электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.4'23

Обзор подходов и методов к генерации регрессионных модульных тестов

***Дерягин Д.А.**, магистр*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии»

Научный руководитель: Ломовской И.В., старший преподаватель
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
irudakov@bmstu.ru

Введение

Тестирование является трудоемкой и ресурсоемкой задачей. Примерно половина времени разработки всего проекта, как правило, тратиться на тестирование (Myers, Sandler, & Badget, 2011). В связи с этим, многие исследования посвящены теме автоматизации процессов создания и исполнения тестов. Важно отметить, что успешное тестирование не может гарантировать отсутствие ошибок в программе. Тем не менее, большой набор тестов может существенно помочь в обнаружении ошибок в логике работы программы и ее реализации, и тем самым повысить вероятность того, что программа будет успешно решать поставленные задачи.

Существуют различные подходы, где при тестировании проводится проверка соответствия спецификации системы ее реализации. Спецификация, заданная на естественном языке, пригодна только для ручного тестирования, поэтому предлагается записывать ее на формальном языке. На практике очень часто встречается ситуация, когда ПО уже разработано, а спецификация утеряна или не соответствует конечному продукту. Отсутствие спецификации не мешает системе работать должным образом, но в определенный момент проявляется неизвестная ранее ошибка или возникает новое требование. Внесение исправлений в программу на данном этапе очень трудоемко, более того, исправление одной ошибки, может привести к появлению другой. В такой ситуации наличие автоматических тестов очень желательно, однако их может и не быть, ведь система могла быть протестирована вручную. Более того, в некоторых случаях ограничиваются только системным тестированием. Написать автоматические модульные тесты не представляется возможным, поскольку нет четкого представления о том, как

должны функционировать отдельные модули. В такой ситуации рекомендуется писать именно модульные тесты, прежде чем приступать к исправлениям (Feathers, 2004).

Возникает проблема автоматического создания тестов, которые помогли бы зафиксировать текущее поведение программы. В данной работе проводится обзор существующих методов и подходов к решению задачи автоматизации тестирования. Рассматриваются их сильные и слабые стороны, области применения, возможность использования применительно к поставленной задаче.

Описание проблемы генерации тестов

На данный момент уже существуют некоторые инструменты, позволяющие автоматически создавать тесты. Тем не менее, подобные подходы требуют наличия определенной метаинформации, а именно спецификации к тестируемому модулю. Это обусловлено тем, что тесты разрабатываются именно на основе этой спецификации.

Спецификация – законченное описание поведения программы, которую требуется разработать (Спецификация).

Использование методов, позволяющих проводить проверку соответствия спецификации системы ее реализации, в некоторых случаях не всегда приемлемо. Эффективность подобных подходов напрямую зависит от полноты спецификации. Процесс ее описания на формальном языке является достаточно трудоемкой задачей, что может свести на нет преимущества автоматической генерации тестов.

Более того, спецификация может быть утеряна или не соответствовать конечному продукту. Такое может быть в случае недобросовестности компании-разработчика, которая не соблюдает все этапы жизненного цикла ПО. Также в случае недостаточного контроля качества может возникнуть ситуация, когда программисты не пишут модульные тесты и документацию к разрабатываемым классам (в случае объектно-ориентированного подхода). В ряде случаев компании ограничиваются лишь системным тестированием, поскольку затраты на ручное написание модульных тестов достаточно велики. Хотя автоматические модульные тесты позволяют обнаруживать ошибки на самых ранних стадиях.

Порой такие системы называют унаследованными (Feathers, 2004). Но отсутствие спецификации не мешает системе работать должным образом. Возможно, она была тщательно протестирована вручную и проработала у заказчика продолжительно время. Иногда наступает такой момент, когда в разработанное ПО необходимо внести изменения. Как правило, это возникает в одном из трех случаев:

- проявляется не известная ранее ошибка;
- возникает новое требование;
- требуется проведение рефакторинга.

Первые два случая достаточно очевидны, в то время как решение о проведении рефакторинга, как правило, принимается, когда планируется большой объем изменений в системе. Мартин Фаулер рекомендует предварительно покрывать код тестами, прежде, чем приступать к этой процедуре (Фаулер & Бек, 2008).

Внесение изменений в ПО на последних этапах очень трудоемко, более того, оно может привести к появлению ошибок. В такой ситуации наличие автоматических тестов просто необходимо, однако написать их не представляется возможным, поскольку нет четкого представления о том, как должна работать система в целом, а уж тем более отдельные модули. Иными словами, спецификация к системе отсутствует. Вместо нее существует работающая система и запрос на модификацию.

При внесении изменений мы хотим убедиться, что наши правки не добавят каких-либо побочных эффектов, то есть мы имеем дело с регрессионным тестированием, а тесты, которые необходимо сгенерировать называются регрессионными.

Таким образом, необходимо сгенерировать регрессионные модульные тесты, которые помогут некоторым образом зафиксировать текущее поведение программы. При наличии таких тестов, внесение каких-либо исправлений будет легко отслеживаться путем перезапуска этих тестов. Подобный процесс модификации модулей позволит различать отвечающие требованиям изменения от побочных эффектов.

Существующие подходы к решению задачи

В области автоматизации тестирования ПО уже существует ряд методов для генерации тестов и верификации программ. Существующие подходы можно разделить на три основные категории: статические, динамические и формальные. Для наиболее полного анализа необходимо рассмотреть возможности все трех категорий инструментов.

Статические методы

Статический анализ – это анализ, выполняемый без запуска на исполнение анализируемой программы.

Основные задачи, решаемые статическими методами, представлены на Рис. 1.

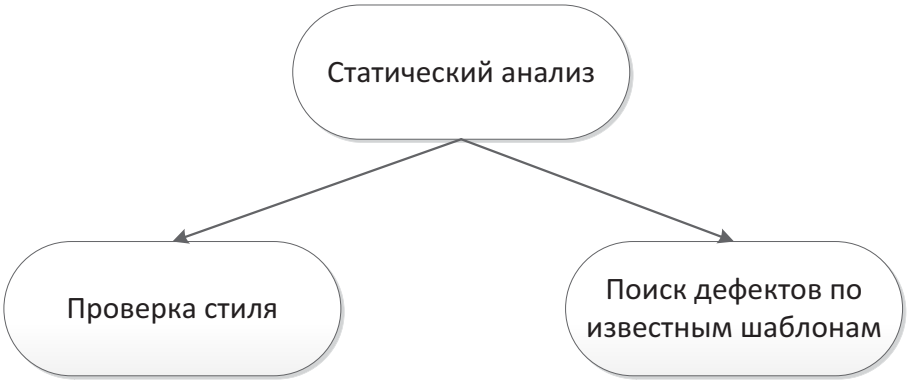

Рис. 1. Основные задачи, решаемые статическими методами

Существует ряд инструментов, позволяющих задать некий шаблон оформления кода, принятого в компании, и проверить, что написанный код соответствует этому шаблону. Те же самые инструменты, как правило, включают в себя набор заранее известных шаблонов «плохого кода», который может привести к ошибкам. Самой простой пример такого шаблона – это проверка, что переменная была инициализирована до момента ее использования. При помощи таких утилит проводят автоматическое рецензирование кода. Например, для языка Java одним из наиболее известных статических анализаторов является FindBugs (FindBugs - Find Bugs in Java Programs). В изобилии присутствуют и другие более сложные инструменты, которые позволяют доказывать некоторые свойства программы.

Несмотря на ряд преимуществ статического анализа кода, его трудно применить к генерации тестовых данных. Намного более широкие возможности для решения поставленной задачи предоставляют формальные методы и динамический анализ.

Динамические методы

Динамический анализ – это анализ программ, во время которого оценка свойств системы производится по результатам ее реальной работы.

Классификация динамических методов представлена на Рис. 2.

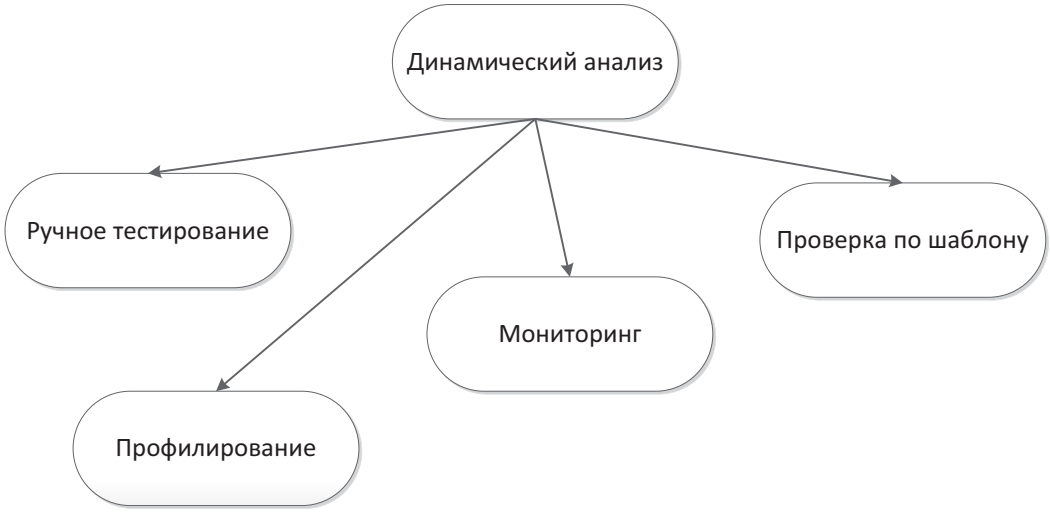

Рис. 2. Классификация методов динамического анализа

Недостатки ручного тестирования вполне очевидны. Основной проблемой являются высокие затраты на организацию этого процесса, именно поэтому этот этап разработки ПО следует автоматизировать. Как правило, ручное тестирование не относят ни к какому из перечисленных подходов, а рассматривают отдельно. Тем не менее, если все же пытаться подвести его под описанную классификацию, ручной подход относится скорее к динамическим методам, когда результат проверяется на работающей программе.

Мониторинг и профилирование преследуют вполне определенные цели. Системы мониторинга позволяют контролировать различные параметры системы, например, расход памяти, количество используемых ресурсов и т.д. Профилирование чаще всего используется для поиска проблем с производительностью. Ряд профайлеров позволяет контролировать расходуемую память и отслеживать время нахождения потоков системы в различных состояниях. Примером такого профайлера служит VisualVM (VisualVM), предназначенный для профилирования Java-программ и поставляемый вместе с JDK.

Динамические методы, также как и статические, позволяют проводить проверку по заранее заданным шаблонам. Широко известный инструмент Valgrind (Valgrind) проводит анализ, в ходе которого выявляются утечки памяти, проблемы взаимодействия потоков, попытки использования неинициализированной памяти и т.д.

Стоит отметить, что для применения динамических методов необходимо иметь работающую систему или хотя бы некоторые ее компоненты, поэтому нельзя использовать их на первых стадиях разработки. Зато при помощи динамических методов

можно контролировать характеристики работы системы в ее реальном окружении, которые не всегда возможно проверить другими методами.

Основными недостатками динамических методов верификации является необходимость наличия тестовых сценариев с подготовленным набором данных и поиск ошибок только по заранее известным шаблонам. Однако эти проблемы решаются при комбинированном подходе, использующем динамические методы вместе с формальными.

Формальные методы

Формальная верификация программ – это приемы и методы формального доказательства (или опровержения) того, что модель программной системы удовлетворяет заданной спецификации (Карпов, 2010).

На Рис. 3 представлены две основные категории, на которые можно разделить инструменты формальной верификации.

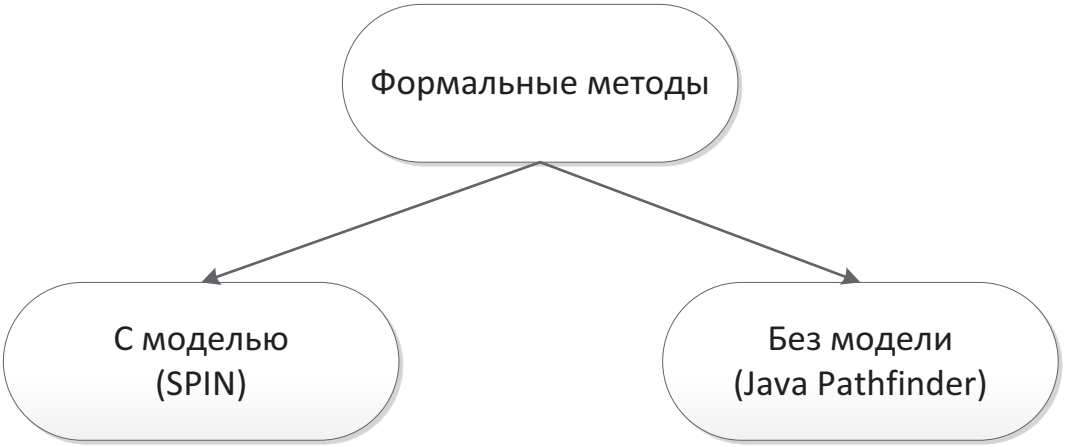

Рис. 3. Классификация формальных методов верификации

Первая категория инструментов требует наличия модели и спецификации.

«Модель – это некоторое отражение структуры и поведения системы, описываемая в терминах состояния системы, входных воздействий на нее, конечного набора состояний, потоков данных и потоков управления, возвращаемых системой результатов» (Кулямин).

Формальная спецификация представляет собой требования к поведению системы в терминах того или иного формального метода.

При тестировании на основе модели тестовые сценарии частично или целиком создаются по модели, описывающей функциональные аспекты тестируемой системы (Пелед, Грамберг, & Кларк, 2002).

Тестирование на основе модели обладает следующими преимуществами:

- тесты на основе спецификации более эффективны, так как они в большей степени проверяют функциональные требования, а не аспекты, построенные на знании реализации;
- на основе формальных спецификаций можно тесты можно создавать автоматически.

Основным недостатком тестирования на основе моделей является необходимость создания модели. Кроме того, отсутствие ошибок в модели не гарантирует отсутствие ошибок в программе, так как при построении модели могли быть допущены ошибки или не учтены особенности реализации.

На текущий момент существует достаточно инструментов, позволяющих создавать модели и генерировать тесты для проверки этих моделей. В первую очередь стоит упомянуть верификатор SPIN (Spin - Formal Verification).

Spin – система верификации, которая поддерживает разработку и анализ корректности параллельных и распределенных систем с конечным числом состояний, спецификация свойств которых представлена формулами LTL. Основная цель использования пакета Spin – это проверка корректности взаимодействующих параллельных асинхронных процессов (Карпов, 2010).

Система Spin имеет свой собственный язык Promela для описания спецификаций, который предоставляет возможность легко описывать различные схемы асинхронных взаимодействий. По заданной модели Spin позволяет выявлять основные проблемы многопоточных программ.

Существуют также инструменты, позволяющие задавать модели на широко используемых языках, таких как C# и Java. NModel (NModel) позволяет описывать модели на языке C#, строить по ним конечные автоматы и генерировать тестовые сценарии для верификаций этой модели. Также присутствует инструмент ModelJUnit (The ModelJUnit test generation tool) для описания конечных автоматов как классов на языке Java.

Такое требование как наличие модели является достаточно существенным ограничением для использования подобных методов, поэтому были разработаны инструменты, которые не требуют наличия модели для верификации программы.

Ярким представителем таких инструментов является Java Pathfinder. Сам инструмент представляет собой Java машину. На вход он принимает программы на языке Java. В отличие от обычной JVM Pathfinder исполняет все пути в графе потока управления

поданной на вход программы. В работах (Visser, Pasareanu, & Khurshid) и (Symbolic Execution) поясняется принцип работы символьного исполнения и SAT решателей для преодоления условных переходов.

Символьное выполнение – это анализ программы для определения входных значений для различных путей выполнения.

SAT Solver решает задачу выполнимости булевых формул. Если формула выполнима, то возвращает диапазон значений на которых она выполнима (Решатель).

Используя символьное выполнение и решатели, интерпретатор имеет возможность исполнения всех ветвей программы для проверки каких-либо утверждений. Также в его распоряжении имеется набор входных значений для каждого пути в графе потока управления, что позволяет генерировать входные данные и автоматические тесты по этим данным.

Согласно проведенным исследованиям (Visser, Pasareanu, & Khurshid) символьное выполнение хорошо справляется с генерацией тестовых данных. Тем не менее, данный подход не годится для создания модульных тестов. Ведь интерпретатор умеет исполнять программу, которая имеет вполне определенную точку входа. В случае модуля мы имеем дело с набором открытых методов, которые можно вызывать в любой последовательности. Вызов метода может менять состояние модуля, а значит, тестовый сценарий представляет собой последовательность вызовов методов с определенными параметрами. Таким образом, инструменты, использующие символьное выполнение хорошо подходят для системного тестирования, но не для модульного. То есть, если в системе какой-то конкретный модуль используется вполне определенным образом и такой вариант использования не приводит к ошибке, то можно считать, что система функционирует корректно. Модульные же тесты должны учитывать любые возможные варианты вызовов методов. Иными словами, они фиксирует поведение модуля при любых вариантах использования.

Институт системного программирования РАН представляет набор инструментов с общим названием UniTesK (Баранцев, Бурдонов, & Демаков, 2004) для создания тестов на основе спецификаций.

Подход, предлагаемый РАН, предписывает использовать формальные спецификации ограничений, задаваемые в виде пред и постусловий процедур и инвариантных ограничений на типы данных (Петренко, Бритвина, Грошев, Монахов, & Петренко, 2008). Инструменты позволяют задавать спецификации на различных языках общего назначения, таких как Java, C#, C. Тестовые сценарии в данном случае создаются

автоматически на основе заданной спецификации. В качестве модели используется конечный автомат, который также как и модуль имеет состояние, а переходы а автомате соответствуют вызовам операций над модулем.

Использование формальных спецификаций – это серьезных шаг в автоматизации тестирования программ. Можно сказать, что это некий инструмент декларативного описания тестовых сценариев. Применение этого подхода может существенно снизить затраты на тестирование и повысить качество разрабатываемого ПО. Однако, накладные расходы на создание спецификаций также высоки. Более того, в постановке задачи генерации модульных тестов упоминалась ситуация при которой создание спецификации попросту невозможно. А ведь весь процесс создания тестов в данном подходе упирается лишь в полноту формальной спецификации. Пример использования JavaTESK можно посмотреть в работе (JavaTESK: первое знакомство). Проделав аналогичный пример, можно оценить величину накладных расходов на создание спецификации к тестируемому классу, а также оценить зависимость между ее полнотой и достигаемым процентом покрытия кода.

Заключение

В ходе анализа предметной области была сформулирована проблема генерации регрессионных модульных тестов и рассмотрены различные инструменты, предоставляющие возможности создания тестов. В ходе обзора были выявлены некоторые недостатки рассмотренных инструментов применительно к поставленной задаче. Таким образом, был сделан вывод, что перечисленные подходы решают поставленную задачу лишь частично. Тем не менее, существующие наработки и подходы имеют свои сильные стороны, которые можно использовать при создании комбинированного подхода, полностью решающего поставленную задачу.

Список литературы

1. Myers G.J., Sandler C., Badget T. The Art of Software Testing. 3rd ed. New Jersey: Wiley, 2011. 240 p.
2. Feathers M. Working Effectively with Legacy Code. 1st ed. New Jersey: Prentice Hall, 2004. 456 p.
3. Спецификация // Википедия. Режим доступа: http://ru.wikipedia.org/wiki/%25D0%25A1%25D0%25BF%25D0%25B5%25D1%2586%25D0%25B8%25D1%2584%25D0%25B8%25D0%25BA%25D0%25B0%25D1%2586%25D0%25B8%25D1%258F (дата обращения: 15.04.2014).

4. Фаулер М., Бек К. Рефакторинг. Улучшение существующего кода. СПб: Символ-Плюс, 2008. 432 с.
5. FindBugs - Find Bugs in Java Programs. Available at: http://findbugs.sourceforge.net/, accessed 15.04.2014.
6. VisualVM. Available at: http://visualvm.java.net/, accessed 15.04.2014.
7. Valgrind. Available at: http://valgrind.org/, accessed 15.04.2014.
8. Карпов Ю. Model Checking. Верификация параллельных и распределенных программных систем. СПб: БХВ-Петербург, 2010. 552 с.
9. Кулямин В.В. Методы верификации программного обеспечения. Режим доступа: http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf (дата обращения 11.04.2014).
10. Пелед Д., Грамберг О., Кларк Э.М. Верификация моделей программ. Model Checking. Москва: МЦНМО, 2002. 416 с.
11. Spin - Formal Verification. Available at: http://spinroot.com/spin/whatispin.html, accessed 15.04.2014.
12. NModel. Available at: http://nmodel.codeplex.com, accessed 15.04.2014.
13. The ModelJUnit test generation tool. Available at: http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/, accessed 15.04.2014.
14. Visser W., Pasareanu C.S., Khurshid S. Test Input Generation with Java PathFinder. Режим доступа: https://users.ece.utexas.edu/~khurshid/papers/JPF-issta04.pdf
15. Symbolic Execution. Available at: http://javapathfinder.sourceforge.net/extensions/symbc/doc/, accessed 14.04.2014.
16. Решатель. Режим доступа: http://ru.wikipedia.org/wiki/%25D0%25A0%25D0%25B5%25D1%2588%25D0%25B0%25D1%2582%25D0%25B5%25D0%25BB%25D1%258C (дата обращения: 15.04.2014).
17. Баранцев А.В., Бурдонов И.Б., Демаков А.В. Подход UniTesK к разработке тестов: достижения и перспективы Режим доступа: http://citforum.ru/SE/testing/unitesk/ (дата обращения: 11.04.2014).
18. Петренко А., Бритвина Е., Грошев С., Монахов А., Петренко О. Тестирование на основе моделей. Режим доступа: http://www.osp.ru/os/2003/09/183388/ (дата обращения: 15.04.2014).
19. JavaTESK: первое знакомство. Режим доступа: http://www.unitesk.ru/download/tools/javatesk/JavaTESKGettingStarted2.0.rus.pdf (дата обращения: 15.04.2014).