ПРОГРАММИРОВАНИЕ, 2010, № 5, с. 54–75

ТЕСТИРОВАНИЕ, ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ПРОГРАММ

УДК 681.3.06

КОМПОНЕНТНАЯ АРХИТЕКТУРА СРЕДЫ ДЛЯ ТЕСТИРОВАНИЯ НА ОСНОВЕ МОДЕЛЕЙ

© 2010 г. В. В. Кулямин

Институт системного программирования РАН

109004 Москва, ул. Солженицына, 25

E-mail: kuliamin@ispras.ru

Поступила в редакцию 21.01.2010 г.

В статье представлен подход к построению архитектуры инструментария для тестирования на основе моделей, использующего современные компонентные технологии. Одна из основных идей, лежащих в его основе – применение техник неинвазивной композиции, позволяющих с минимальными затратами интегрировать множество независимо разработанных компонентов в сложную систему и переконфигурировать ее, не изменяя кода компонентов. Представленный подход является одним из первых применений компонентных технологий для проектирования тестовых систем. Также описывается прототипная реализация предложенного подхода на базе свободно доступных библиотек и пример ее использования для построения тестов.

1. ВВЕДЕНИЕ

Рост количества и разнообразия задач, возлагаемых в современном мире на вычислительные системы, приводит к постоянному их усложнению и увеличению сложности всех видов деятельности, связанных с их построением и сопровождением. На решение проблем, связанных с этим ростом сложности, нацелены активно развивающиеся в последние десятилетия *компонентные технологии* [1, 2] построения вычислительных систем.

При их применении системы выстраиваются из независимых *компонентов*, каждый из которых решает узкий набор задач и взаимодействует с окружением только через четко определенный интерфейс, т.е. является *модулем*. Помимо этого, программные компонентные технологии облегчают интеграцию компонентов, автоматически выполняя связывание и запуск компонентов (часто на лету, без прерывания работы системы), помещенных в бинарном виде в рамки некоторой технологической инфраструктуры (*компонентной среды*). В результате компоненты могут создаваться и поддерживаться независимыми разработчиками, что снижает расходы на разработку и сопровождение сложных систем.

Компонентные технологии позволили значительно повысить сложность создаваемых систем, однако их развитие не сопровождается соответствующим прогрессом в технологиях контроля качества программного обеспечения (ПО). В связи с этим проблемы обеспечения качества компонентных систем, включающие проверку корректности их работы, лишь усугубляются с возрастанием сложности и ответственности решаемых ими задач.

Источником таких проблем служит чрезвычайно высокая трудоемкость проверки корректности систем при нелинейно зависящем от числа компонентов количестве возможных сценариев их взаимодействия и неявных зависимостей между ними. В свою очередь, эти трудности имеют следующие причины.

- **Сложившаяся практика неполного и неточного описания интерфейсов компонентов** приводит к возникновению множества неявных зависимостей между ними. Согласно Парнасу [3], одному из основоположников модульного подхода, полный интерфейс модуля должен задавать не только его синтаксическую структуру (имена операций, типы их параметров и результатов),

но и семантику (правила использования операций и ожидаемое поведение модуля). На практике же обычно фиксируют только синтаксис, семантика описывается, чаще всего не полностью и не однозначно, только для системных интерфейсов, поскольку с некорректным их использованием часто связаны высокие риски. Однако "менее рискованных" интерфейсов, смысл которых обычно не описывается совсем, гораздо больше, и суммарные расходы, связанные с ошибками в их работе, весьма значительны [4].

- **Трудности интеграции средств верификации в процессы разработки ПО**. Такие средства по большей части являются "монолитными" инструментами или интегрированными решениями с большим количеством функций. Набор деятельностей по контролю качества ПО, связи между ними, а также поддерживаемые техники верификации задаются разработчиками инструментов и не могут быть существенно изменены или расширены. Однако процессы разработки ПО в разных организациях сильно различаются, поэтому все более востребованы модульные инструменты разработки, способные легко интегрироваться с другими инструментами и включать новые модули, реализующие передовые техники разработки и анализа ПО. Инструменты разработки сложных систем тоже должны быть компонентными и требовать минимума затрат на свое включение в разные технологические цепочки. Среди средств верификации такими свойствами пока обладают лишь инструменты *модульного тестирования* (unit testing) [5] (наиболее известен JUnit [6]), модульной поддержки более сложных видов верификации или тестирования на соответствие требованиям практически нет.

- **Слабая поддержка многократного использования артефактов контроля качества** – тестов, верифицированных утверждений, моделей и пр. Это особенно важно для компонентных технологий, где верификация одного компонента чаще всего имеет приемлемую стоимость, в ее ходе создается много вспомогательных артефактов, а использовать их далее при проверке взаимодействующих компонентов или гораздо более дорогостоящем контроле качества систем, включающих проверенный компонент, уже не удается. Обеспечение многократного использования таких артефактов могло бы значительно снизить издержки верификации компонентного ПО.

Мотивацией данной работы является убежденность, что систематическое использование компонентных технологий при построении верифицирующих систем поможет справиться с проблемами, связанными с разрывом между сложностью современных систем и возможностями контроля их качества. Выход из сложившейся ситуации – создание *компонентных технологий верификации* программных и аппаратных систем, которые должны обеспечить независимую проверку качества отдельных компонентов в соответствии с требованиями к ним, а затем использовать получаемые при этом артефакты для более аккуратного контроля их интеграции и качества системы в целом, постепенно наращивая количество и сложность проверенных компонентов и подсистем. Эти технологии должны расширять существующие компонентные технологии разработки и задействовать элементы, уже знакомые разработчикам.

Гибкость технологии и возможность ее использования для решения разных задач во многом обуславливаются лежащей в ее основе архитектурой. Для компонентных технологий архитектурные решения еще более значимы, поскольку они непосредственно определяют систему правил создания, развития и добавления новых компонентов, их разновидности, шаблоны организации их взаимодействий, возможные конфигурации и т.д.

Одним из перспективных подходов к систематической проверке корректности сложных систем является тестирование на основе моделей, сочетающее строгость формальных методов и аккуратность выявления ошибок с гибкостью и практичностью обычного тестирования.

В данной работе представлены некоторые элементы компонентной технологии верификации, использующей методы тестирования на основе моделей и базирующейся на технологиях платформы Java, однако представляемая технология

пока находится в процессе создания. Основная же часть статьи посвящена ее архитектурному каркасу (framework), включающему инструментальные библиотеки, а также набор основных типов компонентов и правила их интеграции. Для тестирования на основе моделей подобная компонентная архитектура предлагается впервые, хотя ряд ее элементов заимствованы из наиболее передовых инструментов модульного тестирования.

Во втором разделе статьи рассматривается подход к тестированию на основе моделей, его требования к архитектуре поддерживающего инструментария и их имеющиеся реализации. Затем формулируются принципы компонентной разработки и представляются основные элементы предлагаемого архитектурного каркаса. Далее описан небольшой пример создания тестов с его помощью. Заключение статьи резюмирует ее содержание и обрисовывает направления дальнейшего развития изложенных идей.

2. ТЕСТИРОВАНИЕ НА ОСНОВЕ МОДЕЛЕЙ И ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ

Тестирование на основе моделей (model based testing) [7, 8] представляет собой подход к тестированию, в рамках которого тесты строятся вручную, автоматизированным образом или генерируются полностью автоматически на основе *модели поведения* тестируемой системы и *модели ситуаций*, связанных с ее работой.

- *Модель поведения* (behavior model) формализует требования к тестируемой системе, т.е. описывает, какие внешние воздействия на нее в каких ситуациях допустимы, и как она должна реагировать на эти воздействия.

  Модель поведения служит основой для *тестового оракула* [9–11] – компонента, производящего оценку поведения системы во время тестирования.

  Чаще всего при тестировании используются модели в виде автоматов разного вида: конечные или расширенные автоматы, системы переходов [7], Statecharts [12, 13], временные автоматы [14, 15] и т.д. Можно использовать другие разновидности моделей: контрактные спецификации в виде пред- и постусловий операций, алгебраические спецификации в виде правил эквивалентности различных цепочек вызовов операций, трассовые модели, описывающие возможные цепочки воздействий и реакций системы.

- *Модель ситуаций* (situation model) формализует структуру возможных тестовых ситуаций, составляющими которых являются внешние воздействия и состояния самой системы и ее окружения, определяет различные классы ситуаций и их важность с точки зрения контроля качества. Обычно такая модель задает конечный набор классов эквивалентности ситуаций, подразумевая, что при тестировании достаточно проверить работу системы хотя бы в одной ситуации из каждого класса. Часто модель ситуаций описывает конечный набор элементарных событий, при этом каждый класс ситуаций соответствует некоторому множеству таких событий. Примером такого события может являться выполнение заданной инструкции в коде тестируемой системы. В более сложных случаях определяемые классы могут пересекаться, им могут приписываться веса, показывающие, насколько важно проверить одну из ситуаций такого класса.

  Модель ситуаций используется для решения двух тесно связанных задач: определения критерия адекватности или полноты тестирования [16] и определения числовых метрик полноты тестирования. Если полнота тестирования определяется на основе покрытия классов эквивалентных ситуаций или элементарных событий, говорят о *критерии тестового покрытия, метрике тестового покрытия*, а достигнутый в ходе тестирования процент покрытых классов называют *тестовым покрытием*.

Построение тестов на основе моделей заключается в том, что создается (извлекается из проектных документов или берется откуда-то в готовом виде) модель поведения тестируемой системы, создается модель ситуаций, отражающая основные приоритеты проекта и использующая структурные элементы модели поведения, и затем строится (генерируется автоматически, создается вручную или с помощью инструментов)

комплект тестов. Эти тесты проверяют соответствие между реальным поведением тестируемой системы и ее моделью поведения. При этом тестовый набор создается так, чтобы он удовлетворял критерию полноты тестирования, заданному моделью ситуаций.

Используемые для построения тестов методы можно разделить на три типа.

- *Вероятностные методы* используют псевдослучайную генерацию данных простых типов, псевдослучайное комбинирование их в более сложные структуры и псевдослучайную же генерацию последовательностей тестовых воздействий, если необходимо проверять поведение системы в разных состояниях. При этом полноту тестирования стараются обеспечить за счет большого количества тестов, так как построение каждого отдельного теста требует небольших затрат. Модель ситуаций здесь имеет вид распределения вероятностей различных видов событий.

- *Нацеленные методы* строят тестовые данные и последовательности целенаправленно, так, чтобы те удовлетворяли определенным свойствам, чаще всего – реализовывали ситуации из классов, задаваемых моделью ситуаций. Критерий полноты тестирования при использовании таких методов удовлетворяется близким к минимальному количеством тестов, но построение каждого теста обычно требует значительных усилий человека и/или затрат вычислительных ресурсов.

- *Комбинаторные методы* строят тестовые данные и последовательности, комбинируя различные виды их элементов по определенным схемам. На создание одного теста при этом нужно существенно меньше затрат, чем при нацеленном построении, но несколько больше, чем при вероятностном. Тестовый набор получается больше, чем при нацеленном тестировании, но много меньше, чем при вероятностном, хотя в нем и достигается большее разнообразие. Кроме того, в полученном наборе обычно нет идентичных тестов, которые очень часто возникают в случайно сгенерированных. Комбинаторные методы, использующие эффективную фильтрацию не вносящих собственный вклад в достижение полноты тестов, вполне сопоставимы с нацеленными методами. Примером служат техники построения тестов на основе *автоматных моделей*, создающие тесты в виде набора путей по графу переходов автомата, минимизируя его с точки зрения покрытия всех возможных состояний и переходов.

2.1. Требования к представлениям моделей и инструментам тестирования

Инструменты тестирования на основе моделей работают с какими-то представлениями моделей поведения и моделей ситуаций. Для полноценной поддержки этого подхода и повышения удобства его использования эти представления должны обладать следующими свойствами.

- Модели поведения:
  - Средствами для описания требований в разных стилях: в виде чистых декларативных ограничений (ограничений на входные данные операций и их результаты), в виде контрактных спецификаций (пред- и постусловий операций), использующих модельное состояние, в виде исполнимых моделей, определяющих способ работы проверяемой системы на более высоком уровне абстракции. Это требование обусловлено необходимостью как можно более ясно отражать имеющиеся требования, сформулированные на естественном языке, допуская их переформулирование в другом стиле в ходе построения моделей, поскольку это позволяет выявить неточности, несогласованность и неполноту в исходных формулировках и углубить их понимание [17].
  - Средствами для явного указания связей с документами и стандартами, содержащими требования. Это позволяет прослеживать связи исходных требований с моделями и полученными автоматически тестами.

  - Возможностью автоматического преобразования моделей в тестовые оракулы.
  - Возможностью использования структуры моделей для задания моделей ситуаций.
  - Стоит поддерживать возможности использования таких моделей в рамках других техник верификации (например, проверки моделей или статического анализа).
  - Полезна также возможность преобразования модели поведения в однозначную и полную документацию на моделируемую систему. Однако способы реализации этой возможности остаются за рамками данной работы.

- Модели ситуаций:
  - Средствами для описания ситуаций, связанных с различными аспектами поведения и структуры тестируемой системы. Это относится к структуре входных данных и результатов, состояниям системы, выполняемым инструкциям кода и вызываемым функциям, шаблонам взаимодействия внутри системы, возможным ошибкам в системе и пр.
  - Средствами для явного указания связей определяемых классов ситуаций с исходными требованиями к системе.
  - Возможностью автоматического преобразования в компоненты, определяющие класс текущей ситуации, измеряющие достигнутое покрытие.
  - Возможностью автоматического извлечения из кода тестируемой системы или ее моделей, для нацеленного построения соответствующих тестов.

Можно заметить, что требования к представлению моделей делятся на три типа: достаточная для практических целей выразительность, прослеживаемость требований и возможность как можно более полной автоматизации или облегчения решения различных задач тестирования. Такие задачи представлены в следующем списке.

- Выполнение тестов. Без автоматизации выполнения тестов различных уровней невозможно организовать систематическое тестирование сложной системы.

- Проверка результатов тестирования. При автоматическом выполнении большого количества тестов человеку проверить их результаты становится уже невозможно, поэтому нужно иметь специальные компоненты тестов (*оракулы*), автоматически оценивающие, правильно ли повела себя тестируемая система в каждом случае.

- Создание тестовых данных и тестовых последовательностей. И те, и другие важны, если тестируется сложная система с поведением, зависящим от состояния. И те, и другие должны формироваться так, чтобы повышать полноту тестов согласно используемой модели ситуаций. Автоматизация построения тестов предполагает автоматическое создание нужных последовательностей обращений и данных отдельных вызовов.

- Конфигурирование тестовых наборов. Тестовый набор для сложной системы состоит из огромного количества тестов, нацеленных на проверку разных аспектов ее поведения и различных ее структурных элементов. Для его эффективного использования в процессе развития системы необходимо иметь гибкие средства конфигурирования, позволяющие в каждом конкретном случае достаточно легко определить, какие тесты выполнять, а какие нет, и соблюдающие ограничения на взаимодействие между различными тестами.

2.2. *Инструменты модульного тестирования*

Рассмотрим теперь имеющиеся инструменты тестирования с точки зрения их приближения к инструментарию на основе компонентной технологии.

В большой степени обладают нужными свойствами средства *модульного тестирования* (unit testing) [5]. Наиболее известен из таких инструментов JUnit [6], написанный на Java и предназначенный для тестирования кода на этом языке,

хотя исторически первым был SUnit [18, 19] для программ на Smalltalk.

Для этих инструментов характерны высокая гибкость, возможность подключения совершенно независимых модулей и возможность использования в рамках более сложных тестовых систем. Одним из инструментов модульного тестирования, обладающих наиболее богатой функциональностью, является TestNG [20, 21]. Его основные характеристики таковы.

- Конфигурация тестов.
  - Основные элементы тестов – тестовые классы и тестовые методы, описываемые как классы и методы языка Java, имеющие аннотацию @Test.
  - Комплект тестов имеет иерархическую структуру: в нем выделяются тестовые наборы (test suites), состоящие из тестов (tests). Далее в этом списке эти термины используются только в специфическом для TestNG смысле. Тесты и тестовые наборы определяются *конфигурацией тестов*, описываемой в некотором формате на базе XML. Тест составляется из методов, выбираемых либо на основе их имен, либо по принадлежности к *группам*, указываемым в аннотациях методов.
  - Методы инициализации (set-up) и финализации (tear-down), используемые для инициализации данных перед выполнением тестов и освобождения занятых ресурсов после, могут быть определены для всех элементов иерархии: для наборов, тестов, классов и методов.
  - Можно задавать зависимости между тестовыми методами и группами, управляющие порядком их выполнения и отменяющие выполнение тестового метода, если один из методов, от которых он зависит, выполнился с ошибками.
- Тестовые данные и объекты.
  - Тестовые методы в TestNG могут быть параметризованными. Набор значений параметров, используемый в рамках теста, указывается с помощью дополнительной аннотации или в конфигурационном файле и должен представляться либо в виде коллекции объектов, либо как последовательность результатов, возвращаемых некоторым методом при работе теста.
  - Можно создавать фабрики объектов, строящие разнообразные объекты тестовых классов. Все входящие в тест методы выполняются для каждого такого объекта.
- Проверка результатов тестирования.
  - Выполняемые проверки, как и в других инструментах модульного тестирования, задаются с помощью вызовов методов-утверждений. Каждый такой метод (их названия начинаются с assert) проверяет простое свойство своих аргументов (равенство, неравенство null, вхождение объекта в коллекцию и пр.) и при его нарушении выдает в трассу сообщение, также указываемое в виде аргумента.
  - Дополнительно TestNG поддерживает указание возможных исключений и ограничений на время работы тестового метода в его аннотации.

Инструменты модульного тестирования активно используют независимые модули для решения более специфичных задач. Например, dbUnit [22] – для организации работы с базами данных, httpUnit [23] – для обработки HTTP-запросов. Для более наглядной записи выполняемых в тестах проверок (близкой к формулировкам естественных языков) можно применять библиотеки, входящие в инструменты разработки на основе функциональности (behavior driven development), например, JBehave [24] или NSpecify [25], для организации тестовых заглушек – библиотеки Mockito [26], EasyMock [27] и т.д.

2.3. Инструменты тестирования на основе моделей

Инструменты тестирования на основе моделей с точки зрения их приближения к модульной архитектуре можно разделить на три группы.

- Традиционные “монолитные” инструменты, использующие специфические языки для оформления моделей. Добавление новых компонентов в них может на практике осуществляться только их разработчиками, а их использование в рамках более широкого инструментария возможно, только если разработчики позаботились о предоставлении подходящих интерфейсов.

  К инструментам такого типа относятся практически все исследовательские средства тестирования на основе моделей и ряд более стабильных инструментов, использовавшихся во многих разных проектах – TorX [28, 29], TGV [30, 31], BZ-TT [32] и Gotcha-TCBeans [33, 34]. Все они используют различные автоматные модели. На тех же принципах построены и коммерческие инструменты Conformic Qtronic [35] и Smartesting Test Designer (ранее Leirios) [36].

- “Монолитные” инструменты на основе расширений широко используемых языков программирования. Примеры – поддерживающие технологию UniTESK [37–39] инструменты CTESK и JavaTESK, а также Spec-Explorer [40, 41], созданный в Microsoft Research. В обоих случаях для моделирования поведения используется комбинация из автоматных моделей и контрактных спецификаций.

- Инструменты, использующие для оформления моделей обычные языки программирования и обладающие рядом характеристик модульности. В частности, они достаточно легко интегрируются с компонентами от независимых разработчиков и используются в рамках более широкого инструментария.

  Такие инструменты начали появляться не так давно, около 4–5 лет назад. Два наиболее известных примера – это ModelJUnit [8, 42] и NModel [43, 44]. Похожий инструмент mbt.tigris.org [45] использует для описания моделей графическую нотацию, поэтому гораздо менее приспособлен для использования в рамках чужих разработок. Иного рода пример – недавно созданная в Microsoft Research библиотека для описания и контроля декларативных ограничений на поведение .NET-компонентов CodeContracts [46, 47].

Возможности инструментов последнего вида, имеющих хорошую модульную структуру, стоит рассмотреть подробнее.

ModelJUnit и NModel построены как расширения простых средств модульного тестирования. Модель ситуаций в них не задается, неявно критерии полноты закладываются в используемые алгоритмы построения тестов. Модель поведения преобразуется в сопряженную *модель теста*, определяющую не само требуемое поведение, а способ его исследования и проверки.

- Модель теста задается как расширенный конечный автомат, представленный на языке программирования (Java или C#) в виде тестового класса, как и в средствах модульного тестирования. Такой класс либо помечается с помощью аннотации (атрибута в C#), либо реализует некоторый интерфейс. Элементы автоматной модели – состояния, переходы, охранные условия переходов – задаются с помощью методов и полей этого класса.
  - Состояние модели теста в NModel задается набором значений полей тестового класса (можно определять поля, не включаемые в состояние), а в ModelJUnit – результатом метода getState().
  - *Действия*, выполнение которых соответствует переходам в модели теста, представляются методами, помеченными определенной аннотацией (атрибутом). В NModel действия могут быть параметризованными, причем набор значений параметров, используемый в рамках теста, указывается с помощью дополнительной аннотации в виде коллекции объектов соответствующего типа.
  - Действия могут иметь *охранные условия*, которые должны быть выполнены, чтобы можно было выполнять соответствующее действие. Эти условия представлены как методы с именами, построенными из имени соответствующего метода-действия с некоторым постфиксом.

- NModel дополнительно имеет следующие возможности.
  - Композиция нескольких моделей, в которых одноименные действия выполняются одновременно. Модели для композиции указываются инструменту построения тестов перечислением имен соответствующих классов.
  - Проверка моделей теста (model checking). Свойства безопасности соответствуют инвариантам, представленным как методы с особым атрибутом. Свойства живучести могут проверяться за счет анализа достижимости состояний, в которых специально отмеченные характеристические методы возвращают true.

Библиотека CodeContracts предоставляет средства для описания чисто декларативных ограничений на свойства входных параметров и результатов операций. Моделирование состояния не поддерживается. Имеются следующие возможности.

- Ограничения записываются на языке C# в виде булевских выражений, передаваемых как аргументы библиотечным методам.
- Можно описывать ограничения следующих видов.
  - Предусловия операций (метод Contract.Requires).
  - Постусловия операций при нормальной работе (Contract.Ensures).
  - Постусловия операций при выдаче исключения (Contract.EnsuresOnThrow).
  - Утверждения о выполнении ограничений в какой-то точке кода внутри метода (Contract.Assert).
  - Инварианты классов – помеченные особым атрибутом методы, вызывающие Contract.Invariant.
- Помимо обычных выражений на C#, можно использовать следующие.
  - Кванторные выражения в виде обращений к Contract.Exists и Contract.ForAll.
  - Обращение к значениям выражений до выполнения проверяемого метода в постусловиях в виде вызова метода Contract.OldValue.
  - Обращение к значению результата в постусловиях с помощью Contract.Result.
- Библиотека CodeContracts дополняется двумя инструментами: для статической проверки сформулированных ограничений на основе дедуктивного анализа и для их динамической проверки при выполнении методов, ограничения на которые описаны.

2.4. Итоги обзора существующих инструментов

Из приведенного обзора видно, что модульные средства тестирования на основе моделей активно развиваются, однако пока отстают по ряду возможностей от инструментов модульного тестирования. Необходимо дополнить имеющиеся наработки следующими возможностями.

- **Явное определение модели поведения проверяемого компонента, отдельное от модели теста**. Такое разделение необходимо для поддержки точности и полноты моделирования. Оно также дает возможность строить на основе одной модели поведения разнообразные тесты, направленные на достижение различных целей, как для этого компонента, так и для содержащих его подсистем. Еще одна полезная возможность – использование таких моделей в других техниках верификации. Пример выделения модели поведения дает библиотека CodeContracts.
- **Расширенные выразительные возможности для описания моделей поведения**. В частности, нужно обеспечить возможность использования разных стилей моделирования и разнообразных техник построения тестов. Это требование является следствием сложности и разнообразия требований к современному ПО [48, 49]. Библиотека CodeContracts позволяет задавать только ограничения на входные данные и результаты, что

существенно снижает возможности ее использования для практически значимых систем, поведение которых зависит от состояния.

- **Явное определение моделей ситуаций для построения тестов**. Неявное задание модели ситуаций в имеющихся инструментах ограничивает возможности использования различных критериев полноты тестирования и затрудняет их осознанный выбор разработчиками тестов. Их явное задание позволит также комбинировать различные критерии полноты, учитывая покрытие как проверяемого кода, так и требований к нему.

- **Средства явной привязки моделей к требованиям на естественном языке**. Такие возможности, необходимые для прослеживания требований, имеются в инструментах CTESK и SpecExplorer. В средствах модульного тестирования в этом качестве можно использовать текстовые сообщения в методах проверки утверждений, однако подобных возможностей лишены NModel и ModelJUnit.

- **Совместное использование моделей поведения, ситуаций и тестов, связанных с различными аспектами функциональности одного компонента на основе неинвазивной композиции**. За счет этого можно существенно облегчить многократное использование моделей в тестировании и других техниках верификации. NModel частично решает эту задачу для моделей тестов за счет использования возможности расширять определения классов новыми полями и методами в C#.

3. АРХИТЕКТУРНЫЙ КАРКАС ДЛЯ ТЕСТИРОВАНИЯ НА ОСНОВЕ МОДЕЛЕЙ

В данном разделе описывается предлагаемая архитектура инструментов тестирования на основе моделей, удовлетворяющая сформулированным выше требованиям. Однако прежде сделаем ряд замечаний, касающихся выбираемых средств решения поставленных задач.

- Как уже отмечалось, полезно использовать разные техники описания поведения тестируемых компонентов. Однако поддержка совершенно произвольных видов моделей в одной технологии, скорее всего, недостижима. Поэтому важно сразу отметить ряд практически важных походов, совместная поддержка которых реализуема.

  При моделировании программных интерфейсов с устоявшимися требованиями достаточно удобно применять *контрактные спецификации* в виде пред- и постусловий, опирающихся на модельное состояние компонентов. Подходы на их основе продемонстрировали достаточную масштабируемость и эффективность в терминах трудозатрат на описание некоторого набора элементов интерфейса [38, 50, 51].

  С другой стороны, для моделирования вычислений с плавающей точкой, сложных протоколов и ряда других видов ПО, иногда более удобно использовать *исполнимые модели*, являющиеся более простыми реализациями той же функциональности. Наиболее удобными на практике моделями такого вида оказываются расширенные автоматы и системы переходов с возможностью их композиции.

  При моделировании ряда реактивных систем, обрабатывающих большие потоки событий, или служб, регулярно обрабатывающих данные из большой базы, полезными оказываются *потоковые контракты*, описывающие ограничения не на конечный результат, а на обработку одного элемента во входном потоке данных.

- Модели поведения и модели ситуаций, а также сами тесты и элементы инструментария разработки предполагается оформлять в виде компонентов или наборов компонентов (подсистем) в рамках выбранной базовой компонентной технологии, с минимальным добавлением каких-либо новых конструкций, требующих дополнительной языковой и инструментальной поддержки.

  Такой подход позволит применять для работы с этими моделями и для их интеграции

с проверяемыми компонентами все инструменты, средства и техники, предлагаемые базовой компонентной технологией. Это значительно снижает затраты на развитие инструментария, поддерживающего такую технологию, и обеспечивает возможность использования того же инструментария и созданных моделей компонентов при разработке тестов для крупных систем. Тем самым создается основа для выполнения требований к компонентным технологиям верификации.

- Применение широко распространенных компонентных технологий и языков программирования позволяет снизить трудности обучения использованию инструментария, а также использовать многочисленные вспомогательные библиотеки, разработанные для решения специфических задач модульного тестирования.

- Верификационные системы часто из-за более многочисленного ассортимента компонентов получаются сложнее систем, для проверки которых предназначены. Поэтому еще более важно снижать их сложность и трудоемкость их развития. Для облегчения интеграции и переконфигурирования систем, состоящих из многочисленных компонентов, предлагается везде, где можно, использовать неинвазивные техники интеграции компонентов, избегающие внесения каких-либо изменений в их код. Это можно обеспечить за счет широкого использования образца *внедрения зависимостей* (dependency injection) [52] и поддерживающих его библиотек-контейнеров.

3.1. Виды компонентов и их интеграция

Основой инструментария тестирования на основе моделей предлагается сделать *контейнер внедрения зависимостей* (dependency injection container), позволяющий задавать список компонентов тестовой системы, инициализировать их и определять связи между ними внешним образом, без вмешательства в код этих компонентов.

Верификационная система строится из компонентов различных типов.

- Собственно, проверяемые компоненты.

  Они должны поддерживать возможность внешней инициализации с помощью контейнера внедрения зависимостей. Если это ограничение не выполнено, обычно достаточно просто написать компонент-обертку, удовлетворяющий ему и предоставляющий доступ к проверяемым операциям исходного компонента.

  Проверяемые компоненты не имеют зависимостей от тестовой системы, за исключением использования заглушек, подменяющих необходимые им для работы компоненты.

- Модели поведения (обобщенные контракты).

  Они оформляются на базовом языке программирования как классы с несколькими методами, выполняющими определенные роли. Например, если используется чисто декларативная спецификация, в ней должны быть определены пред- и постусловия, причем любой метод без побочных эффектов, возвращающий булевское значение, может играть эти роли. Для спецификации, использующей модельное состояние компонента, необходим синхронизатор состояния, вызываемый, чтобы поддерживать в соответствии состояние модели и реальное состояние проверяемого компонента. Исполнимые спецификации должны определять предусловия и изменение состояния.

  Модели поведения зависят от проверяемых компонентов или, в случае различия интерфейсов модели и компонента, – от адаптеров, устраняющих такие различия.

- Модели взаимодействия.

  При описании многокомпонентных систем иногда, помимо моделей отдельных компонентов, необходимо явно вводить модель их взаимодействия, позволяющую оценить корректность сложных наборов воздействий и реакций, в которых задействовано несколько компонентов, каждый из которых осведомлен лишь о части событий. Например, моделью взаимодействия является так называемая *семантика чередования* для асинхронных взаимодействий параллельно работающих компонентов, в рамках которой

корректен любой набор событий, который можно линейно упорядочить так, чтобы каждое отдельное событие в таком порядке стало корректным относительно моделей компонентов, создающих или обрабатывающих его [50].

Модели взаимодействия оформляются в виде шаблонных библиотечных модулей, привязываемых в конфигурационном файле к соответствующим группам компонентов. Для каждого конкретного взаимодействия порождается экземпляр такого шаблона, зависящий от моделей поведения вовлеченных в него компонентов.

- Модели ситуаций.

  Модели ситуаций оформляются на базовом языке программирования в виде методов, фиксирующих наступление определенных ситуаций после проверки описывающих их ограничений. Модели ситуаций для некоторой операции могут включать как пре-ситуации, определяемые аргументами операции и состояниями компонентов до ее вызова, так и пост-ситуации, соответствующие определенным свойствам результатов и состояний после работы операции. Модели пост-ситуаций могут иметь модельное состояние и методы-синхронизаторы, так же, как и контрактные спецификации.

  Модели ситуаций зависят от проверяемых компонентов, моделей поведения или теста, сообразно тому, в каких терминах они описывают ситуации.

  Модели ситуаций могут быть извлечены автоматически из моделей поведения, интерфейсов и кода проверяемых компонентов, поскольку критерии полноты тестирования на основе структуры кода или функциональности часто используются при построении тестов. Такие генерируемые компоненты далее будем называть *вторичными*.

- Тесты.

  Тесты, так же, как и модели ситуаций, могут создаваться разработчиками или генерироваться из моделей поведения и интерфейсов тестируемых компонентов. Каждый тест должен определять последовательность обращений к операциям тестируемого компонента (быть может, состоящую из единственного обращения) и значения параметров этих обращений, тестовые данные.

  - Для решения первой задачи можно использовать два подхода.
    - На практике более часто применяется связка из автоматной модели теста и генератора путей по графу переходов автомата. Эта техника положена в основу ModelJUnit и NModel.

      В этом случае автоматная модель теста должна определять методы, играющие роль действий, охранных условий, а также возвращающие текущее состояние автомата. Модель теста может зависеть от проверяемого компонента или от его модели поведения.
    - Для ряда случаев более эффективно применять монолитный генератор последовательностей на базе информации о тестируемом интерфейсе.
  - Генерация тестовых данных, особенно данных сложной структуры, может использовать большое количество компонентов, играющих различные роли.
    - Первичные генераторы, которые строят объекты некоторого типа. Такой генератор может быть устроен как итератор по некоторой коллекции.
    - Фильтры, выполняющие отсев данных, не удовлетворяющих определенным ограничениям.
    - Решатели ограничений, прямым образом строящие данные, удовлетворяющие некоторым ограничениям.
    - Комбинаторы, строящие данные сложного типа из простых объектов.
    - Преобразователи, генерирующие данные некоторого типа по более простому кодированному их представлению.

- Адаптеры.

  Адаптеры устраняют возможные расхождения между интерфейсами моделей и моделируемых ими компонентов.

  Адаптеры зависят от проверяемых компонентов. В тех случаях, когда они отвечают за синхронизацию модельного состояния, имеется зависимость и от соответствующей модели поведения.

  Можно отметить, что во многих случаях небольшие расхождения между модельным и проверяемым интерфейсами не требуют написания адаптера и могут быть устранены указанием библиотечной процедуры преобразования. Это относится к случаям, в которых различия сводятся к отсутствию ряда параметров, перестановке параметров местами или простым преобразованиям типов, например, чисел в строки и обратно. Во всех этих случаях адаптер строится не вручную, а автоматически, лишь по указанию соответствующего преобразования в конфигурационном файле тестовой системы.

- Заглушки (stubs, test doubles).

  Заглушки подменяют во время теста компоненты, от которых зависят проверяемые. Они бывают двух видов.

  - *Управляющая заглушка* (mock) передает в проверяемый компонент какие-то данные в виде возвращаемых ее методами результатов и служит дополнительным источником воздействий на тестируемый компонент.
  - *Наблюдающая заглушка* (test spy) фиксирует вызовы ее операций и их аргументы для проверки корректности сделанных тестируемым компонентом обращений.

  В принципе, одна заглушка может играть обе роли одновременно, но на практике такая потребность возникает крайне редко (это признак очень сложной организации теста, которую, возможно, имеет смысл пересмотреть).

  Если заглушки используются, проверяемый компонент зависит от них. Тест или модель поведения зависят от наблюдающей заглушки, которую они используют. И наоборот, управляющая заглушка сама зависит от теста, поскольку именно тест должен определять результаты очередного вызова ее операций.

- Вспомогательные компоненты. К вспомогательным относятся компоненты, решающие многочисленные задачи организации работы верификационной системы, интеграции ее составляющих и сбора информации о происходящих событиях.

  - Трассировщики различных видов событий. По сути, к каждому компоненту прикрепляется отдельный трассировщик событий, связанных с этим компонентом. Все трассировочные сообщения собираются одним или несколькими генераторами трасс.
  - Планировщики тестов, включающих асинхронные воздействия, отвечающие за создание отдельных процессов и нитей внутри тестовой системы и распределение воздействия по ним.
  - Диспетчеры и синхронизаторы отдельных операций в асинхронных и параллельных тестах.
  - Конфигураторы, определяющие связи внутри некоторых групп компонентов.

Рис. 1 демонстрирует одну из возможных конфигураций тестовой системы на основе предлагаемой архитектуры. Связи между компонентами, изображенные на рисунке, представляют собой зависимости, характерные для компонентов такого типа (хотя не все возможные зависимости изображены). Связи генератора трассы и конфигуратора не показаны, поскольку все или почти все компоненты связаны с ними.

Конкретный набор компонентов и связи между ними описываются в конфигурационном файле в XML-формате, поступающем в начале работы на вход контейнеру внедрения зависимостей, который инициализирует все компоненты и связывает их нужным образом. Такой способ задания связей позволяет строить различные конфигурации тестовой системы, не меняя кода ее компонентов и даже не имея к нему доступа. Вместе

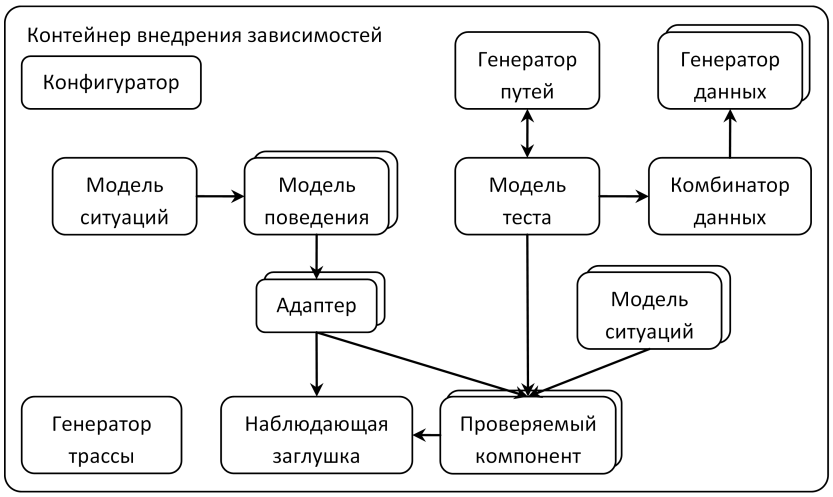

Рис. 1. Схема построения тестовой системы.

с тем, возможно определение жестких связей в самом коде, а также более гибкое связывание с помощью аннотаций и создание специальных компонентов-конфигураторов, которые содержат явную инициализацию компонентов и связей между ними на базовом языке программирования.

3.2. Реализация предложенного подхода

Для реализации предложенной архитектуры в качестве базового языка программирования был выбран язык Java. Он обладает возможностями, необходимыми для описания различных ролей классов и методов, а также связей между компонентами: средствами описания декларативной информации об элементах кода в виде аннотаций и поддержкой получения в динамике информации о структуре компонентов и сигнатурах их операций (интроспекция или рефлексия). Некоторые другие языки, скажем, C#, также обладают этими возможностями.

В качестве контейнера внедрения зависимостей была выбрана открытая библиотека Spring [45, 46], поддерживающая достаточно большой набор функций системы такого типа.

Для описания моделей поведения была разработана небольшая библиотека, похожая, с одной стороны, на библиотеки проверки утверждений в средствах модульного тестирования (используются разнообразные методы assert()) и, с другой стороны, на Microsoft CodeContracts (для доступа к результату операции и пре-значениям выражений используются методы result() и oldValue()). В отличие от CodeContracts поддерживается создание контрактных спецификаций, использующих модельное состояние. В разработанной библиотеке отсутствуют имеющиеся в CodeContracts кванторные выражения, и статический анализ ограничений не поддерживается.

Для описания моделей ситуаций также создана небольшая библиотека, обеспечивающая трассировку информации о покрытии указываемых ситуаций.

Тесты оформляются в стиле, аналогичном ModelJUnit и NModel, но с некоторыми расширениями, частично заимствованными из TestNG.

- Поддерживается расширенная иерархия элементов тестов: тестовые наборы, тесты, тестовые классы, тестовые методы. Тестовый набор состоит из тестов, один тест может включать в себя несколько тестовых классов.

- Тестовый класс описывает расширенный конечный автомат.

- Состоянием этого автомата считается результат работы всех методов, помеченных аннотацией State. Такое решение делает возможным добавление новых элементов в состояние без модификации ранее написанного кода. Состояние теста является списком состояний входящих в него классов.

- Тестовые методы определяют действия автомата, возможно, параметризованные. Значения параметров извлекаются из связанного

с тестовым методом провайдера данных. Провайдер может быть генератором наборов значений, определенных, например, как элементы некоторой коллекции, а может быть построен динамически из генераторов данных для разных параметров с определенной стратегией их комбинирования (выбирать все комбинации, каждое значение каждого параметра, все возможные пары значений и пр.). Провайдеры данных и способ их комбинирования задаются с помощью аннотаций метода и его отдельных параметров.

- Действия могут иметь охранные условия, оформляемые в виде методов, возвращающих булевский результат и зависящих от состояния объекта тестового класса. Охранное условие привязывается к тестовому методу при помощи аннотаций, без использования соглашений об именовании методов. Поэтому одно и то же условие может быть использовано для разных методов, и один метод может иметь несколько охранных условий. Кроме того, охранные условия могут иметь в качестве параметров любой набор, являющийся началом набора параметров соответствующего метода, в том числе пустой (в этом случае охранное условие зависит только от текущего состояния).

- Так же, как в TestNG, любой тестовый элемент – набор, тест, класс, метод – может иметь методы инициализации и финализации. Дополнительно можно определять конфигурационные методы, вызываемые при посещении очередного состояния.

Для построения заглушек используется свободная библиотека Mockito [26]. Она имеет достаточно богатые возможности для определения управляющих и наблюдающих заглушек и использует интуитивно понятный синтаксис при их описании. Этот пример показывает, что при наличии Java-библиотеки с необходимой функциональностью она без особых усилий может быть использована в рамках предлагаемой архитектуры.

4. ПРИМЕР ПОСТРОЕНИЯ ТЕСТА

Далее описывается пример использования предложенных решений при построении тестов для простой реализации функциональности банковского счета. Интерфейс тестируемого компонента приведен ниже.

```
public interface Account
{
  int getBalance();
  int getMaxCredit();

  Validator getValidator();
  void setValidator(Validator p);

  AuditLog getLog();
  void setLog(AuditLog log);

  int transfer(int sum);
}
```

Методы `getBalance()` и `getMaxCredit()` служат для получения текущих значений баланса и максимально возможного кредита. Баланс не может быть отрицательным и превосходящим максимально возможный кредит по абсолютной величине.

Метод `int transfer()` осуществляет перевод денег со счета или на счет, в зависимости от знака своего аргумента. Если аргумент положителен, соответствующая сумма добавляется на счет, увеличивая его текущий баланс. Если отрицателен, эта сумма списывается со счета, если при этом баланс не выходит за рамки максимального кредита. Результат этого метода – переведенная сумма или 0, если перевод не был сделан.

Данный счет позволяет использовать специализированный валидатор транзакций, `Validator`, который опрашивается при любом переводе с помощью предоставляемого им метода `boolean validateTransfer(Account a, int sum)` и может разрешить или заблокировать перевод.

Еще одна функция счета – запись данных о попытках перевода денег в трассу для последующего аудита. При этом вызываются методы интерфейса `AuditLog: logKind(String s)`, `logOldBalance(int b)`, `logSum(int sum)`, `logNewBalance(int b)`, записывающие, соответственно, итог транзакции (SUCCESS в случае успешного перевода, BANNED в случае его блокировки валидатором, IMPROPER в случае попытки снятия слишком большой суммы), предшествующее значение баланса, переводимую сумму и новое значение баланса.

```
public class AccountContract
{
  int balance;
  int maxCredit;

  Account checkedObject;

  public void setCheckedObject(Account checkedObject)
  {
    this.checkedObject = checkedObject;
    this.balance       = checkedObject.getBalance();
    this.maxCredit     = checkedObject.getMaxCredit();
  }


  public boolean possibleTransfer(int sum)
  {
    if (balance + sum > maxCredit) return true;
    else                           return false;
  }

  public boolean transferPostcondition(int sum)
  {
    boolean permission =
      checkedObject.getValidator().validateTransfer(checkedObject, sum);

    if (Contract.oldBooleanValue(possibleTransfer(sum)) && permission)
      return
         Contract.assertEqualsInt(Contract.intResult(), sum
                  , "Result should be equal to the argument")
      && Contract.assertEqualsInt(balance, Contract.oldIntValue(balance)+sum
                  , "Balance should be increased on the argument")
      && Contract.assertEqualsInt(maxCredit, Contract.oldIntValue(maxCredit)
                  , "Max credit should not change");
    else
      return
         Contract.assertEqualsInt(Contract.intResult(), 0
                  , "Result should be 0")
      && Contract.assertEqualsInt(balance, Contract.oldIntValue(balance)
                  , "Balance should not change")
      && Contract.assertEqualsInt(maxCredit, Contract.oldIntValue(maxCredit)
                  , "Max credit should not change");
  }

  public void transferUpdate(int sum)
  {
    if(   possibleTransfer(sum)
       && checkedObject.getValidator().validateTransfer(checkedObject, sum))
      balance += sum;
  }
}
```

Рис. 2. Модель основной функциональности счета.

Модель поведения для счета описана в виде двух независимых компонентов: модели основной функциональности и модели работы с трассировкой переводов. Это позволяет изменять и проверять эти две группы ограничений независимо. Описание основной функциональности выглядит так, как показано на рис. 2. Здесь показаны постусловие метода `transfer()` и соответствующий синхронизатор модельного состояния.

Описание требований к работе с трассой для аудита дано на рис. 3. Оно использует свободно распространяемую библиотеку для организации заглушек Mockito, вставляя заглушку для наблюдения за сделанными вызовами между счетом и связанным с ним трассировщиком. В ходе работы заглушка проверяет, что методы трассировщика вызывались в нужном порядке и с нужными

```
public class AccountLogSpy
{
  int balance;
  int maxCredit;

  Account checkedObject;
  AuditLog logSpy;

  public void setCheckedObject(Account checkedObject)
  {
    this.checkedObject = checkedObject;
    this.balance       = checkedObject.getBalance();
    this.maxCredit     = checkedObject.getMaxCredit();
    logSpy = Mockito.spy(checkedObject.getLog());
    checkedObject.setLog(logSpy);
  }

  int oldBalance;
  boolean wasPossible;

  public boolean possibleTransfer(int sum)
  {
    if (balance + sum > maxCredit) return true;
    else                           return false;
  }

  public void initSpy(int sum)
  {
    Mockito.reset(logSpy);
    oldBalance = balance;
  }

  public void transferLogSpy(int sum)
  {
    boolean permission = checkedObject.getValidator().validateTransfer(checkedObject, sum);

    if (wasPossible && permission)
    {
      Mockito.verify(logSpy).logKind("SUCCESS");
      Mockito.verify(logSpy).logNewBalance(balance);
    }
    else if (!permission) Mockito.verify(logSpy).logKind("BANNED");
    else                  Mockito.verify(logSpy).logKind("IMPROPER");

    Mockito.verify(logSpy).logOldBalance(oldBalance);
    Mockito.verify(logSpy).logSum(sum);
  }

  public void transferUpdate(int sum)
  {
    if(    possibleTransfer(sum)
       && checkedObject.getValidator().validateTransfer(checkedObject, sum))
    {
      wasPossible = true;
      balance += sum;
    }
    else
      wasPossible = false;
  }
}
```

Рис. 3. Модель работы с трассой для аудита.

```
public class AccountCoverage extends AccountContract
{
  public void transferCoverage(int sum)
  {
    boolean permission =
      checkedObject.getValidator().validateTransfer(checkedObject, sum);

    if (possibleTransfer(sum)) Coverage.addDescriptor("Possible transfer");
    else                       Coverage.addDescriptor("Too big sum");

    if (permission)            Coverage.addDescriptor("Permitted");
    else                       Coverage.addDescriptor("Not permitted");

    if(balance == 0)           Coverage.addDescriptor("Zero balance");
    else if(balance > 0)       Coverage.addDescriptor("Positive balance");
    else                       Coverage.addDescriptor("Negative balance");

    if(sum == 0)               Coverage.addDescriptor("Zero sum");
    else if(sum > 0)           Coverage.addDescriptor("Positive sum");
    else                       Coverage.addDescriptor("Negative sum");
  }
}
```

Рис. 4. Модель ситуаций.

```
@Test public class AccountTest
{
  Account account;
  boolean permission = true;

  @Mock Validator validatorStub;

  public AccountTest()
  {
    MockitoAnnotations.initMocks(this);
    Mockito.when(validatorStub.validateTransfer(Mockito.<Account>any()
                                , Mockito.anyInt())).thenReturn(true);
  }

  public void setAccount(Account account)
  {
    this.account = account;
    account.setValidator(validatorStub);
  }

  public Validator getPermitterStub() { return validatorStub; }

  @State public int getBalance() { return account.getBalance(); }

  @State public boolean getPermission() { return permission; }

  @Test
  @DataProvider(name = "sumArray")
  @Guard(name = "bound")
  public void testDeposit(int x)
  {
    account.transfer(x);
  }
```

Рис. 5. Модель теста.

аргументами. Поскольку построенная заглушка имеет модельное состояние, в ней также определен метод-синхронизатор этого состояния. Заглушка должна инициализироваться после каждого вызова `transfer()`, для этого в ней определен метод `initSpy()`.

Описание модели ситуаций представлено на рис. 4. В ней ситуации классифицируются по

```
  @Test
  @DataProvider(name = "sumIterator")
  public void testWithdraw(int x)
  {
    account.transfer(-x);
  }

  @Test
  @Guard(name = "bound")
  public void testIncrement()
  {
    account.transfer(1);
  }

  @Test
  public void switchPermission()
  {
    permission = !permission;
    Mockito.when(validatorStub.validateTransfer(Mockito.<Account>any()
                                , Mockito.anyInt())).thenReturn(permission);
  }

  public boolean bound()
  {
    return getBalance() < 5 || !permission;
  }

  public int[] sumArray = new int[]{1, 2};

  public Iterator<Integer> sumIterator()
  {
    return (Utils.ArrayToTypedList(sumArray)).iterator();
  }
}
```

Рис. 5 Модель теста (окончание).

четырем характеристикам: по корректности перевода, по прохождению валидации, по знаку предшествовавшего значения баланса и по знаку переводимой суммы. Поскольку определение ситуации зависит от модельного состояния счета и нуждается в синхронизаторе состояния, эта модель наследует модели функциональности, используя повторно определенные в ней элементы кода.

Модель теста для счета выглядит так, как показано на рис. 5.

Состояние теста состоит из двух элементов: текущего значения баланса и значения поля `permission`, определяющего результаты работы управляющей заглушки валидатора. Тестирование снятия денег и помещения их на счет разнесено по разным тестовым методам, хотя при этом вызывается один и тот же метод тестируемого объекта. Всего имеется четыре тестовых метода, соответствующих действиям в описываемом автомате.

- Метод `testDeposit()` проверяет помещение денег на счет. Он параметризован, значения параметров при работе теста берутся из массива `sumArray`. Кроме того, этот метод имеет охранное условие, позволяющее вызывать его только в тех случаях, когда текущий баланс не превосходит 5 и валидатор-заглушка допускает выполнение операций.
- Метод `testWithdraw()` проверяет снятие денег со счета. Значения его параметра берутся из того же массива, но с использованием метода-итератора.
- Метод `testIncrement()` проверяет добавление на счет суммы, равной 1. Он имеет то же самое охранное условие, что и метод `testDeposit()`.
- Метод `switchPermission()` ничего не проверяет, он только переключает текущее значение поля `permission`, чтобы протестировать работу счета с разными балансами и

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                  http://www.springframework.org/schema/aop
                  http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="accountImpl" class="mbtest.tests.AccountImpl"></bean>

  <bean id="accountTest" class="mbtest.tests.AccountTest">
    <property name="account" ref="accountImpl"/>
  </bean>

  <bean id="accountContract" class="mbtest.tests.AccountContract">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>

  <bean id="accountCoverage" class="mbtest.tests.AccountCoverage">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>

  <bean id="accountLogSpy" class="mbtest.tests.AccountLogSpy">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>

  <bean id="accountContractExecutor" class="mbtest.contracts.ContractExecutor">
    <property name="postcondition"
              value="mbtest.tests.AccountContract.transferPostcondition"/>
    <property name="updater" value="mbtest.tests.AccountContract.transferUpdate"/>
    <property name="object" ref="accountContract"/>
  </bean>

  <bean id="accountCoverageExecutor" class="mbtest.coverage.CoverageExecutor">
    <property name="coverage"
              value="mbtest.tests.AccountCoverage.transferCoverage"/>
    <property name="updater" value="mbtest.tests.AccountCoverage.transferUpdate"/>
    <property name="object" ref="accountCoverage"/>
  </bean>
```

Рис. 6. Конфигурация тестовой системы.

разными результатами валидации переводов.

Наконец, конфигурационный файл для среды Spring, определяющий связи между всеми перечисленными компонентами, приведен на рис. 6.

В этой конфигурации указано, как инициализировать объекты всех перечисленных типов, и, кроме того, определена привязка постусловий и синхронизаторов всех моделей к методу `transfer()` с помощью поддерживаемой Spring техники привязки аспектов.

Приведенный пример демонстрирует неинвазивность использованного метода построения тестовой системы из заданных компонентов – все эти компоненты ничего не знают друг о друге, кроме типов объектов, от которых они непосредственно зависят. В данной конфигурации модель основной функциональности и модель ситуаций представлены разными объектами, однако, поскольку вторая наследует первой, можно было бы реализовать их при помощи одного и того же компонента, играющего две разные роли.

5. ЗАКЛЮЧЕНИЕ

В работе представлена компонентная архитектура инструментария для тестирования на основе моделей, построенная на основе компонентных технологий с использованием принципа неинвазивной композиции. Впервые дается целостное представление такой архитектуры, хотя отдельные ее элементы уже использовались в инструментах типа TestNG, ModelJUnit и NModel. Описана реализация предложенного подхода на базе

```
  <bean id="accountSpyExecutor" class="mbtest.contracts.SpyExecutor">
    <property name="initialization" value="mbtest.tests.AccountLogSpy.initSpy"/>
    <property name="postcondition"
              value="mbtest.tests.AccountLogSpy.transferLogSpy"/>
    <property name="updater" value="mbtest.tests.AccountLogSpy.transferUpdate"/>
    <property name="object" ref="accountLogSpy"/>
  </bean>

  <aop:config>
    <aop:aspect id="accountContractAspect" ref="accountContractExecutor">
      <aop:pointcut id="accoutTransfer"
                    expression="execution(* mbtest.tests.Account.transfer(..))"/>
      <aop:around pointcut-ref="accoutTransfer" method="execute"/>
    </aop:aspect>

    <aop:aspect id="accountCoverageAspect" ref="accountCoverageExecutor">
      <aop:pointcut id="accoutCTransfer"
                    expression="execution(* mbtest.tests.Account.transfer(..))"/>
      <aop:around pointcut-ref="accoutCTransfer" method="execute"/>
    </aop:aspect>

    <aop:aspect id="accountSpyAspect" ref="accountSpyExecutor">
      <aop:pointcut id="accoutSTransfer"
                    expression="execution(* mbtest.tests.Account.transfer(..))"/>
      <aop:around pointcut-ref="accoutSTransfer" method="execute"/>
    </aop:aspect>
  </aop:config>
</beans>
```

Рис. 6 Конфигурация тестовой системы (окончание).

среды Spring, реализующей техники внедрения зависимостей. Кроме того, приведен пример его использования для построения теста, включающего несколько моделей разных аспектов поведения тестируемого компонента.

Имеющаяся на настоящий момент реализация описываемого подхода содержит следующие недостатки, которые нужно устранить.

- Во-первых, нужно модифицировать стандартный контекст внедрения зависимостей в Spring, чтобы он распознавал специфичные для тестовых систем виды компонентов (модель поведения, заглушку, модель ситуаций, модель теста и пр.) и требовал меньше параметров для их инициализации, а также автоматически строил их аспектную привязку к тестируемым компонентам. Это позволит значительно упростить создание и модификацию конфигурационных файлов, удалив из приведенного выше примера весь текст в рамках элемента `<aop:config>` и определения последних трех компонентов. Написание и понимание конфигурационных файлов значительно упростятся, и такие файлы смогут стать полноценным инструментом конфигурирования и настройки сложных тестовых наборов, не требующим их перекомпиляции.

- Во-вторых, пока не реализованы инструменты для генерации вторичных компонентов, моделей ситуаций и моделей тестов. Предполагается разработать их на основе одной из открытых библиотек для трансформации байт-кода Java. Такая реализация сделает возможной генерацию вторичных компонентов без доступа к исходному коду их прообразов. При этом удобство измерения покрытия кода или моделей поведения при использовании предлагаемой архитектуры должно стать сопоставимым с использованием для этого ведущих специализированных инструментов.

- В-третьих, логически различные элементы каркаса для построения тестовых систем – генераторы путей по автоматной модели, библиотечные генераторы данных, комбинаторы и пр. – также нужно выделить в виде внешне определяемых и конфигурируемых компонентов. Это позволит внести дополнительную гибкость в возможные конфигурации тестовой системы и проще интегрировать в рассматриваемый подход новые тех-

ники построения тестов.

Однако уже сейчас предложенная архитектура демонстрирует свои основные достоинства по сравнению с традиционными "монолитными" инструментами построения тестов – высокую гибкость, возможность совместного использования с разнообразными библиотеками, многочисленными инструментами, предназначенными для работы с компонентами Java (средами разработки, анализаторами кода, отладчиками и т.д.), возможность интеграции в более мощные среды.

СПИСОК ЛИТЕРАТУРЫ

1. *Szyperski C.* Component Software: Beyond Object-Oriented Programming. 2nd ed. Boston: Addison-Wesley Professional, 2002.
2. *Heineman G.T., Councill W.T.* Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, 2001.
3. *Parnas D.* Information Distribution Aspects of Design Methodology. Proc. of 1971 IFIP Congress. North Holland, 1971.
4. Tassey G. (ed.) The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Report. 2002.
5. *Hamill P.* Unit Test Frameworks. Tools for High-Quality Software Development. O'Reilly Media, 2004.
6. http://www.junit.org.
7. Broy M., Jonsson B., Katoen J.-P., Leucker M., Pretschner A. (eds.) Model-Based Testing of Reactive Systems. Advanced Lectures // LNCS. V. 3472. Springer-Verlag, 2005.
8. *Utting M., Legeard B.* Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, 2007.
9. *Peters D., Parnas D.* Using Test Oracles Generated from Program Documentation // IEEE Trans. on Software Engineering. 1998. V. 24. № 3. P. 161–173.
10. *Hoffman D.* Analysis of a Taxonomy for Test Oracles. Quality Week, 1998.
11. *Baresi L., Young M.* Test Oracles. Tech. Report CIS-TR-01-02. http://www.cs.uoregon.edu/∼michal/pubs/oracles.html.
12. *Harel D.* Statecharts: A visual formalism for complex systems // Science of Computer Programming. June 1987. V. 8. № 3. P. 231–274.
13. *Drusinsky D.* Modeling and verification using UML statecharts. Elsevier, 2006.
14. *Alur R., Dill D.L.* A Theory of Timed Automata // Journal of Theoretical Computer Science. 1994. V. 126. № 2. P. 183–235.
15. *Springintveld J., Vaandrager F., D'Argenio P.R.* Testing Timed Automata // Theoretical Computer Science. March 2001. V. 254. № 1–2. P. 225–257.
16. *Zhu H., Hall P., May J.* Software Unit Test Coverage and Adequacy // ACM Computing Surveys. December 1997. V. 29. № 4. P. 366–427.
17. *Кулямин В.В., Пакулин Н.В., Петренко О.Л., Сортов А.А., Хорошилов А.В.* Формализация требований на практике. Препринт № 13. ИСП РАН. Москва, 2006.
18. *Beck K.* Kent Beck's Guide to Better Smalltalk: A Sorted Collection. Cambridge University Press, 1998.
19. http://sunit.sourceforge.net/.
20. *Beust C., Suleiman H.* Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley Professional, 2007.
21. http://testng.org/.
22. http://www.dbunit.org.
23. http://www.httpunit.org.
24. http://jbehave.org/.
25. http://nspecify.sourceforge.net/.
26. http://mockito.org/.
27. http://easymock.org/.
28. *Tretmans J., Brinksma E.* TorX: Automated Model-Based Testing. Proc. of the 1st European Conference on Model-Driven Software Engineering. Nuremberg, Germany. December 2003. P. 31–43.
29. http: // fmt.cs.utwente.nl / tools / torx / introduction.html.
30. *Fernandez J.-C., Jard C., Jéron T., Nedelka L., Viho C.* Using On-the-Fly Verification Techniques for the Generation of Test Suites. Proc. of the 8th International Conference on Computer-Aided Verification // LNCS. V. 1102. P. 348–359. Springer, 1996.
31. http://www.inrialpes.fr/vasy/cadp/man/tgv.html.
32. *Ambert F., Bouquet F., Chemin S., Guenaud S., Legeard B., Peureux F., Vacelet N., Utting M.* Z-TT: A tool-set for test generation from Z and B using constraint logic programming. Proc. of Formal Approaches to Testing of Software. Brno, Czech Republic. August 2002. P. 105–119.

33. *Hartman A., Nagin K.* TCBeans Software Test Toolkit. Proc. of the 12th International Software Quality Week. May 1999.

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

35. http://www.conformiq.com/qtronic.php.

36. http://www.smartesting.com/index.php/cms/en/explore/products.

37. *Bourdonov I., Kossatchev A., Kuliamin V., Petrenko A.* UniTesK Test Suite Architecture. Proc. of FME 2002 // LNCS. V. 2391. P. 77–88. Springer, 2002.

38. *Кулямин В.В., Петренко А.К., Косачев А.С., Бурдонов И.Б.* Подход UniTesK к разработке тестов // Программирование. 2003. № 6. P. 25–43.

39. http://www.unitesk.ru.

40. *Campbell C., Grieskamp W., Nachmanson L., Schulte W., Tillmann N., Veanes M.* Testing Concurrent Object-Oriented Systems with Spec Explorer. Proc. of Formal Methods Europe // LNCS. V. 582. P. 542-547. Springer, 2005.

41. http://research.microsoft.com/en-us/projects/SpecExplorer/.

42. http://www.cs.waikato.ac.nz/∼marku/mbt/model-junit/.

43. *Jacky J., Veanes M., Campbell C., Schulte W.* Model-based Software Testing and Analysis with C#. Cambridge University Press, 2007.

44. http://nmodel.codeplex.com/.

45. http://mbt.tigris.org/.

46. *Barnett M., Fahndrich M., de Halleux P., Logozzo F., Tillmann N.* Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract. Proc. of ICSE 2009. Vancouver, Canada. May 2009.

47. http://research.microsoft.com/en-us/projects/contracts/.

48. *Kaner C., Bach J., Pettichord B.* Lessons Learned in Software Testing. John Wiley & Sons, 2002.

49. *Кулямин В.В.* Интеграция методов верификации программных систем // Программирование. 2009. № 4. P. 41–55.

50. *Kuliamin V., Petrenko A., Pakoulin N.* Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proc. of ISAS'2005. Berlin, Germany. Malek M., Nett E., Suri N. (eds.) Service Availability // LNCS. V. 3694. P. 68–83. Springer-Verlag, 2005.

51. *Grinevich A., Khoroshilov A., Kuliamin V., Markovtsev D., Petrenko A., Rubanov V.* Formal Methods in Industrial Software Standards Enforcement. Proc. of PSI'2006. Novosibirsk, Russia. 2006.

52. *Fowler M.* Inversion of Control Containers and the Dependency Injection Pattern. 2004. http://www.martinfowler.com/articles/injection.html.

53. *Johnson R., Hoeller J., Arendsen A., Risberg T., Sampaleanu C.* Professional Java Development with the Spring Framework. Wrox, 2005.

54. http://www.springsource.org.