ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ
Российской академии наук
(ИСП РАН)

Описание работы

Разработка и внедрение технологии автоматизированного тестирования программного обеспечения на основе формальных спецификаций

Авторы:
Петренко Александр Константинович,
доктор физико-математических наук,
заведующий отделом ИСП РАН, руководитель работы
Демаков Алексей Васильевич,
научный сотрудник ИСП РАН
Кулямин Виктор Вячеславович,
кандидат физико-математических наук,
старший научный сотрудник ИСП РАН
Пакулин Николай Витальевич,
кандидат физико-математических наук,
младший научный сотрудник ИСП РАН
Рубанов Владимир Васильевич,
научный сотрудник ИСП РАН

Москва – 2007

Содержание

Введение

В современном мире компьютерные системы играют важную роль в общей инфраструктуре обеспечения деятельности человека. На них возлагается решение все более ответственных задач, поэтому экономическое и культурное развитие человечества требует создания все более сложных таких систем, обладающих большим количеством разнообразных функций. При усложнении вычислительных комплексов все б***о***льшая часть их стоимости и трудозатрат на разработку и поддержание в рабочем состоянии падает на программное обеспечение, а ошибки в его работе обходятся все дороже. Широко известны примеры дорогостоящих ошибок в программных системах.

- Ошибка в программном обеспечении прибора радиационной терапии Терак-25, использовавшегося в 1985-1987 годах, привела к 3-м зафиксированным летальным исходам, непосредственно связанным с неправильной работой прибора [1].
- Многочисленные ошибки в системе управления двигателями и навигационной системе считаются наиболее вероятной причиной катастрофы вертолета Chinook ZD 576, произошедшей 2 июня 1994 года на мысе Кинтайр. В этой катастрофе погибли 25 экспертов и высокопоставленных сотрудников отдела разведки Великобритании в Северной Ирландии, что на значительное время парализовало работу этого отдела [2].
- Ошибка в системе управления полетом ракеты Ариан-5, приведшая к ее уничтожению при первом запуске 4 июня 1996 года, стоила по оценкам экспертов около 500 миллионов долларов США [3].
- Ошибки в программных системах управления космическими кораблями привели к потере кораблей NASA Mariner-1 (1962), Mars Climate Orbiter (1999), Mars Polar Lander (1999), а также советских кораблей Фобос-1 (1988) и Фобос-2 (1989).
- Наконец, одной из причин сбоя в электроснабжении северо-востока Северной Америки, произошедшего 14 августа 2003 года, на несколько

часов оставившего без электричества около 50 миллионов человек и приведшего к потерям на сумму 6 миллиардов долларов США, была ошибка в программной системе оповещения о сбоях на электростанции [4].

На программное обеспечение возлагается управление все более сложными и критическими системами, например, транспортными узлами, системами энергообеспечения, процессами производства на больших предприятиях. При этом такие программы часто используют в своей работе данные, полученные от другого программного обеспечения, с менее высокими требованиями к их надежности и качеству. Колоссальная сложность возникающих комплексов не всегда позволяет проанализировать все возможности проявления ошибок или все последствия использования некорректных данных. Поэтому даже небольшие ошибки в работе таких сложных комплексов способны в критической ситуации, в сочетании с другими проблемами, отвлекающими внимание людей, приводить к катастрофическим последствиям.

Такая ситуация делает весьма актуальной разработку технологий создания сложного программного обеспечения (ПО) высокой надежности.

Одним из наиболее широко применяемых подходов к созданию высоконадежного ПО является тщательное тестирование как программной системы в целом, так и каждого из ее компонентов, начиная с самых ранних стадий разработки, на которых такое тестирование становится возможным. Как показывают исследования [5,6], тестирование является единственным методом контроля качества ПО, который позволяет проверять корректность и надежность функционирования системы в ее рабочем окружении.

Однако при росте сложности и количества функций программной системы трудоемкость ее тщательного тестирования растет значительно быстрее. Новые технологии разработки программного обеспечения позволили за последние двадцать лет повысить размеры разрабатываемых в промышленности программных систем с десятков тысяч до десятков миллионов строк кода. В то же время, аналогичного повышения

эффективности технологий обеспечения качества ПО не произошло, что привело к увеличению как количества инцидентов, связанных с некорректной работой программных систем, так и ущерба от них. В результате разработчики программных систем с повышенными требованиями к надежности вынуждены либо снижать эти требования, либо тратить на тестирование традиционными методами значительные усилия, до 80% всех трудозатрат на разработку таких систем. По оценке института NIST использование на практике технологий тестирования, неадекватных сложности современного ПО, стоило экономике США только в 2001 году около 60 миллиардов долларов [7].

Цели работы

Представляемая работа имеет своей целью создание и внедрение в промышленную практику технологии автоматизированного тестирования, позволяющей проводить тщательное тестирование сложного ПО.

Такая технология должна поддерживать решение всего комплекса задач, связанного с построением тестов высокого качества при приемлемом уровне затрат по сравнению с обычной разработкой тестов. При этом она должна хорошо интегрироваться с современными технологиями и инструментами разработки программного обеспечения.

Широкое использование подобных технологий при создании систем с повышенными требованиями к надежности позволило бы экономически эффективно использовать современные подходы к разработке сложных систем без снижения их итогового качества и без увеличения риска возникновения ошибок, критических для их пользователей.

Основные результаты работы

Основные результаты представляемой работы следующие.

- Предложен ряд методов построения тестов для сложного программного обеспечения на основе формальных спецификаций требований к его поведению. Эти методы включают использование унифицированной

архитектуры тестового набора, приспособленной для автоматизации тестирования ПО из различных предметных областей.

- Разработаны инструментальные программные средства, автоматизирующие выполнение ряда шагов предложенных методов для программного обеспечения, имеющего программный интерфейс на широко используемых языках программирования — C, Java, C#. Эти инструменты вместе с предложенными методами являются составными элементами технологии разработки тестов UniTESK.
- Разработаны методики формализации требований стандартов для некоторых классов промышленных программных систем, нацеленные на получение спецификаций, пригодных для использования предложенного метода и поддерживающих его инструментов. Охваченные классы программных систем — это реализации телекоммуникационных протоколов и библиотеки программных интерфейсов операционных систем (ОС). Такие методики позволяют при помощи технологии UniTESK автоматизировать разработку наборов тестов для проверки соответствия стандартам на программные системы этих классов.
- Разработанная технология UniTESK апробирована в нескольких пилотных проектах по созданию тестов для программных систем различных классов.
    - Для реализации Интернет-протокола следующего поколения IPv6, сделанной в Microsoft Research, в 2001-2002 годах. В ходе этого проекта обнаружены критические для работы такой системы ошибки, в частности, позволяющие удаленным образом останавливать работу узла сети.
    - Для компонентов распределенной ОС TinyOS в 2003 году.
    - Для банковской системы управления данными о клиентах, построенной на основе технологий Java 2 Enterprise Edition в 2004 году. В ходе этого проекта обнаружены ошибки, приводившие к помещению в базу данных некорректной информации о клиентах.

- Технологии UniTESK и разработанные методики формализации требований использованы в нескольких крупных проектах по созданию тестов для промышленных программных систем.
  - В проекте по разработке тестов для реализации мобильного протокола Интернет Mobile IPv6 в 2003-2004 годах. В ходе проекта обнаружено значительное количество ошибок, включая несколько критических.
  - В проекте по разработке тестового набора для тестирования на соответствие стандарту протокола обеспечения безопасности IPsec в 2003-2006 годах. При выполнении полученного набора тестов на одной из доступных реализаций этого протокола также было обнаружено несколько ошибок.
  - В проекте по разработке тестов для части инфраструктуры платформы Java, включавшей программный интерфейс управления работой виртуальной машины и несколько компонентов базовых библиотек Java Development Kit в 2005 году. В результате было обнаружено несколько серьезных ошибок в реализациях инфраструктуры Java от компаний Sun и Bea.
  - Для части системы интеграции информационных ресурсов компании «Вымпелком» в 2005-2006 годах. Результаты этого проекта позволили заказчику существенно сократить время введения в эксплуатацию компонентов ее информационной системы.
  - В проекте по разработке тестов и тестированию базовых библиотек отечественной POSIX-совместимой ОС реального времени ОС 2000 в 2005-2006 годах. В результате этого проекта заказчиком отмечено увеличение полноты производимого тестирования и повышение качества новых версий тестируемой системы.
  - В проекте по разработке тестов для проверки соответствия стандарту Linux Standard Base на программный интерфейс

библиотек операционной системы Linux в 2006 году. В результате этого проекта найдено несколько ошибок в базовых библиотеках Linux и зафиксировано несколько десятков замечаний к тексту стандарта, которые приняты его разработчиками.

Во всех проектах отмечалась приемлемая трудоемкость разработки тестов при значительном повышении их качества по сравнению с традиционной разработкой тестов. Доля затрат на создание тестов и тестирование в общих расходах на разработку оставалась на обычном уровне, общие сроки разработки не увеличивались, а иногда и сокращались, надежность же создаваемых в итоге программных систем существенно повышалась.

В перспективе результаты работы позволят значительно повысить надежность разрабатываемого сложного программного обеспечения при сохранении текущего уровня затрат на его разработку по сравнению с существующими методами.

Показатели работы

Как комплекс методик и средств автоматизации разработки тестов технология UniTESK является новой. Наиболее близкие ее аналоги — инструменты автоматизации разработки, созданные в IBM и Microsoft Research, не имеют всех функций данной технологии и не применялись в практических проектах такого масштаба.

Помимо использования разработанной технологии создания тестов в нескольких крупных и целом ряде небольших проектов, были получены следующие показатели.

- По теме данной работы ее авторами в соавторстве с сотрудниками ИСП РАН и других организаций за 2000-2006 годы опубликовано 40 работ.
- По теме данной работы ее авторами за 2000-2006 годы проведено 24 доклада на научных конференциях и семинарах, в том числе 12 — на международных конференциях и семинарах.

- По теме данной работы студентами профильных кафедр факультета вычислительной математики и кибернетики Московского государственного университета и Московского физико-технического института, в обучении которых участвовали авторы работы, за 2000-2006 годы подготовлено 12 дипломных работ, получивших отличные оценки.
- На основании результатов, полученных при выполнении данной работы, двое из ее авторов в 2006 году защитили диссертации на соискание учёной степени кандидата физико-математических наук.
- При участии авторов данной работы были разработаны обучающие курсы по инструментам технологии UniTESK. За 2000-2006 годы эти курсы были проведены для сотрудников следующих организаций.
  - Кафедра системного программирования факультета математики и механики Санкт-Петербургского государственного университета.
  - Научно-исследовательский институт системных исследований Российской академии наук.
  - Российские компании-разработчики ПО Oktet Labs, Luxoft, RTSoft, Компас.
  - Лаборатория разработки ПО закрытого акционерного общества Интел в Санкт-Петербурге, Россия.
  - Отдел верификации аппаратного обеспечения института Фраунгофера, Германия.
  - Компания-разработчик программного обеспечения Systematic Inc., Дания.
  - Индийские компании-разработчики ПО Quality Kiosk, Ready Test Go, SG Smith, IDRBT, Secure Matrix.

Общие задачи тестирования

Данный раздел содержит определения и краткое обсуждение используемых далее понятий, а также формулирует основные задачи тестирования и разработки тестов в том виде, как они понимаются большинством специалистов в этой области на сегодняшний день. Такое обсуждение

необходимо, поскольку терминология в этой области меняется заметным образом каждые 5-10 лет.

Тестирование, один из видов деятельности в рамках полного жизненного цикла ПО, определяется [6,8,9] как проверка выполнения требований к программной системе при помощи наблюдения за ее работой в конечном наборе специальным образом выбранных ситуаций. В этом определении стоит выделить следующие аспекты.

Проверка требований

Тестирование — это проверка выполнения требований. Таким образом, тестирование всегда подразумевает использование *требований* к тестируемой системе, которые определяют, какое ее поведение в некоторой ситуации должно рассматриваться как правильное, а какое — как неправильное. Соответственно, чем четче и полнее сформулированы требования, тем большую пользу может принести тестирование, а чем они запутаннее и неопределеннее, тем меньше смысла имеет тщательное тестирование такой системы — оно в этих случаях приводит к большим трудозатратам при незначительном итоговом повышении ее качества.

Одним из крайних случаев является полное отсутствие документально зафиксированных требований. При этом, однако, часто руководствуются тем правилом, что система, по крайней мере, не должна «падать», демонстрировать сбои, и выполняют простое тестирование, нацеленное на их обнаружение.

Противоположный крайний случай — наличие детально проработанного стандарта, описывающего требования к поведению системы. В этом случае тестирование может надежно подтвердить высокое качество системы или существенным образом помочь в его достижении.

Использование реальной работы системы

Тестирование подразумевает реальную работу тестируемой системы. Этим оно отличается от многих других методов верификации ПО, которые

работают с исходным кодом системы (статический анализ) или проектной документацией и моделями (проверка моделей, аналитическая верификация) и не требуют, чтобы в ходе выполняемых проверок система действительно работала. Использование реальной работы системы позволяет применять тестирование для проверки корректности функционирования системы в ее рабочем окружении, в котором она будет работать в ходе эксплуатации, что невозможно сделать при помощи других методов верификации.

Тестовые ситуации и тесты

Тестирование всегда выполняется в наборе специально созданных ситуаций. Такие ситуации называются *тестовыми ситуациями* и обычно включают несколько элементов.

- Набор внешних воздействий, которые оказываются на систему в такой ситуации. Эти воздействия называют *тестовыми воздействиями*. Примерами тестовых воздействий могут служить вызов интерфейсной функции системы, нажатие на кнопку графического интерфейса пользователя, выполнение команды в командной строке. Чаще всего воздействия связаны с определенными данными, которые они передают в систему — это, например, аргументы вызванной функции или выполненной команды, содержимое полей редактирования формы, где нажимается кнопка. Эти данные называют *тестовыми данными*.
- Внутреннее состояние системы в этой ситуации. Чаще всего в современных сложных программных системах внутреннее состояние не наблюдаемо полностью извне системы и не может быть непосредственным образом установлено в определенное значение. Чтобы несмотря на это управлять внутренним состоянием во время тестирования и иметь возможность проверить поведение системы в разных состояниях, используют различные последовательности тестовых воздействий, называемые *тестовыми последовательностями*.
- Внешние условия, воспринимаемые системой самостоятельно, без оказания на нее специальных воздействий, и влияющие на ее работу. В

тех случаях, когда кроме тестовой системы, осуществляющей тестирование, никакие другие факторы не могут воздействовать на тестируемую систему (такое тестирование называется *полностью контролируемым*), можно моделировать внешние условия специальными воздействиями. На практике такое моделирование возможно только для некоторых классов систем, например, для тех, чье поведение целиком определяется воздействиями через определенный интерфейс и рядом конфигурационных параметров, которые можно изменять через этот же интерфейс или просто изменяя записи в определенных файлах или базах данных.

Программа или четко описанная процедура, при выполнении которой создается одна или несколько тестовых ситуаций и проверяется правильность поведения системы в этих ситуациях, называется *тестом*.

Тестирование чаще всего организуется как выполнение некоторого набора тестов. Такой *тестовый набор* создается заранее, до проведения тестирования, при *разработке тестов*.

Тестовый набор для сложной программной системы чаще всего тоже представляет собой сложную (чаще всего тоже программную) систему. Эта система иногда создается при разработке тестируемой системы, а потом выбрасывается, иногда поставляется как часть тестируемой системы, а иногда является отдельным программным продуктом. Например, тестовые наборы для проверки соответствия стандартам на программные библиотеки, протоколы и языки программирования иногда стоят десятки и даже сотни тысяч долларов. Это показывает, что разработка хорошего тестового набора для сложной программной системы требует значительных трудозатрат.

Полнота тестирования

Что же значит «хороший» тестовый набор? В определении тестирования говорится, что оно выполняется всегда лишь в *конечном* наборе ситуаций. Здесь имеется в виду, что тестирование всегда проводится за ограниченное время в рамках разработки ПО. При этом требуется достичь компромисса

между затратами на тестирование, которые растут вместе с количеством выполняемых тестов и временем, потраченным на их разработку, и пользой от тестирования в виде подтверждения высокого качества тестируемого ПО или обнаружения сбоев и нарушений требований.

Это означает, с одной стороны, что успешно проведенное тестирование не может дать полной гарантии корректности тестируемой системы, поскольку для всех практически значимых систем существует настолько много различных ситуаций, в которых их поведение различается, что выполнение полного тестирования никогда бы не окупилось. Но, с другой стороны, выбирая тесты определенным образом, чтобы они проверяли наиболее важные аспекты поведения системы в наиболее важных с точки зрения ее эксплуатации ситуациях, чтобы они не выполняли совершенно лишних проверок, и чтобы в них учитывались разнообразные факторы, способные влиять на поведение системы, мы можем, выполнив относительно небольшой тестовый набор, обеспечить обнаружение практически всех серьезных ошибок или получить достаточно твердую уверенность в высоком качестве тестируемого ПО.

Такой тестовый набор называют *полным* или *адекватным*. Для определения того, насколько адекватен данный тестовый набор, используют *критерии полноты тестирования* или *критерии адекватности тестирования* [8,10]. Чаще всего при этом рассматривают некоторые из возможных тестовых ситуаций как эквивалентные и измеряют количество неэквивалентных тестовых ситуаций, встретившихся или «*покрытых*» во время тестирования. Такие критерии полноты называются *критериями тестового покрытия* [8,10,11].

Критерии покрытия различаются по источнику информации, на основании которой тестовые ситуации считаются эквивалентными. Те критерии, в которых эквивалентными объявляются ситуации, задействующие одни и те же структурные элементы тестируемой системы, называются *структурными*. Примерами структурных критериев покрытия являются критерии покрытия

инструкций или ветвлений в коде системы. Если тестовые ситуации считаются эквивалентными на основании того, что они затрагивают один и тот же набор требований к системе, соответствующий критерий покрытия называется *функциональным*. Примером функционального критерия является критерий покрытия проверяемых утверждений, извлекаемых из требований к системе.

Использование структурных и функциональных критериев покрытия по отдельности не дает хороших гарантий полноты тестирования. Структурные критерии хорошо отражают полноту проверки кода, но ничего не могут сказать о том, насколько полно реализованы в нем требования к системе. Функциональные критерии, наоборот, хорошо отражают проверенные в ходе тестирования требования, но ничего не могут сказать о том, насколько полно протестированы различные элементы самой системы. Только комбинация структурных и функциональных критериев позволяет адекватно оценивать полноту тестирования сложных программных систем.

Задачи разработки тестов

Суммируя сказанное выше, можно следующим образом сформулировать основные задачи разработки тестов.

- Тесты должны проверять корректность поведения тестируемой системы относительно требований к ней. Часть теста, которая отвечает только за проверку соответствия наблюдаемого поведения тестируемой системы требованиям и за вынесение вердикта о том, правильно или неправильно сработала система в каждом конкретном случае, называется *тестовым оракулом*. Часто роль оракула играет человек, выполняющий тест и самостоятельно проверяющий корректность работы ПО.
- Тестовый набор должен обеспечивать достаточное полное тестирование относительно некоторого критерия тестового покрытия. Часть теста, отвечающая за создание тестовых воздействий, за принятие решения о том, пора или нет остановить тестирование, и, если нет, то какие еще

тестовые воздействия нужно создать, называется *генератором тестовых воздействий*.

Если ограничить тестирование только программным интерфейсом и сводимыми к нему видами воздействий, каждый тест может быть представлен в виде программы. Такое ограничение достаточно удобно для автоматизации тестирования, поскольку выполнение таких тестов может быть полностью возложено на компьютер. В то же время для практически важных классов программных систем, таких как интерфейсные библиотеки операционных систем, библиотеки поддержки выполнения программ на различных языках программирования, реализации телекоммуникационных протоколов, интерфейсные библиотеки промежуточного ПО — Web-серверов, серверов приложений и серверов баз данных — это ограничение выполняется.

При построении тестов в виде программ естественным образом возникают задачи построения программных модулей, играющих в них роли оракула и генератора тестовых воздействий. Технология автоматизации построения тестов UniTESK [12,13], разработанная в рамках представляемой работы, предлагает методику построения этих модулей.

Помимо основных задач разработка тестов должна решать и ряд вспомогательных.

- Тестирование, после которого остается только информация о том, найдены или нет ошибки в тестируемой системе, практически бесполезно. На практике нужны тесты, при выполнении которых собирается, как минимум, следующая информация.
  - Данные обо всех обнаруженных при тестировании ошибках: при каком именно воздействии зафиксирована ошибка, в чем именно она состояла, какое требование нарушено, какие воздействия ей предшествовали.
  - Данные о полноте тестирования по нескольким критериям: какие тестовые ситуации возникали в ходе тестирования, а какие нет

- Тестовый набор должен быть организован так, чтобы максимально облегчать сопровождение тестов. Это особенно важно для стандартов, которые должны иметь тестовые наборы для проверки соответствия им, а также для систем, которые планируется сопровождать и развивать в течение долгого времени. Помимо общего для всех программных систем требования использовать продуманную модульную структуру, для тестовых наборов важны следующие аспекты их организации.
  - Должны быть явно определены связи между тестами и проверяемыми в них требованиями.

    Это позволяет при изменении стандарта или требований к системе быстро выделить часть тестового набора, которую нужно модифицировать.
  - Должны быть явно определены связи между тестами и проверяемыми с их помощью компонентами и модулями тестируемой системы.

    Это позволяет при внесении поправок и изменений в тестируемую систему, без изменения требований к ней, быстро выделить часть тестов, которые должны быть вновь выполнены для проверки корректности новой версии системы. При этом тест должен выбираться для такой проверки не только в том случае, если он тестирует измененный модуль, но и если он тестирует модуль, прямо или косвенно использующий измененные модули.
  - В тестовых наборах для сложных систем часто присутствуют сложные тесты, проверяющие тонкие аспекты поведения отдельных функций тестируемой системы или взаимодействие между несколькими ее функциями. В таком наборе должны также быть более простые тесты, проверяющие базовые свойства этих функций. При выполнении тестового набора простые тесты должны выполняться перед сложными, и, если простой тест находит ошибку в одной из функций, сложные тесты, использующие эту функцию,

не должны выполняться.

Такая организация удобна для более эффективной организации тестирования и анализа ошибок. Сложные тесты обычно выполняются достаточно долго. При отсутствии простых тестов грубая ошибка в одной из функций, проверяемых сложным тестом, часто приводит к невозможности анализа создавшейся ситуации, в силу сложности получаемого отчета. Простой тест, обнаруживающий эту ошибку, позволяет быстро ее зафиксировать и не тратить время на выполнение затрагиваемых ею сложных тестов.

Базовые принципы технологии UniTESK

Данный раздел содержит краткое изложение целей создания технологии автоматизированной разработки тестов UniTESK, а также формулировку ее основных принципов и описание используемой в ней унифицированной архитектуры тестов.

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.

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* интерфейса тестируемой системы, которые можно автоматически преобразовать в тестовые оракулы.

Несмотря на активное развитие методов построения тестов на основе формальных спецификаций или формальных моделей в академическом сообществе [14,15,16], 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, в данном случае — к созданию тестов и тестированию. Именно это — разработка инструментально поддержанной технологии создания и выполнения тестов — являлось целью разработки UniTESK.

Технология построения тестов для ПО общего назначения пригодна для широкого использования в промышленной практике, если она обладает следующими характеристиками.

- Во-первых, все входящие в нее методы и техники, все определяемые ею шаги, где это возможно, должны поддерживаться инструментами.
- Во-вторых, она должна обладать широким набором функций, позволяющим использовать эту технологию в проектах, имеющих различные цели, для тестирования ПО из разных предметных областей, построенного с использованием различных архитектур и технологий.
- В-третьих, она должна сопровождаться методиками, позволяющими в большинстве практических проектов получать из набора проектных документов все исходные данные, необходимые для работы по этой технологии.
- 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 дорогостоящей переподготовки персонала.

Основные решения, используемые в UniTESK

Для обеспечения таких характеристик при разработке технологии UniTESK были предложены следующие решения [12,13,17].

1. Для обеспечения максимальной гибкости технологии, была спроектирована *унифицированная архитектура теста*, определяющая набор компонентов теста с ясным разделением функций и четкими интерфейсами, и при этом так, чтобы большое многообразие различных

видов тестов для ПО различных видов 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 [18,19]), состоящих из *предусловий* и *постусловий* интерфейсных операций и *инвариантов* типов данных. Эти конструкции заимствованы из логики Хоара [20], но, в отличие от нее, применяются только к интерфейсным элементам ПО. Программные контракты, с одной стороны, достаточно удобны для разработчиков, поскольку хорошо привязываются к архитектуре ПО, с другой стороны, в силу своего представления стимулируют усилия по созданию независимых от реализации критериев корректности тестируемой системы. Основное же их преимущество в том, что они позволяют автоматически построить *оракулы* [21-23], проверяющие соответствие поведения тестируемой 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 полноты тестирования. 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 достигнутое тестовое покрытие. Фильтры для этого генерируются автоматически из определения критерия тестового покрытия (см. пункт 6).

5. Для автоматического построения последовательности тестовых воздействий в каждом тесте используется модель тестируемой системы в виде *конечного автомата* (КА). Тестовая последовательность строится как последовательность обращений к тестируемым 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 тестируемой системы, осуществляемой оракулами, можно не задавать автомат полностью, а лишь указать способ идентификации его состояний и способ итерации стимулов в зависимости от текущего состояния. Такое неявное описание автомата называется в UniTESK *тестовым сценарием*. Часто тестовый сценарий можно сгенерировать автоматически на основе спецификации тестируемых операций, способа итерации наборов их аргументов и выбранного в качестве цели критерия покрытия.
6. Для определения момента, когда тестирование можно закончить, UniTESK предлагает опираться на достигнутое тестовое покрытие в соответствии с некоторым критерием. Из структуры спецификаций, разработанных в соответствии с технологией UniTESK, можно

автоматически извлечь несколько таких критериев. Пользователь имеет возможность гибко использовать эти критерии в тестах, а также определять и использовать дополнительные критерии.

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 тестируемой систем, необходимую для проведения тестирования. На данный момент в ИСП РАН разработаны инструменты, поддерживающие работу по технологии UniTESK с использованием расширений языков программирования Java, C и C#.
8. Технология UniTESK включает в себя набор методик, предназначенных для облегчения формализации требований в специфических предметных областях, где сложилась определенная практика представления требований к программным системам. На данный момент такие методики сформулированы для стандартов на телекоммуникационное ПО и на интерфейсные библиотеки операционных систем [24-26]. Поскольку в этих областях стандартизация интерфейсов выполняется достаточно давно и разработка большей части современного ПО ведется с учетом этих стандартов, применении технологии автоматизированного построения тестов здесь способно принести значительные выгоды.

9. Спецификации на основе программных контрактов в рамках технологии UniTESK могут быть отделены от кода тестируемой системы и использоваться в неизменном виде для тестирования различных реализаций одной и той же функциональности, представляя собой формализацию некоторого набора функциональных требований к ПО. Для определения связи между спецификациями и тестируемым ПО 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 тестируемого ПО, а иногда и для продуктов разных поставщиков. Для переработки тестового набора под новую версию, в которой меняются внешние интерфейсы, но не их функции, достаточно заменить медиаторы. Во многих случаях такая замена может быть автоматизирована.
    - Становится возможным многократное использование одних и тех же спецификаций и тестов, которое значительно повышает отдачу от вложенных в их разработку ресурсов.

При применении технологии UniTESK в специфической области зачастую используются не все входящие в нее техники построения тестов и не все компоненты из унифицированной архитектуры теста. В некоторых случаях использование каких-то техник невозможно или требует слишком больших затрат. Тем не менее, в этих случаях можно использовать специализированные техники 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, в частности, выражать тот факт, что оптимизация программы действительно была проведена, но итоговая программа полностью эквивалентна исходной по поведению. 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 [27,28].

Унифицированная архитектура тестового набора

Гибкость технологии или инструмента, возможность их использования в разнообразных контекстах, определяется, в первую очередь, лежащей в их основе архитектурой. Архитектура теста, используемая в UniTESK, проектировалась на основе многолетнего опыта проведения тестирования сложного промышленного ПО из разных предметных областей и нацелена на решение двух основных проблем.

- Невозможно полностью автоматизировать разработку тестов, поскольку критерии корректности ПО и стратегию проведения тестирования может определить только человек. Тем не менее, очень многое может и должно быть автоматизировано.
- Выбранная архитектура должна совмещать единообразие с возможностью тестирования программных систем, относящихся к разным предметным областям, 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 человека.

Архитектура теста UniTESK [17] основана на следующем разделении задач тестирования.

1. Задача проверки корректности поведения системы в ответ на единичное воздействие. Выполняется оракулом.
2. Задача создания единичного тестового воздействия. Выполняется в рамках генератора тестовых воздействий.
3. Задача построения последовательности таких воздействий, нацеленной на достижение максимального тестового покрытия по выбранному критерию. Выполняется генератором тестовых воздействий.
4. Задача установления связи между тестовой системой, построенной на основе абстрактных моделей, и конкретной тестируемой системой. Выполняется медиаторами.

Для решения каждой из этих задач предусмотрена технологическая поддержка при создании соответствующего компонента теста.

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

   Такие оракулы легко строятся из спецификаций требований в виде пред- и постусловий интерфейсных операций и инвариантов типов, формулирующих условия целостности данных [22,23]. При этом 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 распределенных систем и техника, позволяющая описывать требования к ним в виде программных контрактов.

2. Единичные тестовые воздействия строятся при помощи механизма перебора всех интерфейсных операций тестируемого модуля и итерации некоторого широкого множества наборов аргументов для каждой фиксированной операции. Этот механизм дополняется фильтрацией полученных наборов тестовых данных по критерию покрытия, выбранному в качестве цели тестирования. Если полученное так воздействие в некоторой ситуации увеличивает итоговое тестовое покрытие, такое воздействие применяется к тестируемой системе, иначе оно отбрасывается и задействуется тот же механизм для получения следующего воздействия.
3. Для тестирования ПО со сложным поведением, зависящим от предшествующего взаимодействия с окружением, недостаточно набора единичных тестовых воздействий. Необходимая часть истории взаимодействия представляется в таких случаях как внутреннее состояние тестируемой системы. При тестировании таких систем используют тестовые последовательности, построенные таким образом, чтобы проверить поведение системы в различных состояниях, определяемых последовательностью предшествовавших обращений к системе и ее ответных реакций.

   Для построения тестовой последовательности используется конечно-автоматная модель системы. Состояниями такой модели являются классы внутренних состояний тестируемой системы, выделенные для удобства получения максимального покрытия по выбранному критерию. Переходами в этой модели служат вызовы некоторой группы логически связанных операций тестируемой системы. Конечные автоматы достаточно просты, знакомы большинству разработчиков и могут быть

использованы для моделирования разнообразных программ. При тестировании параллелизма или распределенных систем используется особый подход, описанный ниже. Конечно-автоматная модель системы представляется в виде специального программного компонента — *итератора тестовых воздействий*, являющегося частью генератора тестовых воздействий. Он имеет интерфейс, предоставляющий вовне идентификатор текущего состояния, идентификатор очередного воздействия, допустимого в данном состоянии, и возможность выполнения воздействия по его идентификатору.

Тестовая последовательность строится динамически, т.е. не задается заранее, а возникает лишь во время тестирования, за счет построения некоторого "исчерпывающего" пути по переходам автоматной модели тестируемой системы. Это может быть обход всех ее состояний, всех переходов, всех пар смежных переходов и т.п. Алгоритм построения такого пути на достаточно широком классе автоматов оформлен в виде другого компонента теста, *обходчика*, также являющегося частью генератора тестовых воздействий.

Удобное для человека описание используемой при тестировании конечно-автоматной модели называется в UniTESK *тестовым сценарием*. Из тестового сценария автоматически генерируется итератор тестовых воздействий. Сценарии могут разрабатываться вручную, но для многих случаев достаточно сценариев, которые также можно сгенерировать 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 подразделе о построении тестовых сценариев. Обходчики нескольких разных видов предоставляются в виде библиотечных классов, и пользователю нет нужды разрабатывать их самому.

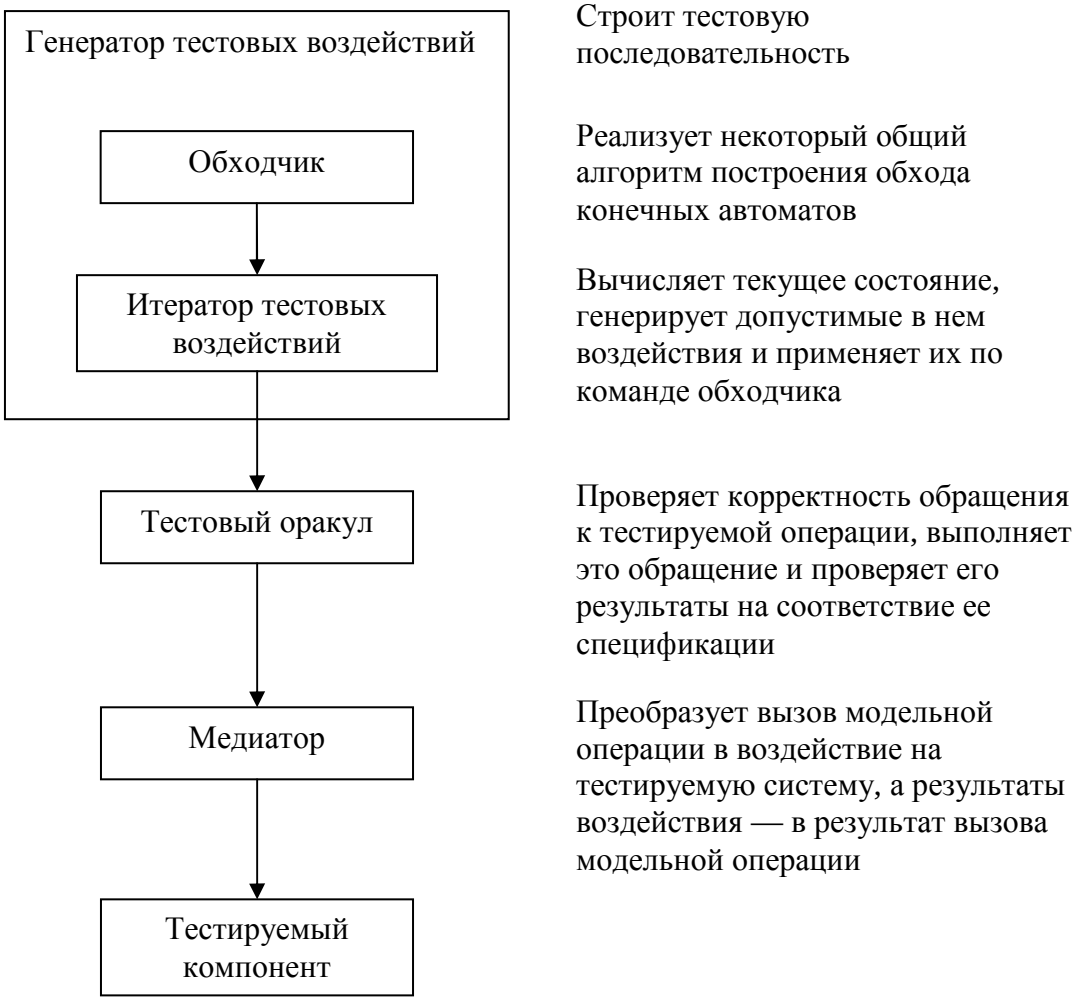

Рисунок 1. Архитектура теста UniTESK.

4. Чтобы использовать при тестировании спецификации, написанные на более высоком уровне абстракции, чем сама тестируемая система, UniTESK предоставляет возможность использовать *медиаторы*. Медиатор задает связь между некоторой спецификацией операции и конкретной реализацией соответствующей функциональности в тестируемой системе. При этом он определяет преобразование модельных воздействий (вызовов модельных операций) в реализационное (вызовы функций или методов в тестируемой системе, посылку ей сообщений и пр.), и обратное преобразование реакций тестируемой системы в их модельное представление (результат, возвращаемый модельной операцией).

   Медиаторы удобно представлять на расширении языка программирования, где можно описывать только 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 UniTESK 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 и реализованы в виде библиотеки поддержки выполнения тестов.

Тестирование распределенных систем и параллелизма

При тестировании распределенных систем и параллелизма систем общая архитектура теста остается той же самой, но несколько изменяются области ответственности и правила построения ее компонентов.

Параллелизм многократно увеличивает разнообразие ситуаций, которые могут возникать при тестировании, и делает задачу проверки соблюдения требований гораздо более сложной. Чтобы решить эту задачу, в рамках технологии UniTESK предлагается использовать одно и то же описание функциональности тестируемой системы для поведения, не затрагивающего параллелизм и для поведения в случае параллельных воздействий на нее.

Чтобы получить возможность проверять требования к системе в обоих видах ситуаций на основе одних и тех же описаний, предполагается, что тестируемая система удовлетворяет *аксиоме простого параллелизма* (или же *семантике чередования*):

> Любое корректное поведение системы в случае нескольких параллельно выполняемых воздействий эквивалентно ее поведению при выполнении этих же воздействий в некотором порядке.

В случае распределенных систем параллелизм выполнения отдельных действий в различных компонентах является внутренне присущим свойством системы. Для его адекватного моделирования, помимо использования параллельных воздействий, необходимо использовать *асинхронные события*, создаваемые компонентами такой системы и видимые извне. При рассмотрении таких систем понятия тестового воздействия и реакции системы на него обобщаются до понятия *наблюдаемого события* [29-32]. Наблюдаемые события делятся на *стимулы*, создаваемые окружением и воспринимаемые системой, и *реакции*, создаваемые системой. Обычное воздействие при этом является стимулом, а его результат — реакцией. Однако в распределенных системах возникают и другие виды реакций — асинхронные события, создаваемые различными компонентами системы, и обратные вызовы, предназначенные для получения дополнительных данных от окружения при выполнении его предшествующих запросов.

При проверке корректности набора событий, возникновение которых невозможно четко упорядочить по времени, в силу чего их можно считать возникшими параллельно, действует та же семантика чередования:

> Любое корректное поведение системы в случае нескольких возникших событий эквивалентно ее поведению при возникновении этих же воздействий в некотором порядке.

Для представления требований к корректному поведению системы в рамках одного события используется *контракт этого события* — его предусловие и постусловие [31,32].

При тестировании распределенных и параллельных систем с использованием этой семантики компоненты теста UniTESK строятся следующим образом.

1. Обходчик может быть оставлен неизменным.

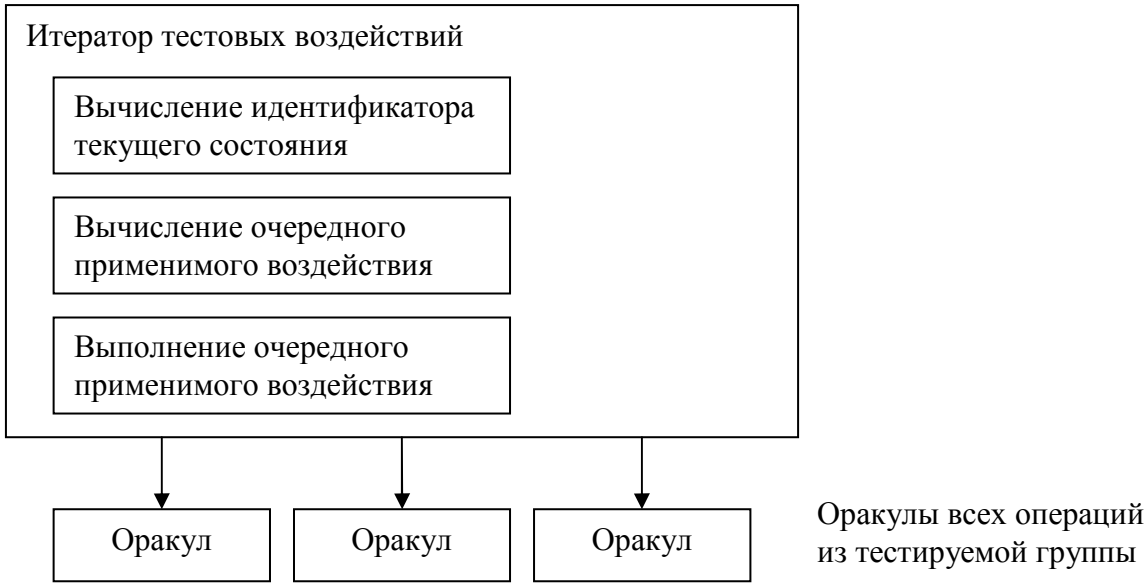

Рисунок 2. Структура итератора тестовых воздействий при тестировании без параллелизма или асинхронности.

2. Итератор тестовых воздействий представляет в этом случае более сложную модель системы. В ней выделяются *стационарные состояния*, в которых не может возникать реакций системы, если на нее не будут подаваться стимулы. Конечно-автоматная модель системы, которую использует обходчик для построения обхода, имеет в качестве состояний только стационарные состояния.

   Переходы между состояниями определяются как наборы событий, включающие один или несколько параллельных стимулов и все приходящие от системы в ответ на них реакции до ее перехода в стационарное состояние. Такой набор событий называется *тестовым фреймом*. Вместо очередного отдельного воздействия, как в случае тестирования последовательных систем, итератор тестовых воздействий определяет очередной набор стимулов, которые надо будет применить.

   Дополнительной задачей итератора тестовых воздействий является генерация всех возможных комбинаций параллельных воздействий по некоторой схеме, определяемой разработчиком теста, для построения разнообразных тестовых фреймов. Такая схема может выглядеть как «все возможные пары воздействий» или «все тройки, в которых два

воздействия одного типа и одно воздействие другого типа» и т. д.

Другая дополнительная задача итератора тестовых воздействий — вынесение вердикта о корректности всего набора событий в рамках одного тестового фрейма. Для этого используется дополнительный компонент — *сериализатор*, задача которого — строить все возможные цепочки из всех событий данного фрейма и проверять их корректность при помощи последовательных обращений к оракулам отдельных событий. При этом если находится хотя бы одна цепочка, в которой все оракулы событий возвращают положительные вердикты, в соответствии с семантикой чередования фрейм должен быть признан отработавшим корректно. Иначе, если ни одной такой цепочки из заданного набора событий построить не удается, это означает наличие несоответствия между требованиями, зафиксированными в спецификациях событий, и реальным поведением тестируемой системы.

Различия в построении итератора тестовых воздействий для тестирования, не затрагивающего параллелизм и асинхронность, и в случае тестирования параллельного и асинхронного поведения, проиллюстрированы на Рис. 2 и 3.

3. Оракулы остаются неизменными. Одно из отличий тестирования параллелизма и распределенных систем состоит в том, что каждое отдельное событие (вызов функции системы и возвращение ее результата являются двумя разными событиями) имеет свой собственный оракул. Другое отличие — в особом способе использования оракулов — они вызываются не в ходе создания соответствующего события, а позже, при оценке корректности уже полученного в рамках фрейма набора событий с помощью сериализатора. При создании воздействия оракул играет только роль переходника, передавая вызов медиатору без всякой дополнительной обработки, за исключением записи его данных в трассу теста.
4. Медиаторы остаются неизменными.

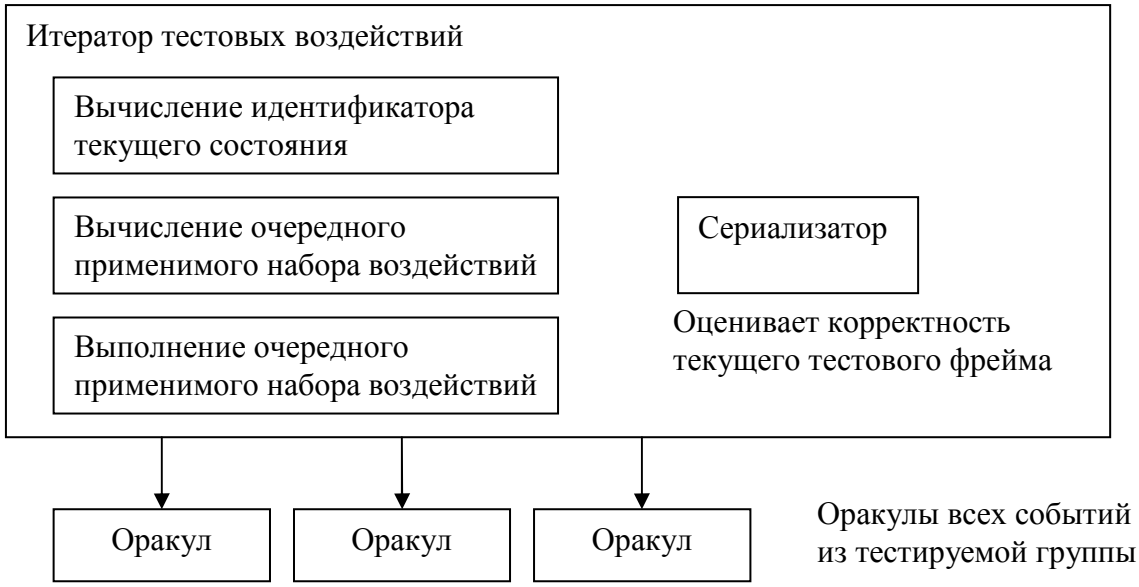

Рисунок 3. Структура итератора тестовых воздействий при тестировании параллелизма и асинхронности.

Этапы разработки тестов по технологии UniTESK

Процесс разработки тестов по технологии UniTESK может быть представлен в виде следующего набора действий [13,25].

1. **Декомпозиция интерфейса тестируемой системы.**

   Входными данными для этого этапа служат документы, фиксирующие требования к системе, стандарты, затрагивающие тестируемую систему, требования взаимодействующих с ней систем, знания и опыт экспертов в предметной области, архитекторов и разработчиков системы. На этом этапе должно быть проведено разбиение полного интерфейса тестируемой системы на группы логически связанных друг с другом операций. Впоследствии почти каждый тест для такой группы затрагивает несколько ее операций, поскольку их тестирование не может быть осуществлено в отрыве друга от друга.

   Если интерфейс исходной системы ограничен десятком операций, этот этап не выполняется — считается, что все эти операции входят в одну группу.

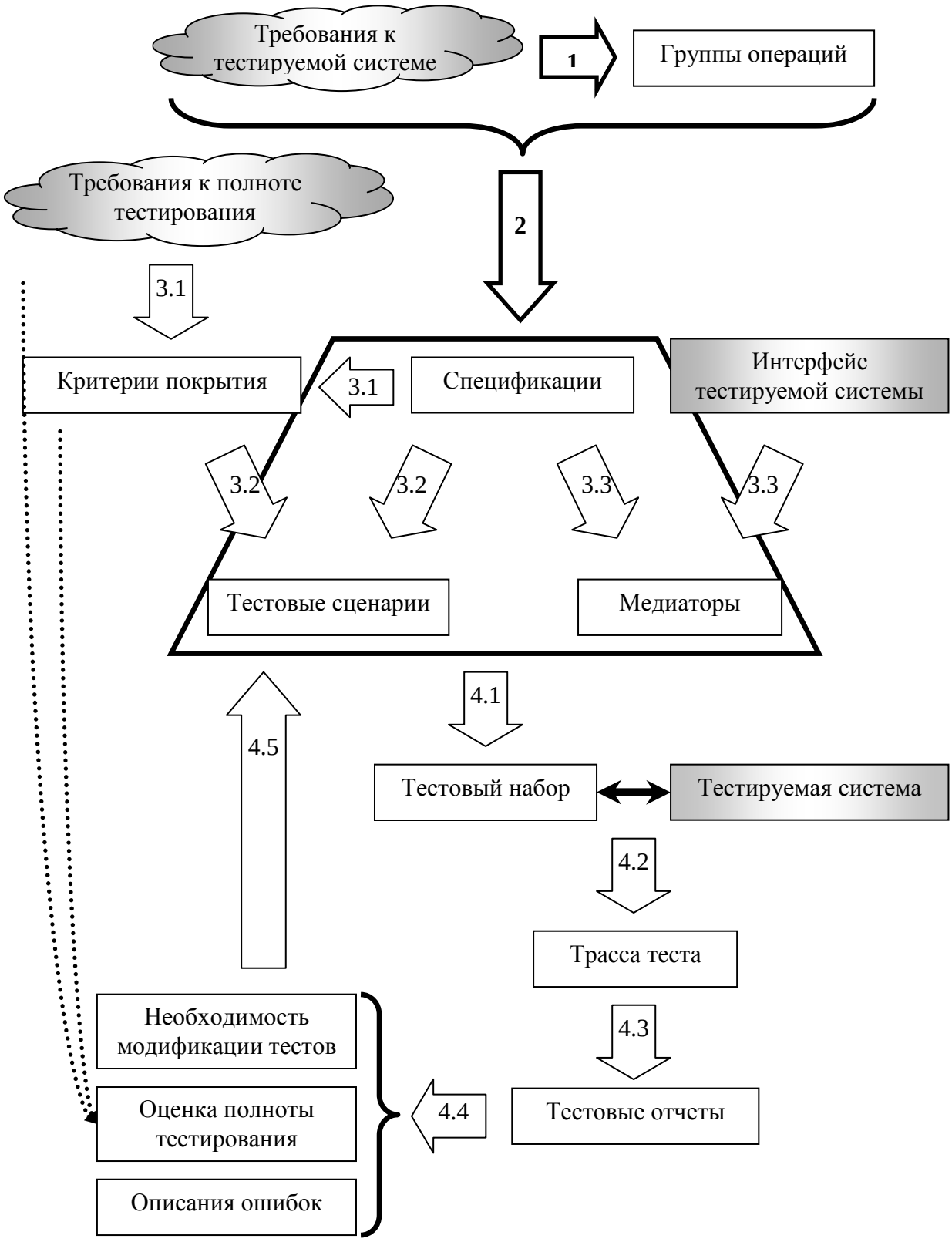

Рисунок 4. Процесс разработки тестов по UniTESK.

2. **Разработка формальных спецификаций требований.**

   Входные данные для этого этапа те же, что и для предыдущего, дополненные разбиением интерфейса системы на группы.

Разработка спецификаций производится совместно для операций из одной группы, поскольку чаще всего все их спецификации должны иметь дело с общими данными, моделирующими часть внутреннего состояния системы, связанную с работой операций этой группы. Эта работа может выполняться одним разработчиком или несколькими, тесно взаимодействующими друг с другом.

Результатом этого этапа являются формальные спецификации, описывающие требования к интерфейсным операциям системы.

3. **Разработка тестовых сценариев.**

Входными данными для этого этапа служат формальные спецификации требований к тестируемой системе и требования к полноте тестирования, определяемые на основания опыта разработчиков тестов и пожеланий заказчиков проекта и пользователей разрабатываемых тестов. В его ходе разрабатывается набор тестовых сценариев и медиаторов. Для каждой из тестируемых операций тестовой системы — один медиатор, и для каждой выделенной ранее группы операций — один или несколько тестовых сценариев.

Действия, выполняемые в ходе этого этапа для каждой группы операций, можно разбить на три фазы.

3.1 **Определение критериев покрытия.**

На основании требований к полноте тестирования и структуры спецификаций определяются критерии тестового покрытия, достижение максимального покрытия по которым будет целью разрабатываемых тестов.

3.2 **Разработка автоматных моделей сценариев.**

На основе выбранного критерия тестового покрытия и спецификаций определяется конечно-автоматная модель поведения заданной группы операций, такая, что выполнение всех переходов в ней дает полное покрытие по выбранному критерию.

В тех случаях, когда построение одной такой модели невозможно,

строятся несколько автоматных моделей, обход всех переходов которых в совокупности дает полное покрытие по выбранному критерию.

Построенные модели оформляются в виде тестовых сценариев.

3.3 **Разработка медиаторов.**

Если это необходимо, для каждой из операций группы разрабатывается медиатор, предназначенный для использования в построенном наборе сценариев. Медиаторы нужны, если разрабатываемые тесты предполагается использовать для тестирования реализаций с различным интерфейсом, или если для разработки тестов по каким-то причинам удобнее использовать интерфейс, несколько отличающий от реального интерфейса тестируемой системы.

Для разработки медиаторов необходимо знать точный интерфейс тестируемой системы и привязку функциональности, определенной в требованиях, к ее интерфейсу.

4. **Выполнение тестов и анализ их результатов.**

Входными данными данного этапа служат разработанные спецификации, медиаторы и сценарии, а также готовая к выполнению тестов тестируемая система.

В результате выполнения этого этапа набор спецификаций, медиаторов и сценариев отлаживается, устраняются все ошибки в этих компонентах, оформляется описание ошибок в самой тестируемой системе.

В итоге получается отлаженный набор тестов, набор отчетов, полученных в результате его выполнения, и выводы о качестве тестируемой системы и дальнейшем продолжении тестирования.

В рамках этого этапа можно выделить следующие действия.

4.1 **Получение готового к исполнению тестового набора.**

Для этого необходимо оттранслировать разработанные спецификации, сценарии и медиаторы в тесты на базовом языке

программирования и скомпилировать их компилятором этого языка. Это действие автоматизируется транслятором спецификационного языка.

4.2 **Выполнение тестов.**

Это действие состоит в выполнении полученного исполнимого тестового набора. В ходе выполнения теста, построенного по технологии UniTESK, автоматически генерируется трасса теста, содержащая информацию о происходивших в его ходе событиях и об обнаруженных нарушениях.

4.3 **Построение тестовых отчетов.**

На этом шаге с помощью специализированных инструментальных средств из трасс тестов генерируются отчеты о тестировании. Отчеты предоставляют в явном виде информацию о найденных нарушениях и о достигнутом тестовом покрытии по ряду критериев.

4.4 **Анализ результатов тестирования.**

На основе полученных отчетов анализируются обнаруженные тестами нарушения, которые могут быть проявлениями как ошибок в тестируемой системе, так и ошибок в спецификациях, сценариях и медиаторах. Этот анализ упрощается благодаря используемой в инструментах UniTESK классификации нарушений, позволяющей быстро отделить первые от вторых в большинстве случаев. Кроме этого, в ряде случаев для такого разделения необходим дополнительный анализ требований и/или привлечение экспертов в связанной предметной области.

Для каждой ошибки в тестируемой системе оформляется специальный отчет.

Кроме того, анализируется достигнутое тестовое покрытие, и делается вывод о необходимости дополнительной доработки тестов для его увеличения или о достаточной полноте уже имеющихся тестов.

4.5 **Внесение модификаций и исправление ошибок в тестах.**

По результатам анализа нарушений и полноты тестирования в спецификации, сценарии и медиаторы вносятся дополнения, и исправляются ошибки, которые находятся в этих компонентах.

Последовательное выполнение этих действий не всегда возможно. Например, при обнаружении недостаточной детальности спецификаций или серьезных ошибок в них возникает необходимость повторно возвращаться к первым этапам разработки. Чаще всего на практике эти этапы представляют собой виды деятельности, которые выполняются в рамках проекта в итеративной манере, пока не будут достигнуты цели проекта.

Процесс разработки тестов по технологии UniTESK проиллюстрирован на Рис. 4.

Элементы технологии UniTESK

Данный раздел содержит описания и обсуждение методов и техник, предназначенных для выполнения различных шагов разработки тестов и входящих в состав технологии UniTESK, а также различных аспектов использования этой технологии в проектах разработки ПО.

Место технологии UniTESK в жизненном цикле ПО

Технология UniTESK может использоваться на разных этапах жизненного цикла ПО и с разными целями. Примеры ее применения (см. ниже) показывают, что успешными и практическими значимыми могут быть как минимум три различных способа ее использования.

- Технология UniTESK может использоваться при разработке некоторой новой программной системы.

  В этом случае использование UniTESK экономически оправданно при условии, что данная система будет эксплуатироваться, сопровождаться и развиваться в течение достаточно долгого времени (5 и более лет).

  Технология UniTESK при этом в рамках так называемого *ко-верификационного процесса разработки*, в котором одновременно

производится программный продукт и комплект средств и инструментов, позволяющий проверить качество этого продукта.

В случае технологии UniTESK на этапе анализа требований необходимо сразу учитывать необходимость их формализации и пытаться формализовать каждое возникающее требование, а также анализировать необходимую полноту тестов для его проверки. Использование UniTESK в этом случае позволяет систематизировать сбор и анализ требований, обеспечить их проверяемость и прослеживаемость в тестах.

На этапе проектирования ПО одновременно происходит проектирование и разработка тестовых сценариев UniTESK.

На этапе кодирования и отладки в этом случае можно использовать уже готовые тесты, разработанные на предыдущих этапах. Качество тестов, создаваемых по технологии UniTESK и используемых при отладке и модульном тестировании, позволяет добиваться очень высокого качества разрабатываемого ПО.

- Технология UniTESK может использоваться для разработки тестов для проверки соответствия некоторому стандарту.

  Экономически это становится оправданным, если стандарт уже достаточно долгое время (~5-10 лет) использовался на практике и, быть может, подвергался ревизиям, устранившим большинство ошибок первой версии, иначе трудоемкость его формализации значительно возрастает. Кроме того, он должен использоваться в будущем еще достаточно долгое время (5 лет), иначе тестовый набор для него уже не будет представлять ценности.

  Экономическая целесообразность разработки тестов проверки соответствия возрастает в том случае, если имеется несколько различных реализаций этого стандарта от разных поставщиков. При этом тесты, разработанные при помощи данной технологии, обеспечивают обнаружение многих проблем переносимости в реализациях стандартов, а анализ требований стандарта, проводимый при разработке тестов,

выявляет большинство проблем этого стандарта, связанных с неясностью и неполнотой требований, противоречиями в них, и пр.

- Технология UniTESK может использоваться для разработки тестов, проверяющих корректность проводимой обратной разработки ПО или переноса его на новую платформу.

  Экономически эффективным такой способ использования UniTESK становится при наличии достаточно сложных систем с повышенными требованиями к надежности и корректности, которые построены на основе данной и зависят от корректности ее работы.

  В этом случае разработка спецификаций и тестов должна идти параллельно обратной разработке требований к имеющейся программной системе.
- UniTESK обеспечивает технологическую поддержку для создания инфраструктуры распространения программных компонентов на коммерческой основе, позволяющей установить более доверительные отношения между поставщиками и потребителями. Для функциональности, реализуемой такими компонентами можно иметь общедоступные, написанные один раз 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 третьему лицу убедиться в ее правильности. Пользователи таких компонентов могут использовать для тестирования тестовые наборы, пополненные нужным им способом, и нацеленные на тщательную проверку важных именно им аспектов функциональности.

Методика формализации требований стандартов

Применение на практике технологии, использующей формальные спецификации, невозможно без методик разработки этих спецификаций на

базе набора проектных документов и стандартов, касающихся тестируемой системы.

Для двух классов систем — для стандартов, регламентирующих реализацию телекоммуникационных протоколов, и для стандартов, содержащих требования к интерфейсным библиотекам операционных систем, — такие методы были разработаны авторами данной работы [24-26].

Ниже представлено общее описание метода формализации требований стандартов, нацеленного на получение формальных спецификаций в форме, подходящей для использования в технологии UniTESK, для которого обе разработанные методики являются уточнениями, учитывающими специфику соответствующих классов программных систем.

Применение технологии UniTESK к разработке тестов, проверяющих соответствие стандартам, обусловлено тем, что стандарты обычно представляют собой документы, содержащие уже прошедшие долгую проверку практикой, достаточно четкие и полные формулировки требований к системам определенного вида. Формализация таких требований не слишком трудоемка и в то же время имеет реальную возможность окупиться благодаря выгодам, предоставляемым наличием тестовых наборов для стандартов.

Формализация требований стандарта имеет на входе текст стандарта, а результатом ее являются формальные спецификации требований этого стандарта в виде пред- и постусловий операций, описываемых в стандарте, и инвариантов типов данных, участвующих в качестве типов параметров и результатов этих операций. Помимо этого, все ограничения, содержащиеся в итоговых спецификациях должны содержать точную привязку к конкретным разделам и фразам стандарта, в которых сформулированы эти ограничения или требования, из которых они следуют. Это необходимо для прослеживаемости требований в спецификациях и построенных на их основе тестах. Прослеживаемость требований, в свою очередь, позволяет

- наглядно показать, что в спецификациях нет ограничений, не вытекающих из текста стандарта, т.е. ничего лишнего не проверяется;
- наглядно показать, что, и наоборот, все требования, сформулированные в тексте стандарта, где-то проверяются;
- привязывать каждую из обнаруженных ошибок к конкретному месту в тексте стандарта;
- измерять полноту тестирования в терминах проверенных в его ходе требований из текста стандарта, что позволяет разработчикам тестируемой системы быстро выделить места стандарта, недостаточно аккуратно реализованные в текущей версии системы.

Формализация требований стандарта выполняется в три этапа.

1. **Составление каталога требований для данного стандарта.**

   Входные данные этого этапа — текст стандарта и связанных документов. На выходе должен быть *каталог требований*, т.е. структурированный набор всех проверяемых утверждений о поведении соответствующей стандарту системы, которые находятся в тексте стандарта и всех документах, на которые он ссылается. Каждое требование должно быть сформулировано максимально однозначно, привязано к четко определенным фразам в тексте стандарта, и каждому должен быть присвоен уникальный идентификатор, который позволяет ссылаться на него из других документов.

   При составлении каталога вначале определяется полный список документов, на которые ссылается данный стандарт. Многие требования к затрагиваемым данным стандартом системам часто определяются в одном из таких документов, а не непосредственно в тексте самого стандарта. По ходу анализа этот список может уточняться и пополняться. В итоговый каталог должны войти все требования, которые стандарт налагает на удовлетворяющие ему системы, независимо от их непосредственного источника.

   Помимо занесения требований в каталог, присваивания им

идентификаторов и указания мест в стандарте, в которых они формулируются, производится классификация требований по ряду признаков.

- По тому, к каким объектам они относятся — к данным определенного типа, к интерфейсным операциям системы, к связям между ее структурными элементами.
- По тому, на что налагаются ограничения — на системы, реализующие стандарт, или на системы, взаимодействующие с первыми.
- По степени обязательности и условиям выполнения — всегда ли должно выполняться некоторое требование или оно только может выполняться, или же его выполнение или невыполнение определяется некоторыми конфигурационными параметрами описываемой стандартом системы.
- По позитивности/негативности — некоторые требования предписывают системе делать определенные вещи, а другие, наоборот, запрещают делать что-то. Требования первого вида называются *позитивными*, второго — *негативными*. Негативные и позитивные требования по-разному представляются в формальных спецификациях.

В инструментальном наборе для создания тестов по технологии UniTESK реализована поддержка выделения требований в тексте стандарта и организации каталога требований.

2. **Построение концептуальной модели требований.**

Входными данными этого этапа являются текст стандарта и связанных с ним документов, знания и опыт экспертов в предметной области, а также каталог требований. В результате его выполнения появляется *концептуальная модель* требований стандарта.

Концептуальная модель представляет собой полное модульное полуформальное описание требований стандарта к поведению систем,

регламентируемых им. Она используется для обеспечения полноты и адекватности отражения требований в итоговой формальной модели.

При построении концептуальной модели требования, зафиксированные в каталоге, распределяются по различным аспектам и модулям, так, чтобы в одном модуле или аспекте модели остались только логически связанные требования, касающиеся одних и тех же деталей функциональности.

При этом модулям соответствуют группы требований, налагаемых на небольшую группу логически связанных операций. Аспектами становятся группы логически связанных требований, относящихся к большим наборам операций системы, если удаление этих требований не приводит к обессмысливанию ни одной из этих операций, т.е. они продолжают выполнять некоторые важные функции, но, может быть, не в полном объеме.

Каждый элемент концептуальной модели привязывается к определенному требованию из каталога или к контекстным условиям, которые не вошли в стандарт и сопровождающие его документы явно, но подразумевались их авторами. Впоследствии, при изменении стандарта, такие контекстные условия чаще всего заносятся в него как явные требования и ограничения.

При создании концептуальной модели возможно повторное возвращение к анализу текста стандарта и обращение к его авторам и другим экспертам в данной области в связи с неясными и противоречивыми требованиями в стандарте.

Конкретный вид (метамодель) используемой концептуальной модели определяется спецификой группы операций, поведение которых моделируется.

- Для операций, не зависящих от внутреннего состояния и вычисляющих функции своих аргументов, являющихся данными простых типов, достаточно удобно использовать в качестве

концептуальной модели разбиение области определения такой функции на подобласти, где ее результат описывается одним и тем же выражением.

- Для телекоммуникационных протоколов, в которых роль операций играют абстрактные операции посылки или приема сообщений, наиболее удобно использовать концептуальные модели в виде *расширенных автоматов*. Состояния в такой модели соответствуют разным логическим состояниям обработки сообщений протокола, а переходы — выполнению определенных операций или истечению тайм-аутов. Данные расширенного автомата моделируют счетчики, значения таймеров или данные сообщений, которые передаются или принимаются.
- Для большинства операций управления ресурсами операционной системы, например, для операций управления памятью, управления потоками или операций межпроцессного взаимодействия, также удобно использовать для концептуального моделирования расширенные автоматы.
- Для операций, основная функциональность которых связана с обработкой данных некоторой структуры, например, разбирающих сложные сообщения протоколов или выполняющих форматированный ввод-вывод, в качестве концептуальной модели можно использовать преобразование дерева абстрактного синтаксиса, заданного описанием этой структуры.

3. **Построение формальной модели требований.**

Входными данными для этого этапа являются концептуальная модель, а также каталог требований, текст стандарта, знания и опыт экспертов в данной предметной области.

Результатом его являются формальные спецификации требований стандарта ко всем затрагиваемым им операциям в виде их пред- и постусловий, а также инвариантов типов общих данных и параметров.

При разработке формальных спецификаций снова может не хватить информации, зафиксированной в концептуальной модели и каталоге требований, и потребуется еще раз проанализировать стандарт и связанные документы или провести опрос среди экспертов в данной области.

Описание функциональных требований в формальных спецификациях

UniTESK поддерживает автоматическую генерацию тестовых оракулов из спецификаций в виде *программных контрактов*. Программный контракт может оформляться для операций [12,13] или, в случае распределенных и асинхронно работающих систем, для событий [26,31,32]. При таком способе описания функциональности программной системы она моделируется следующим образом.

- Система является набором компонентов, взаимодействующих друг с другом и с окружением только через определенные интерфейсы.
- Интерфейс каждого компонента представляет собой пару из набора типов событий, которые этот компонент может воспринимать (*стимулов*), и набора типов событий, которые он способен создавать (*реакций*). Эти два набора не должны пересекаться.

  Выполнение операции моделируется как два события — первое событие имеет тип «вызов операции», второе — «возврат из операции».
- Каждый компонент может обладать *внутренним состоянием*. Внутреннее состояние компонента моделируется только в тех рамках, в которых оно имеет влияние на его внешнее поведение. То есть все данные, входящие во внутреннее состояние компонента, но не оказывающие влияние на происходящие с его участием события, не указываются в его модели.
- Каждый тип события имеет программный контракт, описывающий его предусловие и постусловие.
  - *Предусловие* типа событий определяет ограничения на данные события и внутренние состояния, при которых события такого типа

могут возникать.

Предусловие данного типа событий является предикатом, зависящим от данных (параметров) этого типа событий и внутреннего состояния соответствующего компонента.

Если это тип стимулов, то предусловие задает обязательства окружения не создавать события такого типа, если компонент неспособен их воспринимать. Поведение компонента при возникновении события-стимула с нарушением предусловия не определено и может привести к разрушению всей системы. Поэтому при тестировании такие стимулы не должны создаваться.

Если это тип реакций, предусловие описывает ситуации, в которых компонент имеет право создать такое событие. Нарушение предусловия в этом случае является ошибкой в соответствующем компоненте.

- *Постусловие* типа событий определяет ограничения на внутреннее состояние, в котором компонент оказывается после него, в зависимости от предшествовавшего событию состояния и данных события. Таким образом, события могут изменять внутреннее состояние компонентов, и поведение системы в целом может зависеть от истории ее взаимодействия с окружением.

  Постусловие данного типа событий является предикатом, зависящим от данных (параметров) этого типа событий, внутреннего состояния соответствующего компонента при возникновении события и его же внутреннего состояния сразу после этого события.

  И для стимулов, и для реакций нарушение постусловия означает ошибку в работе соответствующего компонента.

- Ограничения целостности данных внутреннего состояния, а также структур данных параметров событий, являются общими частями всех пред- и постусловий типов событий, имеющих такие типы параметров

или связанных с компонентами, имеющими такое состояние.

Они оформляются отдельно в виде *инвариантов* типов данных (параметров или компонентов) или инвариантов отдельных объектов.

Инвариант типа данных является предикатом, зависящим от элементов данных объекта такого типа. Инвариант объекта является предикатом, зависящим от элементов данных этого объекта.

Программные контракты были выбраны в качестве основной техники представления спецификаций, поскольку они достаточно удобны для фиксации требований и применимы для систем из очень многих предметных областей. Обычного разработчика ПО можно научить понимать их и пользоваться ими без особых усилий. Кроме того, они могут быть сделаны достаточно абстрактными или достаточно детальными по мере необходимости и могут описывать как полный комплект требований к компоненту, так и только некоторые из них, те, которые наиболее критичны в данном проекте.

Таким образом, переработка требований в программные контракты не требует больших затрат, а полученные спецификации обычно не слишком близки к описанию конкретных алгоритмов, используемых в реализации, что предотвращает во многих случаях появление ошибок одного вида и в реализации, и в спецификациях. Ошибки же разных видов достаточно легко находятся при тестировании, поскольку оно определяет места несоответствия между поведением тестируемой системы и проверяемыми ограничениями. При анализе обнаруженных несоответствий ошибки в спецификациях достаточно легко идентифицируются, за исключением тех случаев, когда они связаны с неясностью и противоречивостью самих исходных требований.

Определение критериев покрытия

В UniTESK в качестве базового критерия тестового покрытия, на максимизацию покрытия по которому нацеливаются тесты при разработке их на первой итерации, используется один из функциональных критериев покрытия (см. выше раздел об общих задачах тестирования). Затем, если

возникает необходимость в более глубоком и детальном тестировании отдельных аспектов поведения, могут использоваться структурные критерии покрытия. Такая организация разработки тестов позволяет

- начинать ее еще до того, как создается код тестируемой системы;
- с максимальной пользой задействовать требования и отразить точку зрения заказчиков и пользователей системы при разработке тестов;
- взглянуть на систему с иных позиций, чем на нее смотрят разработчики, что, как показывает практика, в большинстве случаев способствует достаточно эффективному обнаружению важных проблем в ее работе.

Использование же структурных критериев покрытия на первых шагах разработки тестов часто приводит к значительным трудозатратам на этих этапах без серьезного эффекта с точки зрения повышения качества системы в целом.

Для определения базовых функциональных критериев покрытия в UniTESK используется структура программных контрактов. Чтобы сделать возможным автоматическое извлечение критериев покрытия из них, на их организацию накладываются дополнительные ограничения. Вводятся дополнительные операторы для определения *ветвей функциональности*, расстановка которых в постусловии является обязанностью разработчика тестов. Ветвь функциональности соответствует части области определения операции, в которой операция ведет себя однородным образом, точнее, ограничения на ее поведение в этой части описываются в постусловии одними и теми же выражениями. Таким образом, различные ветви функциональности соответствуют разным наборам выражений, описывающим ограничения на результаты работы операции в ее постусловии.

В графе потока управления постусловия на каждом пути от входа к любому из выходов должен находиться ровно один оператор, определяющий ветвь функциональности, причем все ветвления, предшествующие такому оператору должны зависеть только от аргументов операции и состояния компонента при ее вызове. При этих условиях каждый допустимый вызов

данной операции может быть однозначно отнесен к одной из ветвей функциональности. Таким образом, можно измерять качество тестирования операции как процент ее ветвей функциональности, покрытых во время работы теста. Кроме того, определить ветвь функциональности можно по текущему состоянию компонента и набору аргументов операции, не выполняя саму операцию, что позволяет автоматически построить фильтр, отсеивающий наборы аргументов, не увеличивающие уже достигнутого покрытия.

Отталкиваясь от определения ветвей функциональности в постусловии, можно автоматически извлечь более детальные критерии покрытия, основанные на структуре ветвлений в пред- и постусловиях. Наиболее детальный такой критерий, поддерживаемый инструментами UniTESK, — критерий покрытия дизъюнктов — определяется всеми возможными комбинациями значений элементарных логических формул, использованных в этих ветвлениях.

При тестировании, нацеленном на достижение высокого уровня покрытия по дизъюнктам, возможны проблемы, 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.

Другой способ определения критериев покрытия — расстановка меток в коде спецификаций и сценариев. Таким образом можно определять покрытия, зависящие от результатов работы операций и от асинхронных событий, возникающих в системе после передачи ей стимулов. Еще одно применение меток в программных контрактах — определение критериев покрытий на базе исходных, неформальных требований к поведению системы. Для этого

достаточно привязать метки к требованиям, например, при помощи их идентификаторов в каталоге требований.

Помимо возможности управлять автоматически извлекаемыми из структуры спецификаций критериями покрытия, пользователь может описать свой собственный критерий покрытия спецификаций в виде набора предикатов, зависящих от аргументов операций и состояния, и использовать его для определения метрики полноты тестирования. Таким способом можно ввести и структурные критерии, если в них возникает необходимость.

Построение тестовых сценариев

В UniTESK тестовые последовательности генерируются динамически, во время выполнения теста, на основе обхода конечно-автоматной модели, описанной в тестовом сценарии.

Сценарий определяет, что именно считается состоянием этой модели, 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 тестируемых операций, и что во время тестирования будут возникать только достижимые этим способом состояния. Таким образом, перебор состояний осуществляется автоматически, и разработчику теста достаточно указать только нужный способ перебора аргументов вызываемых операций.

Базовая методика UniTESK, используемая при построении тестовых сценариев, нацелена на максимизацию достигаемого в ходе теста покрытия по некоторому критерию. Основные ее шаги следующие.

- Определяется группа операций, которые будут тестироваться совместно, и критерий тестового покрытия, по которому нужно достичь максимально возможного покрытия. Элементы покрытия по такому критерию для каждой операции — это некоторые подмножества

декартова произведения пространства параметров операции на пространство модельных состояний, связанных с данной группой операций.

- Для всех элементов покрытий по этому критерию всех операций группы вычисляются их проекции на пространство состояний. Вычисляются все непустые пересечения этих проекций. Результатом является некоторое разбиение пространства состояний на непустые подмножества. При вызове некоторой операции из данной группы в паре состояний из этого подмножества, в зависимости от аргументов вызова, можно реализовать один и тот же набор элементов тестового покрытия. Следовательно, для каждого такого множества корректно определен набор элементов покрытия, реализуемых в нем для заданной операции.
- Подмножества пространства состояний, полученные на предыдущем шаге, объявляются обобщенными состояниями группы операций. Стимулами объявляются классы эквивалентных вызовов, реализующих один и тот же элемент покрытия для некоторой операции в рамках одного из таких обобщенных состояний.

  При этом получается неявно описанный конечный автомат — определен набор его состояний и для каждого состояния — набор стимулов, которые в нем можно применять. Определить, куда ведут переходы, соответствующие этим стимулам, можно из постусловий спецификаций операций данной группы.

  При этом может оказаться, что конечные состояния переходов не определяются однозначно, т.е. автомат недетерминирован. Тогда, применяя технику, описанную в [35], и разбивая обобщенные состояния на более мелкие, можно либо трансформировать этот автомат в детерминированный, либо показать, что сделать это невозможно. В последнем случае необходимо разделить тестирование операций группы на несколько сценариев так, чтобы каждый из них определял детерминированный автомат.

- Поскольку стимулы полученных автоматов соответствуют элементам покрытия по исходному критерию, и каждый достижимый элемент покрытия имеет соответствующий стимул в одном из состояний, при обходе всех переходов в каждом из автоматов все достижимые элементы покрытия по выбранному критерию будут покрыты.

При проведении тестирования используются автоматически сгенерированные из описаний критериев покрытий 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 в них модификации.

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, например, определенные структурные критерии покрытия. С другой стороны, можно также абстрагироваться от каких-то деталей в спецификациях, уменьшая тем самым число состояний в

результирующей модели (см. [35]). Таким образом, стратегия построения теста может варьироваться независимо от спецификаций.

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 определенных в спецификациях критериев.

Тестовые сценарии, написанные в терминах спецификаций, тем самым определяют абстрактные тесты, которые можно использовать для тестирования любой системы, имеющей данную функциональность. Кроме того, сценарии, оформляемые на расширении объектно-ориентированных языков, имеют дополнительные возможности повторного использования при помощи механизма наследования. Сценарий, наследующий другому, может переопределить в нем процедуру вычисления состояния и переопределить или пополнить набор тестовых воздействий, оказываемых на систему в каждом состоянии.

Помимо указанных выше возможностей, тестовые сценарии UniTESK дают возможность проводить тестирование, основанное на обычных тестовых вариантах, т.е. последовательностях воздействий, формируемых по указанному разработчиком теста правилу.

Построение медиаторов

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 сопоставлять с требованиями и можно использовать для тестирования нескольких версий одной системы или нескольких систем с одинаковой функциональностью. Тесты становятся более абстрактными и

многократно используемыми. Помимо этого преимущества в качестве дополнительных выгод, предоставляемых таким способом организации тестов, можно указать поддержку ко-верификационного процесса разработки и инфраструктуры распространения программных компонентов с открытыми спецификациями и тестами функциональности (см. выше раздел о месте UniTESK в жизненном цикле ПО).

Медиаторы можно разрабатывать вручную и определять при этом довольно сложные преобразования между интерфейсом модели и интерфейсом реализации. 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.

Кроме передачи вызовов из тестов в тестируемую систему, а их результатов в обратном направлении, медиатор должен осуществлять синхронизацию состояний модели и тестируемой системы. UniTESK позволяет определить процедуру синхронизации состояний в медиаторе двумя способами.

- Если модельное состояние целиком строится на основе доступной достоверной информации о состоянии реализации, независимо от вызываемых операций, такое тестирование называется *тестированием с открытым состоянием*. Процедура построения модельного состояния при этом оформляется в виде отдельной функции в медиаторе, автоматически вызываемой тестовой системой после каждого вызова любой из тестируемых операций.

  Такая организация тестов более удобна для анализа обнаруживаемых ошибок, поскольку ошибка всегда связана с работой последней вызванной операции. Однако при этом требуются надежные процедуры для получения необходимой части информации о внутреннем состоянии

тестируемой системы. Кроме того, при тестировании параллелизма и асинхронного поведения такая организация тестов также не подходит, поскольку может привести к построению некорректного модельного состояния.

- Если же надежная информация, достаточная для построения модельного состояния, недоступна или выполняется тестирование параллелизма или асинхронного поведения, используется *тестирование со скрытым состоянием*. В этом случае модельное состояние после вызова некоторой операции строится на основе предшествовавшего вызову модельного состояния, а также 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 процедуру построения модельного состояния после вызова этой операции. Однако анализ ошибок при таком тестировании более сложен, поскольку обнаруженное тестом нарушение может быть следствием ошибки, случившейся при работе одного из предшествовавших вызовов операций.

Унифицированное 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, типов данных и реализована инструментальная поддержка такого преобразования.

Однако, во многих случаях, специализированные языки формальных спецификаций тяжело использовать для тестирования из-за трудностей при

определении указанных преобразований. Эти трудности связаны с несовпадением парадигм, лежащих в основе спецификационных языков и обычных языков программирования, с отсутствием в спецификационных языках аналогов широко используемых понятий обычных языков, например, указателей, с несовпадением семантики базовых типов и операций над ними, и пр. Поэтому определение необходимых правил преобразования требует обычно больших затрат труда высококвалифицированных специалистов, хорошо знакомых с обоими языками. Кроме того, обучение работе с полученными инструментами также весьма трудоемко и начинает давать практические результаты только по истечении значительного времени.

Чтобы сделать технологию более доступной обычным разработчикам, и для облегчения разработки медиаторов UniTESK предлагает использовать для представления спецификаций и сценариев расширения широко используемых языков программирования. Для этого построена [12,13,31] (см. также раздел об архитектуре средств поддержки UniTESK) 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 из этих понятий сформулированы правила дополнения языка соответствующей конструкцией. Для языков, в которых уже имеются конструкции для ряда из указанных понятий, пополнение производится только дополнительными конструкциями.

Значительное преимущество использования расширения языка, на котором написана тестируемая система, для спецификаций состоит в том, что связывать такую спецификацию с реализацией гораздо проще. При использовании расширения языка программирования, обучиться работе с ним достаточно быстро может обычный разработчик, имеющий опыт работы с базовым языком.

Возникающие при использовании расширений языков программирования проблемы могут быть решены при помощи различных техник.

- Проблема недостаточной выразительности языков программирования по сравнению с языками формальных спецификаций в большинстве современных объектно-ориентированных языков достаточно просто решается при помощи использования библиотек абстрактных типов.
- Проблема возможной зависимости смысла спецификации от платформы может решаться несколькими способами.
    - В спецификациях запрещено использование конструкций, имеющих недостаточно четкий смысл и по-разному интерпретируемых для разных платформ. Чаще всего такие конструкции могут быть без потери смысла заменены другими конструкциями того же языка, с более точно и переносимым образом определенной семантикой.
    - Рекомендуется использовать библиотеки, реализованные так, чтобы работать одинаково на всех поддерживаемых ими платформах.
    - В особо специфических случаях проводится удаленное тестирование, при котором тестовая система исполняется на той же платформе, на которой разрабатывались спецификации, а передача генерируемых ею тестовых воздействий на машину, где работает тестируемая система, осуществляется по сети.

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, содержащих следующую информацию в понятном для обычного программиста виде.

- Структуру конечного автомата, использованного для построения сценария. В отчете имеется графическое изображение автомата, а также таблица всех его состояний и выходящих из них переходов, с указанием конечного состояния каждого перехода и количества его выполнений во время тестирования.

- Достигнутые значения для тестового покрытия по всем критериям, определенным в затронутых тестом спецификациях. Эти значения представлены как для отдельных спецификаций, так и суммарно по компонентам и подсистемам, в которые сгруппированы спецификации операций тестируемой системы.
- Информацию об обнаруженных в ходе теста нарушениях, связанных с ошибками в тестируемой системе или в спецификациях, сценариях и медиаторах.

  Для каждого нарушения указывается его вид, действие, при котором оно зафиксировано, текстовые сообщения полученных исключений, если таковые были созданы, и пр.

  Нарушения группируются по ряду признаков, которые помогают отнести их к проявлениям одной и той же ошибки.

  Кроме того, при подключении базы данных по уже зафиксированным ошибкам, при обнаружении у нарушения характерных признаков такой ошибки появляется указание на то, что данное нарушение, скорее всего, является проявлением этой известной ошибки, и ссылка на ее описание.

Математические основы технологии UniTESK

Данный раздел содержит описание некоторых математических моделей, используемых в рамках технологии UniTESK, и формулировку ряда результатов, обосновывающих применимость техник UniTESK для решения задач тестирования.

Базовые модели и техники

В основе технологии UniTESK лежат два вида математических моделей тестируемого программного обеспечения.

Модели первого вида — *событийные контракты* [31,32] — используются в качестве формального представления требований к поведению сложных программных систем.

Модели второго вида — разновидности *автоматов ввода-вывода* [29,36,37] — используются для формального представления тестов и анализа тестовых возможностей.

Приведем здесь определение только моделей второго вида.

Автоматом ввода-вывода или *системой переходов по вводу-выводу* L называется пятерка (Q, I, O, T, Q_0), состоящая из следующих элементов.

- Q — непустое множество, элементы которого называются *состояниями* L. $Q_0 \subseteq Q$ — подмножество *начальных состояний* L.
- I и O — непересекающиеся множества меток, метки из I называются *стимулами* или *входными символами*, метки из O — *реакциями* или *выходными символами*. Кроме этого имеется специальный символ τ, не являющийся ни входным, ни выходным.
- $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ — *набор переходов* L. Переход (q_1, x, q_2) считается начинающимся в состоянии q_1, заканчивающимся в состоянии q_2 и помеченным символом x.

При построении обхода переходов конечно-автоматной модели обходчиком применяется один из алгоритмов построения обхода графа, использующих только ту информацию о его структуре, которая выявлена уже выполненными переходами по его ребрам. Алгоритмы, обладающие таким свойством, называются *неизбыточными*. Несколько неизбыточных алгоритмов построения обхода всех ребер графа для разных классов графов описаны в работах [38-41].

Предположения о тестируемой системе и гарантируемые результаты

Предположения о тестируемой системе, на которых основываются методы построения тестов, используемые в UniTESK, можно сформулировать следующим образом.

- Внешние условия, существенные с точки зрения возможных ошибок системы, либо полностью управляемы через определенный интерфейс, либо все различные варианты внешних условий проявляются за

фиксированное количество повторений одного и того же тестового воздействия в одном и том же внутреннем состоянии тестируемой системы.

В первом случае для обеспечения необходимой полноты тестирования обращения, устанавливающие некоторые внешние условия, вносятся в набор тестовых воздействий. В отличие от остальных воздействий, они не нуждаются в проверке корректности результатов их работы.

Во втором случае для обеспечения полноты тестирования каждое воздействие в некоторой ситуации приходится повторять несколько раз.

Эти предположения выполняются для систем, которым не нужно реагировать на определенные условия внешней среды, помимо прямых обращений к ним человека или других систем. Для других классов систем иногда существует программный интерфейс или набор конфигурационных файлов, позволяющий полностью управлять информацией, воспринимаемой ими извне помимо прямых обращений.

- Интерфейс системы сводится к программному. При этом управление всеми аспектами ее поведения может быть целиком возложено на программу, в частности, все тесты можно считать программами.

  Для многих классов практически важных систем это допущение выполнено или может быть реализовано с помощью специализированных библиотек или инструментов, например, предоставляющих программный интерфейс для эмуляции воздействий пользователя на графический интерфейс пользователя.

Помимо этих ограничений, позволяющих проводить тестирование теми методами, которые используются в UniTESK, для гарантий адекватности этих методов необходимы дополнительные гипотезы об устройстве тестируемой системы.

- Реальное поведение тестируемой системы, включая проявления всех имеющихся в ней ошибок, может быть полностью описано некоторым автоматом ввода-вывода.

- Предусловия, извлеченные из требований, адекватно описывают безопасные для тестируемой системы действия. То есть, если в некоторой ситуации определенное действие может привести к разрушению тестируемой системы, его предусловие запрещает выполнение этого действия в соответствующей модельной ситуации.
- Соответствие между модельным представлением внутреннего состояния системы и ее реальным внутренним состоянием однородно относительно содержащихся в ней ошибок. То есть, если ошибка возможна в определенном состоянии, она возможна и в любом другом состоянии с тем же модельным представлением, и, соответственно, может быть обнаружена тестами, построенными на основе этой модели.
- Критерий полноты тестирования, используемый при построении тестов, адекватен, то есть в каждом классе воздействий, рассматриваемых как эквивалентные с точки зрения этого критерия в тестируемой системе либо нет ошибок, либо ошибка проявляется при любом воздействии этого класса.
- Автомат ввода-вывода, построенный на основе используемого критерия тестового покрытия по методике, описанной в разделе о построении тестовых сценариев, допускает обход по стимулам из любого своего состояния. Это условие может быть формализовано требованием сильной Δ-связности графа его переходов [41].

При выполнении этих гипотез можно строго доказать [12,37,40,41], что используемые в UniTESK техники построения тестов гарантируют обнаружение всех несоответствий между поведением тестируемой системы и требованиями.

Архитектура инструментальных средств поддержки технологии UniTESK

Данный раздел описывает общую архитектуру инструментальных средств поддержки технологии UniTESK и основные функции и особенности реализации каждого инструмента, входящего в набор средств поддержки.

Средства поддержки технологии UniTESK на некоторой языковой платформе, т.е. для ПО, написанного на некотором языке программирования, включают следующие составляющие.

- *Расширение базового языка программирования* с помощью общего набора необходимых для разработки тестов по UniTESK конструкций. Это расширение предназначено для того, чтобы на нем разрабатывались спецификации, сценарии и медиаторы.
- *Инструменты поддержки разработки тестов*. Этот набор инструментов облегчает разработку и сборку набора спецификаций, медиаторов и сценариев. Он реализуется как модуль поддержки разработки на расширении базового языка в одной из широко используемых сред разработки программного обеспечения. Такой модуль включает в себя редактор текстов на расширении базового языка, библиотеку типов внутренних данных, пользовательский интерфейс для вызова транслятора расширения базового языка, инструмента выполнения тестов и инструментов генерации отчетов.
- *Транслятор* расширения базового языка программирования. Он преобразует спецификации, сценарии и медиаторы на расширении базового языка в компоненты тестового набора на самом языке программирования, а затем — в исполнимый набор тестовых программ. Последний этап выполняется компилятором самого языка программирования, который только запускается транслятором.
- *Библиотека поддержки выполнения тестов*. Она включает необходимые для выполнения тестов компоненты — обходчики, подсистему сбора трассы, подсистему автоматической синхронизации состояния модели и пр., а также библиотеки вспомогательных компонентов для решения ряда задач организации и выполнения тестов — библиотеки итераторов данных простых типов, библиотеки классов коллекций, часто используемых при моделировании и т.д.

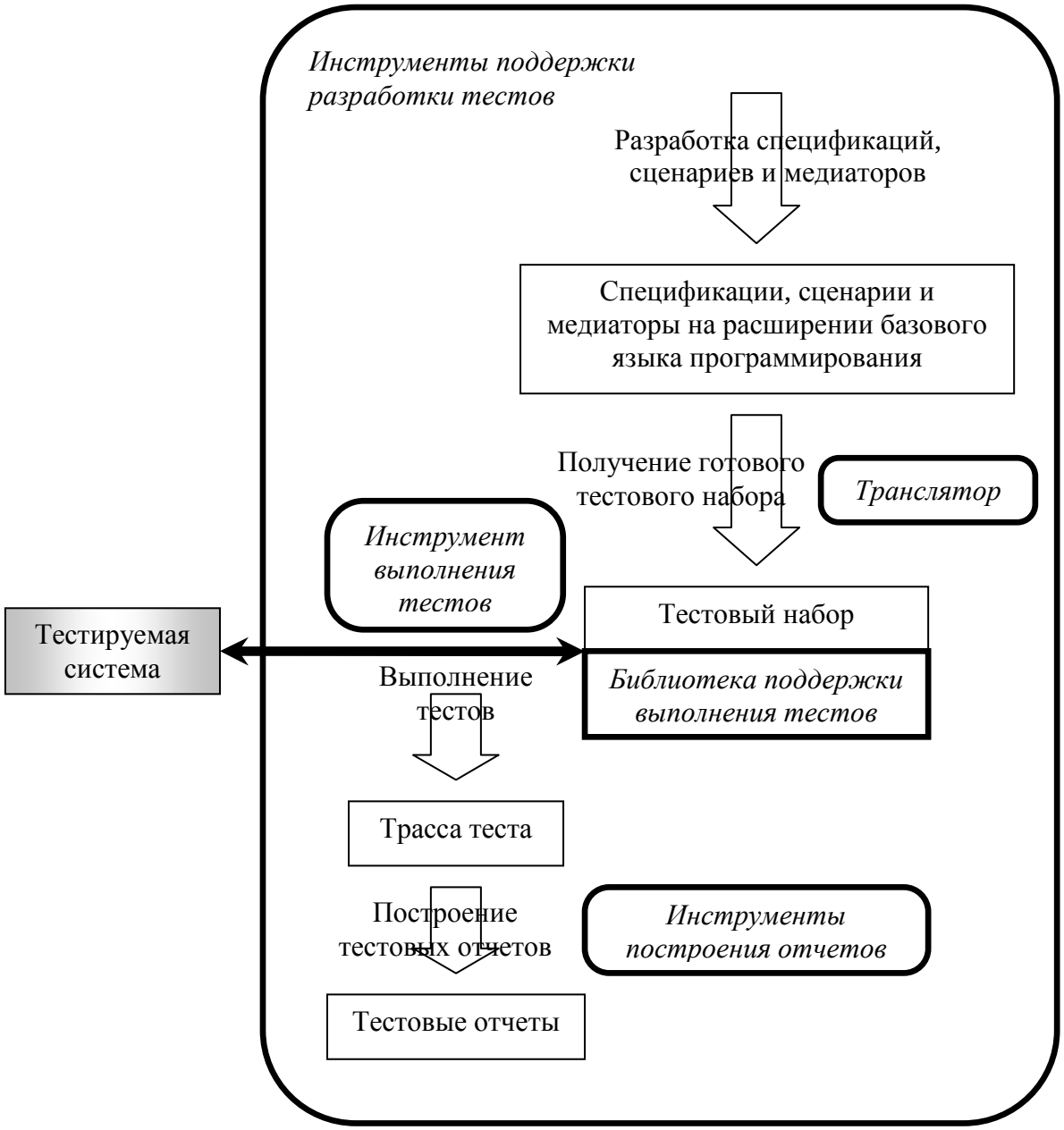

Рисунок 5. Использование инструментов в ходе разработки тестов по технологии UniTESK.

- *Инструмент выполнения тестов*. На некоторых языковых платформах для повышения удобства запуска тестов требуется дополнительная автоматизация. В этом случае создается дополнительный инструмент запуска тестов, задача которого — запускать на выполнение тесты по указанию идентификаторов соответствующих им тестовых сценариев, с определяемыми при запуске значениями параметров. Для некоторых платформ, например, для языка C, такой инструмент не нужен.

- *Инструменты построения отчетов* по трассе теста. Они предназначены для анализа полученной в ходе тестирования трассы, объединения содержащейся в ней информации с данными из других источников, например, из базы данных известных ошибок, и генерации на этой основе наглядных отчетов, содержащих основные результаты тестирования.

Инструменты генерации отчетов, в отличие от других составляющих, реализованы независимо от базового языка программирования.

Использование средств поддержки технологии UniTESK в процессе разработки тестов проиллюстрировано на Рис. 5.

За 2000-2003 годы средства поддержки технологии UniTESK были реализованы для трех широко используемых языков программирования — Java, C и C#.

Общая структура расширения базового языка программирования

Для поддержки методов разработки тестов на основе формальных спецификаций базовый язык расширяется следующим набором конструкций [31,42]. Если конструкция с аналогичной семантикой уже присутствует в базовом языке, соответствующая дополнительная конструкция не вводится.

- Спецификационный класс или структура.

  Эта конструкция предназначена для совместного описания группы логически связанных спецификаций операций и событий и общего для них модельного состояния.

  В объектно-ориентированных языках она представляется в виде класса, помеченного специальным модификатором.

  В рамках спецификационных классов или структур используются следующие дополнительные конструкции.

  - Спецификация операции или события.

    Оформляется как функция или метод в базовом языке с дополнительными модификаторами.

- Блоки предусловия и постусловия.

  Являются частью спецификации операции или события. Каждый такой блок содержит тело функции, возвращающей в качестве результата логическое значение.
- Описания критериев покрытия.

  Могут иметь два вида. Описание первого вида представляет собой блок кода, возвращающий в качестве результата идентификатор элемента покрытия. Описание второго вида представляет собой перечисление предикатов, соответствующих отдельным элементам покрытия.
- Оператор пре-выражения. Обозначает значение соответствующего выражения до вызова операции или возникновения события. Может использоваться в постусловиях.
- Идентификатор результата операции. Обозначает результат работы данной операции или значение немедленной реакции на данное событие. Может использоваться в постусловиях.
- Операторы выделения ветвей функциональности. Могут использоваться в постусловиях для определения ветвей функциональности данной операции или стимула.
- Описания связанных событий.

  В спецификацию стимула могут включаться спецификации других типов событий, которые всегда происходят в качестве реакции на стимул данного типа.

- Медиаторный класс или структура.

  Эта конструкция предназначена для совместного описания медиаторов для группы логически связанных спецификаций операций и событий, а также процедур синхронизации общего модельного состояния.

  В объектно-ориентированных языках она представляется в виде класса, помеченного специальным модификатором.

В рамках медиаторных классов или структур используются следующие дополнительные конструкции.

- Медиатор операции или события.

  Оформляется как функция или метод в базовом языке с дополнительными модификаторами и указанием привязки к определенной спецификационной операции или событию.

- Блоки синхронизации состояния.

  Описывают процедуру синхронизации модельного состояния. Могут входить как непосредственно в медиаторный класс или структуру, так и в описание медиатора операции или события.

- Сценарный класс или структура.

Эта конструкция предназначена для совместного описания процедуры вычисления состояния автоматной модели сценария, набора тестовых воздействий, которые можно выполнять в произвольном состоянии, и указания используемого для выполнения сценария обходчика.

В объектно-ориентированных языках она представляется в виде класса, помеченного специальным модификатором.

В рамках сценарных классов или структур используются следующие дополнительные конструкции.

- Блок вычисления состояния.

  Вычисляет и возвращает объект, представляющий текущее состояние автоматной модели сценария.

- Сценарные функции или методы.

  Каждая сценарная функция описывает параметризованный набор тестовых воздействий, которые можно выполнять в произвольном состоянии.

  Сценарная функция может описывать последовательность выполнения некоторых действий, используя при этом условные операторы или операторы циклов.

- Блоки выделения тестовых фреймов.

  Служат для выделения наборов действий, которые должны быть выполнены в рамках одного фрейма. Используются в сценарных функциях.
- Итерационные блоки.

  Служат для указания набора значений параметров, с которыми должен быть выполнен набор действий в рамках сценарной функции.
- Оператор фильтрации.

  Указывает, что некоторые действия в сценарной функции нужно выполнять только в том случае, если они увеличивают тестовое покрытие по заданному критерию. Может использоваться только для критериев покрытия, зависящих только от параметров операций и состояния, в котором эти операции выполняются.

Помимо указанных, вводятся еще два вида конструкций, которые могут использоваться всюду в спецификациях, сценариях и медиаторах.

- Оператор создания медиатора.

  Вводится в объектно-ориентированных языках для поддержки однозначного соответствия между медиаторами в тестовой системе и интерфейсными объектами в тестируемой системе.
- Оператор метки, используемой для оценки покрытия.

  Используется для определения специальных критериев покрытия и сбора информации о достигнутом покрытии по такому критерию.

Архитектура библиотеки поддержки выполнения тестов

Библиотека поддержки выполнения тестов состоит из следующих элементов.

- Базовые типы данных для результатов трансляции конструкций расширения, основные операции для работы с этими типами, а также ряд вспомогательных типов для представления событий, возникающих во время выполнения тестов, данных о тестовом покрытии и пр.

- Типы и операции для поддержки синхронизации отдельных потоков и событий во время выполнения тестов, а также для выполнения сериализации событий при тестировании параллелизма и асинхронного поведения.
- Типы и операции для поддержки синхронизации модельного состояния и однозначного соответствия между медиаторами и интерфейсными объектами тестируемой системы.
- Типы исключительных ситуаций, представляющих нарушения, возникновение которых возможно во время выполнения тестов.
- Типы и операции для поддержки и управления генерацией трассы теста.
- Библиотека компонентов-обходчиков разных видов.
- Библиотека вспомогательных классов-коллекций, часто используемых в спецификациях.
- Библиотека итераторов данных основных типов базового языка.

Архитектура транслятора расширения базового языка

Транслятор расширения базового языка предназначен для трансляции спецификаций, сценариев и медиаторов на расширении базового языка программирования в код на самом этот языке. Помимо этого он выполняет компиляцию полученного кода в готовый к исполнению тестовый набор при помощи вызова компилятора базового языка программирования.

Поскольку во всех случаях средства поддержки технологии UniTESK реализовывались для широко используемых языков программирования, использовался один из уже имеющихся компиляторов базового языка.

Основные компоненты транслятора расширения базового языка следующие.

- Парсер расширения базового языка.

  Он обладает функциональностью и парсера подмножества самого базового языка, использование которого допустимо в спецификациях, медиаторах и сценариях.

  По исходному тексту программы на расширении базового языка строит ее дерево абстрактного синтаксиса.

Такой парсер разрабатывается с помощью одной из технологий построения парсеров. В имеющихся реализациях средств поддержки UniTESK использовались два инструмента поддержки разработки трансляторов: ANTLR [43,44] и JavaCC [45].

- Библиотека типов внутренних данных транслятора (модель программы).

  Предназначена для представления внутренних данных транслятора. Такое представление обеспечивает единый интерфейс между его компонентами, а также возможность доступа других инструментов к этим данным.

  Типы этой библиотеки являются типами узлов дерева абстрактного синтаксиса программ на расширении базового языка.

- Семантический анализатор базового языка.

  Определяет семантические атрибуты узлов деревьев абстрактного синтаксиса и анализирует корректность программ с точки зрения правил статической семантики базового языка.

- Семантический анализатор расширения базового языка.

  Определяет дополнительные семантические атрибуты узлов деревьев абстрактного синтаксиса и анализирует корректность программ с точки зрения правил статической семантики расширения базового языка.

- Транслятор абстрактного синтаксиса.

  По дереву абстрактного синтаксиса программы на расширении базового языка программирования строит соответствующую программу на самом этом языке.

  Правила преобразования, на основе которых создается транслятор абстрактного синтаксиса, описываются в виде отдельного проектного документа — схемы трансляции расширения базового языка.

Разработка трансляторов расширений языков программирования и разработка библиотек типов внутренних данных выполнялась на основе технологии разработки инструментов для работы с формальными языками.

Архитектура инструментов построения тестовых отчетов

Инструменты построения тестовых отчетов состоят из следующих компонентов.

- Парсер трассы теста.
- Анализатор данных об известных ошибках в тестируемой системе.

  Используется, если есть база данных известных ошибок.
- Библиотека типов внутренних данных, представляющих результаты тестирования (модель данных трассы теста).
- Модуль группировки информации о нарушениях, объединяющий найденные тестом нарушения в группы при совпадении определенных характеристик.
- Генератор отчетов для каждого вида отчетов.

  На данный момент имеется 3 вида отчетов, представляемых в виде гипертекстовых HTML-файлов — отчет о найденных нарушениях, отчет о структуре автоматной модели сценария, отчет о тестовом покрытии по определенным в спецификациях критериям, — и 2 вида отчета в виде графических диаграмм — граф переходов автоматной модели сценария и MSC-диаграмма выполнения теста.

При разработке инструментов построения отчетов тоже использовалась технология разработки инструментов для работы с формальными языками, созданная в рамках диссертации А. В. Демакова [46,47].

Практическое использование технологии UniTESK

В данном разделе представлены результаты практического применения технологии UniTESK в ряде пилотных и промышленных проектов по разработке тестов для сложного программного обеспечения.

После создания первых версий инструментов, поддерживающих технологию, в 2001-2005 годах они использовались примерно в двух десятках пилотных проектах по разработке тестов для программных систем различных классов.

- Для компонентов операционной системы Tiny OS.

- Для основных операций файловой системы в рамках интерфейса POSIX, полученные тесты выполнялись на операционных системах Linux и Windows NT.
- Для части стандартных библиотек языков программирования C и Java.
- Для отдельных компонентов информационной системы коммерческой компании.
- Для части банковской системы управления данными о клиентах.
- Для компонентов системы управления равноранговыми соединениями.
- Для компонентов библиотек поддержки выполнения тестов в самих инструментах UniTESK.
- Для протокола защиты цифровых данных IPMP2.

Во всех этих проектах достаточно быстро обнаруживались те или иные ошибки в тестируемом ПО, а трудоемкость разработки тестов оставалась на приемлемом уровне. Поэтому были сделаны выводы о возможности использования UniTESK в более масштабных проектах.

Проекты разработки тестов для реализаций телекоммуникационных протоколов

Первые промышленные проекты с использованием технологии UniTESK были связаны с разработкой тестов для тестирования реализаций телекоммуникационных протоколов на соответствие стандартам. Всего с 2001 по 2006 год было проведено 3 таких проекта.

- Разработка тестов и тестирование реализации Интернет-протокола нового поколения IPv6 от Microsoft Research [17,48-50]. Выполнялся в 2001-2002 годах. В ходе проекта были обнаружены критические ошибки, позволяющие при помощи посылки определенным образом сформированного сообщения вызвать перезагрузку любого узла в сети, работающего под управлением Windows 2000 с указанной реализацией протокола IPv6.

  Несмотря на то, что тестирование той же реализации проводилось силами

нескольких других групп исследователей по всему миру, другими группами эти ошибки не были обнаружены.

- Разработка тестов для реализации мобильного Интернет-протокола нового поколения Mobile IPv6 Draft 13 в Windows XP и Windows CE [51]. Выполнялся в 2002-2003 годы. В его ходе были найдены критические ошибки, в частности, отступления реализации от стандарта, препятствующие ее корректному функционированию в сетях IPv6.
- Разработка тестов проверки соответствия стандартам служебных протоколов сетей IPv6 [52,53]. Выполнялся в 2003-2006 годы. В ходе этого проекта были разработаны формальные спецификации и тестовые наборы для двух служебных протоколов IPv6 — протокола безопасности IPsec и окончательной версии стандарта протокола мобильности Mobile IPv6. Было проведено тестирование их открытых реализаций. Выявлены критические ошибки, приводящие к перезагрузке операционной системы, и несоответствия стандартам, препятствующие нормальному функционированию реализаций в сетях IPv6.

В настоящее время выполняются работы по новому проекту, рассчитанному на три года и имеющего целью разработку тестового набора для второй версии протокола безопасности IPsec.

Проект разработки тестов для части инфраструктуры поддержки языка Java

Проект разработки тестов для части инфраструктуры поддержки языка Java проводился в 2005 году по заказу коммерческой компании. В тестируемую систему входил интерфейс внешнего управления работой виртуальной машины Java и несколько компонентов стандартной библиотеки языка Java.

Поскольку механизм управления виртуальной машиной имеет интерфейс на языке C, для его тестирования использовался инструмент, поддерживающий технологию UniTESK для ПО, разработанного на C. Для тестирования библиотек Java применялся аналогичный инструмент для языка Java.

В результате проекта был разработан набор тестов, исходный код которого на расширениях C и Java занимал около 30000 строк.

При тестировании с помощью полученного набора тестов реализаций Java от компаний Sun и Bea были обнаружены несколько серьезных ошибок, некоторые из которых приводят к прекращению работы Java-машины. Было также обнаружено несколько несоответствий между функциональностью, описанной в стандартной документации по Java, и реализованной в тестируемых системах.

Проект разработки тестов для ОС 2000

Проект разработки тестов на соответствие стандарту POSIX и требованиям документации для интерфейсных библиотек операционной системы реального времени ОС 2000 проводился в 2005-2006 годах. Этот проект выполнялся Институтом системного программирования РАН в сотрудничестве с Научно исследовательским институтом системных исследований РАН, который являлся заказчиком проводимых работ.

В ходе проекта заказчиком отмечено существенное увеличение полноты проводимого тестирования и качества тестов, что позволило заметным образом повысить качество и надежность версий ОС 2000, выпущенных с учетом результатов проекта.

Проект разработки тестов для Linux Standard Base

Проект разработки тестов для тестирования соответствия стандарту Linux Standard Base (LSB) выполнялся в 2005-2006 годах [25,33,34] в Центре верификации Linux при Институте системного программирования РАН. Его целью было создание тестов на основе формальных спецификаций требований этого стандарта в версии 3.1 к поведению 1532 функций, описанных в разделе LSB Core.

Непосредственно в стандарте LSB описываются требования только к 15% этих функций, требования к остальным функциям формулируются в других стандартах — POSIX, X/Open Curses, System V Interface Definition,

ISO/IEC 9899 (стандарт на язык программирования C) — на которые LSB просто ссылается. Все вместе эти требования занимают около 6000 страниц текста.

В результате этого проекта создан тестовый набор, содержащий формальные спецификации и тесты для всех 1532 функций LSB Core 3.1. Их исходный код на расширении языка C занимает около 500 тысяч строк.

При формализации требований стандартов было сформулировано около 100 замечаний к тексту стандартов. Каждое такое замечание означает неясность, неоднозначность и противоречие в стандарте. 67 замечаний принято разработчиками стандартов к исправлению.

Кроме того, несколько ошибок было обнаружено в реализации стандартной библиотеки языка C glibc, содержащей многие затрагиваемые в проекте функции.

В настоящее время проводится проект по разработке тестов проверки соответствия для остальных частей стандарта LSB, охватывающих около 30 тысяч библиотечных функций.

Сравнение с другими подходами к автоматизации тестирования

Данный раздел содержит обзор других промышленно применимых подходов к автоматизации разработки тестов и их сравнение с технологией UniTESK.

Многие элементы этой технологии используются в других подходах к разработке тестов, однако ни один из таких подходов, предлагаемых в академическом сообществе или используемых в индустрии разработки ПО, не обладает всей совокупностью ее характеристик.

В этом обзоре упоминаются только методы разработки тестов, поддержанные инструментами и нацеленные на использование в промышленной разработке ПО. Таким образом, множество интересных исследовательских методов осталось за его рамками, поскольку они пока не могут применяться в масштабах, сравнимых с UniTESK.

Архитектура теста

Большинство имеющихся подходов к разработке тестов используют 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 тестируемой системы в определенной ситуации. Такая архитектура используется часто и при автоматическом построении тестов, но при этом имеет ограничения, связанные с огромным количеством генерируемых тестовых вариантов и трудностью управления полученным тестовым набором.

В UniTESK тесты строятся в виде сценариев, каждый из которых содержит целый набор тестовых вариантов, проверяющих работу тестируемой системы при обращении к выделенной группе интерфейсов во многих различных ситуациях. В результате, в тестовом наборе больше уровней иерархии, что удобно при тестировании больших и сложных систем. Кроме того, сокращается размер кода, соответствующего одному тестовому варианту. Поэтому тестовый набор UniTESK с той же полнотой тестирования, что и традиционный, существенно компактнее. Для регрессионного тестирования обе схемы пригодны в равной степени, поскольку при этом обычно требуется возможность прогона всего набора тестов.

Автоматическое построение тестовых оракулов

Автоматическое построение тестовых оракулов на основе спецификаций отличает UniTESK от инструментов, предназначенных только для автоматизации выполнения тестов, таких как JUnit [54] для языка Java и его многочисленные клоны, предназначенные для других языков программирования. Вместе с тем, построение оракулов из контрактных спецификаций поддерживается многими существующими инструментами [55,56], например, следующими.

- iContract [57], JMSAssert [58], JML [59,60], jContractor [61], Jass [62], Handshake [63], JISL [64] используют контрактные спецификации,

написанные в исходном коде тестируемой системы в виде комментариев на расширении Java.

- SLIC [65] позволяет оформлять контрактные спецификации на расширении C с использованием предикатов временных логик
- Rational Test RealTime [66] от IBM использует контракты и описание структуры конечно-автоматной модели тестируемого компонента в виде специальных скриптов
- JTest/JContract [67] от Parasoft и Korat [68] позволяют оформлять предусловия, постусловия и инварианты в виде особых комментариев в Java-программах
- ATG-Rover [69] использует спецификации в виде программных контрактов-комментариев на C, Java или Verilog, которые могут содержать предикаты временных логик LTL или MTL
- Семейство инструментов ADL [23] основано на расширениях C, C++, Java и IDL, которые используются для разработки контрактных спецификаций, не привязанных жестко к конкретному коду
- T-VEC [70] использует пред- и постусловия, оформленные в виде таблиц в нотации SCR [71]
- Инструмент SpecExplorer [72,73] использует для построения тестов пред- и постусловия на расширении языка C#, названном Spec#.

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 на основе сценариев.

В инструменте JTest от Parasoft есть возможность автоматической генерации тестовых последовательностей, но получаемые 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 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 тестируемый объект в нужном состоянии. С другой стороны, используемые спецификации должны быть жестко привязаны к реализации и не должны допускать таких состояний тестируемого компонента, которые не могут возникнуть в ходе его работы, иначе много сгенерированных тестов будут соответствовать недостижимым состояниям.

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, обязательно ли эти шаблоны должны 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», т.е. соответствующих ветвям 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, авторам 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 количество различных операций, которые были вызваны.

Инструмент SpecExplorer разработан в Microsoft Research в 2004 году. Он поддерживает построение тестов на основе пред- и постусловий операций, дополненных полной исполнимой моделью их корректной работы. При построении тестов возможен учет только определенных элементов или свойств модельного состояния системы. Тестовая последовательность строится как путь по графу переходов модели поведения, описанной в спецификациях. В целом SpecExplorer имеет функциональность, аналогичную инструментам UniTESK, за исключением возможности отслеживания тестового покрытия по критериям, основанным на структуре спецификаций. Стоит отметить, что его разработчики постоянно взаимодействуют с авторами данной работы с 2001 года и идея использования контрактных спецификаций была заимствована ими из презентаций технологии UniTESK.

Генерация тестовых последовательностей

Генерация тестовых последовательностей поддерживается многими инструментами, использующими модели тестируемой системы в виде автоматных моделей различного рода: конечных автоматов, систем переходов, расширенных автоматов, взаимодействующих автоматов, автоматов ввода/вывода, сетей Петри и пр.

Такие инструменты используют для представления моделей формальные языки спецификаций SDL, LOTOS, Lustre или графический язык Statecharts. Стоит отметить также использование моделей в виде набора возможных сценариев работы тестируемой системы, чаще всего оформляемых в нотации Message Sequence Charts (MSC). В настоящее время как Statcharts, так и MSC являются частью универсального языка моделирования UML 2.0, поэтому многие современные инструменты используют его для представления моделей поведения тестируемой системы.

Наиболее известными инструментами, поддерживающими автоматическую генерацию тестовых последовательностей, являются следующие.

- SpecExplorer [72,73], описанный выше, и его предшественник AsmL Tester [74]. AsmL Tester построен на тех же принципах, что SpecExplorer, но использует исполнимую модель в виде машины с абстрактным состоянием на специальном языке AsmL.
- AGEDIS [75,76] является общим интерфейсом для нескольких инструментов построения тестов на основе моделей. Среди этих инструментов наиболее часто используются TGV (см. ниже) и разработанный в IBM инструмент GOTCHA-TCBeans.

  GOTCHA-TCBeans использует автоматные модели на языке Murφ. Помимо модели поведения, возможно задание тестовых директив, определяющих стратегию тестирования и играющих роль, аналогичную тестовым сценариям. Кроме того, инструмент поддерживает редукцию автоматных моделей за счет отбрасывания части данных, определяющих состояние системы.
- Инструменты AutoFocus [77,78] и TGV [79,80], хотя разработаны первый — в Техническом университете Мюнхена, Германия, второй — в совместном проекте IRISA и INRIA, Франция, построены на общих принципах. Оба используют автоматную модель поведения системы и набор целей тестирования, заданных в виде сценариев. TGV встроен в набор инструментов для моделирования и верификации Ceasar-Aldebaran Toolbox (CADP), а также имеет интерфейс AGEDIS.
- Инструменты Qtronic [81] и Test Generator [82] от Conformiq, Lerious Test Generator [83], Reactis Tester [84] и Tau Tester [85] от Telelogic используют для построения тестов модели поведения в виде диаграмм Statecharts. Большинство из них предназначено для тестирования телекоммуникационных систем и генерирует тесты в виде программ на специализированном языке TTCN-3.

- На похожих принципах построены инструменты GATeL [86] и Lurette [87], использующие язык Lustre для описания автоматной модели поведения, а также Simulink Tester [88], использующий графические модели на языках Simulink и Stateflow.
- Инструмент TorX [89,90], разработанный в университете города Твент, Голландия. Так же, как и TGV, описанный выше, он встроен в набор инструментов CAPD. Так же, как и TGV, он использует модель поведения в виде системы помеченных переходов, но, в отличие от TGV, генерирует тесты при помощи случайного выбора путей в этой модели.

Многие из перечисленных инструментов позволяют указывать определенные правила выбора путей в автоматной модели, аналогично тому, как это делают тестовые сценарии UniTESK.

Однако в целом использование автоматных моделей для описания сложных систем имеет два серьезных недостатка.

- Быстрый рост сложности. Сложность автоматных моделей — количество состояний и переходов — возрастает экспоненциально в зависимости от количества реализуемых функций. Автоматную модель крайне сложно декомпозировать. Контрактные спецификации, используемые в UniTESK, лишены этого недостатка и позволяют рассматривать сложную систему как набор более простых компонентов.
- Проблемы описания недетерминизма. При отвлечении от ряда несущественных характеристик тестируемой системы модели ее поведения становятся недетерминированными — входные данные и состояние уже не могут полностью определить результат работы операции. В автоматных моделях при описании недетерминизма необходимо добавлять переходы из исходного состояния во все возможные итоговые, то делает модель более сложной и запутанной. В контрактных спецификациях такое описание может быть сделано кратким и естественным, поскольку они допускают использование любых предикатов в качестве постусловий операций.

Наиболее близкие аналоги

Наиболее близки к UniTESK по поддерживаемым возможностям инструменты GOTCHA-TCBeans и SpecExplorer. Оба они используют автоматные модели тестируемого ПО. Для GOTCHA-TCBeans такая модель должна быть описана на расширении языка Murφ, второй инструмент использует в качестве спецификаций пред- и постусловия, пополненные исполнимой моделью на расширении C#.

Объединяет все три подхода использование *разных видов моделей* для построения теста, что позволяет строить более эффективные, гибкие и масштабируемые тесты, а также осуществлять повторное использование многих компонентов тестов. 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 SpecExplorer — модель поведения системы и множество наблюдаемых величин, наборы значений которых определяют состояния конечного автомата, используемого для построения тестовой последовательности.

В обоих инструментах используются техники уменьшения размера модели, аналогичные факторизации автоматной модели в UniTESK. Инструмент GOTCHA-TCBeans может применять частный случай факторизации, при котором игнорируются значения некоторых полей в состоянии исходной модели [91]. SpecExplorer может строить тестовую последовательность на основе конечного автомата, состояния которого получаются редукцией полного модельного состояния до набора значений выражений, указанных разработчиком теста [72].

Однако, для этих инструментов неизвестны практические применения такого масштаба, как для технологии UniTESK. Инструмент GOTCHA-TCBeans использовался для тестирования лишь небольших программных систем и небольших модулей аппаратного обеспечения (арифметическое устройство).

Инструмент SpecExplorer используется, в основном, внутри компании Microsoft, и хотя известен ряд его применений для тестирования отдельных компонентов ПО этой компании, их сложность ограничена несколькими десятками операций.

Заключение

Основные результаты представляемой работы можно сформулировать следующим образом.

- Разработаны методы построения тестов на основе формальных спецификаций требований к программному обеспечению.
- Созданы инструменты, автоматизирующие разработку тестов по предложенным методам для программных систем на языках программирования C, Java и C#.
- Предложены методики формализации требований стандартов, описывающих телекоммуникационные протоколы и интерфейсные библиотеки операционных систем. В совокупности с разработанными методами и инструментами построения тестов они дают возможность разрабатывать тестовые наборы для проверки соответствия этим стандартам.
- Разработанная технология применена в ряде крупных проектов по разработке тестов для промышленных программных систем. Ее применение показало приемлемую трудоемкость разработки тестов по сравнению с традиционными подходами при существенном росте их качества. Это способствовало более эффективному обнаружению проблем в тестируемом ПО, и, вследствие их исправления, значительному росту его качества и надежности.

Хотя большинство элементов технологии UniTESK используются и в других подходах к разработке тестов или имеют аналоги, как комплексный подход совокупность разработанных методов и техник является новой.

Использование в рамках UniTESK контрактных спецификаций поведения тестируемой системы и упрощенных автоматных моделей теста способствует

более масштабируемой разработке тестов для сложных программных систем и выгодным образом отличает ее почти от всех имеющихся аналогов.

Технология UniTESK является единственной технологией разработки тестов на основе формальных моделей, систематически применяемой в крупных проектах по тестированию промышленного программного обеспечения.

По совокупности полученных результатов можно заключить, что поставленные цели работы — создание и внедрение в промышленную практику технологии автоматизированного тестирования, позволяющей проводить тщательное тестирование сложных программных систем, — достигнуты.

Список литературы, опубликованной авторами по теме работы

1. И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Использование конечных автоматов для тестирования программ. Программирование, 26(2):61-73, 2000.
2. A. Petrenko, I. Bourdonov, A. Kossatchev, V. Kuliamin. Experiences in Using Testing Tools and Technology in Real-life Applications. Proccedings of SETT'2001, pp. 3-39, Pune, India, 2001.
3. I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, S. V. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'2001, Novosibirsk, Russia, LNCS 2244:301-307, Springer-Verlag, 2001.
4. A. Koptelov, V. Kuliamin, A. Petrenko. VDM++TesK: Testing of VDM++ Programs. Proceedings of 3-rd VDM Workshop pp. 41-56, Kopenhagen, Denmark, 2002.
5. I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77-88, Springer-Verlag, 2002.
6. И. Агамирзян, С. Грошев, А. Хорошилов, Г. Ключников, А. Косачев, В. Омельченко, Н. Пакулин, А. Петренко, В. Шнитман. Применение

формальных методов для тестирования MSR IPv6. Сборник тезисов международной конференции «Интернет нового поколения», Ярославль, Россия, 2002.

7. Г. В. Ключников, А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. Применение формальных методов для тестирования Mobile IPv6. Сборник тезисов 2-й международной конференции «Интернет нового поколения», стр. 20-25, Ярославль, Россия, 2003.
8. И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Асинхронные автоматы: классификация и тестирование. Труды ИСП РАН, 4:7-84, 2003.
9. Г. В. Ключников, А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. Применение формальных методов для тестирования реализации IPv6. Труды ИСП РАН, 4:121-140, 2003.
10. В. В. Кулямин, О. Л. Петренко. Место тестирования среди методов оценки качества ПО. Труды ИСП РАН, 4:163-176, 2003.
11. V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin, A. S. Kossatchev, I. B. Bourdonov. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. Proceedings of PSI'2003, Novosibirsk, Russia, LNCS 2890:450-461, Springer-Verlag, 2003.
12. И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов: детерминированный случай. Программирование, 29(5):59-69, 2003.
13. В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
14. А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. Формализация требований Интернет–стандартов для тестирования реализаций коммуникационных протоколов. Труды конференции «Научный сервис в сети Интернет», М.: МГУ, стр. 318-321, 2003.
15. V. Kuliamin, A. Petrenko, A. Kossatchev, I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. Proceedings of ECMDSE'2003, pp. 55-63, Nurnberg, Germany, 2003.

16. И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов: недетерминированный случай. Программирование, 30(1):2-17, 2004.
17. V. Kuliamin. Multi-paradigm Models as Source for Automated Test Construction. Proceedings of MBT'2004, Barcelona, Spain, 2004.
18. Н. В. Пакулин. Формальная спецификация IPsec. Сборник тезисов 3-й международной конференции «Интернет нового поколения», стр. 14-23, Москва, 2004.
19. А. В. Баранцев, И. Б. Бурдонов, А. В. Демаков, С. В. Зеленов, А. С. Косачев, В. В. Кулямин, В. А. Омельченко, Н. В. Пакулин, А. К. Петренко, А. В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. Труды ИСП РАН, 5:121-156, 2004.
20. А. С. Косачев, И. Б. Бурдонов, В. В. Кулямин. Мета-модель функциональной спецификации распределенной системы, пригодная для тестирования. Труды конференции «Научный сервис в сети Интернет», М.: МГУ, 2004.
21. V. Kuliamin, A. Petrenko. Applying Model Based Testing in Different Contexts. Proceedings of Seminar on Perspectives on Model Based Testing, Dagstuhl, Germany, 2004.
22. V. Kuliamin. Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System. Proceedings of ISOLA'2004, pp. 311-316, Paphos, Cyprus, 2004.
23. N. Pakoulin, V. Omelchenko, A. Koptelov, A. Petrenko, A. Kossatchev, C. Cheng. MPEG–2 IPMP Conformance Test Suite Development. AVS M1263, 2004.
24. Н. В. Пакулин. Применение UniTesK к тестированию встроенных систем. Труды ИСП РАН, 8(1):117-158, 2004.
25. Е. Н. Бритвина, Н. Л. Казакова, В. В. Кулямин, А. К. Петренко. UniTesK — технология тестирования программного обеспечения. Научная сессия МИФИ’2005, стр. 51-52, Москва, Россия, 2005.

26. V. Kuliamin, A. Petrenko, N. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proceedings of ISAS'2005, Berlin, Germany, LNCS 3694:68-83, Springer-Verlag, 2005.
27. V. Kuliamin, A. Petrenko, N. Pakoulin. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. Proceedings of WMSCI'2005, v. VII. Model Based Development and Testing, pp. 65-70, Orlando, USA, 2005.
28. Г. В. Ключников, Н. В. Пакулин, В. З. Шнитман. Автоматизированное тестирование сетевых сервисов Интернет-протокола. Труды конференции «Научный сервис в сети Интернет», стр. 168-170, М.: МГУ, 2005.
29. В. П. Иванников, А. С. Камкин, В. В. Кулямин, А. К. Петренко. Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения. Препринт 8 ИСП РАН, Москва, 2005.
30. I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proceedings of MBT'2006, Vienna, Austria, 2006.
31. А. В. Демаков, С. В. Зеленов, С. А. Зеленова. Генерация тестовых данных сложной структуры с учетом контекстных ограничений. Труды ИСП РАН, 9:83-96, 2006.
32. В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт 13, ИСП РАН, Москва, 2006.
33. Н. В. Пакулин, В. З. Шнитман. Валидация транспортной подсистемы распределённого приложения. Труды конференции «Научный сервис в сети Интернет», М.: МГУ, 2006.
34. A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov. Formal Methods in Industrial Software Standards Enforcement. Proceedings of PSI'2006, Novosibirsk, Russia, LNCS 4378:459-469, 2006.

35. А. В. Демаков. Язык описания абстрактного синтаксиса TreeDL и его использование. Препринт 17 ИСП РАН, 2006.
36. А. И. Гриневич, В. В. Кулямин, Д. А. Марковцев, А. К. Петренко, В. В. Рубанов, А. В. Хорошилов. Использование формальных методов для обеспечения соблюдения программных стандартов. Труды ИСП РАН, Обеспечение надежности и совместимости Linux-систем, 10:51-68, 2006.
37. В. В. Кулямин. Формальные подходы к тестированию математических функций. Труды ИСП РАН, Обеспечение надежности и совместимости Linux-систем, 10:69-114, 2006.
38. В. В. Кулямин, А. К. Петренко, В. В. Рубанов, А. В. Хорошилов. Формализация интерфейсных стандартов и автоматическое построение тестов соответствия. Труды Второй российской конференции по программной инженерии, SEC(R) 2006, Москва, Россия, 2006.
39. Н. В. Пакулин. Формализация стандартов и тестовых наборов протоколов Интернета. Диссертация на соискание учёной степени кандидата физико-математических наук. Москва, 2006.
40. А. В. Демаков. Объектно-ориентированное описание графового представления программ и моделей. Диссертация на соискание учёной степени кандидата физико-математических наук. Москва, 2006.

Список использованных источников

[1] N. Levenson, C. S. Turner. An Investigation of the Therac-25 Accidents. IEEE Computer, 26(7):18-41, July 1993.
[2] Chinook ZD 576 — Report. http://www.publications.parliament.uk/pa/ld200102/ldselect/ldchin/25/2501.htm.
[3] Ariane 5 Flight 501 Failure. Report by the Inquiry Board. http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[4] NYISO Final Report on the Blackout. http://www.nyiso.com/public/webdocs/newsroom/press_releases/2005/blackout_rpt_final.pdf.

[5] В. В. Липаев. Обеспечение качества программных систем. Методы и стандарты. М.: Синтег, 2001.

[6] В. В. Кулямин, О. Л. Петренко. Место тестирования среди методов оценки качества ПО. Труды ИСП РАН, 4:163-176, 2003.

[7] The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Report, May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[8] Software Engineering Body of Knowledge, 2004. http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf.

[9] IEEE 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology, 2002.

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

[11] B. Beizer. Software Testing Techniques. International Thomson Press, 1990.

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

[13] А. В. Баранцев, И. Б. Бурдонов, А. В. Демаков, С. В. Зеленов, А. С. Косачев, В. В. Кулямин, В. А. Омельченко, Н. В. Пакулин, А. К. Петренко, А. В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. Труды ИСП РАН, 5:121-156, 2004.

[14] M. Broy, B. Jonsson, J.-P. Katoen, A. Pretshner, eds. Model Based Testing of Reactive Systems. Advanced Lectures. LNCS 3472, Springer, 2005.

[15] A. P. Mathur. Foundations of Software Testing. Copymat Services, 2006.

[16] R. Binder. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 1999.

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

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

[19] B. Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall, 1997.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-585, October 1969.

[21] D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Transactions on Software Engineering, 24(3):161-173, 1998.

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

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

[24] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт 13, ИСП РАН, Москва, 2006.

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

[26] Н. В. Пакулин. Формализация стандартов и тестовых наборов протоколов Интернета. Диссертация на соискание учёной степени кандидата физико-математических наук. Москва, 2006.

[27] A. Kossatchev, A. Petrenko, S. Zelenov, S. Zelenova. Using Model-Based Approach for Automated Testing of Optimizing Compilers. Proccedings of Intl. Workshop on Program Undestanding, Gorno-Altaisk, Russia, 2003.

[28] V. Kuliamin, A. Petrenko. Applying Model Based Testing in Different Contexts. Proceedings of Seminar on Perspectives on Model Based Testing, Dagstuhl, Germany, 2004.

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

[30] V. Kuliamin, A. Petrenko, A. Kossatchev, I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. Proceedings of ECMDSE'2003, pp. 55-63, Nurnberg, Germany, 2003.

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

[32] V. Kuliamin , A. Petrenko, N. Pakoulin. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. Proceedings of WMSCI'2005, v. VII. Model Based Development and Testing, pp. 65-70, Orlando, USA, 2005.

[33] А. И. Гриневич, В. В. Кулямин, Д. А. Марковцев, А. К. Петренко, В. В. Рубанов, А. В. Хорошилов. Использование формальных методов для обеспечения соблюдения программных стандартов. Труды ИСП РАН, Обеспечение надежности и совместимости Linux-систем, 10:51-68, 2006.

[34] В. В. Кулямин, А. К. Петренко, В. В. Рубанов, А. В. Хорошилов. Формализация интерфейсных стандартов и автоматическое построение тестов соответствия. Труды Второй российской конференции по программной инженерии, SEC(R) 2006, Москва, Россия, 2006.

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

[36] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Асинхронные автоматы: классификация и тестирование. Труды ИСП РАН, 4:7-84, 2003.

[37] I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proceedings of MBT'2006, Vienna, Austria, 2006.

[38] S. Albers, M. R. Henzinger. Exploring Unknown Environments. SIAM Journal on Computing, 29(40:1164-1188, 2000.

[39] R. Fleisher, G. Trippen. Exploring an Unknown Graph Efficiently. Proccedings of ESA'2005, LNCS 3669:11-22, 2005.

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

[41] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов: недетерминированный случай. Программирование, 30(1):2-17, 2004.

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

[43] T. Parr, R. Quong. ANTLR: A Predicated-LL(k) Parser Generator. Journal of Software Practice and Experience, 25(7):789-810, July 1995.

[44] Web-сайт поддержки инструментария ANTLR http://www.antlr.org.

[45] Web-сайт поддержки инструментария JavaCC https://javacc.dev.java.net/.

[46] А. В. Демаков. Язык описания абстрактного синтаксиса TreeDL и его использование. Препринт 17 ИСП РАН, 2006.

[47] А. В. Демаков. Объектно-ориентированное описание графового представления программ и моделей. Диссертация на соискание учёной степени кандидата физико-математических наук. Москва, 2006.

[48] И. Агамирзян, С. Грошев, А. Хорошилов, Г. Ключников, А. Косачев, В. Омельченко, Н. Пакулин, А. Петренко, В. Шнитман. Применение

формальных методов для тестирования MSR IPv6. Сборник тезисов международной конференции «Интернет нового поколения», Ярославль, Россия, 2002.

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

[50] А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. Формализация требований Интернет–стандартов для тестирования реализаций коммуникационных протоколов. Труды конференции «Научный сервис в сети Интернет», М.: МГУ, стр. 318-321, 2003.

[51] Г. В. Ключников, А. С. Косачев, Н. В. Пакулин, А. К. Петренко, В. З. Шнитман. Применение формальных методов для тестирования Mobile IPv6. Сборник тезисов 2-й международной конференции «Интернет нового поколения», стр. 20-25, Ярославль, Россия, 2003.

[52] Н. В. Пакулин. Формальная спецификация IPsec. Сборник тезисов 3-й международной конференции «Интернет нового поколения», стр. 14-23, Москва, 2004.

[53] Г. В. Ключников, Н. В. Пакулин, В. З. Шнитман. Автоматизированное тестирование сетевых сервисов Интернет-протокола. Труды конференции «Научный сервис в сети Интернет», стр. 168-170, М.: МГУ, 2005.

[54] Web-сайт проекта JUnit http://www.junit.org/index.htm.

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

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

[57] 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.

[58] http://www.mmsindia.com/JMSAssert.html.

[59] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, B. Jacobs. JML: notations and tools supporting detailed design in Java. Proceedings of OOPSLA'2000 Companion, pp. 105-106, Minneapolis, Minnesota, 2000.

[60] Web-сайт проекта JML http://www.cs.iastate.edu/~leavens/JML/.

[61] M. Karaorman, U. Holzle, 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.

[62] D. Bartetzko, C. Fisher, M. Moller, H. Wehrheim. Jass — Java with assertions. In K. Havelund and G. Rosu, editors, Proceeding of the First Workshop on Runtime Verification RV'2001, July 2001.

[63] A. Duncan, U. Holzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, 1998.

[64] P. Muller, J. Meyer, 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.

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

[66] http://www-128.ibm.com/developerworks/rational/products/testrealtime.

[67] http://www.parasoft.com.

[68] C. Boyapati, S. Khurshid, D. Marinov. Korat: Automated Testing Based on Java Predicates. Proceedings of ISSTA'2002, Rome, Italy, July 2002.

[69] http://www.time-rover.com.

[70] http://www.t-vec.com.

[71] C. Heitmeyer. Software Cost Reduction. Encyclopedia of Software Engineering, Two Volumes, John J. Marciniak, editor, ISBN: 0-471-02895-9, January 2002.

[72] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Microsoft Research Technical Report, May 2005.

[73] Web-сайт проекта SpecExplorer http://research.microsoft.com/specexplorer/.

[74] W. Grieskamp, Y. Gurevich, W. Schulte, M. Veanes. Testing with Abstract State Machines. Proceedings of Eurocast'2001, pp. 257-261, Canary Islands, Spain, February 2001.

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

[76] Web-сайт проекта AGEDIS http://www.agedis.de/.

[77] F. Huber, B. Schätz, A. Schmidt, K. Spies. AutoFocus — A Tool for Distributed Systems Specification. Proceedings of FTRTFT'96, LNCS 1135:467-470, Springer-Verlag, 1996.

[78] http://autofocus.informatik.tu-muenchen.de/.

[79] J.-C. Fernandez, C. Jard, T. Jeron, 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, eds, Elsevier Science publisher, 1996.

[80] http://www-verimag.imag.fr/~async/TGV/index.shtml.en.

[81] http://www.conformiq.com/qtronic.php.

[82] http://www.conformiq.com/ctg.php.

[83] http://www.leirios.com/.

[84] http://www.reactive-systems.com/.

[85] http://www.telelogic.com/products/tau/index.cfm.

[86] http://www-list.cea.fr/labos/gb/LSL/test/gatel/index.html.

[87] http://www-verimag.imag.fr/~synchron/index.php?page=lurette/lurette.

[88] http://www.t-vec.com/solutions/simulink.php.

[89] J. Tretmans, A. Belinfante. Automatic testing with formal methods. Proceedings of EuroSTAR'99, Barcelona, Spain, November 1999.

[90] http://fmt.cs.utwente.nl/tools/torx/introduction.html.

[91] G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected state machine coverage for software testing. Proceedings of ISSTA'2002, Rome, Italy. July 2002.