СТРУКТУРНОЕ ТЕСТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ. ПРОБЛЕМЫ И ПУТИ РЕШЕНИЯ

Вигура А.Н., Горбатовский В.В., Ломакина Л.С.
ФГБОУ ВО Нижегородский государственный технический университет им. Р.Е. Алексеева
anton.vigura@gmail.com, vladarc@gmail.ru, llomakina@list.ru

Аннотация: Приведен аналитический обзор подходов к верификации и актуальных проблем тестирования программного обеспечения. Рассмотрены основные трудности, возникающие при структурном тестировании современных программных систем, его автоматизации, а также пути их решения.

Ключевые слова: структурное тестирование, автоматизация тестирования, полнота тестирования, генерация тестов.

Введение

В настоящее время во многих отраслях народного хозяйства прослеживается тенденция ко все большей информатизации. Информационные технологии внедряются для решения самых разных задач, программные системы применяются для оценки ситуации в той или иной отрасли и принятия решений на разном уровне, в том числе и на глобальном. Программные системы усложняются с каждым днем, причиной этого является как стремление к углублению автоматизации, так и постоянный рост объемов хранимых и обрабатываемых в мире данных.

Быстрое развитие информационных технологий в настоящее время ставит новые вызовы перед разработчиками технических и программных систем в плане обеспечения достаточного уровня их качества. В настоящем обзоре приведены существующие проблемы, встающие перед верификацией и управлением качеством программного обеспечения, и возможные пути их решения.

1 Подходы к верификации ПО

Можно определить верификацию программного обеспечения как деятельность, направленную на определение соответствия требований к программному обеспечению их реализации, а также

соответствие процесса разработки принятым правилам и стандартам. Существует несколько классов методов верификации: экспертиза (чтение и анализ кода, аудит, анализ архитектуры и т. п.), статический анализ, формальные методы (доказательство корректности, проверка моделей и т.п.) динамические методы верификации (мониторинг, тестирование), синтетические методы [1]. В большинстве случаев применяются совместно методы экспертизы (code review), статический анализ, мониторинг и тестирование. Формальные методы в силу своей сложности и дороговизны применяются в ответственных отраслях. В настоящем обзоре рассматриваются именно методы тестирования, их актуальные проблемы и возможные пути решения.

Вообще, тестирование — это анализ поведения программной системы на входных данных из некоторого набора тестов. Существуют различные критерии выбора множества тестовых воздействий. Функциональные критерии не используют информацию о внутренней структуре программы; тесты выбираются, исходя из требований на программный продукт. В этом случае программа предстает в виде черного ящика, поэтому такого рода тестирование называют также тестированием «черного ящика». Структурные критерии, напротив, предполагают выбор тестов на основе структуры программы (тестирование «белого ящика»). Функциональные и структурные критерии дополняют друг друга и на практике обычно используются совместно.

2 Актуальные проблемы тестирования ПО

В современной индустрии разработки ПО изменение требований к программным системам в течение их жизненного цикла не является чем-то необычным. Процесс разработки строится с учетом возможных изменений потребностей пользователей, появления новых технологий и изменения ситуации на рынке, при этом применяются гибкие методологии разработки (Agile), охватывающие весь жизненный цикл ПО. На каждом этапе учитывается возможность изменения требований:

- Анализ требований: требования формулируются на естественном языке, обычно нечетко — в виде пользовательских историй.
- Проектирование: в архитектуру программной системы закладывается возможность расширения функционала без существенной переработки. Все чаще применяется подход «эволюционных архитектур» [2], при котором на этапе проектирования и разработки реализуются прототипы разных модификаций программной системы, на основании испытаний которых выбирается путь дальнейшего развития проекта.
- Разработка: выполняется в цикле «разработка — статический анализ — модульные тесты — рефакторинг», таким образом, позволяя находить внесенные проблемы уже на этой стадии.
- Верификация: основана на автоматизированном тестировании, мануальные тесты выполняются при реализации нового функционала и в дальнейшем реализуются в виде автоматических, которые проводятся на регулярной основе в рамках регрессионного тестирования. На этом этапе также выполняется проверка нефункциональных требований (требования производительности, надежности, безопасности и т. п.).
- Внедрение и эксплуатация: здесь важна возможность предоставления качественной обратной связи от пользователей, поэтому еще на этапе проектирования архитектуры должны закладываться средства диагностики.

Отметим, что в реальности приведенные этапы тесно связаны, частично выполняются параллельно, и есть обратные связи от всех этапов к предыдущим. Роль верификации здесь особая — она предоставляет информацию для принятия решений на всех прочих этапах — например, при использовании подхода эволюционных архитектур результат верификации является целевой функции для принятия дальнейшего пути развития проекта. Приведем далее некоторые актуальные проблемы, встающие перед верификацией, и предполагаемые пути их решения.

2.1 Проблемы требований

В большинстве случаев (исключая разработку программных систем с повышенными требованиями к качеству в ответственных отраслях) требования к программной системе могут выставляться нечетко. Более всего это характерно для функциональных требований — например, при применении гибких методологий разработки ПО требования могут формулироваться в виде пользовательских историй, представляющих собой верхнеуровневое описание функциональности и набор т.н. приемочных тестов. Отказ от формализма здесь сознательный — требования пишутся таким образом, чтобы их можно было без дополнительных проблем согласовать с заказчиками и разработчиками. Однако при выполнении тестирования это может приводить к неоднозначности

трактовок [3], кроме того, перейти от нечетких требований к автоматической генерации и проверке тестов весьма непросто — нужно в какой-то момент выполнять их формализацию.

Гибкие методологии разработки ПО предполагают также, что требования могут изменяться в ходе разработки системы. С учетом их возможной нечеткости появляется проблема с определением множества уже созданных тестов, которые нужно обновить с учетом новых требований. Это касается как функциональных, так и структурных (например, модульных) тестов. Поскольку решение о передаче программного продукта о внедрении выполняется на основании результатов верификации, при изменении требований следует включать в план выпуска также время на обновление тестов.

Здесь встает вопрос — какую стратегию обновления выбрать. Для структурных тестов известно, какие конкретно программные модули (подпрограммы, строки кода) они покрывают, но в общем неизвестно, какой функционал они затрагивают (например, возможны модульные тесты, которые затрагивают вообще весь функционал, но при этом их не нужно обновлять в данном конкретном случае).

Возможное решение здесь следующее. Предположим, что в процессе разработки используется непрерывная интеграция (continuous integration) — тесты выполняются на периодической основе (например, каждой ночью или при изменении кодовой базы). При этом для всех тестов производится определение тестового покрытия на программной системе под тестированием — определяются все затронутые участки кода. В дальнейшем при изменении функциональных требований определяется множество затронутых функциональных тестов, а исходя из пересечения тестовых покрытий можно определить и затронутые структурные тесты тоже.

2.2 Как оценить тестовое покрытие на программной системе, передаваемой во внедрение?

Как уже было отмечено в предыдущем пункте, даже для функциональных тестов информация о протестированных участках кода полезна. Практическая проблема здесь одна — каким образом определить тестовое покрытие на той самой системе, которая тестируется и которая потом будет передаваться во внедрение. Существуют следующие варианты:

- Определение покрытия с помощью статистического профилировщика (perf и т. п.). Покрытые участки кода определяются с помощью функционала ОС (периодически по таймеру). Например, в исследовании [4] предложен метод оценки тестовых покрытий на основе данных от аппаратных счетчиков производительности. Не все покрытие может быть выявлено таким образом, а реальная точность определения покрытия сильно зависит от тестовой нагрузки.
- Определение покрытия с помощью динамического инструментирования (valgrind, paradyn и т.п.). Строго говоря, это интрузивный метод, который представляет собой модификацию программой системы во время выполнения (хотя и не требуется готовить специальную версию программой системы для подсчета покрытий). Как следствие, данный метод влияет на характеристики работы системы, на ее соответствие нефункциональным требованиям (в основном требованиям по производительности) поэтому он может быть неприменим в ряде случаев.
- Теоретически есть возможность применения аппаратных средств для точного определения тестового покрытия — например, если при компиляции расположить машинный код в памяти таким образом, чтобы ветвления располагались в разных страницах виртуальной памяти, то при соответствующей настройке аппаратуры трансляции виртуальных адресов возможно получение аппаратных прерываний при прохождении ветвлений. Однако требуется исследование на предмет влияния такого порядка сборки программной системы на ее эффективность.

Приведенные подходы либо дают неточный результат, либо могут изменять поведение тестируемой системы.

2.3 Графовые модели не отражают всех вариантов передачи управления

«Классический» управляющий граф отражает передачу управления в программе между ее элементами, такими, как отдельные операторы, подпрограммы или модули. Он может быть построен с помощью разбора исходного текста либо машинного кода. Тем не менее, возможны такие варианты передачи управления, выявить которые проблематично:

- Любой доступ к памяти может вызвать аварийное завершение программы либо переход к обработчику прерывания. Например, на определенных входных данных происходит деление на ноль или доступ в уже освобожденную память.
- В современных языках программирования предусмотрен механизм обработки исключений, при котором возможна передача управления из подпрограммы на уровни выше (в вызывающую подпрограмму), причем с учетом динамического полиморфизма неизвестно заранее, в какую именно точку.

Таким образом, в управляющем графе каждую вершину формально можно рассматривать как ветвление. Однако, получаемый таким образом управляющий граф будет проблематично использовать для выбора тестовых наборов или определения полноты тестирования, поскольку большая часть ветвлений в таком графе может быть пройдена только по одному (успешному, т. е. не вызывающему аварийного завершения) пути. Есть исследования по этому направлению, но проблема пока далека от решения [5].

2.4 Графовые модели и метапрограммирование

Если язык программирования поддерживает метапрограммирование, один и тот же исходный текст может порождать совершенно разный машинный код. Типичный пример — стандартная библиотека шаблонов C++ и анонимные функции в этом же языке. При тестировании шаблонного кода возникают следующие вопросы:

- Какую именно реализацию шаблонного кода проверять в модульных тестах?
- Как тестировать сами стандартные библиотеки шаблонов? Неизвестно точно, какие конкретно параметры шаблонов будут использоваться на практике.
- Как представить в модели вызов анонимной функции, если неизвестно, какое действие она выполняет? Возможный вариант решения — трактовать анонимные функции так же, как полиморфные методы в ООП (и использовать наработки из этой области).

Обойти эту проблему позволяет анализ программной системы на уровне машинного кода, но на этом уровне теряется информация о высокоуровневых сущностях и структурах данных, обрабатываемых системой.

2.5 Автоматизация выбора тестовых наборов

Одной из основных проблем структурного тестирования является проблема выбора конкретных тестовых наборов таким образом, чтобы максимизировать критерий полноты тестирования (например, покрытие кода). Для современного ПО в силу его сложности возможны варианты, когда тесты для полного покрытия проблематично выбрать даже вручную. Однако с учетом растущих темпов разработки в настоящее время нужно сводить время тестирования к минимуму при сохранении его полноты.

Задача выбора тестовых наборов при структурном тестировании может решаться по-разному:

- Случайное тестирование. В общем случае не может гарантировать удовлетворительную полноту тестирования, но позволяет в ряде случаев выявить нарушение нефункциональных требований (например, требований надежности — т. н. fuzz-тестирование).
- Символьное выполнение. Для путей в управляющем графе определяется условия их прохождения, исходя из которых подбираются тесты. Символьное выполнение может быть использовано для генерации тестов в следующих подходах:
  - Генерация тестов по путям - в управляющем графе выбираются тестовые пути, обеспечивающие необходимое тестовое покрытие, затем тестовые воздействия выбираются исходя из условий прохождения тестовых путей, получаемых с помощью символьного выполнения. Основной проблемой данного подхода является выбор нереализуемых тестовых путей.
  - Генерация тестов по целям - тестовые воздействия выбираются таким образом, чтобы достичь определенной вершины управляющего графа (например, оператора *assert*), при этом у генератора тестов есть свобода выбора тестовых путей, что снижает вероятность выбора нереализуемых путей.
  - Мутационное тестирование (Mutation testing) является разновидностью тестирования на основе путей, допускающее модификации программы с целью достижения необходимых участков кода.

- Генерация цепочек вызовов методов при тестировании объектно-ориентированных программ на основе отношений определения и использования переменных (def-use) и символьного выполнения методов.

Основная проблема применения символьного выполнения на практике заключается в том, что если программа выполняет сложную обработку данных, от полученных условий практически невозможно будет перейти к конкретным данным.

Тесты могут выбираться динамически на основании информации о тестовом покрытии и состоянии условий ветвлений при предыдущем тестовом прогоне (динамическое символьное выполнение [6]), при этом в программную систему требуется только вставить инструментальный код, вычисляющий тестовое покрытие, а решение о выборе теста может приниматься отдельно на основе граф-модели (управляющего графа, раскрашенного условиями ветвлений).

Еще один вариант — использование технологий машинного обучения — при этом на основании информации о соответствии входных данных и ветвлений, собранных при предыдущем тестовом прогоне, можно делать вывод о классах входных данных и генерировать новые данные.

2.6 Генерирование сложных тестовых данных

Зачастую входные данные тестируемой программной системы имеют сложную структуру. Например, они представляют собой текст на некотором языке, удовлетворяющий некоторой выводящей грамматике. Примеры таких программных систем - Web-сервера, использующие протокол HTTP, разного рода анализаторы текста, автоматические переводчики.

Генерирование тестовых данных здесь имеет свои особенности:

- Текст имеет переменную длину, при этом длина может передаваться в самом тексте (это особенность сетевых протоколов, например).
- Среди всех возможных текстов на заданном алфавите доля корректных (соответствующих грамматике) может быть малой. Это практически исключает случайное тестирование (оно может выполняться как fuzz-тестирование для выявления проблем надежности, но вероятнее всего не позволит проверить отдаленные от точки входа участки программы).
- Программная система может обрабатывать текст потоком (по частям), при этом временные интервалы между отдельными фрагментами текста могут влиять на поведение системы.

В настоящее время для тестирования систем, обрабатывающих данные такого рода, используются либо шаблонные данные (в этом случае от итерации к итерации выполняются одни и те же тексты), либо генераторы данных, написанные под конкретный язык. Автоматизация здесь возможна с помощью применения технологий синтеза текстов, причем время между блоками текста в потоке может моделироваться специальным символом расширенного алфавита.

2.7 Автоматизация проверки корректности

Даже если тестовые наборы выбраны, остается проблема проверки корректности работы программы — проблема синтеза тестовых оракулов. Обычно оракулы создаются вручную, однако в условиях постоянно меняющихся требований и обновляющихся тестов при ручном создании оракулов можно допустить ошибку и здесь. Сейчас выполняются исследования в этой области [7], но проблема также далека от полного решения.

Заключение

Постоянное развитие технологий и меняющиеся требования к ПО ставят новые вызовы перед верификацией ПО. Должна обеспечиваться не только полнота тестирования, но также возможность своевременного обновления тестов под новые требования – она достигается применением автоматизации структурного тестирования. В данном обзоре рассмотрены связанные с этим проблемы, и возможные пути решения требуют применения методов из смежных дисциплин, таких, как методы теории компиляции, методы машинного обучения, методы синтеза текстовых структур.

Литература

1. *Кулямин, В.В.* Методы верификации программного обеспечения / Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению "Информационно-телекоммуникационные системы", 2008. - 117 с;
2. *Patrick Kua, Rebecca Parsons, Neal Ford.* Building Evolutionary Architectures / O'Reilly Media, Inc, 2017.

3. *Bik, Niels & Lucassen, Garm & Brinkkemper, Sjaak. (2017).* A Reference Method for User Story Requirements in Agile Systems Development // 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW). P.292-298.
4. *Alex Shye, Matthew Iyer, Vijay Janapa Reddi, Daniel A. Connors.* Code coverage testing using hardware performance monitoring support, Proceedings of the sixth international symposium on Automated analysis-driven debugging, p.159-163, September 19-21, 2005, Monterey, California, USA
5. *Jo, J.W., Chang, B.M.* Constructing control flow graph for java by decoupling exception flow from normal flow. In: ICCSA (1). pp. 106–113 (2004).
6. *A.Yu. Gerasimov, S.S. Sargsyan, S.Sh. Kurmangaleev, J.A. Hakobyan, S.A. Asryan, M.K. Ermakov* Combining dynamic symbolic execution, code static analysis and fuzzing. Proceedings of the Institute for System Programming, vol. 30, issue 6, 2018, pp. 25-38. DOI: 10.15514/ISPRAS-2018-30(6)-2.
7. *Hai-Feng Guo.* A semantic approach for automated test oracle generation // Computer Languages, Systems & Structures. Vol 45, April 2016, P. 204-219.