ПРОГРАММИРОВАНИЕ, 2003, № 6, с. 25–43

ТЕСТИРОВАНИЕ, ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ПРОГРАММ

УДК 681.3.06

ПОДХОД UniTesK К РАЗРАБОТКЕ ТЕСТОВ*

© 2003 г. В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов
Институт системного программирования РАН
109004 Москва, ул. Б. Коммунистическая, 25
E-mail: igor@ispras.ru, kos@ispras.ru, kuliamin@ispras.ru
Поступила в редакцию 09.06.2003 г.

Статья излагает основные принципы технологии разработки тестов UniTesK, основанной на использовании формальных моделей целевого программного обеспечения (ПО). Данная технология была разработана в группе спецификации, верификации и тестирования [1] ИСП РАН на основе многолетнего опыта проведения верификации и тестирования сложного промышленного ПО.

1. ВВЕДЕНИЕ

В настоящее время производство программного обеспечения (ПО) достигло таких масштабов и такой степени сложности, что общепризнана необходимость в индустриально применимых технологиях тестирования. Актуальной задачей является создание таких технологий, которые обеспечивают одновременно качественное, систематическое тестирование целевого ПО и высокую степень автоматизации разработки тестов. Традиционные же методы разработки тестов вручную уже не годятся для качественного тестирования больших и сложных 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, собственно, и должно быть проверено. Требования же в лучшем случае представлены в виде неформальных документов, а иногда – только как знания и опыт аналитиков, проектировщиков и разработчиков ПО.

Для того, чтобы вовлечь требования в автоматизированный процесс разработки тестов, необходимо перевести их в формальное представление, которое может восприниматься и обрабатываться машиной. Для этой цели требования описывают в виде *формальных спецификаций* целевой системы, которые уже в принципе возможно преобразовать в программы, выполняющие проверку соответствия работы целевого ПО требованиям.

Несмотря на активное развитие методов построения тестов на основе формальных спецификаций в академическом сообществе, лишь немногие из них оказываются применимыми в индустриальном производстве ПО. Основная проблема здесь в том, что индустрии нужны не отдельные методы, а именно *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 нефункциональных требований выходят за рамки данной работы.

Структура статьи такова. Следующий за введением раздел содержит описание основных

*Работа выполнена в рамках Государственного контракта N 10002-251/П-21/019-027/101043-573 и поддержана грантом РФФИ 02-01-00959.

элементов технологии UniTesK, начинаясь общим обзором ее базовых принципов и дальше раскрывая некоторые из них в деталях. В третьем разделе UniTesK сравнивается с другими подходами к разработке тестов на основе моделей. И, наконец, заключительный раздел кратко описывает примеры приложений UniTesK и рассказывает о направлениях дальнейшего развития этой технологии.

2. ОПИСАНИЕ ТЕХНОЛОГИИ UniTesK

2.1. *Основные принципы UniTesK*

Мы убеждены, что 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, и для тестирования ПО из разных предметных областей. 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.

- Для обеспечения максимальной гибкости 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.

- Чтобы сделать возможной значительную степень автоматизации, в рамках полученной архитектуры вся информация, которая должна быть предоставлена разработчиком теста, 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.

- В качестве метамодели для представления функциональных спецификаций, моделирующих требования, был выбран широко известный подход на основе *программных контрактов* (Design by Contract [2–4]), состоящих из *предусловий* и *постусловий* интерфейсных операций и *инвариантов* типов данных. Программные контракты, с одной стороны, достаточно удобны для проектировщиков и разработчиков, поскольку хорошо привязываются к архитектуре ПО, с другой стороны, стимулируют усилия по созданию независимых от реализации критериев корректности целевой системы. Основное же их преимущество в том, что они позволяют автоматически построить *оракулы* [5–7], проверяющие соответствие поведения целевой системы спецификациям, и *критерии тестового покрытия,* которые достаточно близки к критериям покрытия требований.

- 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 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 из них, которые не увеличивают достиг-

нутый уровень покрытия. Фильтры для этого генерируются автоматически из определения критерия тестового покрытия (см. пункт после следующего).

- 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 некоторому маршруту 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.

- Стратегия тестирования должна определять, когда тестирование можно заканчивать. UniTesK предлагает при этом опираться на достигнутый уровень тестового покрытия в соответствии с некоторым критерием покрытия. Из структуры функциональных спецификаций, разработанных в соответствии с технологией UniTesK, можно автоматически извлечь несколько таких критериев. Пользователь имеет возможность гибко управлять этими критериями или определять свои собственные.

- 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#.

- 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. Тогда возможно использование специализированных вариантов технологии и поддерживающих их инструментов.

Например, при тестировании блоков оптимизации в компиляторах разработка спецификаций функциональности такого блока в полном объеме очень трудоемка, поскольку они должны, например, выражать тот факт, что оптимизация программы 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 [8].

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 использования для работы с ПО, относящимся 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 [9] 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 [6, 7]. 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 механизмов 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. Такая модель предполагает, что зависимость поведения системы от истории ее взаимодействия с окружением можно свести к его зависимости от текущего внутреннего состояния системы, изменяющегося в ответ на обращения к ней, причем множество достижимых состояний конечно. 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* [10], 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.

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).

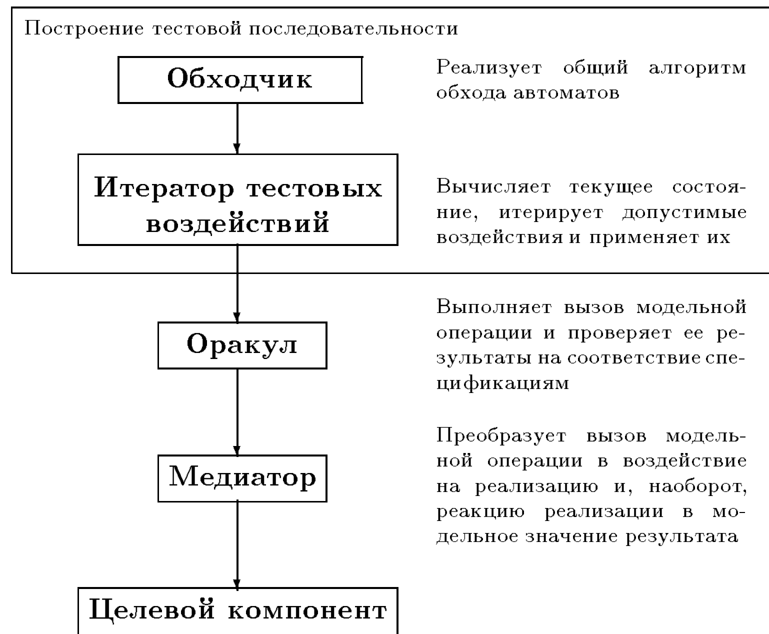

Рис. 1. Архитектура теста UniTesK.

Медиаторы удобно разрабатывать в расширении целевого языка, где можно 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 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 реализационными объектами, и пр. Эти вспомогательные компоненты не зависят от тестируемого ПО и выбранной стратегии тестирования.

2.3. Техника описания 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 типов 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 и, в то же время, применимы для ПО из очень многих предметных областей. Контрактные спецификации 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.

2.4. *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 детальный из таких критериев – критерий покрытия дизъюнктов – определяется всеми возможными комбинациями значений элементарных логических формул, использованных в этих ветвлениях. Он является аналогом критерия MC/DC [11] 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.

2.5. *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 этих проекций для всех тестируемых операций, мы получим набор таких множеств состояний, что, вызывая любые операции в двух состояниях из одного 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 состояниями результирующего автомата полученные множества состояний системы. Стимулами в таком автомате считаются классы эквивалентности вызовов операций по выбранному критерию покрытия, т.е. покрывающие один и тот же его элемент. Может потребоваться дополнительное преобразование полученного автомата, чтобы сделать его детерминированным, подробности см. в [12].

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 автоматных 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. [12]). 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 механизма наследования. Сценарий, наследующий данному, может переопределить 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 [71].

Без спецификаций асинхронных реакций можно тестировать системы, удовлетворяющие *аксиоме чистого параллелизма* (*plain concurrency axiom*): результат параллельного выполнения любого набора вызовов операций такой системы совпадает с результатом выполнения 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 [10]. 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, основан-

ное 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.

2.6. 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 применять 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 досто-

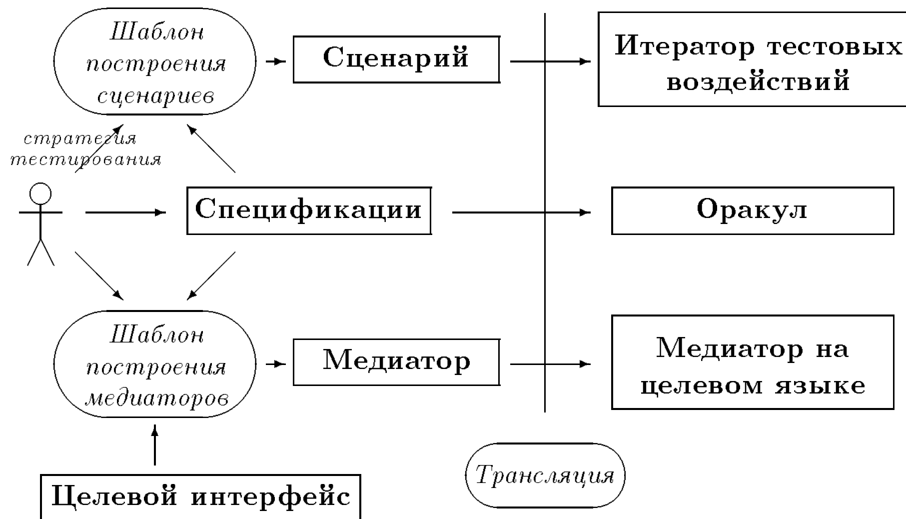

Рис. 2. Процесс разработки тестов UniTesK.

верной 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.

Рис. 2 представляет общую схему разработки тестов по технологии UniTesK. После разработки спецификаций, сценариев и медиаторов они транслируются, соответственно, в оракулы, итераторы тестовых воздействий и медиаторы на языке целевой системы, компонуясь в готовую к использованию тестовую программу.

2.7. 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, указате-

лей), 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 написания спецификаций состоит в том, что связывать такую спецификацию с реализацией гораздо проще. При использовании для написания спецификаций расширения целевого языка обучиться работе с ними может обычный разработчик, имеющий опыт работы с целевым языком. Проблема недостаточной выразительности в большинстве 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, рекомендовать к использованию 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.

2.8. *Используемые гипотезы и формальные заключения по результатам тестирования*

При проведении тестирования по технологии UniTesK в первую очередь происходит выявление структуры графа состояний конечного автомата, описанного неявно в тестовом сценарии. Дальнейшие действия определяются видом тестирования, его целями и видом используемых моделей на основе ряда теоретических результатов о тестировании конечных автоматов на соответствие [13–15]. Основной используемый результат, относящийся к простейшему случаю проверки на совпадение двух конечных детерминированных автоматов, один из которых представляет собой модель, а другой – реализацию, состоит в следующем.

Тестирование проводится при выполнении некоторого набора гипотез тестирования, а именно:

1. Множества стимулов и реакций модельного и реализационного автоматов совпадают.
2. Множества их состояний являются подмножествами одного множества S.
3. Их начальные состояния тоже совпадают.
4. В каждом состоянии, принадлежащем им обоим, множества допустимых стимулов одинаковы.

При самом тестировании проверяется, что в каждом достигнутом в ходе работы теста модельном состоянии для каждого допустимого в модели стимула реализация выдает реакцию, разрешенную в модели. Если при этом модельный автомат сильно связен, и помимо правильности реакций мы проверяем также тождественность состояний, в которые попадают оба автомата на каждом шаге (тестирование с открытым состоянием), то при отсутствии обнаруженных ошибок модельный и реализационный автоматы совпадают. Точнее говоря, совпадают их подавтоматы, достижимые из начальных состояний.

Отметим здесь разницу между тестированием с открытым состоянием и тестированием с

закрытым состоянием. При тестировании с открытым состоянием мы фактически точно знаем текущее реализационное состояние и можем проверить его соответствие модельному. В этом случае для того, чтобы сделать вывод, что все нужные переходы реализационного автомата проверены, достаточно просто обойти все переходы модельного.

При тестировании со скрытым состоянием, у нас нет достоверной информации о том, в каком именно реализационном состоянии мы оказываемся после очередного перехода. Для проверки всех переходов реализации при этом не достаточно обойти модельный автомат, и надо применять более сложные методы тестирования на соответствие, например, описанный в [13] метод, основанный на характеристических множествах последовательностей, или приведенный там же вероятностный алгоритм тестирования для этого случая.

При тестировании, допускающем недетерминизм обоих автоматов, нужно ввести дополнительную гипотезу, которая бы позволила сделать заключение о корректности работы недетерминированных переходов. Для этого можно использовать два вида гипотез. Гипотезы первого рода предполагают, что, воздействуя на реализационный автомат в некотором состоянии некоторым стимулом "достаточно много раз", мы получим и, соответственно, проверим все возможные по данному стимулу переходы (см., например, [16]). Гипотезы второго рода предполагают "однородность" недетерминизма системы относительно ошибок, т.е. наличие несоответствия реализации модели для одного из переходов из некоторого состояния по некоторому стимулу влечет ее наличие в реализации всех переходов по этому стимулу из этого же состояния [17].

Кроме того, в случае недетерминированности модели и реализации требование сильной связности модельного автомата надо заменить требованием сильной Δ-связности, обеспечивающей возможность построения обхода недетерминированного автомата (см. детали в [17]).

Приведенное выше допущение о том, что состояния модельного и реализационного автоматов принадлежат одному множеству или эквивалентное ему допущение о наличии взаимно однозначного соответствия между ними достаточно редко можно использовать на практике из-за различий в уровнях абстракции модели и реализации. В общем случае можно говорить о наличии соответствия между последовательностями стимулов и реакций в модельном и реализационном автоматах, даже не сводящегося естественным образом к соответствию между их состояниями, но соответствующий теоретический аппарат еще не полностью разработан.

В достаточно распространенном частном случае соответствие между моделью и реализацией может быть представлено в виде факторизации [12]. При этом существует отображение $\phi : S_I \to S_M$ множества состояний реализации в множество модельных состояний, при котором прообраз одного модельного состояния относительно ϕ может содержать несколько элементов. Для возможности проведения тестирования требуется гипотеза об "однородности" множеств допустимых стимулов относительно факторизации: стимулы, допустимые в модельном состоянии $s \in S_M$, должны быть допустимы во всех соответствующих реализационных состояниях $t \in \phi^{-1}s$. Для того, чтобы сделать заключение об отсутствии несоответствий между реализацией и моделью, требуется гипотеза об "однородности ошибок" относительно факторизации: если для некоторого модельного состояния s и некоторого модельного стимула x, определяющего переход, выдающий реакцию y и приводящий в состояние s' в модели, найдется реализационное состояние $t \in \phi^{-1}s$, такое, что переход по x в t не соответствует модели (т.е. выдает реакцию, не равную y, или приводит в состояние t', такое, что $\phi(t') \neq s'$), то и для всех $u \in \phi^{-1}s$ переходы по стимулу x из u не соответствуют модели.

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.

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 таких инструментов, как JUnit [18], автоматизирующих только выполнение тестов. Вместе с тем, оно поддерживается очень многими существующими инструментами, например, следующими:

- iContract [19, 20], JMSAssert [21], JML [22, 23], jContractor [24, 25], Jass [26, 27], Handshake [28], JISL [29] используют контрактные спецификации, написанные в исходном коде целевой системы в виде комментариев на расширении Java (обзоры таких систем можно найти в [30] и [31]).

- SLIC [32] позволяет оформлять контрактные спецификации на расширении C с использованием предикатов временных логик.

- Test RealTime [33] от Rational/IBM использует контракты и описание структуры конечно-автоматной модели целевого компонента в виде специальных скриптов.

- JTest/JContract [34] от Parasoft и Korat [35] позволяют писать предусловия, постусловия и инварианты в виде особых комментариев в Java-программах.

- ATG-Rover [36] использует спецификации в виде программных контрактов-комментариев на C, Java или Verilog, которые могут содержать предикаты временных логик LTL или MTL.

- Семейство инструментов ADL [7] основано на расширениях C, C++, Java и IDL, которые используются для разработки кон-

трактных спецификаций, не привязанных жестко к конкретному коду.

- T-VEC [37] использует пред- и постусловия, оформленные в виде таблиц в нотации SCR [38].

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 трех 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 [39] 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 [40–42], LOTOS [43], Estelle [44], ESTEREL [45, 46] или Lustre [47]. 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 *замыслом теста* (*test purpose*) и заданный пользователем в виде последовательности сообщений, которой обмениваются компоненты ПО (MSC), или небольшого автомата (см., например, [48–51]). Другая часть использует явно опи-

санные автоматные модели для генерации тестовых последовательностей, нацеленных на достижение определенного уровня покрытия согласно какому-либо критерию (см. [52–54]). UniTesK, как уже говорилось, поддерживает построение тестовых последовательности и из заданных пользователем сценариев, и на основе автоматной модели системы, интегрируя оба подхода.

Наиболее близки к UniTesK по поддерживаемым возможностям инструменты GOTCHA-TCBeans [55, 56] (один из инструментов генерации тестов, объединяемых в рамках проекта AGEDIS [57, 58]) и AsmL Test Tool [59, 60]. Оба они используют автоматные модели целевого ПО. Для GOTCHA-TCBeans такая модель должна быть описана на расширении языка Murphi [61], AsmL Test Tool использует в качестве спецификаций описание целевой системы как машины с абстрактным состоянием (abstract state machine, ASM, см. [62, 63]).

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 [64]. 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 [65].

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. ЗАКЛЮЧЕНИЕ

Технология UniTesK разрабатывалась на основе как опыта проведения проектов по тестированию сложного промышленного ПО, так и опыта внедрения предшественницы UniTesK, технологии KVEST [6, 66], в процессы разработки ПО Nortel Networks. Опыт KVEST, как и вообще опыт внедрения технологий, разработанных в академическом сообществе, в промышленное производство ПО показывает, что для успеха подобного проекта внедряемая технология должна обладать большим набором функций и использовать знакомые обычным разработчикам ПО понятия и обозначения. Оба этих фактора были учтены при разработке UniTesK.

Дальнейшее развитие технологии предполагается вести по нескольким направлениям.

- Планируется разработка специализированных инструментов автоматизированного создания медиаторов для широко распространенных видов компонентного ПО.

- Шаблон построения сценариев будет расширен возможностью генерации тестов, использующей только описание тестируемого интерфейса.

- Ведутся исследования по автоматизации разработки тестов для программных компонентов, функциональность которых предполагает обращение к окружению с некоторыми запросами и использование результатов этих запросов в их работе (так называемый *обратный интерфейс*).

- Исследуются возможности интеграции методов тестирования систем реального времени, а также усложненных подходов к тестированию параллельных и распределенных систем в технологию UniTesK.

В настоящий момент в ИСП РАН, совместно с Arithnet Technical Services [67], созданы инструменты, поддерживающие разработку тестов по технологии UniTesK для ПО, написанного на Java [68] и C#. Для тестирования компонентов C++ можно использовать спецификации и тесты, разработанные на расширении Java, и связываемые с кодом на C++ при помощи дополнительного слоя медиаторов, автоматически генерируемых по заголовочным файлам C++ с описанием тестируемого интерфейса. Кроме того, в ИСП РАН разработан аналогичный инструмент CTesK для тестирования программ, написанных на C [69].

Созданные инструменты успешно применялись для тестирования ПО, разрабатываемого в ИСП РАН, прежде всего, самих инструментов поддержки UniTesK. CTesK с успехом использовался для тестирования нескольких различных реализаций протокола IPv6 [70]. На основе принципов UniTesK создана специализированная технология тестирования оптимизационных блоков компиляторов, успешно опробованная на промышленных компиляторах компании Intel [8]. Полный список проектов, проводимых с использованием технологии UniTesK, можно найти на сайте группы спецификации, верификации и тестирования ИСП РАН [1].

СПИСОК ЛИТЕРАТУРЫ

1. http://www.ispras.ru/groups/rv/rv.html.
2. *Meyer B.* Applying “Design by Contract” // IEEE Computer. 1992. V. 25. № 10. P. 40–51.
3. *Meyer B.* Object-Oriented Software Construction. Second Edition. Prentice Hall, 1997.
4. *Meyer B.* Eiffel: The Language. Prentice Hall, 1992.
5. *Peters D., Parnas D.* Using Test Oracles Generated from Program Documentation // IEEE Transactions on Software Engineering. 1998. V. 24. № 3. P. 161–173.
6. *Bourdonov I., Kossatchev A., Petrenko A., Galter D.* KVEST: Automated Generation of Test Suites from Formal Specifications. FM’99: Formal Methods // LNCS. V. 1708. Springer-Verlag, 1999. P. 608–621.
7. *Obayashi M., Kubota H., McCarron S.P., Mallet L.* The Assertion Based Testing Tool for OOP: ADL2. Available via http://adl.opengroup.org/.
8. *Kossatchev A., Petrenko A., Zelenov S., Zelenova S.* Using Model-Based Approach for Automated Testing of Optimizing Compilers. Proccedings of Intl. Workshop on Program Undestanding. Gorno-Altaisk, 2003.
9. *Bourdonov I., Kossatchev A., Kuliamin V., Petrenko A.* UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391. Springer-Verlag, 2002. P. 77–88.
10. *Zafiropulo P., West C.H., Rudin H., Cowan D.D., Brand D.* Towards Analysing and Synthesizing Protocols // IEEE Transactions on Communications. 1980. V. COM-28. № 4. P. 651–660.
11. *Chilenski J.J., Miller S.P.* Applicability of modified condition/decision coverage to software testing // Software Engineering Journal. September 1994. P. 193–200.
12. *Бурдонов И.Б., Косачев А.С., Кулямин В.В.* Применение конечных автоматов для тестирования программ // Программирование. 2000. № 2. P. 61–73.
13. *Lee D., Yannakakis M.* Principles and Methods of Testing Finite-State Machines. A survey. Proceedings of the IEEE. 1996. V. 84. № 8. P. 1090–1123.
14. *von Bochmann G., Petrenko A.* Protocol Testing: Review of Methods and Relevance for Software Testing. Proceedings of ACM International Symposium on Software Testing and Analysis. Seattle, USA. 1994. P. 109–123.
15. *Бурдонов И.Б., Косачев А.С., Кулямин В.В.* Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай // Программирование. 2003. № 5. С. 11–30.
16. *Fujiwara S., von Bochmann G.* Testing Nondeterministic Finite State Machine with Fault Coverage. IFIP Transactions. Kroon J., Heijink R.J., Brinksma E. (eds.) Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems. 1991. North-Holland, 1992. P. 267–280.
17. *Бурдонов И.Б., Косачев А.С., Кулямин В.В.* Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай // Программирование. В печати.
18. http://www.junit.org/index.htm.
19. *Kramer R.* iContract – The Java Design by Contract Tool. Proceedings of TOOLS26: Technology of Object-Oriented Languages and Systems. IEEE Computer Society, 1998. P. 295–307.
20. http://www.reliable-systems.com/.

21. http://www.mmsindia.com/JMSAssert.html.

22. *Bhorkar A.* A Run-time Assertion Checker for Java using JML. Technical Report 00-08. Department of Computer Science, Iowa State University. 2000.

23. http://www.cs.iastate.edu/ leavens/JML.html.

24. *Karaorman M., Holzle U., Bruno J.* jContractor: A reflective Java library to support design by contract. Technical Report TRCCS98-31. University of California, Santa Barbara. Computer Science. January 19, 1999.

25. http://jcontractor.sourceforge.net/.

26. *Bartetzko D., Fisher C., Moller M., Wehrheim H.* Jass – Java with assertions // Havelund K., Rosu G. (eds.) Proceeding of the First Workshop on Runtime Verification RV'01. Electronic Notes in Theoretical Compter Science. V. 55. Elsevier Science, July 2001.

27. http://semantik.informatik.uni-oldenburg.de/∼jass.

28. *Duncan A., Hlzle U.* Adding Contracts to Java with Handshake. Technical Report TRCS98-32. University of California, Santa Barbara. 1998.

29. *Muller P., Meyer J., Poetzsch-Heffter A.* Making executable interface specifications more expressive / Cap C.H. (ed.) JIT'99 Java-Informations-Tage 1999. Informatik Aktuell. Springer-Verlag, 1999.

30. *Barnett M., Schulte W.* Contracts, Components, and their Runtime Verification on the .NET Platform. Technical Report TR-2001-56. Microsotf Research.

31. *Baresi L., Young M.* Test Oracles. Tech. Report CIS-TR-01-02. Available at http://www.cs.uoregon.edu/∼michal/pubs/oracles.html.

32. *Ball T., Rajamani S.* SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21. Microsoft Research. January 2002.

33. http://www.rational.com.

34. http://www.parasoft.com.

35. *Boyapati C., Khurshid S., Marinov D.* Korat: Automated Testing Based on Java Predicates. Proc. of ISSTA 2002. Rome, Italy. July 2002.

36. http://www.time-rover.com.

37. http://www.t-vec.com.

38. *Heitmeyer C.* Software Cost Reduction / Marciniak J.J. (ed.) Encyclopedia of Software Engineering. Two Volumes. ISBN: 0-471-02895-9. January 2002.

39. http://mulsaw.lcs.mit.edu/.

40. *Ellsberger J., Hogrefe D., Sarma A.* SDL – Formal Object-Oriented Language for Communicating Systems. Prentice Hall, 1997.

41. ITU-T. Recommendation Z.100: Specification and Description Language (SDL). Geneva: ITU-T, 1996.

42. ITU-T. Recommendation Z.100 Annex F1: SDL formal definition – General. 2000.

43. 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.

44. 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.

45. *Berry G.* The Foundations of Esterel / Plotkin G., Stirling C., Tofte M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, 1998.

46. *Boussinot F., de Simone R.* The Esterel language // Proc. IEEE. Sept. 1991. V. 79. P. 1293–1304.

47. *Halbwachs N., Caspi P., Raymond P., Pilaud D.* The synchronous data flow programming language LUSTRE // Proc. IEEE. Sept. 1991. V. 79. P. 1305–1320.

48. *Grabowski J., Hogrefe D., Nahm R.* Test case generation with test purpose specification by MSCs / Faergemand O., Sarma A. (eds.) 6th SDL Forum. Darmstadt, Germany: North-Holland, 1993. P. 253–266.

49. *Wang C.J., Liu M.T.* Automatic test case generation for Estelle. International Conference on Network Protocols. San Francisco, CA, USA, 1993. P. 225–232.

50. *Garavel H., Lang F., Mateescu R.* An overview of CADP 2001. INRIA Technical Report TR-254. December 2001.

51. *Fernandez J.-C., Jard C., Jeron T., Viho C.* An experiment in automatic generation of test suites for protocols with verification technology / Groote J.F., Rem M. (eds.) Special Issue on Industrially Relevant Applications of Formal Analysis Techniques. Elsevier Science publisher, 1996.

52. *Tretmans J., Belinfante A.* Automatic testing with formal methods. 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.

53. *Bourhfir C., Aboulhamid E., Dssouli R., Rico N.* A test case generation approach for conformance testing of SDL systems // Computer Com- munications. 2001. V. 24. № 3–4. P. 319–333.

54. *Chun W., Amer P.D.* Test case generation for protocols specified in Estelle / Quemada J., Mañas J., Vázquez E. (eds.) Formal Description Techniques III. Madrid, Spain: North-Holland, 1990. P. 191–206.

55. *Farchi E., Hartman A., Pinter S.S.* Using a model-based test generator to test for standard conformance // IBM Systems Journal. 2002. V. 41. № 1. P. 89–110.

56. http://www.haifa.il.ibm.com/projects/verification /gtcb/documentation.html.

57. *Gronau I., Hartman A., Kirshin A., Nagin K., Olvovsky S.* A Methodology and Architecture for Automated Software Testing. Available at http://www.haifa.il.ibm.com/projects/verification /gtcb/papers/gtcbmanda.pdf.

58. http://www.agedis.de/.

59. *Grieskamp W., Gurevich Y., Schulte W., Veanes M.* Testing with Abstract State Machines / Moreno-Diaz R., Quesada-Arencibia A. (eds.) Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001). Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain. February 2001. P. 257–261.

60. http://research.microsoft.com/fse/asml/.

61. http://verify.stanford.edu/dill/murphi.html.

62. *Gurevich Y.* Evolving Algebras: An Attempt to Discover Semantics / Rozenberg G., Salomaa A. (eds.) Current Trends in Theoretical Computer Science. World Scientific, 1993. P. 266–292.

63. *Börger E., Stark R.* Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.

64. *Friedman G., Hartman A., Nagin K., Shiran T.* Projected state machine coverage for software testing. Proc. of ISSTA 2002. Rome, Italy. July 2002.

65. *Grieskamp W., Gurevich Y., Schulte W., Veanes M.* Generating Finite State Machines from Abstract State Machines. Proc. of ISSTA'2002. Also: Microsoft Research Technical Report MSR-TR-2001-97.

66. http://www.fmeurope.org/databases/fmadb088.html.

67. http://www.atssoft.com.

68. *Bourdonov I.B., Demakov A.V., Jarov A.A., Kossatchev A.S., Kuliamin V.V., Petrenko A.K., Zelenov S.V.* Java Specification Extension for Automated Test Development. Proceedings of PSI'01 // LNCS. V. 2244. Springer-Verlag, 2001. P. 301–307.

69. http://unitesk.ispras.ru.

70. http://www.ispras.ru/~RedVerst/RedVerst/White Papers/MSRIPv6 Verification Project/Main.html.

71. *Kuliamin V., Petrenko A., Pakoulin N., Kossatchev A., Bourdonov I.* Integration of Functional and Timed Testing of Real-time and Concurrent Systems. Материалы PSI'03. Новосибирск, 2003.