Подход к тестированию наведенных дефектов

В. А. Тихомиров tikhomirov@motorola.com
В. П. Котляров r36959@motorola.com

Санкт-Петербургский государственный политехнический университет

При разработке программного обеспечения часто возникает ситуация, когда в уже существующую систему добавляется новая функциональность. После этого необходимо тщательное тестирование всей системы, что требует больших затрат. В работе предложен метод решения этой проблемы, основанный на широко известном верификационном подходе — проверке на моделях (model checking). Используется идея частичных спецификаций (partial specifications) — то есть при тестировании не происходит построения и использования полной модели системы и интегрируемой компоненты. Вместо этого моделируются только их отдельные участки (так называемые регионы), которые отвечают за взаимодействие. Для построения модели используются базовые протоколы Летичевского. По полученной модели на основе уже существующих инструментов верификации и тестирования VRS и TAT генерируются тесты.

Введение

В настоящее время, в связи с постоянным увеличением объема разрабатываемых систем, процесс интеграции новых компонент в уже существующую версию программного продукта является ключевым. Это связано с тем, что процесс разработки компоненты проводится часто независимо от других компонент, и от того, как будет

проводиться интеграция компонентов, зависит работоспособность всей системы. Целью интеграционного тестирования системы, расширенной новой функциональностью, является проверка того, как новый модуль согласуется с остальной частью программного продукта (с остальными модулями продукта).

Использование при интеграционном тестировании проверки на моделях (model checking) — известного подхода к верификации и тестированию программных систем — затруднено по причине отсутствия средств настройки на локальную задачу — тестирование «стыка» система / новая компонента. Если формальных моделей системы и компоненты еще нет, то для тестирования «стыка» нужно потратить много усилий для их создания. Если модели системы и новой компоненты уже есть, то из них нужно как-то выделить ту часть, которая нужна в рамках данной задачи — ведь считается, что и система, и компонента оттестированы по отдельности. Это можно делать путем определения параметров и фильтров при генерации тестовых трасс, а можно создать новую формальную модель, включив в нее спецификации только тех участков кода системы и интегрируемой компоненты, которые отвечают за взаимодействие, создавая, таким образом, частичные модели (partial specifications) системы и компоненты. Последний вариант более выигрышный, потому что не требует наличия полных моделей системы и компоненты. Именно такой подход предлагается в данной статье. Несмотря на большое количество исследований в области частичных спецификаций (обзоры можно найти в работах [1–3]), до сих пор эта идея не использовалась для интеграционного тестирования.

Наш подход использует в качестве языка построения моделей базовые протоколы Летичевского [4]. Этот формализм создан Институтом Кибернетики НАН Украины совместно с компанией Motorola и предназначен для формализации поведенческих аспектов программных систем. В качестве формы записи базовых протоколов мы, вслед за [5], используем MSC-диаграммы [4], которые являются языком описания поведения системы в виде последовательности событий и хорошо согласуются с парадигмой базовых протоколов. В качестве инструментальных технологий мы используем верификатор VRS [5] и систему тестирования TAT [6], которые созданы специально для базовых протоколов Летичевского.

В работе представлены схема и описание самого подхода, а также приводится пример — в телекоммуникационную систему добав-

ляется компонента, отвечающая за реализацию сервиса коротких сообщений (Short Message Service — SMS).

1. Частичные спецификации

При создании и использовании полной модели программного продукта возникают следующие трудности: создание такой модели требует больших трудозатрат, полные модели очень объемны, для их анализа и обработки требуется много времени и ресурсов. Построение и анализ моделей для фрагментов программного продукта во многом решает описанные трудности. Такой подход носит название частичных спецификаций (partial specifications). Этот подход используется многими исследователями в различных областях программной инженерии (подробные обзоры содержатся в работах [1–3]). Мы остановимся лишь на основных работах последних лет.

В [7] рассматривается ситуация, когда необходимо построить модель по требованиям к системе для последующего тестирования. В этом случае для каждого требования строится своя спецификация на основе логики первого порядка. После построения получается набор частичных спецификаций, каждая из которых соответствует определенному требованию. Разработка таких спецификаций требует меньших трудозатрат, чем построение полной спецификации системы, поскольку не тратится ресурсов на детальное описание связей между требованиями. Такая же идея лежит в основе использования базовых протоколов, используемых в данной работе как средство создания моделей.

В [8] рассматривается способ задания конечного автомата с помощью частичных спецификаций. Описывается метод, позволяющий на основе этих частичных спецификаций обнаружить недостижимые или дублированные состояния автомата. После применения этого метода становится возможным упростить конечный автомат и обнаружить возможные ошибки в его описании.

С помощью частичных спецификаций конечного автомата в [9] решается задача получения тестов, проверяющих интересующую область функциональности автомата. Традиционно сложно достижимые состояния автомата являются проблемным участком тестирования. В статье показано, как с помощью частичных спецификаций протестировать сложно достижимые состояния с необходимым функциональным покрытием.

Интересна работа норвежских ученых [2], где для объектно-ориентированной системы предлагается создавать формальные спецификации для каждого из объектов в отдельности. Построение таких спецификаций для объекта, по сути являющихся частичной спецификаций всей системы, может проводиться независимо от других объектов. Эта ситуация актуальна, поскольку разработкой различных частей системы часто занимаются разные команды инженеров. В работе приводится методика объединения частичных спецификаций для получения полной спецификации системы, которая впоследствии может быть подвержена верификации или тестированию. Похожие принципы заложены в объектно-ориентированный язык программирования Eiffel[1], где при определении методов классов можно описывать пред- и постусловия, а также инварианты класса.

Тестирование взаимодействия системы с другими модулями на основе частичных спецификаций рассматривается в статье [10]. Для исследуемой системы строится диаграмма внешних вызовов. Затем среди различных цепочек вызовов выделяются наиболее частые и для них строятся спецификации, основанные на исходном коде системы. Благодаря построенным частичным спецификациям становится возможным сфокусироваться при верификации и тестировании на наиболее вероятных сценариях поведения системы. Однако при таком подходе цепочки вызова, которые не были включены в наиболее вероятные сценарии, вообще не подвергаются верификации, что отрицательно сказывается на качестве продукта.

Особняком стоит работа японских ученых [11], в которой описывается теория построения частичных спецификаций программных компонентов, образующих целостную систему. По результатам исследования кода функциональность компоненты разделяется на два типа: с локальной и внешней областью действия. Фрагмент функциональности считается внешним, если ответственен за взаимодействие с другими компонентами — содержит вызовы их функций, использует глобальные переменные, доступные другим модулям. В противном случае он является локальным. После построения спецификаций становится возможным проводить тестирование внешнего взаимодействия отдельно от тестирования локальной функциональности компонента. Однако в статье основной акцент сделан на теоретических аспектах разделения функциональности,

[1] http://www.Eiffel.com.

при этом практическое применение для реальных проектов не рассматривается.

2. Описание метода

Предлагаемый нами метод подразумевает следующую последовательность шагов.

Шаг № 1: идентификация интерфейса взаимодействия.

Шаг № 2: идентификация регионов взаимодействующего кода.

Шаг № 3: построение модели.

Шаг № 4: верификация и тестирование.

Общая схема метода представлена на рис. 1.

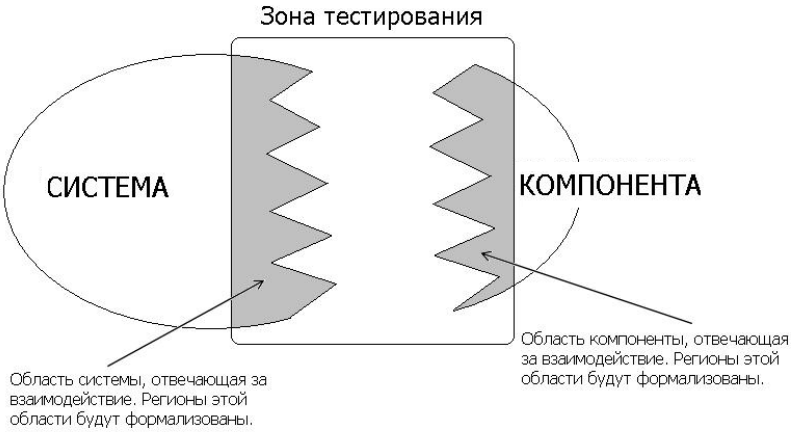

Рис. 1. Общая схема метода.

Шаг № 1. Все начинается с определения интерфейса взаимодействия. Артефакты компоненты, используемые системой, и артефакты системы, используемые компонентой, по сути, составляют интерфейс взаимодействия. Под артефактом понимается любая программная сущность, например функция, переменная или класс. Если доступен исходный код компоненты и системы, то артефакты, как правило, объявлены соответствующим образом (в отдельной секции или с использованием ключевого слова языка программи-

рования). В случае если исходный код недоступен, используются декомпиляторы. Формат записи интерфейса может быть любым, однако наряду с именем артефакта и его описанием удобно также описывать ограничения, накладываемые на артефакт, например, с помощью формального языка. Полезно зафиксировать тот факт, что функция вызывается в компоненте только с отрицательными параметрами. Добавление такой информации в ряде случаев позволяет уже на данном шаге обнаружить некоторые ошибки.

Результатом шага является построенный интерфейс взаимодействия.

Шаг № 2. Для каждого артефакта интерфейса взаимодействия определяется область кода, в которой он реализован (для системных артефактов — в системе, для компонентных артефактов — в компоненте). Такие области назовем внутренними регионами. Поиск внутреннего региона сводится либо к поиску реализации (в случае функции или класса), либо к поиску инициализации и объявления (в случае переменной). Под внешним регионом будем понимать область кода компоненты или системы, которая определяет параметры вызова внешней функции (изменения переменной или использования класса) и параметры, которые изменяются после внешнего вызова. Например, компонента в своем коде вызывает функцию системы с некоторым параметром. Внешним регионом вызова функции системы будет считаться то место в коде компоненты, которое определяет значение параметра перед вызовом этой функции, и те артефакты компоненты, которые могут измениться после вызова. Определение регионов осуществляется с помощью построенного на предыдущем шаге интерфейса взаимодействия. Поиск внешнего региона осуществляется следующим образом: если исходный код доступен, то производится поиск точек вызова функциональности системы из компоненты (или вызов функциональности компоненты из системы) в соответствии с построенным интерфейсом. Затем определяется окружение обнаруженной точки вызова. Чем более детально будет описано окружение для внешних регионов и реализация для внутренних регионов, тем более будет точна спецификация и результаты верификации. На данном шаге целесообразно использовать автоматизированные кодоанализаторы (например, KlocWork[2]). Они позволяют определить места изменения отслеживаемых параметров и тем самым обнаружить регион.

[2]KlocWork. http://www.klocwork.com.

В случае если код недоступен, используется декомпиляция кода.

Шаг № 3. На основе полученных регионов взаимодействующего кода строится модель взаимодействия. Каждый регион подвергается формализации отдельно от других регионов в терминах базовых протоколов. В результате получаем набор спецификаций, соответствующий взаимодействующим регионам системы и компоненты. По сути, в случае внешнего региона спецификация описывает состояние компоненты/системы до вызова внешней функциональности, вызов внешней функциональности и состояние компоненты/системы после внешнего вызова. В случае внутреннего региона спецификация описывает реализацию системной или компонентной функции. Таким образом, модель состоит из двух наборов спецификаций — формализованные регионы взаимодействующего кода системы и формализованные регионы кода компоненты. Результат шага — построенная модель взаимодействия в терминах базовых протоколов.

Шаг № 4. Здесь построенная модель подвергается верификации и тестированию (например, используется система VRS). Другими словами, проверяется соответствие спецификаций окружения внешних вызовов (внешние регионы) и реализации внешних артефактов (внутренние регионы). На данном шаге осуществляется контроль синтаксиса базовых протоколов, контроль связанности базовых протоколов друг с другом и генерация тестовых трасс. Полученные тестовые трассы являются готовыми сценариями, обеспечивающими полное функциональное покрытие выделенных регионов. Трассы можно использовать в качестве тестовых процедур в ручном тестировании или для генерации автоматических тестов для конкретной платформы, например, используя технологию TAT. Если в процессе верификации и тестирования были найдены критические ошибки, то исходная система подвергается переработке и процесс повторяется, начиная с первого шага. Критерий критичности ошибок в каждом проекте свой, например в случае прототипа допускаются ошибки, не мешающие демонстрации основной функциональности. После данного шага принимается решение об исправлении ошибок или готовности продукта.

3. Пример

Рассмотрим применение предлагаемого метода на примере добавления сервиса SMS сообщений в ПО мобильного устройства. В

качестве новой компоненты выступает набор классов, отвечающих за SMS-сервис. Методы этих классов используют функции, которые реализованы в системных модулях. Для отправки SMS-сообщения необходимо инициализировать адрес, по которому будет, осуществляется отправка, — открыть соединение. Ниже приведен упрощенный пример реализации открытия соединения, описанный в компоненте.

```
Class Connection {
   open();
   close();
}
function open(port) {
   if (port >=0) {
      SYS\_OpenLowLevel(port);~//вызов функции системы
      status = OPENED;~//соединение открыто
   } else throw DataException;
}
```

Если значение порта отрицательное, то функция open завершается (бросается исключение DataException). Системная функция SYS_OpenLowLevel ответственна за низкоуровневое открытие соединения. После вызова метода SYS_OpenLowLevel соединение считается открытым (status = OPENED).

Рассмотрим применение метода по шагам.

Шаг № 1. Идентификация интерфейса взаимодействия. Компонента использует системную функцию SYS_OpenLowLevel. Ввиду архитектурных особенностей система не использует компонентные функции, поэтому интерфейс взаимодействия ограничивается одной функцией.

Шаг № 2. Идентификация регионов взаимодействующего кода. Внешним регионом для SYS_OpenLowLevel является область кода функции check, где отфильтровываются не прошедшие проверку значения номера порта. Внутренним регионом является область кода в системе, где эта функция реализована. Ниже представлена реализация данной функции:

```
function SYS\_OpenLowLevel(port) {
if (port <= 0) TERMINATE\_PROGRAM; else SYS\_HWOPEN;
}
```

Шаг № 3. Построение модели. Для формализации внешнего региона необходимо рассмотреть только случай неотрицательного значения порта (в случае отрицательного значения вызов внешней функции не производится). На рис. 2 представлена спецификация на языке MSC, описывающая внешний регион.

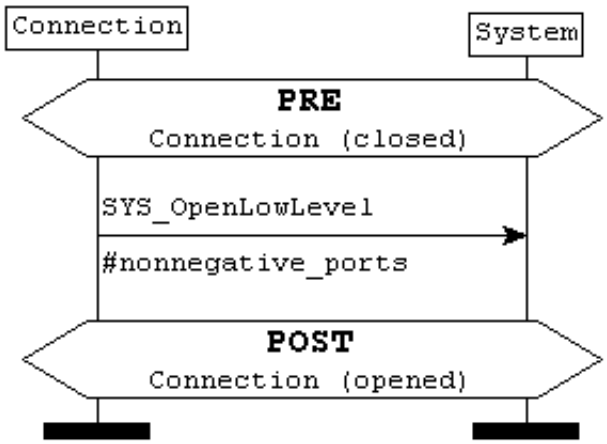

Рис. 2. Внешний регион.

Рассмотрим спецификацию детально. Итак, на рис. 2. взаимодействуют две сущности — Connection (соответствует классу Connection) и System (система). Предусловие требует (обозначается словом «PRE»), чтобы соединение было в состоянии «закрыто». Постусловие (POST) отражает состояние соединения после вызова SYS_OpenLowLevel. И в соответствии с реализацией соединение должно открыться (STATUS = OPENED). Сигнал SYS_OpenLowLevel соответствует вызову функции SYS_OpenLowLevel. В качестве параметра сигнала выступает макрос # nonnegative_ports, который принимает множество неотрицательных значений.

При формализации внутреннего региона системной функции SYS_OpenLowLevel интересно рассмотреть случай, когда порт равен нулю. Спецификация для этого случая представлена на рис. 3.

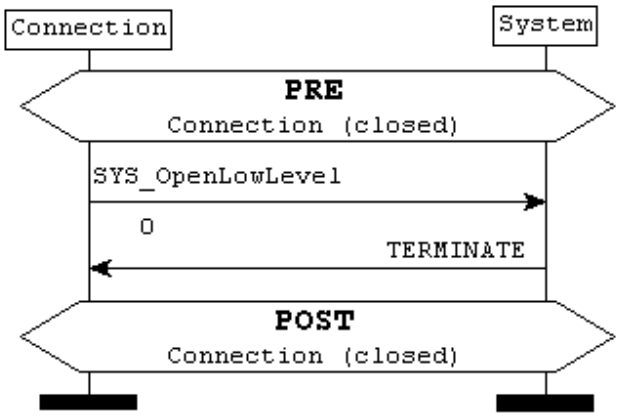

Рис. 3. Внутренний регион.

На текущем шаге (см. рис. 2 и рис. 3) можно заметить ошибку, которую с легкостью нашла бы верификационная система для построенной модели. Когда порт нулевой, компонента вызывает системную функцию SYS_OpenLowLevel, которая терминирует выполнение программы при нулевом значении порта. Данную ошибку можно было бы избежать, добавив проверку на ноль в компоненту (вместо «if (port >= 0)» написать «if (port > 0)»).

Шаг № 4. Верификация и тестирование. после построения спецификаций относится к технической части метода и не представляет практического интереса после того, как спецификации построены.

Заключение

В рамках данной работы была предложена методика тестирования взаимодействия компоненты и системы. Кроме того, метод также позволяет протестировать взаимодействие нескольких компонент или нескольких систем. Предлагаемый подход автоматизирует и упрощает тестовый процесс и гарантирует полное покрытие функциональности при генерировании тестов. Данная методика была апробирована на нескольких проектах. Предварительные

оценки трудоемкости, на примерах проектов среднего размера (около 100 требований), подтвердили его эффективность: цикл тестирования был сокращен на 2 человеко-месяца. В настоящее время метод применяется в больших индустриальных проектах. В дальнейшем планируется выработать стратегии поисков регионов, автоматизировать этапы метода для различных языков программирования, определить формат и способы составления словарей внешних вызовов для компоненты и системы. Формализация словарей упростит многие шаги метода и позволит проводить верификацию уже на первых этапах. После разработки указанных улучшений станет возможным практически полностью автоматизировать процесс интеграционного тестирования, обеспечивая необходимое тестовое покрытие.

Список литературы

[1] *Easterbrook S., Callahan J.* Formal Methods for Verification and Validation of partial specifications: A Case Study // Virginia University symposium. Vol. 1. 1997. P. 26–37.

[2] *Johnsen E. B., Owe O.* Composition and Refinement for Partial Object Specifications // Parallel and Distributed Processing Symposium, 2002. P. 210–217.

[3] *Hendrix J., Clavel M., Meseguer J.* A Sufficient Completeness Reasoning Tool for Partial Specifications // Proceedings of the 16h International Conference on Rewriting. LNCS. Vol. 3467. 2005. May 31. Springer. P. 165–174.

[4] *Letichevsky A. A., Kapitonova J. K.* Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications // Proceedings of International Workshop, WITUL. 2004. P. 30–38.

[5] *Baranov S. N., Kotlyarov V. P., Letichevsky A. A.* The technology of automation verification and testing in industrial projects // IEEE Russia Northwest Section, The International Scientific Conference «110 Anniversary of Radio Invention» Saint Petersburg, 2005. P. 12–34.

[6] *Kotlyarov V. P., Golubev A. A., Karpov A. N.* Testing Automation for system core kJava applications // Consumer Electronics, 2006. P. 1–4.

[7] *Falcone Y., Fernandez J. C., Mounier L. et al.* A Compositional Testing Framework Driven by Partial Specifications // TestCom FATES, 2007. P. 107–122.

[8] *Petrenko A., Yevtushenko N.* Testing from Partial Deterministic FSM Specifications // IEEE Trans. Computers. Vol. 54. N 9. 2005. P. 1154–1165.

[9] *Petrenko A., Yevtushenko N.* On Test Derivation from Partial Specifications // Procedings of IFIP Joint International Conference Formal Description Techniques, 2000. P. 85–102.

[10] *Acharya M., Xie T., Pei J. et al.* Mining API patterns as partial orders from source code: from usage scenarios to specifications // Proceedings of SIGSOFT Seminar, 2007. P. 25–34.

[11] *Aoshima T., Yonezaki N.* An Efficient Tableau-Based Verification Method with Partial Evaluation for Reactive System Specifications // Proceedings of EJC Conference, 2000. P. 363–374.