Подход UniTesK к разработке тестов: достижения и перспективы

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

Аннотация. Данная статья излагает базовые принципы технологии разработки тестов UniTesK, основанной на использовании формальных моделей тестируемого программного обеспечения (ПО). Кроме того, излагается опыт использования этой технологии для тестирования ПО разных видов, описываются проблемы внедрения в промышленность, свойственные всем технологиям, основанным на формальных методах, и формулируются возможные пути их решения. Технология UniTesK была разработана в группе спецификации, верификации и тестирования [RedVerst] ИСП РАН на основе многолетнего опыта проведения проектов верификации и тестирования сложного промышленного ПО.

Статья содержит материал, который заинтересует как исследователей в области формальных методов, так и практиков, которые хотели бы ознакомиться с потенциалом тестирования на основе моделей в реальных крупномасштабных приложениях. Читателям из последней группы рекомендуется после разделов «Основные принципы UniTesK» и «Процесс построения тестов по UniTesK» перейти к разделу «Опыт использования UniTesK», а затем выбрать те разделы, которые покажутся им интересными.

1. Введение

В настоящее время промышленное производство программного обеспечения (ПО) достигло таких масштабов и такой степени сложности, что необходимость в индустриально применимых технологиях систематического тестирования общепризнана. Особенно актуальным является создание таких технологий, которые обеспечивают одновременно качественное, систематическое тестирование целевого ПО и высокую степень автоматизации разработки тестов. Традиционные методы разработки тестов вручную уже не могут обеспечить качественное тестирование современных программных систем.

Обычно автоматизация тестирования сводится к автоматизации выполнения тестов и генерации отчетов по их результатам. Автоматизировать подготовку тестов и анализ полученных результатов труднее, поскольку при этом необходимо обращение к *требованиям к ПО,* соответствие которым, должно быть проверено во время тестирования. Требования же часто представлены в

виде неформальных документов, а иногда — только как знания и опыт экспертов, аналитиков и проектировщиков ПО.

Для того, чтобы вовлечь требования в автоматизированный процесс разработки тестов, необходимо перевести их в формальное представление, которое может быть обработано полностью автоматически. Для этой цели требования описывают в виде *формальных спецификаций* целевой системы, которые можно преобразовать в программы, выполняющие проверку соответствия работы целевого ПО зафиксированным в них требованиям.

Несмотря на активное развитие методов построения тестов на основе формальных спецификаций или формальных моделей в академическом сообществе, лишь немногие из них оказываются применимыми в индустрии производства ПО. Основная проблема здесь в том, что индустрии нужны не отдельные методы, а *технологии,* т.е. инструментально поддержанные системы методов для решения наборов связанных задач, относящихся к выделенному аспекту разработки ПО.

Данная статья представляет описание технологии UniTesK, которая была разработана в ИСП РАН на основе опыта нескольких проектов по верификации сложного промышленного ПО и нацелена на то, чтобы сделать возможным использование передовых методов тестирования в контексте индустриального производства ПО. UniTesK в первую очередь предназначена для разработки функциональных тестов на основе моделей требований к функциональности целевой системы. Проблемы построения тестов для проверки нефункцио-нальных требований выходят за рамки данной работы.

Структура статьи такова. Следующий за введением раздел содержит описание основных элементов технологии UniTesK, начиная с общего обзора ее базовых принципов и дальше раскрывая некоторые из них в деталях. В третьем разделе проводится сравнение UniTesK с другими подходами к разработке тестов на основе моделей. В четвертом разделе кратко описываются примеры приложений UniTesK и опыт использования этой технологии для тестирования промышленного ПО. В заключении рассматриваются направления дальнейшего развития этой технологии.

2. Описание технологии UniTesK

2.1. Основные принципы UniTesK

Технология построения тестов для ПО общего назначения становится пригодной для широкого использования в промышленной практике, только когда она обладает следующими характеристиками. Во-первых, все определяемые ею операции, где это возможно, должны поддерживаться инструментами. Во-вторых, она должна обладать широким набором функций, позволяющим использовать ее в проектах, имеющих различные цели, для тестирования ПО из разных предметных областей и построенного с

использованием различных архитектур и технологий. И, наконец, она должна достаточно хорошо интегрироваться с имеющимися процессами разработки, в частности, быть основана на системе понятий и обозначений, достаточно простой и широко используемой, чтобы не требовать долгой и дорогой переподготовки персонала.

Для обеспечения таких характеристик при разработке UniTesK были предложены следующие решения.

1. Для обеспечения максимальной гибкости технологии, была спроектирована *универсальная архитектура теста,* определяющая набор компонентов теста с ясным разделением функций и четкими интерфейсами, так, чтобы большое многообразие различных видов тестов для разных программ можно было реализовать в ее рамках.
2. Чтобы сделать возможной значительную степень автоматизации, в рамках полученной архитектуры 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. Во многих случаях *все* изменяемые компоненты теста, кроме спецификаций, определяющих критерии корректности ПО, могут быть сгенерированы интерактивно, на основе ответов пользователя на ряд четко поставленных вопросов.
3. В качестве метамодели для представления функциональных спецификаций, моделирующих требования, был выбран широко известный подход на основе *программных контрактов* (Design by Contract [DBCA,DBCO,DBCE]), состоящих из *предусловий* и *постусловий* интерфейсных операций и *инвариантов* типов данных. Программные контракты, с одной стороны, достаточно удобны для разработчиков ПО, поскольку хорошо привязываются к архитектуре ПО, с другой стороны, в силу своего представления стимулируют усилия по созданию независимых от реализации критериев корректности целевой системы. Основное же их преимущество в том, что они позволяют автоматически построить *оракулы* [Parnas,KVEST,ADLt], проверяющие соответствие поведения целевой системы спецификациям, и *критерии тестового покрытия,* которые достаточно близки к критериям покрытия требований.
4. Практически невозможно обеспечить универсальный механизм построения единичных тестовых воздействий (например, вызовов операций с разными наборами аргументов), который был бы достаточно эффективен как по времени, затраченному на тестирование, так и с точки зрения достижения высокого тестового покрытия. В то же время, довольно просто построить итератор, перебирающий большое множество значений некоторого типа. Инструменты, поддерживающие UniTesK, предоставляют пользователям библиотеки базовых *итераторов* значений простых типов, которые могут быть непосредственно использованы для генерации тестовых воздействий,

а могут быть скомпонованы в более сложные генераторы. Для уменьшения затрат времени на тестирование сгенерированные тестовые воздействия можно фильтровать, отбрасывая те и них, которые не увеличивают достигнутый уровень покрытия. Фильтры для этого генерируются автоматически из определения критерия тестового покрытия (см. пункт 6).
5. Для автоматического построения последовательности тестовых воздействий используются модели тестируемой системы в виде *конечных автоматов* (КА). Тестовая последовательность строится как последовательность обращений к целевым операциям, соответствующая некоторым путям в графе переходов КА, например, обходу всех переходов автомата. Поскольку конечно-автоматная модель используется только для построения тестовой последовательности, а не для проверки корректности поведения целевой системы, осуществляемой оракулами, можно не задавать автомат полностью, а лишь указать способ идентификации его состояний и способ итерации входных воздействий в зависимости от текущего состояния. Представленные таким, *неявным,* образом автоматы удобно задавать в виде *тестовых сценариев.* Часто тестовый сценарий можно сгенерировать автоматически на основе спецификации целевых операций, способа итерации наборов их аргументов и стратегии тестирования.
6. Стратегия тестирования определяет, когда тестирование можно заканчивать. UniTesK предлагает при этом опираться на достигнутый уровень тестового покрытия в соответствии с некоторым критерием покрытия. Из структуры спецификаций, разработанных в соответствии с технологией UniTesK, можно автоматически извлечь несколько таких критериев. Пользователь имеет возможность гибко управлять этими критериями или определять свои собственные.
7. Чтобы обеспечить более удобную интеграцию в существующие процессы разработки, UniTesK может использовать для представления спецификаций и тестовых сценариев расширения широко используемых языков программирования, построенные на основе единой системы понятий (хотя классические языки формальных спецификаций тоже могут использоваться). Такое представление делает спецификации и сценарии понятнее для обычного разработчика ПО и позволяет сократить срок освоения основных элементов технологии до одной недели. Сразу после этого обучения разработчик тестов может использовать UniTesK для получения практически значимых результатов. Кроме того, использование расширений известных языков программирования вместо специального языка значительно облегчает интеграцию тестовой и целевой систем, необходимую для проведения тестирования. На данный момент в ИСП РАН разработаны инструменты, поддерживающие работу по технологии UniTesK с использованием расширений Java, C и C#.
8. Спецификации на основе программных контрактов в рамках технологии UniTesK могут использоваться не только как вставки в исходный код

целевой системы. Они могут быть отделены от целевого кода и использоваться в неизменном виде для тестирования различных реализаций одной и той же функциональности, таким образом представляя собой формализацию функциональных требований к ПО. Для определения связи между спецификациями и конкретной реализацией используются специальные компоненты, *медиаторы,* которые могут осуществлять довольно сложные преобразования интерфейсов. Использование медиаторов открывает дорогу следующим возможностям.

- Спецификации могут быть гораздо более абстрактными, чем реализация, и, тем самым, более близкими к естественному представлению функциональных требований.
- Спецификации остаются актуальными для нескольких версий целевого ПО. Для переработки тестового набора под новую версию, в которой изменились внешние интерфейсы, но не их функции, достаточно заменить медиаторы. Во многих случаях такая замена может быть автоматизирована.
- Становится возможным широкое переиспользование спецификаций и тестов, которое значительно повышает отдачу от вложенных в их разработку ресурсов.

При использовании технологии UniTesK в специфической области зачастую бывают нужны не все техники построения тестов из набора входящих в технологию, и не все компоненты из универсальной архитектуры теста бывает необходимо строить. А в некоторых случаях использование каких-то техник невозможно или требует слишком больших затрат. Тем не менее, в этих случаях можно использовать специализированные варианты технологии и поддерживающие их инструменты.

Например, при тестировании блоков оптимизации в компиляторах разработка спецификаций функциональности такого блока в полном объеме, если и возможна, то очень трудоемка, поскольку они должны, например, выражать тот факт, что оптимизация программы действительно была проведена. В то же время, сравнить быстродействие и проверить неизменность функциональности тестовых программ специального вида довольно легко, выполняя их на конечном наборе входных значений, что дает способ построения оракулов, хотя и не столь общий, как описанный выше, но достаточный для практических целей [OPT].

2.2. Универсальная архитектура теста

Гибкость технологии или инструмента, возможность использовать их в большом многообразии различных ситуаций и контекстов, определяется, в первую очередь, лежащей в основе данной технологии или данного инструмента архитектурой. Архитектура теста, используемая в UniTesK проектировалась на основе опыта проведения тестирования сложного промышленного ПО. Она нацелена на решение двух основных проблем.

- Невозможно полностью автоматизировать разработку тестов, поскольку критерии корректности целевого ПО и стратегии проведения тестирования может предоставить только человек. Тем не менее, очень многое может и должно быть автоматизировано.
- Выбранная архитектура должна совмещать единообразие с возможностью тестирования ПО, относящегося к разным предметным областям, и в проектах, решающих различные задачи.

Основная идея архитектуры теста UniTesK состоит в том, что разрабатывается набор компонентов, пригодный для тестирования различных видов ПО с использованием разных стратегий тестирования. Эти компоненты должны иметь четко определенные обязанности в системе и интерфейсы для взаимодействия друг с другом. Далее, информация, которую в общем случае может предоставить только разработчик тестов, концентрируется в небольшом числе компонентов с четко определенными ролями. Для каждого такого компонента разрабатывается компактное и простое представление, создание которого потребует минимальных усилий со стороны пользователя.

Архитектура теста UniTesK [UniArch] основана на следующем разделении задачи тестирования на подзадачи:

1. Задача проверки корректности поведения системы в ответ на единичное воздействие
2. Задача создания единичного тестового воздействия
3. Задача построения последовательности таких воздействий, нацеленной на достижение нужного покрытия
4. Задача установления связи между тестовой системой, построенной на основе абстрактного моделирования, и конкретной реализацией целевой системы

Для решения каждой из этих задач предусмотрена технологическая поддержка.

Для проверки корректности реакции целевого ПО в ответ на одно воздействие используются *тестовые оракулы.* Поскольку генерация тестовых воздействий отделена от проверки реакции системы на них, нужно уметь оценить поведение системы при достаточно произвольном воздействии. Для этого не подходит распространенный способ получения оракулов, основанный на вычислении корректных результатов для фиксированного набора воздействий. Используются оракулы общего вида, основанные на предикатах, связывающих воздействие и ответную реакцию системы.

Такие оракулы легко строятся из спецификаций программного контракта в виде пред- и постусловий интерфейсных операций и инвариантов типов, формулирующих условия целостности данных [KVEST,ADLt]. При таком подходе каждое возможное воздействие моделируется как обращение к одной из интерфейсных операций с некоторым набором аргументов, а ответ системы на него — в виде результата этого вызова. Далее будут более детально

рассмотрены специфика моделирования асинхронных реакций целевой системы и используемые техники специфицирования.

Единичные тестовые воздействия строятся при помощи механизма перебора операций и итерации некоторого широкого множества наборов аргументов для фиксированной операции, которые дополняются фильтрацией полученных наборов по критерию покрытия, выбранному в качестве цели тестирования.

Для тестирования ПО со сложным поведением, зависящим от предшествующего взаимодействия ПО с его окружением, недостаточно набора единичных тестовых воздействий. При тестировании таких систем используют последовательности тестовых воздействий, называемые *тестовыми последовательностями* и построенные таким образом, чтобы проверить поведение системы в различных ситуациях, определяемых последовательностью предшествовавших обращений к ней и ее ответных реакций.

Для построения последовательности тестовых воздействий используется конечно-автоматная модель системы. Конечные автоматы достаточно просты, знакомы большинству разработчиков и могут быть использованы для моделирования практически любой программы. Для тестирования параллелизма или распределенных систем используется разновидность *автоматов ввода/вывода* [IOSMA], в которых переходы помечаются только входным или только выходным символом. Итоговый конечный автомат представлен в виде *итератора тестовых воздействий.* Этот компонент имеет интерфейс для получения идентификатора текущего состояния, получения идентификатора очередного воздействия, допустимого в данном состоянии, и для выполнения воздействия по его идентификатору.

Подробнее об используемых автоматных моделях для тестирования параллелизма можно прочитать в [AsSM].

Тестовая последовательность строится во время тестирования динамически, за счет построения некоторого "исчерпывающего" пути по переходам автомата. Это может быть обход всех его состояний, всех его переходов, всех пар смежных переходов и т.п. Алгоритм построения такого пути на достаточно широком классе автоматов оформлен в виде другого компонента теста, *обходчика.*

Удобное для человека описание используемой при тестировании конечно-автоматной модели мы называем *тестовым сценарием.* Из тестового сценария генерируется итератор тестовых воздействий. Сценарии могут разрабатываться вручную, но для многих случаев достаточно сценариев, которые можно получить автоматически на основе набора спецификаций операций, указания целевого критерия покрытия, способа итерации параметров операций и способа вычисления идентификатора состояния. Более детально методы построения тестовых последовательностей рассматриваются ниже, в соответствующем подразделе. Обходчики нескольких разных видов предоставляются в виде библиотечных классов, и пользователю нет нужды разрабатывать их самому.

Для того, чтобы использовать в тестировании спецификации, написанные на более высоком уровне абстракции, чем сама целевая система, UniTesK предоставляет возможность использовать *медиаторы.* Медиатор задает связь между некоторой спецификацией и конкретной реализацией соответствующей функциональности. При этом он определяет преобразование модельных представлений воздействий (вызовов модельных операций) в реализационное, и обратное преобразование реакций целевой системы в их модельное представление (результат, возвращаемый модельной операцией).

Медиаторы удобно разрабатывать в расширении целевого языка, где описывать только сами перечисленные преобразования. Требуется дополнительная обработка полученного кода, поскольку помимо своих основных функций медиатор выполняет дополнительные действия, связанные со спецификой среды реализации и с трассировкой хода теста. Код этих действий автоматически добавляется к процедурам преобразования стимулов и реакций, описанным пользователем.

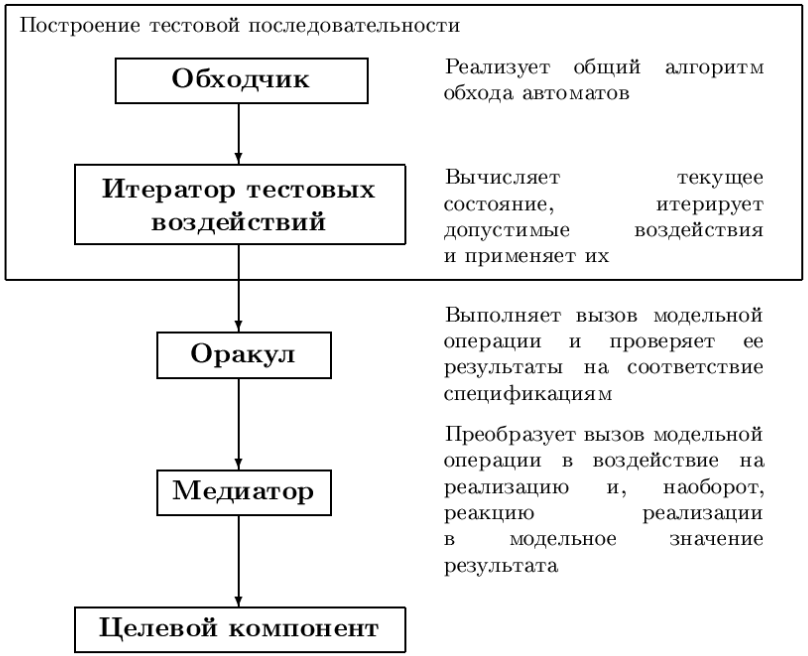


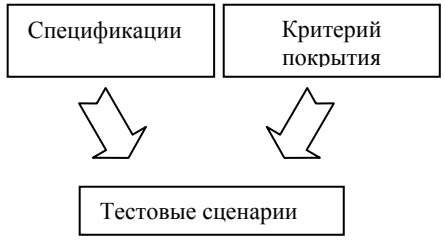
Рис. 1. Архитектура теста UniTesK.

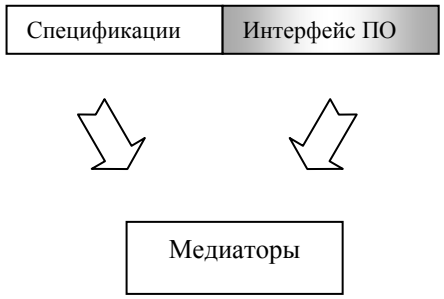
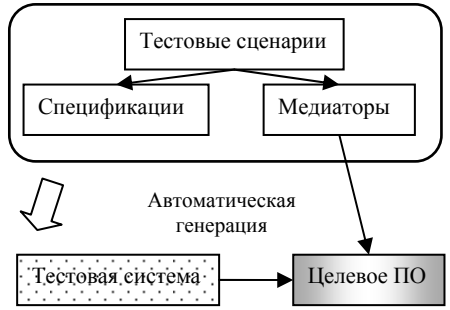
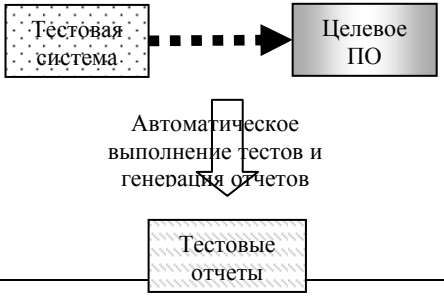
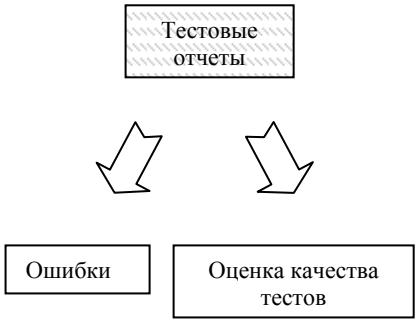
Рис. 1 представляет основные компоненты архитектуры теста, используемой UniTesK. В дополнение к этим компонентам тестовая система содержит несколько вспомогательных, отвечающих за трассировку хода тестирования, своевременную синхронизацию состояний между модельными и реализа-

ционными объектами, и пр. Эти вспомогательные компоненты не зависят от тестируемого ПО и выбранной стратегии тестирования.

2.3. Процесс построения тестов по UniTesK

В приведенной ниже таблице процесс разработки спецификаций и тестов и собственно тестирования в UniTesK представлен как последовательность шагов. На практике, конечно, бывает необходимо вернуться к ранее пройденным шагам и пересмотреть уже принятые решения, но общая логика процесса — это движение от требований к получению тестов и анализу результатов тестирования.

1. Анализ функциональных требований к целевому ПО на основе имеющихся документов или знаний участников проекта, запись требований в виде формальных спецификаций.	
2. Формулировка требования к качеству тестирования — какой уровень тестового покрытия будет считаться достаточным, чтобы прекратить тестирование.	
3. Разработка набора тестовых сценариев, обеспечивающего достижение нужного уровня покрытия. Сценарии разрабатываются на основе спецификаций и не привязаны к конкретной реализации целевого ПО или его конкретной версии.	

4. Привязка полученных тестовых сценариев к конкретной реализации целевой системы. Для этого надо разработать набор медиаторов.	
5. Получение готовой к исполнению тестовой программы. Для того нужно оттранслировать спецификации, медиаторы и сценарии с расширения языка программирования в целостную тестовую систему на целевом языке программирования и скомпилировать ее.	
6. Выполнение тестовой программы, возможно, отладка спецификаций, медиаторов и тестовых сценариев.	
7. Анализ результатов, а именн,о анализ полноты тестового покрытие и принятие решения и продолжении или прекра-щении тестирования. Оформ-ление описания ошибок.	

Представленная ниже, на Рис. 2, схема показывает процесс получения основных компонентов архитектуры тестового набора по технологии UniTesK.

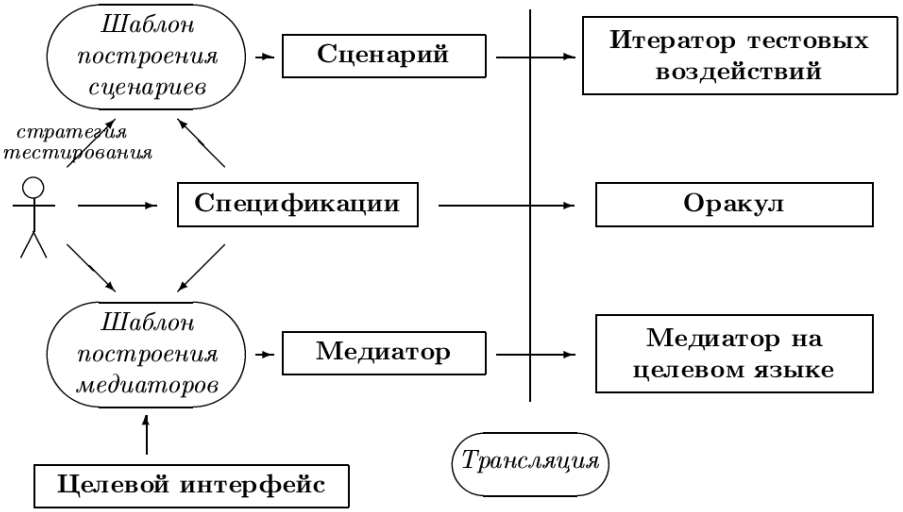


Рис. 2. Получение компонентов теста по UniTesK.

Теперь рассмотрим действия, которые нужно предпринимать на разных шагах технологии более подробно.

2.4. Описание функциональных требований

UniTesK поддерживает автоматическую генерацию тестовых оракулов из спецификаций в виде программных контрактов. При использовании такого способа описания функциональности целевой системы, она моделируется как набор компонентов, каждый из которых имеет несколько интерфейсных операций с некоторыми параметрами. Окружение системы может вызывать интерфейсные операции и получать результаты их работы. Эти результаты определяются вызванной операцией, ее аргументами и историей взаимодействий системы с ее окружением, предшествовавших данному. Существенная информация об истории моделируется как *внутреннее состояние* компонентов целевой системы. Таким образом, поведение операций, вообще говоря, зависит от внутреннего состояния и может его менять.

Каждая операция описывается при помощи *предусловия* и *постусловия.* Предусловие определяет условия, при которых данная операция может быть вызвана извне, причем за соблюдение этих условий ответственно окружение, клиенты данного компонента. Можно сказать, что предусловие описывает область определения операции в пространстве возможных состояний и

наборов ее аргументов. Постусловие устанавливает ограничения на исходное состояние, аргументы, результат операции и итоговое состояние, которые должны быть выполнены, если перед обращением к данной операции было выполнено ее предусловие.

Операция может иметь параметры некоторых типов. Такие типы, типы полей модельного состояния, а также сами типы модельных компонентов называются *интерфейсными типами.* Для всех интерфейсных типы описывается их структура данных, которая может иметь ограничения на их целостность, выраженные в виде *инвариантов.* Структура данных модельных компонентов определяет возможные модельные состояния системы.

Программные контракты были выбраны в качестве основной техники специфицирования, поскольку они достаточно просты, применимы для ПО из очень многих предметных областей и могут быть сделаны достаточно абстрактными или достаточно детальными по мере необходимости. Обычного разработчика ПО можно научить понимать их и пользоваться ими без особых усилий.

Кроме того, программные контракты, будучи по структуре близки к архитектуре целевой системы, что делает их понятными для разработчиков, по внутреннему содержанию достаточно близки к требованиям к системе. Таким образом, во-первых, переработка требований в контракты не требует больших затрат, а, во-вторых, результат обычно не слишком близок к описанию конкретных алгоритмов, используемых в реализации, что предотвращает во многих случаях появления ошибок одного вида и в реализации, и в спецификациях.

Контрактные спецификации не единственный вид спецификаций, поддерживаемый технологией UniTesK. В ее рамках возможно использование исполнимых спецификаций, явно описывающих, как вычисляется результат вызванной операции и как преобразуется внутреннее состояние компонента, к которому обратились. При этом, однако, дополнительно надо определить критерии эквивалентности модельных и реализационных результатов, которые не во всех случаях обязаны быть совпадающими.

Аксиоматические спецификации часто не могут быть напрямую преобразованы в оракулы, оценивающие корректность поведения системы в ответ на произвольное воздействие. Поэтому аксиоматические спецификации используются только как дополнительные критерии проверки корректности на основе реакции системы на некоторые последовательности воздействий, и служат для построения тестовых сценариев.

2.5. Критерии тестового покрытия, основанные на спецификациях

Структура программных контрактов используется в UniTesK для определения критериев покрытия спецификаций, которые необходимы, чтобы оценить качество тестирования с точки зрения требований. Для того, чтобы сделать возможным автоматическое извлечение такого рода критериев, накладываются дополнительные ограничения на структуру постусловий. А именно, вводятся дополнительные операторы для определения *ветвей функциональности,*

расстановка которых в постусловии лежит на пользователе. Ветвь функциональности соответствует подобласти в области определения операции, в которой операция ведет себя "одинаково". Для большей определенности "одинаковым" можно считать такое поведение, при котором ограничения на результат работы операции и изменение состояния описываются для всех точек подобласти одним и тем же выражением в постусловии.

В графе потока управления постусловия на каждом пути от входа к любому из выходов должен находиться ровно один оператор, определяющий ветвь функциональности, причем на части пути до такого оператора не должно быть ветвлений, зависящих от результатов работы операции. Тогда, во-первых, каждый допустимый вызов данной операции может быть однозначно отнесен к одной из ветвей функциональности, и, таким образом, можно измерять качество тестирования операции как процент покрытых во время теста ее ветвей функциональности. Во-вторых, определить ветвь функциональности можно по текущему состоянию компонента и набору аргументов операции, не выполняя саму операцию, что позволяет построить фильтр, отсеивающий наборы аргументов, не добавляющие ничего к уже достигнутому покрытию.

Отталкиваясь от определения ветвей функциональности в постусловии, можно автоматически извлечь более детальные критерии покрытия, основанные на структуре ветвлений в пред- и постусловиях. Наиболее детальным является критерий покрытия дизъюнктов. Он определяется всеми возможными комбинациями значений элементарных логических формул, использованных в ветвлениях. Этот критерий является аналогом критерия MC/DC [MCDC] для покрытия кода.

При тестировании, нацеленном на достижение высокого уровня покрытия по дизъюнктам, возможны проблемы (аналогичные проблемам, возникающим при использовании критерия MC/DC), связанные с недостижимостью некоторых дизъюнктов в силу наличия неявных семантических связей между используемыми логическими формулами. Такие проблемы решаются при помощи явного описания имеющихся связей в виде тавтологий, т.е. логических выражений, построенных из элементарных формул и являющихся тождественно истинными в силу зависимостей между значениями формул.

Помимо возможности управлять автоматически извлекаемыми из структуры спецификаций критериями покрытия, пользователь может описать свои собственные критерии покрытия спецификаций в виде наборов предикатов, зависящих от аргументов операций и состояния, и использовать их для определения целей тестирования.

2.6. Построение тестовых последовательностей

UniTesK использует конечно-автоматные модели целевого ПО в виде тестовых сценариев для динамической генерации последовательностей тестовых воздействий. Сценарий определяет, что именно рассматривается как состояние автомата и какие операции с какими наборами аргументов должны быть

вызваны в каждом состоянии. Во время выполнения теста обходчик строит некоторый "исчерпывающий" путь по переходам автомата, порождая тем самым тестовую последовательность.

Такой метод построения теста гарантирует, что состояние системы изменяется только за счет вызовов целевых операций, и только достижимые этим способом состояния будут возникать во время тестирования. Таким образом, перебор состояний осуществляется автоматически, и разработчику теста достаточно указать только нужный способ перебора аргументов вызываемых операций.

При разработке сценария можно использовать некоторый критерий покрытия спецификаций в качестве целевого и определить набор состояний и переходов таким образом, чтобы обход всех переходов в получившемся автомате гарантировал достижение нужного покрытия. Для этого достаточно рассмотреть набор предикатов, определяющий элементы выбранного критерия покрытия для некоторой тестируемой операции, как набор областей в пространстве состояний и аргументов этой операции, и взять проекции полученных областей на множество состояний.

Построив все возможные пересечения таких проекций для всех тестируемых операций, мы получим набор таких множеств состояний, что, вызывая любые операции в двух состояниях из одного такого множества, можно покрыть одни и те же элементы по выбранному критерию покрытия. Следовательно, все такие состояния системы эквивалентны с точки зрения выбранного критерия покрытия, и можно объявить состоянием результирующего автомата полученные множества. Стимулами в таком автомате считаются классы эквивалентности вызовов операций по выбранному критерию покрытия, т.е. покрывающие один и тот же его элемент. Может потребоваться дополнительно преобразовать полученный автомат, чтобы сделать его детерминированным, подробности см. в [FACTOR].

При проведении тестирования можно использовать автоматически сгенерированные из спецификаций фильтры, отсеивающие наборы аргументов, не дающие вклада в уже достигнутое покрытие. Наличие таких фильтров позволяет во многих случаях не тратить усилий человека на вычисление необходимых для достижения нужного покрытия аргументов, а указать в качестве перебираемого набора их значений некоторое достаточно большое множество, которое наверняка содержит нужные значения. Так UniTesK позволяет проводить тестирование, нацеленное на достижение высоких уровней покрытия, не затрачивая на это значительных ресурсов.

Тестовый сценарий представляет конечный автомат в неявном виде, т.е. состояния и переходы не перечисляются явно, и для переходов не указываются конечные состояния. Вместо этого определяется способ вычисления текущего состояния и метод сравнения состояний, способ перебора допустимых воздействий (тестируемых операций и их аргументов), зависящий от состояния, и процедура применения воздействия. Хотя такое представление авто-

матных моделей необычно, оно позволяет описать в компактном виде довольно сложные модели, а также легко вносить модификации в полученные модели.

Сценарий может определять состояния описываемой автоматной модели, основываясь не только на модельном состоянии, описанном в спецификациях, но и учитывая какие-то аспекты реализации, не нашедшие отражения в спецификациях. С другой стороны, можно также абстрагироваться от каких-то деталей в спецификациях, уменьшая тем самым число состояний в результирующей модели (см. [FACTOR]). Таким образом, способ построения теста может варьироваться независимо от спецификаций, и, следовательно, независимо от механизма проверки корректности поведения при единичном воздействии.

Тестовые сценарии можно разрабатывать вручную, но в большинстве случаев они могут быть сгенерированы при помощи интерактивного инструмента, *шаблона построения сценариев,* который запрашивает у пользователя только необходимую информацию, и может использовать разумные умолчания. Шаблон построения сценариев помогает строить как сценарии, не использующие фильтрацию тестовых воздействий, так и нацеленные на достижение высокого уровня покрытия по одному из извлекаемых из спецификаций критериев.

Тестовые сценарии, написанные в терминах спецификаций, тем самым определяют абстрактные тесты, которые можно использовать для тестирования любой системы, описываемой данными спецификациями. Кроме того, сценарии имеют дополнительные возможности для переиспользования при помощи ме-ханизма наследования. Сценарий, наследующий данному, может переопреде-лить в нем процедуру вычисления состояния и переопределить или пополнить набор тестовых воздействий, оказываемых на систему в каждом состоянии.

Для тестирования параллелизма и распределенных систем UniTesK предполагает использование специального вида обходчиков, которые генерируют пары, тройки и более широкие наборы параллельных воздействий в каждом состоянии, и слегка расширенных спецификаций. В дополнение к спецификациям операций, моделирующих воздействия на целевую систему и ее синхронные реакции на эти воздействия, можно специфицировать *асинхронные реакции* системы, каждая из которых оформляется в виде операции без параметров, имеющей пред- и постусловия.

Без спецификаций асинхронных реакций можно тестировать системы, удовлетворяющие *аксиоме чистого параллелизма (plain concurrency axiom):* результат параллельного выполнения любого набора вызовов операций такой системы такой же, как при выполнении того же набора вызовов в некотором порядке. Для систем, не удовлетворяющих этой аксиоме, можно ввести дополнительные "срабатывания", соответствующие выдаче асинхронных реакций или внутренним, не наблюдаемым извне, изменениям состояния системы, таким образом, что полученная модель уже будет "чистой" (plain). Автоматные модели, используемые для тестирования таких систем, являются некоторым обобщением автоматов ввода/вывода [IOSMA]. При проведении

тестирования "чистой" системы используется следующий метод проверки корректности ее поведения. Если обработанные системой воздействия и полученные от нее асинхронные реакции можно линейно упорядочить таким образом, что в полученной последовательности перед каждым вызовом или реакцией будет выполнено его/ее предусловие, а после — постусловие, то система ведет себя корректно. Фактически, это означает, что ее наблюдаемое поведение не противоречит спецификациям. Если такого упорядочения построить нельзя, значит обнаружено несоответствие поведения системы спецификациям.

Помимо указанных выше возможностей, тестовые сценарии UniTesK дают пользователю возможность проводить тестирование, основанное на обычных сценариях, т.е, последовательностях воздействий, формируемых по указанному разработчиком теста правилу, корректность поведения системы при которых тоже оценивается задаваемым разработчиком способом. В качестве таких сценариев для системного тестирования можно, в частности, использовать сценарии, уточняющие варианты использования целевой системы.

Другой способ построения сценариев дают аксиоматические спецификации, описывающие правильное поведение системы в виде ограничений на результаты выполнения некоторых цепочек вызовов целевых операций. Каждая такая цепочка вместе с проверкой наложенных на ее результаты ограничений может быть оформлена в тестовом сценарии в виде одного воздействия, которое будет выполняться во всех состояниях, где оно допустимо. Аксиомы алгебраического вида, требующие эквивалентных результатов от двух или нескольких цепочек вызовов, также могут быть проверены за счет оформления каждой цепочки в виде отдельного воздействия и сравнения ее результатов с результатами ранее выполненных в том же самом состоянии цепочек. Тестовые сценарии предоставляют удобный механизм для хранения промежуточных данных (в данном случае, результатов предыдущих цепочек) при идентификаторе состояния.

2.7. Определение связи спецификаций и реализации

Спецификации, используемые UniTesK для разработки тестов, могут быть связаны с реализацией не прямо, а при помощи медиаторов. Это делает возможной разработку и использование более абстрактных спецификаций, которые гораздо удобнее получать из требований и можно использовать для тестирования нескольких версий целевого ПО. Таким образом, тесты становятся более абстрактными и многократно используемыми. Помимо преимуществ, перечисленных в начале данного раздела, можно указать дополнительные выгоды от такого способа организации разработки теста.

- Соответствие между требованиями, представленными в виде спецификаций, и тестами может отслеживаться полностью автоматически.
- Поддерживается ко-верификационная разработка ПО, при которой сама целевая система и тесты к ней разрабатываются одновременно и

параллельно, и сокращается общий срок разработки ПО с определенным уровнем качества.

- Появляется поддержка для более эффективной инфраструктуры распространения готовых компонентов ПО на коммерческой основе. Для функциональности, реализуемой такими компонентами можно иметь общедоступные, один раз написанные спецификации, дополненные тестовым набором, убедительно показывающим, что компонент действительно реализует указанные функции. Разработчик компонента может сопроводить свою реализацию медиаторами, связывающими ее с общедоступными спецификациями, тем самым, позволяя любому пользователю или независимому тестировщику убедиться в ее правильности. Кроме того, пользователи таких компонентов могут использовать для тестирования тестовые наборы, пополненные нужным им способом.

Медиаторы можно разрабатывать вручную и определять, таким образом, довольно сложные преобразования между интерфейсом модели и интерфейсом реализации целевой системы. В простых случаях можно использовать *шаблон построения медиаторов,* который позволяет сгенерировать медиатор автоматически, указав спецификационный и реализационный компоненты, которые нужно связать, и определив соответствие между их операциями. Для каждой операции при этом нужно указать способ преобразования модельных аргументов в реализационные и реализационных результатов в модельные, если эти преобразования не тождественны.

UniTesK позволяет использовать доступную извне информацию о состоянии реализации для построения модельного состояния. Способ тестирования, при котором модельное состояние целиком строится на основе доступной достоверной информации о состоянии реализации, независимо от вызываемых целевых операций, называется *тестированием с открытым состоянием.* Процедура построения модельного состояния при таком тестировании оформляется в отдельную операцию в медиаторе, автоматически вызываемую тестовой системой после каждого вызова целевой операции (если нет параллельных обращений к целевой системе или асинхронных реакций).

Если же нам недоступна информация, достаточная для построения модельного состояния (или проводится тестирование параллельных обращений, или система может создавать асинхронные реакции), используется *тестирование со скрытым состоянием.* При таком тестировании модельное состояние после вызова некоторой операции строится на основе предшествовавшего вызову модельного состояния, аргументов и результатов данного вызова. Этот способ дает гипотетическое очередное модельное состояние при условии, что наблюдаемые результаты вызова не противоречат спецификациям. Он корректен, если ограничения, указанные в постусловии любой операции, можно однозначно разрешить относительно модельного состояния компонента после вызова. Медиаторы для такого тестирования должны содержать для каждой модельной операции построение модельного состояния после вызова этой операции.

2.8. Универсальное расширение языков программирования

Обычно формальные спецификации записываются на специализированных языках, имеющих большой набор выразительных возможностей и строго определенную семантику. UniTesK позволяет использовать такие языки, если для каждой используемой пары (язык спецификаций, язык реализации) сформулированы четкие правила преобразования интерфейсов и реализована инструментальная поддержка такого преобразования.

Однако, во многих случаях, несмотря на эти преимущества, специализированные языки формальных спецификаций тяжело использовать для тестирования из-за трудностей при определении указанных преобразований. Эти трудности связаны с несовпадением парадигм, лежащих в основе двух языков, спецификационного и языка реализации, с отсутствием в языке спецификаций аналогов понятий, широко используемых в реализации (например, указателей), с несовпадением семантики базовых типов и пр. Поэтому такая работа требует обычно больших затрат труда высококвалифицированных специалистов, хорошо знакомых с обоими языками. Кроме того, обучение такой работе также весьма трудоемко и начинает давать практические результаты только по истечении значительного времени.

Для того, чтобы сделать технологию более доступной обычным разработчикам, и для облегчения разработки медиаторов UniTesK поддерживает написание спецификаций и сценариев на расширениях широко используемых языков программирования. Для этого построена единая система базовых понятий, используемых при разработке спецификаций и сценариев, таких как предусловие, постусловие, инвариант, ветвь функциональности, сценарный метод (определяющий в сценарии однородное семейство тестовых воздействий), и для каждого их этих понятий сформулированы правила дополнения языка соответствующей конструкцией. Для языка, в котором уже имеются средства для выражения понятий, аналогичных выделенным, пополнение производится только конструкциями, не имеющими аналогов.

Значительное преимущество использования расширения языка целевой системы для спецификаций состоит в том, что связывать такую спецификацию с реализацией гораздо проще. При использовании для спецификаций расширения целевого языка, обучиться работе с ними может обычный разработчик, имеющий опыт работы с целевым языком. Проблема недостаточной выразительности в большинство современных объектно-ориентированных языков решается при помощи использования библиотек абстрактных типов.

Проблема возможной зависимости смысла спецификации от платформы может решаться несколькими способами. Во-первых, можно запретить использование в спецификациях конструкций, имеющих недостаточно четкий смысл и по-разному интерпретируемых для разных платформ. Во-вторых, можно рекомендовать использовать библиотеки, реализованные так, чтобы работать

одинаково на всех поддерживаемых платформах. В-третьих, в особо специфических случаях можно проводить удаленное тестирование, при котором тестовая система исполняется на той же платформе, на которой разрабатывались спецификации.

В качестве примера мы рассмотрим спецификации функции, вычисляющей квадратный корень. Спецификации на расширениях различных языков программирования приводятся в левом столбце таблиц, представленных ниже. В правом столбце даются краткие пояснения, касающиеся структуры спецификаций.

Java

`specification package example;`	Декларация пакета
`class SqrtSpecification`	Декларация класса
`{`	
`specification static double sqrt ( double x )`	Сигнатура операции
`reads x, epsilon`	Описание доступа на чтение/запись
`{`	
`pre { return x >= 0; }`	Предусловие
`post`	Постусловие
`{`	
`if(x == 0)`	
`{`	
`branch "Zero argument";`	Определение ветви функциональности
`return sqrt == 0;`	Ограничения на результат
`}`	
`else`	
`{`	
`branch "Positive argument";`	Определение ветви функциональности
`return sqrt >= 0 &&`	Ограничения на результат
`Math.abs((sqrt*sqrt-x)/x)<epsilon;`	
`}`	
`}`	
`}`	
`}`	

C#

`namespace Examples`	Декларация пространства имен
`{`	
`specification class SqrtSpecification`	Декларация класса
`{`	
`specification static double Sqrt ( double x )`	Сигнатура операции
`reads x, epsilon`	Описание доступа на чтение/запись
`{`	
`pre { return x >= 0; }`	Предусловие
`post`	Постусловие
`{`	
`if(x == 0)`	
`{`	
`branch ZERO ("Zero argument");`	Определение ветви функциональности
`return $this.Result == 0;`	Ограничения на результат
`}`	
`else`	
`{`	
`branch POS ("Positive argument");`	Определение ветви функциональности
`return $this.Result >= 0 &&`	Ограничения на результат
`Math.Abs( ($this.Result * $this.`	
`Result - x)/x)`	
`< epsilon;`	
`}`	
`}`	
`}`	
`}`	
`}`	

C

`specification double SQRT ( double x )`	Сигнатура операции
`reads x, epsilon`	Описание доступа на чтение/запись
`{`	
`pre { return x >= 0.; }`	Предусловие
`coverage BRANCHES`	Описание структуры тестового покрытия
`{`	
`if(x == 0)`	
`return(ZERO, "Zero argument");`	Определение ветви функциональности
`else`	
`return(POS, "Positive argument");`	Определение ветви функциональности
`}`	
`post`	Постусловие
`{`	
`if(coverage(BRANCHES)==ZERO)`	
`return SQRT == 0.;`	Ограничения на результат
`else`	
`return SQRT >= 0. &&`	Ограничения на результат
`abs((SQRT*SQRT - x)/x) < epsilon;`	
`}`	
`}`	
`}`	

2.9. Выполнение тестов и анализ их результатов

Инструменты UniTesK поддерживают автоматическое выполнение тестов, разработанных с их помощью, и автоматический сбор трассировочной информации. После окончания работы теста на основе его трассы можно сгенерировать набор дополнительных тестовых отчетов. Эти отчеты показывают структуру автомата, выявленную в ходе тестирования, уровень достигнутого тестового покрытия для всех критериев, определенных для некоторой спецификационной операции, и информацию об обнаруженных в ходе теста нарушениях, связанных с ошибками в целевой системе или с ошибками в спецификациях, сценариях и медиаторах.

Трасса теста может служить для получения дополнительной информации, например, о зафиксированных нарушениях. Так, из трассы можно узнать вид нарушения, значения аргументов вызова операции, при выполнении которого это нарушение было обнаружено, какое именно ограничение в постусловии было нарушено, и т.д. Представленная в трассе и других отчетах информация достаточна, как для отладки тестовой системы, так и для оценки качества тестирования и, зачастую, для предварительной локализации обнаруженных ошибок.

Ниже приведены примеры отчетов, которые инструмент J@T, поддерживающий технологию UniTesK для программ на Java, автоматически формирует на основе трассы теста.

J@T Failure Report
generated: 12.12.2003 17:48:48

Report Overview

All Failures
Specifications Coverage
Scenarios Coverage

test situation		
branch	Withdrawn sum is too large	
mark	Withdrawal from empty account; Withdrawn sum is too large	
predicate	(!(0 < balance) && (balance == 0)) && (((balance - sum) < -maximumCredit))	
disjunct	true	0 < sum
	false	0 < balance
	true	balance == 0
	true	(balance - sum) < -maximumCredit
	true	balance == @balance
	false	withdraw == 0
	-	balance == (@balance – sum)
	-	withdraw == sum
	-	reads sum
	-	reads maximumCredit

Рис. 3. Описание найденной ошибки.

- Report Overview
- All Failures
- Specifications Coverage
  - Failures
  - Branches
  - Marks
  - Predicates
  - Disjuncts
- Scenarios Coverage
- Packages Overview
- ru.ispras.redverst.se .java.exam
- AccountSpecifi-cation
  - deposit(int)
  - withdraw(int)

deposit(int)										total
branches	marks	predicates	disjuncts							hits/fails
100% (1/1)	100% (3/3)	100% (3/3)	100% (3/3)							28
branches	marks	predicates	disjuncts							total
			f1	f2	f3	f4	f5	f6	f7	hits/fails
Single branch	Deposit on account with negative balance; Single branch	predicate1	+	+	-	-	*	*	*	6
	Deposit on empty account; Single branch	predicate2	+	+	-	+	*	*	*	2
	Deposit on account with positive balance; Single branch	predicate3	+	+	+	*	*	*	*	20

predicates	
identifier	meaning
predicate1	(!(0 < balance) && !(balance == 0))
predicate2	(!(0 < balance) && (balance == 0))
predicate3	((0 < balance))

prime formulas	
identifier	meaning
f1	0 < sum
f2	!((Integer.MAX_VALUE - sum) < balance)
f3	0 < balance
f4	balance == 0
f5	balance == (@balance + sum)
f6	reads sum
f7	reads Integer.MAX_VALUE

Рис. 4. Отчет о тестовом покрытии.

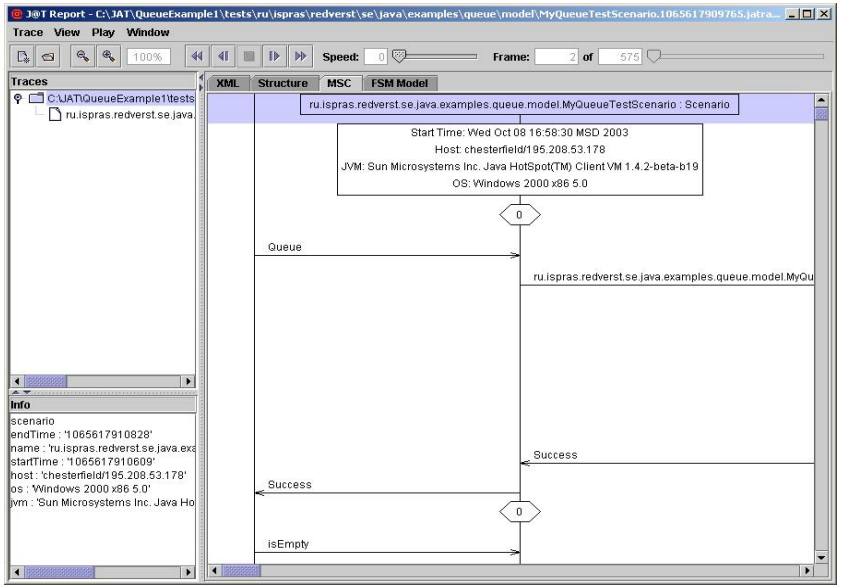


Рис. 5. Трасса в форме MSC.

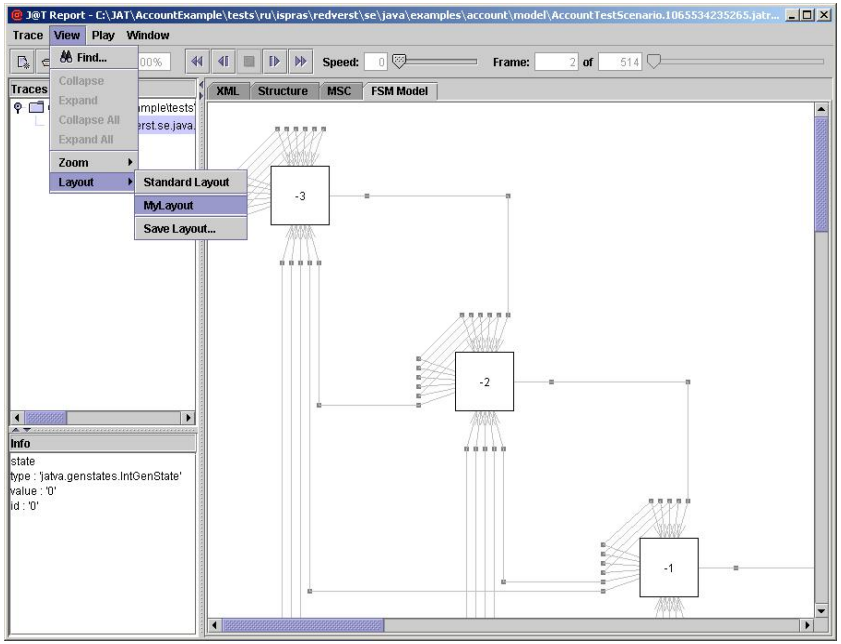


Рис. 6. Трасса в виде графа состояний конечного автомата.

3. Сравнение с другими подходами к разработке тестов на основе моделей

Хотя практически любая функциональная характеристика технологии UniTesK может быть найдена и в других технологиях и методах разработки тестов, иногда даже в более развитой форме, ни один из имеющихся подходов к разработке тестов, предлагаемый в академическом сообществе или в индустрии разработки ПО, не обладает всей совокупностью характеристик UniTesK.

В кратком обзоре, помещенном в этом разделе, мы сконцентрировали внимание на методах разработки тестов, поддержанных инструментами и нацеленных на использование в промышленной разработке ПО. Таким образом, множество интересных методов и техник осталось за рамками обзора.

Имеющиеся подходы к разработке тестов, в основном, используют стандартную, сложившуюся еще несколько десятилетий назад *архитектуру теста*. Тест в ней представляет собой набор тестовых вариантов (test cases), каждый из которых служит для проверки некоторого свойства целевой системы в определенной ситуации. В UniTesK тесты строятся в виде сценариев, каждый из которых, по существу, исполняет роль целого набора тестовых вариантов, проверяющих работу целевой системы при обращении к выделенной группе интерфейсов в различных ситуациях. В результате, в тестовом наборе UniTesK больше уровней иерархии, что удобно при тестировании больших и сложных систем. С другой стороны, тестовые варианты позволяют более эффективно воспроизводить нужные ситуации при повторном тестировании, например, на наличие ранее обнаруженной ошибки. Для регрессионного тестирования обе схемы пригодны в равной степени, поскольку при этом обычно требуется возможность прогона всего набора тестов.

Автоматическое построение тестовых оракулов на основе спецификаций отличает UniTesK от инструментов, таких как JUnit [JUNIT], автоматизирующих только выполнение тестов. Вместе с тем, оно поддерживается очень многими существующими инструментами, например, следующими:

- iContract [iContract,iContractW], JMSAssert [JMS], JML [JML,JMLW], jContractor [jContr,jContrW], Jass [Jass,JassW], Handshake [Handshake], JISL [JISL] используют контрактные спецификации, написанные в исходном коде целевой системы в виде комментариев на расширении Java (обзоры таких систем можно найти в [CCNET] и [ORA])
- SLIC [SLIC] позволяет оформлять контрактные спецификации на расширении C с использованием предикатов временных логик
- Test RealTime [TRT] от Rational/IBM использует контракты и описание структуры конечно-автоматной модели целевого компонента в виде специальных скриптов

- JTest/JContract [PARASOFT] от Parasoft и Korat [KORAT] позволяют писать предусловия, постусловия и инварианты в виде особых комментариев в Java-программах
- ATG-Rover [TRover] использует спецификации в виде программных контрактов-комментариев на C, Java или Verilog, которые могут содержать предикаты временных логик LTL или MTL
- Семейство инструментов ADL [ADLt] основано на расширениях C, C++, Java и IDL, которые используются для разработки контрактных спецификаций, не привязанных жестко к конкретному коду
- T-VEC [TVEC] использует пред- и постусловия, оформленные в виде таблиц в нотации SCR [SCR]

От *инструментов*, перечисленных в первых трех пунктах, UniTesK отличает наличие существенной поддержки разработки тестов, в частности, определение критериев покрытия на основе спецификаций и механизм генерации тестовых последовательностей из сценариев. В инструменте JTest возможность автоматической генерации тестовых последовательностей заявлена, но генерируемые последовательности могут содержать не более 3-х вызовов операций, и строятся случайным образом, без возможности нацелить их на достижение высокого тестового покрытия.

Инструмент Korat является одним из инструментов, разработанных в рамках проекта MulSaw [MulSaw] лаборатории информатики MIT. Он использует контракты, оформленные на JML, для генерации множества наборов входных данных одного метода в классе Java, включая и сам объект, в котором данный метод вызывается, гарантирующего покрытие всех логических ветвлений в спецификациях. Таким образом, вместо построения тестовой последовательности можно сразу получить целевой объект в нужном состоянии. С другой стороны, спецификации должны быть жестко привязаны к реализации. В частности, они не должны не допускать таких состояний целевого компонента, которые не могут возникнуть в ходе его работы, иначе много сгенерированных тестов будут соответствовать недостижимым состояниям компонента.

Инструменты ADL предоставляют поддержку разработки тестов только в виде библиотеки генераторов входных данных, аналогичной библиотеке итераторов в UniTesK. ATG-Rover позволяет автоматически генерировать шаблоны тестовых последовательностей для покрытия спецификаций. Из доступной документации неясно, должны ли эти шаблоны дорабатываться вручную, чтобы превратиться в тестовые последовательности, но возможность такой доработки присутствует.

T-VEC использует специальный вид спецификаций для автоматического извлечения информации о граничных значениях областей, в которых описываемая спецификациями функция ведет себя "одинаково" (ср. определение ветвей функциональности в UniTesK). Тестовые воздействия генерируются таким образом, чтобы покрывать граничные точки ветвей

функциональности для данной функции. Полный тест представляет собой список пар, первым элементом которых является набор аргументов тестируемой операции, а вторым --- корректный результат ее работы на данном наборе аргументов, вычисленный по спецификациям. Генерация тестовых последовательностей не поддерживается.

Кроме T-VEC, нам не известны инструменты, поддерживающие, подобно UniTesK, генерацию тестов, нацеленных на достижение высокого покрытия по критериям, построенным по внутренней структуре контрактных спецификаций. Большинство имеющихся инструментов способно отслеживать покрытие спецификаций только как процент операций, которые были вызваны.

Генерация тестовых последовательностей поддерживается многими инструментами, использующими модели целевой системы в виде различного рода автоматов: расширенных конечных автоматов, взаимодействующих конечных автоматов, автоматов ввода/вывода, систем помеченных переходов, сетей Петри и пр. Такие инструменты хорошо подходят для верификации телекоммуникационного ПО, при разработке которого зачастую используются формальные языки спецификаций, основанные на перечисленных представлениях ПО --- SDL [SDL,ITUSDL,ITUSDLN], LOTOS [LOTOS], Estelle [Estelle], ESTEREL [ESTEREL,ESTERELL] или Lustre [Lustre]. Большинство этих инструментов использует в качестве спецификаций описание поведения системы на одном из указанных языков, трансформируя его в автоматную модель нужного вида.

Часть таких инструментов использует, помимо спецификаций поведения системы, сценарий тестирования, называемый обычно *целью теста (test purpose)* и заданный пользователем в виде последовательности сообщений, которой обмениваются компоненты ПО (MSC), или небольшого автомата (см., например, [TPA,TPB,CADPO,TGV]). Другая часть использует явно описанные автоматные модели для генерации тестовых последовательностей, нацеленных на достижение определенного уровня покрытия согласно какому-либо критерию (см. [TorX,SDLt,EST]). UniTesK, как уже говорилось, поддерживает построение тестовых последовательностей и из заданных пользователем сценариев, и на основе автоматной модели системы, интегрируя оба подхода.

Наиболее близки к UniTesK по поддерживаемым возможностям инструменты GOTCHA-TCBeans [UMBTG,GOTCHA] (один из инструментов генерации тестов, объединяемых в рамках проекта AGEDIS [AGEDIS,AGEDISW]), и AsmL Test Tool [ASMT,ASMTW]. Оба они используют автоматные модели целевого ПО. Для GOTCHA-TCBeans такая модель должна быть описана на расширении языка Murphi [Murphi], AsmL Test Tool использует в качестве спецификаций описание целевой системы как машины с абстрактным состоянием (abstract state machine, ASM, см. [ASMI,ASMB]).

Объединяет все три подхода использование *разных видов моделей* для построения теста, что позволяет строить более эффективные, гибкие и масштабируемые тесты, а также иметь больше компонентов для повторного

использования. В UniTesK это модель поведения в виде спецификаций и модель тестирования в виде сценария, в GOTCHA-TCBeans и других инструментах проекта AGEDIS, — автоматная модель системы и набор тестовых директив, управляющих процессом создания тестов на ее основе, в последних версиях AsmL Test Tool — ASM-модель системы и множество наблюдаемых величин, наборы значений которых определяют состояния конечного автомата, используемого для построения тестовой последовательности.

В указанных инструментах используются техники уменьшения размера модели, аналогичные факторизации в UniTesK. Инструмент GOTCHA-TCBeans может применять частный случай факторизации, при котором игнорируются значения некоторых полей в состоянии исходной модели [PROJ]. AsmL Test Tool может строить тестовую последовательность на основе конечного автомата, состояния которого получаются редукцией полного состояния машины до набора значений элементарных логических формул, используемых в описании ее переходов [FfASM].

Основными отличиями UniTesK от GOTCHA-TCBeans и AsmL Test Tool являются поддержка расширений языков программирования для разработки спецификаций, использование контрактных спецификаций, автоматизация отслеживания покрытия спецификаций и использование фильтров для получения тестовых воздействий, нацеленных на его повышение.

4. Опыт использования UniTesK

Работы над инструментами для поддержки технологии UniTesK начались в конце 1999 года. К тому времени уже было проведено несколько проектов, где технология KVEST использовалась для разработки тестов и регрессионного тестирования крупных промышленных систем компании Nortel Networks, производителя цифрового телекоммуникационного оборудования. Примерами систем Nortel Networks, к которым применялся KVEST были:

- Ядро операционной системы реального времени (размер около 250 тысяч строк, язык реализации ПРОТЕЛ, близок к C).
- Система хранения и быстрого поиска сообщений.
- Утилиты базисного уровня телефонных сервисов.
- Распределенная система управления сообщениями P2P.

Первой была готова для использования реализация UniTesK для программ на языке C — CTesK. Упрощенный вариант инструментов для поддержки этой технологии был готов к концу 2000 года. К середине 2001 они уже активно использовались в исследовательском проекте по гранту Microsoft Research. Целью проекта было тестирование реализации стека интернет-протоколов нового поколения. IPv6. Кроме того, ставилась задача проверки пригодности UniTesK для тестирования задач нового класса — реализаций телекоммуникационных протоколов. К концу 2001 года проект был завершен.

Были найдены серьезные ошибки в реализации, способные привести к нарушению работы произвольного узла в сети на основе IPv6. Поскольку данная реализация тестировалась одновременно несколькими командами тестировщиков в разных странах мира, и никто не обнаружил этих ошибок, была показана высокая результативность UniTesK. В рамках данного проекта был впервые опробован новый механизм тестирования распределенных и асинхронных систем на основе автоматных моделей в виде IOSM.

В 2003 году был проведен пилотный проект применения CTesK для тестирования ПО реального времени (алгоритмы управления навигационными устройствами, ГосНИИАС). Несмотря на сжатые сроки проекта удалось найти серьезную ошибку в алгоритме.

В середине 2002 года была готова первая коммерческая версия инструмента J@T, инструмента разработки тестов для Java программ[*]. По состоянию на конец 2003 кода проведено несколько пилотных проектов с использованием J@T:

- Система ведения информации по клиентам и сделкам для банка, построенная по трехзвенной архитектуре с использованием технологии EJB (Luxoft).
- Подсистема создания запросов и преобразования данных в распределенном банковском Web-приложении (Tarang)
- Подсистема связи с СУБД в библиотеке классов для разработки многоуровневых бизнес-приложений (VisualSoft)
- Подсистема контроля времени работы клиентов в биллинговой системе (VebTel).

Все перечисленные пилотные показали высокую результативность технологии. В рамках всех проектов удалось найти ошибки Часть ошибок владельцы кода определили как очень серьезные. Например, была найдена ситуация, когда результаты незавершенных транзакций сбрасывались в базу данных. При этом некоторые из проектов проводились в чрезвычайно сжатые сроки — 1-2 дня.

В 2001 группа RedVerst начала совместные научные исследования с компанией Интел. Целью исследований была адаптация технологии тестирования на основе моделей к задачам тестирования оптимизирующих компиляторов. Сначала эксперименты проводились на свободно распространяемых компиляторах (gcc и Open64). Уже первые опыты показали, что для компилятора gcc, например, удается добиться полноты тестового покрытия операторов реализации около 90-95%, что для промышленного тестирования является очень высоким показателем. Затем опыты были перенесены на последнее поколение коммерческих компиляторов, разрабатываемых Интел для 64-битной платформы Itanium. Результативность адаптированной технологии UniTesK была подтверждена и там.

К середине 2003 года на основе опыта проведенных исследований и экспериментов был разработан набор программных инструментов, который

[*] Имеется возможность использования J@T для тестирования программ на C++.

позволяет строить тесты для компиляторов и других текстовых процессоров. Эти инструменты были опробованы на задачах генерации пакетов протокольных сообщений.

Надо отметить, что «адаптация» UniTesK к задачам тестирования компиляторов потребовала решения ряда принципиально новых задач и разработки новых технологических приемов. Так были решены

- задача описания модельного языка, являющегося упрощением целевого языка компилятора,
- задача построения итераторов конструкций модельного языка — модельных блоков,
- задача построения тестового оракула для анализа корректности оптимизаторов.

Инструмент для тестирования компиляторов вошел в набор инструментов для поддержки разработки тестов по технологии UniTesK. Тем самым было показано, что концептуально схема построения тестов на основе моделей оказалась достаточно общей.

К концу 2003 года была завершена первая версия инструмента Ch@se, инструмента разработки тестов для .NET приложений. Это очередная реализация технологии UniTesK, теперь на базе расширения языка C#.

Полный список проектов, проводимых с использованием технологии UniTesK, а также описания результатов этих проектов, можно найти на сайте группы спецификации, верификации и тестирования ИСП РАН [RedVerst].

5. Заключение. Открытые проблемы и направления дальнейшего развития

Опыт разработки и использования UniTesK, как и работы других исследователей в области тестирования на основе моделей, показывает, что данное направление является весьма перспективным. Каковы же главные проблемы, которые мешают более широкому распространению тестирования на основе моделей? Их можно разделить на три основные группы:

- методические — как разрабатывать спецификации и тесты?
- технические — как унифицировать методы разработки спецификаций и тестов и инструменты, которые поддерживают эти работы?
- организационные — как внедрить новые методики и инструменты в реальные процессы разработки ПО?

Сложность успешного решения перечисленных проблем возрастает от первой группы (методика) к последней (организационная перестройка). В подтверждение этого тезиса можно сослаться на опыт многих групп, использующих методы тестирования на основе спецификаций в промышленных приложениях. За исключением единичных случаев всегда удается найти подходящий способ моделирования, который дает хорошие

результаты как в плане качества тестирования, так и плане достижения хороших стоимостных показателей. Тем самым показано, что неразвитость теории и методики не является сдерживающим фактором распространения данного подхода.

Сейчас еще нет общего согласия по поводу нотаций и методов, при помощи которых ведется разработка моделей и тестов на их основе, нет унифицированной, общепринятой архитектуры тестирующей системы*, отдельные инструменты для статического и динамического анализа программ и представления результатов анализа еще далеки от унификации. Тем не менее потребность в унификации уже назрела. Наиболее известным продвижением в области унификации является разработка документа UML Testing Profile [UML-TP], подготовленного в рамках программы MDA (Model Driven Architecture, [MDA]). Вместе с тем следует отметить, что авторы UML Testing Profile, возможно, пытаясь найти компромисс пригодный для всех участников консорциума, предлагают архитектуру, нацеленную больше на ручную разработку тестов. Этот подход в некоторой степени может быть использован при разработке тестов на основе спецификаций сценариев использования, но он становится ограничивающим фактором, если в качестве модели целевого ПО берутся автоматные модели или спецификации ограничений, разработанные по методу Design-by-Contract. Следствием такого одностороннего подхода является непонимание потенциальных преимуществ использования спецификаций ограничений, и, в частности, языка OCL (Object Constraint Language), в промышленном программировании. Тем самым, проблема унификации представляется не только технической. Важно найти унифицированный подход, который позволил бы на базе единой концепции разрабатывать тесты на основе разных видов моделей с использованием ручных, автоматизированных и автоматических технологий.

Организационные проблемы на данный момент являются наиболее острыми [Robinson, Manage]. Как уже отмечалось выше, новая технология тестирования должна достаточно хорошо интегрироваться с имеющимися процессами разработки, и не требовать долгой и дорогой переподготовки персонала. Есть ли шанс найти для методов разработки тестов на основе моделей возможности сочетаться с традиционными методами разработки таким образом, что это позволит ввести новые технологии без ломки уже сложившихся процессов?

По-видимому, абсолютно безболезненным внедрение не будет, и причиной этого являются не столько технические проблемы, сколько проблемы персонала, причем как технического, так и управляющего. В настоящее время при промышленной разработке ПО широко используется подход к разработке тестов и тестированию как вспомогательной деятельности, не требующей

* Верификационного набора в терминах UniTesK — совокупности компонентов модели и производных от них компонентов тестирующей программы, рассматриваемой вместе со связями между всеми компонентами.

навыков программирования, соответственно, в качестве тестировщиков используется, в основном, персонал без знания языков программирования и без базовых программистских умений. Пытаясь упростить проблемы организации взаимодействия команд разработчиков и тестировщиков, руководители проектов ориентируют тестировщиков только на системное тестирование, что влечет практическое отсутствие модульного и компонентного. В этом случае появляется иллюзия, что тестировщики, имея большой запас времени, могут создать качественные тесты. В реальности же разработка обычно затягивается, и к моменту приемо-сдаточных испытаний система в состоянии продемонстрировать работоспособность на ограниченном количестве наиболее вероятных сценариев, но нуждается в серьезных доработках на большинстве нестандартных сценариев использования.

Очевидно, что «сломать» негативные тенденции нельзя, они могут быть изжиты только вследствие эволюционных изменений. Сейчас такими эволюционными шагами могут стать дополнительные работы, которые поручаются проектировщикам и разработчикам. Эти дополнительные работы войдут в обиход реальных процессов разработки только тогда, когда их выполнение окажет положительный эффект на основную деятельность проектировщиков и программистов. Поэтому сейчас средства автоматизации тестирования на основе моделей имеет смысл вписывать в среду разработки, тем самым привлекая к тестированию не только профессиональных тестировщиков, но и разработчиков.*

Для того чтобы расширить сообщество пользователей методов тестирования на основе моделей надо развивать формы обучения этим методам. Все инструменты UniTesK сопровождаются не только обычной пользовательской документацией, но и материалами для обучения. Предлагаются две формы обучения: традиционная университетская форма и форма интенсивного тренинга. Университетский курс требует от 15 до 30 часов занятий, тренинг требует 3-4 учебных дня по 8 академических часов каждый день. Материалы для университетов также как и лицензии на инструменты UniTesK для использования в образовательных целях предоставляются бесплатно. Также бесплатно предоставляются примеры спецификаций и тестов для разнообразных приложений.

Основной стратегической целью дальнейших работ над UniTesK является построение полномасштабных индустриально применимых технологий тестирования на основе моделей для всех видов приложений, разрабатываемых в промышленности.

Для успешного развития технологий тестирования на основе моделей нужно одновременно поддерживать их эволюцию в трех направлениях: наращивать

* Многие считают, что разработчик не в состоянии тестировать свою программу. Однако, в случае использования моделей по структуре отличающихся от реализации (например, спецификаций ограничений), такое совмещение вполне возможно.

функциональность технологий, наращивать сопряженность этих технологий с современными процессами разработки и наращивать удобство их использования. Особенно важно увеличивать удобство использования технологий для решения наиболее часто встречающихся задач.

Исследования по расширению области применимости подхода к тестированию на основе моделей должны проводиться по целому ряду направлений

1. Разработка полномасштабных технологий тестирования для всех компонентов таких видов приложений, как компиляторы, интерпретаторы, СУБД и др., функции которых связаны с обработкой запросов на хорошо структурированных формальных языках.
2. Автоматизация тестирования приложений через графический пользовательский интерфейс.
3. Разработка технологий полномасштабного тестирования распределенных систем, из которых наиболее широко встречаются сейчас многоуровневые приложению в архитектуре клиент-сервер приложений-сервер данных.
4. Тестирование компонентов, имеющих очень сложную функциональность и очень ограниченный интерфейс: планировщики задач, сборщики мусора, менеджеры транзакций и пр.
5. Тестированию приложений с элементами искусственного интеллекта: распознавание образов, выбор стратегии достижения цели в неопределенной ситуации, нечеткая логика, интеллектуальные агенты, и пр.

Помимо решения задач собственно тестирования для привлечения внимания к технологиям тестирования на основе моделей необходимо искать подходящие метрики качества ПО, которые позволили бы продемонстрировать преимущества этого подхода. На данный момент многие метрики, используемые в промышленности для оценки качества ПО и состояния процесса тестирования, подходят для традиционной разработки тестов вручную, но приводят к парадоксам при попытке их использования в проектах, где тесты строятся автоматизировано на основе моделей. Кроме того, необходимо искать пути повышения прозрачности связей между используемыми моделями и требованиями к ПО, а также метрики переиспользуемости, позволяющие объективно оценить влияние изменений в требованиях к ПО, в требованиях к качеству тестирования, в архитектуре целевого ПО, в используемых технологиях разработки и пр. на изменения в тестах. Разработка и апробация таких метрик в промышленных проектах позволит повысить значимость технологий, нацеленных на повышение качества ПО, что, в свою очередь придаст новый импульс работам по автоматизации тестирования и по тестированию на основе моделей.

Литература

1. [ADLt] M. Obayashi, H. Kubota, S. P. McCarron, and L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.opengroup.org/

2. [AGEDIS] I. Gronau, A. Hartman, A. Kirshin, K. Nagin, and S. Olvovsky. A Methodology and Architecture for Automated Software Testing. Available at http://www.haifa.il.ibm.com/projects/verification/gtcb/papers/gtcbmanda.pdf
3. [AGEDISW] http://www.agedis.de/
4. [ASMB] E. Börger and R. Stark. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.
5. [ASMI] Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In Current Trends in Theoretical Computer Science, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, pp. 266–292.
6. [ASMT] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Testing with Abstract State Machines. In R. Moreno-Diaz and A. Quesada-Arencibia, eds., Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001), Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001, pp. 257–261.
7. [ASMTW] http://research.microsoft.com/fse/asml/
8. [AsSM] И. Бурдонов, А. Косачев, В. Кулямин. Асинхронные автоматы: классификация и тестирование, Труды ИСП РАН, 4:7-84, 2003.
9. [ASSUM] S. Fujiwara and G. von Bochmann. Testing Nondeterministic Finite State Machine with Fault Coverage. IFIP Transactions, Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems, 1991, Ed. by Jan Kroon, Rudolf J. Heijink, and Ed Brinksma, 1992, North-Holland, pp. 267–280.
10. [ATS] http://www.atssoft.com
11. [BP] G. v. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In Proceedings of ACM International Symposium on Software Testing and Analysis. Seattle, USA, 1994, pp. 109–123.
12. [CADPO] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. INRIA Technical Report TR-254, December 2001.
13. [CCNET] M. Barnett and W. Schulte. Contracts, Components, and their Runtime Verification on the .NET Platform. Technical Report TR-2001-56, Microsotf Research.
14. [DBCA] Bertrand Meyer. Applying `Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992, pp. 40–51.
15. [DBCE] Bertrand Meyer. Eiffel: The Language. Prentice Hall, 1992.
16. [DBCO] Bertrand Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall, 1997.
17. [DETFSM] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. Программирование, 2003.
18. [EST] W. Chun, P. D. Amer. Test case generation for protocols specified in Estelle. In J. Quemada, J. Mañas, and E. Vázquez, editors. Formal Description Techniques, III, Madrid, Spain, North-Holland 1990, pp. 191–206.
19. [Estelle] ISO/TC97/SC21. Information Processing Systems — Open Systems Interconnection — Estelle — A Formal Description Technique based on an Extended State Transition Model. ISO 9074:1997, International Organization for Standardization, Geneva, Switzerland, 1997.
20. [ESTEREL] G. Berry. The Foundations of Esterel. In Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, 1998.
21. [ESTERELL] F. Boussinot and R. de Simone. The Esterel language. Proc. IEEE, vol. 79, pp. 1293–1304, Sept. 1991.

22. [FACTOR] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Применение конечных автоматов для тестирования программ. Программирование, 26(2):61–73, 2000.
23. [FfASM] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In Proc. of ISSTA'2002. Also: Microsoft Research Technical Report MSR-TR-2001-97.
24. [FMEAD] http://www.fmeurope.org/databases/fmadb088.html
25. [GOTCHA] http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html
26. [Handshake] A. Duncan and U. Hlzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, 1998.
27. [iContract] R. Kramer. iContract — The Java Design by Contract Tool. In Proceedings of TOOLS26: Technology of Object-Oriented Languages and Systems, pp. 295–307. IEEE Computer Society, 1998.
28. [iContractW] http://www.reliable-systems.com/
29. [IOSMA] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand. Towards Analysing and Synthesizing Protocols. IEEE Transactions on Communications, COM-28(4):651–660, April 1980.
30. [ITUSDL] ITU-T, Recommendation Z.100: Specification and Description Language (SDL), ITU-T, Geneva, 1996.
31. [ITUSDLN] ITU-T, Recommendation Z.100 Annex F1: SDL formal definition — General, 2000.
32. [Jass] D. Bartetzko, C. Fisher, M. Moller, and H. Wehrheim. Jass — Java with assertions. In K. Havelund and G. Rosu, editors, Proceeding of the First Workshop on Runtime Verification RV'01, Vol. 55 of Electronic Notes in Theoretical Compter Science, Elsevier Science, July 2001.
33. [JassW] http://semantik.informatik.uni-oldenburg.de/~jass
34. [Jatva] Igor B. Bourdonov, Alexey V. Demakov, Andrew A. Jarov, Alexander S. Kossatchev, Victor V. Kuliamin, Alexander K. Petrenko, Sergey V. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'01. LNCS 2244, pp. 301–307. Springer-Verlag, 2001.
35. [jContr] M. Karaorman, U. Holzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. Technical Report TRCCS98-31, University of California, Santa Barbara. Computer Science, January 19, 1999.
36. [jContrW] http://jcontractor.sourceforge.net/
37. [JISL] P. Muller, J. Meyer, and A. Poetzsch-Heffter. Making executable interface specifications more expressive. In C. H. Cap, editor, JIT'99 Java-Informations-Tage 1999, Informatik Aktuell. Springer-Verlag, 1999.
38. [JML] A. Bhorkar. A Run-time Assertion Checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 2000.
39. [JMLW] http://www.cs.iastate.edu/~leavens/JML.html
40. [JMS] http://www.mmsindia.com/JMSAssert.html
41. [JUNIT] http://www.junit.org/index.htm
42. [KORAT] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. Proc. of ISSTA 2002, Rome, Italy. Jul 2002.
43. [KVEST] I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999, pp. 608–621.
44. [LOTOS] ISO/IEC. Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique based on the Temporal Ordering of

Observational Behaviour. ISO/IEC 8807:1989, International Organization for Standardization, Geneva, Switzerland, 1989.
45. [Lustre] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. Proc. IEEE, vol. 79, pp. 1305–1320, Sept. 1991.
46. [LY] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite-State Machines. A survey. Proceedings of the IEEE, Vol. 84, No. 8, 1996, pp. 1090–1123.
47. [MCDC] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, pp. 193–200, September 1994.
48. [MSRIPreport] http://www.ispras.ru/~RedVerst/RedVerst/White Papers/MSRIPv6 Verification Project/Main.html
49. [MulSaw] http://mulsaw.lcs.mit.edu/
50. [Murphi] http://verify.stanford.edu/dill/murphi.html
51. [NDFSM] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. Программирование, 2003. В этом номере.
52. [Manage] D.Stidolph, J.Whitehead. Managerial Issues for the Consideration and Use of Formal Methods, LNCS No.2805, 2003, pp.170-186.]
53. [MDA] http://www.omg.org/mda/
54. [OPT] A. Kossatchev, A. Petrenko, S. Zelenov, and S. Zelenova. Using Model-Based Approach for Automated Testing of Optimizing Compilers. In Proccedings of Intl. Workshop on Program Undestanding, Gorno-Altaisk, 2003.
55. [ORA] L. Baresi and M. Young. Test Oracles. Tech. Report CIS-TR-01-02. Available at http://www.cs.uoregon.edu/~michal/pubs/oracles.html
56. [PARASOFT] http://www.parasoft.com
57. [Parnas] D. Peters and D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Transactions on Software Engineering, 24(3):161–173, 1998.
58. [PROJ] G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected state machine coverage for software testing. Proc. of ISSTA 2002, Rome, Italy. Jul 2002.
59. [RedVerst] http://www.ispras.ru/groups/rv/rv.html
60. [Robinson] H. Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. In proceedings of 1-st ECMDSE, 2003
61. [SCR] C. Heitmeyer. Software Cost Reduction. Encyclopedia of Software Engineering, Two Volumes, John J. Marciniak, editor, ISBN: 0-471-02895-9, January 2002.
62. [SDL] J. Ellsberger, D. Hogrefe, and A. Sarma, SDL — Formal Object-Oriented Language for Communicating Systems, Prentice Hall, 1997.
63. [SDLt] C. Bourhfir, E. Aboulhamid, R. Dssouli, N. Rico. A test case generation approach for conformance testing of SDL systems. Computer Communications 24(3-4): 319–333 (2001).
64. [SLIC] T. Ball and S. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report, MSR-TR-2001-21, Microsoft Research, January 2002.
65. [TGV] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. In Special Issue on Industrially Relevant Applications of Formal Analysis Techniques, J. F. Groote and M. Rem, editors, Elsevier Science publisher, 1996.
66. [TorX] J. Tretmans, A. Belinfante. Automatic testing with formal methods. In EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis and Review, Barcelona, Spain, November 8-12, 1999. EuroStar Conferences, Galway, Ireland. Also:

Technical Report TRCTIT-17, Centre for Telematics and Information Technology, University of Twente, The Netherlands.
67. [TPA] J. Grabowski, D. Hogrefe, R. Nahm. Test case generation with test purpose specification by MSCs. In O. Faergemand and A. Sarma, editors, 6th SDL Forum, pages 253–266, Darmstadt, Germany, North-Holland 1993.
68. [TPB] C. J. Wang, M. T. Liu. Automatic test case generation for Estelle. In International Conference on Network Protocols, pages 225–232, San Francisco, CA, USA, 1993.
69. [TRover] http://www.time-rover.com
70. [TRT] http://www.rational.com
71. [TVEC] http://www.t-vec.com
72. [UMBTG] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. IBM Systems Journal, volume 41, Number 1, 2002, pp. 89–110.
73. [UML-TP] UML Testing Profile. http://www.omg.org/docs/ptc/03-07-01.pdf
74. [UniArch] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.
75. [UNITESK] http://unitesk.ispras.ru