Архитектура среды тестирования на основе моделей, построенная на базе компонентных технологий

Кулямин В. В.
kuliamin@ispras.ru

Аннотация. 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. Также описывается прототипная реализации предложенного подхода на базе свободно доступных библиотек и пример ее использования для построения тестов.

1. Введение

Увеличение разнообразия и количества задач, возлагаемых в современном мире на вычислительные системы, приводит к постоянному росту их сложности и усложнению 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 как программных, так и аппаратных составляющих вычислительных систем.

При их применении системы выстраиваются из относительно 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 обеспечивают совместную работу и облегченную интеграцию компонентов без необходимости специальных операций сборки и связывания (точнее, эти операции выполняются автоматически и часто на лету, без прерывания работы системы), только за счет помещения готовых компонентов в бинарном виде в рамки технологической инфраструктуры системы. В результате компоненты могут создаваться и поддерживаться совершенно независимыми организациями и разработчиками, что дает возможность построения весьма сложных систем без существенных расходов на поддержку их интегрированной разработки и сопровождения.

Компонентные технологии позволили значительно повысить сложность создаваемых продуктов и изделий, однако их развитие и распространение пока не сопровождается соответствующим прогрессом в технологиях контроля качества программного обеспечения (ПО). В связи с этим проблемы обеспечения качества компонентных систем, включающие обеспечение и проверку надежности и корректности их работы, совместимости и защищенности, не получают адекватных технологических решений, а лишь усугубляются с возрастанием сложности систем и ответственности решаемых ими задач.

Источником таких проблем служит обычно чрезвычайно высокая трудоемкость проверки корректности поведения систем при нелинейно растущем с увеличением числа компонентов количестве возможных сценариев их взаимодействия и сложных, часто неявных связей и зависимостей между ними. В свою очередь, эти трудности имеют следующие причины.

- Сложившаяся практика неполного и неточного описания интерфейсов компонентов. Эти неточности и неполнота и приводят к возникновению множества трудноуловимых зависимостей между компонентами, проявляющихся при замене компонентов на их новые версии или на аналоги от других поставщиков. Дело в том, что полный интерфейс модуля, как он понимался в работах Парнаса [3], одного из основоположников модульного подхода к разработке ПО, должен задавать не только его синтаксическую структуру (имена реализуемых операций, типы их параметров и результатов, и соответствующие структуры данных), но и семантику (правила использования операций и ожидаемое в ответ поведение модуля). На практике же обычно ограничиваются только фиксацией синтаксиса, семантика описывается, да и то, чаще всего не полностью и не вполне однозначно, только для интерфейсов, входящих в системные библиотеки и имеющих достаточно долгую историю использования. С такими интерфейсами обычно связаны высокие риски и накладные расходы в случае несовместимости или некорректного взаимодействия компонентов, использующих их. Однако «менее рискованных» интерфейсов, для которых семантические требования не описываются совсем или фиксируются в виде одной фразы, указывающей их основную функциональность в нормальной ситуации, гораздо больше, и меньшие расходы, связанные с отдельными ошибками, в сумме дают очень значительную величину. В 2002 году годовые потери одной только экономики США от многочисленных ошибок в ПО были оценены в 60 миллиардов долларов [4].

- Трудности интеграции в технологические процессы разработки ПО средств верификации и тестирования на соответствие требованиям.

Такие средства по большей части представлены в виде «монолитных» инструментов, реализующих большое число разнообразных функций, от оформления требований до создания отчетов о проведенном тестировании. Причем набор деятельностей, выполняемых в ходе контроля качества, связи между ними, а также поддерживаемые техники верификации и тестирования, определяются разработчиками инструмента и не могут быть изменены и расширены сколь-нибудь существенным образом. Более того, в последнее время многие компании, предоставляющие средства разработки ПО, предлагают интегрированные решения, охватывающие практически весь процесс разработки, причем включение в них посторонних, заранее не предусмотренных техник и средств невозможно, либо очень трудоемко. Вместе с тем, процессы разработки ПО в разных организациях сильно различаются, в связи с отличиями в создаваемых ими программах, в требованиях клиентов и способах взаимодействия с ними, в используемых бизнес-моделях и т.д. Поэтому все более востребованы инструменты разработки, имеющие модульную архитектуру, способные с минимальными накладными расходами интегрироваться с другими инструментами и включать модули, реализующие передовые, более эффективные техники выполнения отдельных видов деятельности по разработке ПО. Эти требования можно суммировать так: инструменты разработки сложных систем тоже должны быть компонентными и облегчать свое включение в разнообразные технологические цепочки. Среди средств верификации такими свойствами пока обладают лишь инструменты *модульного тестирования* (unit testing) [5], самым известным из которых является JUnit [6], модульных инструментов, поддерживающих более сложные виды верификации или тестирование на соответствие требованиям, практически нет.

- Отсутствие поддержки масштабного многократного использования возникающих в ходе контроля качества артефактов — тестов, верифицированных утверждений, моделей и пр. Эта проблема особенно важна именно для компонентных технологий, поскольку верификация одного компонента чаще всего имеет приемлемую стоимость, в ее ходе создаются многочисленные вспомогательные артефакты, 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.

Преодолеть возникающие трудности можно с помощью *компонентных технологий верификации* программных и аппаратных систем. Такие

технологии позволили бы независимо проверять качество отдельных компонентов в соответствии с требованиями к ним, и использовать получаемые при этом модели, тесты и пр. для более аккуратного контроля корректности их интеграции и качества компонентных систем, постепенно наращивая количество проверенных компонентов, подсистем, различных требований к системе в целом. Более детально, подобная технология должна обладать следующими характеристиками.

- Предоставлять средства определения требуемых свойств отдельного компонента, включающих требования компонента к его окружению и его ответные обязательства, предоставляемые им службы и их характеристики.
- Давать возможность использовать те же средства для фиксации необходимых свойств подсистем, объединяющих в себе несколько компонентов (и других подсистем).
- Предоставлять инструментарий для независимой проверки компонентов и подсистем на соответствие требованиям к ним.
- Обеспечивать контроль взаимного соответствия взаимных требований и обязательств компонентов и подсистем при их интеграции и совместной работе.
- Позволять эффективно использовать результаты проверки отдельных компонентов и подсистем и наработанные в ее ходе материалы в виде формальных моделей, ограничений использования, верифицированных утверждений, тестов и т.п. в неизменном виде при контроле качества включающих их подсистем и систем.

Помимо этих свойств, чтобы получить признание на практике, компонентная технология верификации должна быть расширением одной из существующих компонентных технологий разработки и задействовать по большей части ее элементы, уже знакомые и привычные для разработчиков.

Свойства технологии, связанные с ее гибкостью, возможностью использования для решения широкого множества задач, во многом обуславливаются лежащей в ее основе архитектурой, определяющей общую организацию работ, входные и выходные данные на каждом этапе. Для компонентной технологии 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 пр.

В данной работе представлены некоторые элементы компонентной технологии верификации, использующей методы тестирования на основе моделей и базирующейся на компонентных технологиях платформы Java, однако представленная технология находится в процессе создания и в окончательном виде пока не сформирована. Основная же часть статьи посвящена архитектурному каркасу (framework) этой технологии,

включающему ее инструментальные библиотеки, а также набор основных типов компонентов и правила их интеграции. Этот каркас может быть использован в несколько более широком контексте, чем сама технология, поэтому целесообразно иметь его отдельное описание.

Изложение организовано следующим образом. Во втором разделе рассматривается подход к тестированию на основе моделей, анализируются его требования к архитектуре поддерживающего инструментария и реализация этих требований в уже имеющихся инструментах. 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, как она должна работать.
  Модель поведения служит основой для *тестового оракула* [9-11] — компонента или процедуры, производящей оценку поведения системы во время тестирования.
  Чаще всего при тестировании используются модели в виде автоматов разного вида: наиболее простые конечные автоматы, расширенные автоматы, системы помеченных переходов [7], Statecharts [12,13], временные автоматы [14,15] и т.д. Однако в таком качестве можно использовать другие разновидности моделей: контрактные спецификации в виде пред- и постусловий операций, алгебраические спецификации в виде правил эквивалентности различных цепочек вызовов операций, трассовые или сценарные модели, описывающие возможные последовательности 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 определяет конечный набор классов эквивалентности ситуаций, подразумевая, что при тестировании достаточно проверить работу системы хотя бы в одной ситуации из каждого класса. Достаточно часто также для задания модели ситуаций используется конечный набор элементарных событий, при этом каждая ситуация соответствует некоторому множеству таких событий. Примером такого события может являться выполнение заданной инструкции в коде тестируемой системы. В более сложных случаях определяемые классы ситуаций могут пересекаться, и им могут приписываться приоритеты и 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. Заметим, что под полнотой тестирования здесь имеется в виду удовлетворение определенному критерию (скажем, достижение 80% покрытия кода), а совсем не исчерпывающий перебор всех практически возможных ситуаций. Критерий полноты тестирования задает свойства набора тестов, позволяющие считать его достаточно хорошо представляющим разнообразное поведение проверяемой системы для решения некоторого круга задач и больше тестов не создавать. Метрики полноты тестов представляются в терминах процентной доли проверяемых ими классов ситуаций разных видов. Обычно в качестве критерия полноты тестирования устанавливается достижение какого-то значения метрики, но при этом само значение метрики бывает полезно знать дополнительно. Чаще всего полнота тестирования определяется на основе покрытия классов эквивалентных ситуаций или элементарных событий, и в этих случаях говорят о *критерии тестового покрытия*, *метрике тестового покрытия*, а достигнутый в ходе тестирования процент покрытых классов ситуаций называют *тестовым покрытием*.

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

- *Вероятностные методы* используют псевдослучайную генерацию данных элементарных 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 разнообразных тестов, так как построение каждого отдельного теста требует небольших затрат.

- *Нацеленные методы*, наоборот, строят тестовые данные и последовательности целенаправленно, так, чтобы те удовлетворяли определенным свойствам, чаще всего — реализовывали ситуации из определенных 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 с помощью комбинирования различных типов их элементов по определенным схемам. Они занимают промежуточное положение: на создание одного теста нужно существенно меньше усилий, чем при нацеленном построении, но несколько больше, чем при вероятностном. Тестовый набор получается значительно больше, чем при нацеленном тестировании, но много меньше, чем при вероятностном, хотя в нем и достигается большее разнообразие. Кроме того, комбинаторные методы обычно гарантируют, что в полученном наборе нет совершенно идентичных тестов, которые очень часто возникают в случайно сгенерированных наборах. Некоторые комбинаторные методы, использующие эффективную фильтрацию «лишних», не вносящих собственный вклад в достижение полноты тестов, вполне сопоставимы с нацеленными методами по характеристикам получаемого тестового набора. 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 и проверки всех возможных состояний и переходов.

2.1. Требования к представлениям моделей

Инструменты, поддерживающие тестирование на основе моделей, должны работать с какими-то представлениями моделей поведения и моделей ситуаций. Для полноценной поддержки этого подхода к тестированию и повышения удобства его использования в практике разработки ПО, эти представления должны обладать следующими свойствами.

- Модели поведения.

  - Средства для описания требований в разных стилях, как минимум в виде чистых декларативных ограничений (утверждений о свойствах входных данных операций и их результатов), 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, с одной стороны, как можно более ясно отражать имеющиеся требования, сформулированные на естественном языке, и, с другой стороны, допускать возможность их переформулирования в другом стиле в ходе построения моделей, поскольку оно позволяет выявить неточности, рассогласования и неполноту в исходных требованиях и углубить их понимание [17].
  - Средства для явного указания связей с документами, содержащими требования, и стандартами, если они относятся к проверяемой системе. Такие связи используются для прослеживания исходных требований и установления того, что все необходимые ограничения действительно присутствуют в модели, а лишних ограничений в ней нет. В дальнейшем они же позволяют проследить связи требований с полученными автоматически тестами.
  - Возможность автоматического преобразования моделей в тестовые оракулы.
  - Возможность использования структуры моделей для определения критериев тестового покрытия.
  - Стоит поддерживать возможности использования таких моделей в рамках других техник верификации.
  - 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.

  - Средства для описания ситуаций, связанных с различными аспектами поведения и структуры тестируемой системы. На практике критерии полноты тестирования используют разнообразные свойства для характеризации ситуаций: структуру входных данных и результатов, внутренние состояния системы, выполняемые инструкции кода и вызываемые внутри функции, шаблоны передачи данных внутри системы, шаблоны взаимодействия различных частей

системы при выполнении операции, характеристики ожидаемого поведения, возможные ошибки в системе и пр.
  - Средства для явного указания связей определяемых классов ситуаций с исходными требованиями к системе.
  - Возможность автоматического преобразования в компоненты, определяющие класс текущей ситуации, измеряющие достигнутое покрытие.
  - Возможность автоматического извлечения из описания классов ситуаций ограничений на входные данные и состояние тестируемой системы или моделей ее поведения, чтобы можно было их использовать для нацеленного построения соответствующих тестов.

Как видно, требования к представлению моделей делятся на три типа: достаточная для практических целей выразительность, прослеживаемость к требованиям и возможность автоматического решения различных задач построения тестов.

2.2. Инструменты модульного тестирования

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

Важным классом инструментов тестирования, в большой степени обладающих свойствами модульности, являются 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 для реализации дополнительных функций 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 структуризацию: в нем выделяются тестовые наборы (test suites), состоящие из тестов (tests). Далее в этом списке характеристик эти термины используются только в узком, специфическом для TestNG смысле. Тесты и тестовые наборы определяются

конфигурацией тестов, описываемой в некотором формате на базе XML. Тест составляется из методов нескольких классов, 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 некоторых данных перед выполнением тестов и освобождения занятых ресурсов после, могут быть определены для всех элементов иерархии: для наборов, тестов, классов и методов.
  - Возможно определение зависимостей между тестовыми методами и между методами и группами, позволяющее управлять порядком их выполнения и отменять выполнение тестового метода, если один из тестовых 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 заданной коллекции объектов, либо как последовательность результатов, возвращаемых определенным 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.
  - Основной способ описания выполняемых тестовыми методами проверок, как и во многих других инструментах модульного тестирования — это использование библиотеки методов-утверждений. Каждый такой метод (обычно их названия начинаются с assert) проверяет простое свойство своих аргументов (равенство, неравенство null, вхождение символа в строку, вхождение объекта в коллекцию и пр.) и при его нарушении выдает трассировочное 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 развиваемые независимо модули для решения разных более специфичных задач. Например,

dbUnit [22] — для организации работы с базами данных в модульных тестах используется, httpUnit [23] — для обработки HTTP-запросов. Для более наглядной записи выполняемых в тестах проверок (близкой к формулировкам естественных языков) можно применять библиотеки, предоставляемые инструментами 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 желательной компонентной 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 заранее позаботились о предоставлении подходящего набора интерфейсов.
  К инструментам такого типа относятся практически все многочисленные исследовательские прототипные средства тестирования на основе моделей и ряд более стабильных инструментов, использовавшихся во многих разных проектах. В этой второй категории находятся TorX [28,29], TGV [30,31], BZ-TT [32] и Gotcha-TCBeans [33,34]. Все они основаны на моделировании проверяемой системы в виде различных автоматных моделей. На тех же принципах в целом построены и коммерческие инструменты Conformic Qtronic [35] и Smartesting Test Designer (ранее Leirios) [36].

- Инструменты на основе расширений широко используемых языков программирования, которые сохраняют «монолитность». Примерами являются поддерживающие технологию UniTESK [37-39] инструменты CTESK и JavaTESK, а также SpecExplorer [40,41], разрабатываемый в Microsoft Research. В обоих случаях для моделирования поведения тестируемых систем используется комбинация из автоматных моделей и контрактных спецификаций.

- Инструменты, использующие для оформления моделей обычные языки программирования и обладающие рядом характеристик модульности. В частности, эти характеристики включают возможности достаточно легкой интеграции с компонентами от независимых разработчиков и использования самих таких инструментов как части 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 стоит рассмотреть подробнее.

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

- Модель теста описывается в виде расширенного конечного автомата, представленного на языке программирования (Java или C#) в виде тестового класса, как и в средствах модульного тестирования. Такой класс либо помечается с помощью некоторой аннотации (атрибута в C#), либо реализует определенный интерфейс. Элементы автоматной модели — состояния, переходы, охранные условия переходов — задаются с помощью методов и полей этого класса.
- Состояние в NModel задается набором значений полей тестового класса (можно определять поля, не включаемые в состояние, помечая их специальными атрибутом), а в ModelJUnit — результатом метода getState().
- *Действия*, в результате выполнения которых выполняются переходы 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. В обоих случаях охранные условия представлены как методы, имеющие имена, построенные из имени соответствующего метода-действия с некоторым постфиксом.
- NModel дополнительно имеет следующие возможности.
  - Композиция нескольких моделей, в которых одноименные действия рассматриваются как выполняемые одновременно. Модели для композиции указываются инструменту построения тестов перечислением имен соответствующих классов.

- Проверка моделей (model checking). Свойства безопасности проверяются за счет анализа возможности нарушения инвариантов состояний, представленных как методы с особым атрибутом. Свойства живучести могут удостоверяться за счет анализа достижимости стабильных 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 некоторых свойств в какой-то точке кода внутри метода (Contract.Assert).
  - Инварианты классов — оформляются в виде методов со специальной аннотацией, содержащих обращения к Contract.Invariant.
- Помимо обычных выражений на C# можно использовать следующие.
  - Кванторные выражения, оформленные в виде обращений к методам 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 к которым описаны.

2.4. Итоги обзора существующих инструментов

Из приведенного обзора видно, что компонентные средства тестирования на основе моделей в последние годы активно развиваются, однако пока не достигли необходимой функциональности, отставая 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, прежде всего, для поддержки точности и полноты моделирования. Не менее важно и многократное использование моделей, поскольку на основе одной модели поведения компонента могут быть построены разнообразные тесты, нацеленные на достижение различных целей, как для этого компонента, так и для содержащих его подсистем. Еще одна полезная возможность — использование таких моделей в рамках других техник 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 сочетания разнообразных методов и техник построения тестов, а также других методов верификации. Это требование является следствием сложности и разнообразия требований к современным программным компонентам [48,49]. Библиотека CodeContracts позволяет указывать только чистые декларативные ограничения, что существенно снижает возможности ее использования для большинства практически значимых систем, поведение которых зависит от внутренних состояний их компонентов.

- **Явное определение моделей ситуаций для построения тестов.** Неявное задание модели ситуаций в имеющихся сейчас инструментах ограничивает возможности использования различных критериев полноты тестирования и затрудняет их осознанный выбор разработчиками тестов.

- **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 так можно использовать текстовые сообщения в методах проверки утверждений, однако подобных возможностей лишены NModel и ModelJUnit. Отметим, что добавление такой функциональности в модульный инструмент не представляет большого труда.

- **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 описывает предлагаемый подход к построению компонентной архитектуры инструментов тестирования на основе моделей, удовлетворяющей сформулированным выше требованиям. Однако прежде стоит сделать ряд замечаний, касающихся выбираемых средств решения поставленных задач.

- Как отмечалось выше, необходимо использовать в моделях разные техники описания ограничений на поведение тестируемых компонентов. При этом нужно отдавать себе отчет в том, что поддержка совершенно произвольных видов моделей в рамках одной технологии, скорее всего, недостижима. Поэтому нужно 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 моделируемых 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 *операционные модели*, являющиеся, по сути, альтернативными реализациями той же функциональности. Наиболее удобными на практике моделями такого вида оказываются расширенные автоматы и системы переходов с возможностью их композиции.
  При моделировании некоторых реактивных систем, обрабатывающих большие потоки событий, или служб, предназначенных для регулярной обработки данных из большой базы, полезными оказываются *потоковые контракты*, описывающие ограничения не на конечный результат обработки, а на обработку одного структурного 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. Соответственно, значительно снижаются затраты на сопровождение и развитие инструментария, поддерживающего такую технологию. Это обеспечивает возможность использования того же инструментария и созданных моделей компонентов при разработке тестов для подсистем и 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 при необходимости использовать многочисленные вспомогательные библиотеки, разработанные для решения специфических задач модульного тестирования.

- Верификационные системы часто из-за более многочисленного ассортимента различных компонентов получаются сложнее систем, для проверки которых предназначены. Поэтому еще более важно снижать их сложность и трудоемкость их поддержки и развития. Для облегчения интеграции и переконфигурирования систем из многочисленных компонентов предлагается везде, где можно, использовать неинвазивные техники композиции и 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, а также других средств современных компонентных технологий.

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, входящих в систему, инициализировать их состояние и определять связи между ними внешним образом, без вмешательства в код этих компонентов.

Верификационная система строится из компонентов различных типов.

- Собственно, проверяемые компоненты.
  На них накладывается только одно ограничение: возможность их внешней инициализации с помощью контейнера внедрения зависимостей. В большинстве случаев это ограничение выполняется, иначе 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 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, возвращающий булевское значение, может играть эти роли. Для спецификации, использующей модельное состояние компонента, необходимо определить синхронизатор состояния, вызываемый, чтобы поддерживать в соответствии состояние модели и реальное состояние проверяемого компонента. Исполнимые спецификации должны определять предусловия и модельные операции.
  Модели поведения зависят от проверяемых компонентов или, в случае существенных различий в интерфейсах между моделью и моделируемым компонентом — 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 конгломератов воздействий и реакций, в которых задействовано несколько компонентов, каждый из которых осведомлен лишь о событиях, относящихся к его интерфейсу. 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 им 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 входными аргументами операции и состояниями вовлеченных компонентов до ее вызова, так и пост-ситуации, соответствующие

определенным свойствам результатов и состояний компонентов 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 сгенерированы 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 компоненты, генерируемые из других, 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, использующий информацию 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 требует написания адаптера, поскольку 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.

Рисунок 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.

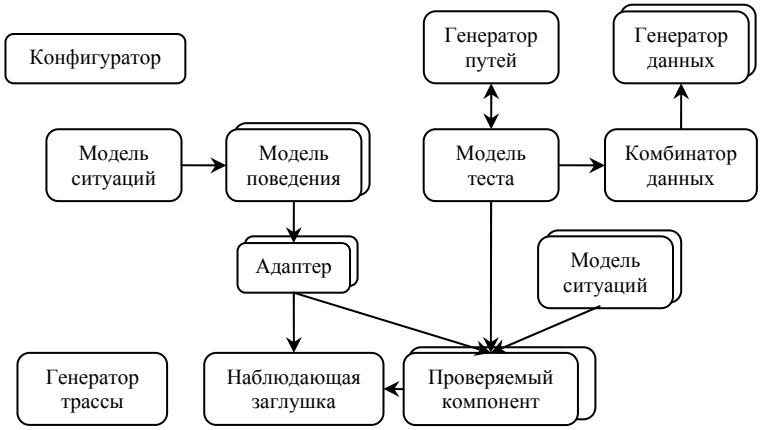

Рис. 1. Схема построения тестовой системы.

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 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 многими языковыми 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).

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 поддержка статического анализа ограничений.

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 конфигурационные 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.

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.

```
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;
  }
}
```

Здесь показаны постусловие метода `transfer()` и соответствующий синхронизатор модельного состояния.

Описание требований к работе с трассой для аудита дано ниже. 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 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()`.

```
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;
  }
}
```

Описание модели ситуаций представлено ниже. В ней ситуации классифицируются по четырем характеристикам: корректность перевода, прохождение валидации, знак предшествовавшего значения баланса и знак 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.

```
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");
  }
}
```

Модель теста для счета выглядит следующим образом.

```
@Test public class AccountTest
{
```



```
  Account account;
  boolean permission = true;

  @Mock Validator validatorStub;

  public AccountTest()
  {
    MockitoAnnotations.initMocks(this);

Mockito.when(validatorStub.validateTransfer(Mockit
o.<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);
  }

  @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();
  }
}
```

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 добавления 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 разными результатами валидации переводов.

Наконец, конфигурационный файл для среды Spring, определяющий связи между всеми перечисленными компонентами, выглядит так.

```
<?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>

  <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>
```

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. Описана реализация предложенного подхода на базе 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, заглушка, 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 почти весь текст в рамках элемента `<aop:config>`.

- 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.

- 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.

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.

Литература

[1] C. Szyperski. Component Software: Beyond Object-Oriented Programming. 2-nd ed. Addison-Wesley Professional, Boston, 2002.
[2] G. T. Heineman, W. T. Councill. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, 2001.
[3] D. Parnas. Information Distribution Aspects of Design Methodology. Proc. of 1971 IFIP Congress, North Holland, 1971.
[4] G. Tassey, ed. The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Report, 2002.
[5] P. Hamill. Unit Test Frameworks. Tools for High-Quality Software Development. O'Reilly Media, 2004.

[6] http://www.junit.org.
[7] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (Eds.) Model-Based Testing of Reactive Systems. Advanced Lectures. LNCS 3472, Springer-Verlag, 2005.
[8] M. Utting, B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, 2007.
[9] D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Trans. on Software Engineering, 24(3):161–173, 1998.
[10] D. Hoffman. Analysis of a Taxonomy for Test Oracles. Quality Week, 1998.
[11] L. Baresi, M. Young. Test Oracles. Tech. Report CIS-TR-01-02. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.
[12] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231-274, June 1987.
[13] D. Drusinsky. Modeling and verification using UML statecharts. Elsevier, 2006
[14] R. Alur, D. L. Dill. A Theory of Timed Automata. Journal of Theoretical Computer Science, 126(2):183-235, 1994.
[15] J. Springintveld, F. Vaandrager, P. R. D'Argenio. Testing Timed Automata. Theoretical Computer Science, 254(1-2):225-257, March 2001.
[16] H. Zhu, P. Hall, J. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427, December 1997.
[17] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт 13, ИСП РАН, Москва, 2006.
[18] K. Beck. Kent Beck's Guide to Better Smalltalk: A Sorted Collection. Cambridge University Press, 1998.
[19] http://sunit.sourceforge.net/.
[20] C. Beust, H. Suleiman. 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] J. Tretmans, E. Brinksma. TorX: Automated Model-Based Testing. Proc. of 1-st European Conference on Model-Driven Software Engineering, Nuremberg, Germany. pp. 31-43, December 2003.
[29] http://fmt.cs.utwente.nl/tools/torx/introduction.html.
[30] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, C. Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. Proc of 8-th International Conference on Computer-Aided Verification, LNCS 1102:348-359, Springer, 1996.
[31] http://www.inrialpes.fr/vasy/cadp/man/tgv.html.
[32] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, M. Utting. Z-TT: A tool-set for test generation from Z and B using constraint logic programming, Proc. of Formal Approaches to Testing of Software, pp. 105-119, Brno, Czech Republic, August 2002.
[33] A. Hartman, K. Nagin. TCBeans Software Test Toolkit. Proc. of 12-th International Software Quality Week, May 1999.
[34] E. Farchi, A. Hartman, S. S. Pinter. Using a model-based test generator to test for standard conformance. IBM Systems Journal, 41(1):89-110, 2002.

[35] http://www.conformiq.com/qtronic.php.
[36] http://www.smartesting.com/index.php/cms/en/explore/products.
[37] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391:77-88, Springer, 2002.
[38] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
[39] http://www.unitesk.ru.
[40] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes. Testing Concurrent Object-Oriented Systems with Spec Explorerю Proc. of Formal Methods Europe, LNCS582:542-547, Springer, 2005.
[41] http://research.microsoft.com/en-us/projects/SpecExplorer/.
[42] http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/.
[43] J. Jacky, M. Veanes, C. Campbell, W. Schulte. Model-based Software Testing and Analysis with C#. Cambridge University Press, 2007.
[44] http://nmodel.codeplex.com/.
[45] http://mbt.tigris.org/.
[46] M. Barnett, M. Fahndrich, P. de Halleux, F. Logozzo, N. Tillmann. 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] C. Kaner, J. Bach, B. Pettichord. Lessons Learned in Software Testing. John Wiley & Sons, 2002.
[49] В. В. Кулямин. Интеграция методов верификации программных систем. Программирование, 35(4):41-55, 2009.
[50] V. Kuliamin, A. Petrenko, N. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. Proc. of ISAS'2005, Berlin, Germany. M. Malek, E. Nett, N. Suri , eds. Service Availability. LNCS 3694, pp. 68-83, Springer-Verlag, 2005.
[51] A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov. Formal Methods in Industrial Software Standards Enforcement. Proc. of PSI'2006, Novosibirsk, Russia, 2006.
[52] M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern. 2004. http://www.martinfowler.com/articles/injection.html.
[53] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu. Professional Java Development with the Spring Framework. Wrox, 2005.
[54] http://www.springsource.org.