Перспективы интеграции методов верификации программного обеспечения

В. В. Кулямин
kuliamin@ispras.ru

Аннотация. В статье предлагается подход к построению расширяемой среды верификации программных систем, которая, по мнению автора, поможет решить проблемы практической применимости современных строгих методов верификации к практически значимым программам, сложность которых все время растет. Она же может стать аналогом испытательного стенда для апробации и отладки большого числа новых предлагаемых техник формальных верификации и статического анализа на разнообразном промышленном программном обеспечении.

1. Введение

Прогресс технологий разработки программного обеспечения (ПО) в последние десятилетия значительно увеличил производительность программистов в терминах количества кода, создаваемого ими в единицу времени. Это проявляется, в частности, в увеличении размеров наиболее сложных программных систем, разрабатываемых сейчас, до десятков миллионов строк кода [1,2]. Однако качество программ при этом заметным образом не изменилось — среднее количество ошибок на тысячу строк кода, еще не прошедшего тестирование, по-прежнему колеблется в пределах 10-50 [3]. Таким образом, совершенствование методов разработки ПО, давая возможность создавать все более сложные системы, необходимые современной экономике, науке и государственным организациям, парадоксальным образом лишь увеличивает количество дефектов в них и связанные с этим риски.

Борьба с дефектами и ошибками в программном обеспечении ведется при помощи его *верификации*. В ходе ее выполнения проверяется взаимная согласованность всех артефактов разработки — проектной и пользовательской документации, исходного кода, конфигураций развертывания, — а также их соответствие требованиям к данной системе и нормам применимых к ней стандартов. Методы верификации ПО также активно развиваются, однако их прогресс менее заметен. Поэтому предельная сложность ПО, которое можно сделать надежно и корректно работающим, существенно меньше сложности систем, востребованных современным обществом.

Различные методы проведения верификации ПО можно (больше по историческим, чем содержательным причинам) разделить [4] на *формальные методы*, использующие строгий анализ математических моделей проверяемых артефактов и требуемых свойств; *методы статического анализа*, в ходе которых возможные ошибки ищутся без исполнения проверяемого ПО; *методы динамического анализа*, проводящие проверку реального поведения проверяемой системы в рамках некоторых сценариев ее работы; и *экспертизу (review, inspection)*, выполняемую людьми на основе их знаний и опыта.

Все эти методы имеют разные достоинства и недостатки, различные области применимости, и эффективность их применения сильно отличается в разных контекстах. Но полноценная верификация крупномасштабных сложных систем невозможна без совместного использования всех этих методов, поскольку только их сочетание позволяет преодолеть недостатки каждого. При этом на каждом уровне рассмотрения системы и для каждого вида компонентов хотелось бы выбирать самый эффективный метод, дающий наиболее достоверный вклад в оценку качества системы в целом и требующий минимальных затрат. К сожалению, пока не существует общего подхода, позволяющего сопоставлять и сравнивать различные методы верификации и их сочетания в различных контекстах при применении к реальным программным системам.

Чтобы справиться со все возрастающей сложностью реальных систем, исследователями за последние 20-30 лет создано огромное количество разнообразных методов и техник верификации [4], особенно в рамках статического анализа и формальных методов. Но для их эффективного использования чаще всего нужно быть специалистом в соответствующей области. Многие из таких работ ограничиваются формулировкой идей и алгоритмов, несколько реже создаются прототипные реализации, цель которых — на двух-трех примерах продемонстрировать, что предложенная техника работает. Эти прототипы невозможно использовать для индустриальной разработки ПО, в рамках которой инструменты должны быть работоспособны и эффективны в очень широком контексте. У исследователей же почти никогда нет ресурсов и времени разрабатывать промышленно применимые инструменты.

В тех очень редких случаях, когда удается все же сделать пригодный к использованию на практике инструмент, он объединяет десяток-два разнообразных техник и способен решать две-три задачи верификации. Однако в процессе промышленной разработки ПО таких задач несколько десятков, а большинству организаций удается успешно внедрить и начать активно использовать лишь два-три таких инструмента.

Другой проблемой является растущая сложность создания и апробации новых техник верификации. Все необходимое для их работы окружение — инструменты анализа исходного кода, описания формальных моделей,

библиотеки для работы с внутренним представлением моделей и кода, инструменты, реализующие различные виды анализа кода и моделей, средства получения отчетов — невозможно разработать заново. Исследователю для проверки работоспособности его идеи приходится на скорую руку собирать это окружение из разнородных компонентов и библиотек, которые можно найти в свободном доступе. В лучшем случае удается создать прототип, который способен справиться с парой нужных примеров. Но таким способом невозможно создать среду, в рамках которой можно было бы проанализировать работоспособность и эффективность новой идеи в широком множестве разнообразных ситуаций, на разных видах приложений и требований к ним. Поэтому большинство новых идей применяются лишь в «тепличных условиях», а эффекты от их применения в широком контексте остаются неясными и непредсказуемыми.

Решением для упомянутых проблем могла бы стать *унифицированная расширяемая среда верификации программных систем*, предоставляющая общее окружение для решения задач верификации и библиотеки готовых компонентов, реализующих типовые техники. Такая среда могла бы существенно упростить интеграцию модулей, реализующих различные техники верификации, за счет унифицированных интерфейсов ее расширения.

Исследователи могли бы использовать ее для значительного снижения затрат на апробацию новых методов и анализ их работоспособности в разнообразных ситуациях. Промышленные разработчики — для интеграции нужного им набора техник в рамках единого инструмента и эффективного внедрения их в практическое использование.

Подтверждением работоспособности и эффективности интеграции различных методов верификации ПО в разнообразных ситуациях являются многочисленные *синтетические методы верификации*.

2. Синтетические методы верификации ПО

Синтетические методы верификации используют техники различных видов по приведенной выше классификации, а также комбинируют идеи различных подходов для получения большей эффективности верификации в терминах затрат на ее проведение и достоверности получаемых результатов.

На данный момент такие методы относятся к одной из следующих групп.

- Статический анализ предполагает построение некоторых моделей кода проверяемой системы, чаще всего, в виде размеченных графов потоков управления и данных, и анализ свойств этих моделей, например, поиск ошибок определенного рода по соответствующим им шаблонам в потоках данных. Сейчас все чаще используются специфические виды статического анализа, в рамках которых находят применения формальные модели и специализированные инструменты разрешения ограничений.

  - *Расширенный статический анализ (extended static checking)* [5,6] проверяет соответствие кода ПО требованиям, обычно записываемым тоже в коде в виде комментариев к его отдельным элементам (процедурам, типам данных и методам классов). При этом на основе результатов анализа кода автоматически строятся формальные модели его поведения, выполнение требований для которых проверяется чаще всего с помощью дедуктивного анализа и специализированных решателей (solvers).
  - *Статический анализ на базе автоматической абстракции* [7-10]. В рамках такого подхода на основе результатов статического анализа кода автоматически строятся более абстрактные, а потому более простые модели работы проверяемого ПО, которые затем подвергаются проверке на выполнение определенных свойств с помощью инструментов проверки моделей или решателей. Обычно проверяемые свойства фиксированы для данного инструмента или формулируются в конфигурационном файле. При нарушении требования в модели инструменты этого типа пытаются построить соответствующий сценарий работы кода. Если это не получается из-за упрощений, сделанных при построении модели, определяются элементы кода, препятствующие выполнению такого сценария, и в модель вносятся уточнения, более аккуратно описывающие работу именно этих элементов, после уточненная модель снова проверяется на выполнение заданного свойства. В итоге инструмент либо подтверждает выполнение требований, либо находит контрпример, либо завершает работу по истечении некоторого времени или из-за исчерпания ресурсов, не приходя к определенным выводам.

- *Синтетическое структурное тестирование* [11-16] при котором после первого случайно выбранного теста остальные тесты генерируются автоматически так, чтобы обеспечить покрытие еще не покрытых ранее элементов кода. Для выбора подходящих тестовых данных используются решатели, учитывающие символическую информацию о ранее выполненных тестах (ограничения на данные, отделяющие прошедшие тесты от еще не покрытого кода), а для построения нужных последовательностей воздействий — случайная генерация, направляемая как этой же символической информацией, так и некоторыми эвристическими абстракциями, уменьшающими пространство состояний проверяемой системы.*Тестирование на основе моделей (model based testing)* [17-19] сочетает разработку формальных моделей требований к проверяемому ПО и построение

тестов на базе этих моделей. Структура модели при этом служит основой для критерия полноты тестирования, а ограничения модели на корректные результаты работы ПО используются в качестве тестовых оракулов, оценивающих правильность поведения ПО в ходе тестирования.
В рамках последних двух подходов (или отдельно от них) применяются специфические техники построения тестов, сами по себе сочетающие разные методы верификации.

- *Построение тестов с помощью разрешения ограничений* [20-22]. Часто при разработке тестов на основе критериев полноты тестирования формулируются так называемые *цели тестирования* (test objectives), представляющие собой специфические ситуации, в которых необходимо проверить поведение тестируемой системы для достижения необходимой уверенности в ее корректной работе. Цель тестирования формулируется как набор ограничений на проходимые во время теста состояния системы и данные выполняемых воздействий. Для построения теста, достигающего такую цель, можно использовать специализированные решатели (solvers). Такой решатель либо автоматически находит необходимые данные и последовательность вызовов операций как решение заданной системы ограничений, либо показывает, что эта система неразрешима, т.е. заданная цель тестирования недостижима и строить нацеленные на нее тесты не имеет смысла.
- *Построение тестов как контрпримеров с помощью инструментов проверки моделей* [23-26]. Другой способ построения тестов — сформулировать отрицание ограничений, задающих цель тестирования, как свойство, которое можно проверить или опровергнуть с помощью инструмента проверки моделей. Если это свойство подтверждается, значит, цель тестирования недостижима, если же оно опровергается, то инструмент строит контрпример, являющийся в данном случае необходимым тестом.

- *Мониторинг формальных свойств (runtime verification, passive testing)* [27-30] тоже использует формальные модели требований для оценки правильности поведения проверяемой системы, но только в ходе ее обычной работы, без использования специально построенных тестов.

Как видно, все синтетические методы так или иначе пытаются соединить достоинства различных подходов к верификации, купируя их недостатки. В

настоящее время достигнуты значительные успехи в разработке таких методов и внедрении их в практику промышленной разработки ПО, например, в следующих случаях.

- Многочисленные проекты NASA по разработке ПО управления для космических спутников, челноков и специализированных исследовательских аппаратов, проводимые с использованием инструментов проверки моделей, генерации тестов на их основе и мониторинга [31-33]. Из используемых в этих проектах инструментах наиболее известны инструменты проверки моделей Spin [34,35], генератор тестов T-VEC [36,37] и инструмент символического выполнения Java PathFinder [38,39], используемый для проверки свойств Java программ, их мониторинга и тестирования.

- Создание и использование в Microsoft инструмента Static Driver Verifier, использующего статический анализ с автоматической абстракций для проверки корректности работы драйверов Windows [40]. Сначала в проекте использовался инструмент проверки моделей SLAM, который затем был значительно доработан и дополнен возможностями анализа произвольного кода на языке C и автоматической абстракции, направляемой контрпримерами [41,42].

- Внутренний проект Microsoft по проведению формальной спецификации и генерации тестовых наборов для разнообразных клиент-серверных протоколов, используемых в продуктах этой компании [43]. В рамках этого проекта используется, в основном, инструмент SpecExplorer [44], разработанный в Microsoft Research, а объем работ по анализу и формализации документации на протоколы оценивается в несколько десятков человеко-лет.

- Проводившиеся и идущие в настоящее время в ИСП РАН проекты по созданию тестов на основе формальных моделей базовых библиотек операционных систем, телекоммуникационных протоколов семейства IPv6, оптимизирующих блоков компиляторов [45-47], использующие семейство инструментов тестирования на основе моделей UniTESK [48].

- Использование формальных методов верификации и инструментов расширенного статического анализа при создании систем авионики в Airbus и Boeing [10,49,50]. В частности, в Airbus использовался инструмент статического анализа на основе формальных моделей ASTREE [10].

- Использование формальных методов, тестирования на основе моделей и средств мониторинга при разработке ПО для смарт-карт [51,52].

Все эти примеры подтверждают эффективность интеграции различных верификационных методов на практике. Тем не менее, несмотря на достигнутые успехи, каждый из имеющихся синтетических подходов использует лишь часть имеющегося потенциала и не предоставляет единой среды интеграции для всего многообразия различных техник верификации ПО.

3. Подход к построению расширяемой среды верификации ПО

Проблемы возрастающей сложности при создании и апробации новых методов верификации ПО и необходимость создания расширяемой среды, позволяющей интегрировать различные техники и инструменты, уже обсуждались различными авторами (см., например, [53]). Однако в доступной литературе пока не было представлено систематичного подхода к построению подобной среды.

Чтобы стать реализуемым на практике, такой подход должен предлагать адекватные решения для нескольких методологических и организационных проблем, которые обсуждаются ниже.

3.1. Анализ требований

Никакая верификация немыслима без предварительной четкой формулировки проверяемых требований, и на практике почти всегда любая деятельность по верификации предваряется анализом требований к проверяемой системе и (обычно, частичной) их формализацией.

Однако методически единого подхода к вопросам анализа и представления требований не существует, и, скорее всего, не будет выработано в течение достаточно долгого времени. Как в этом случае можно надеяться построить единую среду верификации, интегрирующие разные подходы, в том числе и использующие различные методики анализа требований?

Для этого предлагается оставить проблематику анализа требований за рамками обсуждаемой среды и определить четкий интерфейс между ней и деятельностью по выделению требований. Для обоснования адекватности проводимой верификации необходимо, чтобы каждая часть используемых моделей и каждый элемент отчетов о найденных проблемах могли быть соотнесены с каким-то элементом исходных требований. Поэтому, отвлекаясь от проблем формализации, установления взаимосвязей между требованиями, обеспечения их адекватности и полноты, можно считать требования лишь набором некоторых объектов с уникальными идентификаторами, позволяющими привязывать к ним элементы моделей, тесты, проводимые проверки и обнаруживаемые дефекты. Какова природа этих объектов — являются ли они текстами, формулами, изображениями, схемами и т.п. — при этом не важно.

Еще один важный аспект — место экспертизы среди поддерживаемых рассматриваемой средой подходов к верификации. Экспертиза применима к любым свойствам ПО и любым артефактам, хотя для разных целей используются разные ее виды. Она позволяет выявлять все виды ошибок, причем делать это на ранних этапах, тем самым минимизируя время существования дефекта в рамках жизненного цикла ПО и ресурсы, требующиеся на его устранение. Эмпирические исследования показывают, что эффективность экспертиз, измеряемая как отношение количества обнаруживаемых дефектов к затрачиваемым на это ресурсам, выше, чем для других методов верификации. Согласно различным отчетам от 50% до 90% всех зафиксированных в жизненном цикле ПО ошибок может быть обнаружено с помощью экспертиз [54-56].

В то же время проведение экспертизы не может быть автоматизировано и всегда требует привлечения людей, а ее эффективность существенно зависит от их опыта и мотивации, организации процесса разработки и профессионального взаимодействия между его участниками. Это накладывает серьезные ограничения на распределение ресурсов в проектах и может приводить к конфликтам, если мало внимания уделяется организационным аспектам проведения экспертиз.

В рамках рассматриваемой среды предлагается по максимуму использовать формализованные представления для всех артефактов разработки, с тем чтобы к ним можно было применить автоматизированный анализ того или иного рода. Использовать экспертизу нужно в ходе анализа требований и их формализации, что позволит наиболее выгодным образом сочетать достоинства различных методов верификации. Экспертиза с ее включением людей и их возможности находить решения в неформализованных, неясных ситуациях, наиболее эффективна именно во время определения и уточнения требований, где не работают все остальные методы. При анализе же формализованных артефактов более эффективным представляется применять автоматизированный анализ.

3.2. Поддержка различных языков и нотаций

Очень многие инструменты верификации ориентируются на определенные языки представления моделей и требований. При интеграции различных методов сразу же встанет вопрос о том, какие языки использовать вообще, и поддержку каких из них стоит реализовать в первую очередь.

Опыт, полученный на основе большого числа проектов по верификации промышленного ПО [18,19,43,44,46,48], позволяет утверждать, что использование в рамках технологий и инструментов языков, которые как можно меньше отличаются от привычных обычным разработчикам, существенно облегчает их внедрение и использование в промышленности. Поэтому в рамках среды в первую очередь необходимо использовать такие формы представления моделей, которые были бы минимально необходимыми

расширениями широко распространенных языков программирования. В дальнейшем можно добавлять поддержку для наиболее известных языков формальных спецификаций.

Поддержка различных языков и нотаций должна быть организована на уровне некоторого общего промежуточного представления языковых конструкций, используемого всеми инструментами анализа, но не пользователями непосредственно. Разработка такого промежуточного представления, применимого для многих разных языков, является нетривиальной задачей. Как показывает опыт, создать адекватное общее представление программ для сильно отличающихся языков (например, для Java и Пролога), практически невозможно. Поэтому построение интерфейса для используемого средой промежуточного представления будет постепенным, опирающимся на накапливаемый опыт работы с различными языками. Сам набор понятий, на базе которого можно создать такое представление, пока не выработан, и строить его придется уже в ходе создания описываемой среды верификации.

При наличии такой возможности стоит использовать уже имеющиеся стандартные или широко используемые библиотеки для работы с промежуточным представлением для ряда языков. Например, для C и C++ стандартом де-факто постепенно становится промежуточное представление, используемое в рамках компилятора GCC [57]. Значительным преимуществом использования результатов подобных проектов является гарантированные их поддержка и развитие в будущем в течение обозримого времени.

3.3. Архитектурная основа среды верификации

Архитектура рассматриваемой среды верификации — набор ее основных компонентов, их внешних интерфейсов и правил взаимодействия, а также правил добавления новых компонентов — в значительной степени определяет одно из важнейших свойств этой среды — ее расширяемость. С другой стороны, решения, касающиеся базовых принципов построения среды могут влиять на возможности ее интеграции с другими инструментами разработки ПО.

Прежде всего, для облегчения ее использования в промышленной разработке ПО, такая среда должна быть встроена в одну из широко используемых сред разработки, таких как Eclipse или Microsoft Visual Studio. Eclipse [58,59] является наиболее подходящей средой интеграции для первых версий, поскольку обладает огромным набором модулей расширения, в том числе являющихся инструментами верификации и модулями поддержки различных языков программирования. Кроме того, процессом создания таких модулей хорошо документирован.

Кроме внешнего окружения, в рамках которой должна работать среда верификации, необходимо также определить ее каркас — некоторый базовый набор компонентов, реализующих основной набор функций и поддерживающие основные потоки данных внутри системы. К этому каркасу

будут добавляться другие компоненты, поддерживающие вспомогательные и менее значимые функции.

Предлагается использовать в качестве основы для построения среды верификации архитектурный каркас инструментов тестирования на основе моделей. Это решение вызвано тем обстоятельством, что такое тестирование является одним из самых сложно организованных процессов верификации — в его ходе обычно необходимо сделать следующее.

- Определить модель поведения тестируемой системы, формализующую требования к этому поведению.
- Проанализировать структуру модели для выбора критериев покрытия и отдельных целей тестирования, и определить эти критерии и цели.
- Построить среду выполнения тестов, включающую средства мониторинга и тестовые оракулы — программные компоненты, определяющие соответствие или несоответствие наблюдаемого поведения системы и модели. Обычно такая среда состоит из библиотеки поддержки выполнения тестов, набора тестовых оракулов для всех проверяемых компонентов и набора адаптеров, связывающих эти компоненты с их оракулами. Оракулы в большинстве случаев генерируются автоматически из модели поведения системы.
- Построить, автоматически или с привлечением человека, набор тестовых сценариев, определяющих последовательности вызова различных операций тестируемой системы или посылки ей сообщений или сигналов и данные, передаваемые в качестве параметров операций и сообщений.
- Выполнить тестовые сценарии, протоколируя всю информацию, касающуюся соответствия наблюдаемого поведения системы и ее модели, а также покрытых во время тестирования ситуаций.
- Провести анализ результатов тестов, в ходе которого выявляются и анализируются ошибки в системе или ее модели (проявляющиеся как несоответствия между ожидаемым и реальным поведением), а также анализируется достигнутое тестовое покрытие и принимается решение либо о создании дополнительных тестов, либо об окончании их разработки.

Чтобы в архитектурный каркас тестирования на основе моделей вложить другие синтетические методы верификации, необходимо лишь добавить модули для анализа исходного кода проверяемых компонентов — все остальные необходимые модули в нем фактически уже имеются (см. также [60]).

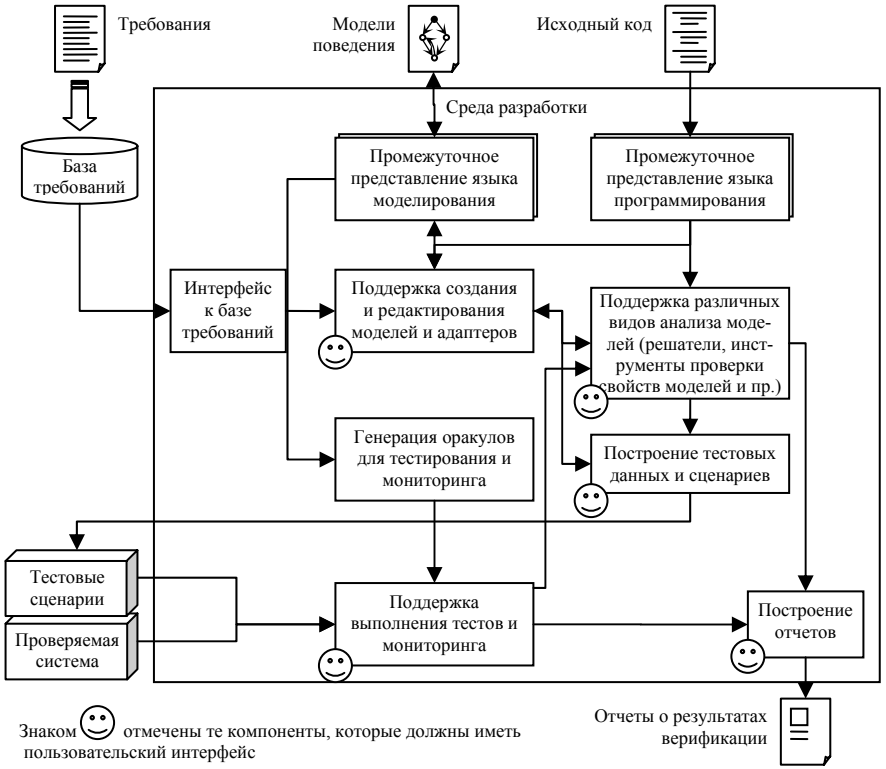

Рис. 1. Предварительная архитектура расширяемой среды верификации ПО.

Предварительный вариант архитектуры унифицированной расширяемой среды верификации ПО изображен на рис. 1. На нем присутствуют только наиболее крупные компоненты. При более детальной проработке архитектуры может потребоваться их разбиение на более мелкие и добавление других модулей, решающих вспомогательные задачи.

В рамках такой архитектуры можно проводить как тестирование на основе моделей и верификационный мониторинг (без построения и использования тестов), так и расширенный статический анализ (используя только различные виды анализа исходного кода системы и моделей требований к ней) или синтетическое структурное тестирование (используя при построении тестов информацию о структуре исходного кода).

3.4. Организация разработки среды верификации

Для создания описанной среды верификации потребуется затратить значительные ресурсы, даже если удастся использовать имеющиеся компоненты, реализующие различные виды анализа, алгоритмы построения тестов или синтаксический разбор текстов на определенных языках программирования. Необходимые трудозатраты делают ее разработку силами небольшой группы практически нереальной.

Поэтому разработка и развитие такой среды могут быть организованы как открытый проект в Интернет с возможностью включения в него любых участников, согласных следовать предложенным архитектурным решениям и другим правилам проекта.

Для начала такого проекта важно подготовить общий каркас среды и реализацию некоторой значимой части ее функциональности. В качестве первого варианта можно рассматривать расширение системы модульного тестирования TestNG [61,62] с помощью средств построения тестов на основе моделей. TestNG — это популярная среда с открытым кодом, поддерживающая разработку модульных и интеграционных тестов для Java приложений и позволяющая гибко конфигурировать выполнение полученных тестовых наборов. Расширение ее возможностями тестирования на основе моделей и хотя бы одним-двумя видами анализа моделей и кода, позволяющими, например, реализовать синтетическое структурное тестирование в ряде ситуаций, позволит наглядно продемонстрировать интеграционные возможности предлагаемого подхода.

4. Заключение

В данной статье предложен подход к интеграции различных методов верификации ПО. Целью его является существенное повышение сложности программных систем, для которых проведение верификации с помощью строгих методов, использующих формальные модели в явном или скрытом виде, сможет давать практически значимые результаты при приемлемых затратах.

Предлагаемый подход основан на объединении нескольких успешно применяемых на практике синтетических методов верификации (расширенный статический анализ, синтетическое структурное тестирование, тестирования на основе моделей и мониторинг формальных свойств) в рамках единой расширяемой среды верификации ПО. В качестве базовой архитектуры для такой среды предложено использовать хорошо зарекомендовавшую себя архитектуру средств тестирования на основе моделей [48], расширенную дополнительными компонентами для анализа исходного кода проверяемых компонентов и для различных видов анализа моделей, в том числе разнообразными решателями. Тестирование на основе

моделей выбрано основой предложенной архитектуры, поскольку оно является самым сложным видом верификации из объединяемых методов.

Представлен также ряд методических и технических решений, который, по мнению автора, позволит сделать создание описываемой среды верификации практически выполнимым, а кроме того, облегчит ее использование для решения практических задач верификации промышленного ПО.

Еще одной сферой применения такой среды может стать апробация и отладка многочисленных новых техник верификации и анализа свойств ПО, нацеленного на его верификацию, на практически значимых системах разных классов.

Литература

[1] V. Maraia. The Build Master: Microsoft's Software Configuration Management Best Practices. Addison-Wesley Professional, 2005.

[2] G. Robles. Debian Counting. http://libresoft.dat.escet.urjc.es/debian-counting/.

[3] С. Макконнелл. Совершенный код. М.: Русская редакция, 2005.

[4] В. В. Кулямин. Методы верификации программного обеспечения. Всероссийский конкурс обзорно-аналитических статей по приоритетному направлению "Информационно-телекоммуникационные системы", 2008. http://window.edu.ru/window/library?p_rid=56168.

[5] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe. Extended static checking. Technical Report SRC-RR-159, Digital Equipment Corporation, Systems Research Center, 1998.

[6] D. R. Cok, J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Proc. of International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04), LNCS 3362:108-128, Springer-Verlag, January 2005.

[7] P. Emanuelsson, U. Nilsson. A Comparative Study of Industrial Static Analysis Tools. Technical Report 2008:3, Linkoping University, 2008. http://www.ep.liu.se/ea/trcis/2008/003/trcis08003.pdf.

[8] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Software Verification with Blast. Proc. of 10-th SPIN Workshop on Model Checking Software (SPIN 2003), LNCS 2648:235-239, Springer-Verlag, 2003.

[9] T. Ball, S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Proc. of Model Checking of Software, LNCS 2057:103-122, Springer, 2001.

[10] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. T. Mogensen, D. A. Schmidt, I. H. Sudborough, eds. The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS 2566:85-108, Springer-Verlag 2002.

[11] Y. Smaragdakis, C. Csallner. Combining Static and Dynamic Reasoning for Bug Detection. Proc. of TAP 2007, LNCS 4454:1-16, Springer, 2007.

[12] K. Sen, G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. Proc. of Computer Aided Verification, pp.419-423, August 2006.

[13] T. Xie, D. Marinov, W. Schulte, D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. Proc. of 11-th International

Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, UK, pp. 365-381, April 2005.

[14] N. Tillmann, W. Schulte. Parameterized Unit Tests with Unit Meister. ACM SIGSOFT Software Engineering Notes, 30(5):241-244, September 2005.

[15] 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.

[16] P. Godefroid. Compositional dynamic test generation. Proc. of 34-th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PLOP 2007), pp. 47-54, 2007.

[17] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner, eds. Model Based Testing of Reactive Systems. LNCS 3472, Springer, 2005.

[18] M. Utting, B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, 2007.

[19] J. Jacky, M. Veanes, C. Campbell, W. Schulte. Model-Based Software Testing and Analysis with C#. Cambridge University Press, 2007.

[20] B. Korel. Automated Test Data Generation. IEEE Trans. on Software Engineering, 16(8):870-879, 1990.

[21] R. DeMillo, A. Offutt. Constraint-based automatic test data generation. IEEE Trans. on Software Engineering, 17(9):900-910, 1991.

[22] A. Gotlieb, B. Botella, M. Rueher. Automatic test data generation using constraint solving techniques. ACM SIGSOFT Software Engineering Notes, 23(2):53-62, 1998.

[23] A. Gargantini, C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. Proc. of Joint 7-th European Software Engineering Conference and 7-th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE99), ACM Press, September 1999, pp. 146-162.

[24] H. S. Hong, I. Lee, O. Sokolsky, S. D. Cha. Automatic Test Generation from Statecharts Using Model Checking. Technical Report MS-CIS-01-07, Feb 2001.

[25] G. Hamon, L. de Moura, J. Rushby. Generating Efficient Test Sets with a Model Checker. Proc. of the 2-nd Software Engineering and Formal Methods International Conference, p. 261-270, 2004.

[26] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, R. Majumdar. Generating tests from counterexamples. Proc. of 26-th International Conference on Software Engineering (ICSE), p. 326-335, 2004.

[27] I. Lee, S. Kannan, M. Kim, O. Sokolsky, M. Viswanathan. Runtime Assurance Based On Formal Specifications. Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'1999, pp. 279-287, 1999.

[28] Y. Cheon, G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). Proc. of International Conference on Software Engineering Research and Practice (SERP'02), pp. 322-328, CSREA Press, June 2002.

[29] A. Cavalli, C. Gervy, S. Prokopenko. New approaches for passive testing using an Extended Finite State Machine Specification. Information and Software Technology, 45(12):837-852, Elsevier, September 2003.

[30] D. Drusinsky. Modeling and Verification Using UML Statecharts. Newnes, 2006.

[31] M. R. Blackburn, R. D. Busser, A. M. Nauman. Interface-Driven, Model-Based Test Automation. CrossTalk, The Journal of Defense Software Engineering, May 2003.

[32] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, R. Washington. Combining test case

generation and runtime verification. Theoretical Computer Science, 336(2-3):209-234, May 2005.
[33] G. Brat, K. Havelund, S. Park, W. Visser. Model Checking Programs. Proc. of 15-th IEEE International Conference on Automated Software Engineering, Grenoble, France, pp. 3-11, September 2000.
[34] G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003.
[35] http://spinroot.com/.
[36] M. Blackburn, R. D. Busser, J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. Proc. of 12-th Annual Conference on Computer Assurance, June 1997, pp. 54-67.
[37] http://www.t-vec.com/.
[38] G. Brat, W. Visser, K. Havelund, S. Park. Java PathFinder — second generation of a Java model checker. Proc. of Workshop on Advances in Verification, Chicago, Illinois, July 2000.
[39] http://javapathfinder.sourceforge.net/.
[40] http://www.microsoft.com/whdc/devtools/tools/SDV.mspx.
[41] T. Ball, S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Proc. of Model Checking of Software, LNCS 2057:103-122, Springer, 2001.
[42] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, A. Ustuner. Thorough Static Analysis of Device Drivers. ACM SIGOPS Operating Systems Review 40(4):73-85, 2006.
[43] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, F. L. Wurden. Model-Based Quality Assurance of Windows Protocol Documentation. Proc. of 1-st International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 2008, pp. 502-506.
[44] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Formal Methods and Testing, LNCS 4949:39-76, Springer Verlag, 2008.
[45] V. Kuliamin, A. Petrenko, N. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. In M. Malek, E. Nett, N. Suri, eds. Service Availability. LNCS 3694, pp. 68-83, Springer-Verlag, 2005.
[46] 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, LNCS 4378:459-469, Springer-Verlag, 2006.
[47] С. В. Зеленов, С. А. Зеленова, А. С. Косачев, А. К. Петренко. Генерация тестов для компиляторов и других текстовых процессоров. Программирование, 29(2):59–69, 2003.
[48] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
[49] J. Souyris, D. Delmas. Experimental Assessment of ASTRÉE on Safety-Critical Avionics Software. Proc. of Int. Conf. on Computer Safety, Reliability, and Security, SAFECOMP 2007, F. Saglietti, N. Oster, eds., Nuremberg, Germany, September 2007, LNCS 4680:479-490, Springer, 2007.
[50] P. Manolios, G. Subramanian, D. Vroon. Automating component-based system assembly. Proc. of ISSTA 2007, London, UK, 2007, pp. 61-72.
[51] E. Poll, J. van den Berg, B. Jacobs. Specification of the JavaCard API in JML. In Proc. of CARDIS'00. Kluwer Academic Publishers, 2000.

[52] F. Bouquet, B. Legeard. Reification of executable test scripts in formal specification-based test generation: The Java card transaction mechanism case study. In Proc. of the International Symposium of Formal Methods Europe, pp. 778-795, Springer-Verlag, 2003.
[53] A. R. Bradley, H. B. Sipma, S. Solter, Z. Manna. Integrating tools for practical software analysis. Proc. of 2004 CUE Workshop, Vienna, Austria, October 2004.
[54] T. Gilb, D. Graham. Software Inspection. Addison-Wesley, 1993.
[55] A. Porter, H. Siy, L. Votta. A Review of Software Inspections. University of Maryland at College Park, Technical Report CS-TR-3552, 1995.
[56] O. Laitenberger. A Survey of Software Inspection Technologies. In Handbook on Software Engineering and Knowledge Engineering, v. 2, pp. 517-555. World Scientific Publishing, 2002.
[57] GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint/index.html.
[58] B. Daum. Professional Eclipse 3 for Java Developers. Wrox, 2004.
[59] http://www.eclipse.org/.
[60] G. Yorsh, T. Ball, M. Sagiv. Testing, abstraction, theorem proving: better together! Proc. of ISSTA 2006, Partland, Maine, USA, 2006, pp. 145-156.
[61] C. Beust, H. Suleiman. Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley Professional, 2007.
[62] http://testng.org/doc/.