Автоматизированный синтез тестов для Java-программ на основе анализа программ и учета контрактов

Алефтина Андрианова, Владимир Ицыксон

Санкт-Петербургский государственный политехнический университет
aleftina.andrianova@gmail.com, vlad@ftk.spbstu.ru

Аннотация.

Обеспечение качества создаваемого программного обеспечения является одной из основных проблем программной инженерии. Одним из путей, используемых для повышения качества программ, является автоматизируемый синтез тестов. В статье предлагается технология автоматизированного создания модульных тестов, комбинирующая подходы "белого" и черного" ящика. При этом для обеспечения покрытия тестами путей программы используется информация, извлекаемая из исходного программы, а для формирования тестовых оракулов и распределения параметров тестов по области определения используются частичные спецификации, заданные в форме контрактов. Разработанный подход реализован в виде инструментального средства, анализирующего программы на языке Java, и формирующего тест-кейсы для методов классов в формате JUnit, используя COFOJA для задания контрактов. Тестирование разработанного средства на ряде тестовых примеров показало работоспособность подхода.

Ключевые слова: Автоматизированное тестирование программ, синтез тестов, контрактное программирование, анализ кода, SMT-solver

1 Введение

1.1 Автоматизация тестирования программного обеспечения

Проблема качества программного обеспечения является одной из самых острых в компьютерной индустрии. Огромные ресурсы вкладываются в различные мероприятия, направленные на повышение качества выпускаемых программных систем, так как потенциальный ущерб от сбоев программного обеспечения может быть очень велик. Одним из самых распространенных и легко реализуемых методов повышения качества является тестирование. Различные технологии тестирования активно используются практически во всех компаниях, занимающихся разработкой про-

граммного обеспечения. Однако, эффективность проведения тестирования определяется не только количеством разработанных тестов, но и их качеством. Зачастую качество тестов зависит только от профессиональных навыков тестировщика, в то время как требуется, чтобы качество создаваемых тестов больше зависело от разработанного программного обеспечения и спецификации требований. Поэтому в последнее время активно развивается направление программной инженерии, связанное с автоматизацией синтеза тестов на основе различных проектных артефактов. В качестве таких артефактов в разных исследованиях используются спецификации требований, аннотации, контракты, исходный код программного обеспечения и т.п.

При формировании тестов перед разработчиками встают следующие основные задачи:

- обеспечение максимального покрытия тестами программы, при этом обычно рассматривается один следующих критериев покрытия: покрытие операторов, покрытие ветвлений или покрытие путей;
- формирование тестов, покрывающих максимально возможный диапазон входных значений функций и методов;
- проверка максимально возможного числа требований, предъявляемых к программе.

В данной статье описывается разработанный авторами подход, целями которого было решение перечисленных выше задач. Подход основан на интеграции двух методов тестирования: структурного, базирующегося на анализе исходного кода программы, и функционального, использующего спецификации. Разработаны методы автоматизации создания тестов, учитывающие как внутреннее устройство программы, так и функциональные требования к ней. Для построения модели программы и извлечения трасс выполнения используются методы статического анализа. В качестве спецификаций требований используются контракты, позволяющие описывать требования, предъявляемые к отдельным методам и классам. Совместный анализ извлеченных трасс и контрактов позволяет синтезировать комплекты тестов, обеспечивающих покрытие путей программы. Подход реализован в виде инструментального средства генерации тестов на основе анализа Java-программ и системы контрактов CoFoJa[1].

1.2 Результаты

В данной работе получены следующие теоретические и практические результаты:

- разработаны принципы генерации тестов, форсирующих прохождения выбранной трассы программы;
- разработаны методы комплексирования синтеза функционального и структурного тестирования, основанные на объединении анализа кода и контрактов;

- предложены методы формирования множественных тестов для обеспечения покрытия всей области определения;
- разработано инструментальное средство генерации тестов для Java-программ.

1.3 Структура статьи

Оставшаяся часть работы организована следующим образом. Второй раздел содержит описание разработанного авторами подхода к генерации тестов. Подробно рассматривается построение модели программы, извлечение трасс исполнения и формирование системы утверждений для SMT-солверов, а также генерация множественных тестов. Третий раздел посвящен описанию созданного прототипа генератора тестов. В четвертом разделе приводится информация об апробации подхода. Пятый раздел содержит обзор текущего состояния дел в области автоматизированного синтеза тестов на основе анализа программы и учета контрактов. В заключении подводится итог проведенной работы, формулируются направления дальнейших исследований.

2 Предлагаемый подход

В данной работе предлагается подход к автоматизации создания модульных тестов, обеспечивающий покрытие путей программы. Далее в этой работе не умаляя общности мы будем рассматривать создание тестов для одного метода какого-либо класса. Для обеспечения покрытия путей метода необходимо для каждой возможной трассы исполнения программы сгенерировать такой набор значений аргументов метода, который форсирует прохождение программой этой трассы. То есть необходимо так подобрать значения аргументов, чтобы при каждом возможном ветвлении выбиралась требуемая ветвь программы. Построение для заданных значений аргументов соответствующих трасс исполнения является прямой задачей. Вычисление же требуемых значений аргументов метода для удовлетворения заданной трассы - это обратная задача. В общем случае обратная задача - сложная научная NP-полная проблема, требующая решения сложных систем уравнений. В данной статье описывается способ решения этой задачи с определенными ограничениями, позволяющий вычислить требуемые значения аргументов за приемлемое время. Если ограничиться формированием модульных тестов только на основе критерия покрытия, то полученные тесты никак не будут учитывать требования, предъявляемые ко всей программе вообще или к конкретному методу в частности. Чтобы использовать информацию о требованиях, описанный способ формирования тестов на основе анализа программы может быть расширен с помощью учета спецификаций. Спецификация (или частичная спецификация) может быть задана с помощью контрактов [2]. Анализ предусловий метода и инвариантов класса может сузить множество возможных

значений параметров тестов, исключив из них не соответствующие контракту. Постусловия могут использоваться для формирования заготовки тестовых оракулов, упрощая тем самым работу разработчиков тестов.

Предлагаемый авторами подход подразумевает выполнение следующих шагов:

1. формирование модели программы;
2. формирование списка путей выполнения метода;
3. преобразование каждого пути в цепочку утверждений, верных для этого пути;
4. дополнение цепочки утверждений составляющими контрактов: предусловиями метода и инвариантами класса;
5. разрешение системы утверждений относительно аргументов метода;
6. формирование экземпляра/экземпляров теста на основе решения системы утверждений;
7. формирование тестового оракула на основе постусловий метода и инвариантов класса.

Рассмотрим перечисленные шаги подробнее.

2.1 Модель анализируемой программы

Для извлечения отдельных путей выполнения программы необходимо использовать такую модель, которая с одной стороны содержит все необходимые данные для построения трасс, а с другой стороны является довольно компактной. Так как требуется извлекать информацию о динамике программы, то в качестве модели не могут использоваться структурные модели, (например, абстрактное синтаксическое дерево), не содержащие информации о последовательности выполнения операторов, использование же гибридных моделей (например, абстрактный семантический граф) неоправданно, так как такие модели слишком избыточны. Среди поведенческих моделей программ для решения задачи извлечения путей наиболее подходят модели, основанные на потоке управления, такие как граф потока управления (Control Flow Graph, CFG) или модель статического однократного присваивания (Single Static Assignment, SSA). В данной работе на фазе построения мы ограничиваемся построением классического графа потока управления, элементы же однократного статического присваивания используются позже на этапе преобразования путей в цепочки утверждений.

2.2 Извлечение трасс исполнения

Извлечение трасс исполнения осуществляется с помощью анализа графа потока управления. Для этого применяется рекурсивный алгоритм поиска, выявляющий все возможные уникальные пути от начальной точки к конечной. При этом по-

разному обрабатываются узлы, соответствующие вычислительным операторам и операторам ветвления. Для вычислительных узлов в список элементов пути добавляются операторы, соответствующие этим узлам. Для узлов ветвления вида "if (condition) ..." в список добавляется условие "condition" или "~condition" в зависимости от выбранной при обходе графа ветви условного условного оператора. Таким образом, для каждого пути графа будет собран список операторов и условий, которые должны выполниться для того, чтобы данный путь был пройден.

2.3 Формирование системы утверждений

Для того, чтобы по построенному пути вычислить подходящие значения аргументов, форсирующих выполнение этого пути, необходимо решить обратную задачу. Для этого требуется преобразовать полученный путь в систему уравнений и разрешить ее относительно аргументов метода. Для этого необходимо для каждого накопленного пути проделать следующие операции:

- преобразовать путь в форму однократного статического присваивания (SSA);
- все присваивания преобразовать в утверждения "равенства";
- все условия преобразовать в соответствующие утверждения.

Для того, чтобы построить такую систему уравнений необходимо каждой конструкции сохраненной трассы поставить в соответствие алгебраическое уравнение. Для этого все присваивания преобразуются в алгебраические равенства, а сохраненные условия веток ветвления в логические утверждения. Кроме того, необходимо перейти от переменных языка программирования, утверждения над которыми истинны только в определенном контексте выполнения, к алгебраическим переменным, истинность утверждений над которыми не меняется во времени.

Пример простой программы:

```
x = 10;
y = 20;
if (z > 5) {
  x = x + y;
}
```

Соответствующая система утверждений для пути, проходящего через ветку “then”:

```
1: x=10
2: y=20
3: z>5
4: x=x+y
```

Очевидно, что такая система утверждений неразрешима, так как с одной стороны x=10 (строка 1), а с другой стороны x=30 (строка 4). Такая коллизия обусловлена кардинальными семантическими отличиями переменных императивных языков программирования от переменных, используемых в логиках. Для разрешения этой коллизии используется известный подход, применяемый при построении модели SSA, основанный на введении дополнительных алгебраических переменных, отражающих состояния переменных языка программирования после выполнения присваивания. Такие переменные называются версиями исходной переменной, а сама операция - введением версионирования. Таким образом, после введения версионирования каждой верисонированной переменной в системе уравнений значение будет присвоено только один раз. Для приведенного примера система уравнений будет следующей:

1: $x_1=10$
2: $y_1=20$
3: $z_1>5$
4: $x_2=x_1+y_1$

Полученная таким образом система становится алгебраически разрешимой. Она относится к классу Satisfiability Modulo Theories (SMT) [3]. Как следствие, она может быть разрешена с помощью какого-либо SMT-солвера. Для этого необходимо:

- преобразовать систему утверждений в формат, являющийся входным для SMT-солвера;
- запустить SMT-солвер для решения системы уравнений;
- проинтерпретировать результаты работы солвера.

В случае успеха из результатов извлекаются значения аргументов метода, форсирующие выполнение указанного пути. Если солвер не смог разрешить систему уравнений, то это свидетельствует либо о наличии в программе “мертвого” кода, либо о некорректности допущений, сделанных при построении модели программы, либо о недостаточной мощности математического аппарата логики первого порядка, либо о недостаточной мощности самого солвера. Во всех случаях будет принято решение о невозможности создания теста для покрытия данного пути.

2.4 Расширение системы утверждений с помощью контрактов

Построенные тесты обеспечивают прохождение программой соответствующих трасс исполнения, при этом значения аргументов методов расположены в области допустимых значений произвольно, в соответствии с правилами вывода решений выбранного SMT-солвера [3]. То есть сгененрированные значения аргументов могут нарушать контракты методов, определяющиеся с помощью предусловий, постусло-

вий и инвариантов [2]. Конкретно в данном случае могут нарушаться предусловия методов или инварианты классов. Решение этой проблемы заключается в расширении построенной системы утверждений с помощью инвариантов классов и предусловий методов. Так как контракты уже задаются в форме утверждений логики первого порядка, то преобразование их в утверждения для SMT-солвера не представляет никакого труда.

В результате после решения расширенной системы утверждений будут находиться аргументы метода, удовлетворяющие контракту анализируемого метода.

2.5 Генерация тестовых оракулов

Для генерации полноценной системы тестов кроме значений аргументов методов необходимо также формировать тестовые оракулы, проверяющие корректность выполнения тестов. Постусловия, используемые в контрактном программировании, являются частью спецификаций методов и задают свойства, гарантируемые методом после его завершения в случае выполнения предусловий. В данной работе мы предлагаем генерировать основу тестовых оракулов на базе постусловий методов и инвариантов классов, как это сделано в работе [4].

Заданные в форме логических утверждений постусловия и инварианты транслируются в эквивалентные конструкции на целевом языке программирования и интегрируются в созданные тесты. При этом у тестировщика остается возможность расширить сгенерированные автоматически оракулы дополнительными проверками.

2.6 Формирование множественных тестов

Представленная технология формирования тестов обеспечивает генерацию по одному набору тестов для каждого класса эквивалентности. Этого достаточно для того, чтобы по одному разу протестировать все пути в графе потока управления. Однако на практике часто имеет смысл иметь несколько тестов для каждого класса эквивалентности, так как необходимо проверить не только сам факт прохождения программы по заданной трассе, но и другие свойства. Например, часто требуется проверять корректность работы программы на граничных значениях интервалов, правильность отработки начала и конца цикла и т.п.

Для этого необходимо для каждого класса эквивалентности генерировать множество тестов, обеспечивающих определенное покрытие области допустимых значений аргументов в соответствии с выбранной эвристикой.

В данной работе реализован метод формирования множественных тестов, основанный на последовательной генерации новых тестов для классов эквивалентности за счет вычисления области допустимых значений для аргументов метода. Исходя из количества тестов, которые должны быть сгененированы для каждого класса

эквивалентности (задано пользователем), строится равномерное покрытие n-мерного пространства аргументов функции, на основе ограничений области допустимых значений, заданных в контрактах. Для каждого набора параметров генерируется свой модульный тест в результирующем тестовом наборе. Алгоритм формирования равномерного покрытия пространства аргументов с ограничениями имеет итеративный характер, а его подробное описание выходит за рамки настоящей статьи.

В дальнейшем авторы предполагают использовать более сложные эвристики для формирования тестовых наборов, более чувствительных к стандартным ошибкам, встречающихся в программах.

3 Описание созданного прототипа

Разработанный подход был реализован в виде инструментального средства генерации тестов для языка Java. В качестве системы задания контрактов используется CoFoJa[1]. В качестве SMT-солвера применяется солвер Z3 [3], разработанный Microsoft Research, тесты генерируются в формате JUnit. Общая архитектура системы вместе с артефактами изображена на рис.1. Блоками белого цвета изображены модули, разработанные авторами, серым цветом отмечены сторонние компоненты.

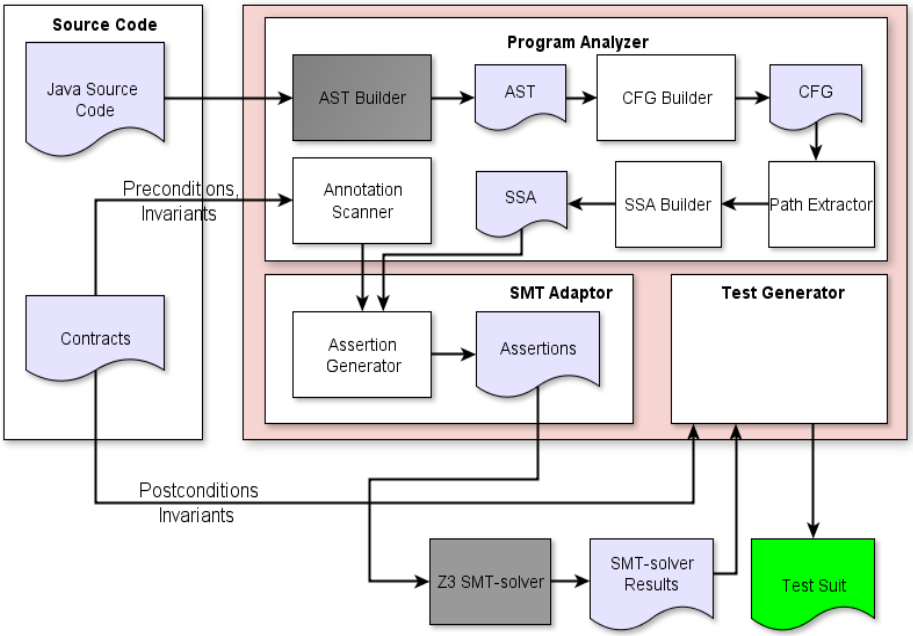

Рис. 1. Общая схема системы генерации тестов

На вход системе передается файл, содержащий исходные коды исследуемой программы на языке Java. Построитель AST формирует абстрактное синтаксическое дерево, которое является входом для построителя графа потока управления (CFG Builder). Абстрактное синтаксическое дерево анализируется и на основе семантики языка Java строится граф потока управления. Извлечение трасс исполнения (Path Extractor) осуществляется с помощью анализа графа потока управления. Полученные пути преобразуются к форме однократного статического присваивания построителем SSA (SSA Builder). Приведение пути в форму SSA с версионированием позволяет рассматривать его как систему алгебраических уравнений. Полученная система утверждений в совокупности с инвариантами класса и предусловиями метода преобразуется к формату SMT-LIB для внешнего SMT-солвера Z3 и передается ему для решения. Генерация тестов осуществляется модулем Test Generator, который использует результаты работы солвера, при этом генерация тестовых оракулов, проверяющих корректность выполнения тестов, осуществляется на основе анализа постусловий метода и инвариантов класса.

Результатом работы системы являются синтезированные модульные тесты, обеспечивающие определенный заданный уровень покрытия исходного кода.

4 Экспериментальные исследования

Было проведено тестирование разработанного инструментального средства на тестовом наборе программ. Набор состоял из искусственных тестов и нескольких реальных программ, содержащих различные конструкции языка Java: составные выражения, простые и сложные ветвления, циклы и т.п. Искусственные тесты были призваны проверить корректность функционирования разных возможностей, заложенных в разработанном анализаторе. Реальные Java-программы использовались для проверки анализатора в реальных условиях функционирования.

На всех искусственных и реальных примерах были получены результаты, соответствующие ожидаемым, а сгенерированные тесты действительно обеспечивали проверку всех путей исследуемых программ. Эксперименты с генерацией множественных тестов также показали работоспособность подхода.

5 Обзор существующих подходов в области синтеза тестов

Различные аспекты автоматизации тестирования программного обеспечения и подходы к организации статического и динамического анализа являются популярными направлениями исследований. Существует целый ряд коммерческих и свободно распространяемых средств автоматизации тестирования, фреймворков для органи-

зации разработки, систем для синтеза тестов. Многие из них используют в качестве основы подход синтетического структурного тестирования. В этом случае после первого случайно выбранного теста остальные тесты генерируются автоматически так, чтобы обеспечить покрытие еще не покрытых ранее элементов кода. Для выбора подходящих тестовых данных используются решатели, учитывающие символическую информацию об ограничениях на данные, отделяющие прошедшие тесты от еще не покрытого кода, а для построения нужных последовательностей воздействий — случайная генерация, направляемая как этой же символической информацией, так и некоторыми эвристическими абстракциями, уменьшающими пространство состояний проверяемой системы. В рамках этого подхода интегрируются статический анализ кода, символическое исполнение, структурное тестирование и дедуктивный анализ, выполняемый решателями. Одними из представителей этого класса являются Microsoft Pex and Moles, два исследовательских проекта, направленных на повышение производительности тестировщиков и качества кода.

Microsoft Pex [5,6] – набор инструментов, выполняющих анализ тестируемого кода и подбирающий параметры, которые позволяют покрыть максимально возможный объем кода. Microsoft Pex работает в петле обратной связи: код выполняется несколько раз, и производится анализ поведения программы на уровне промежуточного языка .NET CIL (Common Intermediate Language) с помощью мониторинга потока управления и потока данных. После каждого запуска Pex выбирает ветвь, которая не была покрыта ранее, создает систему ограничений, которая описывает, как достичь этой ветви, использует решатель Z3 для определения новых значений входных данных, которые удовлетворяют ограничениям, если таковые существуют. Код выполняется снова с новым набором входных значений, и процесс повторяется. В результате работы создается отчет, который можно проанализировать вручную, и набор unit-тестов для дальнейшего использования. Microsoft Moles [6,7] – фреймворк, также разработанный в лаборатории Microsoft Research и позволяющий разработчикам создавать тестовые заглушки и избегать непосредственного вызовов методов в .NET. Moles расширяет Pex за счет формирования тестового окружения и гибкого манипулирования драйверами и заглушками. Существует ряд ограничений при использовании указанных инструментов. Во-первых, невозможно обнаруживать ошибки, которые вносятся при параллельном выполнении задач, во-вторых, указанные средства не могут справиться с недетерминизмом. Это приводит к разным результатам при каждом запуске Pex. В-третьих, Pex и Moles не могут быть использованы для анализа кода, который не работает под управлением .NET. Множество поддерживаемых языков ограничено только языком C#. Возможность использования внешнего солвера, отличного от Z3, также не предусмотрена.

Еще один подход к автоматизации модульного тестирования был предложен C. Engel и R Hähnle. В работе [8] рассматривается метод автоматической генерации тестов для JAVA CARD, базирующийся на формальной проверке выполнения тестируемого кода. В его основе лежат два подхода к тестированию: по методу “бело-

го” и “черного” ящика. Автономные модульные тесты в формате JUnit генерируются автоматически. VBT (verification-based test generation) использует полную информацию, содержащуюся в формальной спецификации и лежащую в основе реализацию тестируемого кода (implementation under test, IUT). В качестве языка описания спецификации используется Java Modeling Language (JML). При этом полная функциональная спецификация тестируемого кода не требуется, поскольку генерация тестов реализуется на основе символического исполнения, а заданные в пред- и постусловиях ограничения требуются только для синтеза тестовых оракулов.

Система Kiasan [9,13] представляет собой механизм символического исполнения для последовательных Java программ, основанный на фреймворке Bogor, использующем технологию проверки модели (model checking). Предложенный подход был позднее расширен в фреймворке KUnit [9], который автоматически генерирует тесты для контрактно-аннотированых методов и отображает множество объектов (heap objects), получаемых и возвращаемых методами, что может быть использовано разработчиками для анализа сложных методов и диагностики причин ошибок в программе. Фреймворк KUnit использует тестирование по методу “черного ящика” с помощью генерации входных значений на основе символического исполнения, постусловия используются в качестве тестовых оракулов. При наличии формальной спецификации подход, используемый в KUnit, может реализовывать такие техники тестирования, как: разделение на классы эквивалентности, анализ граничных значений. В общем случае традиционное тестирование обычно не основывается на формальной спецификации, в то время, как KUnit требует формального описания требований.

Инструмент Unit Meister [10], разработанный компанией Microsoft, использует символическое исполнение для анализа трасс программы. Подход во многом схож с Kiasan, однако существует ряд отличий. Unit Meister включает функциональные символы для композиционной проверки, в то время, как Kisan допускает только исходные символы, а композиционная проверка осуществляется с помощью спецификаций. В то время как Kiasan является полностью автоматическим, Unit Meister, зависимый от решателя системы ограничений, требует от пользователя указания области определения параметров.

Symstra [11,12] – еще одно средство символического исполнения для создания минимальной последовательности вызовов публичных методов для проверки класса. Symstra использует примитивные символические значения, конкретные структуры множеств и предполагаемые состояния для генерации неизоморфных конечных состояний. Для генерации подмножества возможных предварительных состояний в Symstra применяются последовательности вызовов публичных методов. В разработанном прототипе предварительные состояния задаются предусловиями и инвариантами. Пользователь может изменять предусловия и инварианты, более точно определяя их значения.

Одним из инструментов, ориентированных на использование контрактных спецификаций в процессе создания тестов, является AutoTest framework [4]. Он автоматизирует процесс тестирования программного обеспечения, опираясь на программы, содержащие инструменты их собственной проверки, в форме контрактов для классов и отдельных методов. Предусловия и инварианты позволяют ограничить множество входных данных для тестирования, постусловия преобразуются в тестовые оракулы. Основная особенность AutoTest заключается в способе генерации тестов. Тестовые последовательность формируются случайно, опираясь только на предсуловия методов и инварианты классов. Успешность прохождения тестов анализируется в тестовых оракулах. Отдельно стоит отметить механизм Test Extraction, который автоматически создает тесты по результатам отказов программы. Основными ограничениями подхода являются отсутствие гарантий покрытия путей (из-за сугубо случайной генерации тестов), а также поддержка только языка Eiffel и среды EiffelStudio.

Все рассмотренные подходы подтверждают эффективность интеграции различных методологий процесса тестирования на практике. Тем не менее, несмотря на достигнутые успехи, каждый из имеющихся подходов использует лишь часть имеющегося потенциала, ограничен анализом программ, написанных на каком-то определенном языке программирования, и не предоставляет единой интеграционной платформы для всего многообразия различных техник верификации ПО. Кроме того, большинство из рассмотренных подходов являются академическими и неприменимы для анализа сложных программных систем. [14,15].

6 Заключение

В результате проведенного исследования разработан подход к синтезу тестов для языка Java, обеспечивающих покрытие трасс исполнения методов. Разработанные методы с помощью применения SMT-солвера генерируют параметры модульных тестов, форсирующие реализацию заданных трасс исполнения. Учет контрактов методов позволяет с одной стороны с помощью постусловий частично автоматизировать синтез тестовых оракулов, а с другой с помощью предусловий ограничивать генерацию параметров тестов. Применение разработанного на основе подхода прототипа на наборе тестовых примеров показало полную работоспособность предложенных методов. Основными направлениями развития теоретических и практических исследований являются:

- разработка новых алгоритмов генерации множественных тестов, более эффективно распределяющих значения переменных по области определения;
- совершенствование анализатора (прототипа) с целью обеспечения анализа более широкого класса Java-программ;

- расширение функциональных возможностей и реализация подхода генерации модульных тестов для программ, написанных на других языках программирования.

Литература

1. Contracts for Java. https://code.google.com/p/cofoja/
2. Meyer, B.: Design by Contract, in Advances in Object-Oriented Software Engineering, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, pp. 1–50
3. Z3 An Efficient SMT Solver. http://z3.codeplex.com/
4. B. Meyer, I. Ciupa, A. Leitner, A. Fiva, Yi Wei, E. Stapf: Programs that Test Themselves, IEEE Computer, vol. 42, no. 9, pages 46-55, September 2009
5. Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In Proceedings of the 2nd international conference on Tests and proofs, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag
6. Pex and Moles. http://research.microsoft.com/en-us/projects/pex/
7. Unit Testing with Microsoft Moles. http://research.microsoft.com/en-us/projects/pex/molestutorial.pdf
8. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y. (ed.) Proceedings, Testing and Proofs, Zürich, Switzerland. LNCS, Springer,Heidelberg, 2007
9. Deng, X., Robby, Hatcliff, J.: Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques, Washington, 2007
10. Nikolai Tillmann, and Wolfram Schulte. Parameterized unit tests with unit meister. ESEC/SIGSOFT FSE, page 241-244. ACM, 2005
11. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In TACAS '05: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, volume 3440 of LNCS, pages 365–381. Springer-Verlag, Apr. 2005
12. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. http://people.engr.ncsu.edu/txie/publications/tacas05.pdf
13. Robby: Bogor/Kiasan: Combining symbolic execution, model checking, and theorem proving. Presentation at European Science Foundation Exploratory Workshop on Challenges in Program Verification, University of Nijmegen (October 2006)
14. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In ECOOP '03: European Conference on Object-Oriented Programming, pages 431–456, 2003
15. T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In ICSE '02: Proceedings of the 24th International Conference on Software Engineering, pages 478–488, New York, NY, USA, 2002. ACM Press