае косвенной адресации необходимо знать содержимое регистров общего назначения. Дополнительно необходимы: счетчик команд, слово состояния, значения некоторых системных регистров, например, отвечающих за декодирование и порядок выполнения команд, идентификацию процессов. Перечисленные данные обязательны для восстановления потока управления.

Помимо того, крайне важно контролировать входящие и выходящие потоки данных в виртуальной машине. Если изучается программа, взаимодействующая по сети, то необходимо вести трассу входящих сетевых пакетов. Аналогично, но на другом уровне, требуется вести трассу записей в оперативную память, осуществляемых периферийными DMA-устройствами. В случае внешней памяти (винчестер, USB-накопитель и т.д.) важно уметь отслеживать номера блоков, в которые пишутся данные и из которых идет чтение; знание номеров блоков позволяет более точно восстанавливать поток данных.

Однако не все программные эмуляторы могут быть использованы для исчерпывающего сбора данных. В процессе развития среды анализа бинарного кода произошла смена нескольких полносистемных эмуляторов, использующихся в качестве основного средства сбора трасс. Первый трассировщик был реализован для архитектуры x86 в рамках эмулятора Simics [2] компании Virtutech. Simics поддерживает значительное количество целевых архитектур, куда кроме x86 входят ARM, PowerPC, MIPS, и множество периферийных устройств. Эмулятор обладает развитым и удобным SDK, который позволяет разрабатывать расширения, но при этом сам эмулятор – закрытый коммерческий продукт, полученный для исследований в рамках академической лицензии. В 2010 году компания Virtutech была поглощена Intel и в настоящее время доступность этого эмулятора стала еще более ограниченной. Использование трассировщика в Simics было недолгим из-за того, что в результате исследований в эмуляторе были выявлены неточности. Это привело к тому, что сбор трассы был перенесен в эмулятор SimNow [3] компании AMD. SimNow поддерживает исключительно процессоры производства AMD, но зато обеспечивает большую точность. Однако и он не свободен от ошибок, исправление некоторых сильно затягивалось. Тем не менее, именно этот эмулятор долгое время оставался

основным средством сбора трасс для архитектуры x86 и продолжает использоваться по сей день.

Расширение среды анализа на другие процессорные архитектуры потребовало создания новых трассировщиков. Эмулятор с открытым исходным кодом Dynamips [4] использовался для получения трасс на архитектурах MIPS и PowerPC. Однако его применение ограничено, поскольку в нем эмулируются конкретные модели сетевых маршрутизаторов Cisco. Еще одним негативным фактором стало отсутствие сообщества разработчиков – в период своего существования (2005-2008 гг.) проект фактически развивался единственным человеком. Следует упомянуть, что летом 2013 г. группа из трех заинтересованных разработчиков попыталась возродить проект. Выпускаемый ей эмулятор dynamips-community [5] основан на «официальной» кодовой базе Dynamips, новшества в основном заключаются в исправлении ошибок, улучшении процесса сборки, разработке документации.

Для сбора трасс на архитектуре ARM в первую очередь был разработан трассировщик на основе эмулятора ARMulator [6], который входит в состав ARM Development Studio компании ARM Holdings. Следует различать данный полносистемный эмулятор с одноименным проектом, в котором пытаются разработать эмулятор уровня приложения. К сожалению, скорость работы самого эмулятора оказалась неприемлемо низкой, а накладные расходы, связанные с трассировкой, только ухудшили ситуацию. В качестве альтернативы был выбран эмулятор с открытым исходным кодом QEMU [7], который помимо ARM поддерживает большое количество других процессорных архитектур.

Вне зависимости от того, насколько точно перечисленные выше эмуляторы выполняют команды целевых архитектур, все они страдают от одной существенной проблемы – крайне низкой скорости работы во время снятия трассы выполнения. Замедление эмулятора по сравнению с обычной работой составляет 4-5 порядков. При анализе «замкнутых» программ такое замедление негативно сказывается только на требуемом времени. Но если программа состоит из нескольких частей и происходит взаимодействие по сети одной (исследуемой) части программы с другой (неподконтрольной), то замедление при сборе трассы гарантированно повлияет на работу исследуемого кода: как минимум вызовет обрыв сетевых соединений из-за превышения времени ожидания, а в худшем случае поменяет поведение исследуемых алгоритмов. Описанная проблема делает необходимым наличие скоростной трассировки, когда удаленная часть программы не сможет зафиксировать замедления работы других частей, возникающего из-за сбора трассы.

Единственным известным на данный момент способом организации скоростной трассировки без ограничений по длительности сбора трассы является так называемая двухпроходная трассировка на основе детерминированного воспроизведения работы виртуальной машины [8].

Детерминированное воспроизведение – это процесс восстановления хода выполнения программы с использованием заранее записанных входных данных. Происходит два запуска программы: во время первого все входные данные (события) записываются в журнал, во время второго – считываются из журнала. Цель детерминированного воспроизведения – максимально точно повторить ход выполнения программы, которое происходило во время записи событий. В нашем случае в качестве «программы» выступает снимок состояния виртуальной машины, входные данные – внешние, недетерминированные события: асинхронные прерывания, пользовательский ввод, входящие сетевые пакеты.

Полносистемное воспроизведение имеет такие преимущества, как возможности отладки и анализа системных компонентов и многопоточных приложений, а также возможность выполнять анализ любой операционной системы из числа поддерживаемых аппаратной платформой. Когда журнал событий записан, появляется возможность его многократного воспроизведения с целью анализа поведения программы.

Существует ряд подходов к реализации полносистемного воспроизведения. Все они основаны на использовании различных виртуальных машин, и, в зависимости от технологии, используемой при реализации виртуальной машины, могут быть разбиты на три группы: аппаратная виртуализация, «чистая» программная эмуляция, бинарная трансляция.

Первый подход заключается в перехвате внешних событий средствами аппаратной виртуализации; в качестве примеров реализации можно привести системы XenLR [9] и XTRec [10]. Они отличаются невысокими накладными расходами в процессе записи журнала. В качестве недостатка можно отметить, что все эти системы детерминированного воспроизведения ориентированы только на аппаратную платформу x86. Кроме того, XenLR ограничивается одной модельной платформой: операционная система MiniOS собственной разработки и три периферийных устройства (таймер, клавиатура, жесткий диск), что не позволяет применять его для анализа сколь либо содержательных программ.

Второй вариант реализации детерминированного воспроизведения (система ExecRecorder) был реализован в эмуляторе Bochs [11]. Этот эмулятор поддерживает только платформу x86 и работает значительно медленнее, чем основанные на виртуализации и динамической трансляции аналоги.

Третий вариант подхода к реализации детерминированного воспроизведения основан на виртуальной машине с динамической трансляцией бинарного кода. Наиболее интересные результаты были заявлены в публикациях сотрудников VMware [12, 13]: объем журнала – в среднем 4.8 байта на каждую 1000 выполнившихся машинных команд, замедление – около 5% при включении записи журнала. В отладочной версии 6.5 VMware Workstation присутствовал механизм воспроизведения, причем при проигрывании журнала присутствовала возможность получения трассы машинных команд,

дополненных содержимом регистров общего назначения. Проведенные эксперименты показали перспективность подхода, замедление при получении журнала действительно оказалось незначительном, что позволяло анализировать код нового класса программ – работающих с сетью. Однако информации, содержащейся в трассе, было недостаточно. Например, отсутствовали данные о прерываниях и изменениях системных регистров. В отдельных случаях некоторые требуемые данные удавалось восстановить, исходя из выполнявшихся команд и их адресов, но полноценный анализ трассы был невозможен. Более того, в следующем выпуске, Workstation 7.0, трассировка уже не поддерживалась, была предоставлена только возможность обратной отладки. А вскоре поддержка детерминированного воспроизведения была полностью остановлена по причине перехода виртуальной машины Workstation с бинарной трансляции на аппаратную виртуализацию.

Было решено реализовать детерминированное воспроизведение на основе программного эмулятора с открытым исходным кодом. Потенциально пригодными для такой реализации являлись три эмулятора: Bochs, QEMU и VirtualBox [14]. Все перечисленные эмуляторы активно используются в индустрии и обладают сообществами разработчиков. Но VirtualBox и Bochs рассчитаны только на одну целевую архитектуру – x86, а Bochs, как уже упоминалось выше, не использует бинарную трансляцию, что негативно сказывается на скорости работы. Таким образом, эмулятор QEMU представился наиболее подходящей платформой для реализации детерминированного воспроизведения: открытый код, приемлемая скорость работы, поддержка большого количества целевых архитектур (x86, ARM, MIPS, PPC, …), эксплуатация на промышленном уровне и сообщество квалифицированных разработчиков, постоянно улучшающих возможности эмулятора. Более того, производители мобильных устройств нередко используют QEMU как базовую платформу для создания «официальных» эмуляторов в составе распространяемых SDK, в качестве примеров можно привести такие платформы, как Symbian, Android, Maemo и MeeGo. Все это позволяет рассматривать QEMU как единую платформу для получения трасс машинных команд различных целевых архитектур и различных классов исследуемых программ.

Следует упомянуть, что спустя незначительное время с начала исследований вышла публикация коллектива тайваньских исследователей о системе FREE [15], в рамках которой детерминированное воспроизведение также было реализовано на основе QEMU. Исходный код системы был не доступен, но авторы заявляли о быстродействии, превосходящем аналог на основе Bochs и уступающем аппаратной виртуализации. Возможности FREE ограничивались только архитектурой x86 и периферийными устройствами не использующими DMA. Дальнейших публикаций об этой системе не наблюдалось, но результаты, заявленные в [15], подтверждали верность выбора QEMU.

3. Устройство механизма детерминированного воспроизведения

Эмулятор QEMU имеет структуру, показанную на рис. 1.

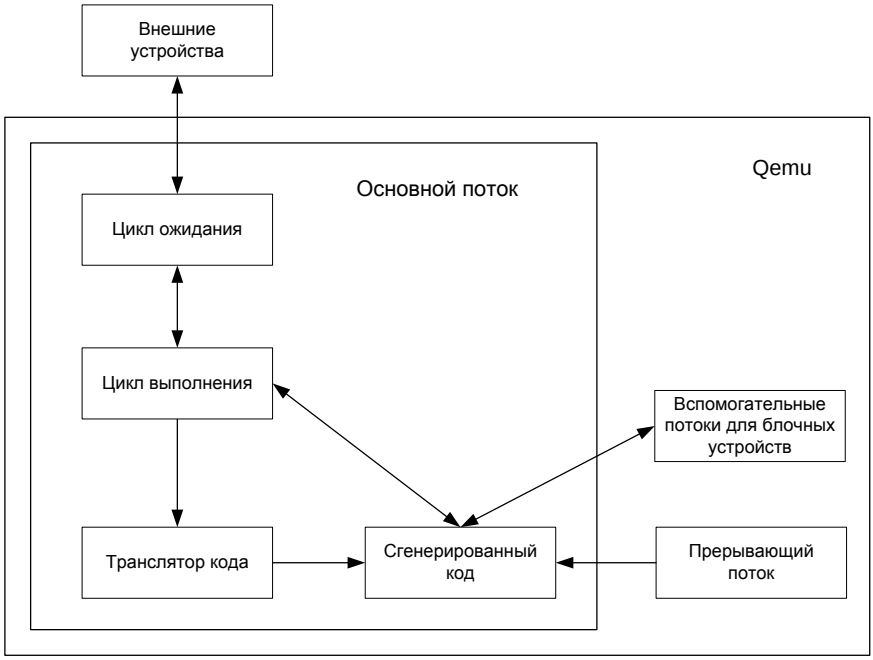


Рис. 1 – Структура эмулятора QEMU.

Цикл выполнения в QEMU производит трансляцию кода симулируемой машины и исполнение сгенерированного кода, обрабатывая небольшую его часть в каждой итерации. Параллельно с этим работает прерывающий поток, останавливающий выполнение кода с определенной периодичностью. При этом цикл выполнения прерывается и происходит возврат в цикл ожидания.

В цикле ожидания эмулятор сначала проверяет состояние внешних устройств и интерфейсов управления эмулятором, а затем возвращает управление в цикл выполнения. Так как прерывающий поток не синхронизирован с основным, детерминированное воспроизведение моментов возникновения прерываний таймера невозможно без переработки этого механизма.

Кроме того, QEMU взаимодействует с внешними устройствами, включая мышь, клавиатуру, сетевую карту, аппаратные таймеры. Через эти внешние устройства в QEMU могут приходить сообщения, которые не являются детерминированными и, следовательно, их обработчики также должны быть доработаны.

Реализация взаимодействия с блочными устройствами, такими как жесткий диск или привод оптических дисков, отличается от взаимодействия с другими внешними устройствами. Блочные устройства, как правило, работают с образами дисков, которые находятся в файлах главной машины. Операции чтения и записи данных из этих файлов выполняются в отдельных потоках для ускорения работы виртуальной машины, что обуславливает необходимость введения дополнительной синхронизации между ними и основным потоком для работы детерминированного воспроизведения.

3.1. Запись журнала событий

Для того чтобы воспроизводить процесс выполнения программ в виртуальной машине, была реализована запись всех недетерминированных событий в специальный журнал.

События, записываемые в журнал, делятся на синхронные и асинхронные. Синхронные события вызываются действиями, выполняемыми основным потоком (выполнение инструкции, итерации цикла ожидания, чтение часов). Асинхронные события приходят в эмулятор извне в произвольный момент времени (прерывание таймерного потока, нажатие клавиши на клавиатуре, движение мыши, приход сетевого пакета).

Выполнение очередной инструкции

Выполнение инструкции является внутренним детерминированным событием. Однако, для того, чтобы корректно воспроизводить моменты возникновения недетерминированных событий (внутренний таймерный поток, события от внешних устройств), необходимо учитывать количество выполненных инструкций. Поэтому был доработан код, отвечающий за трансляцию кода целевой машины для того, чтобы обновлять счетчик выполненных виртуальным процессором инструкций. Данная доработка является платформо-зависимой и выполнялась для всех поддерживаемых аппаратных платформ.

События от часов реального времени

В процессе своей работы эмулятор осуществляет считывание показаний часов реального времени как для своей работы, так и для передачи в виртуальную машину (Рис 2).

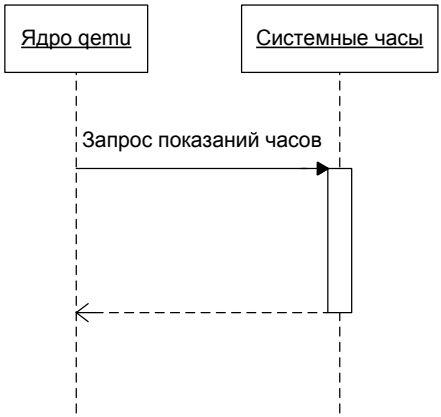


Рис. 2 – Получение показаний часов в исходной версии QEMU.

Чтобы показания часов реального времени, передаваемые внутрь симулируемой системы, не изменились при воспроизведении поведения системы, в модуль, осуществляющий работу с аппаратными часами, были внесены изменения, позволяющие записывать считанные показания часов в журнал (Рис. 3).

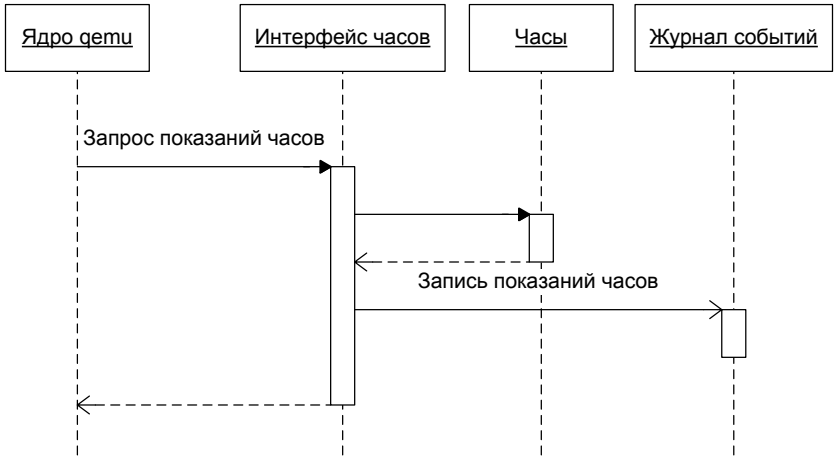


Рис. 3 – Запись показаний часов в журнал.

При воспроизведении журнала событий вместо считывания показаний аппаратных часов, выполняется чтение их из журнала (Рис. 4). Так как воспроизведение является детерминированным, операция чтения происходит в тот же момент (относительно позиции в журнале), что и при записи.

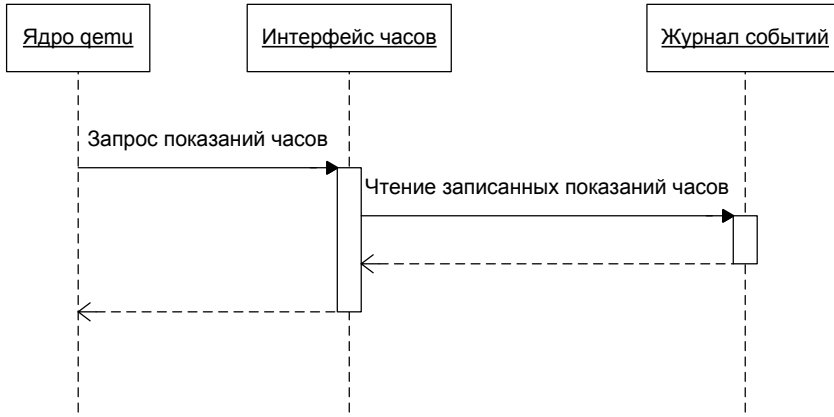


Рис. 4 – Чтение показаний часов из журнала.

Однако, в случае выхода эмулятора в цикл ожидания из цикла выполнения (например, при отладке), могут выполняться операции чтения времени, которых не было в исходном потоке событий. Для того чтобы они завершались успешно, при воспроизведении журнала выполняется кэширование показаний часов. Таким образом, в цикле ожидания будут считываться одни и те же показания, пока выполнение симулируемого кода не продолжится.

Прерывание выполнения симулируемого кода с помощью таймера

Для решения проблемы с синхронизацией потоков внутри QEMU, было решено зафиксировать точки в коде основного потока, в которых он может взаимодействовать с прерывающим потоком. Таким образом, в процессе воспроизведения журнала событий появится возможность точно восстанавливать моменты этих взаимодействий.

Несколько точек возможных прерываний были размещены в циклах ожидания и выполнения, а также был модифицирован код транслятора для добавления таких точек перед выполнением каждой инструкции. Таким образом, становится возможным детерминированное воспроизведение поведения системы даже при пошаговом выполнении кода.

События от периферии

Запись и воспроизведение событий от мыши, клавиатуры и сетевой карты отличается от предыдущих видов событий тем, что они инициируются извне. Поэтому запись информации о них в журнал производится непосредственно в момент возникновения соответствующего события.

Так как основные действия в QEMU, а также запись в журнал, выполняются в одном потоке, сообщения от периферийных устройств будут обрабатываться строго между какими-либо другими событиями, не пересекаясь с ними по времени. Поэтому при воспроизведении журнала (при поиске очередного синхронного события) может встретиться такое асинхронное событие (например, нажатие клавиши), которое может быть тут же обработано.

Работа с физической реализацией блочных устройств выполняется отдельными потоками внутри эмулятора, что позволяет возникнуть состоянию гонок. Например, при одновременной работе с оперативной памятью DMA-контроллера, выполняющего чтение с жесткого диска, и центрального процессора, читающего те же ячейки памяти. Поэтому эмулятор был модифицирован таким образом, чтобы при работе этих вспомогательных потоков выполнение основного потока приостанавливалось.

Экспериментальная оценка характеристик записи журнала

Скорость роста журнала определяет, насколько длительные периоды работы могут быть зафиксированы для последующего воспроизведения. На рис. 5 приводится график скорости роста журнала, т.е. количество байт, записываемых в журнал за 1 секунду. В качестве гостевой системы выступала ОС Windows XP, замер делался на модифицированной версии QEMU 0.13. Никаких внешних воздействий на систему не оказывалось: отсутствуют входящие сетевые пакеты, пользователь не работает с устройствами ввода. Таким образом, график отражает только те данные, которые порождаются работой эмулируемых периферийных устройств. Именно эта составляющая отражает качество механизма воспроизведения – вне зависимости от реализации пользовательский ввод и сеть будут одинаково увеличивать содержимое журнала. После того, как ОС завершила этап загрузки, объем записываемых в журнал данных стабилизировался и, начиная с третьей минуты, составил примерно 32 КБ/с.

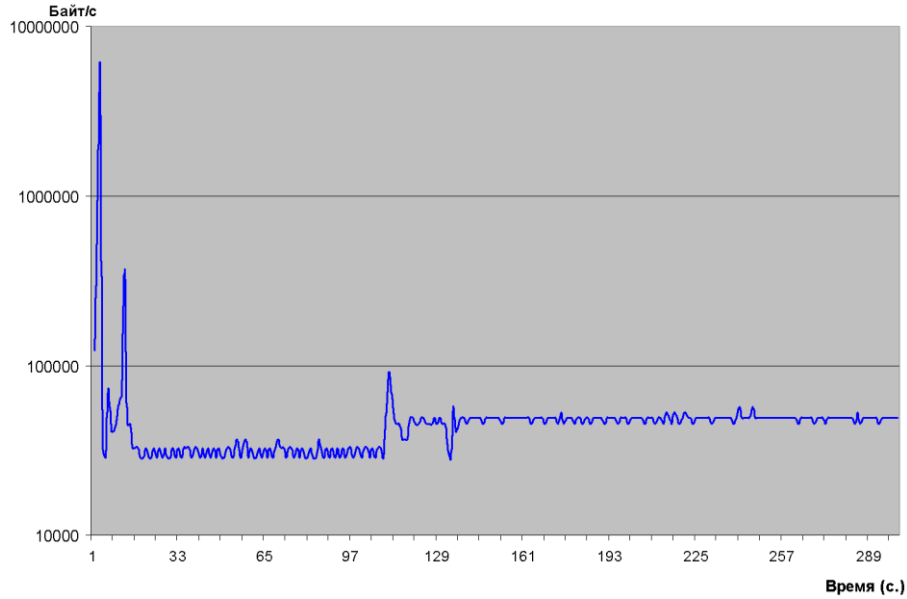


Рис. 5 – Скорость роста журнала.

У системы ExecRecorder (основанной на Bochs) этот показатель составляет примерно 1.5 МБ/с. Сопоставление с системой Retrace затруднительно, т.к. авторами заявлялась характеристика в виде числа записываемых байт на 1000 выполненных команд, но грубый пересчет, исходя из оценки скорости работы виртуального процессора в 500 МГц, дает примерно 2 МБ/с, что также

ощутимо больше, нежели полученные в нашей работе результаты. Без учета входящего сетевого трафика такой характер роста журнала (32 КБ/с) позволяет использовать обычные жесткие диски для записи часов и даже суток непрерывной работы виртуальной машины.

В другой характеристике, замедлении при снятии журнала, имеет место некоторый проигрыш: двукратное замедление (+100%) в сравнении +5% у системы Retrace. Тем не менее, достигнутый уровень замедления уже не позволяет выявить факт снятия журнала неконтролируемой частью исследуемой программы, работающей на другом компьютере. Замедление такого порядка может быть обусловлено различными другими причинами, например: плохим сетевым каналом, недостаточной производительностью или высокой загруженностью хостовой системы.

3.2. Детерминированное воспроизведение журнала событий

Технология детерминированного воспроизведения журнала событий заключается в выполнении кода эмулятора и считывании необходимых для воспроизведения программы данных из журнала:

- Считывание показаний часов из журнала, когда их запрашивает какой-либо код. Если соответствующих показаний нет – возвращается кэшированное значение.
- Передача сообщений от клавиатуры, мыши, звуковой и сетевой карт в тот момент, когда они считываются из журнала в соответствующие обработчики.
- Обновление внутреннего счетчика команд, когда происходит выполнение очередной инструкции.
- Выполнение заданий на чтение/запись данных жесткого диска, когда соответствующие события поступают из журнала.

Чтобы воспроизведение журнала было детерминированным, оно должно начинаться с того же самого состояния симулируемой системы, что и запись. Состояние системы включает в себя состояния всех симулируемых устройств, включая образы используемых дисков. Остальные устройства инициализируются при старте эмулятора и, поэтому, их состояния не отличаются при записи и воспроизведении.

3.3. Контроль корректности воспроизведения журнала

Для того чтобы контролировать, является ли процесс воспроизведения детерминированным, при считывании событий из файла журнала проверяется следующее ограничение: в момент, когда ожидается наступление события «выполнение инструкции» не должно возникать ни события от часов, ни события «цикл ожидания».

Если это ограничение нарушается, это означает, что ход эмуляции был нарушен. В этом случае выдается сообщение об ошибке и процесс воспроизведения останавливается. Это позволяет контролировать отсутствие

ошибок в механизме воспроизведения и выполнять отладку при их возникновении.

4. Сбор трасс для последующего автоматизированного анализа

Одной из целей доработок, вносимых в QEMU, является динамический анализ программ с помощью снятой с них трассы выполнения. Трасса представляет собой последовательность из выполняемых инструкций, а также состояний процессора (включающих значения регистров), соответствующих этим инструкциям. Каждая группа «выполняемая инструкция – значений регистров» называется шагом трассы.

Реализация механизма снятия трасс выполнения состоит из нескольких частей:

1. Платформенно-независимая часть, сохраняющая сформированные шаги трассы в файл. При этом используется сжатие с помощью библиотеки LZO, так как сохраняемый объем данных достаточно велик.

2. Изменения в платформенно-независимом модуле пользовательского монитора, включающие поддержку команд начала и завершения процесса снятия трассы.

3. Изменения в платформенно-зависимом модуле трансляции выполняющегося в виртуальной машине бинарного кода. Транслятор вставляет перед каждой выполняющейся командой дополнительный код, заполняющий шаг трассы и передающий его модулю сохранения для записи в файл.

В настоящее возможность снятия трассы реализована для платформ x86 и ARM. Трудозатраты на реализацию этого механизма для других платформ достаточно малы, т.к. требуется внести изменения только в механизм трансляции, добавив код, который заполняет шаг трассы и вызывает механизм его сохранения.

Снятие трассы предполагает запись выполненных инструкций и значений регистров, а учесть данные, которые пишутся в память или считываются из нее без участия процессора, не представляется возможным. В QEMU был реализован механизм сбора логов обращений к жесткому диску. Лог содержит данные, которые были прочитаны с жесткого диска или записаны на него, а также информацию о номере сектора, количестве секторов, имени устройства и, если QEMU работает в режиме детерминированного воспроизведения, то номер текущего шага воспроизведения (в качестве номера шага выступает число выполненных процессором инструкций, начиная с начала процесса воспроизведения). Следует учитывать, что работа с блочными устройствами ведется в двух режимах – PIO (Programmed Input/Output) и DMA (Direct

Memory Access); в первом случае при считывании или записи данных участвует эмулируемый центральный процессор, а во втором нет. QEMU работает как с PIO, так и с DMA, для DMA транзакций в лог попадает так же адрес памяти, куда контроллер запишет или прочитает блок данных.

5. Расширение возможностей отладки

Детерминированное воспроизведение можно использовать не только для непрерывного многократного выполнения сценария, но и для перехода в произвольные места записанного сценария для просмотра состояния виртуальной машины – ее регистров, памяти и т.п. Также возможность быстрого перехода в нужное состояние дает дополнительные возможности при отладке программы, выполняющейся в виртуальной машине.

5.1. Сохранение состояний виртуальной машины

QEMU изначально включал в себя механизм сохранения состояний виртуальной машины. Сохраненные состояния можно в последующем загружать и, таким образом, неоднократно начинать выполнение с сохраненной позиции. Чтобы иметь возможность неоднократно переходить в нужное состояние, достаточно сохранить стартовое состояние виртуальной машины и, в дальнейшем, загружая его, воспроизводить журнал событий до нужной точки.

Также в механизм записи журнала была внесена возможность автоматического сохранения состояний виртуальной машины с заданной периодичностью. Это позволяет пользователю быстрее переходить к нужной точке журнала, загружая ближайшее состояние.

Для того чтобы пользователь мог перемещаться между точками журнала в режиме его воспроизведения, в подсистему пользовательского монитора была внесена поддержка дополнительной команды, выполняющей данную операцию.

5.2. Детерминированная и обратная отладка

Детерминированная отладка – это способ поиска ошибок в недетерминированных приложениях, при котором недетерминированность устраняется с помощью записи сценария работы системы (или программы) в журнал. Разработанный метод воспроизведения работы программ позволяет выполнять детерминированную отладку недетерминированных приложений следующим образом:

1. Тестировщик записывает сценарий, при выполнении которого проявляется дефект в тестируемой программе, в журнал, а затем передает этот журнал вместе с образами дисков системы разработчику. Здесь сценарий выступает не только в роли исходных

данных для отладки, но и в роли описания способа воспроизведения дефекта.

2. Разработчик может неоднократно проигрывать полученный сценарий в эмуляторе, анализируя причины появления дефекта. Разработчику не нужно настраивать сложное окружение системы так же, как это делал тестировщик, так как все особенности взаимодействия с этим окружением уже записаны в журнал.

Таким образом, отладка недетерминированных приложений с применением разработанного метода становится детерминированной, что позволяет сократить время, затрачиваемое разработчиками на локализацию дефектов в программе, а тестировщиками – на описание процесса их воспроизведения.

Обратная отладка – возможность перехода отлаживаемой программы к ранее пройденным состояниям. В нашем случае вся виртуальная машина (т.е. гостевая операционная система и выполняющиеся в ней программы) рассматривается как отлаживаемая программа. Обратная отладка позволяет пользователю переходить от момента в программе, когда обнаруживается исключительная ситуация (например, обращение к памяти через испорченный указатель) к моменту времени, когда формируются данные, вызвавшие эту ошибку (запись некорректного значения указателя).

5.3. Известные инструменты обратной отладки

Встроенные возможности по обратной отладке в отладчике gdb (для x86), а также в Trace32 (для ARM) реализованы с помощью записи состояний процессора и значений ячеек памяти [16, 17]. Таким образом, можно просматривать ранее пройденные состояния и наблюдать значения переменных. Но такой подход имеет и ограничения. Записывается только определенное число состояний процессора (откат возможен на ограниченное число шагов). Если не использовать специальные аппаратные средства для записи данных процессора, возникает значительное замедление, способное повлиять на ход работы отлаживаемой программы. Кроме того, т.к. записываются только состояния процессора и ячеек памяти, при возврате назад невозможно наблюдать полное состояние системы (например, вывод программы на экран).

Подобный подход также используется в реализации отладчиков для различных программных платформ, например, Java, C# и т.п. При этом в отлаживаемую программу добавляется специальный код, записывающий интересующие пользователя данные, что может повлиять на поведение отлаживаемой программы [18, 19].

Кроме методов, записывающих состояния процессора, существуют также методы обратной отладки, использующие детерминированное воспроизведение программ [20, 21]. Детерминированное воспроизведение в самом простом случае предполагает сохранение начального состояния программы и результатов системных вызовов в процессе ее выполнения. Так

появляется возможность перехода к заданному шагу программы с помощью загрузки ее начального состояния и выполнения с подменой системных вызовов. В случае большого количества системных вызовов, использования разделяемой памяти или многопоточных приложений, использования слишком многих системных библиотек, применение данного подхода становится слишком трудоемким.

Если же детерминированное воспроизведение необходимо для системы целиком, отладка требует выполнения дополнительного прохода с записью журнала недетерминированных событий. В журнал должны попадать как начальное состояние системы, так и все внешние события (пользовательский ввод или работа с сетью), позволяющие восстановить заданное состояние системы в дальнейшем. После записи сценария выполнения программы в журнал событий, становится возможным воспроизводить его и уже выполнять непосредственно отладку.

Преимущества подходов, основанных на детерминированном воспроизведении, в том, что в отлаживаемую программу не вносится никаких изменений, а также становится возможной многократная отладка одного и того же сценария выполнения программы, в котором может проявляться ошибка. Кроме того, процесс отладки становится детерминированным, что особенно важно для ошибок, проявляющихся от случая к случаю.

Однако большинство из современных решений были разработаны для процессоров с архитектурой x86 и не поддерживают широко распространенную в настоящее время архитектуру ARM. Описываемый в данной статье метод лишен этого недостатка.

5.4. Реализация обратной отладки

Эмулятор QEMU включает в себя механизм, позволяющий с помощью отладчика gdb подключаться к виртуальной машине и управлять процессом выполнения ее кода: останавливать и возобновлять процесс выполнения, задавать точки останова и контрольные точки данных, считывать и записывать значения регистров и памяти, а также выполнять пошаговую отладку.

Начиная с 7-й версии, отладчик gdb поддерживает команды обратной отладки, такие как шаг назад и поиск первой сработавшей контрольной точки в обратном направлении. Для того чтобы интерпретировать эти команды, был доработан модуль взаимодействия QEMU с gdb. Кроме этого, были внесены доработки в основной модуль детерминированного воспроизведения для реализации сценариев обратной отладки.

Чтобы выполнить действие «шаг назад», QEMU загружает ближайшее из сохраненных состояний виртуальной машины, предшествующих этому шагу, и продолжает выполнение, пока нужный шаг не будет достигнут.

Команда «выполнение в обратном направлении до первой точки останова» (reverse-continue) выполняется в несколько этапов (Рис. 6). Сначала

загружается состояние виртуальной машины, предшествующее текущему положению, и выполняется поиск точки останова, которая срабатывает последней перед достижением текущего шага. Если такой точки останова не найдено, данный этап повторяется для предыдущего состояния. После нахождения сработавшей точки останова происходит переход к ней с помощью загрузки ближайшего состояния и воспроизведения журнала событий до этой точки.

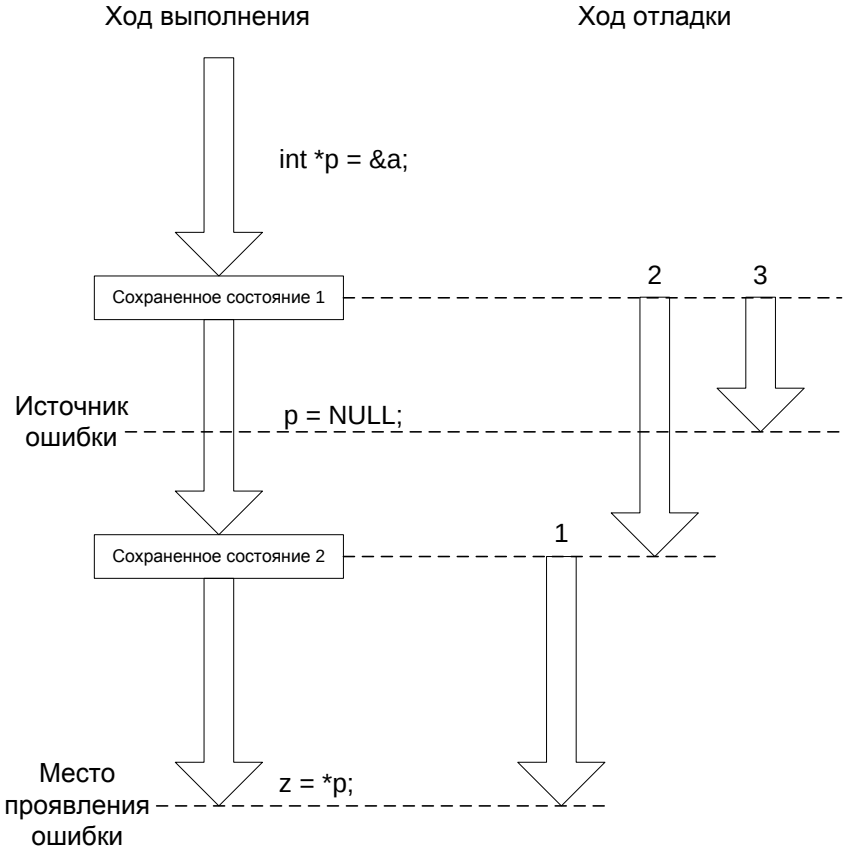


Рис. 6 – Ход выполнения обратной отладки при поиске ближайшей точки останова

Таким образом, использование сохраненных в процессе записи сценария состояний виртуальной машины позволяет находить нужную точку останова за время в лучшем случае равное удвоенному периоду времени между записанными состояниями. В худшем случае это время будет равняться удвоенному времени выполнения сценария целиком. На практике реально

сохранять состояние виртуальной машины каждые 4-5 секунд, что означает, что команда reverse-continue чаще всего будет выполняться за 8-10 секунд.

6. Заключение

В настоящий момент механизм детерминированного воспроизведения работает в модифицированной версии QEMU 1.5 и поддерживает платформы x86 и ARM. Благодаря динамической трансляции эмулятор поддерживает множество других аппаратных платформ [7], для которых также может быть реализовано детерминированное воспроизведение.

Уже поддерживаемый механизмом детерминированного воспроизведения список устройств включает все виртуальные устройства (то есть не взаимодействующие с чем-либо, кроме других виртуальных устройств), а также такие устройства, как аудиоадаптер, сетевая карта, последовательный порт, мышь, клавиатура, внешние USB-устройства.

Открытость исходного кода эмулятора позволяет реализовывать поддержку дополнительных периферийных устройств, не представленных в исходной версии эмулятора. В том случае, если периферийное устройство является виртуальным, для поддержки детерминированного воспроизведения не требуется выполнять никаких дополнительных действий.

Кроме того, в эмулятор можно добавить специфическое устройство, общающееся с внешним миром (например, сетевую карту). В этом случае требуется расширение механизма детерминированного воспроизведения для того, чтобы получаемые извне данные записывались в журнал. Затем, при воспроизведении этого журнала, они должны считываться и подаваться на вход устройства, заменяя реальный обмен данными с внешним миром.

Список литературы

[1] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды Института Системного Программирования, том: 16, 2009. стр. 51-72.

[2] Full System Simulation. http://www.windriver.com/products/simics/ дата обращения 2 декабря 2013

[3] SimNow™ Simulator http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/ дата обращения 2 декабря 2013

[4] Cisco 7200 Simulator http://www.ipflow.utc.fr/blog/ дата обращения 2 декабря 2013

[5] GNS3 / dynamips https://github.com/GNS3/dynamips дата обращения 2 декабря 2013

[6] ARM Software development tools http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0058d/Chdcdbib.html дата обращения 2 декабря 2013

[7] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page дата обращения 2 декабря 2013

[8] Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. // ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th

symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.
[9] Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. // In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology, 2008. pp. 149-154.
[10] Amit Vasudevan, Ning Qu, Adrian Perrig. XTRec: Secure Real-time Execution Trace Recording on Commodity Platforms. // In Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS'11), 2011. pp. 1-10.
[11] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T.Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. // Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71
[12] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. // In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS, San Diego, CA, June, volume 3, pages 4--2, 2007
[13] Jim Chow, Tal Garfinkel, Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. // Proceedings of the 2008 Annual USENIX Technical Conference, June 2008. pp. 1-14
[14] Oracle VM VirtualBox https://www.virtualbox.org/ дата обращения 2 декабря 2013
[15] Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. // The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.
[16] GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html, дата обращения 2 декабря 2013
[17] Microprocessor Development Tools. http://www.lauterbach.com/frames.html?home.html, дата обращения 2 декабря 2013
[18] Omniscient Debugging. http://www.lambdacs.com/debugger/ODBDescription.html, дата обращения 2 декабря 2013
[19] How Does VS2010 Historical Debugging Work? http://www.wintellect.com/CS/blogs/jrobbins/archive/2009/06/16/how-does-vs2010-historical-debugging-work.aspx, дата обращения 2 декабря 2013
[20] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005, pp. 1-15
[21] Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. VEE '05 Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, New York, NY, USA, 2005, pp. 79-88

Application of software emulators for the binary code analysis*

Dovgalyuk P.M., Makarov V.A., Padaryan V.A., M.S. Romaneev, Fursova N.I.
{Pavel.Dovgaluk, vladimir_makarov, vartan, melon, Natalia.Fursova}@ispras.ru
ISP RAS, Moscow, Russia

Annotation. The paper describes the experience of using software emulators as a means of dynamic analysis of binary code tools. Emulator is considered as tracer of machine commands layer and as interactive debugging tool. It describes a mechanism of deterministic replay implemented in emulator QEMU.

Deterministic replay is a process of recovery of program execution using a pre-recorded input. To replay process of program execution in virtual machine recording of all nondeterministic events to journal was implemented. Such events are indications of real time clock, messages from keyboard, mouse, sound and network cards. Currently the deterministic replay mechanism works in a modified version of QEMU 1.5 and supports x86 and ARM platforms.

To solve the problems of binary code analysis tracing is used, but it slows down the system, so it is easier to do with deterministic replay. Trace is a sequence of executed instructions and processor state (including register values). Each group "executed instruction - values of registers" is called a trace step. Currently tracing is implemented for x86 and ARM platforms. Trace does not contain information about the read and written memory, so logging of hard disk drive accesses was implemented.

Deterministic debugging is a way to find errors in nondeterministic applications, in which nondeterminism is eliminated by writing the scenario of system work. By means of deterministic replay nondeterministic debugging becomes deterministic strongly reducing the time spent on the localization of defects in the program and their description.

Reverse debugging is the possibility of studying the past states of the program. In our case the entire virtual machine is considered the program being debugged.

Emulator QEMU includes mechanism to let GNU debugger connect to virtual machine and manage the process of execution. GNU debugger supports reverse debugging commands, such as reverse-step and reverse-continue.

* The paper is supported by RFBR grant 12-01-31417

Keywords: emulator, dynamic analysis, deterministic replay, reverse debugging

References

[1]. Padaryan V.A., Get'man A. I., Solov'ev M. A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 51-72 (in Russian).
[2]. Full System Simulation. http://www.windriver.com/products/simics/
[3]. SimNow™ Simulator. http://developer.amd.com/tools-and-sdks/cpudevelopment/simnow-simulator/
[4]. Cisco 7200 Simulator. http://www.ipflow.utc.fr/blog/
[5]. GNS3 / dynamips. https://github.com/GNS3/dynamips
[6]. ARM Software development tools. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0058d/Chdcdbib.html
[7]. QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page
[8]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.
[9]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology, 2008. pp. 149-154.
[10]. Amit Vasudevan, Ning Qu, Adrian Perrig. XTRec: Secure Real-time Execution Trace Recording on Commodity Platforms. In Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS'11), 2011. pp. 1-10.
[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T.Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71
[12]. M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS, San Diego, CA, June, volume 3, pages 4--2, 2007
[13]. Jim Chow, Tal Garfinkel, Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. Proceedings ofthe 2008 Annual USENIX Technical Conference, June 2008. pp. 1-14
[14]. Oracle VM VirtualBox . https://www.virtualbox.org/
[15]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.
[16]. GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html
[17]. Microprocessor Development Tools. http://www.lauterbach.com/frames.html?home.html
[18]. Omniscient Debugging. http://www.lambdacs.com/debugger/ODBDescription.html

[19]. How Does VS2010 Historical Debugging Work? http://www.wintellect.com/CS/blogs/jrobbins/archive/2009/06/16/how-does-vs2010-historical-debugging-work.aspx
[20]. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005, pp. 1-15
[21]. Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. VEE '05 Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, New York, NY, USA, 2005, pp. 79-88

Методы динамической и предварительной оптимизации программ на языке JavaScript

Роман Жуйков <zhroma@ispras.ru>, Дмитрий Мельник <dm@ispras.ru>,Рубен Бучацкий <ruben@ispras.ru>, Ваагн Варданян <vaag@ispras.ru>, Владислав Иванишин <vladislav.ivanishin@gmail.com>, Евгений Шарыгин <eugene.sharygin@gmail.com>

Аннотация. Работа посвящена улучшению производительности программ на языке JavaScript. В работе рассматриваются особенности динамических оптимизаций в JIT-компиляторе для языка JavaScript, а также основные способы улучшения производительности для таких оптимизаций. Кроме того, предлагается способ организации предварительной компиляции программ на языке JavaScript с их сохранением в виде байткода, что позволяет сократить время запуска приложений за счет выполнения оптимизаций на этапе предварительной компиляции. Предложенные методы были реализованы в библиотеке с открытым исходным кодом для отображения веб-страниц WebKit. В результате удалось добиться значительного увеличения производительности выбранных тестовых JavaScript-приложений на платформе ARM.

Ключевые слова: оптимизация программ; компиляция во время выполнения; предварительная компиляция; граф потока данных; архитектура ARM.

1. Введение

В настоящее время широкое распространение получили программы на нетипизированных скриптовых языках, одним из повсеместно используемых языков является JavaScript. С ростом производительности персональных компьютеров и встраиваемых систем использование языка JavaScript стало возможно не только для выполнения небольших скриптов на веб-страницах, но и целых веб-приложений. Более того, уже имеются разработки операционных систем для телефонов, планшетов и ноутбуков, которые подразумевают использование JavaScript как одного из главных языков для создания приложений. Примерами таких систем могут быть Tizen[1] и FirefoxOS. Некоторая часть пользовательских приложений на этих системах будет представлять собой набор хранящихся на устройстве веб-страниц со скриптами на языке JavaScript. В связи с этим все больше возрастают требования к производительности программ-скриптов.

Многие современные интерпретаторы поддерживают разнообразные режимы компиляции горячих участков кода во время выполнения скриптов. Для обеспечения быстрого выполнения скриптов необходимо создание максимально качественного машинного кода, а также снижение затрат на все этапы оптимизации, выполняемые при компиляции во время выполнения. В данной работе будет рассматриваться интерпретатор JavaScript, называемый JavaScriptCore (JSC)[2] и входящий в состав браузерного движка WebKit[3] для отображения веб-страниц.

Цель данной работы – адаптировать имеющиеся в JavaScriptCore оптимизации для работы на встраиваемых системах. Также необходимо учесть специфику использования JavaScript-программ в составе локально хранящихся приложений. Планируется добиться генерации более качественного кода при компиляции во время выполнения, а также реализовать систему хранения и предварительной компиляции скриптов для приложений.

Дальнейшее изложение построено следующим образом. Сначала будет описана имеющаяся схема работы JavaScriptCore и перечислены предлагаемые методы оптимизации, а потом каждый из них будет рассмотрен более подробно.

2. Устройство JavaScriptCore

Первыми этапами работы JavaScriptCore являются лексический и синтаксический анализ. Исходный код разбивается на токены, методом рекурсивного спуска строится синтаксическое дерево, из которого в свою очередь строится внутреннее представление, называемое байткод (bytecode). В байткоде инструкции хранятся в виде массива ячеек, разные инструкции могут занимать разное количество ячеек. В первой ячейке хранится тип инструкции, в следующих ячейках хранятся адреса операндов и результата. Адреса операндов могут представлять собой ссылки на константы или номера локальных псевдорегистров. При чтении или записи полей объектов, загрузка адреса поля по имени выглядит как отдельная инструкция, один из операндов которой – константная строка, содержащая имя поля. Для многих инструкций последняя ячейка в байткоде выделена для хранения информации о профиле.

В ранних версиях JavaScriptCore байткод сразу передавался на выполнение интерпретатору. Интерпретатор последовательно читал инструкции байткода и выполнял необходимые действия, переходы и циклы организовывались за счет условных и безусловных операций перехода. Переход указывает, что вместо чтения следующей инструкции в байткоде интерпретатор должен перейти в другое место. В современных версиях JavaScriptCore вместо интерпретатора используется низкоуровневый интерпретатор (LLInt). Он фактически выполняет те же самые действия, однако запрограммирован на специальном мультиплатформенном ассемблере (offlineasm). Этот специальный ассемблер может быть скомпилирован на этапе сборки JavaScriptCore в машинный код для x86, ARM или нескольких других

платформ, а также может быть преобразован в исходный код на языке C. LLInt, как и обычный интерпретатор, позволяет начать выполнение байткода, не выполняя никаких подготовительных этапов, тем самым обеспечивает быстрое начало выполнения. Все другие уровни оптимизации требуют предварительных затрат по созданию машинного кода, соответствующего заданному участку байткода. LLInt поддерживает на уровне вызова функций взаимодействие со всеми уровнями оптимизации. Если какая-то функция уже была скомпилирована в машинный код, то вызов этой функции из низкоуровневого интерпретатора будет выглядеть так же, как и переход на точку входа в общий пролог интерпретатора для любой другой неоптимизированной функции. LLInt использует кэширование на уровне байткода для ускорения доступа к полям объектов по имени.

При работе низкоуровневого интерпретатора также происходит сбор информации о профиле – сохраняются типы и последние значения полей объектов. Необходимость оптимизации функций определяется с помощью оценки того, сколько раз в ней выполняются те или иные участки кода. Для перехода на первый уровень оптимизации времени выполнения (JIT-оптимизация) необходимо, чтобы функция набрала не менее 100 “очков выполнения”, при этом за каждую пройденную итерацию цикла прибавляется одно “очко”, а за вызов функции - 15 “очков”. Отметим, что эти числа являются примерными, в реальности дополнительно применяется эвристика, результат работы которой зависит от размера рассматриваемой функции. Таким образом, небольшой функции без циклов достаточно быть вызванной около 7 раз, чтобы для нее была выполнена базовая компиляция времени выполнения (Baseline JIT).

Baseline JIT создает для каждой операции байткода соответствующий машинный код. В этом коде реализуются все возможные случаи для данной операции. Например, операция сложения для чисел будет выполнена как сложение, а для операндов-строк – как конкатенация. Генерируемый код будет содержать множество ветвлений для разбора всех таких случаев. После того> как для функции будет создан машинный код, нет необходимости дожидаться окончания функции для запуска выполнения нового кода. Например, если функция выполняет цикл с большим числом итераций, то может быть выполнен немедленный переход на новый код (on-stack-replacement, OSR). Низкоуровневый интерпретатор закончит обработку очередной инструкции байткода и сразу перейдет в машинном коде в то место, которое соответствует началу следующей инструкции. Конечно, во всех местах вызова этой функции будет произведено перенаправление на новую версию функции – в машинном коде.

Baseline JIT код используется, как базовая версия кода для функций, которые скомпилированы с помощью оптимизирующего JIT-компилятора. Если оптимизированный код сталкивается со случаем, который в нем не поддерживается (например, тип или значение переменной не соответствует

собранному профилю), то происходит обратная замена на стеке (on stack replacement exit, OSR exit) к коду Baseline JIT. На уровне Baseline JIT так же, как и на LLInt, сохраняется профиль – информация о типах полей объектов и аргументов функций, и выполняется кэширование для ускорения доступа к полям объектов.

Информация о профиле, собранная на уровнях Baseline JIT и LLInt, используется для организации спекулятивного выполнения на следующем уровне оптимизации – оптимизации с использованием графа потока данных (Data flow graph, DFG JIT, Speculative JIT). Собранная информация содержит последние значения загруженных аргументов, полей объектов, а также результатов выполнения функций. Кэширование доступа к полям объектов на уровнях LLInt и Baseline JIT устроено так, что позволяет DFG быстро получать необходимую информацию. Например, по информации кэширования легко можно узнать, что некоторое обращение к полю объекта иногда, часто или всегда возвращает значение некоторого конкретного типа.

DFG JIT компиляция выполняется для функций, которые набрали не менее 1000 “очков выполнения”. На уровне DFG выполняются разнообразные оптимизации, опирающиеся на информацию о профиле. Из байткода с учетом профиля создается граф потока данных, в котором инструкции описаны в виде SSA-представления. На этом DFG графе выполняются оптимизации, и в конце итоговый набор инструкций преобразуется в машинный код.

DFG JIT распространяет полученную информацию о типах переменных по всему графу и вставляет в код необходимые проверки типов. Иногда DFG даже выполняет спекулятивную оптимизацию по самому значению переменной. Например, если по результатам профилирования поле объекта является конкретной функцией, ее код может быть встроен в вызывающую функцию, с добавлением необходимой проверки. Как было описано выше, когда одна из проверок не выполняется, происходит деоптимизация, то есть обратная замена на стеке (on stack replacement exit, OSR Exit) на код Baseline JIT.

Таким образом, DFG JIT код и Baseline JIT код могут сменять друг друга посредством замены на стеке (OSR). Когда код функции становится “горячим”, происходит переход на DFG JIT. Когда выполняется деоптимизация, происходит обратный переход. В случае многократного OSR exit сохраненная информация о том, почему произошла деоптимизация, также становится своеобразным профилем, который позволяет организовать реоптимизацию DFG, то есть создание нового DFG графа и машинного кода с учетом новой информации о профиле. Эвристика, оценивающая необходимость реоптимизации, использует экспоненциальную задержку в зависимости от количества уже выполненных реоптимизаций. Это позволяет исключить возникновение больших временных затрат на постоянную реоптимизацию кода и выполнение множества OSR переходов.

Четвертый уровень оптимизации - LLVM JIT, вызывается для функций, набравших не менее 10000 “очков выполнения”. В нем выполняется более широкий набор оптимизаций, а в качестве внутреннего представления помимо DFG графа используется биткод компилятора LLVM. Перед генерацией машинного кода выполняются оптимизации, уже реализованные в LLVM. Данный уровень JIT-оптимизации находится в состоянии разработки и пока не включается по умолчанию.

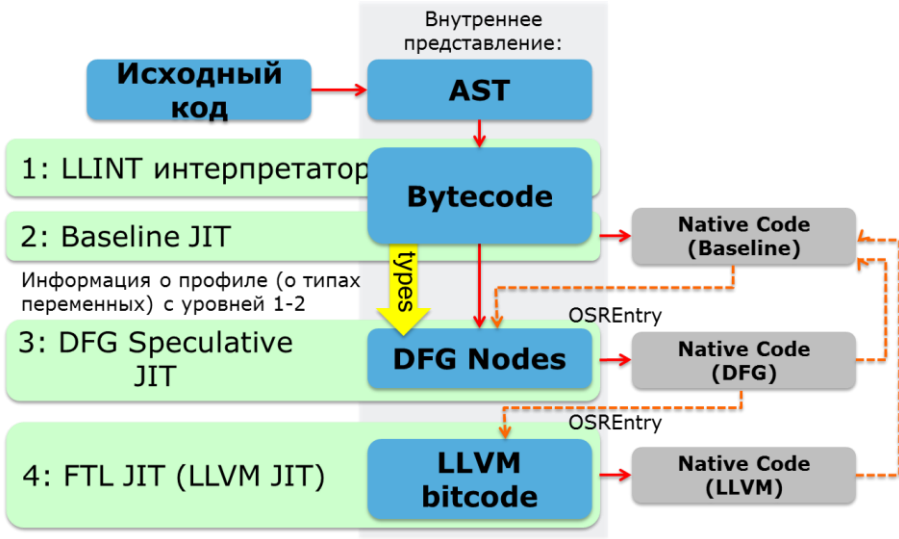


Рис. 1. Устройство JavaScriptCore

Итак, при выполнении скрипта в любой момент времени функции eval-блоки и глобальный код в JSC могут выполняться на любой комбинации LLInt, Baseline JIT и DFG JIT кода. В особом случае при выполнении рекурсивных функций код одной и той же функции может существовать на стеке вызовов в разных вариантах: в одном уровне функция выполняется на LLInt, в другом на Baseline JIT, в третьем на DFG. Возможен еще более сложный случай – допускается выполнение старого варианта DFG кода на одном уровне стека, в то время как на более вложенном уровне рекурсии произошло много деоптимизаций, и была выполнена реоптимизация, после которой был запущен новый вариант DFG кода.

Все уровни выполнения обеспечивают одинаковую семантику выполнения, и единственный эффект переключения между ними – производительность работы JavaScriptCore.

3. Оптимизация производительности

Для обоснования необходимости переключения на более быстрые уровни выполнения приведем два примера: первый – сравнение времени выполнения набора тестов PL benchmark от автора Martin Richard. Этот набор тестов запрограммирован на нескольких языках, поэтому есть возможность взять время выполнения программы на языке C как ориентир.

Таблица 1. Сравнение производительности уровней JSC.

Способ выполнения	Время выполнения, мс
Реализация на языке C	1.2
JavaScript интерпретатор	129
LLInt интерпретатор	58
Baseline JIT	8.4
DFG JIT	2.1

Другой набор тестов – Browsermark. На этом наборе Baseline JIT оказывается в среднем в 2.5 раза быстрее, чем LLInt, причем на некоторых тестах наблюдается различие производительности в 5 раз. Однако оптимизирующий DFG JIT еще в 1.7 раза в среднем быстрее Baseline JIT и позволяет ускорить некоторые тесты до 6 раз.

Как видно из результатов сравнения производительности, в JSC важно добиться, чтобы максимальное количество горячих участков кода выполнялось на уровне DFG JIT. На этом уровне не поддерживается часть операций байткода, поэтому для увеличения эффективности можно рассмотреть возможность реализации поддержки новых операций. Другим направлением может быть изучение причин деоптимизации, то есть обратных переходов на Baseline JIT, и исследование возможности их устранения. Это позволит избежать выполнения более медленной версии кода и последующей перекомпиляции.

Также необходимо улучшать качество генерируемого DFG JIT машинного кода. Здесь важным направлением для исследований является замена в машинном коде реализации некоторых сложных операций, выполняемых с помощью вызова функции. Можно иногда вместо вызова JavaScript-API функции, который требует достаточно больших затрат на подготовку аргументов и стека, выполнять так называемый intrinsic, то есть машинный код, выполняющий необходимые действия. Этот код может также вызывать соответствующую C-функцию, однако временные затраты на такой вызов будут значительно меньше.

Еще одним направлением для рассмотрения является идея предварительной оптимизации. В случае, если программа на языке JavaScript хранится локально, то перед ее выполнением может быть сделана оптимизационная предварительная подготовка, позволяющая ускорить процесс выполнения.

4. Динамические оптимизации

Опишем алгоритм работы с предсказаниями типов на уровне DFG JIT. Определение типов [4] достигается с помощью профилирования и последующего предположения о результатах операций, базирующегося на информации о профиле.

В код вставляются необходимые проверки типов, а далее производится попытка продвижения информации о типах результатов операций. Рассмотрим следующий пример выражения на языке JavaScript: *p.x * p.x + p.y * p.y*

Допустим, что в контексте нашего кода, объект *p* описывает точку на плоскости в декартовой системе координат, соответственно у него два поля *x* и *y*, которые хранят значение типа *double*. Эти значения иногда могут быть целыми, и, несмотря на то, что стандарт языка JavaScript не подразумевает хранения целых чисел, в целях эффективности выполнения следует по возможности хранить целые числа в виде *int32*, а не в виде *double*. Чтобы оценить проблему определения типов и способы ее решения в JavaScriptCore, необходимо в первую очередь оценить, какой объем работы должен выполнить интерпретатор языка JavaScript для вычисления выражения, приведенного выше, если он не имеет никакой информации об объекте *p* и его полях.

- В выражении *p.x* в первую очередь необходимо понять, не имеет ли объект *p* какой-то специальной обработки обращения к полю *x*. *p* может быть, например, DOM-объектом, который нетривиальным образом перехватывает обращения к своим полям. Если нет никакой специальной обработки, нужно проверить для заданного объекта *p* существование поля *x*, где “*x*” - строка из одного символа. Объекты хранятся в виде таблиц, где символьным строкам соответствует значение поля или метод доступа. Если это метод доступа, он должен быть вызван. Если в таблице конкретное значение - оно и должно быть возвращено. Если в объекте *p* нет поля *x*, то необходимо повторить весь процесс поиска в прототипе объекта. Ускорение доступа к полям объектов с помощью профилирования или кэширования не рассматривается более подробно в данной работе.
- Бинарная операция умножения должна в первую очередь проверить типы операндов. Если операнд является объектом — то необходимо вызвать метод *valueOf* для данного объекта. Если

операнд является строкой — должна быть сделана попытка преобразовать строку в число. Когда получены операнды-числа, необходимо проверить, являются ли они целыми. Если да, то выполняется умножение целых чисел. Оно может вызвать переполнение, и тогда будет выполнено преобразование и умножение в типе *double*. Также оно будет выполнено сразу, если один из операндов-чисел не был целым числом. Получается, что результатом выполнения умножения может быть как целое число (*int32*), так и вещественное (*double*). И нет никакого способа в общем случае определить, какое это будет число и как оно будет представлено по результатам умножения.

- Бинарная операция сложения в выражении *p.x * p.x + p.y * p.y* сталкивается почти с теми же сложностями, что и операция умножения. Помимо перечисленных случаев для умножения, при операции сложения дополнительно должен быть рассмотрен вариант, когда оба операнда являются строками — для сложения строк должна быть выполнена их конкатенация. В нашем примере можно доказать, что такой вариант невозможен, поскольку результатом умножения строка быть не может, как и не может быть другой сложный объект. Однако по-прежнему необходимы проверки для *int32* или *double*, поскольку неизвестно, каков будет результат умножения. В итоге, результатом сложения также может быть как целое, так и вещественное число.

Идея определения типов в JSC основана на том, что мы с большой вероятностью можем предсказать типы, которые возвращают арифметические операции, если у нас есть предположение о типах операндов. Таким образом, возникает что-то вроде шага математической индукции — для операций, у которых мы можем предсказать типы операндов, мы можем предсказать и результат. Но для индукции нужна база, и базой становятся все операции, которые загружают внешние для заданной функции значения: например, загрузка полей объектов, использование аргумента функции, или использование возвращаемого значения. Типы значений для этих операций берутся из результатов профилирования значений на уровне LLInt и Baseline JIT. Каждой операции загрузки нелокального значения соответствует ровно одна ячейка информации о профиле, и там хранится последнее значение.

В самом простом виде алгоритм определения типов можно описать так: для каждого из хранящихся последних значений можно узнать тип, а дальше применить индукцию для распространения информации о типах на все операции внутри функции. Это дает нам предсказания типов для всех операций и переменных внутри функции.

В реальности, JavaScriptCore помимо последнего значения, хранит еще одно поле, которое описывает спекулятивный тип *SpecType*, в который вмещается

случайное подмножество значений, виденных ранее. Сначала этот тип заполняется как *SpecNone* — тип, которому не соответствует ни одно значение, аналог пустого множества. Когда выполнение программы проходит через эту точку, иногда включается логика профилирования, и поле типа заполняется новым значением. Новый тип должен включать в себя старый и одновременно допускать хранение записанного последнего значения.

Продвижение информации о типах *SpecType* по всем операциям выполняется с помощью стандартного итеративного алгоритма анализа потоков данных, реализованного как поиск неподвижной точки. На этапе DFG-компиляции этот алгоритм выполняется в одном из первых проходов, который обрабатывает построенный DFG граф.

После того, как для каждой операция в заданной функции был вычислен предсказанный тип значения, вставляются спекулятивные проверки типов. Например, для умножения делается проверка, что операнды являются числами. Если во время выполнения проверка получит отрицательный результат — будет выполнена деоптимизация, и выполнение перейдет на неоптимизированный код Baseline JIT. Это позволяет при выполнении DFG JIT кода использовать информацию о типах в последующих операциях. Например, пусть выполняется сложение *a+b*, и предсказаны целые типы операндов *SpecInt32*. До сложения будет вставлена проверка что *a* и *b* целые, иначе запускается механизм деоптимизации. После сложения будет вставлена проверка на переполнение, при переполнении также будет выполнен OSR exit. После завершения операции сложения можно считать известным, что операнды *a* и *b* и результат их сложения являются целыми числами, помещающимися в *int32*. Это позволяет при выполнении последующих операций не вставлять проверки для этих переменных. Удаление избыточных проверок реализовано с помощью второго алгоритма анализа потоков данных, использующего анализ потока управления на графе DFG. Анализ потока управления также выполняет условное продвижение констант, которое иногда позволяет аналогично информации о типах получить информацию о том, что какое-то значение является постоянным.

Вернемся к рассмотрению нашего примера, выражения *p.x * p.x + p.y * p.y*. Здесь потребуются только проверки загружаемых значений *p.x* и *p.y*. После проверки, что *p.x* и *p.y* являются числами, мы можем для хранения всех промежуточных результатов использовать тип *double*, и остается только выполнить два умножения и одно сложение чисел с плавающей точкой. Можно сказать, что почти всегда после удаления избыточных проверок DFG JIT код будет выполнять проверку типа не более одного раза для каждой загрузки внешнего значения.

4.1. Проверки на отрицательный ноль

Одним из аспектов, который необходимо учитывать при оптимизации арифметических операций, является отрицательный ноль. В стандарте языка

JavaScript числа не делятся на целые и вещественные, и подразумевается поведение всех чисел как вещественных, поэтому следует различать положительный и отрицательный ноль. Например, значениями следующих операций с целыми числами является минус бесконечность (-Infinity). *1 / (-0) ; 1 / (0 / -3) ; 1 / (-4 % 4)*. В случае, если для оптимизации выполнения мы заменяем выполнение операций с числами double операциями с целыми числами *int32*, необходимо, помимо проверок переполнения, учитывать также различие положительного и отрицательного нуля, чтобы не получить неверный с точки зрения стандарта результат. Необходимо отметить, что в некоторых ситуациях такие проверки можно опустить без ущерба для корректности выполнения. Например, при вычислении выражения *5/(a%b+3)*, если результатом выражения *a%b* является ноль, нет необходимости различать положительный ноль и отрицательный ноль.

В JavaScriptCore имеется реализация проверок арифметических операций на отрицательный ноль, однако она содержит несколько ошибок и другие недочеты. Исправление всех проблем, связанных с обработкой случая отрицательного ноля, значительно ускорило выполнение тестов.

Итак, первое исправление в логику было внесено как раз в части алгоритма, отвечающей за необходимость проверок. Для каждой операции в графе DFG выставляется флаг *NodeNeedsNegZero*, который установлен, когда для результата данной операции необходимо различать положительный и отрицательный ноль. Флаг в некоторых случаях расставлялся неправильно, правильный алгоритм таков: если результат вычисления выражения x в дальнейшем участвует в вычислении суммы $x+C$, $C+x$ или разности $C-x$, где C—константа, не являющаяся отрицательным нолем ($C!=-0$), то для вычисления x не нужны проверки на отрицательный ноль. Аналогично для разности $x-C$, где $C!=+0$. Для всех других операций с числами флаг необходимо копировать. То есть операндам, если они сами получены как результат другой операции, он нужен тогда и только тогда, когда нужен для результата данной операции. Для операции унарного минуса флаг ошибочно стирался, это было исправлено.

Необходимо понять, в каких случаях целочисленные операции деления дают в результате отрицательный ноль, не представимый в виде *int32* значения. Операция деления *x/y* дает в результате -0, тогда и только тогда, когда *x* равен нолю и *y* меньше 0. Одна из ошибок реализации была в том, что на платформе ARM при равенстве делимого нулю сразу происходил OSR exit, без проведения проверки, что делитель отрицателен. Это вызывало множество ненужных возвратов на более медленный машинный код Baseline JIT.

Операция взятия остатка *x%y* дает в результате отрицательный ноль тогда и только тогда, когда *x* делится на *y* нацело, и x — отрицателен. Здесь также была реализована правильная проверка, а в имеющейся реализации после проверки результата на равенство нолю сразу выполнялся OSR Exit. Теперь он

выполняется только после дополнительной проверки, что делимое является отрицательным числом.

Для операции взятия остатка в DFG рассмотрен отдельно случай, когда делитель является константой и степенью двойки. В этом случае операция взятия остатка для оптимизации может быть заменена на операцию побитовой конъюнкции с числом на единицу меньшим. В случае отрицательного делимого также возможна такая реализация – необходимо предварительно поменять знак операнда, а в конце поменять знак результата. Однако в имеющейся реализации в этом случае пропускалась проверка на отрицательный ноль. Но при выставленном флаге *NodeNeedsNegZero* ее необходимо выполнять, чтобы не потерять знак в выражениях, таких как *z%4*, при значении *z* равном -4.

4.2. Поддержка новых операций байткода и внедренного машинного кода

Одной из неподдерживаемых операций на уровне DFG JIT является встроенная в JavaScript функция определения типа переменной – typeof. Поддерживался только часто используемый вариант, когда результат вызова typeof сравнивался со строковой константой, равной “number”, “string”, “object”, “function” или “undefined”. Причем, поддерживалось только сравнение на равенство, например, для конструкции отрицания сравнения !(typeof (x) == “string”) выполнялась DFG JIT компиляция. Для сравнения на неравенство (typeof (y) != “object”) компиляция не происходила, и функция, содержащая такую конструкцию, всегда работала на уровнях оптимизации не выше Baseline JIT. Нами было реализована полная поддержка операции typeof на уровне DFG JIT. Теперь поддерживается любой вариант использования, даже без последующего сравнения со строковой константой.

Другим примером неподдерживаемой операции являются циклы, организованные как перечисление всех полей объекта (for-in циклы). Для обработки таких циклов в граф DFG были добавлены все необходимые типы операций, и теперь функции, содержащие такой цикл, также могут эффективно выполняться на DFG JIT.

Посредством внедрения машинного кода вместо вызова с использованием JavaScript-API было реализовано ускоренное выполнение таких функций JavaScript, как *Math.power*, *Math.floor* и *String.fromCharCode*. Вместо того, чтобы организовывать сложный вызов функции на уровне DFG с большими затратами на создание пролога и эпилога для корректной работы со стеком обращения к таким функциям заменялись на более легковесные обращения к функциям в машинном коде, скомпилированном из языка C++ на этапе сборки JavaScriptCore. Для функции *Math.power* упрощенный вызов создавался только для случая, когда степень является целым числом.

Была также сделана попытка реализации *Math.floor* для платформы ARM вообще без вызова функции, с помощью использования операций с

плавающей точкой ARM NEON. Однако тестирование показало, что вариант с вызовом обычной C-функции floor показывает такую же производительность и создает меньший объем машинного кода.

5. Предварительные оптимизации

Одной из идей оптимизации JavaScriptCore является использование компиляции до выполнения (ahead of time compilation, AOTC)[5, 6], то есть добавление в систему предварительных оптимизаций. Изначально JavaScript используется для скриптов на веб-страницах, при загрузке страницы необходимо выполнить весь возникающий на ней код, и заранее про этот код ничего не известно. Однако теперь на языке JavaScript будут разрабатываться и более статичные приложения, хранящиеся на самом устройстве. Получается, что не обязательно использовать только подход интерпретатора, допускается выполнение некоторой подготовки кода. В данной работе разрабатывается идея, что исходный код заранее преобразуется в некоторый набор данных, содержащий байткод и другие внутренние представления, на которых можно провести какие-то предварительные оптимизации. Возможно также добавление сохранения машинного кода. Впоследствии, при выполнении программы загружаются готовые оптимизированные внутренние представления, которые корректируются по мере необходимости.

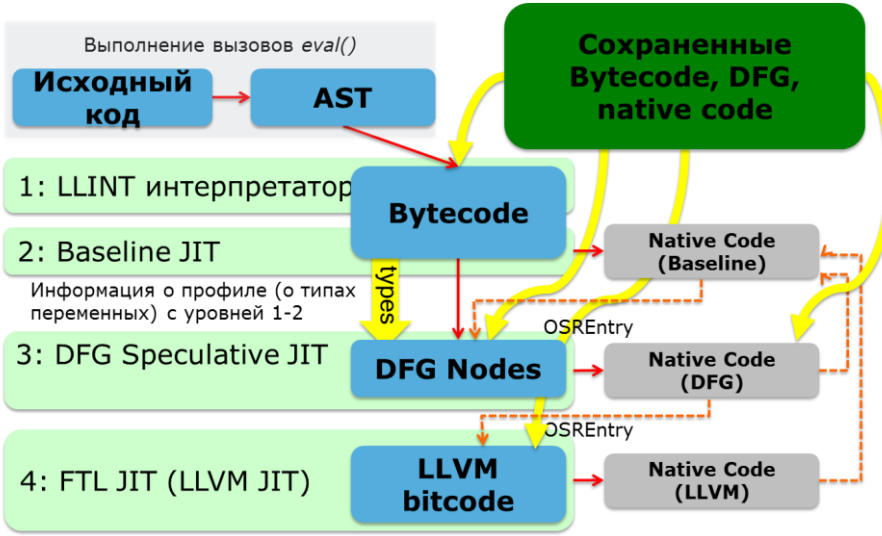


Рис. 2. Устройство системы предварительной компиляции (AOTC).

Такой порядок работы JavaScriptCore позволит получить следующие преимущества.

- В предварительной фазе могут быть выполнены значительно более сложные оптимизации, так как нет жестких ограничений по времени выполнения, имеющихся в JIT-компиляции. Однако в то же время необходимо отметить, что до выполнения программ-скриптов, нет никакой информации о профиле и значениях переменных.
- На этапе работы программы не тратится время на разбор и построение синтаксического дерева, кроме того, возможна экономия времени на генерацию машинного кода.
- Появляется шифрование исходного кода — байткод, другие внутренние представления и машинный код сложнее прочесть, чем исходный кода скрипта.

5.1. Сохранение и загрузка байткода

В данный момент в рамках работы над AOTC был реализован первый этап, который можно назвать AOTB (ahead of time bytecode). Он подразумевает сохранение исходного кода в виде байткода, для последующей загрузки при выполнении.

В обычном режиме работы JavaScriptCore при выполнении скрипта байткод генерировался только при первом вызове каждой функции. Нами была разработана и реализована схема генерации и сохранения байткода без выполнения самого скрипта. Вместе с байткодом сохраняется также вспомогательная информация, такая как таблицы констант, таблицы switch-переходов и исключений, необходимые данные для регулярных выражений. Для сохранения байткода без выполнения потребовалось сэмулировать работу стека пространств имен.

Байткод JavaScriptCore не был задуман как промежуточное внутреннее представление для сохранения, основной его целью является эффективное выполнение и генерация машинного кода на уровне Baseline JIT. Байткод, в отличие от исходной программы на JavaScript, отражает семантику программы только в определенном контексте. Например, в зависимости от свойств объектов, созданных к моменту начала выполнения программы, для нее может быть сгенерирован различный байткод. В основном, эта разница в байткоде относится к дополнительным подсказкам, например, позволяющим быстрее организовать обращение к полям объектов. Однако в некоторых случаях байткод, сохраненный вне того контекста, в котором программа будет исполняться, может приводить к некорректным результатам с точки зрения стандарта JavaScript. Эти особенности были учтены при сохранении байткода без выполнения. Кроме того, обращения к глобальным объектам содержат абсолютные адреса, и необходимо организовать сохранение так, чтобы можно было при загрузке байткода поменять адреса на новые, соответствующие адресам объектов во время выполнения.

Изначально планировалось хранение всей информации в виде базы данных SQLite [7], однако от такого способа хранения пришлось отказаться из соображений эффективной загрузки. Теперь все данные, относящиеся к одной функции, хранятся в виде последовательного набора байтов внутри файла. В начале файла сохраняется карта адресов (смещений), по которым можно найти информацию для каждой из функций. Соответственно, при выполнении из файла читается эта карта смещений и байткод для глобального JavaScript кода, то есть всего кода, описанного вне функций. В дальнейшей работе, при первом вызове функции вместо обычного разбора исходного кода байткод и все необходимые данные подгружается из файла по заданному смещению.

Необходимо отметить один из моментов, который позволил уменьшить размер сохраняемого файла — отказ от хранения двух вариантов байткода для каждой функции. При обычном выполнении JavaScript программ для функций, вызываемых как конструктор с помощью вызова new (“var z = new f()”), создается отдельный байткод. В нашей реализации хранится только байткод для случая обычного вызова функции, который при необходимости преобразуется в вариант “для конструктора”.

Таким образом, первый этап по решению задачи предварительной компиляции javascript программ выполнен. Реализовано сохранение и загрузка байткода. Выполняется статическая компиляция исходного кода в байткод без выполнения скрипта. При выполнении байткода вместо исходного кода стандарт ECMA-262[8] поддерживается полностью, вызовы eval поддерживаются, для них исходный код компилируется обычным образом в процессе работы JavaScriptCore. Исключением является только работа операций, явным образом требующих наличия исходного кода. Примерами таких операций могут служить вызовы function.toString(), либо использование поля line у объекта исключения. В этом поле должен храниться номер строки в исходном коде, которая создала исключение.

6. Результаты тестирования

Все найденные недостатки, связанные с реализацией проверок на отрицательный ноль в DFG коде, были устранены, что позволило исключить генерацию некорректного с точки зрения стандарта машинного кода. Исправление процесса удаления избыточных проверок позволило ускорить на платформе ARM несколько тестов из набора SunSpider. В среднем тестовый набор стал выполняться на 7% быстрее, ускорение конкретных тестов составляет до 35%.

Добавление ускоренного выполнения Math.power для случая целой степени ускоряет на 5% тест math-partial-sums из набора sunspider. Реализация for-in циклов ускоряет тест string-fasta на 2.5%.

Поддержка функции определения типа (typeof) ускоряет на 15% тест ArrayBlur из набора Browsermark. Что касается реализации fromCharCode и Math.floor,

они не дают видимого прироста производительности на исследуемых наборах тестов, однако увеличивают производительность искусственных тестов, проверяющих соответствующую функциональность.

Текущая реализация сохранения и загрузки файла с байткодом (AOTB) успешно проходит регрессионное тестирование на наборах из Webkit JavaScriptCore и V8. За счет уменьшения времени обработки исходного кода на 2-4% ускоряется работа тестов из SunSpider, v8 и kraken. Крупные data-файлы для тестов из kraken обрабатываются значительно быстрее, время их обработки не учитывается в результатах теста. По результатам профилирования работы JavaScriptCore было выявлено, что на больших исходных текстах время, затрачиваемое на загрузку файла с байткодом, может быть до 3-х раз меньше времени, необходимого на обычную обработку исходного кода.

Для тестов из наборов SunSpider, v8, kraken было измерено соотношение размера бинарного файла с сохраненным байткодом и размера исходного JavaScript-файла. Причем был взят пример как использования оригинальных файлов, так и файлов, упакованных с помощью Google Closure Compiler. Во втором случае оба файла дополнительно архивировались с помощью утилиты gzip с использованием максимального сжатия.

Таблица 2. Результаты сравнения объема JS-файлов и файлов с байткодом.

Тестовый набор	Соотношение размеров файла с байткодом и исходного файла	
	Оригинальный JavaScript	Google Closure Compiler + gzip
SunSpider	1.19	1.25
V8-v6	2.3	4.41
Kraken	1.97	1.31

7. Заключение

В рамках данной работы проведен анализ имеющихся в JavaScriptCore оптимизаций. По итогам исследования производительности на наборах тестов были выявлены недостатки в оптимизационных алгоритмах компиляции во время выполнения. Была добавлена поддержка компиляции новых операций, благодаря чему расширен класс функций, которые могут быть скомпилированы в более эффективный машинный код. Также были устранены недочеты, выявленные в системе предсказания типов переменных.

Изменения, внесенные в динамические оптимизации JavaScriptCore, позволили значительно ускорить выполнение тестовых наборов SunSpider, v8, kraken и Browsermark.

Для приложений на языке JavaScript, хранящихся локально на устройстве, была разработана система, позволяющая производить предварительную оптимизацию программ с последующей загрузкой и выполнением оптимизированного кода. Была реализована часть системы, позволяющая сохранять и загружать внутреннее представление – байткод. Она позволяет сократить до 3х раз время, затрачиваемое во время выполнения для получения готового байткода, поскольку компилятору не нужно делать лексический и синтаксический анализ.

В дальнейшем планируется продолжать разработку системы, добавив в нее предварительные оптимизации на уровне байткода. Необходимо также рассмотреть возможность сохранения других промежуточных представлений и оптимизированного машинного кода для последующей загрузки при выполнении.

Список литературы

[1] Веб-сайт платформы Tizen. http://www.tizen.org
[2] Описание реализации JavaScriptCore на веб-сайте разработчиков WebKit http://trac.webkit.org/wiki/JavaScriptCore
[3] Веб-сайт Webkit. http://www.webkit.org
[4] S. Li, B. Cheng, X. Li “TypeCastor: demystify dynamic typing of JavaScript applications”, Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011, pp. 55-65
[5] S. Hong, J. Kim, J. W. Shin, S. Moon, H. Oh, J. Lee, H. Choi “Java client ahead-of-time compiler for embedded systems”, Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, 2007, pp. 63-72
[6] S. Hong, S. Moon “Client-Ahead-Of-Time Compilation for Digital TV Software Platform” 3rd workshop on Dynamic Compilation Everywhere preprint, 2013. http://sites.google.com/site/dynamiccompilationeverywhere/home/dce-2014/DCE-2014-Sunghyun-Hong-article.pdf
[7] Веб-сайт SQLite. http://www.sqlite.org/about.html
[8] Описание стандарта ECMA-262 http://www.ecma-international.org/publications/standards/Ecma-262.htm

Dynamic and ahead of time optimization for JavaScript programs

Roman Zhuykov, ISP RAS, Moscow, Russia zhroma@ispras.ru
Dmitry Melnik, ISP RAS, Moscow, Russia dm@ispras.ru
Ruben Buchatskiy, ISP RAS, Moscow, Russia ruben@ispras.ru
Vaagn Vardanyan, ISP RAS, Moscow, Russia vaag@ispras.ru
Vladislav Ivanishin, ISP RAS, Moscow, Russia vladislav.ivanishin@gmail.com
Evgeniy Sharygin, ISP RAS, Moscow, Russia eugene.sharygin@gmail.com

Abstract. The paper is dedicated to performance improvement of JavaScript programs. In this work we examine the specifics of dynamic optimizations in the WebKit JSC JIT-compiler for JavaScript, and how the performance of such optimizations can be improved. We concentrate on two main directions for performance improvement. First, we strive to move the hot spots code generation to the highest possible speculative JIT level. We achieve this goal by removing the obstacles that prevent speculative optimizations of such code, e.g., by correctly checking for negative zero floating constant we remove unneeded checks and move more code to the speculative optimizations level. Second, we improve code generation quality for the JSC compiler, mainly by implementing some of operations in C++ instead of own JavaScript API.

Also we propose a method for ahead-of-time (AOT) compilation of JavaScript programs, and for saving them as a bytecode. This method allows reducing startup time of applications by moving the optimizations to AOT phase. The proposed methods were implemented in open-source WebKit library, and resulted in significant performance gain for popular JavaScript benchmarks on ARM platform. However, this comes at a cost of increased (up to 3x) source code file sizes.

Keywords: program optimizations; JIT optimization; ahead of time optimization; data flow graph; ARM

References

[1]. Tizen platfrom website. http://tizen.org/
[2]. JavaScriptCore WebKit website. http://trac.webkit.org/wiki/JavaScriptCore
[3]. WebKit Browser Engine website. http://www.webkit.org
[4]. S. Li, B. Cheng, X. Li “TypeCastor: demystify dynamic typing of JavaScript applications”, Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011, pp. 55-65

[5]. S. Hong, J. Kim, J. W. Shin, S. Moon, H. Oh, J. Lee, H. Choi “Java client ahead-of-time compiler for embedded systems”, Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, 2007, pp. 63-72
[6]. S. Hong, S. Moon “Client-Ahead-Of-Time Compilation for Digital TV Software Platform” 3rd workshop on Dynamic Compilation Everywhere preprint, 2013. http://sites.google.com/site/dynamiccompilationeverywhere/home/dce-2014/DCE-2014-Sunghyun-Hong-article.pdf
[7]. SQLite DB website. http://www.sqlite.org/about.html
[8]. ECMA-262 Standard. http://www.ecmainternational.org/publications/standards/Ecma-262.htm

Применение метода двухфазной компиляции на основе LLVM для распространения приложений с использованием облачного хранилища[1]

С.С. Гайсарян, Ш.Ф. Курмангалеев, К.Ю. Долгорукова, В.В. Савченко, С.С. Саргсян
{ssg@ispras.ru, kursh@ispras.ru, unerkannt@ispras.ru, sinmipt@ispras.ru, sevaksargsyan}@ispras.ru

Аннотация. В статье описывается метод двухфазной компиляции программ на языках Си/Си++, позволяющий распространять приложения в промежуточном представлении LLVM. Описывается модификация компонентов LLVM с целью сокращения времени генерации кода. Описываются разработанные оптимизации с использованием профиля выполнения программы. Рассматривается организация специализированного облачного хранилища приложений.

Ключевые слова: Двухфазная компиляция, оптимизация, LLVM, облачное хранилище.

1. Введение

Процесс распространения ПО через магазины приложений состоит в следующем: разработчик передаёт программный продукт владельцу магазина приложений, приложение размещается в интернет-магазине и становится доступным пользователю. Чтобы программный продукт оставался конкурентоспособным в течение длительного времени, разработчику необходимо обеспечивать функционирование приложения на большинстве программно-аппаратных платформ.

Как правило, эта проблема решается следующим образом: приложение или компилируется и оптимизируется разработчиком повторно для каждой новой платформы, с последующим добавлением в магазин новой версии бинарного кода, или реализуется с использованием динамических языков.

При применении JavaVM приложение распространяется в промежуточном представлении. Переносимость достигается ценой потери производительности

[1] Работа выполнена при поддержке РФФИ, грант 11-01-00954-а

из-за использования дополнительного слоя абстракции, скрывающего реальное оборудование от исполняемой программы. В этом случае для обеспечения приемлемого уровня производительности дополнительно применяются динамические оптимизации на целевой архитектуре во время исполнения приложения, в том числе оптимизации, учитывающие профиль исполнения программы.

Разработка на традиционных языках (например, Си/Си++) позволяет учитывать особенности платформ, используя машинно-зависимые оптимизации (распределение регистров, планирование и конвейеризация кода, векторизация), выигрыш от применения которых может достигать нескольких десятков процентов. По этой причине производитель предоставляет Native SDK, позволяющий использовать такие языки. Вместе с тем, разработка приложений на традиционных языках и их распространение через магазины приложений требуют существенных дополнительных накладных расходов разработчиков на перенос приложений в связи с необходимостью поддержки большого количества различных платформ.

В настоящее время активно ведутся работы по автоматизации переноса приложений на языках Си/Си++ на различные платформы. Эти работы базируются на идее компилирования Си/Си++-приложений в промежуточное представление и их распространение по аналогии с приложениями на динамических языках. Наиболее интересные результаты получены компанией Google, которая разрабатывает проект Portable Native Client, имеющий целью обеспечить запуск единой версии программы на архитектурах ARM и x86. Эта разработка рассчитана на небольшие приложения, работающие в браузере Chrome, и имеет ряд ограничений, в том числе ухудшение показателей производительности.

В настоящей статье в разделе 2 описывается система двухэтапной компиляции программ на основе LLVM и изменения, внесенные в динамический компилятор LLVM. Раздел 3 описывает реализованные оптимизации. В разделе 4 описывается сервер приложений. Раздел 5 завершает статью.

2. Двухфазная компиляция

В предлагаемой реализации метода двухфазной компиляции на первом этапе приложение компилируется на машинах разработчиков специальным набором компиляторных инструментов на базе LLVM [1], при этом выполняются лишь машинно-независимые оптимизации. Результат компиляции сохраняется в файлах с биткодом LLVM, дополнительно автоматически генерируется информация об устройстве программного пакета и о схеме его инсталляции. На втором этапе программа оптимизируется на машине пользователя, возможно, с учетом его поведения и особенностей его вычислительной системы. Поддерживается несколько режимов работы: а) автоматическая генерация кода бинарной программы, оптимизированной под конкретную

архитектуру, и ее развертывание с помощью сохраненной на первом этапе информации; б) динамическая оптимизация программы во время её работы с учетом собранного профиля пользователя; в) оптимизация программы с учетом профиля пользователя во время простоя системы (idle-time optimization) для экономии ресурсов.

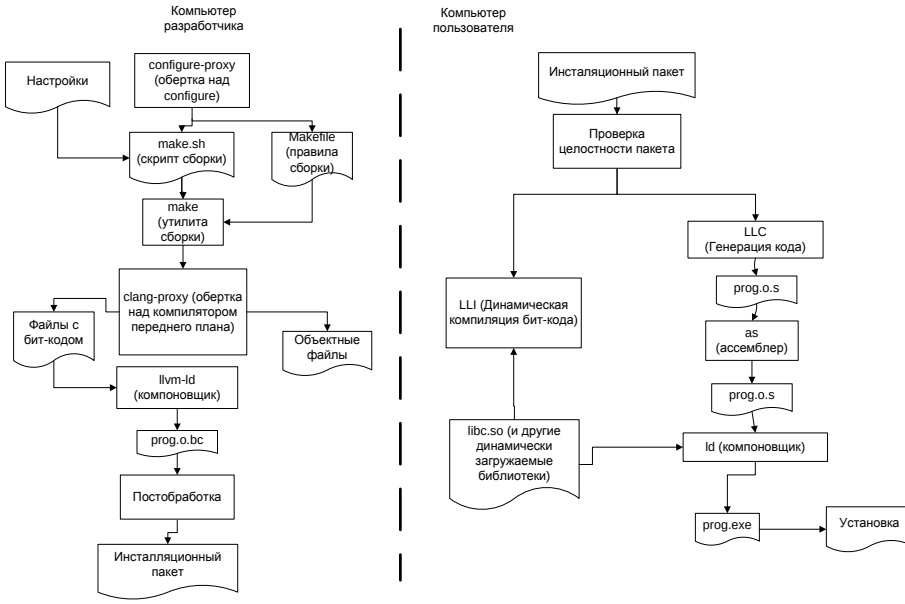


Рис. 1. Схема двухфазной компиляции

Схема работы системы двухфазной компиляции для программ, сборка которых основана на использовании утилит configure и make, представлена на рисунке 1. Указанные на рисунке дополнительные инструменты были разработаны и реализованы из-за того, что в LLVM не предусмотрены средства прозрачного, автоматического получения биткода с учетом зависимостей между модулями, а также отсутствует поддержка динамического связывания модулей с биткодом.

Предлагаемый метод распространения программ, написанных на языках Си/Си++, в промежуточном представлении позволяет решить проблему переносимости программ в пределах одного семейства процессоров с учетом специфических особенностей каждого конкретного процессора, проводить адаптивную компиляцию, учитывая поведение пользователя и характер входных данных. Собирая информацию о профиле программы, поступающую от пользователей, можно применить к промежуточному представлению

машинно-независимые оптимизации для повышения быстродействия программы для наиболее часто встречающихся вариантов использования. Помимо этого, распространение программы в промежуточном представлении позволяет применять средства статического анализа программ для поиска уязвимостей и производить запутывание программ для защиты от обратного проектирования. Все указанные операции могут происходить на машине пользователя, но это может привести к дополнительным накладным расходам, что является важным фактором в случае работы программы на мобильных устройствах. Однако этого можно избежать, переместив второй этап компиляции, анализ и запутывание программ на сервер приложений [2].

2.1. Описание модификаций в стандартных инструментах

Изменения в компиляторе Clang

Компилятор переднего плана CLANG [3] лишь частично поддерживает так называемую "кросс-компиляцию", т.е. запуск на одной архитектуре с генерацией кода для другой. Для обеспечения полной поддержки такого режима работы потребовалось обеспечить возможность указания путей к заголовочным файлам и перекомпилированным библиотекам целевой системы, а также изменить механизм поиска соответствующих файлов. Все необходимые параметры задаются на этапе сборки проекта. Необходимо отметить, что CLANG не имеет стабильной собственной реализации стандартной библиотеки языка Си++ и некоторых других необходимых во время компоновки библиотек, поэтому при сборке программ используются соответствующие библиотеки компилятора GCC[4]. Кроме того, поскольку для обоих вариантов компиляции, описанных в предыдущем разделе, требуется запуск модифицированного компоновщика, были выполнены необходимые изменения в драйвере.

Изменения в компоновщике

Поскольку на этапе компоновки мы можем определить порядок сборки и точные зависимости между модулями программы, были внесены необходимые изменения для сохранения данной информации. Такая информация требуется для формирования корректной очереди компиляции на второй фазе сборки и включения в инстанционный пакет только необходимых файлов. Кроме того, наличие промежуточного представления на этапе компоновки позволяет выполнить дополнительные оптимизации времени связывания (LTO-link time optimizations), область видимости которых будет расширена до нескольких единиц трансляции.

После завершения компоновки модулей, содержащих биткод LLVM, и их оптимизации происходит сохранение объединенного модуля, содержащего биткод, запись всех необходимых ему зависимостей для корректной компиляции на второй фазе сборки и модификация путей к используемым

системным библиотекам. В модули, которые содержат биткод LLVM и соответствуют финальным исполняемым файлам приложения, внедряется дополнительная информация об именах, требуемых для динамической компиляции других модулей, содержащих промежуточное представления LLVM.

Динамический выбор уровня оптимизаций

Механизм динамического переключения между уровнями оптимизации предназначен для решения вопроса экономии времени компиляции на функциях, которые исполняются редко, при одновременном улучшении кода часто используемых функций. Идея состоит в том, чтобы не применять оптимизации для «холодных» функций, поскольку их оптимизация не оказывает существенного влияния на производительность программы. Но поскольку таких функций достаточно много, время, затрачиваемое на их анализ и оптимизацию, может составлять большую часть времени, расходуемого на компиляцию и оптимизацию программы. Такая методика применима как для динамической компиляции, где важна быстрая компиляция, а дополнительная оптимизация может производиться во время работы программы, так и для компиляции больших программных комплексов.

Было реализовано 3 варианта сочетаний наборов оптимизаций:

- минимальный "O0" (не оптимизировать вовсе) для «холодных» и "O2"(стандартный набор) для «горячих»;
- средний "O1" (минимальный набор) для «холодных» и "O3" (агрессивные оптимизации) для «горячих»;
- максимальный – "O2" для «холодных», "O3" для «горячих».

При тестировании на SQLite было выявлено, что при минимальном уровне экономится до 90% времени компиляции при аналогичной производительности. Для среднего уровня экономия составляет 2-5% при производительности, лучшей, чем при обычной компиляции с "O3" на 1-3%, и максимальный уровень дает экономию ~5% при ускорении на 3-4% в сравнении с "O3".

Оптимизация работы ключевых частей инфраструктуры

Дополнительно была проведена оптимизация компонент LLVM для их ускорения их работы и уменьшения потребления памяти, что существенно для встраиваемых архитектур. Для этого был снят и проанализирован профиль выполнения; выяснилось, что в коде присутствовали множественные выделения/освобождения памяти малого размера – порядка нескольких байт, а также неэффективная работа с локальными переменными – массивами. Так, например, во время генерации кода для программы SQLite LLVM выполняет 6098391 операций выделения памяти. Примерно 20% составляют запросы на выделение памяти размером 16 байт, 14% размером 8 байт, 11% размером 48

байт. Для устранения описанных проблем, была применена библиотека DLMalloc и переписан проблемный код. Во время генерации кода для программ на платформе ARM было достигнуто сокращение использования памяти на 1.6-10.9% и времени компиляции на 10-20%.

3. Используемые оптимизации

В настоящем разделе описываются основные оптимизации, выполненные в рамках разработанной инфраструктуры за последние два года. Другие реализованные оптимизации, в частности, на основе профиля программы и исправления кодогенерации LLVM для ARM, можно найти в статье [9].

3.1. Открытая вставка функций

Открытая вставка функций – оптимизирующее преобразование компилятора, вставляющее код функции на место его вызова в тело вызывающей функции. На тестах SQLite, Expedite, Cray и Coremark учет информации о профиле выполнения программы при проведении оптимизации привел к приросту скорости работы ~2%.

3.2. Клонирование блоков

Клонирование блоков - удвоение часто исполняемых базовых блоков графа потока управления, имеющих более одного исходящего ребра и более одного входящего. Суть алгоритма показана на рисунке 2.

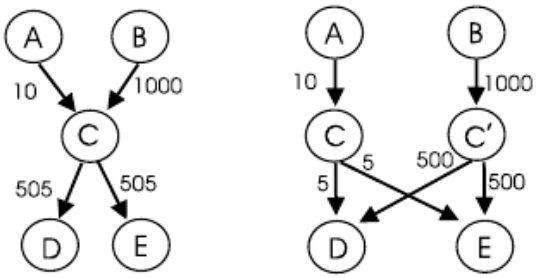


Рис. 2 Клонирование соединенных базовых блоков: слева – до разбиения, справа – после

Данное преобразование не является оптимизирующим само по себе, но позволяет оптимизациям, основанным на анализе потока данных, таким, как удаление загрузок, удаление границ массивов, замещение на стеке и пр., работать более эффективно.

3.3. Вынос "холодных" участков кода в отдельные функции

Для оптимизации предлагается рассматривать функции, которые исполняются наибольшее число раз (”горячие”). Если функцию условно можно разделить на две части: большой редко исполняемый участок кода, относительно малый "горячий" участок. Из таких функций предлагается выносить ”холодную” часть функции в отдельную новую функцию, уменьшая при этом размер рассматриваемой функции и расстояния в памяти между часто исполняемыми участками кода.

Вынос редко исполняемого кода в отдельную функцию [5] повышает эффективность программы за счет более эффективного использования кэша процессора.

На тестах SQLite, Expedite, Cray и Coremark оптимизация дала средний прирост скорости в 0,8%. При использовании ее вместе с оптимизацией открытой вставки функций получен средний прирост в ~3%. Размер исполняемого файла увеличивается на 1- 7% в зависимости от приложения.

3.4. Спекулятивная девиртуализация

Для объектно-ориентированных языков программирования решение о вставке виртуальных функций является проблемой, для решения которой недостаточно знать количество ее вызовов. В программах, написанных на языке Си++, могут быть два типа виртуальных вызовов функций: классические вызовы по указателю на функцию и вызовы виртуальных методов классов. Когда компилятор встречает такие вызовы, он не может определить, какая функция будет вызываться. Чтобы понять, какая функция будет вызвана, необходимо произвести дополнительный анализ [6]. Этот анализ включает в себя: сравнение сигнатур функций, анализ иерархии наследования классов и анализ типов. существующих в точке вызова. Сравнение сигнатур отсекает «неподходящие» по возвращаемому значению и параметрам функции. Анализ иерархии наследования выявляет классы, для которых существует реализация виртуального метода. Анализ существующих в точке вызова объектов рассматривает, объекты каких классов были созданы и еще не уничтожены в момент вызова функции.

Помимо этого, была добавлена возможность инструментирования вызовов виртуальных функций сохранением информации о количестве вызовов конкретной виртуальной функции. Таким образом, используя профиль, мы можем определить наиболее вероятного кандидата на девиртуализацию.

Реализованный алгоритм сочетает в себе вышеописанные методы. После проведения девиртуализации и принятия решения о вставке функции, если оказывается, что кандидат на вставку всего один, вставляется он. Если кандидатов несколько, производится спекулятивная девиртуализация: по данным профилирования: мы можем сказать, какая реализация виртуальной

функции исполнялась наиболее часто, и вставляем инструкцию “if”, в теле которой производится вставка «горячей» функции, а в ветке “else” произведется вызов альтернативной, «холодной» версии функции.

Во время тестирования был отмечен прирост производительности в ~5% при использовании только статического анализа и до ~7% с использованием спекулятивного алгоритма. Тестирование производилось на программе Clucene совместно с оптимизацией вставки функций. Алгоритм успешно проходит синтетические тесты девиртуализации, предложенные сообществом GCC.

3.5. Формирование суперблоков

Классические оптимизации используют статические методы анализа, такие как анализ времени жизни переменных или анализ достигающих определений для обеспечения корректности преобразований кода. Эти методы не различают часто и редко исполняемые пути. Однако часто бывают случаи, когда значение портится на редко исполняемом пути, который существует, например, для обработки событий. В результате невозможно применить оптимизации к часто исполняемым путям, пока редко исполняемые пути не будут исключены из анализа. Это требует точной оценки поведения программы во время исполнения.

Рассмотрим взвешенный граф потока управления, который получился после сбора профиля от отработавшей программы.

Весом ребра является вероятность перехода от одного блока к другому. Для формирования суперблоков мы последовательно находим новую трассу, т.е. последовательность базовых блоков, исполненную чаще других, и копируем хвост трассы для каждого блока, через который можно покинуть трассу. Поиск длится до тех пор, пока он возможен (суперблоки из одного базового блока не имеют смысла).

3.6. Ускорение оптимизаций времени связывания

Распараллеливание этапа LTO (оптимизаций времени компоновки) позволяет существенно сократить общее время компиляции программы, а также уменьшить требования к объему оперативной памяти, что существенно в случае сборки больших программных проектов. На данный момент использование простой эвристики – разбиение на задания с приблизительно равными по размеру наборами модулей биткода – позволило добиться сокращения времени компиляции до 60% (тест Expedite) при использовании 4 потоков, с сохранением производительности выходного приложения. Для малых тестов, таких как Coremark и Cray, сокращение времени сборки находится в пределах погрешности, изменение производительности составляет «-11%» и 12% процентов соответственно.

4. Сервер приложений

Предлагаемый метод двухфазной компиляции позволяет проводить оптимизацию программы с учетом собранного профиля как при динамической компиляции, так и во время простоя системы. Но для мобильных устройств зачастую оптимизация программ на устройстве является затруднительной. Для снижения нагрузки на устройство предлагается использовать специальный сервер приложений. При таком подходе приложения, скомпилированные в промежуточное представление LLVM, будут храниться в специальном облачном хранилище, там же будет происходить генерация бинарного кода и оптимизация программы с учетом информации о ее профиле. Поскольку для каждого приложения будет поступать профиль от нескольких пользователей, то, усреднив полученный набор профилей и проведя машинно-независимую оптимизацию, мы получим промежуточное представление, более полно отвечающее реальным вариантам использования. Используя информацию усредненного профиля, мы можем сократить расходы при компиляции приложений для новых пользователей, а также повысить производительность динамической компиляции для пользователей, использующих ее на своих устройствах.

4.1. Описание механизма применения на сервере приложений профиля, собранного на клиенте

Были реализованы специальные системные сервисы, позволяющие пересылать клиентские данные о профиле выполнения программы на сервер приложений. Клиентская программа осуществляет мониторинг указанной директории и, как только в ней появляются файлы с профилем, производит их пересылку на сервер приложений. Сервер получает данные файлы и перекомпилирует приложение, применяя оптимизации с учетом профиля выполнения программы. После этого оптимизированное приложение пересылается клиенту (рисунок 3).

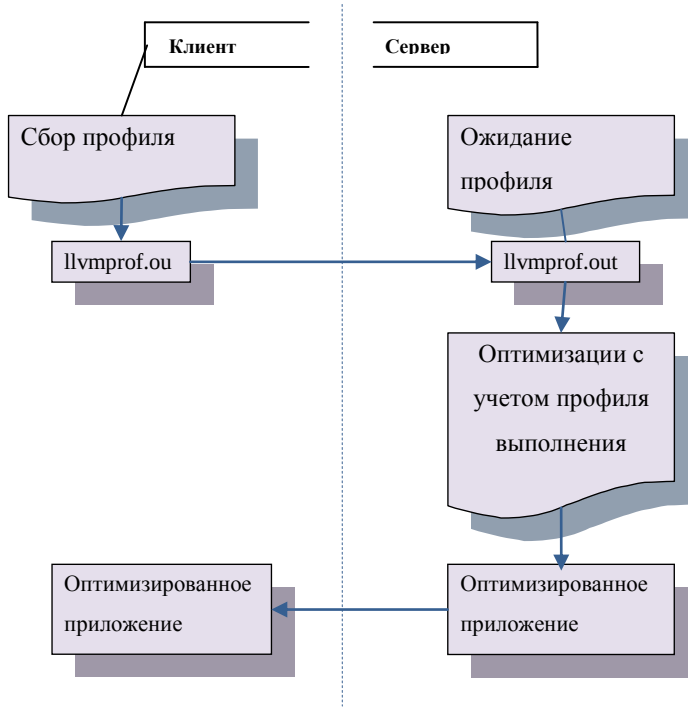


Рис. 3. Схема обмена профилем между клиентом и сервером.

4.2. Описание протокола пересылки файлов

Для реализации протокола используется технология Protocol Buffers [8]. Основное сообщение протокола называется "ProfileData" и содержит информацию об уникальном идентификаторе приложения и о каждом модуле приложения (рисунок 3) . Информация о модуле приложения включает в себя имя файла и данные о профиле выполнения. Поскольку приложения могут состоять из набора модулей, запускаемых в разное время, количество соответствующих полей в сообщении может быть различным.

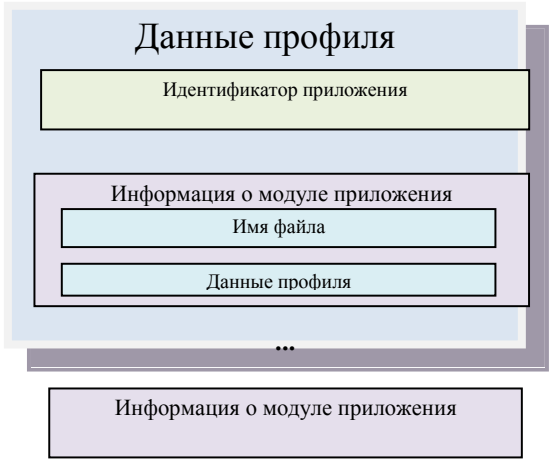


Рис. 4. Описание формата сообщения "Данные профиля"

5. Заключение

В данной статье рассмотрена система двухэтапной компиляции, реализованная на базе LLVM. Описаны изменения, внесенные в компоненты LLVM. Были предложены оптимизации, учитывающие профиль программы. На основе приложенных методов и разработанных технологий можно создать облачное хранилище позволяющее обеспечить как переносимость программ в пределах одной архитектуры, так и учет специфики конкретной аппаратуры, на которой производится развертывание программы.

Список литературы

[1] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[2] А. И. Аветисян. Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения. – Труды ИСП РАН – 2012. - №22. DOI: 10.15514/ISPRAS-2012-22-1.
[3] Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[4] GCC Free software foundation, http://gcc.gnu.org
[5] Peng Zhao "Code and Data Outlining", 2005.
[6] David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[7] Справочное руководство по процессорной архитектуре ARM., http://infocenter.arm.com
[8] Protocol Buffers. URL: https://developers.google.com/protocol-buffers/

[9] Ш.Ф. Курмангалеев. Методы оптимизации Си/Си++ - приложений, распространяемых в биткоде LLVM с учетом специфики оборудования. Труды ИСП РАН, том 24, стр. 127-144, 2013 г. DOI: 10.15514/ISPRAS-2013-24-7.

Applying two-stage LLVM-based compilation approach to application deployment via cloud storage

Sergey Gaissaryan <ssg@ispras.ru> ISP RAS, Moscow, Russia
Shamil Kurmangaleev <kursh@ispras.ru> ISP RAS, Moscow, Russia
Kseniya Dolgorukova <unerkannt@ispras.ru> ISP RAS, Moscow, Russia
Valery Savchenko <sinmipt@ispras.ru> ISP RAS, Moscow, Russia
Sevak Sargsyan <sevaksargsyan@ispras.ru> ISP RAS, Moscow, Russia

Abstract. The process of software distribution via the app stores is as follows: software developer sends application to the online store and it becomes available to the user. To remain competitive software product for a long time, the developer needs to ensure the functioning of the application on most hardware and software platforms. Typically, this problem is solved as follows: an application either compiled and optimized developer repeatedly for every new platform, followed by the addition of a new shop in binary versions, or implemented using a dynamic language.

The paper describes two-stage compilation approach for C/C++ languages that allows deploying application in the LLVM (low level virtual machine) intermediate representation. The LLVM modifications for optimizing code generation time as well as the profile-based optimizations are presented. Also approach is presented and evaluated to dividing the linking process on parallel threads to reduce compile time for large applications.

The specialized application cloud storage architecture is also suggested. We describe implementation of the mechanism for applying profile information, collected on client side on the application server. Also the profile exchange protocol and the specification of message data format are described.

Keywords: Two-stage compilation, program optimization, LLVM, cloud storage.

References

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

[2]. Arutyun Avetisyan. Dvukhehtapnaya kompilyatsiya dlya optimizatsii i razvertyvaniya programm na yazykakh obshhego naznacheniya. [Two-stage compilation for optimizing and deploying programs in general purpose languages]. Trudy ISP RAN [The

Proceedings of ISP RAS], 2012, vol. 22, pp. 11-18. DOI: 10.15514/ISPRAS-2012-22-1. (in Russian).
[3]. Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[4]. GCC Free software foundation, http://gcc.gnu.org
[5]. Peng Zhao “Code and Data Outlining”, 2005.
[6]. David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[7]. ARM architecture., http://infocenter.arm.com
[8]. Protocol Buffers. URL: https://developers.google.com/protocol-buffers/
[9]. Sh.F. Kurmangaleev. Metody optimizatsii Ci/Ci++ - prilozhenij rasprostranyaemykh v bitkode LLVM s uchetom spetsifiki oborudovaniya [Machine-specific optimization methods for C/C++ applications that are distributed in the LLVM intermediate representation format], ISP RAN [The Proceedings of ISP RAS], 2013, vol. 24, pp. 127-144, 2013. DOI: 10.15514/ISPRAS-2013-24-7. (in Russian).

Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM

Виктор Иванников <ivan@ispras.ru>,
Шамиль Курмангалеев <kursh@ispras.ru>,
Андрей Белеванцев <abel@ispras.ru>,
Алексей Нурмухаметов <oleshka@ispras.ru>,
Валерий Савченко <sinmipt@ispras.ru>,
Рипсиме Матевосян <hripsime@ispras.ru>,
Арутюн Аветисян <arut@ispras.ru>

Аннотация. В статье описываются разработанные в ИСП РАН методы запутывания программ, направленные на противодействие методам статического анализа программ. Рассматриваемые методы запутывания реализованы в обфусцирующем компиляторе на базе LLVM. Приводится оценка замедления и увеличения объема потребляемой памяти.

Ключевые слова: обфускация, LLVM, непрозрачный предикат.

1. Введение

В настоящее время актуальна задача защиты программ, как от статического, так и от динамического анализа кода. Доступность качественных средств анализа кода и большой выбор подключаемых модулей, в автоматическом режиме обходящих многие приемы противодействия анализу, понижают планку требований к квалификации аналитика, что ведет к повышению требований к защите программ. Необходимо использовать либо методы противодействия анализу, неизвестные широкому кругу лиц, либо использовать трудоемкие для анализа преобразования.

Оптимальным выбором, позволяющим реализовать стойкие варианты запутывания программ, является создание обфусцирующего компилятора на базе одной из существующих компиляторных инфраструктур. С одной стороны, это позволит производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, а с другой позволит сосредоточиться на разработке защиты, а не на создании требуемой инфраструктуры. Кроме того, такой подход обеспечивает поддержку нескольких архитектур при условии совпадения порядка байтов и

минимального различия в ABI, а также различающийся двоичный образ программы для каждого пользователя, если ввести зависимость заданного набора преобразований от некоторого уникального ключа.

При разработке преобразований необходимо учитывать следующие критерии эффективности:

- Маскирующее преобразование должно затрагивать и поток управления, и поток данных запутываемой программы;
- Стойкость преобразования должна основываться на алгоритмически сложных задачах, например, требовать от атакующего применения анализа указателей для точного восстановления потоков данных защищенной программы [9];
- При разработке преобразования нужно учитывать особенности работы средств анализа [10], например, для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

Компиляторная инфраструктура, на базе которой будет разрабатываться запутывающий компилятор, должна удовлетворять следующему набору требований:

- обеспечивать компиляцию исходных кодов на C/C++ под Windows и Linux;
- иметь открытые исходные коды;
- иметь документацию и поддержку сообщества;
- расширяемость;
- возможность влиять на генерируемый код на любом этапе компиляции, от препроцессора до генерации кода;
- возможность получить различную информацию об обрабатываемой программе на любой стадии компиляции, требуемую для реализации алгоритмов запутывания кода.

LLVM [1] – компиляторная инфраструктура с открытыми исходными кодами, удовлетворяющая перечисленным требованиям и поддерживающая множество целевых архитектур (x86, ARM, MIPS, PowerPC и др.). Промежуточное представление (LLVM IR) машинно-независимого уровня играет центральную роль в процессе компиляции. Все оптимизации реализованы как компиляторные проходы преобразования LLVM IR. Анализ кода может быть реализован как отдельный проход, а его результаты могут разделять несколько проходов, трансформирующих код. Все машинно-зависимые оптимизации происходят отдельно для каждой машины на собственном внутреннем представлении низкого уровня.

В настоящей статье в разделе 2 предлагается обзор существующих решений в области запутывающих компиляторов. Раздел 3 описывает разработанные

методы запутывания. Раздел 4 содержит экспериментальные результаты, а раздел 5 завершает статью.

2. Обзор существующих запутывающих компиляторов

Среди существующих проектов запутывающих компиляторов можно выделить два основных типа. Первые из них предоставляют возможность запутывания программ с применением широкого набора разных взаимодополняющих методов, а другие разработаны для реализации и проверки одного конкретного метода запутывания. Опишем самые известные примеры компиляторов обоих типов.

Совместный исследовательский проект HES-SO/RCSO под руководством Pascal Junod с названием “Obfuscator” [2]. Одна из его частей основана на LLVM [1] и преобразует промежуточное представление LLVM, а другая производит преобразование бинарной программы для архитектуры ARM. На уровне промежуточного представления реализованы некоторые классические методы запутывания. Арифметические и логические инструкции заменяются на эквивалентные им выражения, состоящие из последовательности нескольких инструкций.

Граф потока управления усложняется путем вставки ложных ветвлений, закрытых непрозрачными предикатами. При этом базовый блок разбивается на две части и после первой части вставляется непрозрачный предикат. На всегда выполняющуюся ветвь этого условия помещается вторая часть базового блока, а на ту, которая не выполняется никогда, помещается некоторый мусорный код. С помощью такой конструкции в программу вставляются несводимые участки графа потока управления. Такой подход отличается от обычного метода получения несводимых графов путем добавления дополнительного входа внутрь тела цикла, например, прикрытого непрозрачным предикатом.

Кроме того, реализовано преобразование диспетчера, трансформирующее граф потока управления программы. Эта реализация в целом соответствует подходу, описанному в [12], но имеет небольшие улучшения, касающиеся диспетчеризации управляющих структур (if-then-else, for-loops, switch). Они диспетчеризуются не как единое целое, а разделяются на свои составные части (например, ветви условного оператора), каждой из которых соответствует свое собственное значение переменной диспетчеризации. Оценки производительности приводятся на примере бенчмарка libtomcrypt. Преобразование диспетчеризации увеличивает на 15% размер кода и на треть замедляет производительность.

Другой пример запутывающего компилятора общего назначения – это проект confuse под руководством Chih-Fan Chen et al [3]. В нем каждый метод запутывания реализован как отдельный компиляторный проход,

преобразующий промежуточное представление. Всего представлены три метода: обфускация строк, вставка излишнего кода, запутывание графа потока управления. Обфускация строк происходит в условных операторах, в которых сравниваются значение строковой переменной с константной строкой. Константная строка заменяется на значение ее хэша и больше не содержится в программе, а вместо переменной подставляется вызов хэш-функции от этой переменной. Вставка излишнего кода основана на математических тождествах, позволяющих заменить какую-нибудь простую арифметическую операцию на длинную цепочку излишних вычислений. Комбинирование различной последовательности подобных тождественных замен арифметических выражений на эквивалентные используются для видоизменения кода.

Подход к запутыванию графа потока управления базируется на стандартном методе, описанным еще Коллбергом в [12]. Берется базовый блок и рассекается на две части, а между ними вставляется предикат. Если ставится непрозрачный предикат, то на его всегда исполнимую ветвь ставится вторая часть базового блока либо ее запутанная некоторой последовательностью предыдущих преобразований версия. Ветвь, которая никогда не выполняется, можно оставить пустой или же поместить на нее любой код. Если же ставится прозрачный предикат, то на обе его ветви помещается код, семантически эквивалентный коду из второй части рассеченного базового блока. Это код видоизменяется посредством применения к нему некоторого набора предыдущих оптимизаций. Кроме того, непрозрачные предикаты соединяются с условиями выхода из цикла для их запутывания.

Последним запутывающим компилятором общего назначения рассмотрим коммерческий morpher[4]. По заявлениям разработчиков, в нем реализовано самое большое количество запутывающих преобразований: зацепление дуг графа потока вызовов (CFG arches meshing), клонирование базовых блоков и функций, вставка непрозрачных предикатов, зацепление функций, вставка псевдоциклов длиной 1 в линейную последовательность инструкций, расшифровка и зашифровка констант до и после использования в программе.

Из второй группы запутывающих компиляторов наибольший интерес представляют те, в которых применяются нестандартные методы запутывания. В статье [5] предлагается метод запутывания потока управления программы путем превращения ее в многопоточное приложение. Для передачи управления между потоками используется диспетчер, гарантирующий сохранение исходной семантики последовательной программы. Предложенный метод реализован в рамках инфраструктуры LLVM. Он не зависит он других методов запутывания потока управления, поэтому его можно комбинировать с другими метода для увеличения сложности.

Интересный подход запутывания вредоносных программ реализован в работе Teja Tamboli [6]. На вход компилятору подается исходный код вредоносной программы вместе с исходным кодом какого-нибудь обычного приложения.

Полученные биткоды обоих файлов смешиваются на этапе компоновки путем вставки функций из биткода обычной программы в биткод вредоносного приложения. На выходе получается бинарный код вредоносного приложения, похожий на код обычной программы.

В конце обратимся к методу, описанному в статье [7]. Описывается подход к автоматической защите тригерных вредоносных приложений от средств современного антивирусного анализа, так называемый метод обфускации условного кода. Запутыванию подвергаются условные ветвления с условием, удовлетворяющим некоторым условиям. Пример – сравнение строки, полученной приложением во время исполнения, с некоторым указанным в условии значением. Сравнение самих строк заменяется на сравнение их хэшей. Базовый блок находящийся внутри условия шифруется. В качестве ключа шифрования используется значение из условия. Кроме того, перед ним вставляется функция дешифрования, которая будет во время исполнения расшифровывать зашифрованный базовый блок. В качестве ключа будет использоваться значение, полученное во время исполнения. При реализации использовалась компиляторная инфрастуктура LLVM и утилита анализа и модификации бинарного кода DynInst [8]. В промежуточном представление LLVM находились пригодные для запутывания участки программы. В логическом выражении условия производилась замена значений на их хэши, а внутрь условия вставлялась функция дешифрования и ключ шифрования со специальным маркером. После кодогенерации над бинарным кодом с помощью утилиты DynInst производилась шифровка и удаления из тела условия ключа и специального маркера.

3. Разработанные методы усложнения программного кода

Были разработаны следующие методы усложнения:

- Перенос локальных переменных в глобальную область видимости;
- Шифрование константных строк, используемых программой;
- Вставка в код фиктивных циклов;
- Приведение графа потока управления к плоскому виду с применением алгоритма диспетчеризации;
- Переплетение нескольких функций в одну с заменой всех вызовов отдельных функций на вызов одной общей;
- Сокрытие вызовов функций. Для защищаемой функции создается функция-переходник, внутри которой содержится несколько вызовов различных функций;
- Создание несводимых участков в графе потока управления;
- Разбиение констант;
- Клонирование функций;
- Формирование непрозрачных предикатов.

Рассмотрим подробней предложенные методы.

3.1. Перенос локальных переменных в глобальную область видимости

Перенос локальных переменных в глобальную область видимости с последующим их использованием в разных функциях производится с целью затруднить точный анализ потоков данных в программе.

В общем случае нельзя изменять значения переменных, вынесенных из других функций в произвольном месте программы, так как это может привести к неправильному выполнению компилируемой программы. Поэтому строится граф вызовов для всех функций в модуле, затем для каждой функции вычисляется множество переменных, модификация которых не нарушит работоспособность программы. Такими переменными будут переменные, вынесенные из функций, расположенных на разных путях в дереве вызовов. После формирования множеств подходящих переменных осуществляется добавление мусорного кода, использующего для вычислений «безопасные» переменные. Также найденные переменные используются в предикатах. Функции, передаваемые по адресу в другие функции, не обрабатываются, так как они могут использоваться в многопоточном коде.

При восстановлении алгоритма работы программы используется построение слайсов программы. Выполняется отбор тех операторов программы, выполнение которых влияет на выходные данные или на выполнение которых повлияли входные данные. Во время статического анализа для переменных, расположенных в глобальной области памяти, требуется проводить межпроцедурный анализ.

3.2. Шифрование строк

Во время статического анализа строковые константы, хранящиеся в открытом виде, могут дать аналитику дополнительную информацию о функционировании программы или помочь найти интересующий код по строкам, выводимым во время интересующего его события. Преобразование, маскирующее строковые константы, предназначено для сокрытия информации о строках во время статического анализа программы.

Шифрование строк выполняется следующим образом: вначале все константные строки, кроме тех, что содержатся в агрегатных типах (массивы, контейнеры из стандартной библиотеки), шифруются, в модуль добавляются шифрующая и дешифрующая функции. Перед каждым использованием той или иной строки вставляется вызов функции дешифратора, а после – шифрующей функции. Это справедливо для строк, для которых не выполняются операции с указателями. Если же такие операции имеют место, то для корректной работы запутывающего алгоритма необходим анализ указателей. В таких случаях обратного шифрования строки не производится. Шифрование строк после использования требуется для того, чтобы во время

работы программы все строки не находились в памяти расшифрованными. Шифрование строк производится с помощью операции XOR со случайным ключом.

3.3. Вставка фиктивных циклов

Фиктивный цикл – цикл, в котором никогда не происходит более одной итерации. В коде запутываемой программы происходит поиск участков кода, по структуре напоминающих одну итерацию цикла. Далее в начало участка или в его конец (в зависимости от типа фиктивного цикла) вставляется базовый блок с условным переходом в противоположный конец участка. Условный переход содержит в себе непрозрачный предикат, который и маскирует лишь одно исполнение цикла. В качестве подходящего участка рассматривается участок с одним входом и выходом.

3.4. Преобразование “диспетчер”

Идея маскирующего преобразования «диспетчер» заключается в преобразовании графа потока управления таким образом, что статический анализ переходов между базовыми блоками становится трудной задачей [11]. При этом базовым блокам присваиваются номера. В начало функции вставляют блок «диспетчер» – аналог switch в языке Си. В конец каждого блока дописывается код, устанавливающий номер следующего блока для выполнения и передающий управление на блок-диспетчер, в котором принимается решение, куда дальше передать управление.

Для каждого блока делается до 5 копий, которые так же добавляются в диспетчер. Помимо этого, производится усреднение размера базовых блоков, инструкции “`call`” оцениваются как несколько инструкций, так как передача параметров в промежуточном представлении LLVM производится в той же команде, а на реальных архитектурах по команде на аргумент. Для сокрытия переменной диспетчеризации её значение вычисляется по формуле `I = X1 XOR Z`; а следующее значение Z по формуле `Zслед = X2 XOR Zтекущее`; Z, X1 и X2 выбираются случайным образом для блока, предшествующего диспетчеру, и X2 генерируется случайным образом для каждого блока исходной функции во время его обработки. В каждом блоке выбирается одна переменная подходящего типа, с которой посредством операции XOR происходит сцепление переменной диспетчеризации. Такое преобразование затруднит автоматическое выделение переменной диспетчеризации, так как в ее вычисление будут вовлечены живые переменные, вычисляемые в программе.

3.5. Переплетение функций

Классический подход к переплетению функций обладает малой стойкостью. Он предполагает объединение сигнатур функций и наличие параметра, по которому происходит диспетчеризация [12]. Восстановить исходный код

переплетенных таким образом функций не составляет особого труда. Предложена модификация упомянутого алгоритма таким образом, чтобы, помимо диспетчеризующего условия, переплетаемые функции имели точки пересечения потоков управления и потоков данных. Тогда применение алгоритма обратного слайсинга [13] не позволяет найти единственную точку, в которой производится выбор рабочей функции.

Переплетение происходит следующим образом:

1) Объединяются сигнатуры двух функций, генерируется дополнительный параметр, по которому в процессе выполнения будет производиться выбор функции.
2) Если функции возвращают целочисленное значение, то для реального возврата значения из переплетенной функции используются глобальные перемененные, а сама функция возвращает неиспользуемое значение. Если функции возвращают указатели, то тип возвращаемого значения переплетенной функции становится указателем на void.
3) В новой функции, полученной на основе переплетения двух функций, произвольно выбираются по одному блоку из каждой функции, затем над ними производится преобразование зацепления дуг [14]. В генерируемом общем базовом блоке производятся вычисления с глобальными переменными. Для затруднения анализа потоков данных эти переменные используются для вычислений в и других функциях модуля. Таким образом, у двух переплетенных функций всегда будут общие вычисления. Результат вычислений используется в качестве возвращаемого значения, а также записывается в глобальную переменную, что не позволит исключить добавленные вычисления как мертвый код, результат которого нигде не используется.
4) После генерации переплетенной функции все места вызова оригинальных функций заменяются на вызов переплетенной функции с генерацией дополнительных параметров и изменением обработки возвращаемого значения.

3.6. Сокрытие вызов функций

Преобразование применяется для маскировки вызовов функций, поскольку знание имени вызываемой функции облегчает восстановление алгоритма работы программы. Для маскируемого вызова создается функция-переходник, внутри которой содержится несколько вызовов функций. Аргументы в переходник передаются в измененном виде, после преобразования с помощью битовой операции XOR. Внутри переходника вызов нужной функции диспетчеризуется по значению трудного предиката. Реализовано два варианта

преобразования: только для вызовов внешних функций и для вызовов всех функций.

Для каждого вызова функции производится его замена на вызов функции-переходника. Чтобы избежать чрезмерной вложенности вызовов, переходники на переходники не создаются. Затем для каждой функции создается сортированный список ее аргументов. Для выбора функций, которые будут размещены внутри переходника, была введена мера "близости функций" – число от 0 до 1, которое показывает, насколько функции близки друг к другу по сигнатуре. 1 означает, что функции имеют набор аргументов с одинаковыми типами, 0 означает, что таких аргументов у функций нет. Значение меры – коэффициент Жаккара (Jaccard) для множеств типов аргументов двух функций:

$$\Gamma(f_1, f_2) = \frac{|TypeArgs(f_1) \cap TypeArgs(f_2)|}{|TypeArgs(f_1) \cup TypeArgs(f_2)|}$$

Половина функций в переходнике выбираются, как самые "похожие" по введенной мере, другая половина как "непохожие". После того, как функции были отобраны, для каждой из них производится попытка замены вызовов на переходник. Просматриваются все использования функции в пределах обрабатываемого модуля и отбираются те из них, которые являются непосредственным вызовом функции (call), либо вызовом с возможностью обработки исключений (invoke). Другие использования функции, например, её передача в качестве аргумента, пропускаются.

Функция-переходник принимает аргументами объединение аргументов всех функций внутри переходника, первый аргумент используется для получения истинных значений аргументов, а последний для диспетчеризации нужного вызова с помощью непрозрачного предиката. Аргументы, передаваемые в функцию-переходник, запутываются с помощью битовой операции XOR. Все аргументы преобразуются в тип данных длинной 64 бита (если аргумент имеет больший размер, то он передается как есть, без преобразования) и между всеми аргументами применяется операция XOR, обозначим результат операции за S. Затем к каждому аргументу применяется операция XOR с S, и в таком виде аргумент передается в функцию. Также для распутывания передается само значение S. Внутри функции-переходника происходит распутывание аргументов. Затем вычисляется непрозрачный предикат *P*, по результату которого происходит диспетчеризация вызова функций, основанный на китайской теореме об остатках. Вычисление предиката встраивается в функцию-переходник.

Диспетчеризация вызовов функций производится с помощью большого switch блока. Каждое значение в нем сгенерировано случайным образом и соответствует какой-либо функции. Каждой функции передаются те аргументы, которые соответствуют ей по типу. Все вызовы перемешиваются в

случайном порядке внутри функции-переходника. Для диспетчеризации используется последний аргумент функции-переходника, передаваемое в упомянутый выше непрозрачный предикат *P*: можно подобрать такое значение аргумента, которое соответствовало бы нужной функции. Это значение генерируется случайным образом.

Так как переходник включает в себя и похожие, и непохожие друг на друга функции, то часто количество его аргументов превышает количество аргументов, реально необходимых для вызова защищенной функции. Остальные аргументы берутся из глобальной области видимости. Если в глобальной области видимости переменных с таким типом нет, то они создаются. Внутри функции-переходника они могут быть использованы при вычислении непрозрачного предиката, а также для запутывания могут передаваться в другие вызовы, которые не будут использованы.

3.7. Формирование непрозрачных предикатов

Предикатом является базовый блок или несколько базовых блоков, имеющих один общий терминальный базовый блок. Терминальный базовый блок предиката заканчивается инструкцией условного перехода, которая всегда передает управление только по одной ветке. Причем, основываясь на информации, доступной на этапе компиляции, известно, по какому пути произойдет переход.

Реализована вставка предиката двумя различными способами: после указанного базового блока или добавление инструкций по вычислению предиката в указанный базовый блок. После вставки предиката модифицируются φ-функции в базовых блоках, которые используются в терминальном условном переходе. φ -функция должна быть определена для каждого базового блока, являющегося предшественником данного. Для новых предшественников (базовых блоков предиката) φ-функция доопределяется нулевым значением.

Также реализованы интерфейсы для автоматической генерации предикатов. В запутывающих преобразованиях используются три типа предикатов:

1. Выражения, которые могут быть как истинны, так и ложны в зависимости от выбранных параметров, например, проверка истинности диофантова уравнения $x^2 - n * y^2 = 1$. Если параметр n не является точным квадратом, то это уравнение Пелля. При вставке этого предиката случайным образом выбирается, будет ли он всегда иметь истинное значение либо ложное.
2. Выражения, которые всегда истинны, например, уравнение $(x^3 - x)\, mod\, 3 = 0$. Значение переменной x для вычисления

значения предиката выбирается случайным образом среди целочисленных глобальных переменных. Если таких глобальных переменных нет, то для вычисления предиката используется случайная целочисленная константа.

3. Выражения, которые всегда ложны, например, целочисленное уравнение $7 * y^2 - 1 = x^2$. Значения переменных x и y выбираются так же, как и ранее.

3.8. Другие преобразования

В данном разделе описываются преобразования генерации несводимых участков графа потока управления, разбиение констант, клонирование функций и внесение зависимости от ключа пользователя.

Генерация несводимых участков в графе потока управления применяется для затруднения работы автоматических декомпиляторов. Колберг [12] описывает алгоритм, который приводит граф потока управления к несводимому. Для каждого цикла добавляется «фиктивное» ребро из заголовка цикла в его тело. Добавление такого ребра осуществляется с помощью вставки непрозрачных предикатов.

Предложена модификация упомянутого алгоритма: для всех циклов функции добавляются «фиктивные» ребра из одного цикла в другой. Недостаток такой трансформации состоит в том, что она эффективно запутывает только код функций, содержащих несколько циклов. Поэтому дополнительно производится следующая трансформация: для множества блоков функции выбирается N блоков и между ними случайно добавляются ребра. Фиктивные переходы защищаются непрозрачными предикатами.

Часто в коде в явном виде встречаются константы, характерные для определенных алгоритмов, например константа 0x67452301 для MD5. Поиск констант позволяет определить используемый алгоритм, что упрощает анализ программы. Для противодействия предложен алгоритм разбиения констант. Разбиваются только константы, бóльшие единицы. Для разбиения случайным образом выбирается число меньшее исходного, которое будет выступать в качестве первого слагаемого, второе слагаемое получается автоматически.

При клонировании функций для каждого использования функции внутри программного модуля производится создание своего экземпляра вызываемой функции. Для каждого вызова будет сгенерирована копия тела функции, затем этот вызов будет исправлен на вызов соответствующей копии. Такое преобразование увеличивает размер кода программы и время, требуемое для его автоматического анализа. После применения клонирования совместно с другими запутывающими преобразованиями, зависящими от генератора случайных чисел, функции утратят полную идентичность, что повысит сложность анализа, так как потребуется проанализировать каждую копию.

Чтобы обеспечить одинаковую работу на всех поддерживаемых платформах без зависимости от библиотечной реализации генератора случайных чисел, была реализована поддержка собственной версии генератора. Реализация использует линейный конгруэнтный генератор: при заданном стартовом числе X_0 следующее определяется по формуле: $X_{n+1} = (A * X_n + C) \bmod M$. В качестве параметров генератора выбраны значения, используемые в библиотеке `glibc`. Вместо реализации по умолчанию возможно использование любого другого алгоритма.

Ключ пользователя выступает в роли «затравки» для генератора, позволяя управлять недетерминизмом в преобразованиях, накладывая на программу уникальный для каждого пользователя характер изменения программы.

4. Экспериментальные результаты

Произведем оценку увеличения объема и уменьшения быстродействия запутанной подпрограммы. Во время обфускации на уровне промежуточного представления кода генерируется количество дополнительных инструкций, например, инструкция `call` внутреннего представления LLVM разворачивается в несколько инструкций, одна из которых – это непосредственно вызов функции, а остальные – это инструкции передачи аргументов и обработки возвращаемого значения. Помимо этого, оптимизатор LLVM в зависимости от кода программы и набора оптимизаций может генерировать различный бинарный код для одних и тех же инструкций промежуточного представления. Кроме того, количество примененных преобразований зависит от входной программы (например, если в программе нет циклов, то и переплетать нечего). Поэтому представляется затруднительным дать точную оценку замедления для произвольной программы. В целях практического использования используются коэффициенты, полученные опытным путем с использованием достаточно большой базы программ.

В практических целях был произведен замер замедления на тестах из пакета OpenSSL 1.0.1 (таблица 1).

Таблица 1. Параметры замедления и увеличения потребления памяти

Метод	**Замедление программы**	**Увеличение потребления памяти**
Клонирование функций	1,20	1,10
Переплетение функций	1,20	1,10
Шифрование строк	5,00	1,05
Вставка фиктивных циклов	1,20	1,10
Разбиение констант	1,20	1,05
Сокрытие вывозов внешних функций	5,00	1,50
Сокрытие вывозов всех функций	8,00	1,70
Диспетчер	5,50	2,50
Генерация несводимых участков в графе потока управления	1,20	1,05
Перенос локальных переменных в глобальную область видимости	1,20	1,05

Совместное применение нескольких опций позволит увеличить сложность пропорционально произведению увеличения сложностей каждого преобразования в отдельности. Для примерной оценки сложности анализа был проведен эксперимент.

К программе SQLite были применены преобразования: переплетение функций, перенос локальных переменных в глобальную область видимости, преобразование «Диспетчер», сокрытие вызовов функций. Размер кода приложения увеличился с 2.9 МБ до 15 Мб. Потребление памяти дизассемблером Ida Pro возросло в ~10 раз, время анализа по сравнению с оригинальным кодом возросло примерно в 10 раз. Кроме того, некоторые версии Ida Pro оказались неспособны закончить анализ, поскольку во время работы возникает исключение в одной из библиотек, и программа аварийно завершает работу. Следует отметить, что декомпилятор Hex-Rays с задачей также не справился.

Также было произведено исследование с помощью инструмента комбинированного анализа TrEx [15]. Полученные результаты демонстрируют, что обеспечиваемый уровень защиты сравним с уровнем, обеспечиваемым коммерческими разработками.

5. Заключение

В статье описана реализация запутывающего компилятора на базе инфраструктуры LLVM. Проведен обзор аналогов созданного компилятора. Предложен и реализован набор как новых, так и известных методов запутывания. Приведены результаты тестирования средствами статического и динамического анализа.

Список литературы

[1] The LLVM Compiler Infrastructure. http://LLVM.org/

[2] Obfuscator reloaded, Application Security Forum – Western Switzerland, November 7th, 2012, Yverdon-les-Bains, Switzerland. http://crypto.junod.info/obfuscatorwf12_talk.pdf

[3] Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, Adrian Tang. Confuse: LLVM-based Code Obfuscation. http://www.cs.columbia.edu/~aho/cs4115_Spring-2013/lectures/13-05-16_Team11_Confuse_Paper.pdf

[4] Обфускатор Morpher. http://morpher.com/

[5] Rasha Salah Omar, Ahmed El-Mahdy, Erven Rohou, Thread-Based Obfuscation through Control-Flow Mangling, arXiv:1311.0044

[6] Tamboli, Teja, "Metamorphic Code Generation from LLVM IR Bytecode" (2013). Master's Projects. http://scholarworks.sjsu.edu/etd_projects/301/

[7] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. Informatica, 2008.

[8] Инструмент Dyninst. http://www.dyninst.org/dyninst

[9] Д. А. Щелкунов. Применение запутывающих преобразований и полиморфных технологий для автоматической защиты исполняемых файлов от исследования и модификации. Труды международной конференции РусКрипто. Апрель 2008 г.

[10] А.В. Чернов. Анализ запутывающих преобразований программ. Труды ИСП РАН, том 3, 2002, стр. 7-38.

[11] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages

[12] C. Collberg, C. Thomborson, D. Low. A Taxonomy of Obfuscating Transformations. Departament of Computer Science, the University of Auckland, 1997. URL: http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a

[13] Frank Tip. "A survey of program slicing techniques". Journal of Programming Languages, Volume 3, Issue 3, pages 121–189, September 1995.

[14] А. В. Чернов. Об одном методе маскировки программ. Труды Института системного программирования РАН, том 4, 2003, стр. 85-119.

[15] М.Г. Бакулин, С.С. Гайсарян, Ш.Ф. Курмангалеев, И.Н. Ледовских, В.А. Падарян, С.М. Щевьева. Динамический анализ обфусцированных приложений с диспетчеризацией или виртуализацией кода. Труды Института системного программирования РАН, том 23, 2012, стр. 49-66.2008. DOI: 10.15514/ISPRAS-2012-23-3.

Implementing Obfuscating Transformations in the LLVM Compiler Infrastructure

Victor Ivannikov ivan@ispras.ru ISP RAS, Moscow, Russia
Shamil Kurmangaleev <kursh@ispras.ru> ISP RAS, Moscow, Russia
Arutyun Avetisyan <arut@ispras.ru> ISP RAS, Moscow, Russia
Andrey Belevantsev <abel@ispras.ru> ISP RAS, Moscow, Russia
Alexey Nurmukhametov <oleshka@ispras.ru> ISP RAS, Moscow, Russia
Valery Savchenko <sinmipt@ispras.ru> ISP RAS, Moscow, Russia
Hripsime Matevosyan <hripsime@ispras.ru> ISP RAS, Moscow, Russia

Abstract. Actual task is protecting programs from reverse engineering. The best choice to implement a resistant obfuscation is to create obfuscating compiler based on one of the existing compiler infrastructures. On the one hand, it will produce obfuscated program, with full information about it at all stages of compilation, and the other allows you to focus on the development of protection, rather than on creating the infrastructure required. In addition, this approach provides support for multiple architectures, as well as introduces watermarks for binary images of the program for each user depending from a unique key.

The paper describes the methods for obfuscating C/C++ programs to prevent applying static analyzers to them. Paper observes existing obfuscating compilers. The proposed transformations are based on well-known obfuscation algorithms (including constant string protection, fake cycle insertion, control flow graph flattening, functions merge, function call encapsulation, control flow graph structure obfuscation, opaque predicate insertion and other) and they are specifically improved to resist better to static analysis deobfuscation techniques. The methods are implemented within the LLVM (low level virtual machine) compiler infrastructure. Experimental results presenting resulting program slowdown and used memory growth are given.

Keywords: Obfuscation, LLVM, opaque predicates.

References

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/

[2]. Obfuscator reloaded, Application Security Forum – Western Switzerland, November 7th, 2012, Yverdon-les-Bains, Switzerland. http://crypto.junod.info/obfuscatorwf12_talk.pdf
[3]. Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, Adrian Tang. Confuse: LLVM-based Code Obfuscation. http://www.cs.columbia.edu/~aho/cs4115_Spring-2013/lectures/13-05-16_Team11_Confuse_Paper.pdf
[4]. Morpher http://morpher.com/
[5]. Rasha Salah Omar, Ahmed El-Mahdy, Erven Rohou, Thread-Based Obfuscation through Control-Flow Mangling, arXiv:1311.0044
[6]. Tamboli, Teja, "Metamorphic Code Generation from LLVM IR Bytecode" (2013). Master's Projects. http://scholarworks.sjsu.edu/etd_projects/301/
[7]. Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. Informatica, 2008.
[8]. Dyninst. http://www.dyninst.org/dyninst
[9]. D. A. Shhelkunov. Primenenie zaputyvayushhikh preobrazovanij i polimorfnykh tekhnologij dlya avtomaticheskoj zashhity ispolnyaemykh fajlov ot issledovaniya i modifikatsii. [Applying obfuscation transformations and polymorphic technologies for automatic protection executable files from analysis and modification]. Trudy mezhdunarodnoj konferentsii RusKripto. [Proceedings of international conference RusCrypto]. April 2008 (in Russian).
[10]. A.V. Chernov. Analiz zaputyvayushhikh preobrazovanij programm. [Analysis obfuscating program transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol.3, pp. 7-38 (in Russian).
[11]. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages
[12]. C. Collberg, C. Thomborson, D. Low. A Taxonomy of Obfuscating Transformations. Departament of Computer Science, the University of Auckland, 1997. URL: http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a
[13]. Frank Tip. "A survey of program slicing techniques". Journal of Programming Languages, Volume 3, Issue 3, pages 121–189, September 1995.
[14]. A.V. Chernov Ob odnom metode maskirovki programm [About one method program masking], Trudy ISP RAN [The Proceedings of ISP RAS], 2003, vol.4, pp. 85-119 (in Russian).
[15]. M.G. Bakulin, S.S. Gaissaryan, Sh.F. Kurmangaleev, I.N. Ledovskikh, V.A. Padaryan, S.M. Shchevyeva. Dinamicheskij analiz obfustsirovannykh prilozhenij s dispetcherizatsiej ili virtualizatsiej koda. [Dynamic analysis of virtualization- or dispatching-obfuscated applications]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 49-66. DOI: 10.15514/ISPRAS-2012-23-3. (in Russian).

Оптимизация приложений для заданных статических компиляторов и целевых архитектур: методы и инструменты

Дмитрий Мельник, Шамиль Курмангалеев, Арутюн Аветисян,
Андрей Белеванцев Дмитрий Плотников, Мамикон Варданян
<dm@ispras.ru>, <kursh@ispras.ru>, <arut@ispras.ru>,
<abel@ispras.ru>,<dplotnikov@ispras.ru>, <mamikon@ispras.ru>.

Аннотация. В статье рассматривается рабочий цикл оптимизации производительности приложений для заданной процессорной архитектуры на примере открытых компиляторов GCC и LLVM. Приводятся примеры выполненных оптимизаций и их результаты тестирования на известных наборах тестов. Также описывается инструмент TACT автоматической настройки компилятора на заданное приложение и его примерное использование как прикладным разработчиком, так и компиляторным инженером, приводятся результаты работы инструмента.

Ключевые слова: оптимизация программ; GCC; LLVM; автоматическая настройка компилятора.

1. Введение

Статическая компиляция по-прежнему является основным автоматическим способом повышения производительности программ на языках общего назначения (Си/Си++). Интересно, что актуальность наличия оптимизирующего статического компилятора не только не падает, но и возрастает. Во-первых, появление новых процессорных архитектур влечет за собой необходимость не только переноса на них существующих промышленных компиляторов (GCC, LLVM), но и разработки специфических и настройки имеющихся оптимизаций (распределения регистров, программной конвейеризации, использования особых возможностей набора команд архитектуры). Во-вторых, возрастающая сложность программ обеспечивает необходимость разработки масштабируемых инфраструктур межпроцедурных оптимизаций времени компоновки (LTO). В-третьих, возникает необходимость использовать оптимизации с учетом профиля программы без затрат на инструментирование, с помощью снятия профиля программы через взятие проб (sampling) во время выполнения на нагрузках промышленного типа. В-четвертых, необходима поддержка новых стандартов программирования для эффективного использования современных

многоядерных и гетерогенных архитектур (OpenMP, OpenCL, OpenACC, Cilk+). Наконец, все вышеперечисленное имеет смысл делать лишь в современных открытых компиляторных инфраструктурах типа GCC и LLVM, которые развиваются десятки лет и уже содержат все необходимые базовые машинно-зависимые и машинно-независимые оптимизации, а также средства для их создания.

Так как указанные компиляторы являются многоплатформенными, возникают естественные сложности при их использовании на архитектурах, не являющихся основным фокусом развития (ARM, MIPS и т.д.). Особенности архитектур могут приводить к тому, что машинно-независимые оптимизации, в основном разработанные и протестированные на Intel x64 или IBM Power, будут выполняться неоптимально или даже ухудшать код; то же тем более касается и машинно-зависимых оптимизаций. Имеющиеся трансформации (как правило, 100-200 оптимизаций) могут взаимодействовать ранее не учтенным способом, их настройка по умолчанию должна быть переделана.

В ходе работ сотрудников ИСП РАН над оптимизацией компиляторов GCC и LLVM для платформ Intel x86, ARM, Intel Itanium на примере конкретных наборов приложений выработался следующий подход:

1. Адаптация приложения для автоматического тестирования, или преобразование приложения в *бенчмарк*. Для этого требуется задать ряд наборов входных данных, обеспечить автоматический запуск приложения в пакетном режиме на заданных данных, сверку выходных данных с эталонными. Для известных пакетов тестов типа SPEC CPU 2006 эта работа уже выполнена авторами тестов.

2. Запустить приложение на заданных наборах тестов, получить профиль исполнения. Проанализировать ассемблерный код приложения в горячих местах профиля, выделить оптимизации, которые могли бы улучшить этот код.

3. Если выделенные оптимизации уже существуют в компиляторе, то нужно установить, почему они не отрабатывают в заданном случае с нужным качеством. Если нет, то нужные оптимизации требуется разработать и реализовать.

4. Обеспечить включение разработанных оптимизаций или исправлений в основную ветвь компилятора. Без этого шага самостоятельная поддержка изменений с учетом скорости разработки открытых компиляторов обходится слишком дорого.

В данном подходе исключительно ресурсоемким является второй шаг, требующий ручного анализа большого количества ассемблерного кода. Предложено несколько вариантов уменьшения количества необходимых для этого ресурсов:

- Исходя из общих знаний об используемом компиляторе и целевой

архитектуры, можно сразу сделать вывод о необходимости реализации новой для компилятора оптимизации. Например, работы по реализации нового планировщика команд GCC, учитывающего особенности архитектур с явным параллелизмом типа Itanium, были проведены в ИСП РАН в 2006-2008 годах на основе такого анализа.

- Можно провести автоматическую настройку компилятора на заданное приложение с помощью разработанного в ИСП РАН инструмента TACT (см. раздел 3). Инструмент может как найти опции компилятора, предоставляющие ускорение программы на 10-20% по сравнению со стандартным набором опций, так и подсказать, какие опции было бы оптимальнее исключить из стандартного набора. Оптимизации, соответствующие исключенным опциям, ухудшают код для заданного приложения и могут быть сразу подвергнуты анализу на третьем шаге.

- Можно организовать сравнение поведения компилятора на программе с другим известным компилятором и легче обнаружить недостающие оптимизации на примере более быстрого кода, полученного этим другим компилятором. Так, в оптимизациях LLVM (раздел 4) некоторые улучшения автоматической векторизации были получены как случаи, обрабатываемые компилятором GCC, но не обрабатываемые LLVM.

- Наконец, можно разработать и задействовать инструменты, позволяющие в автоматическом или полуавтоматическом режиме проверить, корректно ли использует приложение предлагаемые компилятором оптимизации. Например, для получения максимальной выгоды от межпроцедурных оптимизаций во время компоновки количество внешних функций приложения должно быть сведено к минимуму. В ИСП РАН разработан несложный инструмент, позволяющий проверить соответствие экспортируемых через заголовочные файлы интерфейсов реально видимым в двоичном файле библиотек функциям и исправить возможные неточности.

Настоящая статья посвящена примерам работ, выполненных для компиляторов GCC и LLVM в ИСП РАН согласно описанному подходу. В разделе 2 предлагается описание ряда выполненных исправлений компилятора GCC для платформы ARM. Раздел 3 кратко описывает инструмент автоматической настройки компилятора TACT и полученные с помощью него результаты. Раздел 4 содержит примеры оптимизационных работ в компиляторе LLVM, а раздел 5 завершает статью.

2. Оптимизации компилятора GCC

В данном разделе рассматриваются некоторые из разработанных нами оптимизаций для компилятора GCC: программная конвейеризация циклов [7] и ряд оптимизаций кодогенерации для архитектуры ARM. Первая из них является платформонезависимой, и разработанные улучшения применимы сразу для нескольких целевых архитектур. Вторая группа оптимизаций нацелена на архитектуру ARM, и их разработка, в основном, стала результатом анализа производительности приложений с помощью инструмента TACT, рассматриваемого в разделе 3.

2.1. Программная конвейеризация циклов

Программная конвейеризация циклов – оптимизация, которая преобразует тело цикла так, что на выполнение могут выдаваться команды из разных итераций исходного цикла, которые выполняются параллельно, подобно конвейеру.

В компиляторе GCC реализован один из вариантов программной конвейеризации циклов – *поворотное модульное планирование* [4]. Однако, реализация данного алгоритма в GCC обладает рядом ограничений, которые не позволяли использовать его на платформе ARM.

Во-первых, она поддерживает только циклы вида *do-loop*, а именно цикл с уменьшающимся на единицу счетчиком, условие выхода из которого выглядит как «счетчик цикла равен нулю (или единице)». Во-вторых, в наборе инструкций целевой архитектуры должна присутствовать команда, целиком реализующая семантику управляющей конструкции *do-loop* цикла (например, команда `loop` в процессорах *x86*). В-третьих, счетчик цикла не должен использоваться в других инструкциях, вне управляющей части цикла, а сам цикл должен состоять из одного базового блока и не содержать вызовов других функций и процедур, а также других инструкций с побочными эффектами.

Нами были исследованы возможности отказа от части этих ограничений и реализована поддержка имеющимся алгоритмом циклов более общего вида. Так, теперь поддерживаются циклы, в которых сравнение счетчика выполняется с произвольной константой, инвариантной относительно цикла, а также допускается чтение счетчика цикла командами внутри цикла. Эти изменения потребовали, в частности, улучшения алгоритма создания пролога и эпилога. Кроме того, был существенно усовершенствован механизм генерации проверки условий для выполнения оптимизированной версии цикла. Также было реализовано внесение в граф зависимостей по данным необходимых изменений для генерации компилятором корректного кода.

В результате было получено увеличение производительности на 1-4% на некоторых приложениях на процессоре ARM Cortex-A8. Тестовый набор для *sqlite* показал ускорение на 3%. Результаты для тестового набора *expedite* для

библиотеки растеризации *libevas* в среднем не изменились, при этом отдельные тесты показывали результаты от замедления на 3% до ускорения на 4%. Дополнительно было проведено тестирование умножения матриц из целых и вещественных чисел с помощью различных алгоритмов. Для матриц из целых чисел среднее ускорение составило 3%, а для случая чисел с плавающей точкой производительность увеличилась на 1%. На промышленном наборе тестов SPEC CPU2000 [5], в среднем производительность изменилась незначительно. Наибольшее ускорение получено на тесте *300.twolf*, оно составило 1.7%. На тестах *252.eon* и *178.galgel* производительность снизилась на 2%.

2.2. Улучшения поддержки набора инструкций NEON и Thumb-2

В результате анализа кода тестовых приложений, которые замедлялись при включении автоматической векторизации GCC для архитектуры ARM, было найдено несколько недостатков в поддержке команд векторного сопроцессора NEON.

Во-первых, в GCC не поддерживались векторные команды сдвига с непосредственным значением величины сдвига, несмотря на наличие таких команд в спецификации NEON. В результате цикл векторизовался, но компилятор был вынужден дополнительно выдавать команды для пересылки четырех одинаковых значений из регистров общего назначения в векторный регистр NEON, что приводило к замедлению даже по сравнению с невекторизованной версией. Кроме того, в компиляторе не была реализована поддержка команды vabd (векторный модуль разности), вместо которой последовательно выдавалось две команды для вычисления векторной разности и модуля.

Оба недостатка были устранены с помощью добавления в архитектурно-зависимую часть GCC для ARM шаблонов для поддержки соответствующих векторных операций. В результате этих изменений набор тестов *expedite* для библиотеки *libevas* ускорился в среднем на 0.5% (до 3% на конкретных тестах), а ускорение на приложении *x264* составило 2.5%.

Другой особенностью архитектуры ARM, в реализации которой в GCC были найдены недостатки, является условное выполнение в наборе команд Thumb-2. В отличие от 32-битного набора команд ARM, в Thumb-2 в целях экономии места предикаты не хранятся непосредственно в кодировке каждой команды, поэтому условным командам должна предшествовать специальная команда IT (*If-Then*), указывающая на начало IT-блока, определяющего условие исполнения до 4-х последующих команд. Ниже приводятся основные найденные недостатки в поддержке таких команд в GCC:

1. Оптимизация выбора кодировки команды происходит раньше преобразования в условную форму. В Thumb-2 команда, в зависимости от своих аргументов, может быть закодирована с

помощью 16 или 32 бит, причем большинство команд может быть представлено в 16-битной форме, только если они находятся внутри IT-блока, либо имеют суффикс *S*, т.е. записывают в регистр флагов по результатам своей работы. Чтобы представить команду в 16-битной форме, компилятор добавляет такой суффикс всегда, когда это допускается семантикой программы, причем соответствующая оптимизация происходит раньше, чем преобразование команд в условную форму, которое уже не может применяться к командам, записывающим результат в регистр флагов. Изменение порядка этих оптимизаций увеличивает количество IT-блоков на 2% в объектном коде для набора тестов SPEC 2000 INT.

2. Ограничение количества команд в условном блоке. Основная идея состоит в том, что на Thumb-2 преобразование в условную форму добавляет одну IT-команду на четыре преобразованных, при этом такое преобразование позволяет удалить одну или две команды перехода, в зависимости от наличия ветки *else* в условии. Таким образом, чтобы избежать увеличения размера кода, необходимо ограничить длину преобразуемого условного блока, соответственно, 4 или 8 командами.
3. Поддержка IT-блоков в планировщике команд. Так как на этапе планирования команд IT-блоков еще не создано, а условные предикаты во внутреннем представлении связаны с отдельными командами, планировщик может «перемешать» команды с разными предикатами, так что для каждой из них понадобится свой собственный IT-блок, что приводит к увеличению размера программы. Поэтому планировщик GCC был изменен так, чтобы отдавать приоритет тем командам, которые могут попасть в один IT-блок с последней выданной командой (т.е. имеющим точно такой же предикат, как последняя команда, или противоположный ей, например, eq/ne).

Рассмотренные улучшения обработки условных блоков существенно улучшили код для некоторых небольших тестов, однако в целом размер полученного кода для SPEC 2000 сократился незначительно.

3. Инструмент TACT автоматической настройки компилятора

Для достижения хорошей производительности сгенерированного кода в компиляторе обычно недостаточно реализовать платформозависимые оптимизации, но также необходима настройка уже существующих платформонезависимых оптимизаций. Такие оптимизации обычно имеют настройки, контролирующие применение тех или иных эвристик, параметры модели оптимизации, настраиваемые пороговые величины и т.п.

Стандартный подход к настройке компилятора под заданную архитектуру обычно состоит в следующем. Сначала определяется набор тестовых приложений (например, может быть использован промышленный тестовый набор SPEC CPU 2000). Для выбранных приложений собирается профиль выполнения, определяются «горячие» места, и вручную выполняется анализ полученного ассемблерного кода. Это довольно трудоемкий процесс, т.к. необходимо не только выделить те ассемблерные команды, которые больше всего влияют на производительность, но и определить, какие именно компиляторные оптимизации или их параметры привели к появлению этих команд. Для того чтобы упростить эту задачу, нами был разработан инструмент TACT [6] (Tool for Automatic Compiler Tuning – Инструмент для автоматической настройки компилятора), позволяющий автоматически находить оптимальный набор параметров компилятора для заданного приложения, а также определять те оптимизации (или их параметры), включенные по умолчанию, которые на данной архитектуре приводят к появлению неоптимального кода. Этот инструмент может быть использован как для непосредственного улучшения производительности приложений, так и для облегчения последующего ручного анализа кода и улучшения оптимизаций в компиляторе. Примеры таких улучшений в компиляторе GCC, сделанных по итогам анализа приложений с помощью TACT, были рассмотрены в разделе 2, а также в работе [8]. Далее в этом разделе мы рассмотрим основные особенности разработанного нами инструмента.

3.1. Генетический алгоритм

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

3.2. Общая схема работы TACT

Система для автоматической настройки состоит из одного управляющего *x86* компьютера, используемого также для кросс-компиляции настраиваемых приложений и нескольких тестовых плат (например, архитектуры ARM), доступных с управляющего компьютера по протоколу SSH и связанных таким образом, что все устройства имеют один общий каталог через сетевую

файловую систему NFS. Общая схема устройства инструмента TACT показана на Рис 1.

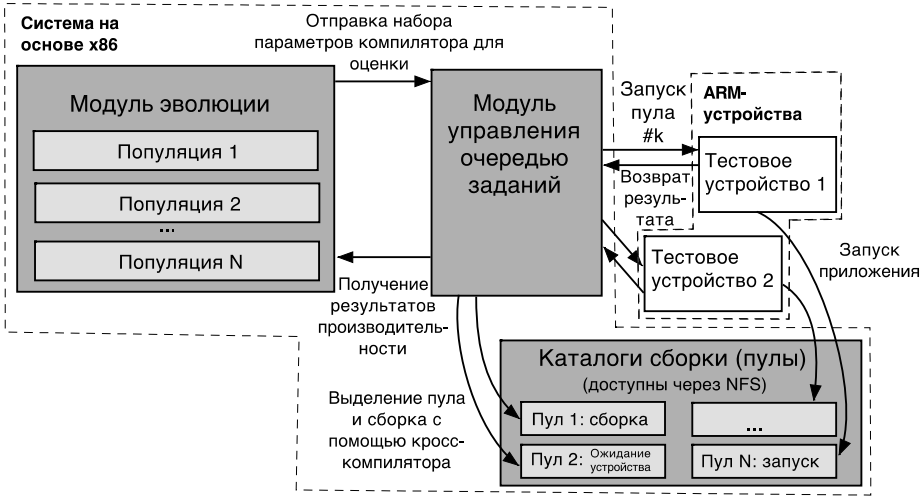


Рис 1. Компоненты TACT и общая схема работы

Модуль эволюции генерирует с помощью генетического алгоритма различные наборы параметров компилятора и отправляет их на выполнение на тестовые устройства, используя *модуль управления очередью заданий*. Данный модуль отвечает за сборку приложения с заданными параметрами компилятора, выбор свободного тестового устройства и запуск на нем скомпилированного приложения. Полученная на тестовых устройствах оценка производительности для заданного набора затем используется модулем эволюции для улучшения параметров компиляции в следующем поколении.

3.3. Определение наиболее значимых параметров компиляции

Набор опций командной строки компилятора, полученных в результате работы генетического алгоритма, может быть избыточным: некоторые опции могут в контексте других опций не изменять поведение компилятора, но они будут по-прежнему присутствовать в результирующей строке. После завершения работы генетического алгоритма, TACT пытается уменьшить количество опций компилятора в результирующих наборах опций за счет удаления тех из них, присутствие которых не влияет на итоговые бинарные файлы. Это происходит путем последовательного удаления опций по одной, и может занять достаточно длительное время. Данный процесс позволяет сократить количество опций в наборах в 3-6 раз.

После описанной процедуры итоговые наборы содержат только опции компиляции, влияющие на получившийся объектный код, но тем не менее большинство из них не оказывает значительного влияния на производительность. Поэтому на следующем этапе TACT уже пытается упорядочить опции в каждом наборе по их вкладу в производительность. Для этого на каждом шаге итерации из набора удаляется одна опция, такая, что ее удаление ведет к наименьшей потере производительности, а сама найденная на текущем шаге опция записывается в таблицу наиболее значимых, начиная снизу вверх. Процесс повторяется, пока в наборе не остается только одна опция базового уровня оптимизации (например, `-O2`), которая записывается в самую верхнюю строку таблицы. Таким образом, построенная таблица содержит все опции исходного набора в порядке убывания их значимости, причем вклад некоторых опций может оказаться отрицательным из-за погрешности измерения производительности на тестовых платах и неточностей, связанных с применением генетического алгоритма.

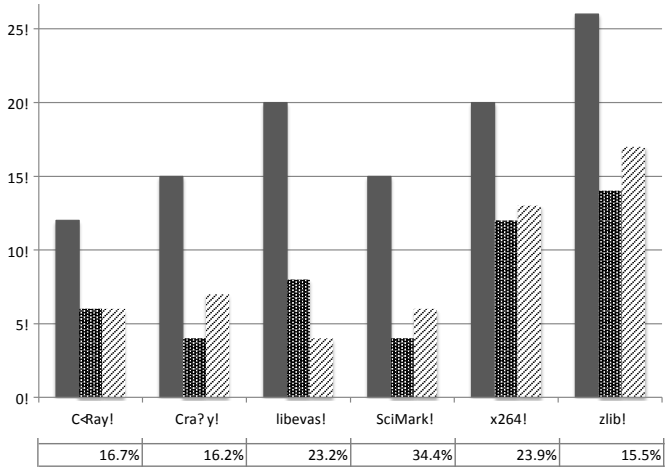


Рис2. Количество опций, существенных с точки зрения производительности приложения и величина прироста производительности

На Рис.2 для выбранных тестовых приложений левый столбец показывает количество опций из 200 исходных, положительно влияющих на производительность, и полученных описанных выше способом. Средний столбец показывает количество лучших опций, которые сохраняют 80% прироста производительности, полученного на лучшем наборе (итоговое улучшение производительности указано под названием приложения). Правый столбец показывает количество оптимизаций в полученном наборе, которые должны быть отключены по сравнению с настройками компилятора по

умолчанию в базовом уровне (`-O2`), а также количество параметров, значение которых отличается от базовых.

С точки зрения разработчика компилятора, именно опции из последней категории представляют наибольший интерес, т.к. могут указывать на недостатки в оптимизациях, используемых по умолчанию, и их исправление может дать прирост производительности сразу для широкого класса приложений. При этом число таких опций сравнительно невелико (5-15 шт. для данных приложений), что позволяет продолжить анализ производительности для найденных оптимизаций в ручном режиме.

4. Оптимизации компилятора LLVM

В ходе оптимизации компилятора LLVM были сделаны некоторые исправления кодогенератора LLVM для платформы ARM, улучшена оптимизация преобразования ветвлений (if-conversion), выполнен ряд оптимизаций с учетом профиля программы [3]. В настоящей работе далее приведены примеры наиболее существенных оптимизаций, сделанных за последние два года.

4.1. Выделение шаблонов перемежающегося доступа к данным

Рассмотрим пример:

```
int a[N/2], b[N+1], c[N+1];
for (i = 0; i < N/2; i++){
  a[i] = b[2i+1] * c[2i+1] - b[2i] * c[2i];
}
```

Листинг 4.1. Пример кода с перемежающимся доступом к данным

Заметим, что использование инструкции загрузки с устранением перемежения vld2.32 {d0, d1} [Rn] (где Rn указывает на ячейку памяти, хранящую элемент b[2i]) позволит загрузить на регистр d0 элементы массива b[2i] и b[2i+2], а на регистр d1 элементы массива b[2i+1] и b[2i+3].

```
int a[N/2], b[N+1], c[N+1];
for (i = 0; i < N/2; i+=2){
  even_b, odd_b = vector_load(&b[2i]);  //
even_b ={b[2i], b[2i+2]}
  even_c, odd_c = vector_load(&c[2i]);  //
odd_b = {b[2i+1], b[2i+3]}
  {a[i], a[i+1] } = odd_b * odd_c –
even_b * even_c;
}
```

Листинг 4.2. Векторизованный код с перемежающимся доступом к данным

Для нахождения шаблонов перемежающегося доступа внутри тела цикла в пределах одной итерации находились все инструкции загрузки данных из памяти. Среди них выделялись те, которые загружают данные из одного источника. С помощью анализа возможных значений скалярных выражений (Scalar Evolution) проводилось сравнение с шаблонами перемежающегося доступа. При совпадении с шаблоном проводилось преобразование промежуточного представления. Скалярные инструкции загрузок удалялись, вместо них вставлялась векторная инструкция загрузки с устранением перемежения.

В случае возможности векторизации цикла проводится сравнение стоимости выполнения векторизованной и скалярной версии цикла. Если стоимость векторной версии меньше, чем скалярной, то принимается решение о векторизации данного цикла.

4.2. Использование векторных загрузок с автоматическим увеличением адреса загрузки

Команды векторной загрузки ARM могут автоматически увеличивать значение базового регистра (по адресу из которого происходит загрузка) на величину загружаемых данных. Это позволяет объединять вычисление адреса и инструкцию векторной загрузки в одну операцию векторной загрузки с автоинкрементом.

Таблица 4.1. Использование команды автоинкрементной загрузки

загрузка и отдельное вычисление адреса	автоинкрементная загрузка
vld1.32 {d16, d17}, [r0] add r0, r0, #16	vld1.32 {d16, d17}, [r0]!

4.3. Использование выравненных векторных загрузок

Время перемещения данных из памяти в регистры инструкциями структурированной векторной загрузки из расширения Neon зависит от их выравнивания. Согласно документации количество циклов, затрачиваемых процессором на выполнение инструкции, больше в случае доступа к невыравненным данным. Например, инструкция vld1.64 при обращении к памяти с 64 битным выравниванием (3 последних бита в адресе равны нулю) затрачивает меньше процессорных циклов, чем при обращении к не выровненным данным.

Данную оптимизацию необходимо проводить на машинно-зависимом уровне. Для этого нужно для каждой встретившейся инструкции векторной загрузки провести анализ расположения данных в памяти

Применение описанных оптимизаций позволило ускорить используемый набор тестов (набор тестов векторизации компилятора GCC) в среднем на ~7%.

4.4. Генерация команд предвыборки при обработке массивов в цикле

Использование команд предвыборки при обработке массивов в цикле повышает эффективность использования кэша процессора во время последовательной загрузки данных.

Процессоры архитектуры ARM серии Cortex-A9 имеют встроенный автоматический механизм предвыборки данных, который загружает данные в кэш, учитывая промахи кэша, массив загружается в кэш после нескольких итераций и промахов кэша. Такое поведение неоптимально и может быть исправлено с помощью команды предвыборки “PLD”, которая указывает процессору, что вскоре будут использованы данные, на которые указывает команда, так что их желательно загрузить в кэш, если их там еще нет.

Предвыборка данных должна осуществляться заранее, до их непосредственного использования. На процессоре ARM Cortex-A9 требуется выполнение около 200 тактов после команды предвыборки, чтобы данные были загружены в кэш [2]. Тогда момент вставки инструкции упреждающей загрузки можно рассчитать, как отношение задержки загрузки данных в кэш после выполнения команды предвыборки к количеству команд в цикле.

Для того, чтобы команды предвыборки не выполнялись слишком часто и не указывали на участки памяти, которые уже загружены в кэш, используется развертывание циклов. Цикл развертывается столько раз, чтобы загружаемые данные за один проход развернутого цикла полностью заполняли одну строку кэша. Например, если размер строки кэша 32 байта (как на процессоре ARM Cortex-A9), а размер загружаемых каждую итерацию данных равен 4 байта, то цикл стоит развернуть 8 раз (32 / 4), и вставить команду предвыборки лишь в первую итерацию.

Тестирование на наборе тестов SPEC CPU 2000 показало, что прирост производительности составляет ~0.9%. На тестах SQLite, Expedite, Cray и Coremark прирост производительности составил 0,5 до 5 %, средний прирост составляет ~2.5%

4.5. Модификация алгоритма распределения регистров

Архитектура ARM поддерживает команды, осуществляющие множественную загрузку/сохранение по последовательным адресам - LDM/STM. Инфраструктура LLVM учитывает эту особенность архитектуры в оптимизирующем проходе "ARM load / store optimization pass", который осуществляет свертку последовательных операций (LDR/STR) в одну или несколько команд множественной загрузки/сохранения. Копирование структур в LLVM осуществляется посредством вызова функции `memcpy`, вызов которой в целях оптимизации заменяется серией команд, осуществляющих загрузку/сохранение. Но алгоритм распределения регистров не учитывает возможность такой оптимизации, поэтому регистры распределяются не в порядке строгого возрастания номеров. Алгоритм распределения регистров был модифицирован таким образом, чтобы обеспечить выбор следующего свободного регистра с учетом его номера, обеспечивая последовательное возрастание номера используемого регистра, что повысило качество работы оптимизации "ARM load / store optimization pass" и привело к росту быстродействия генерируемого кода.

5. Заключение

В статье описаны способы улучшения производительности программ при статической компиляции на примере работ, выполненных в ИСП РАН для компиляторов GCC и LLVM на платформе ARM. Получены результаты ускорения набора тестов в среднем на 1-5%, а конкретного приложения (при автоматической настройке) – на 10-20%. Работы по улучшению статических компиляторов будут продолжаться в направлении разработки новых инструментов, позволяющих уменьшить затраты на ручной анализ ассемблерного кода.

Список литературы

[1] The LLVM Compiler Infrastructure. http://LLVM.org/

[2] Справочное руководство по процессорной архитектуре ARM., http://infocenter.arm.com

[3] Ш.Ф. Курмангалеев. Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования. Труды ИСП РАН, том 24, стр. 127-144, 2013 г. DOI: 10.15514/ISPRAS-2013-24-7.

[4] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. “Lifetime-sensitive modulo scheduling in a production environment”. Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001.

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

[6] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, JH. Lee. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. Procedia Computer Science Volume 18, pp.1312-1321. 2013.

[7] Р. Жуйков, Д. Мельник, Р. Бучацкий. Программная конвейеризация циклов на платформе ARM. Труды ИСП РАН. 2012. №22. С.33-48. DOI: 10.15514/ISPRAS-2012-22-3.
Р. Жуйков, Д. Плотников, М. Варданян. Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM. Труды ИСП РАН. 2012. №22. С.49-66. DOI: 10.15514/ISPRAS-2012-22-4.

Optimizing programs for given hardware architectures with static compilation: methods and tools

Dmitry Melnik dm@ispras.ru
ISP RAS, Moscow, Russia
Shamil Kurmangaleev <kursh@ispras.ru>
ISP RAS, Moscow, Russia
Arutyun Avetisyan <arut@ispras.ru>
ISP RAS, Moscow, Russia
Andrey Belevantsev <abel@ispras.ru>
ISP RAS, Moscow, Russia
Dmitry Plotnikov <dplotnikov@ispras.ru>
ISP RAS, Moscow, Russia

Abstract. The paper describes the workflow for optimizing programs for performance targeting the fixed hardware architecture with static compilation using GCC and LLVM compilers as examples. The workflow has gradually grown within ISP RAS Compiler Technology Team when working on GCC and LLVM compiler optimization. We start with preparing a benchmark using the given application as a source, and then proceed in parallel with manual analysis of generated assembly code and automatic compiler tuning tool. In general, a compiler optimization improvement produced by the manual analysis gives 1-5% speedup, while the automatic tuning results may give up to 10-20% speedup. However, the manual analysis results are usually valid for the broad class of applications and are contributed to the open source compilers, while the automatic tuning results make sense only for the given application.

We present some of the optimizations performed, e.g. improved NEON and Thumb-2 support for GCC, vectorization improvements for LLVM, register allocation improvements for LLVM, and the corresponding evaluation results. We also describe TACT, a tool for automatic compiler tuning for the given application mentioned above, and its example use cases both for an application developer and a compiler engineer. We give the sample of TACT optimization results.

Keywords: Program optimization, GCC, LLVM, automatic compiler tuning.

References

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/

[2]. ARM Architecture Processor Manual, http://infocenter.arm.com

[3]. Kurmangaleev S.F. Metody optimizatsii Ci/Ci++ - prilozhenij rasprostranyaemykh v bitkode LLVM s uchetom spetsifiki oborudovaniya. [Machine-specific optimization methods for C/C++ applications that are distributed in the LLVM intermediate representation format]. Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol.24, pp. 127-144. DOI: 10.15514/ISPRAS-2013-24-7. (in Russian)

[4]. J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. "Lifetime-sensitive modulo scheduling in a production environment". Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001.

[5]. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/

[6]. D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, J.H. Lee. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. Procedia of Computer Science Volume 18, pp.1312-1321. 2013.

[7]. Roman Zhuykov, Dmitry Melnik, Ruben Buchatskiy. Programmnaya konvejerizatsiya tsiklov na platforme ARM [Loop software pipelining on ARM platform]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol.22, pp. 33-48. DOI: 10.15514/ISPRAS-2012-22-3. (in Russian)

[8]. Roman Zhuykov, Dmitry Plotnikov, Mamikon Vardanyan. Avtomaticheskaya nastrojka optimizatsionnykh preobrazovanij kompilyatora GCC dlya platformy ARM [Automatic tuning of GCC optimization passes for ARM platform]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol.22, pp. 49-66. DOI: 10.15514/ISPRAS-2012-22-4. (in Russian)

Инструменты анализа и разработки эффективного кода для параллельных архитектур

Александр Монаков amonakov@ispras.ru, Евгений Велесевич evel@ispras.ru, Владимир Платонов soh@ispras.ru, Арутюн Аветисян <arut@ispras.ru>

Аннотация. В работе предлагаются методы поддержки разработки эффективных программ для современных параллельных архитектур, включая гибридные. Описываются специализированные методы профилирования, предназначенные для программиста, занимающегося распараллеливанием существующего кода, либо для поиска неэффективного использования кеша в многопоточных программах. Рассматривается задача автоматической генерации параллельного кода для гибридных архитектур. В задачах, где для повышения производительности на гибридных архитектурах необходима существенная переработка структур данных или алгоритмов, может использоваться автотюнинг для специализации под конкретную задачу и аппаратуру во время выполнения. Показана оптимизация умножения разреженных матриц на GPU и ее применение для ускорения расчётов в пакете OpenFOAM.

Ключевые слова: оптимизация программ, профилирование, OpenCL, CUDA, разреженные матрицы, OpenFOAM

1. Введение

Эволюция архитектуры графических акселераторов привела к тому, что в настоящее время они являются массивно-параллельными вычислительными устройствами, применение которых уже не ограничивается задачами графического рендеринга. За счет специализированной архитектуры они могут достигать более высокой энергоэффективности и производительности по сравнению с современными многоядерными процессорами. Для облегчения разработки неграфических вычислений на акселераторах были созданы модели программирования CUDA [1] и OpenCL [2].

В модели программирования CUDA явно отображены низкоуровневые принципы работы акселератора. Это дает возможность детально исследовать производительность и выполнять оптимизацию кода, что было важно для обеспечения успешного развития модели. С другой стороны, использование специализированной низкоуровневой модели увеличивает сложность разработки.

Программные решения, использующие акселераторы, не всегда создаются с нуля: напротив, возникает необходимость доработки существующих программ для использования графических акселераторов с целью повышения эффективности расчетов. Для крупных программных проектов перенос всех алгоритмов на акселератор требует неоправданных трудозатрат. В этих случаях требуется анализ с целью выявления участков кода, отвечающих наибольшим затратам времени выполнения, и последующий перенос их на акселератор с сохранением внешних интерфейсов.

Несмотря на то, что OpenCL и CUDA обеспечивают переносимость между различными поколениями акселераторов (в случае OpenCL – также и переносимость между графическими акселераторами и процессорами различных производителей), для достижения высокой производительности требуется разработка отдельных вариантов кода для различных устройств. Даже при использовании CUDA изменение баланса между количеством регистров, активных нитей выполнения и объема разделяемой памяти при выходе новых поколений акселераторов требует пересмотра оптимизаций кода, что означает усложнение долговременной поддержки кода. Частично эта проблема может быть решена за счет разработки механизма автоматической настройки параметров кода, выполняющегося на акселераторе. Автотюнинг позволяет проверить только изначально заложенные в его схему варианты кода. Это компромиссное решение, позволяющее улучшить эффективность без затрат на ручной анализ и оптимизацию.

Цель данной работы — разработка инструментов и библиотек, поддерживающих продуктивность разработки программного кода и обеспечение высокой производительности для широкого спектра акселераторов. Для достижения этой цели предлагаются следующие подходы:

1) Профилирование последовательного кода для поиска тех участков, перенос которых на параллельную архитектуру актуален в первую очередь.
2) Автоматическая кодогенерация для параллельной гибридной архитектуры в процессе компиляции.
3) Автоматическая адаптация библиотечных функций.

Оригинальный код:
`for (init; test; step)`
`  stmt;`
 	
Преобразованный код:
`{`
`  __trace_start(loop_id);`
`  for (init;`
`__trace_test(test); step)`
`    stmt;`
`}`
	`@ match_for @`
	`expression init, test,`
	`step;`
	`statement stmt;`
	`position testp;`
	`@@`
	`- for`
	`+ {`
	`__trace_start(loop_id);`
	`for`
	`( init;`
	`- test@testp`
	`+ __trace_test(test)`
	`; step) stmt`
	`+ }`

Рис. 1. Общий вид инструментированного цикла for и соответствующий Coccinelle-скрипт

Статья имеет следующую структуру. В разд. 2 предлагаются специализированные методы профилирования, предназначенные для программиста, занимающегося распараллеливанием существующего кода, либо для поиска неэффективного использования кеша в многопоточных программах. В разд. 3 рассматривается задача автоматической генерации параллельного кода для гибридных архитектур. В задачах, когда для повышения производительности на гибридных архитектурах необходима существенная переработка структур данных или алгоритмов, может использоваться автотюнинг для специализации под конкретную задачу и аппаратуру во время выполнения. Это демонстрируется в разд. 4, где показана оптимизация умножения разреженных матриц на GPU и применение для ускорения расчётов в пакете OpenFOAM.

2. Инструменты профилирования кода

В случаях, когда перед программистом стоит задача переноса большой базы кода на параллельную архитектуру с целью повышения производительности,

необходимо выделить участки кода, анализ и перенос которых должен производиться в первую очередь. Как правило, распараллеливание кода выполняется в первую очередь на уровне циклов. Соответственно, необходимо собрать информацию о том, каким циклам соответствуют наибольшие затраты времени выполнения (с учетом времени, затраченного во всех вызывавшихся подпрограммах).

Когда для переносимых программ доступны входные данные, поведение на которых отражает реальные сценарии использования, наиболее удобным подходом к сбору характеристик времени выполнения является динамический анализ (профилирование). Кроме того, инструментацию и профилирование можно использовать для поиска типичных ошибок, приводящих к ухудшению производительности параллельных программ, как показано в разделе 2.3.

```
cairo-tor-scan-converter.c:1885:
 glitter_scan_converter_render:
  31056899836    nsec:    11365462    iters:    2733
nsec/iter:
  181819 loops: 63 iters/loop
cairo-image-surface.c:3025:
 _composite_boxes:
  18235085135    nsec:    101067    iters:    180426
nsec/iter:
  101066 loops: 1 iters/loop
cairo-image-surface.c:3031:
 _composite_boxes:
  18204913778 nsec: 187791 iters: 96942 nsec/iter:
  101067 loops: 2 iters/loop
cairo-tor-scan-converter.c:1932:
 glitter_scan_converter_render:
  17959504548 nsec: 76624515 iters: 234 nsec/iter:
  5108301 loops: 15 iters/loop
```

Рис. 2. Пример результатов профилирования циклов

2.1. Профилирование циклов

В настоящее время для профилирования программ используются методы, основанные на периодической записи указателя выполняемой инструкции (и, возможно, стека вызовов). Собираемой таким образом информации не достаточно для программиста, занимающегося поиском параллельных циклов, так как она не привязана к иерархии циклов. Таким образом, необходимо дополнительно собирать динамические характеристики циклов:

- Количество входов в цикл
- Количество итераций
- Время, проведенное в цикле

Запись о времени, затраченном в цикле, позволяет легко идентифицировать важные для распараллеливания циклы. Без этого было бы необходимо выполнять нетривиальную пост-обработку профилировочной информации. Запись количества входов и итераций позволяет программисту исключить из рассмотрения циклы, которые не итерируются вообще, либо содержат слишком мало итераций чтобы распараллеливание имело смысл.

Выполнять такое вспомогательное профилирование можно за счет инструментации кода. Возможно два подхода к организации инструментирования. В первом подходе предлагается для каждого обнаруженного цикла в предзаголовок добавлять код, увеличивающий счетчик входов в цикл, а в заголовочный блок встраивать код, увеличивающий счетчик итераций и добавляющий время последней итерации к счетчику общего времени, затраченного в цикле. Этот подход проще в реализации, но требует выполнения кода инструментации на каждой итерации цикла.

Эти накладные расходы можно сократить, если преобразовать граф потока управления цикла в форму с одним входом и одним выходом. Это позволит перенести большую часть инструментирующего кода (замер времени и запись результатов) из тела цикла к единственному выходу.

Для языков C и C++ инструментирование можно выполнять как source-to-source преобразование кода с помощью Coccinelle [3] (рис. 1).

2.2. Оценка расстояния повторного использования

В некоторых случаях программиста интересует расстояние переиспользования (reuse distance) для некоторых массивов, расположенных в памяти. Пусть функция с прототипом void compute(char out[], size_t out_n, const char in[], size_t in_n) генерирует out_n элементов массива out[], не содержит доступов к глобальным переменным и не содержит побочных эффектов; таким образом, результат ее работы зависит только от содержимого массивов out и in на момент вызова. Это означает, что вычисления можно производить в произвольный момент между точкой вызова и следующим после нее обращениям к массивам out (на чтение или запись) или in (на запись). В частности, вычисления можно вынести в отдельную нить, которая может выполняться параллельно с основной программой, что приведет к ускорению на параллельной архитектуре, если compute выполняет достаточно интенсивные вычисления.

Кроме того, оценка расстояния повторного использования может использоваться для специализации кода в зависимости от уровня кеша, в который должны попасть вычисленные значения.

Для замеров времени до следующего доступа можно использовать механизм защиты виртуальной памяти. Вызов mprotect(void *addr, size_t len, int prot) позволяет изменить права доступа к региону памяти [addr, addr+len-1] в соответствии с маской prot. При следующем доступе программа получит сигнал SIGSEGV, обработчик которого имеет доступ к адресу, который программа пыталась прочитать или записать. Соответственно, обработчик может определить, какому из ранее защищенных регионов принадлежит этот адрес, записать время доступа, восстановить доступ на чтение и запись и продолжить выполнение программы.

2.3. Оценка частоты промахов кеша и поиск ситуаций ложного разделения

Кеширование часто используемых данных из оперативной памяти является одним из ключевых методов обеспечения высокой производительности современных процессоров. Кеши не является общими ресурсами: каждое процессорное ядро, как правило, имеет свой кеш первого уровня; в многопроцессорных системах каждый физический процессор имеет свою иерархию кешей. Кеш хранит множество выровненных участков оперативной памяти, называемых строками. Один и тот же участок оперативной памяти может одновременно находиться в кешах разных процессоров; когда один из процессоров модифицирует эти данные, необходимо обновить или удалить их из кеша другого процессора. Эта задача решается с использованием протокола согласования кешей (cache coherency protocol).

Поскольку модификация даже одного байта приводит к обновлению целой строки кеша, могут возникать ситуации, когда нити, параллельно выполняющиеся на разных ядрах, поочередно обновляют разные переменные, попадающие в одну строку кеша. В таких случаях обновление на одном процессоре будет приводить к вытеснению соответствующей строки из кеша другого процессора и сериализовывать выполнение нитей, как если бы они обновляли одну и ту же переменную. Такое поведение известно как ложное разделение (false sharing) и ухудшает производительность параллельных программ. Соответственно, желательно иметь инструменты для анализа эффективности использования кеша, в том числе, поиска ситуаций ложного разделения.

Для оценки количества промахов можно использовать инструментацию кода. Основная идея заключается в том, чтобы собрать прореженную трассу доступов к памяти, включая расстояния повторного использования, и затем оценить количество промахов, исходя из собранной трассы и некоторой теоретической модели кеша. Хагерстен [4] предлагает вероятностную модель для оценки эффективности использования кеша со случайным вытеснением на основе расстояния повторного использования строк кеша. Недостатком его подхода для поиска ситуаций ложного разделения в многопоточных программах является необходимость дополнительной синхронизации при

сборе трассы. Нашей целью будет обход этой необходимости, так как она может существенно замедлить выполнение программы.

Пусть *L* — количество строк в кеше; тогда вероятность того, что строка, находящаяся в кеше, уже не будет в нем после *n* промахов, равна:

$$f(n) = 1 - (1 - 1/L)^n$$

Пусть *r(i)* — вероятность промаха при *i*-ом обращении, а *A(i)* — значение расстояния повторного использования для *i*-го обращения. Тогда количество промахов кеша между *i*-м и *(i-A(i)-1)*-м обращениями можно оценить как:

$$\sum_{j=i-A(i)}^{i-1} r(j)$$

Тогда вероятность того, что на момент обращения *i* искомая строка не будет находиться в кеше:

$$r(i) = f\left(\sum_{j=i-A(i)}^{i-1} r(j)\right)$$

Предположив, что вероятность промаха постоянна, т.е. *r(i)* = *R* для всех *i* (на некотором участке трассы программы), Хагерстен получает основное уравнение своей модели:

$$R * N = N_{cold} + \sum_{i=1}^{N_{nocold}} f(A(i) * R)$$

В многопоточном случае модели Хагерстена запись в трассе помимо строки кеша и расстояния повторного использования содержит информацио о том, была ли запись в эту строку в другой нити приложения между первым и вторым обращением в первой нити. Такой подход к сбору информации о записях в другой нити требует синхронизации.

Но возможен и другой подход — вычисление вероятности того, что во время обращения, записанного в трассу, и следующего обращения к этой строке в этой же нити, была запись в ту же строку в другой нити (на основе трасс обращений к памяти с временными метками для каждой нити приложения). Эта вероятность и будем искомым процентом промахов когерентности.

Для решения поставленной задачи требуется вычислить априорную вероятность того, что если промах когерентности произошел на обращении,

записанном в трассе, то запись, благодаря которой произошел промах, была записана в трассу другой нити (таких записей может быть несколько).

Пусть N_A — количество записей в кеш-строку A в исследуемом участке трассы второй нити, N — количество всех записей в участке. P — вероятность выборки обращения для записи в трассу. T — продолжительность исследуемого участка. T_A — время между первым и вторым доступом к A в первой нити. Тогда количество записей в участке $N_w = N/P$, среднее время на между записями $T_w = T/N_w$, количество записей во второй нити между первым и вторым обращением в первой нити $N_{Ag} = T_A/T_w$.

Вероятность того, что между первым и вторым доступом будет k записей в ту же строку в другой нити составит:

$$P_{Ak} = C_k^{N_{Ag}} P_A^K (1 - P_A)^{N_{Ag}-k}, \text{где} P_A = N_A/N$$

Тогда вероятность, что в трассе будет обращение, благодаря которому произошел промах (хотя бы одно из них):

$$P_{Ad} = 1 - \sum_{k=1}^{N_{Ag}} P_{Ak} (1 - P)^k$$

А вероятность, что промах произошел, независимо от того, попал он в трассу или нет:

$$P_{Am} = 1 - (1 - P_A)^{N_{Ag}}$$

Однако, нам известна статистическая вероятность попадания промаха в трассу:

$$\bar{P}_{Ad} = \bar{N}_{Ad}/N, \text{где} \bar{N}_{Ad} — \text{количество замеченных промахов.}$$

Тогда условная вероятность попадания промаха в трассу при наличии такого промаха:

$$\frac{\bar{P}_{Ad}}{P} = \frac{P_{Ad}}{P_{Am}}$$

Из этого уравнения можно получить P — искомую вероятность промахов когерентности, или же процент обращений к памяти, повлекших промах когерентности.

Для достаточно точного определения P_{Ad} требуется намного более частая выборка обращений для записи в трассу, так как вероятность, что между выбранными с вероятностью P обращениями в другой нити была выбрана запись, влекущая промах когерентности тоже равна P, т.е. P_{Ad} имеет порядок P^2. Поэтому P следует выбирать как квадратный корень из вероятности в модели Хагерстена.

3. Автоматическая генерация параллельного OpenCL-кода во время компиляции

В некоторых случаях возможно получение параллельного кода автоматически во время компиляции. Несмотря на то, что в результате распараллеливания вручную опытный программист выдаст более эффективный код, автоматическая кодогенерация является способом повышения производительности существующего кода на новых архитектурах.

В рамках предыдущих работ [5], [6] был разработан метод автоматической генерации OpenCL-кода для параллельных гнезд циклов. Использование OpenCL в качестве платформы для выполнения параллельного кода обеспечивает переносимость результирующей программы между как многоядерными процессорами, так и гибридными архитектурами с акселераторами, поддерживающих OpenCL. Отметим, что при использовании OpenCL часть кода программы выделяется в виде отдельных функций – «ядер», которые подаются библиотеке времени выполнения в виде исходного кода. Таким образом, компиляция выделенного кода выполяется во время выполнения драйвером OpenCL. Программа, использующая OpenCL, должна настроить контекст выполнения и передать среде выполнения код ядер. Далее перед запуском выполнения ядер программа должна скопировать на акселератор массивы, к которым будет обращаться ядро (если выполнение происходит на гибридной системе и акселератор имеет доступ к отдельному пространству памяти) и указать значения аргументов ядра. Ядро может выполняться на акселераторе параллельно с основной программой, но копирование вычисленных результатов с акселератора в память программы будет приводить к синхронизации.

В качестве инфраструктуры для реализации анализа и трансформации циклов использовалась система GRAPHITE компилятора GCC. GRAPHITE обеспечивает возможность высокоуровневого анализа циклов в рамках полиэдральной модели. Результат работы GRAPHITE для гнезда циклов представляет собой структуру SCoP (Static Control Part), включающую абстрактное представление исходного графа потока управления в виде CLAST, а также пространство итераций и расписание выполнения. Кроме того, для массивов, к которым выполняются обращения, строится индексная функция, отображающая элемент в пространстве итераций и оператор программы в смещение в массиве. Для возможности построения этого представления требуется, чтобы все циклы имели константные шаги, а индексы в доступах к элементам массивов аффинно зависели от итераторов циклов.

После выполнения анализа и преобразования расписания циклов GRAPHITE выполняет восстановление промежуточного представления GIMPLE из CLAST. На этом шаге структуры передачи управления и индукционные переменные циклов генерируются заново, но помимо этого переписывания оригинального GIMPLE-кода не требуется. Для генерации OpenCL-кода

требуется определить гнезда циклов, для которых возможно параллельное выполнение, и выполнить восстановление из CLAST в гибридный GIMPLE и OpenCL код. В виде GIMPLE необходимо сгенерировать ряд обращений к среде выполнения OpenCL:

1) инициализацию контекста (если еще не производилась);
2) компиляцию кода ядра для SCoP (при первом выполнении SCoP);
3) выделение памяти и копирование массивов на акселератор;
4) установку аргументов ядра;
5) собственно запуск ядра на выполнение;
6) копирование результатов и освобождение памяти на акселераторе.

Таким образом, восстановленный GIMPLE-код не зависит от графа потока управления цикла. Все поведение исходного цикла восстанавливается из CLAST в код на OpenCL C, при этом требуется реализовать выдачу OpenCL-кода из внутреннего представления GIMPLE для базовых блоков цикла. Результирующий код ядра на OpenCL C попадает в скомпилированную программу как константная строка и подается драйверу OpenCL во время выполнения.

4. Автоматическая адаптация

Современные графические акселераторы обеспечивают высокую производительность и энергоэффективность за счет специализированной массивно-параллельной архитектуры, сочетающей параллелизм на уровне независимых вычислительных блоков (мультипроцессоров в терминах CUDA) и параллелизм на уровне ALU, синхронно выполняющих одну команду для группы нитей. Количество групп нитей, которые могут быть одновременно активны на мультипроцессоре, зависит от нескольких факторов:

- количество регистров, требующихся для каждой нити;
- объем общей памяти, требующихся для блоков нитей;
- количество нитей в блоке.

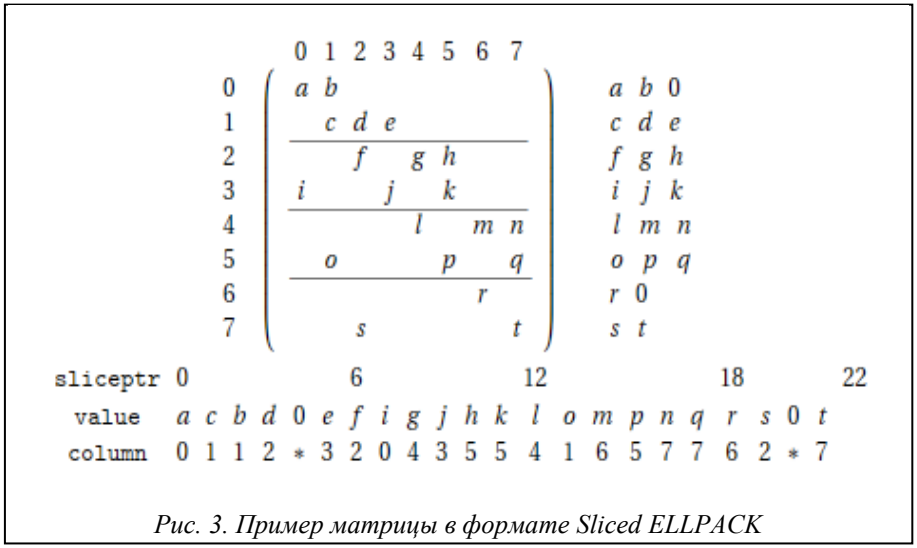


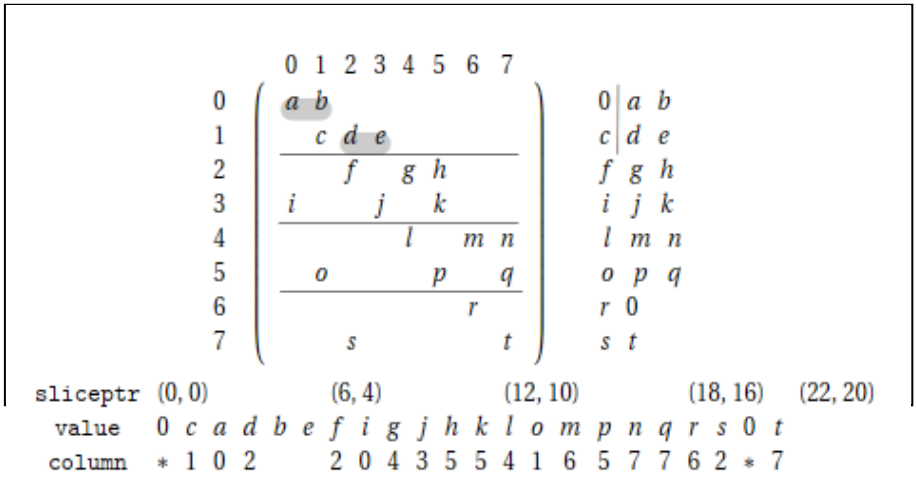
Рис. 3. Пример матрицы в формате Sliced ELLPACK

Количество активных нитей влияет на достигнутую производительность. Увеличение количества нитей позволяет скрывать задержки между зависимыми инструкциями за счет перекрытия выполнения различных нитей на одних и тех же ALU. С другой стороны, уменьшение количества нитей в сочетании с увеличением параллелизма на уровне команд в пределах одной нити (что повышает требования к количеству регистров на нить) в ряде случаев позволяет достичь более высокой производительности.

Как правило, аналитическое предсказание наилучшей конфигурации затруднено или невозможно. Кроме того, производительность может зависеть от неучтенных в аналитической модели факторов.

Для многократно использующихся функций имеет смысл выполнять автоматическую настройку реализации для конкретной архитектуры, на которой будет выполняться код. Автоматическая адаптация как средство повышения производительности используется в библиотеках ATLAS (работа с плотными матрицами) и FFTW (быстрое преобразование Фурье).

Обработка разреженных матриц (а именно, умножение матрицы на вектор) является важным вычислительным ядром в итерационных методах решения систем линейных уравнений, часто использующихся в задачах вычислительной гидродинамики. Особенностью ядра является высокая требовательность к пропускной способности памяти в сочетании с нерегулярными доступами к элементам вектора. Это затрудняет оптимизацию и приводит к низкой загрузке вычислительных ресурсов.



При оптимизации умножения разреженных матриц на GPU основным фактором является эффективность использования пропускной способности памяти. Это означает, что, с одной стороны, необходимо по возможности уменьшать объем памяти, требуемый для хранения матрицы; с другой стороны, необходимо обеспечить, чтобы обращения из синхронно выполняющихся групп нитей выполнялись к соседним ячейкам памяти (это требование архитектуры GPU для достижения высокой пропускной способности памяти). Обеспечить одновременное выполнение этих требований затруднительно: наиболее эффективными с точки зрения доступа к памяти на GPU являются блочные форматы и такие форматы как ELLPACK, но они приводят к высокому расходу памяти для матриц, не имеющих блочную структуру или имеющие различное количество ненулевых элементов в строках. Эффективный с точки зрения расхода памяти формат CSR плохо подходит для GPU, так как при независимой обработке отдельных строк шаблон обращений к памяти будет неэффективным.

Для реализации умножения разреженных матриц на GPU предлагается использовать гибридный формат Sliced ELLPACK [7] (рис. 3). Базовый формат параметризован числом строк в слайсе матрицы S. Исходная разреженная матрица делится на слайсы из S подряд идущих строк, каждый из которых независимо хранится в формате ELLPACK (как две плотные матрицы K × S, где K — максимальное количество ненулевых элементов в строке; матрицы хранят значение и номер столбца ненулевого коэффициента). Итоговая структура данных содержит конкатенацию плотных матриц слайсов и вспомогательный массив, хранящий смещения плотных матриц для каждого слайса. Для S = 1 этот формат совпадает с обычным CSR, а для S = N — количество строк исходной матрицы — с ELLPACK.

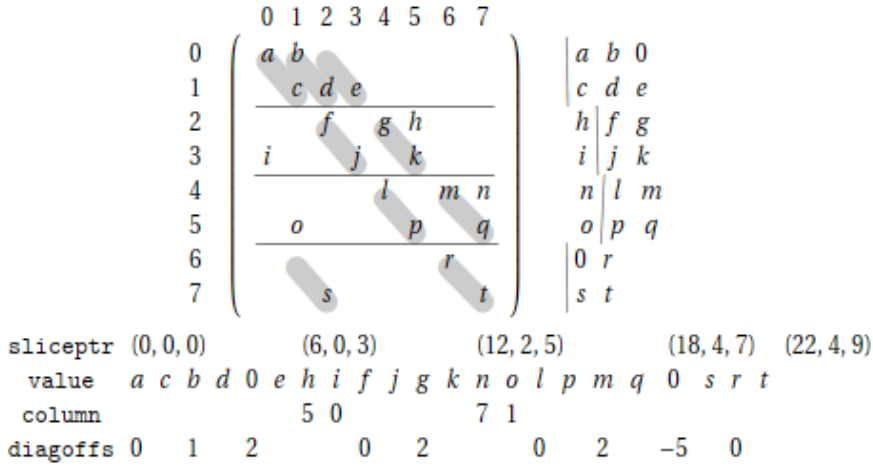


Рис. 5. Пример матрицы с диагональными структурами

Домножение на каждый слайс может выполняться полностью независимо, при этом использование формата ELLPACK в пределах слайса позволяет достичь высокой производительности на GPU. Для матриц с переменным числом ненулевых элементов в строке снижение S позволяет сократить расход памяти, но при этом снижает параллелизм в пределах слайса.

Таким образом, для базового формата Sliced ELLPACK вычислительное ядро параметризовано по S и по Nt — количеству нитей в CUDA-блоке. В качестве кандидатов на значения Nt будем рассматривать только 64, 128, 256, 512 в силу того, что для CUDA предпочтительны размеры блоков нитей, равные степени двойки (иначе возникают конфликты при доступе к регистрам), 512 — максимальный размер блока, а конфигурации с блоками меньших размеров неэффективны. Потребуем также, чтобы S было делителем Nt. Возникающее пространство конфигураций позволяет перебрать все варианты для конкретной разреженной матрицы за короткий период перед решением системы уравнений.

Отметим, что часто в задачах вычислительной гидродинамики матрица с фиксированным расположением ненулевых элементов (но разными значениями коэффициентов, что не влияет на оптимальную конфигурацию ядра) участвует в решении нескольких систем уравнений (при этом каждая система может требовать сотни умножений на матрицу в итерационном методе). Благодаря этому выполненный для первой системы процесс автотюнинга не требуется повторять впоследствии.

Базовый формат Sliced ELLPACK допускает ряд модификаций, сокращающих потребление памяти:

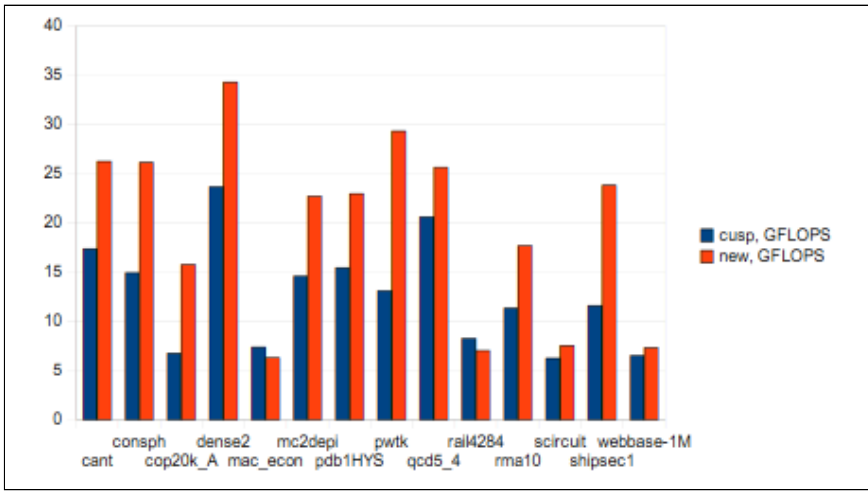


Рис. 6. Производительность реализации умножения разреженных матриц в сравнении с библиотекой cusp на акселераторе Tesla M2090

1) Использование слайсов переменного размера. В этом случае S не фиксировано для всей матрицы, а выбирается отдельно для каждого слайса с целью исключить появление слайсов с большим количеством явно хранимых нулевых элементов из-за большого разброса количества ненулевых элементов в строках слайса.
2) Использование блочных структур (рис. 4). Разреженные матрицы могут не иметь блочной структуры, но доля элементов, принадлежащих плотным однострочным блокам 1 × 2 или 1 × 4 может быть существенна. Поскольку загрузка таких блоков может быть эффективно выполнена на GPU, имеет смысл расширить формат так, чтобы часть элементов можно было хранить в виде плотных блоков. В каждой строке слайса при этом придется выделять одинаковое количество блоков.
3) Использование диагональных структур (рис. 5). Это расширение основывается на аналогичном наблюдении, что доля элементов может располагаться вдоль диагоналей матрицы. При этом достигается дальнейшее сокращение объема памяти, так как для каждого диагонального блока требуется кроме его коэффициентов хранить только одно целое число (смещение относительно главной диагонали). Как и в предыдущем случае, количество и расположение блоков выбирается независимо для каждого слайса [8].

Выбор варианта формата для конкретной матрицы и акселератора производится также за счет автонастройки. В некоторых случаях возможно использование эвристик для выбора подходящего варианта.

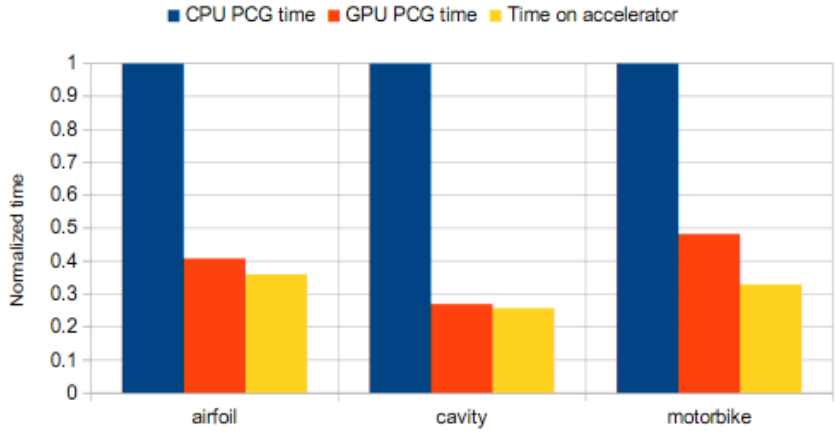


Рис. 6. Производительность реализации умножения разреженных матриц в сравнении с библиотекой cusp на акселераторе Tesla M2090

Разработанная на основе описанного подхода библиотека работы с разреженными матрицами используется в модуле решения систем линейных уравнений в пакете OpenFOAM. Модуль реализует решение линейных систем методом сопряженных градиентов в соответствии с общим интерфейсом, используемым в OpenFOAM для методов решения линейных систем. Поддерживается решение систем в параллельном режиме, когда несколько процессов OpenFOAM взаимодействуют через MPI [9][10].

5. Заключение

В рамках этой работы разработаны методы профилирования кода, поддерживающие задачу переноса кода на параллельные архитектуры. Первый метод направлен на поиск циклов, которые имеет смысл распараллеливать в первую очередь. Второй метод позволяет измерять время переиспользования данных для подпрограмм, которые можно выносить в отдельную нить вычислений.

Разработан метод автоматической генерации параллельного кода во время компиляции для многоядерных или гибридных платформ, поддерживающих OpenCL.

Разработан метод настройки параметров структуры данных и запуска вычислительного ядра для задачи умножения разреженных матриц. Реализованный метод применяется в загружаемом модуле для пакета OpenFOAM, выполняющем решение систем линейных уравнений с использованием GPU акселераторов.

Список литературы

[1] NVIDIA. CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide

[2] Khronos Group. OpenCL. http://www.khronos.org/opencl/
Coccinelle: A Program Matching and Transformation Tool for Systems Code. http://coccinelle.lip6.fr/

[3] E. Berg, H. Zeffer, E. Hagersten. A Statistical Multiprocessor Cache Model. In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006.

[4] А. Белеванцев, А. Кравец, А. Монаков. Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели. Труды Института системного программирования РАН, том 21, стр. 5-22. Москва, 2011

[5] A. Kravets, A. Monakov, A. Belevantsev: GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops. In Proceedings of the GCC Developers' Summit: 9-18, Ottawa, October 2010

[6] A. Monakov, A. Lokhmotov, A. Avetisyan: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In HiPEAC 2010: 111-125, Italy, January 2010

[7] A. Monakov, A. Avetisyan: Specialized Sparse Matrix Formats and SpMV Kernel Tuning for GPUs. In GPU Technology Conference 2012, USA, May 2012

[8] A. Monakov, V. Platonov: Accelerating OpenFOAM with Parallel GPU Linear Solver. In 8th OpenFOAM Workshop, South Korea, June 2013

[9] А. Монаков. Оптимизация расчётов в пакете OpenFOAM на GPU. Труды Института системного программирования РАН, том 22, Стр. 223-232. Москва, 2012. DOI: 10.15514/ISPRAS-2012-22-14.

Analysis and development tools for efficient programs on parallel architectures

Alexander Monakov amonakov@ispras.ru, ISP RAS, Moscow, Russia
Eugene Velesevich evel@ispras.ru, ISP RAS, Moscow, Russia
Vladimir Platonov soh@ispras.ru, ISP RAS, Moscow, Russia
Arutyun Avetisyan arut@ispras.ru, ISP RAS, Moscow, Russia

Abstract. The article proposes methods for supporting development of efficient programs for modern parallel architectures, including hybrid systems. First, specialized profiling methods designed for programmers tasked with parallelizing existing code are proposed. The first method is loop-based profiling via source-level instrumentation done with Coccinelle tool. The second method is memory reuse distance estimation via virtual memory protection mechanism and manual instrumentation. The third method is cache miss and false sharing estimation by collecting a partial trace of memory accesses using compiler instrumentation and estimating cache behavior in postprocessing based on the trace and a cache model. Second, the problem of automatic parallel code generation for hybrid architectures is discussed. Our approach is to generate OpenCL code from parallel loop nests based on GRAPHITE infrastructure in the GCC compiler. Finally, in cases where achieving high efficiency on hybrid systems requires significant rework of data structures or algorithms, one can employ auto-tuning to specialize for specific input data and hardware at run time. This is demonstrated on the problem of optimizing sparse matrix-vector multiplication for GPUs and its use for accelerating linear system solving in OpenFOAM CFD package. We propose a variant of "sliced ELLPACK" sparse matrix storage format with special treatment for small horizontal or diagonal blocks, where the exact parameters of matrix structure and GPU kernel launch should be automatically tuned at runtime for the specific matrix and GPU hardware.

Keywords: optimization, profiling, OpenCL, CUDA, sparse matrices, OpenFOAM

References

[1]. NVIDIA. CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide
[2]. Khronos Group. OpenCL. http://www.khronos.org/opencl/
[3]. Coccinelle: A Program Matching and Transformation Tool for Systems Code. http://coccinelle.lip6.fr/
[4]. E. Berg, H. Zeffer, E. Hagersten. A Statistical Multiprocessor Cache Model. In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006.
[5]. A. Belevantsev, A. Kravets, A. Monakov. Avtomaticheskaya generaciya OpenCL-koda iz gnyozd ciklov s pomoshhyu polie`dral'noy modeli [Automatically generating OpenCL

code from loop nests via a polyhedral model]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, p. 5-22, 2011. (In Russian)

[6]. A. Kravets, A. Monakov, A. Belevantsev: GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops. In Proceedings of the GCC Developers' Summit: 9-18, Ottawa, October 2010

[7]. A. Monakov, A. Lokhmotov, A. Avetisyan: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In HiPEAC 2010: 111-125, Italy, January 2010

[8]. A. Monakov, A. Avetisyan: Specialized Sparse Matrix Formats and SpMV Kernel Tuning for GPUs. In GPU Technology Conference 2012, USA, May 2012

[9]. A. Monakov, V. Platonov: Accelerating OpenFOAM with Parallel GPU Linear Solver. In 8th OpenFOAM Workshop, South Korea, June 2013

[10]. A. Monakov. Optimizaciya raschyotov v pakete OpenFOAM na GPU [On Optimizing OpenFOAM GPU solvers]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 22, p. 223-232, 2012. DOI: 10.15514/ISPRAS-2012-22-14. (In Russian)

Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных

Вартанов С. П., Герасимов А. Ю.
svartanov@ispras.ru, agerasimov@ispras.ru

Аннотация. В статье описываются принципы проведения динамического анализа программ с целью обнаружения в них дефектов различного рода при помощи целенаправленной генерации входных данных. Рассматриваются методы преобразования программ для извлечения трасс выполнения, механизмы отслеживания потоков данных и построения входных данных для покрытия путей выполнения. Показывается, как подобный подход позволяет проводить анализ полностью автоматически без привлечения разработчиков и тестировщиков, на основе исполняемого или интерпретируемого кода. Приводится описание инструментов динамического анализа, разработанных в Институте системного программирования РАН, — инструмента Avalanche, основанного на фреймворке Valgrind, и прототипа инструмента, производящего анализ программ на языке Java. В конце обзора приводятся результаты работы инструмента Avalanche на проектах с открытым кодом и результаты проведения динамического анализа Java-программ с целью поиска ошибок синхронизации.

Ключевые слова: динамический анализ, анализ программ

1. Введение

Современное программное обеспечение используется в различных областях человеческой жизни. Информатизация сфер, связанных с риском для жизни, предъявляет особые требования к качеству выпускаемых программных продуктов.

Известны различные подходы к проверке качества программного продукта:

- тестирование (ручное, полуавтоматическое, автоматическое) — проверка на соответствие спецификации,
- верификация — проверка на соответствие модели или доказательство верности,
- анализ программ на наличие дефектов и уязвимостей.

Все методы можно классифицировать по степени вовлечённости человека в процесс оценки качества программного продукта. Как правило, роль человека в проведении оценки качества связана с необходимостью понимания работы программы, определения критических мест в программе. Это означает требование высокой квалификации проверяющего, знаний об анализируемом продукте и в то же время приводит к значительным временным затратам и не исключает влияние человеческого фактора на оценку качества программ.

В связи с этим желательно, чтобы анализ проводился в полностью автоматическом режиме или роль человека при оценке качества программного продукта была сведена к минимуму. В полностью автоматическом режиме процесс поиска ошибок, связанных с несоответствием готового продукта спецификации, требует наличия спецификации в строго формализованной форме, процесс составления которой также требует участия человека.

При этом очевидно, что необходимо обеспечить отсутствие в программных продуктах ошибок, неизбежно приводящих к нежелательным последствиям, таких как критический отказ программы в случае выполнения недопустимой операции или порча данных, с которыми работает программа. Для поиска ошибок такого рода в настоящее время активно используются два основных метода полностью автоматического анализа программ — статический и динамический.

Структура статьи имеет следующий вид. В разделе 2 и 3 производится обзор методов анализа программ и имеющихся на сегодняшний день решений в области динамического анализа программ. В разделе 4 приведено описание подхода к решению задачи автоматического поиска дефектов в программах путём проведения итеративного динамического анализа программ для целенаправленной генерации входных данных. В разделе 5 рассматриваются особенности практической реализации данного подхода на основе инструментов Avalanche и прототипа инструмента для анализа Java-приложений. В заключении рассматриваются ограничения и сложности, возникающие в процессе динамического анализа программ и проводится обзор возможных направлений дальнейших исследований.

2. Сравнение статического и динамического анализа

Методы *статического анализа* программ предполагают проведение анализа без запуска программы на исполнение с помощью автоматического построения модели программы и последующей обработки построенной модели. Модель программы может быть построена таким образом, что это позволит её проводить частичный или параллельный анализ. Такой подход позволяет существенно сократить время анализа программы. Как правило, модель программы имеет некоторую степень приближения к реальному поведению программы в процессе запуска. В связи с этим поиск ошибок с помощью статического анализа может иметь как ложные срабатывания, так и не обнаруживать некоторые ошибки, присутствующие в программе.

Статический анализ может проводиться, в том числе, в процессе написания исходного кода программы в интегрированной среде разработки, что позволяет разработчикам обнаруживать и исправлять найденные ошибки в процессе написания программы и до её фактической компиляции в исполняемый код [1, 2]. В свою очередь это позволяет уменьшить стоимость исправления ошибки.

Динамический анализ основан на запуске исследуемого продукта на исполнение. Несомненным преимуществом динамического анализа является отсутствие каких-либо предположений о ходе исполнения программы и проверка её в процессе или сразу после исполнения. При этом одно из основных требований, предъявляемых к динамическому анализу — само проведение анализа должно насколько это возможно меньшим образом влиять на ход исполнения. При определённых условиях на детерминированность программы, динамический анализ позволяет избежать проблемы ложных срабатываний.

Главный минус динамического анализа состоит в том, что для получения качественного покрытия анализируемой программы, как правило, требуется неоднократный запуск программы на выполнение, что связано с большими временными затратами. Однако, при обнаружении ошибки в процессе динамического анализа, как правило, возможно сгенерировать входные данные для программы, на которых ошибка воспроизводится. Таким образом появляется возможность исключения ложных срабатываний анализатора.

3. Обзор инструментов динамического анализа программ

На сегодняшний день существует множество инструментов, осуществляющих динамический анализ приложений с целью обнаружения в них ошибок и уязвимостей. Эти инструменты могут быть классифицированы по следующим признакам:

- используемые методы инструментации и построения входных данных,
- зависимость от языков программирования, на которых могут быть написаны анализируемые приложения, или машинная зависимость,
- требование доступности исходного кода приложения,
- необходимость ручного внесения изменений в код,
- типы обнаруживаемых ошибок.

Проведём краткий обзор известных на данный момент решений.

3.1. Инструменты, использующие статическую инструментацию кода программ

Статическая инструментация предполагает предварительную (до начала исполнения) обработку анализируемой программы с целью внедрения в

программу дополнительных инструкций, которые позволяют собирать информацию о ходе выполнения программы, необходимую для анализа.

3.1.1 EXE

EXE [7] — инструмент для поиска ошибок в программах, автоматически генерирующий входные данные, на которых возникают ошибки. Для анализа программ при помощи EXE требуется модификация исходного кода программы. Программист должен указать участки памяти, которые должны восприниматься инструментом, как символьные данные — данные, значения которых изначально не имеют каких-либо ограничений. Например, для того, чтобы пометить 32-битовую переменную `i` как символьную, в код программы необходимо добавить вызов функции `make_symbolic(&i)`. Модифицированный код компилируется при помощи встроенного в EXE компилятора CIL. Полученный в итоге бинарный код используется для осуществления анализа.

EXE выполняет программу *символьно*, т. е. операции не возвращают конкретные значения, в зависимости от своих операндов, а создают ограничения, соответствующие возможности существования тех или иных значений. Если в программе присутствует недетерминизм, из-за подобной модификации EXE может иметь ошибки первого рода.

Если в программе встречается условный переход, зависящий от символьных данных, EXE делает в этом месте развилку, и в одном случае к имеющимся ограничениям добавляются те, что делают переход истинным, в другом — те, что делают переход ложным. Для вычисления выполнимости условий используется решатель Simple Theorem Prover (STP) [5].

В случае возникновения в программе ошибки, которая приводит к аварийному завершению программы, EXE генерирует набор входных данных — тест. EXE обеспечивает высокую степень покрытия исследуемой программы и предоставляет пользователю наборы входных данных для воспроизведения обнаруженных ошибок.

3.2. Инструменты, использующие исследование и обработку трасс выполнения программ без анализа исходного кода

3.2.1 KLEE

Идейное продолжение EXE — инструмент KLEE [8] — символьная виртуальная машина, основанная на компиляторной инфраструктуре LLVM [9] — низкоуровневой виртуальной машине.

Разработкой KLEE занималась та же группа Стенфордского университета, которая работала над созданием EXE. Инструмент возник в результате серьёзной переработки EXE. Главным отличием KLEE от EXE можно считать то обстоятельство, что для анализа приложения при помощи KLEE ручная

модификация кода не требуется. Но необходимо наличие исходного кода и его компиляция при помощи `llvm-gcc`. При этом для повышения эффективности анализа KLEE предоставляет интерфейс для модификации исходного кода без внесения в него побочных эффектов.

Процесс анализа приложения унаследован инструментом KLEE от EXE практически без изменений. Каждый символьный процесс имеет свои файл регистров, стек, кучу, счётчик команд и ограничения пути. KLEE интерпретирует набор инструкций, скомпилированный LLVM, и отображает инструкции в ограничения без аппроксимаций (с точностью до битов). Большое внимание в инструменте уделяется переменным окружения. Как и в EXE, для проверки выполнимости ограничений используется инструмент STP.

3.2.2 SAGE

SAGE [10] (Scalable, Automated, Guided Execution) — инструмент, разработанный компанией Microsoft совместно с Калифорнийским университетом в Беркли, для анализа Windows-приложений для систем x86.

Основным недостатком EXE, сохранившимся и в KLEE, можно считать необходимость наличия исходного кода программы для её анализа. SAGE лишён этого недостатка, поскольку работает на уровне машинных операций. Это делает инструмент независимым от языка программирования, на котором написано приложение, а также от используемого компилятора. При этом инструмент ориентирован на систему команд x86, что делает его машиннозависимым.

Подход, используемый в SAGE схож с подходом EXE и KLEE. SAGE поддерживает фактическое и символьное состояния программы, представленные парой — байтовое значение и символьный тег, — ассоциированной с каждой ячейкой памяти. Символьный тег — выражение, определяющее, какое значение может принимать данная ячейка.

Также в отличие от рассмотренных ранее инструментов, SAGE при осуществлении анализа не проводит целенаправленную проверку опасных операций, а фокусируется на построении как можно более полного покрытия возможных путей исполнения исследуемой программы.

3.3. Инструменты, использующие динамическую инструментацию программ

Динамическая инструментация означает, что все изменения программы производятся непосредственно перед исполнением соответствующего её блока на процессоре. Инструменты этой группы используют концепцию отслеживания потока «помеченных данных».

Следующие инструменты основаны на среде Valgrind [11]. Valgrind — свободное инструментальное программное обеспечение, предназначенное для осуществления динамической инструментации исполняемого кода. Valgrind

предоставляет средства для создания программных модулей, проводящих динамический анализ. В рамках самого проекта в числе прочих разработаны инструменты для автоматического обнаружения ошибок управления памятью Memcheck и потоками Helgrind.

3.3.1 Flayer

Инструмент Flayer [12] предназначен для динамического анализа бинарного кода программы. При помощи Valgrind производится динамическая инструментация кода с целью выявления выполнения условных операторов. Производится проверка на то, являются ли данные, от которых зависит переход, помеченными, т. е. зависящими от входных данных программы. Если это так, значения условий в операторах изменяются для обеспечения обхода различных путей выполнения программы.

В связи с тем, что изменение значений данных в условных переходах происходит непосредственно в ходе выполнения программы, недостатком работы приложения является возможность ложных срабатываний. Это означает, что воспроизводимость найденной ошибки не гарантируется.

3.3.2 Catchconv

Инструмент Catchconv [13] по схеме своей работы схож с инструментом Flayer, однако лишён проблемы воспроизводимости ошибок.

Подобно инструментам EXE и KLEE, в Catchconv используется решатель STP для проверки выполнимости условий на возможные значения аргументов в условных переходах, составленных инструментом в ходе проведения анализа. Как и в инструменте Flayer, извлечение трассы и обнаружение условных переходов производится при помощи динамического анализа с использованием Valgrind. Среди недостатков Catchconv можно отметить обнаружение инструментом ограниченного класса ошибок. К ним относятся ошибки использования знаковых и беззнаковых типов данных.

3.4. Применение динамического анализа для языка Java

Рассмотрим применение динамических методов анализа для программ, написанных на языке Java. Основная особенность в свете рассматриваемой задачи анализа — трансляция языка в промежуточное представление, байт-код, и его интерпретация при помощи виртуальной машины.

3.4.1 Java PathFinder

Идеи символьных вычислений для проведения динамического анализа программ на языке Java нашли своё отражение в инструменте Java PathFinder [15]. В соответствии с этими идеями инструмент строит наборы булевых ограничений (булевых формул без кванторов над символьными переменными) для путей и на их основе формирует дерево символьного выполнения.

Для проведения символьных вычислений используется инструментация кода с целью получения дерева состояний анализируемого приложения. Состояния включают в себя конфигурацию кучи, ограничения пути и планирование потоков. Это дерево как пространство состояний исследуется при помощи основного механизма инструмента — проверки на модели (model checking).

Инструмент используется для генерации тестовых входных данных и построения контрпримеров для отдельных свойств модели.

3.4.2 Java ThreadSanitizer

Инструмент Java ThreadSanitizer [16], созданный в рамках проекта ThreadSanitizer, использует фреймворк ASM для динамической инструментации байт-кода Java, в ходе которой к функциональности анализируемого приложения добавляется генерация трассы событий. К событиям относятся доступы программы к памяти на чтение или запись, инструкции создания и управления потоками и операции управления синхронизацией потоков.

Получаемая после выполнения программы трасса используется экспериментальным инструментом ThreadSanitizer Offline, который на основе построения отношений предшествования осуществляет поиск возможных состояний гонки.

3.5. Выводы

Рассмотренные инструменты демонстрируют разнообразие подходов, применяемых для динамического анализа приложений. Общей чертой для всех программных средств является просмотр различных путей выполнения программы с целью обнаружения ошибок. Это производится при помощи инструментации или предварительного изменения кода и символьного выполнения программы. При этом одни средства нацелены на обеспечение наиболее полного покрытия всех путей, другие — на покрытие наибольшего числа потенциально опасных операций.

Другие различия состоят в том, основывается ли инструмент на исходном коде программ, на машинном коде или каком-то его промежуточном представлении. Также среди различий можно выделить типы обнаруживаемых ошибок: критические ошибки, связанные с аварийным завершением программы, ошибки, связанные с доступом к неинициализированной памяти или выполнением недопустимых операций, ошибки синхронизации в многопоточных приложениях или утечки выделяемой памяти.

4. Итеративный динамический анализ

Далее в статье более подробно рассматриваются аспекты методов проведения динамического анализа, исследованные в рамках разработки инструмента Avalanche и ряда прототипов для анализа программ на языке Java. В этом

разделе приведены используемые методы и решения и их сравнение с альтернативными способами.

4.1. Способы покрытия

При автоматическом анализе программ важно понимать, на каком этапе находится процесс анализа и когда его можно считать завершённым. На этот вопрос можно дать ответ при помощи определения полноты покрытия программы. Можно говорить, что исследуемый продукт проанализирован полностью, если были выполнены все

- вызовы функций или методов,
- операторы,
- условные переходы,
- или пути выполнения.

Некоторые из приведённых здесь вариантов покрытий целиком включают в себя другие. Например, выполнение всех операторов в программе автоматически означает выполнение и всех условных переходов хотя бы один раз.

Одним из наиболее полных покрытий, включающим в себя множество остальных, является покрытие всех достижимых путей выполнения программы. Такой критерий является очень тяжеловесным. Во-первых, для анализа потребуется совершить число запусков приложения, равное количеству достижимых путей, а оно растёт экспоненциально с ростом числа условных переходов. Во-вторых, возникает алгоритмически неразрешимая проблема останова анализируемой программы, когда для конкретного пути невозможно определить, следует ли прекратить анализ, поскольку программа зациклилась, или подождать её завершения.

Однако, несмотря на все издержки, даже при частичном покрытии программы на основе описанного критерия можно говорить о высоком качестве анализа просмотренных путей. Благодаря методам распараллеливания вычислений и применения распределенных вычислений можно частично решить проблему производительности анализа [17].

4.2. Определение путей выполнения

В дальнейшем в статье, если не оговорено обратное, речь будет идти об анализе программ или об анализе тех частей программы, выполнение которых детерминировано. Это означает, что на каждом шаге работы состояние программы полностью и однозначно (на основе имеющегося кода) зависит от её состояния на предыдущем шаге, а также, что последовательность шагов также детерминирована.

Дополнительно к этому следует оговорить, что любая информация, которая будет использоваться в размышлениях в контексте анализируемой программы

— будь то используемые в программе переменные или входные данные — является дискретной и конечной.

4.3. Формализация задачи построения входных данных

Покажем далее, что задача построения входных данных в указанных ограничениях для неинтерактивных программ может быть сведена к задаче проверки выполнимости булевых ограничений.

Поскольку мы считаем, что программа не интерактивна, её входными данными могут быть файлы, аргументы командной строки и т. п. Вся входная информация дискретна, а следовательно может быть представлена в виде булевого вектора — набора булевых переменных:

$$x = (x_1, \ldots, x_n)$$

Будем также считать, что программа работает с ограниченной дискретной памятью, которая состоит из *k* булевых переменных. Будем рассматривать граф потока управления программы — граф, в котором вершинам соответствуют инструкции программы, а рёбрам — возможные переходы.

Любое выполнение программы можно представить как путь (конечный, если программа остановилась, или бесконечный, в случае, если программа зациклилась) в графе потока управления. Рассмотрим определённый путь π. Пусть его длина есть *t*. Для каждой вершины с номером *i* введём вектор

$$z_i = (z_{i_1}, \ldots, z_{i_k}), \forall i = \overline{1,t}$$

переменных, соответствующих используемым в программе переменным.

Поскольку программа является детерминированной, каждое z_{i_j} зависит только от входных данных и предыдущих состояний программы, т. е. представляет собой булеву функцию от переменных

$$\begin{gathered} x_1, x_2, \ldots, x_n, \\ z_{1_1}, z_{1_2}, \ldots, z_{1_t}, \\ z_{2_1}, z_{2_2}, \ldots z_{2_t}, \\ \ldots \\ z_{i-1_1}, z_{i-1_2}, \ldots, z_{i-1_t}. \end{gathered}$$

Индукцией по номеру вершины пути можно доказать, что любая переменная z_{i_j} зависит только от входных данных. В самом деле, это верно для всех переменных первой вершины — $z_{1_j}, j = \overline{1,t}$. Если это верно для всех вершин с номером, меньшим *l*, то каждая переменная этих вершин имеет вид

$$z_{i_j} = g_{l-1_j}(x), i = \overline{1, l-1}, j = \overline{1,t}.$$

Для вершины *l* каждая переменная имеет вид

$$z_{l_i} = \\ = g_{l_i}(x_1,\dots,x_n,z_{l_1},\dots,z_{1_t},\dots,z_{l-1_i}\dots z_{l-1_t}) = \\ = g_{l_i}(x_i,\dots,x_n,g_{1_1}(x),\dots,g_{1_t},g_{1_t}(x),\dots g_{l-1_1}(x),\dots g_{l-1_t}(x)) = \\ = g'_{l_i}(x), j = \overline{1,t}.$$

Это означает, что любая переменная в каждой вершине пути зависит *только* от входных данных. Отметим, что путь однозначно определяется выбором следующего ребра в каждой вершине графа, из которой выходят несколько рёбер. Такие вершины соответствуют условным переходам в программе.

Каждый условный переход определяется условием. В нашем случае условие можно записать как некоторую логическую функцию от переменных из множества *x*. Значение этой функции определяет, будет ли выполнен данный условный переход, т. е. какое из рёбер, исходящих из вершины, соответствующей этому переходу, будет выбрано.

Обозначим множество всех условных переходов в программе как $B = \{B_1,\dots,B_m\}$и запишем для каждого перехода B_iфункцию $f_i(x)$, соответствующую его условию, то есть переход совершается тогда и только тогда, когда значение функции истинно.

Снова рассмотрим путь π. Как было отмечено, он однозначно определяется всеми выборами в условных переходах. Поставим в соответствие π последовательность всех переходов, встречающихся при обходе пути — $B_\pi = \{B_{i_1},\dots,B_{i_r}\}$.

Для каждого перехода B_{i_l}из последовательности определим, какое ребро следует за соответствующей вершиной в пути π. Если ребро имеет пометку, что переход выполнился, поставим ему в соответствие функцию f_{i_1}, иначе — функцию $\overline{f_{i_l}}$. Если функция не тождественно истинна, она может быть представлена в виде конъюнктивной нормальной формы f'_{i_l}.

Выполнимость каждого перехода B_{i_l}определяется равенством истине функции f'_{i_l}. Весь путь однозначно определяется выполнимостью конъюнктивной нормальной формы $F = f'_{i_1} \wedge \dots \wedge f'_{i_t}$.

Задачу можно формализовать следующим образом. Необходимо найти такие значения переменных из набора *x*, чтобы значение функции *F* было истинно. Эта формулировка полностью соответствует формулировке задачи определения выполнимости булевых формул. Согласно теореме Кука — Левина, эта задача является NP-полной [3].

Для решения задачи проверки выполнимости логических формул на сегодняшний день существует множество инструментов, так называемых решателей. В их число входит, например, MINISAT, эффективно решающий сформулированную выше задачу [4], или STP Constraint Solver, который оперирует с битовыми массивами и поддерживает

- арифметические операции,
- операции сравнения,
- логические операции,
- а также конкатенацию, сдвиги и пр. [5]

4.4. Инвертирование условий

Вернёмся теперь к основной задаче — построению всех допустимых путей в программе. Очевидно, не обязательно все возможные пути в терминах графа потока управления программы являются допустимыми путями выполнения программы. Гарантированно полный проход по всем допустимым путям даёт перебор всех возможных входных значений программы. Одним из возможных решений может быть генерация случайных входных данных и запуск программы на них с последующим запоминанием пройденных путей для подсчёта покрытия. Однако, для каждого пути достаточно получить хотя бы один набор входных данных. Рассмотрим далее способ перебора возможных путей, вместо перебора входных данных.

Одним из таких способов является *инвертирование условий*. Суть метода заключается в следующем. Для первого выполнения программы используется некоторый набор входных данных. Концептуально это может быть абсолютно случайный набор. Однако, эффективность метода повышается, если выполнение программы на начальных входных данных будет, например, содержать как можно больше условных переходов.

Во время первого выполнения программы на начальных входных данных для каждого встретившегося условного перехода строится набор условий, при истинности которых выполнение в этой точке сворачивает по альтернативной ветке. Для каждого набора условий строятся (если возможно) удовлетворяющие им входные данные, происходит запуск программы и описанные действия повторяются для инвертирования оставшихся условий.

Такой подход теоретически даёт возможность обойти все возможные пути в программе.

4.5. Отслеживание потока помеченных данных

Для указанного метода инвертирования условий нужен, во-первых, некоторый механизм построения условий в ходе выполнения программы и создания наборов условий, отвечающих новым путям; во-вторых, механизм преобразования построенных ограничений в булевы формулы и проверки их

выполнимости; в-третьих, механизм выбора на каждом новом шаге некоторого нового пути.

Задача преобразования набора логических выражений в булевы формулы, преобразование в конъюнктивную нормальную форму, решение проблемы проверки выполнимости и оптимизации, связанные с каждым из этих процессов, выходят за рамки статьи и не будут рассмотрены здесь. Остановимся на проблеме построения набора ограничений, определяющего каждый отдельный путь на основе запусков анализируемой программы.

Рассмотрим понятие помеченных данных. *Помеченными данными*, используемыми в программе, будем называть любые данные, значения которых зависят от входных данных (в [6] помеченные данные определяются иначе: это либо входные данные, либо результат операции, в которой участвуют помеченные данные). Следует отметить, что свойство «помеченности» данных вообще говоря не является постоянным в ходе выполнения программы. Изначально и постоянно помеченным является исходный набор входных данных. Любые же переменные, используемые в программе могут становиться помеченными или терять это свойство в ходе выполнения, в зависимости от того, в каких операциях они участвуют.

Концепцию помеченных данных легко проиллюстрировать на простом примере:

```
int main(int argc, char** argv)
{
    // Arguments checking

    int a = read_from_file(0);
    int b = 1;

    int c = a + 5;
    int d = b + 5;

    if (c == 6)
    {
        // First block
    }
    else
    {
        // Second block
    }
    if (d == 6)
    {
        // Third block
    }
    else
    {
        // Fourth block
    }
}
```

Переменной `a` вызовом функции `read_from_file` присваивается значение первого байта некоторого файла, который является частью набора входных данных, следовательно, эта переменная «помечается» в данной точке. Переменная `b`, очевидно, помеченной не является — её значение в данной точке программы не зависит от того, какие данные подаются программе на вход. Переменная `c` после присваивания, зависящего от значения переменной `a`, станет косвенно зависящей от входных данных, и значит, станет помеченной. Иначе дело обстоит с переменной `d`, значение которой от входных данных не зависит. Таким образом, результат операции считается помеченным, если в ней *существенным* образом участвуют помеченные данные.

Для каждого условного перехода на основе помеченности данных аналогичным образом определяется, зависит ли его условие транзитивно от входных данных. В условии второй if — then — else конструкции в примере переменная `d` не помечена, а значит условие заведомо не зависит от входных данных. В данном случае всегда будет выполнен третий блок программы, тогда как четвёртый блок — недостижимый код. Условие в первой конструкции существенно зависит от помеченной переменной `c` и это означает, что, возможно, существуют различные наборы входных данных, при которых в данной точке выполнение программы пойдёт по разным веткам, однако их существование не гарантировано.

Следует отметить, что в реализациях обычно снимается требование *существенного* участия помеченных переменных в операциях для того, чтобы результат считался помеченным. Допустим, имеется помеченная переменная `a`. Обе конструкции `int b = a - a;` и `int c = a * 0;` осуществляют присваивание нуля переменным `b` и `c`, однако в обеих операциях участвует помеченная переменная. Для проверки, является ли это участие существенным, необходимо производить разбор правой части выражений и определять области возможных значений, что требует значительных накладных расходов.

Концепция помеченных данных в рассмотренном подходе позволяет определить условные переходы, условия в которых на конкретном пути не зависят от входных данных и, следовательно, на этом пути не могут быть инвертированы путём изменения входных данных. На практике такое отсечение позволяет существенно сократить количество переходов, условия которых требуют проверки выполнимости, что, в свою очередь, важно, поскольку это NP-полная задача и её решение может требовать значительного времени.

4.6. Поиск ошибок и уязвимостей

Описанные методы позволяют генерировать наборы входных данных программы для покрытия различных путей выполнения. Повышение эффективности анализа программы в этом случае может обеспечиваться

использованием эвристических методов выбора нового пути на каждом последующем шаге. Перебор путей может проходить в глубину, в ширину, или основываться на различных метриках: увеличения покрытия числа условий, инструкций и т. д.

Выполнение определённого пути программы (т. е. запуск программы на конкретных входных данных) позволяет извлечь информацию о ходе выполнения. Эта информация может использоваться как для простой проверки хода выполнения и состояния, в котором программа завершилась, на наличие ошибок, так и для построения различного рода моделей и проверки свойств этих моделей.

Отдельно стоит обратить внимание на методы моделирования для поиска ошибок при помощи проверки операций, потенциально способных приводить к аварийному завершению программы при определённых значениях аргументов. К таким операциям можно отнести деление или разыменование указателя. Для таких операций на каждом пути, в котором они присутствуют, необходимо определить, существуют ли входные данные, на которых значения аргументов для них приводят к ошибке. Для этого используются те же принципы, что и при построении входной информации для инвертирования переходов, с тем лишь отличием, что условие инвертирования заменяется ограничением на значение указанных аргументов.

5. Реализация методов динамического анализа

5.1. Инструмент Avalanche

Описанные выше принципы и методы проведения динамического анализа были реализованы в инструменте Avalanche [6].

Инструмент основан на фреймворке Valgrind. В основе инструмента лежит динамическая инструментация кода, проводимая двумя различными компонентами — программными модулями Valgrind — Tracegrind и Covgrind. Первый программный модуль используется для инструментации программы с целью отслеживания помеченных данных и построения наборов ограничений для инвертирования условных переходов и построения входных данных. Таким образом, инструмент реализует описанный подход к построению различных путей выполнения программы.

Второй программный модуль — Covgrind — инструментирует анализируемую программу с целью определения покрытия базовых блоков. Для каждого нового пути, построенного при помощи инвертирования условных переходов, Covgrind вычисляет количество новых базовых блоков, которые будут покрыты после анализа. На основе такой метрики происходит выбор новых путей выполнения анализируемой программы.

Поиск ошибок в Avalanche основан на определении статуса завершения программы, описанном способе проверки опасных операций, а также

использовании различных программных модулей фреймворка Valgrind, таких как Memcheck для проверки операций работы с памятью и Helgrind для проверки управления потоками.

Также в Avalanche есть механизм анализа сетевых приложений. В этом случае помеченными изначально считаются все данные, поступающие анализируемой программе от сервера.

В дополнение к сказанному, Avalanche предоставляет возможности для повышения эффективности анализа:

- Выделение во входных файлах набора байт, которые необходимо считать изначально помеченными. При этом возникает возможность, например, зафиксировать заголовок входного файла с целью подбора его различного содержимого.
- Указание списка сигнатур функций, выражения в условных операторах которых должны быть инвертированы в ходе анализа. Это позволяет производить локальных анализ некоторых участков программы.

В последней версии программы были добавлены возможности проведения параллельного и распределённого анализа. Как было сказано, после каждой итерации анализа, в зависимости от количества встретившихся условных переходов, строится множество наборов ограничений для их инвертирования. Распараллеливание анализа основано на независимости задач определения выполнимости имеющихся ограничений и запуска анализируемой программы с инструментацией для определения прироста покрытия.

Проведение с помощью Avalanche распределённого анализа на нескольких вычислительных единицах или в распределённой вычислительной среде осуществляется на основе деления построенного дерева достижимых путей программы на ряд поддеревьев и проведения их независимого анализа с последующим объединением результатов. Результаты работы в распределенном и параллельном режиме приведены в работе [17].

5.2. Динамический анализ программ на языке Java

Описанные методы применимы для широкого класса императивных языков программирования. Как следует из обзора существующих решений, эффективные средства решения поставленной задачи методами динамического анализа используются и для интерпретируемых языков, таких, как язык Java.

В рамках исследований авторами был создан прототип средства динамического анализа программ на языке Java. В основу были положены, как и в инструменте Avalanche, принципы отслеживания помеченных данных и построения новых входных данных с помощью решения булевых ограничений.

Основная особенность языка Java, как уже отмечалось, — трансляция в байт-код и интерпретация при помощи виртуальной машины. Формат байт-кода

определяется конкретной машиной, при помощи которой будет производиться интерпретация. В связи с этим, в отличие от Avalanche, в прототипе используется статическая инструментация байт-кода с помощью библиотеки BCEL [14]. Это означает, что преобразование кода с целью внесения в него дополнительной функциональности осуществляется до проведения итеративного анализа, а не повторяется на каждой его итерации.

Такое изменение способа инструментации, хотя и может означать инструментацию частей программы, которые никогда не будут выполнены в ходе анализа (например, это может касаться частей используемых библиотек), сокращает время, затрачиваемое на проведение отдельной итерации, а также позволяет проводить анализ на платформах, поддерживающих Java-машины с иным форматом байт-кода, путём предварительного конвертирования инструментированного байт-кода в требуемый формат. Так, прототип был реализован для инструментации байт-кода формата Java Virtual Machine, однако при наличии средства конвертирования в формат DEX позволяет анализировать программы на платформе Android, использующей для интерпретации байт-кода виртуальную машину Dalvik.

Для обнаружения ошибок данный прототип использует механизм исключений языка Java в том смысле, что любое выброшенное и не отловленное программно исключение считается дефектом. В прототипе отсутствует отдельный механизм инструментации для подсчёта полноты покрытия. Вместо этого информация о числе инструкций оценивается на основе предыдущих запусков. Результаты исследования приведены в статье [18].

6. Заключение

Несмотря на тяжеловесность методов динамического анализа, существующие решения уже позволяют автоматически обнаруживать ошибки в программах.

Однако, стоит отметить, что в области исследований методов динамического анализа программ есть ряд известных проблем, которые не позволяют в настоящее время использовать средства динамического анализа программ в каждодневной работе программистов и сотрудников отделов контроля качества программного обеспечения:

- Проблема экспоненциальной зависимости количества запусков анализатора от количества условных переходов в программе.
- Проблема целенаправленного анализа определенной части программы.
- Проблема влияния инструкций инструментации на общую производительность выполнения и анализа программы.

В связи с этим можно сформулировать ряд задач для дальнейших исследований в области анализа программ.

6.1. Преобразование среды выполнения как альтернатива инструментации

В рамках исследований в ИСП РАН был создан прототип средства, производящего анализ выделения и использования памяти в приложениях на платформе Android. Для интерпретации Java-приложениях Android использует виртуальную машину Dalvik. Так как поставленная задача предусматривает определение использования памяти с точностью до байтов, т. е. требует отслеживания всех операций с памятью, изменение хода выполнения анализируемого приложения при помощи инструментации оказывается неэффективным. В связи с этим было принято решение об изменении самой среды выполнения байт-кода для извлечения необходимой информации, вместо инструментации программы.

Технически решение состоит в создании клиент-серверного приложения, отправляющего и принимающего информацию от анализируемого приложения при помощи отправки сообщений, и изменении кода виртуальной машины Dalvik путём добавления функциональности по отправке сообщений с извлечённой информацией о выделении, высвобождении и использовании памяти.

Подобный подход также может быть использован для динамического анализа с целью поиска ошибок, уязвимостей или неэффективностей любого рода. Несмотря на то, что решение зависит от используемой виртуальной машины, оно может быть значительно эффективнее инструментации.

6.2. Интеграция со статическим анализом

Как отмечалось ранее, один из существенных недостатков статического анализа — необходимость проверки воспроизводимости обнаруженных дефектов. Для решения этой проблемы может быть использовано совмещение двух подходов для устранения ряда недостатков каждого из них.

В случае, если существует возможность средствами статического анализа сделать предположение о возможной трассе выполнения программы, приводящей к ошибке, динамический анализ может подтвердить обнаруженный дефект путём построения входных данных, на которых данный дефект воспроизводится, либо опровергнуть его с помощью доказательства невыполнимости набора утверждений, определяющих соответствующий путь и состояние системы.

В таком случае решается две задачи:

- воспроизведение дефекта, найденного статическим анализатором путём вычисления входных данных в динамическом анализаторе,
- решение проблемы анализа определенной части программы динамическим анализатором.

Список литературы

[1] Савицкий В. О., Сидоров Д. В. «Ленивый» анализ исходного кода на языках C и C++. Труды Института системного программирования РАН, том 23, 2012 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2012-23-8, 133–141 с.

[2] Савицкий В. О., Сидоров Д. В. Инкрементальный анализ исходного кода на языках C/C++. Труды Института системного программирования РАН, том 22, 2012 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2012-22-8, 119–130 с.

[3] Новикова Н. М. Основы оптимизации. М.: МГУ, 1998. 17–22 с.

[4] Eén N., Sörensson N. An Extensible SAT-solver. SAT 2003. P. 502-518.

[5] Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays // In Proceedings of Computer Aided Verification. 2007. P. 524–536.

[6] Исаев И. К., Сидоров Д. В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах // Программирование. 2010. № 4. С. 16.

[7] Cadar C., Ganesh V., Pawlowski P., Dill D., Engler D. EXE: Automatically Generating Inputs of Death // Computer System Laboratory Stanford University. P. 14.

[8] Dunbar D., Cadar C., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Stanford University. 2008.

[9] Lattner C. The LLVM Compiler Infrastructure [HTML] (http://llvm.org/)

[10] Goldefroid P., Levin. M. Y, Molnar D. SAGE: Whitebox Fuzzing for Security Testing // Communications of the ACM. 2012. 55. P. 40–44.

[11] Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)

[12] Drewry W., Ormandy T. Flayer: Taint analysis and flow alteration tool [HTML] (http://code.google.com/p/flayer/)

[13] Molnar D., Wagner D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors // UC Berkeley, 2007.

[14] Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[15] Visser W., Păsăreanu C. S., Khurshid S. Test Input Generation with Java PathFinder // Proceeding ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. P. 97–107.

[16] Serebryany K., Iskhodzhanov T. ThreadSanitizer—data race detection in practice. WBIA '09, New York City, NY, USA, 2009

[17] Ермаков М. К., Герасимов А. Ю. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей. Труды Института системного программирования РАН, том 25, 2013 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2013-25-2, стр. 29-38.

[18] Вартанов С. П., Герасимов А. Ю. Применение динамического анализа для поиска дефектов в программах на языке Java. Труды Института системного программирования РАН, том 25, 2013 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2013-25-1, стр. 9-28.

Dynamic program analysis for error detection using goal-seeking input data generation

Vartanov S. P., Gerasimov A. Y.
svartanov@ispras.ru, agerasimov@ispras.ru

Abstract. This paper describes the principles of program dynamic analysis for defect detection using input data generation. Presented comparison of dynamic vs static analysis od programs, overview of existing tools for dynamшс analysis such as EXE, KLEE, SAGE, Flayer, Catchconv, Java PathFinder, Java ThreadSanitizer. Techniques of program transformation allowing execution trace extraction, data flow tracing and input data generation for execution path coverage approaches are considered. We clarify in what way such an approach allows us to perform fully automatic analysis using executable or interpretable code based on iterational dynamic analysis with automatic conditional branches alternation through input data generation for target program. This paper also presents dynamic analysis tools developed at Institute for System Programming RAS---Avalanche (Valgrind-based tool) and a prototype tool for Java applications. These tools allows to find critical defects programs which lead to program crash and generate input data sets for reproducing found defects. The paper concludes with an evaluation of practical results of applying Avalanche tool to a set of open source projects as well as results of applying Java analysis tool to detect concurrency defects and describes possible directions for future research.

Keywords: dynamic analysis, program analysis

References

[1]. V. O. Savitsky, D. V. Sidorov. «Lenivyj» analiz iskhodnogo koda na yazykakh C i C++ [Lazy source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 23, pp. 133-141. DOI: 10.15514/ISPRAS-2012-23-8. (in Russian)

[2]. V. O. Savitsky, D. V. Sidorov, Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++[Incremental source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 22, pp. 119-129. DOI: 10.15514/ISPRAS-2012-22-8. (in Russian)

[3]. Novikova N. M. Osnovy optimizatsii [The Basics of Optimization]. M.: MGU [MSU], 1998. 17–22 p. (in Russian)

[4]. Eén N., Sörensson N. An Extensible SAT-solver. SAT 2003. P. 502-518.

[5]. Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays // In Proceedings of Computer Aided Verification. 2007. P. 524–536.

[6]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1-16. (in Russian)
[7]. Cadar C., Ganesh V., Pawlowski P., Dill D., Engler D. EXE: Automatically Generating Inputs of Death // Computer System Laboratory Stanford University. P. 14.
[8]. Dunbar D., Cadar C., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Stanford University. 2008.
[9]. Lattner C. The LLVM Compiler Infrastructure [HTML] (http://llvm.org/)
[10]. Goldefroid P., Levin. M. Y, Molnar D. SAGE: Whitebox Fuzzing for Security Testing // Communications of the ACM. 2012. 55. P. 40–44.
[11]. Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)
[12]. Drewry W., Ormandy T. Flayer: Taint analysis and flow alteration tool [HTML] (http://code.google.com/p/flayer/)
[13]. Molnar D., Wagner D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors // UC Berkeley, 2007.
[14]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)
[15]. Visser W., Păsăreanu C. S., Khurshid S. Test Input Generation with Java PathFinder // Proceeding ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. P. 97–107.
[16]. Serebryany K., Iskhodzhanov T. ThreadSanitizer—data race detection in practice. WBIA '09, New York City, NY, USA, 2009
[17]. Ermakov M. K., Gerasimov A. Y. Avalanche: adaptation of parallel and distributed computing for dynamic analysis to improve performance of defect detection [Avalanche: primenenie parallel'nogo i raspredelennogo dinamicheskogo analiza programm dlya uskoreniya poiska defektov i uyazvimostej] Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol 25, pp. 29-38. DOI: 10.15514/ISPRAS-2013-25-2. (in Russian)
[18]. Vartanov S. P., Gerasimov A. Y. Applying dynamic analysis for defect detection in Java-applications [Primenenie dinamicheskogo analiza dlya poiska defektov v programmakh na yazyke Java] Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol 25, pp. 9-28. DOI: 10.15514/ISPRAS-2013-25-1. (in Russian)

Рефакторинг в рамках программного проекта

С. В. Сыромятников (syrom@ispras.ru), И. Е. Бронштейн (ibronstein@ispras.ru), Н. Л. Луговской (lugovskoy@ispras.ru)

Аннотация. Рефакторинг является одной из самых популярных и «успешных» техник улучшения исходного кода. Он является неотъемлемой частью гибкой методологии разработки. Однако, до сих пор наблюдается недостаток в существовании «качественных» средств проведения автоматического рефакторинга исходного кода на языках C/C++. В данной статье рассматривается один из подходов к разработке инструмента для проведения такого рефакторинга. Стоит отметить, что возможность проведения рефакторинга только на одной единице компиляции является существенным ограничением любого создаваемого инструмента. Поэтому важной особенностью данной статьи является подробное описание перехода от схемы проведения рефакторинга на одной единице компиляции к схеме проведения рефакторинга в рамках всего проекта. Кроме того, особое внимание в статье отводится рефакторингу «Переименование», так как это один из самых распространенных рефакторингов, проводимых в рамках всего проекта.

Ключевые слова. рефакторинг; переименование; глобальная область видимости; статический анализ

1. Введение

Рефакторинг - это процесс изменения внутренней структуры программы, не влияющий на её видимое поведение и имеющий целью облегчить понимание её работы [1]. Он позволяет упростить исходный код и сделать его более понятным для разработчиков. Кроме того, подобное изменение программы способно облегчить её дальнейшую модификацию и расширение. Существует достаточно большое количество программных средств, позволяющих автоматизировать процесс рефакторинга. В данной статье рассматриваются решения, реализованные в инструменте *Klocwork Insight* [2], разработанном в ИСП РАН в рамках контракта с компанией Klocwork Inc.

Выделяют разные виды автоматического рефакторинга, поддерживаемые, например, программными инструментами *Eclipse CDT*, *CodeRush* [3,4]. Для нас представляет интерес тот факт, что рефакторинг бывает *локальным*, то есть осуществляемым в рамках одной единицы компиляции, и *глобальным*, то есть затрагивающим несколько таких единиц, и потому осуществляемым в рамках всего проекта (здесь и далее под *единицей компиляции* мы имеем в

виду указываемый в команде компиляции исходный файл вместе с подключаемыми в него заголовочными файлами). Так, переименование локальных переменных является исключительно локальным рефакторингом. С другой стороны, для глобальных переменных и глобальных функций именно глобальное переименование является единственно правильным. Распространённой практикой является объявление функций (а в случае C++ - и классов) в заголовочных файлах и подключение этих файлов в несколько единиц компиляции. Очевидно, что локальное переименование таких функций, классов, методов классов, вероятнее всего сделает код некорректным.

В данной статье мы рассмотрим возможное устройство инструмента для проведения глобального рефакторинга «Переименование», опишем проблемы, возникающие в процессе создания такого инструмента, и способы их решения.

2. Локальный рефакторинг

Автоматический рефакторинг для языков C/C++ в инструменте *Klocwork Insight* основан на принципах сохранения синтаксической корректности кода и неизменности поведения программы после преобразования. Глубокая интеграция компилятора и инструмента для проведения рефакторинга позволяет проводить изменение исходного кода максимально точно, вплоть до сохранения пользовательской индентации и комментариев.

Общая схема локального рефакторинга в *Klocwork Insight*, описана ранее [5], и схематически выглядит так:

1. Проводится лексический, синтаксический и семантический анализ исходного кода. В результате строится синтаксическое дерево, некоторые узлы которого снабжены семантической информацией.

2. Выделяется подмножество синтаксического дерева, требующее преобразования. Для этого пользователь тем или иным способом (при помощи графического интерфейса или при помощи опций командной строки) указывает фрагмент программного текста, и по этому фрагменту определяются соответствующие ему узлы синтаксического дерева.

3. Автоматически определяются другие места в синтаксическом дереве или непосредственно в коде, которые должны или могут быть преобразованы. Так, при выделении фрагмента кода в новую функцию бывает полезно найти другие фрагменты, аналогичные выделенному, и предложить пользователю заменить их на вызов новой функции. Эту задачу традиционно называют *выявлением дубликатов*. Более подробно алгоритм выявления дубликатов описан в статье [6]. При переименовании переменной или функции задача преобразования состоит в том, чтобы выявить все вхождения данной переменной в исходном коде, отличив их от других одноимённых переменных и

функций. Эту задачу мы будем называть проблемой *отождествления идентификаторов*. Отметим, что в случае локального переименования она решается за счёт использования семантической информации, поскольку в задачи семантического анализатора как раз входит различение разных одноимённых идентификаторов и связывание разных вхождений одного идентификатора между собой.

4. Осуществляется требуемое преобразование выделенных узлов синтаксического дерева. При этом от пользователя может потребоваться дополнительная информация, которую он предоставляет заранее (при запуске инструмента) или вводит в интерактивном режиме. Так, при выделении новой функции требуется задать её имя.

5. По измененным фрагментам дерева формируются соответствующие текстовые представления. Поскольку окончательное изменение исходного кода, по меньшей мере, требует одобрения пользователя, подобное представление генерируется в так называемом *нормальном формате* утилиты *diff* [7], и дальнейшее изменение исходного кода не представляет труда. Высокая точность генерации изменённого кода обуславливается тем, что в узлах синтаксического дерева, строящегося в инструменте Klocwork Insight, сохраняются ссылки на соответствующие им лексемы и позиции этих лексем. Более того, для узлов, построенных по исходному коду, полученному в результате макроподстановки, хранится текстовый вид изначального макровызова. Означенные лексемы и макровызовы по возможности используются при генерации кода [5].

3. Подход к проведению глобального рефакторинга

Задача проведения рефакторинга в рамках всего проекта, как правило, сводится к задаче проведения соответствующего локального рефакторинга. Последовательно осуществляя требуемое локальное преобразование для всех единиц компиляции, возможно осуществление глобального рефакторинга. В частности, можно проводить глобальное переименование, если удается корректно отождествлять одноимённые идентификаторы. Отметим, однако, что последовательное применение локального преобразования ко всем единицам компиляции требует очень больших временных затрат. В то же время, скажем, что при переименовании идентификатора в конкретной единице компиляции может случиться так, что вхождения интересующей нас функции или переменной отсутствуют. Поэтому встаёт задача минимизации числа команд локального рефакторинга, которые запускаются при проведении рефакторинга глобального.

3.1. Отождествление идентификаторов при глобальном переименовании

В соответствии со спецификациями языков C и C++, некоторые идентификаторы в программе могут быть доступны за пределами своей единицы компиляции [8]. Про такие идентификаторы говорят, что они обладают *внешним связыванием*, и, собственно говоря, именно их переименование должно осуществляться в рамках целого проекта. При этом из спецификаций следует, что два идентификатора, имеющих внешнее связывание, обозначают один и тот же объект, если:

1. их имена совпадают;
2. они являются членами одного и того же пространства имён или одного и того же класса;
3. в случае, если они соответствуют функциям, то эти функции являются одинаковыми с точки зрения перегрузки.

Практически же это означает, что для отождествления идентификаторов при глобальном переименовании достаточно сравнивать их полные квалифицированные имена и (в случае функций) сигнатуры. Под сигнатурой понимается текстовая запись типов всех параметров функции. В этой записи «раскрываются» типы, определённые с помощью конструкции *typedef*, что соответствует семантике перегрузки функций в C++.

3.2. Задача минимизации числа команд локального рефакторинга

Для случая глобального переименования эта задача может быть решена за счёт использования некоторой базы знаний, в которой будет храниться информация, какие идентификаторы в каких исходных файлах встречаются. Подобную базу знаний мы будем называть *индексом*. Далее мы подробно рассмотрим, как он может быть организован.

3.2.1 Индексация

Как было показано в предыдущем разделе, для однозначного задания идентификатора при глобальном переименовании требуется его полное квалифицированное имя и (в случае функций) сигнатура. Это значит, что индекс должен хранить соответствие между полными квалифицированными именами идентификаторов (а в случае функций еще и сигнатур) и файлами, в которых идентификаторы встречаются. Следовательно, построение индекса требует полного синтаксического и семантического анализа всего проекта, что для больших проектов может занимать десятки минут и даже часы.

С другой стороны, за гораздо меньшее время, с привлечением одного лишь препроцессора, может быть построен так называемый *предварительный индекс*, хранящий информацию о соответствии неквалифицированных имён

идентификаторов и содержащих их файлов. Очевидно, что он избыточен, однако во многих случаях позволяет существенно уменьшить количество команд запуска локального переименования, которые необходимо выполнить для осуществления переименования глобального. По мере того, как для тех или иных файлов будет проводиться анализ (например, при автоматическом рефакторинге), предварительная индексная информация будет заменяться точной.

Отметим важный технический момент, касающийся реализации подобного индекса. На первый взгляд, напрашивается решение, при котором в индексе будут храниться соответствия вида *<имя основного исходного файла, идентификатор>*, где *имя исходного файла* - это имя файла, который соответствует единице компиляции. Поскольку назначение индекса в том, чтобы определять, для каких единиц компиляции надо проводить рефакторинг, то нас не интересует, содержится ли идентификатор непосредственно в исходном файле или в каком-то из подключаемых заголовочных. Однако практика показывает, что такое решение неэффективно. Заголовочный файл может содержать большое число идентификаторов и подключаться (непосредственно или транзитивно) в большое число исходных файлов. Информация обо всех идентификаторах из данного заголовочного файла будет продублирована для каждого из подключающих его файлов исходных. Более того, если идентификатор присутствует лишь в некотором заголовочном файле и не встречается в исходных файлах, то в случае его переименования придётся переанализировать все исходные файлы, подключающие данный заголовочный файл, что, очевидно, нерационально. Поэтому более предпочтительным оказывается подход, при котором идентификаторам ставятся в соответствие файлы, которые физически их содержат, и вводится дополнительная таблица, хранящая информацию о том, какие заголовочные файлы подключаются в той или иной единице компиляции. Такую таблицу мы будем называть *таблицей файловых зависимостей*.

При работе с индексом следует придерживаться принципа *инкрементальности*. Если какие-то файлы в проекте были изменены, индекс перестраивается только для тех единиц компиляции, которые этим изменением затронуты. Если изменению подверглись заголовочные файлы, соответствующие им единицы компиляции могут быть определены по таблице файловых зависимостей. Между сеансами работы с проектом построенный индекс сохраняется на диске. Если с момента построения индекса для некоторого файла в него были внесены изменения, то это будет выявлено путем сравнения дат, и индекс с таблицей зависимостей будут инкрементально перестроены.

3.2.2 Алгоритм поиска минимального числа команд локального рефакторинга

При построении множества подлежащих рефакторингу единиц компиляции следует уделять внимание тому, чтобы это множество не было неоправданно велико. В системе *Klocwork Insight* используется следующий алгоритм минимизации этого множества:

1. Все исходные файлы, непосредственно содержащие данный идентификатор, добавляются в множество S^B.
2. Для каждого заголовочного файла h_i (где i принимает значения 1, 2, ..., n), непосредственно содержащего данный идентификатор, вычисляется множество S^i исходных файлов, подключающих заголовочный файл h_i. Очевидно, что для проведения рефакторинга в файле h_i достаточно провести рефакторинг на единице компиляции соответствующей любому исходному файлу из S^i. Если пересечение S^i и S^B непусто, то заголовочный файл h_i исключается из дальнейшего рассмотрения.
3. Строится граф G, причем:
   - 3.1 в граф добавляются вершины vh_i, соответствующие найденным заголовочным файлам h_i;
   - 3.2 для каждого найденного заголовочного файла h_i в граф добавляются узлы vs_i соответствующие исходным файлам из S^i;
   - 3.3 между узлами vh_i и vs_i проставляются дуги, соответствующие подключению исходным файлом s_i заголовочного файла h_i.
4. В графе G ищется вершина vs_n с наибольшим количеством дуг.
5. Исходный файл s_n, соответствующий вершине vs_n добавляется в множество S^I.
6. Из графа G удаляются все vh_j, связанные дугами с vs_n.
7. Из графа G удаляется вершина vs_i.
8. Если в графе G остались вершины vh_i, то переходим к пункту 4.
9. Объединение множеств S^B и S^I даст минимальное множество единиц компиляции, для которых необходимо запустить локальный рефакторинг

4. Заключение

Автоматический рефакторинг, реализованный в системе *Klocwork Insight*, обладает высокой точностью и позволяет сохранять пользовательский стиль в преобразованных фрагментах кода. Однако, до недавнего времени существенным его недостатком была возможность применения только в

рамках одной единицы компиляции, в то время как для ряда рефакторингов возможность глобального преобразования является очень важной. В данной статье мы рассмотрели некоторые проблемы, возникающие при реализации глобального рефакторинга «Переименование» на основе соответствующего локального рефакторинга, и описали пути их решения. В настоящее время авторы продолжают активные исследования в данной области. Изучаются возможности поддержки глобальных преобразований для других видов рефакторинга, исследуются возможности повышения точности преобразований, в частности, за счёт более корректной обработки препроцессорных директив.

Список литературы

[1] Мартин Фаулер. Рефакторинг. Улучшение существующего кода.
[2] http://www.klocwork.com/products/insight/refactoring
[3] http://www.eclipse.org/cdt
[4] https://www.devexpress.com/Products/CodeRush
[5] Н. Л. Луговской. Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. Т. 23. М., ИСП РАН, 2012
[6] Н. Г. Зельцер. Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. т. 25 М., ИСП РАН, 2013
[7] http://www.opennet.ru/docs/RUS/diff/diff-3.html
[8] Working Draft, Standard for Programming Language C++, http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

Refactoring on the whole project

N. L. Lugovskoy, S. V. Syromyatnikov, I. E. Bronstein
lugovskoy@ispras.ru, syrom@ispras.ru, ibronstein@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Refactoring is one of the most popular and successful techniques for improving source code. It is an integral part of agile development methods. However, C/C++ developers still lack effective tools for source code automatic refactoring. It is obviously a serious limitation when refactoring is applicable only to a single translation unit. In many cases applying refactoring to the only source file with its headers may cause errors when linking the whole software project. This article describes in details how whole project (global) refactoring can be implemented on basis of an existing single unit refactoring tool. Special attention is paid here to refactoring «Rename» as it is one of the most widely used transformations applicable to the whole project. Two important problems specific for global renaming are highlighted. First, the article describes a way of matching two different identifiers used in different translation units. Second, a problem of minimizing the number of local renamings required for the given global renaming is also discussed. It is displayed that using a database containing information about identifiers used in a project and positions of those identifiers in project sources can significantly increase the speed of global renaming.

Keywords: refactoring; rename; global scope; static analysis

References

[1]. M. Fowler., K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring. Improve the design of exesting code. Addison-Wesley, 2001
[2]. http://www.klocwork.com/products/insight/refactoring
[3]. http://www.eclipse.org/cdt
[4]. https://www.devexpress.com/Products/CodeRush
[5]. N. L. Lugovskoj. Podkhod dlya provedeniya refaktoringa «Vydelenie funktsii» v instrumente Klocwork Insight [“Extract Function” Refactoring in Klocwork Insight Toolkit]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2012, vol. 23, pp. 107-132 (in Russian).
[6]. N. G. Zeltser. Poisk povtoryayushhikhsya fragmentov iskhodnogo koda pri avtomaticheskom refaktoringe [Automatic clone detection for refactoring]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2013, vol. 25, pp. 39-50 (in Russian).
[7]. http://www.opennet.ru/docs/RUS/diff/diff-3.html
[8]. Working Draft, Standard for Programming Language C++, http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2011/n3242.pdf