Организация сложных тестовых наборов

В. В. Кулямин
kuliamin@ispras.ru

Аннотация. Статья посвящена различным способам организации и структуризации сложных тестовых наборов. Анализируются основные проблемы эксплуатации и развития тестов, для решения которых необходимо введение дополнительной структуры. Рассматриваются три базовых техники структуризации тестов — выделение модулей, определение квалификаторов и введение конфигурационных параметров.

1. Введение

Современное программное обеспечение (ПО) очень сложно, и сложность его постоянно возрастает. Это вызвано естественным стремлением человечества к развитию, к решению новых, все более сложных задач и, соответственно, постоянно возрастающими требованиями к функциональности и удобству используемых на практике программных систем. Поэтому рост сложности практически значимого ПО стоит рассматривать не как проблему, нечто нежелательное, а как объективный фактор, один из вызовов, стоящих перед способностями человека к познанию законов окружающей действительности и умению использовать их.

Сложные программные системы требуют соответствующих их сложности и масштабируемых методов обеспечения надежности и качества. Единственным практически работающим подходом к созданию надежных программных систем является метод корректирующих уточнений, при котором система строится постепенно из небольших частей, при этом каждый компонент системы, каждая их группа и система в целом проходят многочисленные проверки, позволяющие выявлять расхождения с требованиями и шаг за шагом устранять их. Для проверки выполнения требований и обнаружения ошибок чаще всего используется тестирование.

Однако, с ростом сложности систем, сложность разработки и поддержки в рабочем состоянии тестовых наборов для них растет гораздо быстрее. Современные методы разработки ПО позволяют с разумными трудозатратами создавать системы объемом до десятков миллионов строк кода [1,2], хотя еще двадцать лет назад эта планка была на уровне десятков тысяч строк. В то же время используемые на практике техники создания тестов за это время

увеличили свою масштабируемость лишь примерно на порядок, хотя тестовый набор для сложной программной системы сам по себе также является сложной системой. Возникающее расхождение между масштабами систем, которые мы можем создать, и систем, которые мы в состоянии аккуратно проверить, грозит увеличением количества сбоев в ПО и ущерба от них. О масштабах возникающих проблем говорят хотя бы следующие цифры, относящиеся к различным программным продуктам компании Microsoft. Тестовый набор для текстового редактора Microsoft Word XP содержал около 35 тысяч тестов [3], а для Windows XP — более 2-х миллионов. Если в разработке Windows NT 4.0 участвовало около 800 служащих компании, а в ее тестировании — около 700, то у Windows 2000 было около 1400 разработчиков и 1700 тестировщиков, т.е. соотношение программистов и тестировщиков поменялось на обратное [4].

Еще более важным фактором увеличения сложности тестовых наборов является необходимость их развития, следующего за развитием тестируемой системы. Любое практически значимое ПО проходит достаточно долгую эволюцию, постепенно расширяя и изменяя свои функции и пользовательский интерфейс. Соответственно, для снижения затрат на сопровождение и обеспечение большей предсказуемости этого развития аналогичный путь должен проходить и тестовый набор. Но в него вносить изменения сложнее, чем в проверяемую систему, — помимо связей между отдельными его частями, необходимо учитывать и связи с подсистемами и модулями тестируемого ПО, их отдельными функциями и проверяемыми свойствами.

Наиболее значимый вклад в масштабируемость технологий создания и эксплуатации тестов вносит используемая организация тестовых наборов. Количество тестов в тестовых наборах для современного ПО довольно велико и продолжает увеличиваться. Кроме того, часто тесты связаны друг с другом разнообразными зависимостями, которые не всегда выделены явно, но должны учитываться при сопровождении тестов и внесении в них изменений. Проблемам организации сложных тестовых наборов и подходам к их решению и посвящена данная статья.

2. Проблемы организации тестовых наборов

При тестировании всегда используется конечный набор тестов, даже в определении тестирования в SWEBOK [5] сказано, что оно проводится в конечном наборе ситуаций. Однако не менее важен другой аспект тестирования, также зафиксированный в этом определении, — необходимость сделать выводы о качестве проверяемой системы и потребность в том, чтобы такие выводы были достоверными. Чтобы обеспечить достоверность выводов по результатам тестирования, оно должно проводиться так, чтобы все существенные аспекты поведения тестируемой системы и все факторы, способные повлиять на его корректность, были хоть как-то затронуты. Для сложного программного обеспечения таких аспектов и факторов очень много,

что и приводит и к большому количеству необходимых тестов, и к сложности их самих.

Чаще всего тестовые наборы организуются в виде комплектов *тестовых вариантов*. Один тестовый вариант представляет собой последовательность действий, состоящую из следующих частей.

- Сначала выполняются некоторые действия, нацеленные на создание определенной ситуации, приведение тестируемой системы в определенное состояние. Это *преамбула* тестового варианта.
- Затем выполняется основной набор действий, правильность которых в заданной ситуации нужно проверить. Часто этот набор содержит ровно одно действие. Обычно ситуация и действия, которые в ней нужно выполнить, задаются *целью тестирования (test purpose)*, для достижения которой и создается данный тестовый вариант.
  Результаты этих действий проверяются на предмет их соответствия требованиям к тестируемой системе. Правильность остальных действий в тестовом варианте тоже часто проверяется, но проверка корректности основного набора действий является главной целью его создания.
- В конце выполняются некоторые операции, нацеленные на освобождение ресурсов, захваченных предшествовавшими действиями и, возможно, возвращение тестируемой системы в некоторое исходное состояние.

Представление тестовой системы как набора тестовых вариантов сложилось достаточно давно, еще 30-40 лет назад. Оно довольно удобно для разработки тестов человеком — можно поставить определенную задачу (в данном случае представленную целью тестирования), а затем оформить ее решение в виде отдельной процедуры, которую можно использовать независимо от разработчика.

Кроме того, такая организация тестов имеет следующие достоинства.

- Каждый тестовый вариант сам по себе достаточно компактен и легко отделяется от остального набора тестов. Поэтому, если необходимо построить тестовый набор, нацеленный на проверку только определенных функций или определенной части интерфейса тестируемой системы, соответствующие тестовые варианты можно выделить и использовать отдельно от остальных.
  По той же причине достаточно просто выбросить часть тестовых вариантов из набора, если потребуется уменьшить его размер или ускорить выполнение тестов.
- Тестовые варианты облегчают разработчикам анализ возникающих ошибок. Хотя основной целью тестирования является только обнаружение ошибок, а не их локализация, результаты тестирования только в виде вердиктов «ошибок нет» или «ошибки есть» никому не

нужны на практике. В случае обнаружения ошибки разработчики системы надеются получить достаточно информации, чтобы легко восстановить и проанализировать возникшую ситуацию. Для этого тестовый вариант хорошо подходит — он представляет собой единый сценарий событий, достаточно компактен и формирует ровно одну основную ситуацию, так что для выполняющего отладку разработчика область анализа ограничена.

Но у организации тестовой системы как набора тестовых вариантов есть и недостатки, связанные с многократно возросшей сложностью тестируемых систем и необходимостью постоянного обновления и развития наборов тестов.

- В современных тестовых наборах тестовых вариантов часто очень много, иногда десятки и сотни тысяч. Таким количеством тестов уже нельзя эффективно управлять, если не вводить дополнительных уровней иерархии или каких-то классификаторов.
- Очень часто в больших наборах тестов одни и те же их элементы используются многократно. Например, проверка реакции системы на одни и те же действия обычно одинакова, генерация тестовых данных для разных операций может выполняться одними и теми же процедурами. Все это приводит к потребности обеспечения многократного использования одних и тех же решений, которые стоит оформлять в виде отдельных компонентов.
  Таким образом выделяются *тестовые оракулы* — компоненты, чья задача состоит в проверке корректности поведения тестируемой системы в ответ на воздействия определенного типа, возникающие в различных тестах. *Генераторы тестовых данных* тоже часто становятся отдельными компонентами, которые можно использовать в разных тестах.
  Возможные виды компонентов тестовых систем обсуждаются в профиле универсального языка моделирования UML для разработки тестов (UML 2.0 Testing Profile [6-10]), в предложениях общей архитектуры инструментов тестирования, сформулированных в проекте AGEDIS [11-13], а также в работах, посвященных технологии UniTESK [14-16]. Техники выделения модулей в модульных тестах (unit tests), как общего характера, так и связанные с подходом к разработке на основе тестирования (test driven development, TDD), обсуждаются в книгах [17,18].
- Развитие тестируемой системы или потребность в независимом развитии тестового набора (например, для повышения полноты тестирования, добавления проверки ранее игнорируемых свойств и т.п.) вынуждают вносить изменения в тесты. С точки зрения удобства внесения изменений неструктурированный набор тестовых вариантов представляет собой одно из самых худших решений. Без аккуратного анализа всех входящих

в него тестов невозможно понять, какие требования к тестируемой системе и какие ее модули проверяются, а какие нет, какие для этого используются техники и пр. Крайне тяжело вносить изменения, при которых иногда требуется согласованно модифицировать десятки и сотни отдельных тестовых вариантов в связи с изменением лишь одного-двух требований к проверяемой системе.

- Простые, неструктурированные наборы тестовых вариантов не всегда удобны при автоматической генерации тестов из каких-либо моделей. При этом могут возникать произвольно большие тестовые наборы, поскольку количество тестов уже не связано с трудоемкостью их разработки и не является показателем качества тестирования.
  При автоматической генерации тестов также нужно обеспечивать уникальность каждого полученного теста, иначе тестовый набор будет содержать неизвестное количество тестов-дубликатов, не приносящих никакой пользы, но занимающих место и увеличивающих время выполнения набора. Для этого нужно либо сравнивать получаемые в итоге тестовые варианты, что часто не удобно и дает невразумительные результаты, либо организовывать дополнительные структуры данных в памяти, которые позволяют генерировать только уникальные тесты.

В силу указанных причин для сложных тестовых наборов необходимы дополнительные техники организации и введение более богатой структуры, чем выделение тестовых вариантов. Систематизировать задачи, решению которых такая структуризация должна способствовать можно следующим образом.

- Задачи, связанные с удобством выполнения тестов.
  - Возможность выбора лишь части тестов для выполнения при необходимости проверить лишь часть свойств или часть интерфейса тестируемой системы, сократить время их работы, занимаемые ресурсы или по другим мотивам.
  - Конфигурируемость.
    Возможность изменить состав тестов или выполняемые ими проверки за счет небольшой модификации параметров выполнения.
  - Предварительный анализ конфигурации системы.
    Возможность настроить выполнение теста на конфигурацию тестируемой системы, например, при помощи предварительного запуска дополнительных настроечных тестов, собирающих информацию о текущей конфигурации.
- Задачи, связанные с удобством анализа результатов тестирования.
  - Предоставление достаточно полной информации о найденных ошибках, включающей, как минимум, следующее.

    - Тип ошибки по некоторой классификации, например, некорректный результат операции, некорректное итоговое состояние системы, запись неверных данных, не возвращается управление, разрушение процесса, разрушение системы в целом. Другая возможная классификация: ошибка при обычном сценарии использования, ошибка на некорректных тестовых данных, ошибка в очень специфической ситуации.
    - Какая проверка зафиксировала ошибку, что именно было сочтено некорректным, какое требование при этом проверялось.
    - Каков наиболее короткий выделяемый сценарий действий, который позволяет повторить эту ошибку. Иногда достаточно указания только одной неправильно сработавшей операции, но в сложных случаях необходимо повторить некоторую последовательность действий, в совокупности приведших к некорректной работе системы.
  - Предоставление информации о полноте тестирования.
    Эта информация должна включать в себя сведения о достигнутом тестовом покрытии по набору критериев, по которым оно измерялось, и которые хотел бы видеть пользователь. Кроме того, всегда должна быть информация о затронутых элементах тестируемой системы (вызываемые функции и методы, их классы, компоненты, подсистемы) и о проверяемых в ходе тестирования требованиях.
- Задачи, связанные с удобством модификации тестового набора.
  - Обеспечение модульности тестов и возможности многократного использования выделенных модулей.
  - Привязка тестов к требованиям, на проверку которых они нацелены.
    Такая привязка позволяет оценивать полноту текущего тестового набора, а также аккуратно выделять подлежащие модификации тесты при изменении требований.
  - Привязка тестов к элементам тестируемой системы, которые они проверяют.
    С ее помощью можно быстро определять тесты, которые необходимо поправить и выполнить при внесении небольших изменений в структуру или интерфейс тестируемой системы, но не в требования к ней.

- Классификация тестов.
  Классификация может проводиться по разным критериям.
  - Обычно удобно отделять автоматически выполняемые тесты от тех, выполнение которых требует участия человека.
  - Сложность и виды проводимого тестирования.
    Этот аспект связан с различиями в сложности проводимых тестами проверок, в используемых критериях полноты, в разном влиянии результатов различных тестов на общую оценку качества системы.
    Например, тест работоспособности проверяет обычно, что тестируемая функция хоть как-то работает при вызове с корректными входными данными. Найденная таким тестом ошибка обозначает, что эту функцию, скорее всего, вообще нельзя использовать, и практически всегда является критической.
    При тестировании разных аспектов функциональности одна функция может вызываться с разными наборами аргументов и в различных ситуациях. Ошибка, обнаруженная одним из таких тестов, означает, что часть кода функции работает неправильно, и, в зависимости от соответствующей тестовой ситуации, эта ошибка может быть признана не существенной, т.е. подлежащей исправлению только в одной из следующих версий системы.
    Сложный нагрузочный тест может обнаруживать ошибки, которые проявляются очень редко и лишь при специфических сценариях использования. Такие ошибки могут вообще не влиять на воспринимаемое пользователями качество системы.

3. Техники организации тестовых наборов

Основные техники, используемые при структуризации сложных тестовых наборов, связаны с использованием следующих механизмов.

- *Квалификаторы*. Ряд техник используют метки различных типов, расставляемые в коде тестов или их описаниях, чтобы с помощью этих меток характеризовать различные виды тестов и их связи с другими артефактами разработки.
- *Конфигурационные параметры*. Большинство техник конфигурации и определения зависимостей основано на введении набора параметров, которые могут принимать различные значения и за счет этого определять ход выполнения тестов, подключение или отключение отдельных элементов проверяемой системы или тестового набора и другие характеристики.

- *Модульность*. Такие техники используют выделение в тестовой системе модулей, имеющих определенные области ответственности. Модули крайне необходимы для организации повторного использования и повышения удобства сопровождения сложных тестовых наборов.

3.1. Квалификаторы

Наиболее простой способ внесения дополнительной структуры в тестовый набор основан на определении нескольких видов меток или квалификаторов и их использовании в коде тестов или в качестве декларативных описателей тестов.

Декларативными квалификаторами удобно пользоваться для классификации тестов по нескольким аспектам, например, по проверяемым требованиям и затрагиваемым элементам тестируемой системы. Выделить группы тестов так, чтобы они объединяли тесты по обоим этим признакам сразу, чаще всего невозможно. Поэтому лучше преобразовать тестовый набор в своего рода базу данных о тестах, где дополнительные квалификаторы буду играть роль атрибутов теста и позволят выделять подмножества тестов по практически произвольным признакам.

Помимо связей с требованиями и элементами тестируемой системы в виде квалификаторов можно представлять классификацию тестов по целям, видам проводимого тестирования, по сложности или по другим признакам.

Квалификаторы-метки обычно более удобны для выделения статической информации о тестах, однако иногда такие метки в коде тестов могут использоваться и для определения ряда характеристик теста в динамике, во время его выполнения. Например, тест может быть нацелен на достижение определенной ситуации или цели тестирования, что можно указать декларативным квалификатором, но в ходе тестирования такая ситуация не всегда возникает из-за недетерминизма поведения тестируемой системы или ошибок в ней. Чтобы уметь определять, возникала или нет эта ситуация в ходе тестирования, достаточно обеспечить сброс в трассу из кода теста определенной метки в тот момент, когда это становится ясно.

3.2. Конфигурационные параметры

Другой вид структуризации тестовых наборов — определение и использование некоторых *конфигурационных параметров*, управляющих ходом тестирования, набором подключаемых компонентов и выполняемыми проверками.

Во многих случаях достаточно *статически устанавливаемых* конфигурационных параметров, значения которых заносятся в конфигурационные файлы или передаются запускающей тестовой набор программе в качестве аргументов.

Такие параметры могут определять глубину проводимого тестирования, набор выполняемых тестов (например, используя некоторый помечающий их квалификатор), объем проводимых проверок — некоторые проверки в тестах можно помечать как опциональные и выполнять, только если выставлено соответствующее значение некоторого параметра.

Большей гибкости управления тестовым набором можно добиться, используя *динамически устанавливаемые* конфигурационные параметры, хотя они несколько повышают сложность анализа и сопровождения тестового набора. Приведем два примера их использования.

- Такой параметр может своим значением определять присутствие или отсутствие в системе определенной функциональности, объявленной в стандарте опциональной. Если включение этой функциональности связано с использованием определенных конфигурационных параметров самой тестируемой системы или может быть выявлено при помощи простой проверки, специальный модуль теста может в начале его работы определить нужное значение соответствующего параметра теста и выставить его. При дальнейшем выполнении тестов значение такого параметра просто используется, как если бы он был статическим.
  При этом возникает дополнительный модуль тестовой системы, *детектор конфигурации*, работающий до запуска основных тестов и выявляющий текущую конфигурацию тестируемой системы.

- Другое использование динамических параметров связано с повышением удобства анализа результатов тестирования сложной системы.
  В таких системах часто бывают функции, тщательное тестирование которых требует выполнения достаточно сложных сценариев, в которых тяжело разобраться, если возникает какая-либо ошибка. Примерами такой функциональности являются межпроцессное взаимодействие в операционных системах и зависящая от многих факторов обработка заголовков телекоммуникационных протоколов. Ошибка, связанная с полной неработоспособностью такой функции, может сделать результаты выполнения сложных тестов для нее совершенно непонятными — обычно бывает ясно, что ошибка есть, но более точная ее локализация требует значительных усилий.
  Во избежание подобных затрат можно предварять выполнение сложных тестов различных аспектов такой функциональности простыми тестами на работоспособность функции в целом. Сложные тесты должны выполняться только в том случае, если предшествовавшие им простые не нашли ошибок.
  Реализовать описанную процедуру можно с помощью динамически устанавливаемых по результатам простых тестов конфигурационных параметров.
  Аналогично, нагрузочные тесты имеет смысл выполнять только в том

случае, если проверяемые ими элементы тестируемой системы выполняют свои основные функции правильно.

3.3. Модульность

Самой мощной техникой структуризации тестового набора является выделение в нем модулей, ответственных за решение разнообразных задач, возникающих во время работы теста.

Простейший способ выделения таких компонентов — определение групп тестовых вариантов, ответственных за проверку определенных элементов тестируемой системы или же некоторых аспектов требований к ней. Эти группы могут образовать иерархию, в которой группы верхнего уровня далее разбиваются на подгруппы, и т. п. При такой организации тестов выделение основных групп возможно еще на ранней стадии создания тестового набора, что позволяет эффективно распределять усилия по его разработке в большой команде.

Однако более полезным с точки зрения обеспечения многократного использования одного и того же кода является выделение модулей внутри самих тестовых вариантов.

Наиболее четко могут быть выделены следующие виды компонентов.

- При тестировании достаточно широко используются компоненты, решающие задачи системного характера, не специфические именно для тестов. Они применяются для организации взаимодействия между другими компонентами теста и обеспечивают гибкое и точное управление ходом тестирования. К таким компонентам можно отнести *планировщики хода теста* [9] или *диспетчеры*, управляющие синхронизацией действий распределенных тестовых агентов, *таймеры*, используемые для отсчета времени, специализированные компоненты для мониторинга событий определенных видов, а также компоненты, отвечающие за запись информации в трассу теста.

- *Тестовые адаптеры (test adapters)*.
  Компоненты-адаптеры необходимы для привязки теста к тестируемым интерфейсам, если эти интерфейсы могут меняться без изменения их функциональности или если один и тот же тест предназначен для тестирования различных систем, реализующих одни и те же функции. Адаптер реализует абстрактный интерфейс, с которым работает тест, на основе одного из реальных интерфейсов, позволяя остальным компонентам теста не зависеть от конкретного синтаксиса реальных интерфейсов.
  Тестовые адаптеры — один из наиболее широко используемых видов компонентов теста. Адаптеры используются и в UniTESK под именем *медиаторов* [14], и при разработке тестов на TTCN для их привязки к конкретным тестируемым системам. В UML Testing Profile адаптеры не

упоминаются, поскольку он определяет структуру абстрактного тестового набора, не зависящего от синтаксиса обращений к тестируемой системе.

- *Тестовые заглушки (test stubs)*.
  Заглушки используются при тестировании отдельных компонентов, модулей или групп модулей, для работы которых необходимы другие компоненты, если эти другие компоненты недоступны (еще не разработаны) или просто не используются, чтобы не усложнять тестирование и анализ его результатов. Заглушка реализует интерфейс одного из отсутствующих компонентов, заменяя его в ходе теста. В качестве результатов заглушки обычно возвращают произвольные значения — постоянные или сгенерированные случайным образом. Однако иногда используются "умные заглушки" (smart stubs), реализующие какую-то часть функций заменяемого модуля или специфические сценарии его работы.
  Поскольку заглушки часто возникают при модульном тестировании, в книге [18] различным видам заглушек посвящена отдельная глава. В сообществе, связанном с разработкой на основе тестирования (TDD), заглушки предпочитают называть «фиктивными объектами» (mock objects, mocks) или «тестовыми дубликатами» (test doubles). Более точно, в терминологии [18] тестовые дубликаты могут относиться к различным видам.
  - *Фальшивый объект (fake object)*. Это простой тестовый дубликат, способный принимать обращения из тестируемой системы и выдавать какие-то результаты в ответ на них, все равно какие, лишь бы происходило корректное взаимодействие.
  - Собственно, *заглушка (test stub)*. В рамках данного сообщества считается, что такие дубликаты должны уметь выдавать значения возвращаемых тестовой системе результатов в соответствии с целями теста, в котором они используются. Эти значения используются как неявные тестовые данные, с помощью которых тест приводит проверяемую систему в нужное состояние или оказывает на нее нужный набор воздействий.
  - *Тестовый шпион (test spy)*. Это разновидность дубликата, которая умеет протоколировать сделанные к ней обращения из тестируемой системы, чтобы проверить их правильность в конце теста.
  - *Фиктивный объект (mock object)*. Отличается от шпиона только тем, что выполняет проверки корректности производимых к нему обращений прямо в ходе работы теста, а не в его конце.
- *Генераторы тестовых данных*.
  Роль их, как видно из названия, состоит в построении некоторого набора

данных, обычно одного типа. Выгода от их использования появляется при необходимости создавать разнообразные объекты одного типа данных в разных тестах.
Генераторы тестовых данных сложной структуры обычно делаются составными. Например, генератор значений комплексных чисел можно построить из двух генераторов действительных чисел — для вещественной и для мнимой частей. Генератор сложных документов удобно строить в виде системы из взаимодействующих генераторов отдельных частей таких документов — заголовков, отдельных полей и разделов, отдельных фраз и слов.
В технологи UniTESK такого рода компоненты названы *итераторами* [14]. В профиле UML для разработки тестов [9] они названы *селекторами данных* (data selector). Селекторы могут использовать *контейнеры данных* (data pool), хранящие определенный набор данных, выбор из которых может производиться селектором по дополнительным правилам.

- *Тестовые оракулы (test oracles)* или просто *оракулы*.
  Тестовый оракул [14,19,20] — компонент, ответственный за вынесение вердикта о соответствии или несоответствии поведения системы требованиям. Работа оракула часто в большой степени зависит от конкретной тестовой ситуации, от сценария данного теста. Однако оракул для сложного сценария часто получается некоторой композицией проверок корректности его отдельных действий. Проверки корректности работы отдельных операций гораздо проще использовать многократно в разных тестах и удобнее применять для отслеживания более точной информации об ошибке, например, точного нарушенного требования или конкретной операции, выполненной с ошибкой.
  Поэтому удобно определить *оракул типа событий* или *оракул операции*, которые привязываются к событиям соответствующего типа или к вызовам определенной операции и выносят вердикт о том, насколько поведение системы при возникновении событий такого типа или при различных обращениях к этой операции соответствует требованиям.
  Другая возможная разновидность таких оракулов — *ограничения целостности данных*. Они относятся к некоторому типу данных и должны проверяться каждый раз, когда данные такого типа передаются в тестируемую систему или принимаются от нее. Поскольку данные сами по себе не активны, а лишь используются в вызываемых операциях и возникающих событиях, обычно ограничения целостности данных используются всеми оракулами операций и событий, в которых затрагивается соответствующий тип данных.
  Выделение таких модульных оракулов оправдано двумя факторами.
  - Требования формулируются, в основном, именно в отношении различных типов данных, типов событий или операций. Поэтому

при переносе их в тесты достаточно удобно и с точки зрения возможных будущих модификаций, и для обеспечения прослеживаемости требований объединять требования к одному типу событий, типу данных или операции в один компонент.

- Одна и та же операция, данные или события одного и того же типа могут в сложном тестовом наборе использоваться во многих тестах. Это одно из основных отличий «сложных» тестовых наборов от «простых» — во втором случае тестов, затрагивающих одну и ту же операцию, не так много, и проблема многократного использования кода не стоит так остро.

- Кроме описанных выше оракулов, могут использоваться более сложные, *композиционные оракулы*. Они возникают в тех случаях, когда для вынесения вердикта о корректности поведения системы в некоторой ситуации требуется нетривиальный анализ многих разных его аспектов, который неудобно проводить в рамках одного компонента.
  Например, при оценке корректности данных достаточно сложной структуры, таких как XML-документы или программы на языках программирования, или даже документы, в которых может быть смешано несколько языков, проводить такую проверку в рамках одного компонента крайне неудобно — он становится крайне сложным, неудобным для модификаций и сопровождения. В таком случае ограничения целостности данных разбиваются на группы ограничений, относящихся только к определенным конструкциям, и правила корректного связывания конструкций, и для каждой такой группы можно иметь отдельный компонент, проверяющий ее ограничения. Общий оракул для документа в целом получается как некоторая композиция этих компонентов. Композиционные оракулы применяются и при тестировании распределенных систем. При этом обычно используют набор тестовых агентов, каждый из которых отслеживает поведение только одного компонента системы или небольшой их группы. Он сам может выносить вердикт о корректности событий, касающихся отслеживаемых компонентов, в том числе, используя оракулы отдельных событий. Однако к поведению системы в целом могут при этом предъявляться требования, которые ни один из таких агентов не в состоянии проверить самостоятельно. Тогда их проверка организуется в отдельном компоненте, который получает необходимую ему информацию от всех тестовых агентов.
  Примером такого составного оракула является *сериализатор* в технологии UniTESK [15,21], который проверяет правильность набора событий в соответствии с *семантикой чередования*, убеждаясь, что наблюдаемое поведение системы соответствует возникновению этих событий в некотором порядке (все равно в каком). Для этого он вызывает оракулы отдельных событий в разных последовательностях, пока не

будет найдена корректная или не будет показано, что подходящей последовательности нет, что означает ошибку.
Другим примером является *арбитр*, один из компонентов, определяемых в профиле UML для разработки тестов [9]. Однако он играет только роль посредника, позволяя тестовым агентам, выносящим свои вердикты на основе доступной им информации, обмениваться данными об этих вердиктах друг с другом.

Других компонентов тестов, которые выделялись бы в рамках различных методов построения тестов и различными авторами, пока не удается найти. Все остальные виды компонентов специфичны для определенных методов тестирования и чаще всего не имеют аналогов в других подходах.

При определении модульной структуры тестов стоит учитывать, что вместе с появлением возможности многократно использовать одни и те же компоненты, повышается и их сложность для людей, не знающих об используемой архитектуре. Поэтому архитектура таких тестов, с указанием всех видов используемых компонентов и задач, решаемых ими, должна быть описана в документации на тестовый набор. Крайне желательно постепенно стандартизовать достаточно широкий набор видов модулей теста, чтобы сделать возможным их использование в различных инструментах.

Другое препятствие к широкому использованию модульности тестов связано с усложнением анализа ошибок. Тест уже не представляет собой единый, изолированный сценарий работы, для понимания которого не нужно заглядывать в много различных документов или в код различных компонентов. Поток управления в рамках одного тестового варианта или более сложного теста становится иногда очень причудливым и сложным, разобраться в нем становится тем труднее, чем больше разных видов компонентов используется. Поэтому при использовании модульных тестов необходимы дополнительные усилия по упрощению анализа ошибок. Одним из вариантов решения этой проблемы может быть автоматическое создание более простых, классических тестовых вариантов, повторяющих ситуацию, в которой обнаружена ошибка.

4. Заключение

В статье рассмотрен ряд проблем, делающих необходимой более тонкую структуризацию сложных тестовых наборов, чем традиционное разбиение на тестовые варианты. Также рассказывается об основных техниках внесения дополнительной структуры в тестовый набор: выделении модулей, использовании квалификаторов и определении набора конфигурационных параметров.

Наиболее легко с широко применяемыми сейчас подходами к разработке тестов сочетается использование набора квалификаторов для классификации тестов и указания связей между тестами и требованиями или элементами

тестируемой системы. Однако большинство инструментов управления тестами поддерживает использование только предопределенного набора квалификаторов, которые не могут быть расширены. Пока только HP/Mercury TestDirector [22] имеет возможность добавления пользовательских квалификаторов, которые можно затем использовать для построения специфических отчетов — группировки или отбрасыванию результатов тестов по значениям квалификатора.

Более сложно с имеющимися подходами к управлению тестами интегрируются использование системы конфигурационных параметров, а также выделение разнообразных модулей и их многократное использование в тестах. Конфигурационные параметры можно использовать только в виде дополнительных квалификаторов, что далеко не всегда удобно, особенно для динамически устанавливаемых параметров.

Проблема выделения различных модулей в сложных тестах стоит наиболее остро. Только в последние 5 лет появилось несколько подходов [6,11,14,18] к разработке тестов, уделяющих этому аспекту достаточно внимания. В то же время, общим элементом этих подходов, да и то не всех, можно считать выделение заглушек, адаптеров, генераторов тестовых данных и некоторых общесистемных компонентов тестов. Выделение более разнообразного набора модулей, включающего оракулы отдельных операций и событий, с одной стороны, является необходимым для повышения управляемости и гибкости современных сложных тестовых наборов, но с другой стороны, усложняет ряд традиционных видов деятельности — конфигурирование тестового набора и анализ обнаруживаемых им ошибок. Для разрешения возникающих проблем необходимо разработать новые техники решения соответствующих задач в модульных тестовых наборах.

Литература

[1] V. Maraia. The Build Master: Microsoft's Software Configuration Management Best Practices. Addison-Wesley Professional, 2005.
[2] G. Robles. *Debian Counting*. http://libresoft.dat.escet.urjc.es/debian-counting/.
[3] Ю. Гуревич, Microsoft Research. Устное сообщение. 2004.
[4] M. Lucovsky. Windows, a Software Engineering Odyssey. 4-th Usenix Windows Systems Symposium, 2000. http://www.usenix.org/events/usenix-win2000/invitedtalks/lucovsky_html/.
[5] Software Engineering Body of Knowledge, 2004. http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf.
[6] I. Schieferdecker; Z. R. Dai, J. Grabowski. The UML 2.0 Testing Profile and its Relation to TTCN-3. IFIP 15-th Int. Conf. on Testing Communicating Systems — TestCom 2003, Cannes, France, May 2003.
[7] A. Cavarra. The UML Testing Profile. An Overview. 2003. http://www.agedis.de/documents/TheUMLTestingProfile.pdf.
[8] Z. R. Dai. UML 2.0 Testing Profile. In M. Broy et al.,eds. Model-Based testing of Reactive Systems. Advanced Lectures. LNCS 3472:497-521, Springer, 2005.
[9] UML Testing Profile, 2005. http://www.omg.org/docs/formal/05-07-07.pdf.

[10] http://www.fokus.fraunhofer.de/u2tp/.
[11] A. Hartman, K. Nagin. Model Driven Testing — AGEDIS Architecture Interfaces and Tools. Proceedings of the 1-st European Conference on Model Driven Software Engineering, Nuremburg, Germany, December 2003.
[12] A. Hartman. AGEDIS Final Project Report, 2004. http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF.
[13] http://www.agedis.de/.
[14] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77-88, Springer, 2002.
[15] 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, 2003.
[16] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
[17] К. Бек. Экстремальное программирование: разработка через тестирование. СПб.: Питер, 2003.
[18] G. Meszaros. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.
[19] D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Transactions on Software Engineering, 24(3):161-173, 1998.
[20] L. Baresi, M. Young. Test Oracles. Tech. Report CIS-TR-01-02. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.
[21] А. В. Хорошилов. Спецификация и тестирование систем с асинхронным интерфейсом. Препринт 12 ИСП РАН, 2006.
[22] http://www.mercury.com/us/products/quality-center/testdirector/.