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, Россия, г. Нижний Новгород ул. Минина д.24

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

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. 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 зависимости от предназначения программной системы, а также принятой методологии разработки могут выявляться разные нюансы проведения тестирования. Приведем далее некоторые актуальные проблемы, встающие перед верификацией ПО, 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 (Agile) 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 [2], 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. Здесь встает вопрос — какую стратегию обновления тестов выбрать. Для структурных тестов известно, какие конкретно программные модули они покрывают, но в общем неизвестно, какой функционал они затрагивают.

Возможное решение здесь следующее. Предположим, что в процессе разработки используется непрерывная интеграция — тесты выполняются на периодической основе. При этом для всех тестов производится определение тестового покрытия. В дальнейшем при изменении функциональных требований определяется множество затронутых функциональных тестов, а исходя из пересечения тестовых покрытий можно определить и затронутые структурные тесты.

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.). 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. Как следствие, данный метод влияет на характеристики работы системы, на ее соответствие нефункциональным требованиям.

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. 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;
- в современных языках программирования предусмотрен механизм обработки исключений, предполагающий возможность передачи управления в вызывающую подпрограмму.

Таким образом, в управляющем графе каждую вершину формально можно рассматривать как ветвление. Проблема в том, что если делать именно так, то проблематично будет в общем случае такой граф покрыть тестами хотя бы наполовину. Есть исследования по этому направлению, но проблема далека от решения [3].

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 моделировать вызов анонимной функции?

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 тестов 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 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).
- Символьное выполнение. Для путей в управляющем графе определяется условия их прохождения, исходя из которых подбираются тесты. Проблема в том, что если программа выполняет сложную обработку данных, от полученных условий практически невозможно будет перейти к конкретным данным.

Возможное решение здесь — комбинирование подходов. Тесты могут выбираться динамически на основании информации о тестовом покрытии и состоянии условий ветвлений при предыдущем тестовом прогоне (динамическое символьное выполнение [4]). 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.

Генерирование тестовых данных здесь имеет свои особенности:

- Текст имеет переменную длину;
- Среди всех возможных текстов на заданном алфавите доля корректных (соответствующих грамматике) может быть малой. Это практически исключает случайное тестирование.
- 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, либо генераторы данных, написанные под конкретный язык. Автоматизация здесь возможна с помощью применения технологий синтеза текстов.

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 постоянного обновления тестов при ручном создании оракулов можно допустить ошибку и здесь. В настоящее время выполняются исследования в этой области [5], но проблема пока 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. 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.

3. 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).

4. 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.

5. Hai-Feng Guo. A semantic approach for automated test oracle generation // Computer Languages, Systems & Structures. Vol 45, April 2016, P. 204-219.