Тестирование на основе моделей

В. В. Кулямин

Лекция 1. Качество программного обеспечения и методы контроля качества

Данный курс лекций посвящен тестированию программного обеспечения (ПО).

Многим людям, даже считающим себя хорошими программистами, но не имеющим опыта работы над большими проектами с жесткими сроками готовности и требованиями по качеству и надежности разработанного ПО, часто кажется, что тестирование не представляет особых проблем, по сравнению с созданием программ — сиди себе, пробуй, что работает, а что — нет. К большому сожалению, это не так. Тесты для сложной системы сами по себе сложны. Кроме того, для сложных систем требуется очень много тестов, чтобы проверить корректность работы реализуемых ими функций во многих различных ситуациях, поэтому наборы тестов тоже представляют собой сложные программные комплексы, требующие особых методов для их разработки и сопровождения, и, как это не покажется странным, применения творческих способностей от людей, создающих их.

В этом курс рассказывается о специфических методах построения тестов, которые в последнее время обычно выделяются в особую область — *тестирование на основе моделей*. К значению этого термина мы вернемся несколько позже, а сейчас отметим, что эта область находится на границе между *теоретической информатикой (computer science)* и *инженерией программных систем (software engineering)*. Поэтому она требует как определенной математической подготовки, знакомства с техниками строгого описания свойств программного обеспечения, так и умения применять их на практике и оценивать практическую эффективность используемых подходов на пути к приемлемому компромиссу между полнотой тестирования и затраченными на него ресурсами.

Часто говорят, что тестирование предназначено для поиска ошибок и проверки правильности или надежности создаваемых программных систем, или, более точно, для контроля их *качества*. Поскольку понятие качества программной системы не является вполне очевидным, начнем с его более детального рассмотрения.

Качество программного обеспечения

Можно определить качество ПО, как его пригодность и удобство для решения тех задач, для которых оно создано (так же, как и качество любого инструмента). Однако у программных систем есть две особенности, отличающие их, если не от всех, то от многих других инструментов, используемых человеком в своей деятельности.

- Во-первых, цели создания программной системы чаще всего сложны и включают множество аспектов. Программное обеспечение, за редким исключением, не разрабатывается для решения ровно одной задачи. Гораздо чаще это целый набор связанных задач из некоторой области. Кроме того, в него входят и экономическая эффективность использования данной системы и удобство работы с ней для того персонала, который имеется у организации-заказчика.
- Во-вторых, очень часто программы используются для решения несколько не тех задач, для которых они предназначались при создании. Набор целей, для достижения которых применяется данная система, изменяется со временем, что отражается и в постоянном добавлении новых возможностей и функций в новые версии программ.

Поэтому целостное и четкое понятие качества ПО определить очень нелегко. Вместо этого используют различные *модели качества*, систематизирующие набор аспектов, характеристик и метрик качества, рассмотрение которых необходимо для адекватной оценки качества

разнообразных программ. Такие модели могут изменяться со временем, поскольку изменяются потребности индустрии производства ПО, появляются новые группы возможностей или внимание разработчиков привлекается к новым аспектам качества, ранее считавшимся несущественными.

Наиболее широко на данный момент используется модель качества ПО, зафиксированная в наборе стандартов ISO 9126 [1-4]. В несколько упрощенном виде (при рассмотрении так называемого *внутреннего качества*) эта модель определяет 6 основных характеристик качества программного обеспечения. Каждая характеристика уточняется при помощи некоторого набора более детальных атрибутов.

Рисунок 1. Характеристики и атрибуты качества ПО по ISO 9126.

- *Функциональность*.
  Эта характеристика обозначает способность ПО решать определенный круг задач. Функциональность определяет, что именно делает данная программа. Атрибуты функциональности следующие: функциональная пригодность — способность решать нужный набор задач; точность выдаваемых результатов; защищенность — способность предотвращать доступ к функциям и данным ПО людям или другим системам, у которых нет прав на это; способность к взаимодействию с другими системами; и др.

- *Надежность.*
  Это способность ПО поддерживать определенный уровень работоспособности в заданных условиях.
  Надежность является вероятностной характеристикой работоспособности ПО. Атрибуты ее таковы: зрелость — обратная величина к частоте отказов ПО; устойчивость к отказам, способность выполнять определенные задачи и придерживаться некоторых ограничений даже в случае отказов и сбоев; способность к восстановлению после отказов и среднее время такого восстановления; и др.

- *Удобство использования.*
  Удобство использования показывает, насколько ПО привлекательно, удобно в обучении работе с ним и при выполнении самой работы.
  К атрибутам удобства использования относятся: понятность — показатель, обратный к усилиям, затрачиваемым пользователями на понимание основных понятий и способов работы ПО и их применимости для решения нужных им задач; удобство обучения, обратное к усилиям на обучение работе с системой; удобство работы, обратное к

усилиям на выполнение определенного круга задач; привлекательность, способность привлекать новых пользователей; и др.

- *Производительность.*
  Это способность ПО обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам. В соответствии с затратами ресурсов разного вида — времени, памяти, пропускной способности сетевых соединений — выделяются и различные атрибуты производительности.

- *Переносимость.*
  Эта характеристика показывает сохранение работоспособности ПО при изменении его окружения.
  Ее атрибутами являются, например, возможность развертывания или установки ПО в различных окружениях и его адаптируемость — способность приспосабливаться к работе в различных окружениях при помощи действий, зафиксированных в документации.

- *Удобство сопровождения.*
  Удобство сопровождения определяет трудоемкость анализа, исправления ошибок и внесения изменений в ПО.
  Его атрибутами являются, в частности, удобство проведения тестирования, удобство внесения изменений и риск возникновения неожиданных эффектов при изменениях.

В 2011 году принят стандарт ISO 25010 [5], заменяющий ISO 9126-1 и несколько изменяющий набор характеристик и атрибутов внутреннего качества ПО. В его рамках имеются следующие характеристики.

- ***Функциональность*** (теперь называемая functional suitability)
  - Функциональная пригодность (functional appropriateness) — способность ПО решать нужные пользователям задачи;
  - Функциональная полнота (functional completeness) — определяет, насколько полно ПО способно решать нужный набор задач;
  - Точность (functional correctness) — способность выдавать результаты с нужной точностью;

- ***Производительность*** (performance efficiency)
  - Временная эффективность (time behavior) — способность ПО выдавать ожидаемые результаты, а также обеспечивать передачу необходимого объема данных за отведенное время;
  - Эффективность использования ресурсов (вычислительных, resource utilization) — способность решать нужные задачи с использованием определенных объемов ресурсов (памяти различных видов, устройств ввода-вывода и пр.);
  - Пропускная способность каналов связи (capacity) — способность решать нужные задачи при определенных ограничениях на пропускаемые через каналы связи объемы информации;

- ***Совместимость*** (compatibility)
  - Способность к сосуществованию (co-existence) — из переносимости по ISO 9126, способность ПО сосуществовать с другими программами в общем окружении, деля с ними ресурсы;
  - Способность к взаимодействию (interoperability) — из функциональности по ISO 9126, способность взаимодействовать с нужным набором других систем;

- ***Удобство использования*** (usability)
  - Удобство обучения (learnability) — показатель, обратный усилиям, затрачиваемым пользователями на обучение выполнению определенных задач с помощью ПО;

- Удобство работы (operability) — показатель, обратный усилиям, предпринимаемым пользователями для решения своих задач с помощью ПО;
- Понятность (теперь appropriateness recognizability) — показатель, обратный к усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание их применимости для решения своих задач;
- Эстетичность (бывшая привлекательность, user interface aesthetics) — способность ПО быть привлекательным для пользователей, не вызывать эстетического отторжения;
- Защищенность от ошибок пользователей (user error protection) — способность игнорировать или исправлять определенные ошибки пользователей;
- Доступность (при различных способностях пользователей, accessibility) — способность поддерживать работу людей с ограниченными возможностями — при нарушении восприятия цветов и сильных дефектах зрения, некоторых нарушениях координации движений, и пр.;

- ***Надежность*** (reliability)
  - Зрелость (maturity) — показатель, обратный частоте отказов ПО, обычно измеряется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени;
  - Способность к восстановлению (recoverability) — способность восстанавливать определенный уровень работоспособности и целостность данных после отказа, при затрате определенного времени и ресурсов;
  - Устойчивость к ошибкам (fault tolerance) — способность поддерживать заданный уровень работоспособности при отказах и некоторых нарушениях правил взаимодействия с окружением;
  - Работоспособность (availability, иногда также переводится как доступность) — возможность ПО решать задачи и предоставлять пользователям информацию, несмотря на ненадежную работу сетей, отдельных серверов и т.д.;

- ***Защищенность*** (security)
  - Конфиденциальность (confidentiality) — способность ПО защищать свои данные от доступа лиц, которые не имеют к ним допуска;
  - Целостность (integrity) — способность ПО защищать свои данные от изменения теми лицами. которые не имеют на это прав;
  - Строгое выполнение обязательств (неотвергаемость, non-repudiation) — способность дать убедительное подтверждение тому, что заданные операции действительно выполнялись авторизованными пользователями (а их результаты не были внесены каким-либо несанкционированным образом);
  - Авторизуемость (операций, accountability) — возможность проследить, какие пользователи выполняли заданные операции;
  - Аутентичность (authenticity) — способность дать подтверждение собственной идентичности (т.е., отсутствия подмены части кода или модулей по сравнению с проверенными и сертифицированными версиями) и идентичности пользователей (т.е., возможность дать определенные гарантии, что выступающий под некоторым идентификатором пользователя человек, это именно он, или имеющий право действовать от его имени);

- ***Удобство сопровождения*** (maintainability)
  - Удобство проверки (testability) — показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам;
  - Анализируемость (analyzability) — удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий;

    - Модифицируемость (modifiability, удобство внесения изменений + стабильность) — показатель, обратный трудозатратам на выполнение необходимых изменений и риску возникновения неожиданных эффектов после них;
    - Модульность (modularity) — возможность вносить изменения в отдельные модули с минимальным их влиянием на другие;
    - Повторная используемость (reusability) — возможность использовать отдельные модули без модификации в рамках других систем;
- ***Переносимость*** (portability)
    - Адаптируемость (adaptability) — способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных;
    - Удобство замены (replaceability) — возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении;
    - Удобство установки (installability) — способность ПО быть установленным или развернутым в определенном окружении;

Требования к программному обеспечению

Перечисленные характеристики качества ПО представляют собой одну из систематик различных видов *требований* к программным системам. Такая систематика полезна, если необходимо выделить и проанализировать требования для сложной системы. Она позволяет организовать анализ, разбивая возможные требования и ограничения на разные группы и позволяя последовательно рассматривать различные аспекты системы. Сами требования как раз и говорят о том, какие конкретные свойства и характеристики должна иметь система, чтобы с ее помощью можно было успешно решать заданный набор задач.

Требования определяют, какое поведение системы является правильным и желаемым, а какое должно рассматриваться как некорректное. Поэтому они играют первостепенную роль при тестировании — именно они и проверяются с его помощью, а ошибки связаны именно с их нарушениями.

Сами требования определяются при анализе задач, стоящих перед рассматриваемой программной системой и ее окружения. Они могут извлекаться из различных источников.

- При аккуратном следовании стандартам на процессы разработки ПО требования обычно фиксируются в документе, создаваемом при решении о разработке системы и называемом *техническим заданием* (а по-английски — *requirements specification, спецификация требований*). Иногда этот документ создается представителями заказчика, но часто его приходится писать самим разработчикам на основе информации, полученной из других источников.
- Важным источником требований являются *стандарты*, регламентирующие характеристики, функции и состав систем, работающих в определенной предметной области. Такие стандарты создаются на основе опыта, накопленного в большом количестве проектов, в ходе которых такие системы создавались или развивались, и поэтому содержат ценную информацию о желательных характеристиках таких систем. Часто работа системы связана с выполнением каких-то действий, регулируемых существующим законодательством и нормами, действующими в данной области. Нормы и законы тоже являются разновидностью действующих стандартов, хотя обычно они формулируются менее четко и однозначно, чем технические стандарты и правила.
- Разработчики сами могут осознать, что что-то должно быть сделано определенным образом — так обычно возникают *внутренние ограничения задач*, не соблюдая которые,

невозможно решить их правильно. Чаще всего ограничения такого рода можно усмотреть при детальном анализе решаемой задачи.

- Иногда ряд требований к новой системе можно сформулировать на основе анализа *уже существующих систем* для решения схожих задач.
- Б*о*льшая часть требований формулируется на основе явно высказываемых *пожеланий* пользователей системы, их руководителей, заказчиков ее разработки и других заинтересованных лиц.
- Наконец, наиболее нечетким, но, тем не менее, достаточно важным источником требований являются невысказанные явно *потребности и нужды* пользователей создаваемой системы, которые, несмотря на это, все же часто поддаются анализу.

При извлечении требований из перечисленных выше источников их четкость и согласованность возрастают при движении от потребностей и пожеланий к стандартам и техническому заданию.

Тестирование проверяет соответствие требованиям, и поэтому чем более точно и ясно они сформулированы, тем аккуратнее и полнее можно провести тестирование. Если какие-то требования определены нечетко, проверка их тоже может быть выполнена лишь с определенной степенью точностью, и системы, ведущие себя сильно по-разному, могут быть признаны удовлетворяющими ему в одинаковой мере. Получаемые при этом оценки качества таких систем будут во многом субъективными.

В некоторых случаях тестирование кажется возможным и в отсутствии всяких требований, по крайней мере, документально зафиксированных. В этих случаях, однако, используются какие-то свойства, которые считаются само собой разумеющимися и, соответственно, представляющие собой неявные требования. Пример такого свойства — отсутствие сбоев при работе программы. Если такие неявные требования действительно должны быть выполнены, можно проводить тестирование на их основе. Если же это не так, то есть иногда сбой может рассматриваться как корректное поведение системы (если, скажем, вводятся совсем некорректные исходные данные), то подобное тестирование может привести к неправильным выводам о наличии ошибок в системе.

Чтобы требования к программному обеспечению можно было уверенно использовать при его разработке и тестировании, они должны обладать рядом характеристик, которые зафиксированы в стандартах, регламентирующих разработку программного обеспечения.

Два таких стандарта — IEEE 830 [6] и IEEE 1233 [7] — определяют следующие характеристики правильно составленных требований к ПО.

- *Адекватность*, соответствие реальным потребностям пользователей ПО.
- *Однозначность*, отсутствие двусмысленностей и возможностей разного толкования.
- *Полнота* — отражение в требованиях всех существенных потребностей и всех ситуаций, в которых система должна будет функционировать.
- *Непротиворечивость* или согласованность между разными элементами требований.
- *Систематичность* представления — требования должны быть описаны в рамках некоторой системы с четким указанием места каждого требования среди остальных, с определением связей и зависимостей между ними и приоритетности для различных заинтересованных лиц.
- *Прослеживаемость* — требования должны иметь четко определенные связи с модулями разрабатываемой системы, частями проектной документации и тестами, чтобы всегда можно было определить, для выполнения или проверки каких требований создан каждый из этих элементов и насколько он им соответствует.

- *Проверяемость* или возможность для каждого требования однозначно установить при помощи некоторых действий, выполнено это требование или нет.
- *Модифицируемость* или возможность внесения изменений в набор требований с максимально быстрым отслеживанием последствий такой модификации и исправлением всех возникающих при этом дефектов с точки зрения других характеристик.

В ходе работы над создаваемой программной системой или переработки уже существующей всегда, в том или ином виде, проводится *анализ требований*, цель которого — подготовить представление требований к ПО, имеющее все указанные характеристики. Анализ требований обычно включает следующие виды деятельности.

- ***Выделение требований***. Его задача — определить полный список требований, уточнить недостаточно четко сформулированные и определить возможные компромиссы в тех случаях, когда различные заинтересованные лица высказывают противоречащие друг другу требования.
  Выделение требований включает определение доступных источников требований, извлечение требований из них, в ходе чего, собственно, и фиксируются отдельные требования, и согласование требований, полученных от разных источников, при возникновении необходимости в этом.
  При извлечении требований может применяться широкий диапазон различных техник — от простого анализа задач, анализа имеющихся документов, до проведения интервью, опросов и семинаров, с использованием специальных методов для фиксации как высказываемых пожеланий и формулировок, так и эмоционального состояния опрашиваемых, чтобы в дальнейшем оценить правдивость и полноту предоставленных сведений.
- ***Систематизация и описание требований***, их сведение в единую систему и составление представляющих ее моделей, отражающих различные аспекты собранных требований. При систематизации особое внимание уделяется полному отражению всех извлеченных сведений в требованиях, определению связей и зависимостей между требованиями, идентификации требований, необходимой, чтобы иметь возможность ссылаться на них из различных проектных документов.
- ***Валидация и верификация требований.***
  Задача этих видов деятельности — проверка необходимых свойств требований к ПО. Валидация представляет собой проверку адекватности и полноты требований, то есть проверяет, что зафиксированные в требованиях ограничения действительно представляют потребности пользователей, заказчиков и других заинтересованных лиц, а также, что все их существенные потребности нашли соответствующее отражение в требованиях.
  Верификация проверяет внутреннюю согласованность, непротиворечивость и однозначность требований, а также их проверяемость и возможность проследить связи требований друг с другом, с кодом, тестами и другими проектными документами.

Ошибки в программном обеспечении

Наиболее наглядными результатами тестирования являются обнаруженные в тестируемой системе ошибки, то есть расхождения между ее реальным поведением и требованиями.

В литературе по программной инженерии слово «ошибка» используется в нескольких различных значениях.

- Иногда, хотя и достаточно редко, так называют произвольный *дефект* программной системы, будь то сбой, полностью разрушающий данные системы, неточно прорисованная буква на кнопке графического интерфейса пользователя или

нестандартное форматирование исходного кода. В англоязычной литературе в этом смысле чаще всего употребляется термин *defect*.

- Часто ошибкой или *сбоем* называют наблюдаемое нарушение требований, проявляющееся при некотором сценарии работы рассматриваемой системы. В английском языке этому значению соответствует термин *failure*.

- Ошибка в коде программы, вызывающая сбой, и состоящая в неправильном использовании какой-то конструкции языка программирования, употреблении лишней конструкции или в пропуске необходимой. В англоязычной литературе ошибка такого рода называется *fault*.
  Ошибка такого рода определена нечетко, в отличие от предыдущего случая, поскольку неправильная работа некоторого кода часто может быть исправлена несколькими разными способами. Если есть несколько конструкций, исправление каждой из которых удаляет эту ошибку, тяжело определить, какая именно из них ошибочна.

- Ошибка аналитика, архитектора или программиста, заключающаяся в неправильном понимании определенного требования или ограничения, в том, что какое-то требование забыто, или, наоборот, используется лишнее требование. По-английски такая ошибка называется обычно *error*.
  Иногда в англоязычной литературе термин *error* употребляется еще и в другом смысле — так называют некорректные, не соответствующие наложенным ограничениям данные, все равно, возвращаемые системой в ответ на какой-либо запрос или возникающие в ходе внутренних вычислений в системе и не видимые извне.

Основной смысл терминов failure, fault и error достаточно тесно связан с основными источниками ошибок, которых тоже три.

- *Неправильное понимание задач.*
  Очень часто люди не понимают или понимают неправильно то, что им пытаются сказать другие. Разработчики ПО тоже не всегда понимают, что именно нужно сделать. Дополнительным источником трудностей может служить отсутствие четкого понимания задач у самих пользователей и заказчиков — чаще всего они лишь приблизительно могут сформулировать проблему или могут попросить сделать несколько не то, что им действительно нужно.
  Ошибки такого рода тяжелее всего обнаруживаются и стоят достаточно дорого.
  Для их предотвращения нужно проводить тщательный анализ предметной области, уточнять каждое сформулированное требование, анализировать его причины и связи с другими требованиями и ограничениями, переформулировать выявленные пожелания и требования, выяснять и уточнять корректность полученных формулировок у пользователей и экспертов в предметной области. Поскольку суть таких ошибок — неадекватное понимание требований, защититься от них помогает постоянный контроль этого понимания на основе формирования различных его следствий и проверки их корректности у экспертов и пользователей.

- *Неправильное решение задач.*
  Даже правильно поняв, что нужно сделать, разработчики часто выбирают неправильный подход к тому, как это делать. Выбираемые решения могут обеспечивать лишь некоторые из требуемых свойств, могут решать поставленную задачу лишь для одного класса ситуаций из нескольких возможных, они могут подходить для данной задачи в теории, но плохо работать на практике, в конкретных обстоятельствах и в том окружении, в которых должна будет работать создаваемая система.
  Помочь в выборе правильного решения может сопоставление альтернативных решений, тщательный их анализ на предмет соответствия всем требованиям, поддержание постоянной связи с пользователями и заказчиками, предоставление им информации о предлагаемых решениях, демонстрация прототипов, анализ пригодности выбираемых

решений для работы именно в том контексте, в котором они будут использоваться, тестирование прототипов и моделей.

- *Неправильный перенос решений в код.*
  Имея правильное решение правильно понятой задачи, люди, тем не менее, способны сделать достаточно много ошибок при его воплощении. Корректному представлению решений в коде могут помешать как обычные опечатки, так и забывчивость программиста или его нежелание отказаться от привычных приемов, которые не дают возможности аккуратно записать именно то, что нужно.
  Хотя такие ошибки обычно более легко устраняются, они способны нанести не меньший ущерб, чем ошибки предыдущих видов.
  С этими ошибками можно справиться при помощи инспектирования кода, взаимного контроля, при котором разработчики внимательно читают код друг друга, опережающей разработки модульных тестов и тестирования.

Среди специалистов в программной инженерии установилось понимание того, что при достижении системой определенного уровня сложности ошибки в ней становятся неустранимыми — сколько бы усилий не было потрачено на их поиск и отладку, гарантировать безошибочность такой системы невозможно. Это связано, в первую очередь, с существованием определенного порога сложности, за пределами которого человек уже не в состоянии представлять себе происходящее в системе во всех деталях, в частности, с невозможностью абсолютно четко сформулировать полный и согласованный набор требований к системе, решающей большой набор сложных задач. С другой стороны, реальные требования к полезному ПО постоянно изменяются в связи с изменениями в потребностях пользователей, бизнеса и общества, для удовлетворения которых создаются программные системы, поэтому разработка точных и согласованных требований может просто не успевать за развитием рынка.

Ошибки в ПО иногда приводят к серьезным инцидентам и значительным убыткам для использующих его организаций. Конечно, чаще возникают мелкие ошибки, не приводящие к серьезным последствиям. Зависимость между количеством ошибок и размером их последствий подчиняется закону Ципфа (Zipf) [9] — количество случающихся сбоев примерно обратно пропорционально величине наносимого ими ущерба.

Далее описывается несколько примеров ошибок в программном обеспечении, имевших серьезные последствия.

- Ошибка в системе управления космическим аппаратом Mariner 1 [10].
  Эта ошибка привела к уничтожению одного из первых кораблей, направлявшегося к Венере, через несколько минут после запуска 22 июля 1962 года.
  В ходе полета антенна связи вышла из строя, связь со службой управления была потеряна, и управление полетом взял на себя бортовой компьютер. Однако в одной из формул для расчета положения было забыто усреднение скорости по нескольким последовательным измеренным значениям — в результате небольшие колебания скорости, связанные с неточностью измерительной аппаратуры, стали рассматриваться системой как серьезные, она стала предпринимать «корректирующие» действия, в результате чего корабль сошел с курса и был уничтожен.

- Ошибка в программном обеспечении, управляющем аппаратом радиационной терапии Therac-25 [11].
  За 1985-1987 годы зафиксировано 6 инцидентов со смертельным исходом, связанных с его работой. В трех из них непосредственной причиной смерти пациентов было признано именно их повышенное облучение из-за ошибки в программной системе управления аппаратом.
  Аппарат имел два режима облучения — мягкое облучение электронами и рентгеновское облучение. Во втором случае с источника электронных лучей снимался фильтр, который

ослаблял их интенсивность, но между пациентом и источником излучения устанавливался специальный экран, падая на который мощные электронные лучи вызывали рентгеновское излучение.
Ошибка проявлялась, когда оператор сначала включал первый режим, а потом слишком быстро переключал аппарат на второй. При этом ослабляющий фильтр снимался, а экран не устанавливался, и пациент подвергался очень интенсивному облучению электронными лучами. Кроме того, оператору при этом сообщалось, что пациент не получил никакой дозы, что не позволяло адекватно среагировать на происходящее.
Ошибка возникала лишь иногда и была связана с несинхронизованным выполнением модулей, управлявших различными элементами аппарата. При эксплуатационном тестировании она не была обнаружена, поскольку операторы тогда еще не научились переключать режимы достаточно быстро.

- Ошибка в системе управления космическим аппаратом Фобос 1 [12].
Привела к потере связи с кораблем, уже находившимся на пути к Марсу, 2 сентября 1988 года. Корабль перестал ориентироваться в пространстве, не смог сориентировать солнечные батареи и израсходовал аккумуляторы, поскольку были отключены навигационные приборы для определения положения относительно Солнца.
Команда отключения приборов была в тестовой подпрограмме, использовавшейся на Земле для проверки работоспособности отдельных систем, удалить этот код не успели перед вылетом, поскольку он был записан в памяти, предназначенной только для чтения, а для ее замены требовалось существенное время. Казалось, однако, что в ходе полета эта программа никогда не будет вызвана. Но при передаче команд по корректировке курса 29 августа 1988 года была допущена ошибка — пропущен один символ, что привело к выполнению этой тестовой программы.
Другой корабль этой серии, Фобос 2, был также потерян из-за какой-то ошибки в системе управления 27 марта 1989 года, уже на орбите вокруг Марса.

- 25 февраля 1991 года во время Первой войны в Персидском заливе американская система ПВО Patriot не смогла сбить иракскую ракету Скад, которая в результате попала в барак американской армии, убив 28 человек и ранив около ста [13].
Причиной промаха Patriot, как выяснилось, было накопление ошибок округления за время работы системы. Время в ней измерялось в десятых долях секунды, а числа были представлены в 24-битном двоичном формате с плавающей точкой. При представлении 1/10 как двоичной дроби с 24-мя цифрами возникает небольшая ошибка.
В рассматриваемом случае система Patriot работала без перезагрузки около 100 часов. За это время накопление погрешности определения времени дало ошибку около 1/3 секунды. Поскольку ракета Скад летит со скоростью 1700 м/с, ошибка в 1/10 секунды при расчете ее траектории уже не дает возможности ее сбить.

- Многочисленные ошибки в системе управления двигателями и навигационной системе считаются наиболее вероятной причиной катастрофы вертолета Chinook ZD 576 [14], произошедшей 2 июня 1994 года на мысе Кинтайр. В этой катастрофе погибли 25 экспертов и высокопоставленных сотрудников отдела разведки Великобритании в Северной Ирландии, что на значительное время парализовало работу этого отдела.

- Ошибка в системе управления ракеты Ариан-5 привела к ее уничтожению при первом запуске этой ракеты 4 июня 1996 года [15].
Долгое время эта ошибка, приведшая к убыткам в размере 500 миллионов долларов США, считалась самой дорогостоящей ошибкой в программной системе.
Ошибка состояла в том, что без изменений использовался модуль расчета траектории из системы управления ракетой Ариан-4. В нем горизонтальная составляющая скорости ракеты представлялась 16-битным числом. Ариан-5 могла выдерживать более значительные ускорения и большую кривизну траектории, из-за чего в ходе полета значение горизонтальной скорости вышло за пределы представимых 16-ю битами чисел.

Специальной процедуры обработки такой ситуации не было, поэтому возникшее исключение обрабатывалось модулем обработки общих сбоев, который остановил данный процесс и запустил новый с теми же исходными данными, что вновь привело к той же ошибке. В результате система не смогла вычислить правильное текущее положение ракеты и стала использовать ранее полученные данные. Это привело к попытке «скорректировать» курс и «болтанию» ракеты, после чего она была уничтожена.

- Ошибка в системе управления космическим аппаратом Mars Climate Orbiter [16].
  Привела к его выходу на слишком низкую орбиту вокруг Марса 23 сентября 1999 года и к последовавшему за этим разрушению.
  Необходимые корректировки к движению корабля рассчитывались специальной программой на Земле и после передавались в виде команд двигателям аппарата. Ошибка состояла в том, что управляющая программа на Земле использовала значения импульсов в фунтах силы на секунду, а бортовая система передавала ей значения, измеренные в Ньютонах на секунду. В результате были использованы неправильные команды корректировки.
- Одной из причин сбоя в электроснабжении северо-востока Северной Америки 14 августа 2003 года, на несколько часов оставившего без электричества 50 миллионов человек и приведшего к потерям на сумму около 6 миллиардов долларов США, была ошибка в программной системе оповещения о сбоях на электростанции, связанная с неаккуратной синхронизацией параллельно работающих процессов [17].

Можно отметить, что в большинстве примеров ошибок, имевших тяжелые последствия, нельзя однозначно приписать всю вину за случившееся ровно одному недочету или дефекту, одному месту в коде. К таким последствиям чаще всего приводят ошибки системного характера, затрагивающие многие элементы системы и различные аспекты ее устройства. Это значит, что при анализе такого происшествия обычно выявляется множество частных ошибок, нарушений действующих правил, недочетов в инструкциях и требованиях, которые совместно привели к создавшейся ситуации.

Даже если ограничиться рассмотрением только ПО, часто одно проявление ошибки (failure) может выявить несколько дефектов, находящихся в разных местах. Такие ошибки возникают обычно в тех ситуациях, в которых поведение системы недостаточно четко определяется требованиями (а иногда и вообще никак не определяется, вследствие неполного понимания задачи). Поэтому разработчики различных модулей ПО имеют возможность по-разному интерпретировать те части требований, которые относятся непосредственно к их модулям, а также иметь разные мнения по поводу области ответственности каждого из взаимодействующих модулей в данной ситуации. Если различия в их понимании не выявляются достаточно рано, при разработке системы, то становятся «минами замедленного действия» в ее коде.

При подготовке и проведении тестирования эти закономерности стоит учитывать, это помогает как находить ошибки быстрее и с меньшими трудозатратами, так и более аккуратно анализировать их последствия и устранять все возможные связанные с ними эффекты.

Литература

[1] ISO/IEC 9126-1:2001. Software engineering — Software product quality — Part 1: Quality model, 2001.

[2] ISO/IEC TR 9126-2:2003 Software engineering — Product quality — Part 2: External metrics, 2003.

[3] ISO/IEC TR 9126-3:2003 Software engineering — Product quality — Part 3: Internal metrics, 2003.

[4] ISO/IEC TR 9126-4:2004 Software engineering — Product quality — Part 4: Quality in use metrics, 2003.
[5] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011.
[6] IEEE 830-1998. Recommended Practice for Software Requirements Specifications. New York: IEEE, 1998.
[7] IEEE 1233-1998. Guide for Developing System Requirements Specifications. New York: IEEE, 1998.
[8] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт 13, ИСП РАН, Москва, 2006. http://panda.ispras.ru/~kuliamin/docs/Req-2006-ru.pdf.
[9] Статья Wikipedia о законе Ципфа http://en.wikipedia.org/wiki/Zipf%27s_law.
[10] http://nssdc.gsfc.nasa.gov/nmc/tmp/MARIN1.html.
[11] N. Levenson, C. S. Turner. An Investigation of the Therac-25 Accidents. IEEE Computer, 26(7):18-41, July 1993.
[12] R. Z. Sagdeev, A. V. Zakharov. Brief history of the Phobos mission. Nature 341:581-585, 1989.
[13] G. N. Lewis, S. Fetter, L. Gronlund. Casualties and Damage from Scud Attacks in the 1991 Gulf War, 1993. http://web.mit.edu/ssp/Publications/working_papers/wp93-2.pdf.
[14] http://www.publications.parliament.uk/pa/ld200102/ldselect/ldchin/25/2501.htm.
[15] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.
[16] Mars Climate Orbiter Mishap Investigation Board Phase I Report, 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.
[17] http://www.nyiso.com/public/webdocs/newsroom/press_releases/2005/blackout_rpt_final.pdf.

Как защититься от ошибок

В прошлой лекции шла речь о качестве программного обеспечения (ПО) и его возможных дефектах — ошибках. Каким образом можно обеспечить качество и, тем самым, защититься от ошибок в ПО? Эту задачу решают виды деятельности в рамках разработки и сопровождения ПО, связанные с *обеспечением качества*. Любой систематический подход к обеспечению качества должен включать следующие три пункта.

- Методы предотвращения ошибок
- Методы обнаружения ошибок
- Методы исправления ошибок

Если ошибки не обнаружить или не исправить, то они не исчезнут, поэтому последние два пункта необходимы. Методы предотвращения ошибок нужны для снижения нагрузки на работу по оставшимся пунктам — иначе в сложных случаях можно оказаться в ситуации, в которой количество обнаруживаемых ошибок значительно превышает возможности их исправления (на практике так тоже бывает, но величину этого превышения стараются держать ограниченной). Также, стоит помнить, что самые совершенные методы предотвращения ошибок не способны справиться со всеми их видами — все равно ошибки возникают и их надо уметь находить, т.е., наличие высокоэффективных технологий предотвращения ошибок не делает работу по их обнаружению и исправлению ненужной.

Примеры методов предотвращения ошибок.

- Стандартизация интерфейсов широко используемых библиотек, протоколов взаимодействия и создание хорошей документации по их интерфейсам. Примерами таких стандартов являются POSIX, стандартная библиотека Java JDK. Хорошая, соответствующая реальной работе описываемыхого ПО и одновременно

лишенная неполноты, двусмысленностей и рассогласованности документация по стандартам позволяет разработчикам, участвующим в реализации билиотек точнее и полнее понимать, какую функциональность нужно обеспечить и избегать ошибок в ее реализации и несогласованностей при реализации одного стандарта разными разработчиками. Она же дает возможность тем программистам, которые пользуются этими библиотеками, с меньшими усилиями, точнее и полнее понимать их функции, не тратить время на эксперименты и отладку с целью выяснения точных правил их работы, избегать затрат на создание кода, переносимого между несколькими несогласованными реализациями одного стандарта.

- Разработка новых конструкций языков программирования, позволяющих эффективно проверять больше свойств корректности программ, и устранение конструкций, вызывающих многочисленные ошибки.
  Например, частое использование инструкции goto без ограничений на место перехода приводит к запутанному коду, с большим количеством ошибок. Поэтому в большинстве современных языков эта инструкция отсутсвует или употрбеляется крайне ограниченно.
  В Eiffel предложено использование конструкций, которые позволяют избавиться от одной из наиболее часто встречающихся ошибок — попытке обращения к методу или атрибуту по пустой ссылке (NullPointerException). Решение состоит в использовании необнуляемых типов (тип ссыок на объекты, которые не могут быть пустыми) и конструкции завершения цикла при обнаружении пустой ссылки в обрабатываемой им коллекции. После введения этих конструкций компилятор объявляет ошибкой любой проход по ссылке обнуляемого типа, сделанный вне блока, в начале которого стоит проверка этой ссылки на равенство null.
- Внедерние стандартов кодирования, делающих код программ более понятным и позволяющих тратить меньше времени и ресурсов на понимание и внесение изменений в программы.
- Регулярное предварительное обсуждение реализуемых проектных и программистких решений в группе, позволяющее избегать ошибок, связанных с пропуском в коде обработки специфических ситуаций и игнорированием важных элементов требований.

Исправление ошибок по существу мало отличается от собственно разработки кода, используя дополнительно специфические техники отладки — локализации ошибок при помощи постепенного сужения области анализа.

Методы контроля качества ПО

Данный курс посвящен тестированию — одному из наиболее широко используемых методов контроля качества ПО, или, методов обнаружения ошибок, однако оно — не единственный такой метод.

Методы *контроля качества* предназначены для проверки различных характеристик качества ПО и поиска различных дефектов, связанных с недостаточным качеством. Они обычно разделяются на верификацию и валидацию.

В рамках *верификации* качество некоторого артефакта (документа, модели, элемента кода и пр.) проверяется за счет сопоставления его с другим артефактом, на основе которого первый должен был быть разработан или которому он должен соответствовать (а также за счет проверки его соответсвия принятым нормам и стандартам разработки). Так, верифицировать код можно, имея на руках описание требований, проектные решения или стандарты кодирования, верифицировать проектный документ можно с помощью требований или стандартов на оформление подобных документов. Верифицировать можно и реальное

поведение системы — сопоставляя его с требованиями, проектными решениями, принятыми стандартами функционирования систем такого рода.

Валидация обозначает проверку некоторого артефакта разработки на соответствие конечным целям, для достижения которых это ПО предназначено, т. е., нуждам и потребностям его пользователей и заказчиков. При валидации ПО проверяется, что оно действительно решает нужные пользователям задачи и удовлетворяет их потребности (даже если эти задачи и потребности описаны плохо и неполно). Валидация обычно проводится представителями заказчика, пользователями, экспертами в предметной области, т.е. людьми, обладающими достаточной компетентностью, чтобы судить о достижении поставленных целей. Если же эти цели формализовать, описать точно и полно, то проверка на соответствие полученному документу будет верификацией. Валидация необходима, потому что обычно согласованное, полное и точное описание задач сложной системы практически невозможно, разрабатываемые документы со спецификациями требований и пр. являются только приближениями к такому описанию. При валидации, могут использоваться те же техники, что и при выявления требований, поскольку цели этих видов деятельности похожи — преобразовать неясные и неформальные пожелания и представления о работе ПО в более точную форму (при валидации — в оценку проверяемых характеристик качества).

Различие между верификацией и валидацией проиллюстрировано на Рис. 1.

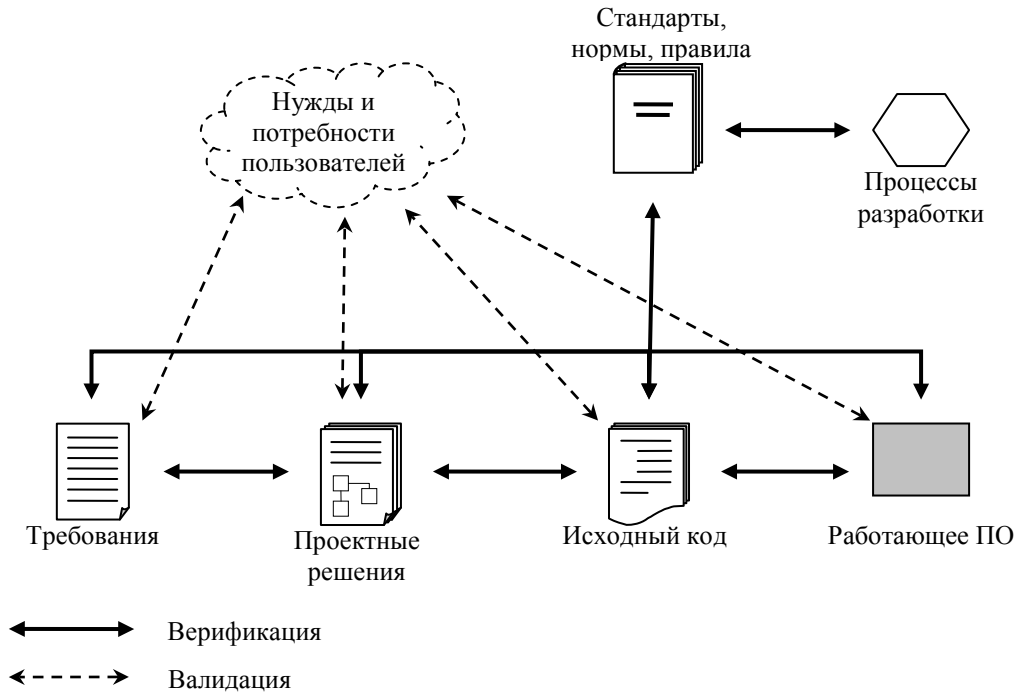


Рисунок 2. Соотношение верификации и валидации.

Методы верификации существенно строже, они чаще могут быть формализованы и автоматизированы. Скорее по историческим, чем по содержательным причинам, методы верификации делятся на следующие группы [1].

- *Методы экспертизы* (*review, inspection*). При экспертизе верификацию проводит человек, обладающий значительным опытом проведения такого рода проверок (часто также в экспертизу включаются неопытные сотрудники с целью их обучения). Экспертиза бывает общей, нацеленной на выявление любых дефектов и ошибок, или специализированной, направленной на оценку отдельных характеристик качества (например, гибкости архитектуры, удобства использования или защищенности ПО). Обычно в качестве видов экспертиз выделяют организационные экспертизы (management review), технические экспертизы (technical review), сквозной контроль

(walkthrough), инспекции (inspection) и аудиты (audit).

Экспертиза применима к любым свойствам ПО и любым артефактам жизненного цикла и на любом этапе проекта, хотя для разных целей могут использоваться разные ее виды. Она позволяет выявлять практически любые виды ошибок, причем делать это на этапе подготовки соответствующего артефакта, тем самым минимизируя время существования дефекта и его последствия для качества итогового продукта. В то же время экспертиза весьма трудоемка, требует активного участия опытных людей и не может быть автоматизирована. Эффективность экспертизы существенно зависит от опыта и мотивации ее участников, организации процесса, а также от обеспечения корректного взаимодействия между различными участниками. Это накладывает дополнительные ограничения на распределение ресурсов в проекте и может приводить к конфликтам между разработчиками, если руководство проекта обращает мало внимания на коммуникативные аспекты проведения экспертиз

- *Методы статического анализа* (*static analysis*). Такие методы используют автоматический анализ кода, моделей или других документов разработки с целью проверки выполнения формализованных правил их оформления (синтаксической корректности) и поиска часто встречающихся ошибок по определенным шаблонам (разыменования нулевых указателей, обращения к неинициализированным данным, деления на 0, взаимной блокировки параллельных процессов и пр.).

  Статический анализ выполняется с помощью специализированных инструментов, техники статического анализа кода, которые достигли достаточной зрелости и поддерживаются эффективными алгоритмами, чаще всего включаются в состав компиляторов. Однако, статический анализ обычно способен выявлять лишь ограниченный набор видов ошибок.

  Основной проблемой многих техник статического анализа является следующая дилемма: либо используются строгие методы анализа, не допускающие пропуска ошибок (тех типов, что ищутся), но приводящие к большому количеству сообщений о возможных ошибках, которые таковыми не являются (ложные сообщения об ошибках), либо набор сообщений об ошибках является точным, но возникает возможность пропустить ошибку. Обычно используются компромиссные решения, позволяющие обнаруживатькак можно больше ошибок, но допускающие не слишком высокий процент ложных сообщений. Для выявления ложных сообщений об ошибках привлекаются достаточно опытные разработчики, поэтому высокий процет таких сообщений означает большие трудозатраты на анализ результатов работы инструментов. В тех случаях, когда определенная техника статического анализа включается в компилятор, часто прибегают к объявлению всех обнаруженных с ее помощью проблем ошибками (даже если при их наличии программу можно скомпилировать в работоспособный код).

- *Методы динамического анализа* (*dynamic analysis*). Эти методы выполняют верификацию реальной работы ПО или работы его кода или исполнимой модели (из которой код получают автоматизированной трансляцией) в специализированном окружении (например, при отсутствии нужного оборудования или внешних систем производится их эмуляция). При этом собирается информация о результатах работы в различных ситуациях, и эта информация далее подвергается анализу на предмет соответсвия требованиям или проектным решениям. Обычно к таким методам относят *тестирование*, при котором работа ПО проверяется на заранее выбранном наборе ситуаций; *мониторинг*, в рамках которого работа ПО протоколируется и оценивается при его обычной или пробной эксплуатации; а также *профилирование*, специфический вид мониторинга временных характеристики работы ПО и использования им отдельных ресурсов. К динамическим методам анализа относят также и имитационное тестирование и имитационный мониторинг, при которых ПО

выполняется в рамках какого-то модельного окружения, построенного с использованием симуляторов и эмуляторов.
Для применения динамических методов необходимо иметь работающую систему или хотя бы некоторые ее компоненты, или же их прототипы, поэтому нельзя использовать их на первых стадиях разработки. Зато с их помощью можно контролировать характеристики работы системы в ее реальном окружении, которые часто невозможно аккуратно проанализировать с помощью других подходов. Динамически методы позволяют обнаруживать в ПО только ошибки, проявляющиеся при его работе, а, скажем, дефекты удобства сопровождения найти не помогут, однако, обнаруживаемые ими ошибки обычно считаются более серьезными. Еще одна проблема динамических методов — получаемые с их помощью результаты сильно засисят от полноты и разнообразия возникающих в ходе выполнения проверяемой системы ситуаций. Если набор исследованных ситуаций недостаточно широк, выводы о качестве системы, сделанные только на основе их выполнения, будут необоснованы. Методы статического анализа этим недостатком не обладают — они позволяют исследовать даже самые запутанные ситуации, которые крайне нелегко воспроизвести при работе проверяемой системы.

- *Формальные методы верификации* (*formal verification*) используют формальные математические модели проверяемого артефакта и того, на соответствие которому проводится проверка. Чаще всего это модели, соответственно, кода или проектных решений и требований. К формальным методам относятся *дедуктивный анализ* и *проверка моделей*. В рамках дедуктивного анализа свойства кода или проектных решений, а также требования формализуются в виде наборов логических утверждений (могущих включать элементы логики высших порядков и различных алгебраических структур), и из первого набора I (решения) пытаются формально вывести второй S (требования), т.е., пытаются строго доказать выводимость $I \vdash S$. Проверка моделей основана на представлении проверяемых решений в виде автоматных моделей M, а требований к ним — в виде наборов логических утверждений S (обычно с элементами временной или модальной логики). Далее выполняется автоматическая проверка выполнения $M \models S$ этих утверждений на полученных моделях с помощью специализированных инструментов. При этом обнаруживаемые ошибки могут быть сразу же оформлены как контрпримеры, конкретные сценарии работы проверяемой модели, при которых сформулированные требования нарушаются.
Формальные методы применимы только к тем свойствам, которые выражены в рамках некоторой математической модели, а также к тем артефактам, для которых можно построить адекватную формальную модель. Для использования таких методов необходимо затратить значительные усилия на построение формальных моделей. К тому же, построить такие модели и провести их анализ могут только специалисты по формальным методам, которых не так много, и чьи услуги стоят достаточно дорого. Построение формальных моделей нельзя автоматизировать, для этого всегда необходим человек. Анализ их свойств в значительной мере может быть автоматизирован, и сейчас уже есть инструменты, способные анализировать формальные модели промышленного уровня сложности, однако чтобы эффективно пользоваться ими часто тоже требуется очень специфический набор навыков и знаний (в специфических разделах математической логики и алгебры).
Тем не менее, в ряде областей, где последствия ошибки в системе могут оказаться чрезвычайно дорогими, формальные методы верификации активно используются. Они способны обнаруживать сложные ошибки, практически не выявляемые с помощью экспертиз или тестирования. Кроме того, формализация требований и проектных решений возможна только при их глубоком понимании, и поэтому вынуждает провести тщательнейший их анализ, для чего часто необходима

совместная работа специалистов по формальным методам и экспертов в предметной области.

В последние 15-20 лет появились основанные на формальных методах инструменты, решающие ряд задач верификации функциональных требований, но зато способные эффективно работать в промышленных проектах.

Гораздо чаще, чем к программам, формальные методы верификации на практике применяются к аппаратному обеспечению. Их использование в этой области имеет более долгую историю, что привело к созданию более зрелых методик и инструментов. Это обусловлено более высокой стоимостью ошибок для аппаратного обеспечения, более однородной его структурой и более простыми базовыми элементами моделей аппаратного обеспечения, более широким многократным использованием проектной информации.

Приведенная классификация методов верификации не является полностью строгой. За последнее десятилетие активно развиваются инструменты верификации ПО, использующие техники, относящиеся одновременно к нескольким из перечисленных пунктов. В качестве примеров таких методов можно привести следующие: расширенный статический анализ, построение полных тестов на основе формальных моделей, синтетическое структурное тестирование.

Тестирование на основе формальных моделей [2,3] использует для построения тестов формальные модели требований к ПО и принятых проектных решений. В рамках тестов проверяются ограничения, описанные в моделях, а критерий полноты, на базе которого строится набор тестов, обычно использует структуру модели и имеет формальное обоснование, позволяющее утверждать, что при соблюдении определенных гипотез (обычно касающихся представимости реального поведения в выбранном формализме, ограниченности реализации, т.е., отсутсвия в ней неограниченно сильно отличающихся от модели элементов поведения, и ее однородности, т.е., отсутсвия в ней большого количества незатрагиваемых моделью деталей) такой набор тестов действительно способен выявить все ошибки (все отклонения в поведении тестируемой реализации от модели).

Расширенный статический анализ [4] (extended static checking) использует формальные спецификации поведения отдельных функций и операций и дедуктивный анализ для проверки выполнения этих спецификаций кодом.

Синтетическое структурное тестирование [5-8] использует элементы формальных спецификаций и статического анализа для генерации тестов, на основе результатов работы которых, опять же, может быть проведен более глубокий статический анализ. Наиболее широко известным представителем этой группы техник является метод DART (Directed Automated Random Testing) [7,8].

Определение тестирования

Для тестирования в литературе можно найти много определений, выделяющих несколько различающиеся понятия, например, следующие.

- Процесс выполнения программы с намерением найти ошибки (Glenford J. Myers [9]).
- Любая деятельность, направленная на оценку некоторых характеристик программного обеспечения и проверку того, что они соответствуют требуемым результатам (William C. Hetzel [10]).
- Процесс анализа программного обеспечения с целью обнаружения расхождений между его реальными и требуемыми свойствами и оценки его характеристик [11].
- Это не деятельность, а дисциплина ума, позволяющая получить программное обеспечение с низкими рисками небольшими усилиями (Boris Beizer [12]).

- Процесс выполнения программного обеспечения в определенных условиях, наблюдения или протоколирования результатов его работы и оценки некоторых его аспектов [13].
- Процесс выполнения программного обеспечения для проверки того, что оно удовлетворяет специфицированным требованиям, и для нахождения ошибок [14]
- Технический анализ программного обеспечения, проводимый эмпирически, для получения информации о его качестве с точки зрения определенного круга заинтересованных лиц (Cem Kaner [15]).
- Проверка соответствия между реальным и ожидаемым поведением программного обеспечения в конечном наборе ситуаций, выбранных определенным образом [16].
- Формальный процесс, выполняемый специализированной командой, при котором поведение программного обеспечения проверяется при его выполнении на компьютере в рамках утвержденных тестовых процедур и вариантов [17].
- Процесс, состоящий из всех деятельностей жизненного цикла, статических и динамических, связанных с оценкой программного обеспечения и относящихся к нему рабочих материалов (а также планированием и подготовкой этой оценки) для определения их соответствия заданным требованиям, демонстрации их пригодности для поставленных целей и обнаружения дефектов [18].

Большинство этих определений пытаются описать понятие тестирования вне контекста других возможных методов верификации, и часто в результате получается само понятие верификации.

Для целей этого курса наиболее подходящим является определение из SWEBOK [16], указанное в восьмом пункте списка. Мы будем использовать его далее в немного уточненном виде.

Тестированием называется проверка соответствия поведения проверяемой системы требованиям, выполняемая по результатам реальной работы этой системы в некотором конечном наборе специально созданных ситуаций [16]. При этом проверяемая система обычно называется *тестируемой системой* (system under test или SUT по-английски).

В этом определении можно отметить следующие аспекты.

- ***Проверка соответствия требованиям.***
  Тестирование в этом смысле возможно лишь при наличии требований к программе. Если от системы ничего не требуется, она может делать все, что угодно, и это нельзя считать неправильным.
  Тестирование позволяет проконтролировать качество программной системы ровно настолько, насколько полно, четко и недвусмысленно определены требований к ней. В тех случаях, когда явные требования не сформулированы, тестирование можно использовать, только основываясь на некоторых неявно подразумеваемых, но желательных свойствах тестируемой системы, например, отсутствии сбоев в ее работе. Однако бывают ситуации, в которых системы определенного типа наоборот, должны демонстрировать сбои (встроенные системы или базовое ПО в определенных случаях должны просто падать, поскольку попытка обработки в значительной мере некорректных данных может существенно снизить общую эффективность таких систем). Поэтому даже в тех случаях, когда пытаются тестировать что-то, не описывая проверяемые требования явно, их выявление и четкая формулировка позволяют полнее понять, что именно и в каких случаях нужно проверить и сэкономить массу усилий при дальнейшем сопровождении и развитии проверяемой системы.

- ***Результаты реальной работы системы.***
  Тестирование основывается на результатах реальной работы тестируемой системы. Проверяемая программа должна выполняться, чтобы проводимую при этом проверку можно было считать тестированием.
  По сути, это означает, что тестирование является разновидностью динамического анализа.
  Использование реальной работы тестируемой системы позволяет применять тестирование для проверки корректности поведения системы в ее рабочем окружении, на месте ее эксплуатации, что невозможно сделать при помощи других методов контроля качества ПО.
  Бывает, однако, имитационное тестирование, при проведении которого основные действия (определение проверяемых совйств, выбор критериев полноты, разработка тестов, выполнение тестов, анализ результатов) примерно такие же, как и при обычном тестировании, но проверяется работа не самой системы, а какой-то ее исполнимой модели или прототипа. Важно, что и в этой ситуации должна использоваться модель, способная исполняться (на симуляторе или виртуальной машине).

- ***Специально созданные ситуации.***
  Тестирование всегда выполняется в специально созданных ситуациях.
  Такие ситуации называются *тестовыми ситуациями*, а процедура или программа, при выполнении которой создается одна или несколько тестовых ситуаций и проверяется правильность поведения системы в них, называется *тестом*. Подготовка к проведению тестирования всегда включает подготовку или разработку тестов.
  Составляющие тестовых ситуаций будут рассматриваться в следующих лекциях, а различным методам разработки тестов посвящено основное содержание этого курса.
  Эта характеристика отличает тестирование от пассивного наблюдения за поведением системы или *мониторинга* (passive testing, runtime verification), при котором собираются и проверяются результаты реальной работы программы, но эта работа не управляется, не направляется на возникновение определенных ситуаций.

- ***Конечный набор ситуаций.***
  Тестирование всегда выполняется в конечном наборе ситуаций. Более того, возможное количество ситуаций, возникающих во время тестирования, ограничивается практическими соображениями достижения приемлемого компромисса между затратами времени и ресурсов проекта на разработку тестов и тестирование и пользой от него — количеством обнаруживаемых ошибок, полнотой и адекватностью получаемой оценки качества тестируемой системы.
  Практически значимые системы сейчас настолько сложны, что количество ситуаций, которые необходимо испытать для полной проверки одной такой системы, превосходит возможности сколь угодно щедро финансируемого проекта и потребует не одной человеческой жизни для выполнения. Поэтому полное тестирование, хотя и возможно теоретически, в силу конечности любой вычислительной системы, практически совершенно невыполнимо.
  С одной стороны, это означает, что проведение тестирования никогда не дает полной гарантии корректности тестируемой системы, полного отсутствия в ней ошибок. С другой стороны, это обстоятельство делает чрезвычайно важным выбор тестов для выполнения. За счет выбора тестов можно получить как большой набор, выполняющийся долго и не дающий существенной информации о качестве тестируемой системы, так и компактный, но проверяющий большое количество разнообразных аспектов поведения системы и позволяющий оценить ее качество достаточно адекватно (хотя и без абсолютных гарантий).
  Критически важно для правильного выбора тестов использовать адекватный, отражающий реальную ситуацию *критерий полноты тестирования*. Различные

способы определения таких критериев и методы выбора тестов в соответствии с ними также рассматриваются в следующих лекциях данного курса.

- Тестирование — это разновидность именно верификации, а не просто анализа. Отличие в том, что анализ дает какие-то результаты в виде чисел или качественных характеристик, а верификация (в данном случае) должна ответить на вопрос о соответствии или несоответствии требованиям. В результате проведения тестирования важно получить ответ в виде «поведение системы в таких-то и таких-то ситуациях неправильно, не соответствует требованиям», а не просто набор числовых характеристик, которые дальше нужно интерпретировать отдельно.
- Ряд специфических видов тестирования, прежде всего, тестирование удобства использования, не укладываются в данное определение. Это связано с тем фактом, что для большинства систем получить аккуратное и полное описание требований к удобству использования невозможно — большинство таких требований остаются неявными, их извлечение и формализация требуют слишком больших усилий. Поэтому для контроля удобства использования используют специфическое тестирование, в виде выполнения ряда сценариев определенным образом отобранными пользователями проверяемой системы, в ходе которого протоколируются их действия и проблемы, связанные с восприятием информации от системы и поиском нужных для выполнения очередной задачи элементов интерфейса. Источником информации для выявления проблем здесь являются не отдельно сформулированные требования, а само поведение пользователей — т.е., это разновидность валидации, а не верификации. Организация такого тестирования сильно отличается от обычной, поэтому в данном курсе эта его разновидность не рассматривается.

Перечисленные аспекты определяют как достоинства, так и недостатки тестирования по сравнению с другими методами контроля качества программного обеспечения. В отличие от аналитической верификации, тестирование не может гарантировать полного отсутствия ошибок в коде системы, но зато может проверить корректность ее работы на месте эксплуатации, при взаимодействии с другими системами, что сделать при помощи аналитической верификации крайне тяжело. Тестирование всегда ограничено по ресурсам, но и предоставляет гибкие возможности управления полнотой и объемом проводимых проверок за счет разнообразных техник разработки и выбора тестов.

Для успешного проведения тестирования огромное значение имеют требования к тестируемой системе, использовавшиеся в ходе тестирования тестовые ситуации и критерии полноты тестирования. В общем случае и требования, и критерии полноты представляются в виде некоторых *моделей* (не обязательно полностью формальных), на основе которых выбираются тесты и выполняются проверки. Даже при отсутствии явных таких моделей, они присутствуют неявно, в сознании проектировщиков тестов и тестировщиков всегда есть какие-то критерии проверки правильности поведения тестируемой программы, представляющие требования, и критерии продолжения или прекращения тестирования после получения ряда результатов, основанные на понимании его неполноты или достаточной полноты. Поэтому можно сказать, что *тестирование всегда проводится на основе некоторых моделей*, быть может, не представленных явно.

Тестирование на основе моделей, которому посвящен этот курс, отличается только тем, что *все используемые при разработке, выборе и выполнении тестов модели формулируются явно*. Наиболее важную роль среди моделей, используемых при тестировании, играют модели требований, описывающие желательное поведение системы, и модели ситуаций или критерии полноты тестирования, определяющие набор разрабатываемых или выбираемых для выполнения тестов.

Литература

[18] В. В. Кулямин. Методы верификации программного обеспечения. 2008.

[19] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner, eds. *Model Based Testing of Reactive Systems*. LNCS 3472, Springer, 2005.

[20] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[21] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe. *Extended static checking.* Technical Report SRC-RR-159, Digital Equipment Corporation, Systems Research Center, 1998.

[22] C. Csallner, Y. Smaragdakis. *DSD-Crasher: A hybrid analysis tool for bug finding*. Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 245-254. ACM, July 2006.

[23] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. *Feedback-Directed Random Test Generation*. Proc. of International Conference on Software Engineering, pp. 75-84, 2007.

[24] P. Godefroid, N. Klarlund, K. Sen. DART: Directed Automated Random Testing. ACM SIGPLAN Notices - Proceedings of PLDI 2005, 40(6):213-223, 2005.

[25] K. Sen, G. Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. LNCS 4144:419-423, Spriger, 2006.

[26] Glenford J. Myers. Software Reliability. Principles and Practices. Wiley, 1976.

[27] William C. Hetzel. The Complete Guide to Software Testing. Wiley, 1984.

[28] IEEE Standard 829: Standard for Software Test Documentation. IEEE, 1983.

[29] B. Beizer. Software Testing Techniques. 2nd edition. Int. Thomson Publishing, 1990.

[30] ANSI/IEEE 610.12-1990. Glossary of Software Engineering Terminology. NY:IEEE, 1987.

[31] BCS 7925-1. Glossary of terms used in Software Testing. 1995.

[32] Cem Kaner. Black box testing course. 1999.

[33] IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004.

[34] Daniel Galin. Software Quality Assurance. From theory to implementation. 2004.

[35] ISTQB Glossary of Terms used in Software Testing. 2006.

Тестирование на основе моделей

В. В. Кулямин

Лекция 2. Основные задачи и виды тестирования

В прошлой лекции тестирование было определено как проверка соответствия поведения программной системы требованиям, выполняемая по результатам реальной работы этой системы в некотором конечном наборе заранее определенных ситуаций [1].

Таким образом, основная задача тестирования — проверка требований. Ее решение чаще всего сводится к последовательному решению ряда промежуточных задач. Кроме того, для решения этих задач в различных обстоятельствах необходимы разные подходы, определяемые технической спецификой и внешними целями — для чего именно тестирование нужно в контексте заданного проекта. Данная лекция посвящена основным целям и задачам тестирования, различным способам его проведения.

Цели тестирования

Тестирование является одним из видов деятельности жизненного цикла программного обеспечения и в рамках всего процесса разработки и сопровождения ПО может использоваться для достижения нескольких целей.

- Наиболее очевидная цель тестирования — ***поиск ошибок***, то есть расхождений между наблюдаемым поведением ПО и требованиями к нему.
  Это наиболее понятная и простая цель, которая первой приходит на ум при разговоре о тестировании.
  Даже весьма опытные разработчики ПО часто считают, что это — его единственная цель. Классическая фраза Дейкстры «тестирование может использоваться для демонстрации наличия ошибок в программе, но не может использоваться для демонстрации их отсутствия» часто интерпретируется именно в том смысле, что единственная польза от тестирования — найденные ошибки.
- Менее очевидная и более сложная для понимания цель тестирования — ***общая оценка качества ПО***. Помимо прямого обнаружения ошибок, оно должно давать информацию о том, где, скорее всего, находятся еще не найденные ошибки, насколько они могут быть серьезны, насколько тестируемая система надежно и корректно работает в целом, насколько корректны и стабильны ее отдельные функции и компоненты.
  Если единственной целью тестирования считать нахождение ошибок, то терпеливое, аккуратное и систематическое тестирование, в ходе которого ошибок не обнаруживается — бесполезно. Это не так, просто потому, что результатом такой работы является ценная информация о свойствах системы, недоступная из других источников, если, конечно, не относиться серьезно к уверенности разработчиков в собственной непогрешимости (обычно считается, что «уж в основном-то все правильно, так, мелочи всякие могут еще не работать»). Такое тестирование, *если оно действительно аккуратное и систематическое*, показывает, что тестируемая система — достаточно хороша. Даже если в ней и есть ошибки (что их нет, доказать при помощи тестирования действительно невозможно), они достаточно редки и случаются в очень специфических ситуациях. Однако проведение подобного тестирования — непростая задача, основная трудность которой — достижение такого уровня аккуратности и систематичности, который позволяет обоснованно делать соответствующие выводы.
  Другой возможной проблемой при понимании тестирования как деятельности, нацеленной только на поиск ошибок, является отсутствие рациональных аргументов для определения момента, когда нужно прекратить тестирование. Если тестировщики занимаются только поиском ошибок, они не могут предоставить руководству проекта обоснованную информацию о том, что «в системе еще очень много недоделок» или,

наоборот, «система уже достаточно надежна». В этом случае решение об остановке тестирования принимается только на основе наличия ресурсов, то есть по остаточному принципу: есть еще есть время и деньги — продолжаем тестировать, нет — прекращаем. Такого рода решения не связаны с реальным качеством системы, и поэтому способны принести большой урон репутации организации-разработчика ПО. Кроме того, они закрепляют плохую практику управления проектами, способствуя перемещению все большей и большей части ресурсов проектов в разработку без тестирования, ведь от тестирования «нет прямой пользы» и его можно проводить на «остатки» ресурсов.

Если руководитель проекта принимает подобные решения, это может означать либо, что он недостаточно компетентен и не воспринимает поступающую от тестировщиков информацию, либо, что тестировщики недостаточно компетентны, чтобы такую информацию предоставить, и тестирование проводится хаотичным, недостаточно эффективным образом, несмотря даже на то, что оно выявляет важные ошибки, — ведь остается непонятным риск наличия еще невыявленных ошибок.

Методом проб и ошибок на практике были выработаны рекомендации по организации тестирования, согласно которым ресурсы для проведения разработки тестов и тестирования необходимо выделять заранее. Их размер варьируется от 20% ресурсов всего проекта при создании достаточно простых систем до 80-90% при разработке критически важных и сложных систем реального времени с жесткими требованиями к их надежности и качеству в целом.

- Еще более редко упоминаемая и долговременная цель тестирования — ***обеспечение стабильного развития системы***, предоставление средств для отслеживания изменений в ней и предсказания возможных проблем системы после внесения в нее тех или иных изменений.

  Хорошие наборы тестов на практике становятся не только средством оценки качества тестируемой системы и сертификации ее пригодности для определенных задач, но и инструментом отслеживания важных изменений при развитии системы и измерения ее общей стабильности и надежности. Для этого, однако, необходимо поддерживать набор тестов в работоспособном состоянии, соответствующем текущим требованиям к системе. При наличии нескольких поддерживаемых версий системы необходимо иметь либо соответствующие версии тестового набора, либо средства его настройки, позволяющие тестировать функциональность, специфичную для каждой из версий. Для сложных систем это возможно только при использовании специфических образцов организации тестов, о которых пойдет речь в следующей лекции.

Задачи тестирования

Чтобы достигать указанных целей управляемым и предсказуемым образом, необходимо уметь решать ряд задач. Эти задачи специфичны для тестирования как особого вида деятельности и всегда должны решаться при его проведении, тем или иным способом. Однако прежде, чем формулировать их необходимо проанализировать точный смысл ряда терминов, повсеместно используемых при разработке тестов и тестировании.

Как уже говорилось, тестирование всегда проводится в некотором конечном наборе заранее определенных ситуаций. Такие ситуации называются *тестовыми ситуациями* и обычно включают несколько элементов.

- Набор внешних воздействий, которые оказываются на систему в такой ситуации. Эти воздействия называют *тестовыми воздействиями*. Примерами тестовых воздействий могут служить вызов интерфейсной функции системы или метода одного из ее объектов, нажатие на кнопку графического интерфейса пользователя, выполнение команды в командной строке, передача сообщения системе по одному из каналов, по которым она такие сообщения ожидает. Обычно воздействия связаны с некоторыми данными, которые они передают в систему — это, например, аргументы вызванной функции или

метода, опции выполненной команды, содержимое полей редактирования формы, где нажимается кнопка, данные пересланного в систему сообщения. Эти данные называют *тестовыми данными*.

- *Внутреннее состояние* системы в этой ситуации. Чаще всего внутреннее состояние сложных программных систем не наблюдаемо полностью извне и не может быть непосредственным образом установлено в определенное значение. Однако для обеспечения полноты проводимых во время тестирования проверок, нужно как-то управлять внутренним состоянием, чтобы иметь возможность проверить поведение системы в разных состояниях.

  Можно заметить, что такая система оказывается в разных состояниях при выполнении разных последовательностей действий. Поэтому для управления внутренним состоянием во время тестирования используют различные последовательности тестовых воздействий, называемые *тестовыми последовательностями*. Таким образом, задача приведения системы в заданное состояние сводится к построению приводящей туда последовательности воздействий.

  Например, при тестировании класса, реализующего очередь каких-либо объектов, необходимо проверить корректность операций добавления и удаления объекта не один раз, а в рамках разных последовательностей, в которых эти операции выполняются при разных заполнениях очереди.

  В еще более сложных случаях, например, при тестировании распределенных систем, состоящих из нескольких работающих параллельно компонентов, состояние, от которого зависят результаы работы в данной ситуации, может включать в себя не только внутренние состояния отдельных компонентов, но и текущее состояние среды передачи сообщений между ними — в каком (необязательно согласованном) состоянии находятся общие данные, какие сообщения уже переданы полностью, какие находятся в стадии приема, какие лежат в буферах для передачи, т.е., уже отправлены, какие еще только передаются в буфер, а какие еще не переданы. Для создания различных внутренних состояний такой системы уже недостаточно последовательностей воздействий — в таких случаях применяются более сложные комбинации воздействий, состоящие, например, из нескольких последовательностей, подаваемых на отдельные параллельно обрабатываемые входы системы (в общем случае получается частично упорядоченное множество воздействий, т.е., между некоторыми воздейсвиями есть порядок, определяющий, какое из них поступило в систему раньше, а между некоторыми парами такого порядка нет).

  Важное отличие состояния системы, работающей детерминировано в рамках одного потока управления, от состояния системы, в которой есть несколько параллельно работающих потоков, в том, что в первом случае, даже если мы не можем наблюдать внутреннее состояние системы, мы можем эффективно его котролировать за счет подачи последовательностей воздействий, т.е., текущее состояние модет быть однозначно полностью вычислено на основе истории. Во втором случае мы не можем ни наблюдать текущее состоянии, ни контролировать его — при подаче одних и и тех же наборов воздействий на внешние входы, даже при соблюдении определенных временных интервалов между ними, в общем случае невозможно предсказать в каком точном состоянии окажется система. Это зависит еще и от внутренних процессов, и от точных моментов времени, в которые она воспринимает те или иные воздействия (а точные такие моменты контролировать извне обычно невозможно, всегда есть некоторая погрешность).

- В тестовую ситуацию входят и *внешние условия*, воспринимаемые системой самостоятельно, без оказания на нее специальных воздействий, и влияющие на ее работу в данной ситуации.

  Например, в автоматическую систему кондиционирования здания могут входить датчики температуры и влажности в помещениях самого здания и на улице. При ее

тестировании не достаточно просто проверить, что команды с пультов управления выполняются ею правильно, но и удостовериться, что это так при различных значениях температуры и влажности, при которых реакции системы на команды должна будет изменяться.

Адекватное моделирование различных внешних условий часто очень сложно. Ведь при этом нужно не только создавать определенные условия (что иногда трудно, например, в требованиях к системе могут быть определены специфические ограничения на ее работу при очень низких температурах или высоком уровне радиации), но иногда и моделировать различные сценарии их изменения, которые, однако, должны быть реалистичны — годятся не произвольные сценарии, а те, что могут случиться на самом деле. Например, температура в комнате обычного здания не может меняться за 10 секунд от -100°C до +100°C, а, скажем, температура элемента внешней облочки искусственного спутника вполне может иногда так себя вести. Или, при тестировании системы управления движением автомобиля можно смоделировать его движение по горячему песку или по обледеневшему асфальту, но вряд ли стоит проверять работоспособность этой системы на дороге, где горячий песок и обледеневший асфальт то и дело сменяют друг друга.

В ряде случаев кроме тестовой системы, осуществляющей тестирование, никакие другие факторы не могут воздействовать на тестируемую систему. Это случай изолированного тестирования, в котором тошько тесты оказывают воздействия на систему и контролируют (на сколько могут) ее поведение, других субъектов контроля нет. Во многих других ситуациях можно смоделировать внешние условия специальными воздействиями, например, отделить от системы датчики температуры и вместо них подавать на соответствующие входы сигналы, соответствующие модельным значениям температуры. Еще в ряде систем все значимые внешние факторы представляются в виде конфигурационных параметров, значения которых определяются в конфигурационных файлах и базах данных при развертывании системы. Во всех этих случаях все значимые внешние условия либо могут быть сымитированы программно, либо могут управляться достаточно простым образом, и тестирование может быть выполнено программными тестами за счет сведения внешних условий к специфическим воздействиям.

Если же часть значимых условий не может управляться через программный интерфейс или информацию, заносимую в файлы и базы данных системы, необходимо специальное оборудование и существенные затраты ресурсов для их моделирования при тестировании. При этом создаются достаточно дорогие имитационные стенды для тестирования поведения системы в различных внешних условиях.

Ситуации, в которых будет проверяться поведение тестируемой системы, и сами выполняемые проверки обычно формализуют и представляют в таком виде, чтобы их мог выполнить любой человек, желательно даже незнакомый с предметной областью и задачами системы, или, еще лучше, компьютер.

Программа или четко описанная процедура, при выполнении которой создается одна или несколько тестовых ситуаций и проверяется правильность поведения системы в этих ситуациях, называется *тестом*. Важно, что тест включает в себя не только инструкцию по достижению определенной ситуации, но и инструкцию по проверке того, что проверяемая система отработала в этой ситуации правильно. Тестирование обычно организуется как выполнение некоторого набора тестов, который так и называется — *тестовый набор*.

Тестовые наборы чаще всего создаются заранее, до проведения тестирования, при разработке тестов. Иногда, однако, используются техники генерации тестов уже в процессе самого тестирования, когда никакого заранее подготовленного набора тестов нет. Такая генерация, тем не менее, всегда основана на каких-то правилах, использующих определенные данные об устройстве тестируемой системы и требованиях к ней. В этих случаях разработкой тестов естественно считать разработку этих правил.

Чтобы уметь целенаправленно и предсказуемым образом создавать полноценные наборы тестов, необходимо уметь решать следующие задачи.

- *Проверка выполнения требований.*
- *Определение критериев полноты тестирования.*
- *Построение полного набора тестовых ситуаций.*
- *Создание отчетов с информацией о результатах тестирования.*
- *Организация тестового набора для обеспечения удобства его модификации, выполнения и анализа получаемых результатов.*

Обсуждению нескольких из этих задач посвящены следующие разделы данной лекции. Критерии полноты тестирования подробно рассматриваются в следующей лекции, а различные техники построения полных наборов тестов являются основным содержанием всех дальнейших лекций.

Проверка выполнения требований

На практике решение этой задачи очень часто затрудняется отсутствием документов с понятным, однозначным, непротиворечивым и полным описанием требований. Чаще всего при разработке тестов приходится заодно уточнять требования к тестируемой системе и дорабатывать представляющие их документы, делая их более ясными и полными, а также устраняя имеющиеся противоречия.

Для построения систематичных и корректных тестов нужно адекватное понимание требований, то есть понимание того, что именно они означают, что из этого может быть проверено и, наконец, что именно должно быть проверено в каждой конкретной тестовой ситуации.

Рассмотрим, например, функцию abs(x), вычисляющую абсолютную величину числа. Вроде бы ясно, что при этом вычисляется — должен возвращаться x, если он неотрицателен, или –x иначе. На языках C, C++, Java или C# это может быть передано так: (x >=0)? x : -x.

Можно, например, проверять, что abs(x) >= 0 для любого x.

Попробуем взять x = –2147483648, например, в Java. Каков будет результат?

Правильный ответ: –2147483648.

Этот неожиданный результат — отрицательное число в качестве абсолютной величины — объясняется тем, что в машинной арифметике 32-битных целых чисел выполнено соотношение –(–2147483648) = –2147483648. Приведенное выше определение abs(x) остается правильным, неверно только заключение о том, что abs(x) >= 0.

Чтобы объяснить полученное «странное» соотношение, надо понимать, чем руководствовались создатели целочисленной машинной арифметики. Им нужно было целые числа, которых бесконечно много, представить в машине как некоторое конечное множество. При этом, однако, надо сохранить основополагающие свойства арифметических действий — сложения, вычитания и умножения. Наиболее похожими на целые числа конечными множествами с таким набором операций являются множества (кольца) классов целых чисел по какому-то модулю, например Z_2 = {[0], [1]}, где [0] — класс четных чисел, а [1] — класс нечетных, или Z_3 = {[0], [1], [2]}, где [n] — класс чисел, имеющих остаток n при делении на 3. Поскольку в машине удобно представлять числа в двоичной записи, для эффективного расхода памяти стоит взять модуль равным степени 2, например 2^{32}. То есть, машинное число n будет обозначать класс чисел, равных n по модулю 2^{32}. Если к тому же хочется, чтобы действия с небольшими числами приводили к привычным результатам: 2+2 = 4, а 3–5 = –2, в качестве представителей классов чисел по модулю 2^{32} должны использоваться небольшие положительные и отрицательные числа. Таким образом, в качестве представителей удобно выбрать числа 0, 1, –1, 2, –2, 3, –3, 4, –4 и т.д. Но, поскольку всего их должно быть 2^{32}, для некоторого числа n, мы в итоге выберем n, но не выберем –n или

наоборот — ведь для 0 в этой последовательности уже нет соответствующего отрицательного числа. Соответственно, конец ее выглядит как 2147483647, –2147483647, –2147483648 = -2^{31}. По причинам, связанным с эффективностью и простотой реализации операций, проще выбрать -2^{31}, чем 2^{31} — при этом можно использовать первый бит представления числа в значении его знака, а все вычисления производить побитно, по тем же причинам не стоит вводить специальное число –0, что, например, сделано в арифметике чисел с плавающей точкой. По модулю 2^{32} выполнено $-[-2^{31}] = [2^{31}] = [2^{32}-2^{31}] = [-2^{31}]$, что и соответствует выписанному выше соотношению.

Другой пример. Как ведет себя tg(arctg(x)) при возрастающем x, которое является числом с плавающей точкой двойной точности? При $x <= 10^{16}$ все идет как ожидается: tg(arctg(x)) = x с небольшой погрешностью, а вот дальше, как бы ни было велико x, tg(arctg(x)) дает один и тот же результат $1.633123935319537 \cdot 10^{16}$. Для объяснения этого нужно знать, как устроены числа с плавающей точкой, что служащее для представления π/2 число с плавающей точкой двойной точности X = 884279719003555/562949953421312 меньше π/2 примерно на $6.1232339957367658 \cdot 10^{-17}$, и что получаемый результат является как раз значением $\mathrm{tg}(X) \approx 1/(\pi/2 - X)$, вычисленным с двойной точностью.

Двоичное число с плавающей точкой имеет следующую структуру [2,3].

- Число представлено в виде набора из *n* бит, из которых первый бит является *знаковым битом числа*, следующие *k* бит представляют его *порядок E*, а оставшиеся *(n-k-1)* бит представляют его *мантиссу M*.
- Знаковый бит *S*, порядок *E* и мантисса *M* числа *x* определяют его значение по следующим правилам.
  $x = (-1)^S \cdot 2^e \cdot m$, где
  - *S* — знаковый бит, равный 0 для положительных чисел, и 1 для отрицательных;
  - если $0 < E < 2^k - 1$, то $e = E - 2^{(k-1)} + 1$;
    иначе, если $E = 0$, $e = -2^{(k-1)} + 2$;
    число $b = 2^{(k-1)} - 1$ называется *смещением порядка (exponent bias)*;
  - если $0 < E < 2^k - 1$, то $m = 1 + M/2^{n-k-1}$. Иначе говоря, *m* имеет двоичное представление 1.*M*, т.е. целая часть *m* равна 1, а последовательность цифр дробной части совпадает с последовательностью бит *M*;
    если же $E = 0$, то $m = M/2^{n-k-1}$, или *m* имеет двоичное представление 0.*M*.
    Числа с нулевой экспонентой называются *денормализованными*, а все остальные — *нормализованными*.
- Максимальное возможное значение порядка $E = 2^k - 1$ зарезервировано для представления *исключительных чисел*: положительной и отрицательной бесконечностей, +∞ и -∞, а также специального значения NaN (not-a-number, не число). NaN используется, если результат выполняемых действий нельзя корректно представить ни обычным числом, ни бесконечностью, как, например, результаты 0/0 или (-∞) + (+∞).
  +∞ имеет нулевой знаковый бит, максимальный порядок и нулевую мантиссу; -∞ отличается только единичным знаковым битом.
  Любое число, имеющее максимальный порядок и ненулевую мантиссу, считается представлением NaN.
- Стандарты IEEE 754 и IEEE 854 определяют несколько возможных типов чисел с плавающей точкой, из которых чаще всего используются *числа однократной точности* (single precision), *числа двойной точности* (double precision) и *числа расширенной двойной точности* (double-extended precision).
  Для чисел однократной точности $n = 32$ и $k = 8$. Соответственно, для мантиссы

используется 23 бита и смещение порядка равно 127.

Для чисел двойной точности $n = 64$ и $k = 11$. Для мантиссы используется 52 бита и смещение порядка равно 1023.

Для чисел расширенной двойной точности определенные значения k и n не фиксируются в стандартах, вводятся лишь ограничения $128 \geq n \geq 80$ и $k \geq 15$. В процессорах 32-битной архитектуры Intel $n = 80$ и $k = 15$. При этом для мантиссы используется 64 бита и смещение порядка равно 16383.

Кроме этого, иногда используются числа *четырехкратной точности*, для которых $n = 128$ и $k = 15$. Соответственно, для мантиссы используется 112 бит и смещение порядка равно 16383. Они не специфицированы в стандарте, но имеют аналогичную структуру.

В качестве примера укажем представление числа -17_{10} в виде чисел однократной и двойной точности. Поскольку $-17_{10} = (-1)^1 \cdot 2^{131-127} \cdot 1.0001_2 = (-1)^1 \cdot 2^{1027-1023} \cdot 1.0001_2$, соответственно, оно представляется в виде числа однократной точности как
1 10000011 00010000000000000000000, а в виде числа двойной точности как
1 10000000011 0001000000000000000000000000000000000000000000000000.

Некоторые граничные значения таких чисел для разных форматов указаны в Таблице 1. При этом для расширенной двойной точности используются параметры 32-битной архитектуры Intel.

	Общий вид	Однократная точность	Двойная точность	Расширенная двойная точность	Четырехкратная точность
Самое маленькое денормализованное положительное число	$2^{-b-n+k+2}$	2^{-149}	2^{-1074}	2^{-16446}	2^{-16494}
Самое маленькое нормализованное положительное число	2^{-b+1}	2^{-126}	2^{-1022}	2^{-16382}	2^{-16382}
Самое большое положительное число	$2^{b-n+k+1} \cdot (2^{n-k}-1)$	$2^{104} \cdot (2^{24}-1)$	$2^{971} \cdot (2^{53}-1)$	$2^{16319} \cdot (2^{65}-1)$	$2^{16271} \cdot (2^{113}-1)$
Самое большое число, меньшее 1	$1-2^{-n+k}$	$1-2^{-24}$	$1-2^{-53}$	$1-2^{-65}$	$1-2^{-113}$
Самое маленькое число, большее 1	$1+2^{-n+k+1}$	$1+2^{-23}$	$1+2^{-52}$	$1+2^{-64}$	$1+2^{-112}$

Таблица 1. Граничные значения чисел с плавающей точкой.

Заметьте, что половину страницы текста заняло объяснение лишь небольшой «странности» поведения операции взятия противоположного целого числа — одной из простейших и наиболее точно определенных, которые только можно себе представить. Для полного и аккуратного разбора примера с арктангенсом потребовалась бы еще дополнительно пара страниц. При рассмотрении же более сложных функций программных систем, не имеющих таких точных математических аналогов, возникают гораздо более серьезные проблемы.

Именно поэтому здесь в качестве примеров взяты математические функции — любой человек, окончивший два курса ВУЗа по физико-математической или технической специальности, достаточно хорошо представляет себе, что это такое, и при желании может самостоятельно понять, как должна работать такая функция. Если же надо понять, как работают менее однозначно определенные операции, возникает такое количество возможных разнообразных смыслов, что чисто умозрительно решить, какой именно правилен, практически невозможно — необходима документация, зафиксированные в документах

требования, общение с разработчиками системы или с экспертами в данной предметной области.

Например, для адекватного понимания работы операции чтения заданного количества байт из файла нужно уметь четко отвечать на множество вопросов. Что происходит, если файла нет? Что будет, если он есть, но у данного процесса нет прав на работу с ним? Что будет результатом, если размер файла меньше запрашиваемого количества байт? А если другие операции в тоже время записывают данные в этот же файл? И пр., и т.д., и т.п.

В то же время для операций ввода-вывода, как и для большинства других операций, реализованных в рамках широко используемых библиотек, можно, после определенных усилий, достаточно строго определить математическую модель, адекватно описывающую их работу. Для многих же практических примеров — операций биллинговой системы, системы автоматического управления боевым кораблем или гидроэлектростанцией — таких моделей вообще нет, никто никогда не продумывал такие сложные системы во всех их деталях. На формальную проработку их при современных технологиях уйдет времени и усилий гораздо больше, чем это допустимо с точки зрения экономической оправданности таких систем.

Таким образом, одна из наиболее сложных задач — *адекватно понять требования*, понять, что именно обозначает каждое утверждение в документации, которое относится к данной операции. В примере с абсолютной величиной вроде бы удалось сразу написать четкое определение, но *понять* его помогает только приведенный пример «странного» поведения. Только после проведенного дополнительного анализа становится возможным без ошибок выводить следствия из этого определения (первоначальный вывод о том, что абсолютная величина больше 0 был ошибкой). Во многих других случаях даже просто написать четкое определение нелегко. Поэтому всегда необходим вдумчивый *анализ требований*, извлечение всех сведений, которые только можно получить из документации, стандартов, а также из личного общения с экспертами, архитекторами, разработчиками и пользователями тестируемой системы и другими заинтересованными лицами.

Помогают в этом анализе попытки четко определить, как можно проверить требования. Например, попытка проверить правильность вычисление абсолютной величины при помощи тождества abs(x) >= 0 проваливается, и этот факт вынуждает задуматься об основных принципах машинной арифметики. Формулируя требования в проверяемом виде, мы сразу получаем очень хорошее определение правильного поведения, более аккуратное, чем существующие описания для подавляющего большинства программных систем.

Если тесты создаются как обычно, в виде тестовых вариантов, их разработчик сразу пытается определить, как именно проверять требования в той конкретной ситуации, которая возникает в данном тесте. Методы тестирования на основе моделей требуют описывать требования к поведению операции «вообще», в произвольной ситуации, в которой она должна работать. При этом в качестве результата анализа требований получается некоторая ***модель требований*** — целостное, полное, непротиворечивое и точное описание того, что должна делать система. Она указывает, как проверять правильность работы системы и дает возможность разделить формулировку проверяемых требований и придумывание ситуаций для их проверки на два отдельных вида деятельности, повышая качество и того, и другого.

Отчеты о результатах тестирования

Хотя основной целью тестирования является лишь проверка соответствия требованиям, или же, наоборот, проверка наличия ошибок в тестируемой системе, тестирование, после которого остается только информация вида «ошибок нет» или «ошибки есть», практически бесполезно. На практике нужны тесты, после выполнения которых остается достаточно информации о найденных ошибках, чтобы разработчик мог локализовать их в ходе отладки с небольшими усилиями. Кроме того, нужна еще и информация о полноте выполненных тестов. В общем случае отчеты о ходе тестирования должны содержать следующее.

- Данные обо всех обнаруженных ошибках.

- Тип ошибки по некоторой классификации, который поясняет, что же произошло. Например: затребовано слишком много памяти, функция работает слишком долго, выдается некорректный результат, выполняется запись неверных данных куда-либо, не возвращается управление, разрушение процесса.
- Какая проверка зафиксировала ошибку, что именно было сочтено некорректным, какое требование при этом проверялось.
- Каков наиболее короткий выделяемый сценарий действий, который позволяет повторить эту ошибку. Иногда достаточно указания только одной неправильно сработавшей операции или функции, но в сложных случаях необходимо повторить некоторый набор действий, в совокупности приведший к некорректной работе системы. Операция, при выполнении которой ошибка проявляется, вовсе не обязательно сама содержит ошибку, она может просто использовать некорректные результаты работы других операций.

- Данные о полноте тестирования по нескольким критериям — какие тестовые ситуации возникали в ходе тестирования, а какие нет, каковы значения измерявшихся метрик тестового покрытия.
  Информацию о затронутых элементах тестируемой системы — вызываемых функциях и методах, выполняемых командах, затрагиваемых классах, компонентах, подсистемах — а также о проверяемых в ходе тестирования требованиях всегда полезно иметь, даже когда полнота тестирования определяется другими способами.

Организация тестовых наборов

Поскольку часто набор тестов представляет собой довольно сложную систему, она должна быть хорошо организована. Это особенно важно для тестов, которые планируется поддерживать, сопровождать и пополнять долгое время. При организации тестов нужно также аккуратно выделять отдельные модули, учитывая возможности их повторного использования, как и при проектировании любой другой программной системы. Таким образом, хорошо спроектированный тестовый набор всегда использует те же базовые принципы программной инженерии, что и любая хорошо спроектированная программная система — абстракцию и уточнение, модульность и многократное использование [4].

Помимо этого, хорошая организация тестовых наборов должна подчиняться дополнительным ограничениям, связанным уже с самой природой тестов.

- Тесты должны иметь дополнительную структуру, основанную на их связи с компонентами тестируемой системы. Она может быть представлена, например, в явно указанной в описании каждого теста или в комментариях к нему ссылке на соответствующие компоненты.
  Наличие такой информации облегчает сопровождение тестового набора при небольших изменениях в тестируемой системе, без изменения ее функциональности. В этих случаях становится возможным быстро найти все тесты, подлежащие корректировке в связи с такими изменениями.
  Другая выгода от наличия явных связей с тестируемыми компонентами — возможность минимизации количества повторно выполняемых тестов при небольших изменениях, которые никак не отражаются на интерфейсе системы или ее функциях. При этом можно выполнять только те тесты, которые затрагивают измененные или зависящие от измененных компоненты.
- Тесты должны быть явно связаны и с проверяемыми требованиями.
  Такая привязка позволяет, помимо оценки полноты тестового набора, снижать трудозатраты на его модификацию при изменениях в требованиях, быстро выделяя только те тесты, которые должны измениться.
  Если возникает необходимость проверить только заданное подмножество функций

системы, привязка тестов к требованиям также поможет минимизировать исполняемый тестовый набор.

- Тесты должны быть разбиты на группы по нескольким различным аспектам. Это разбиение может влиять на порядок выполнения тестов.
  - Например, полезно отделять тесты, выполнение которых полностью автоматизировано, от тех, где требуется вмешательство человека. При этом часто автоматические тесты удобно выполнять раньше, поскольку обнаруживаемые ими ошибки могут сделать выполнение тестов под контролем человека более трудным.
  - При тестировании сложных систем часто требуются достаточно сложные сценарии работы теста. Например, тест, проверяющий корректность управления потоками в операционной системе, может запускать много потоков различной конфигурации, в разных потоках использовать различные механизмы синхронизации, проверяя, что они работают корректно. Однако, если ошибка в функции создания потока просто не дает создать новый поток в программе, пытаться выполнить этот тест бессмысленно — большая его часть так и будет выполнена, а та, что будет, может просто попасть в тупик или выдать невнятную информацию о множестве обнаруженных ошибок, анализировать которую будет непросто.

    Избежать такой ситуации поможет простой тест, проверяющий только, что новый поток действительно создается. Прежде, чем выполнять сложные и запутанные тесты, использующие много функций, желательно проверить выполнение некоторых базовых требований каждой из этих функций с помощью простых тестов. При нарушении таких требований, сложный тест не стоит выполнять вообще.

    Такая организация тестов для сложных систем позволяет минимизировать совокупные затраты на выполнение тестов и анализ их результатов.
  - Если тестируемая система может иметь опциональные, необязательные функции, которые, при их наличии, тоже нужно протестировать, появляется необходимость в опциональных проверках, которые при некоторых условиях выполняются, а при других нет.

    Проще всего эта задача решается при помощи введения в тест *конфигурационных параметров*, значения которых могут управлять выполняемыми тестами и проводимыми в них проверками. Конфигурационные параметры могут настраиваться независимо и не изменяться в ходе работы теста, а могут определяться динамически, при помощи специального модуля, проверяющего наличие или отсутствие в системе опциональных функций. Нацеленные на эти функции тесты должны выполняться только после работы такого модуля.

Наиболее мощной техникой структуризации тестовых наборов является выделение модулей или компонентов, отвечающих каждый за решение своей специфической задачи.

Виды компонентов тестового набора

- Оракулы.

  Тестовым оракулом называется компонент теста, принимающий решение о том, правильно или неправильно сработала тестируемая система в данном тесте.

  Способы построения оракулов.
  - Чаще всего роль оракула выполняет человек. Он просто анализирует тестовые данные и результаты теста и сопоставляет их со своим пониманием требований.
  - Достаточно часто используются оракулы на основе таблиц тестовых данных и правильных результатов. Такой оракул находит в таблице строку, в соответствующих столбцах которой записаны использованные тестовые данные, и

проверяет, что полученный результат равен тому, который находится в столбце результатов в этой же строке.

- Если есть другая реализация той же функции, которую мы собираемся тестировать, можно построить оракул, использующий сравнение результатов, полученных от тестируемой системы, с выдаваемыми этой реализацией. Другая реализация может быть прототипной, не такой эффективной, как тестируемая, но зато правильной. В качестве другой реализации может использоваться предыдущая версия тестируемой системы, если к ней нет нареканий по поводу корректности работы.
- При наличии нескольких версий тестируемой системы или нескольких других реализаций проверяемой функции можно использовать оракул, построенный на базе голосования. Правильным результатом при этом считается тот, что вернуло большинство из имеющихся реализаций, и результат тестируемой системы сравнивается с ним.
- Если тестируемая функция имеет обратную, ее можно использовать в оракуле — вычислять по результатам входные данные и проверять их совпадение с использованными реально входными данными.
- Самый общий случай — проверка каких-то свойств результата. На основе требований формулируются ожидаемые свойства результата, и оракул проверяет именно их.

  Например, если от операции требуется возвращать список клиентов, имеющих задолженность, превышающую заданное значение, такой оракул для каждого из клиентов в полученном списке проверяет, действительно ли его задолженность превышает это значение.

- Генераторы тестовых данных.

  Генератор тестовых данных отвечает за построение входных данных, используемых тестами.

  Способы организации генераторов.

  - Пул данных. Заранее приготовленные данные сохраняются в виде коллекции или таблицы и используются по мере необходимости.
  - Другой генератор с фильтрацией. Генератор данных, удовлетворяющих некоторому условию можно организовать, используя простой генератор данных этого типа и фильтрацию по данному условию — если поставляемые простым генератором данные удовлетворяют условию, они передаются вовне, если нет — вычисляется следующая порция данных.
  - Составной генератор. Генератор данных сложной структуры строится из генераторов элементов этой структуры. Например, если нужно построить объекты, у которых есть три поля, можно по отдельности строить значения полей, а потом объединять их в объект. При объединении можно использовать декартово произведение множеств значений, то есть строить по объекту на каждую создаваемую тройку значений полей, или использовать «диагональ» — объединять только одновременно сгенерированные тройки значений полей.

    В составных генераторах данных, удовлетворяющих некоторым ограничениям целостности (например, значение первого поля всегда должно быть больше значения второго), используются фильтры.

- Тестовые адаптеры.

  Тестовый адаптер предназначен для соединения теста с тестируемой системой в тех случаях, когда интерфейс тестируемой системы не соответствует тому, на который рассчитан тест.

  Такая ситуация может сложиться, если тесты для старой версии нужно перенести на

новую версию системы, причем изменений в проверяемой функциональности мало, но есть изменения в интерфейсах — функции или команды названы по-другому, в них изменен порядок параметров или типы параметров получили другие имена.
Другая возможность — тесты создавались заранее, до разработки системы или параллельно ей, на основе требований и проектной документации, а когда была разработана сама тестируемая система, часть ее интерфейсов изменилась по сравнению с проектом.
Третий случай использования адаптеров — тестирование на соответствие некоторому общему стандарту, для которого может быть много реализаций от разных поставщиков. Такая ситуация, например, в телекоммуникационном ПО — стандарты на протоколы взаимодействия фиксированы, но есть много разных разработчиков этого ПО и их системы должны успешно взаимодействовать друг с другом. При этом тестовый набор для стандарта делается при самых общих предположениях об интерфейсе (просто наличие определенных функций и структура данных их параметров) — такой тестовый набор называется *абстрактным*. Чтобы использовать абстрактный тестовый набор для тестирования некоторой реализации, он дополняется набором адаптеров, привязывающих использованный в нем абстрактный интерфейс к конкретному интерфейсу данной реализации.

- Заглушки.
  Тестовой заглушкой называется компонент, играющий роль необходимого тестируемой системе компонента, который еще не разработан.
  При интеграционном и модульном тестировании иногда приходится тестировать отдельно компоненты, которым для работоспособности необходимы другие. Эти другие компоненты могут быть еще не готовы или же они не включаются в тест, поскольку сильно усложнили бы его. Чтобы тестируемый компонент мог работать во время тестирования, вместо отсутствующих компонентов подставляются заглушки, которые имеют такой же интерфейс, что и отсутствующие компоненты, но устроены как можно более просто, например, возвращают один и тот же результат или генерируют его случайно.

Виды тестирования

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы ПО. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты удобства сопровождения. Поэтому выделяют виды тестирования, связанные с проверкой определенных характеристик и атрибутов качества — тестирование функциональности, надежности, удобства использования, переносимости и производительности, а также тестирование отдельных атрибутов — защищенности, функциональной пригодности и пр. Кроме того, особо выделяют ***нагрузочное*** или ***стрессовое тестирование***, проверяющее работоспособность, надежность ПО и показатели его производительности в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных, и пр.

Рассматриваемые в данном курсе методы построения тестов ориентированы в большей мере на тестирование функциональности. Их можно использовать и для тестирования переносимости, производительности или надежности. Однако при их использовании нужно иметь в виду следующее.

- Для тестирования производительности необходим дополнительный анализ факторов возможного снижения производительности — объема входных данных, объема базы данных системы, количества пользователей, количества процессов и потоков в системе, объема данных, передаваемых между компонентами системы и пр. Полнота такого

тестирования сильно зависит от используемого в тестах набора факторов. Выделение адекватного набора аспектов, влияющих на производительность, не рассматривается в данном курсе.

- При тестировании надежности должны измеряться статистические показатели работы системы и должны использоваться близкие к реальным по своим статическим свойствам входные данные. Вопросы статистического моделирования различных аспектов входных данных, поведения пользователей и других систем, которые могут оказывать влияние на работоспособность тестируемой системы, не рассматриваются в данном курсе. Также не обсуждаются методы определения статистических показателей ее работы.

Тестирование удобства использования сильно отличается от других видов тестирования, поскольку всегда связано с оценкой удобства выполняемых действий и представления их результатов в системе. По этой причине при таком тестировании всегда используются экспертные оценки, требующие вовлечения квалифицированных специалистов по удобству использования. Автоматизировать этот вид тестирования в той же мере, как другие, не удается.

На основе исходных данных, используемых для построения тестов, тестирование делят на следующие виды.

- ***Тестирование черного ящика***, нацеленное на проверку требований. Тесты для него и критерий полноты тестирования строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется ***тестированием на соответствие*** (conformance testing). Частным случаем его является ***функциональное тестирование*** — тесты для него, а также используемые критерии полноты проведенного тестирования определяют на основе требований к функциональности.
  Еще одним примером тестирования на соответствие является ***аттестационное*** или ***сертификационное тестирование***, по результатам которого программная система получает (или не получает) официальный документ, подтверждающий ее соответствие определенным требованиям и стандартам.
- ***Тестирование белого ящика***, оно же ***структурное тестирование*** — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы (например, на значения параметров вызовов, результатов и локальных переменных).
- Тестирование, при котором используются как требования, так и знания о внутреннем устройстве системы, используется на практике чаще, чем указанные выше крайние разновидности. Оно иногда называется ***тестированием серого ящика***, но термин этот упоминается реже, чем тестирование черного или белого ящика.
- Тестирование, нацеленное на определенные ошибки. Тесты для такого тестирования строятся так, чтобы гарантированно выявлять определенные виды ошибок. Полнота тестирования определяется на основе количества проверенных ситуаций по отношению к общему числу ситуаций, которые мы пытались проверить. К этому виду относится, например, ***тестирование на отказ*** (smoke testing), в ходе которого просто пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с нарочно внесенными ошибками.

Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено. Эти же разновидности тестирования можно связать с фазой жизненного цикла тестируемой системы, на которой они выполняются.

- ***Модульное тестирование*** (unit testing) предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют ***программные контракты*** — ***предусловия***, описывающие для каждой операции, на каких входных данных она предназначена работать, ***постусловия***, описывающие для каждой операции, как должны соотноситься входные данные с возвращаемыми ею результатами, и ***инварианты,*** определяющие критерии целостности внутренних данных модуля.
  Модульное тестирование является важной составной частью ***отладочного тестирования***, выполняемого разработчиками для отладки написанного ими кода.
- ***Интеграционное тестирование*** (integration testing) предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций. Интеграционное тестирование также используется при отладке, но на более позднем этапе разработки.
  При интеграционном тестировании могут использоваться различные стратегии присоединения новых компонентов к тестируемой системе.
  - При стратегии «сверху вниз» сначала тестируют модули, находящиеся на самом верхнем уровне и непосредственно взаимодействующие с пользователями или внешними системами. Затем к ним постепенно добавляют модули, вызываемые ими, выполняя тестирование после добавления каждого модуля, затем — модули следующих уровней. На каждом шаге, кроме последнего, в котором участвуют все модули системы, вместо отсутствующих модулей используются заглушки.
  - При стратегии «снизу вверх» сначала тестируются модули нижнего уровня, не зависящие от других модулей системы, затем добавляются модули, зависящие от них, и т.д., вплоть до модулей самого верхнего уровня. При этом заглушки используются редко, только в тех случаях, когда только что добавленный в тестируемую систему модуль зависит от других модулей того же уровня.

  Часто применяются смешанные стратегии — часть этапов интеграции выполняется «сверху вниз», часть — «снизу вверх».
  Метод, которым настоятельно не рекомендуется пользоваться, — проведение интеграционного тестирования сразу для всех модулей большой системы, без предварительной отладки взаимодействия внутри отдельных групп модулей.
- ***Системное тестирование*** (system testing) предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователями задачи в различных ситуациях.
  Системное тестирование выполняется через внешние интерфейсы ПО и тесно связано с ***тестированием пользовательского интерфейса*** (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса. Частными случаями этого вида тестирования являются ***тестирование графического пользовательского интерфейса*** (Graphical User Interface, GUI) и ***пользовательского интерфейса Web-приложений*** (WebUI).
  Если интеграционное и модульное тестирование чаще всего проводят, воздействуя на компоненты системы при помощи операций предоставляемого ими программного интерфейса (Application Programming Interface, API), то на системном уровне без использования пользовательского интерфейса не обойтись, хотя тестирование через API в этом случае также вполне возможно.

Особняком стоит ***регрессионное тестирование***, используемое для проверки того, что вносимые небольшие изменения и исправления ошибок не нарушают стабильность и не

снижают работоспособность системы. Регрессионное тестирование используется на этапе сопровождения, после внесения изменений и исправлений в систему, при выпуске ее очередной версии.

Литература

[1] IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004.

[2] IEEE 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. NY: IEEE, 1985.

[3] D. Goldberg. *What Every Computer Scientist Should Know about Floating-Point Arithmetic*. ACM Computing Surveys, 23(1):5-48, 1991.

[4] В. В. Кулямин. Технологии программирования. Компонентный подход. М: Интернет-университет информационных технологий — БИНОМ. Лаборатория знаний, 2007. http://www.ispras.ru/~kuliamin/lectures-sdt/Lecture01.pdf.

Тестирование на основе моделей

В. В. Кулямин

Лекция 3. Критерии полноты тестирования

Набор тестов, используемый при тестировании, всегда конечен и, более того, ограничен соображениями экономической эффективности распределения ресурсов между разными видами деятельности при разработке ПО. Поэтому крайне важно строить его так, чтобы используемые тесты проверяли как можно больше разных аспектов функциональности системы в как можно большем разнообразии ситуаций. Чтобы систематическим образом перебирать существенно отличающиеся друг от друга ситуации, используют *критерии полноты тестирования* или *критерии адекватности тестирования* [1,2]. Тестовый набор, удовлетворяющий заданному критерию полноты, называют *полным* по этому критерию.

Чаще всего для определения критерия полноты некоторые из возможных тестовых ситуаций рассматривают как эквивалентные и определяют количество классов неэквивалентных тестовых ситуаций, встретившихся или «*покрытых*» во время тестирования. Такие критерии полноты называются *критериями тестового покрытия* [1-3]. При этом определяется и числовая *метрика тестового покрытия* — доля покрытых классов ситуаций среди всех возможных. Критерий полноты может использовать различные значения метрики, например, он может требовать, чтобы полный тестовый набор всегда покрывал 100% выделенных классов ситуаций, или же считать достаточным покрытие 85% классов ситуаций. Поскольку для одной метрики покрытия можно определить много критериев полноты, далее речь, чаще всего, идет о различных метриках тестового покрытия.

Полноту тестирования можно определять по-разному, но в основе любого критерия полноты лежит представление о возможных ошибках в тестируемой системе. Различные способы классификации ситуаций, отражающие их разнообразие с точки зрения тестирования, перечислены ниже. Каждый из них и любое их подмножество совместно могут использоваться для определения метрик тестового покрытия. Классифицировать ситуации можно следующим образом.

- На основе *структурных элементов тестируемой системы*, которые выполняются или задействуются в ходе тестирования.
- На основе *структуры входных данных*, используемых во время тестирования.
- На основе *элементов требований*, проверяемых при выполнении тестов.
- На основе явно сформулированных *предположений об ошибках*, выявление которых должны обеспечить тесты.
- На основе *произвольных моделей устройства или функционирования* тестируемой системы.

Последний вид метрик покрытия является самым общим — все критерии полноты используют, так или иначе, какие-то модели системы. Дополняя такую модель некоторыми гипотезами о возможных ошибках в системе — в чем именно она может отличаться от этой модели, мы всегда получим основу для определения метрики тестового покрытия. Первые четыре вида, однако, выделены, поскольку используемые в них модели имеют четко определенную природу — это модели структуры самой системы, модели структуры ее входных данных, модели требований и модели ошибок определенного вида. К пятой группе относятся метрики, основанные на моделях, не принадлежащих к этим разновидностям.

Структурные критерии

Критерии полноты тестирования и метрики тестового покрытия, основанные на структуре тестируемой системы, называются *структурными*, а тестирование, проводимое с их использованием — *структурным тестированием*.

В основе структурных критериев полноты лежит простая идея: если ошибка находится в какой-то конструкции кода, в каком-то компоненте тестируемой системы, то выполнив эту конструкцию или заставив работать этот компонент, мы, скорее всего, сможем ее обнаружить. Соответственно, если в двух ситуациях выполняются одни и те же элементы кода, такая ошибка будет либо проявляться в обеих ситуациях, либо не проявляться ни в одной, поэтому их можно объявить эквивалентными и проверять всегда только одну из таких ситуаций.

Это предположение редко выполняется на практике, однако как эвристика для определения метрик тестового покрытия, оно достаточно полезно. Далее для некоторых конкретных структурных метрик будут приведены примеры простых программ, в которых выполнение одной и той же конструкции в некоторых случаях вскрывает ошибку, а в некоторых — нет.

Структурные метрики покрытия различаются в зависимости от размера элементов системы, используемых при их определении. Можно выделить три уровня структурных метрик — *уровень отдельной функции* или отдельного метода класса, *уровень компонента* или класса, включающего несколько операций, и *уровень подсистемы* или системы в целом, в составе которых может быть много компонентов.

Вне зависимости от уровня структурные метрики могут быть основаны на информации двух видов — на информации о передаче управления между разными исполняемыми элементами системы или на информации об использовании и записи данных. Метрики первого типа называются *основанными на потоке управления*, второго типа — *основанными на потоках данных*.

Важным достоинством структурных метрик покрытия является возможность их автоматизированного вычисления при наличии доступа к коду или схемам архитектуры тестируемой системы. Существенным недостатком является отсутствие учета требований — мы можем покрыть все элементы структуры, но не обнаружим, что какое-то требование просто забыли реализовать.

Структурные критерии на уровне отдельной функции

Структурные метрики покрытия для одной функции или метода на основе потока управления базируются на исполняемых в ходе теста элементах кода этой функции или этого метода.

Метрики покрытия на основе потока управления

Наиболее простая из таких метрик — *метрика покрытия инструкций* (statement coverage), равная доле выполненных во время тестирования инструкций кода функции по отношению ко всем ее инструкциям. Поскольку в большинстве современных языков программирования принято писать не более одной инструкции в строке, эта метрика чаще всего коррелирует с метрикой покрытия строк исходного кода. Однако всегда при разговоре о строках кода стоит уточнять, насколько они соответствуют инструкциям, потому что часть строк содержит декларативную, неисполняемую информацию, и информация об инструкциях позволяет точнее оценить ситуацию. В том случае, если часть инструкций не достижима, т.е. не может быть выполнена ни при каких условиях, долю покрытых инструкций определяют только по отношению ко всем достижимым инструкциям.

Рассмотрим следующий пример.

```
1  int gcd(int a, int b)
2  {
3    if(a == 0)
4      return b;
5    if(b == 0)
6      return a;
7    if(a > 0 && b < 0 || a < 0 && b > 0)
```

```
 8      b = -b;
 9
10    while(b != 0)
11    {
12      if(b > a && a > 0 || b < a && a < 0)
13      {
14        a = b-a;
15        b = b-a;
16        a = a+b;
17      }
18
19      b = a-b;
20      a = a-b;
21    }
22
23    return a;
24  }
```

Приведенная функция вычисляет наибольший общий делитель своих аргументов.

При вызове этой функции с аргументами 0 и 1 выполняются только инструкции в строках 3 и 4. При вызове с аргументами 1 и 0 будут выполнены строки 3, 5, 6. При вызове с аргументами 1 и –2 выполняются строки 3, 5, 7, 8, 10, 12, 14, 15, 16, 19, 20, 23. Таким образом, набор тестовых данных, состоящий из пар <0, 1>, <1, 0>, <1, –2> обеспечивает полное покрытие инструкций этой функции.

Заметим, что вместо <1, –2> можно было бы использовать два набора аргументов, например, <1, –1> и <1, 2>, первый из которых покрывает инструкцию 8, но не покрывает 14, 15, 16, а второй — покрывает эти три инструкции. С точки зрения получаемого покрытия все равно, какой набор тестовых данных выбрать. Однако могут быть существенны другие аспекты, например, время работы тестового набора и удобство анализа результатов тестирования. Время выполнения тестов обычно сокращается при уменьшении их количества, но сложный тест, эквивалентный по покрытию нескольким простым, в ряде случаев может выполняться дольше, чем все они вместе взятые. С точки зрения удобства анализа результатов, чем проще тесты, тем лучше, поскольку меньше различных факторов приходится рассматривать при локализации ошибки, найденной таким тестом.

В примерах, приведенных ниже, обычно используются наиболее компактные полные тестовые наборы для заданной метрики, но на практике всегда стоит рассмотреть вопросы эффективности выполнения тестов и удобства анализа результатов, прежде чем пытаться минимизировать их количество.

Для обнаружения всех ошибок покрытия 100% инструкций недостаточно. В следующем примере приведен код функции, которая должна по значению целого числа печатать его простую характеристику — ноль это, четное или нечетное число, положительное или отрицательное. В этом коде пропущена вставка слова «нечетное» в описание нечетных чисел. Однако тесты с входными данными 0, 2 и –2 дадут 100% покрытия строк и не обнаружат никаких ошибок.

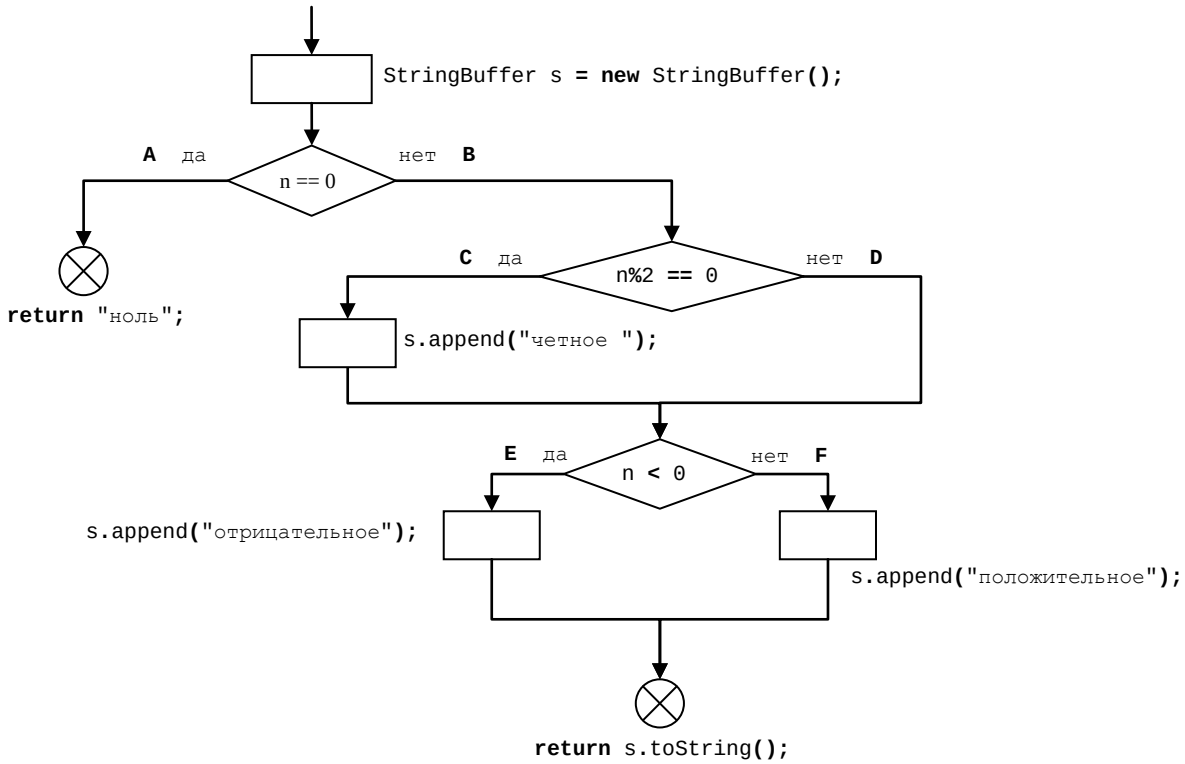
```
 1  String classifier(int n)
 2  {
 3    StringBuffer s = new StringBuffer();
 4
 5    if(n == 0)
 6      return "ноль";
 7
 8    if(n%2 == 0)
 9      s.append("четное ");
10
11    if(n < 0)
12      s.append("отрицательное");
13    else
14      s.append("положительное");
```

```
15
16    return s.toString();
17  }
```

Проблема здесь в том, что определенный код ошибочно пропущен, а покрытие инструкций, естественно, не гарантирует обнаружения пропущенных инструкций, поскольку оно вычисляется только по имеющимся. Чтобы решить эту проблему, используют *покрытие ветвей*.

Для определения ветвей нужно рассмотреть граф потока управления программы. Граф потока управления для приведенного выше примера изображен ниже. Для пояснений у блоков кода, которые всегда выполняются в одной последовательности, помещены соответствующие инструкции. Каждый условный оператор (также как и оператор цикла или выбора) имеет несколько ребер графа, ведущих из него, в соответствии с возможным ходом выполнения инструкций, то есть определяет разветвление потока управления. Каждое ребро, выходящее из вершины графа, из которой выходят и другие ребра, называется *ветвью*.

Условному оператору соответствуют два выходящих из его вершины ребра, оператору проверки условия цикла — тоже два (выйти из цикла или нет), а оператору выбора может соответствовать много выходящих ребер — по числу указанных вариантов значений выражения, по которому осуществляется выбор.



Покрыть ветвь означает обеспечить такой ход выполнения инструкций, что после инструкции ветвления, которая является начальной вершиной этой ветви, выполнится инструкция в концевой вершине ветви. Недостижимой считается ветвь, которая не может быть покрыта ни при каком выполнении кода. *Метрика покрытия ветвей* (branch coverage или decision coverage) вычисляется аналогично покрытию инструкций — это доля покрытых в ходе теста ветвей по отношению к общему количеству достижимых ветвей.

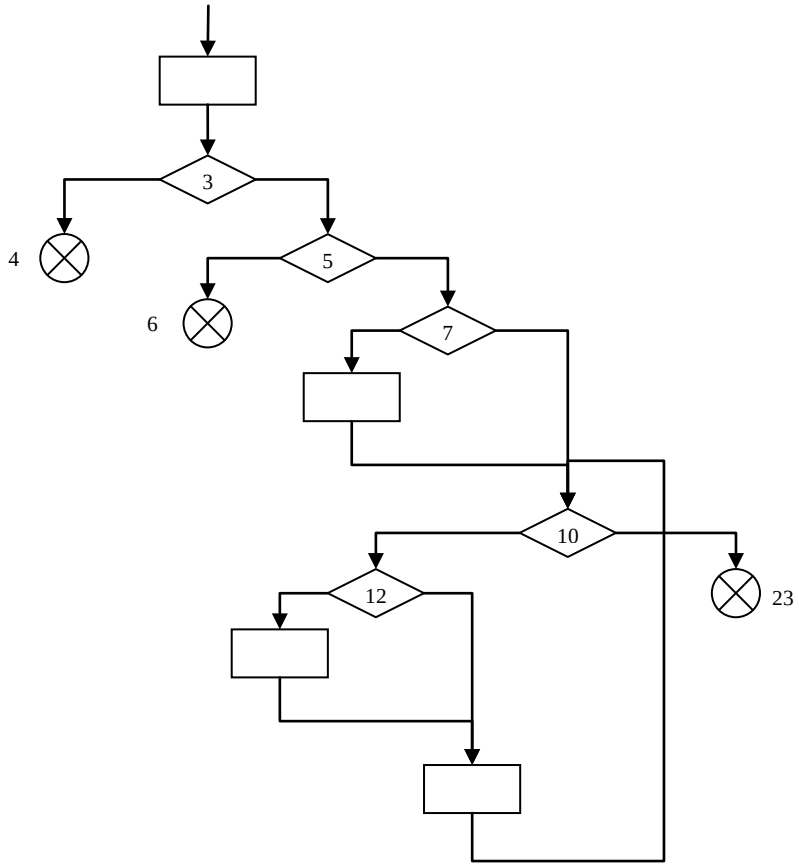
В рассматриваемом примере имеется шесть ветвей — на схеме графа они помечены латинскими буквами. Для каждой ветви, кроме D, есть соответствующий набор инструкций, который выполняется, когда покрывается эта ветвь. В нашем случае концевая вершина ветви D соответствует условному оператору, который выполнится и при покрытии другой ветви. Поэтому покрытие всех инструкций не гарантирует покрытия всех ветвей. Для полного покрытия ветвей во втором примере можно использовать, например, тестовые данные 0, 1, 2

и –2. Тест, использующий значение параметра 1, покажет, что для нечетных чисел соответствующая пометка не выдается.

Однако верно, что, если тесты дают 100% покрытия ветвей, они же всегда дадут и 100% покрытия инструкций. Для величин покрытия, меньших 100%, никаких общих соотношений между покрытием инструкций и ветвей указать нельзя. С одной стороны, количество инструкций может значительно превышать количество ветвей, с другой стороны, половина ветвей программы может не содержать инструкций, как в приведенном примере. Поэтому покрытие 95% инструкций может соответствовать только 5% покрытых ветвей и наоборот, покрыв 95% ветвей, можно покрыть только 5% инструкций.

Но тот факт, что 100% покрытие ветвей автоматически означает 100% покрытия инструкций, уже достаточно важен. Он показывает, что метрика покрытия ветвей «не грубее» метрики покрытия инструкций, выделяет не меньше разнообразных ситуаций. Если 100% покрытия по одной метрике тестового покрытия влечет 100% покрытия по другой метрике, говорят, что первая метрика *сильнее* или *тоньше*. Соответственно, вторая *слабее* или *грубее* первой. Если одна метрика сильнее другой, а та тоже сильнее первой, они *эквивалентны*. В этой ситуации все равно, какую из них использовать, если нас интересует только полное, 100% покрытие. Метрика покрытия ветвей сильнее метрики покрытия инструкций, а обратное не верно. Поэтому при достижении 100% покрытия ветвей тестирование в общем случае оказывается более полным, чем при покрытии 100% инструкций.

Чтобы покрыть все ветви в коде, вычисляющем наибольший общий делитель, тоже нужно дополнить набор тестов, покрывающий все инструкции, например, взять тестовые данные <0, 1>, <1, 0>, <1, –2>, <1, 2>. Чтобы убедиться в этом, достаточно изучить графа потока управления этого кода, представленный ниже. В нем вершины, соответствующие операторам ветвления и выхода из тела функции помечены номерами строк, содержащих эти операторы. В этом примере две ветви — отрицательные ветви для условных операторов в строках 7 и 12 — тоже не содержат соответствующих инструкций.



Допустим, однако, что мы, исправив последний пример, нечаянно внесли в него еще одну ошибку. Теперь нечетные числа помечаются как надо, но условие пометки отрицательных чисел неправильное.

```
 1  String classifier(int n)
 2  {
 3    StringBuffer s = new StringBuffer();
 4
 5    if(n == 0)
 6      return "ноль";
 7
 8    if(n%2 == 0)
 9      s.append("четное ");
10    else
11      s.append("нечетное ");
12
13    if(n < 0 && n%2 == 0)
14      s.append("отрицательное");
15    else
16      s.append("положительное");
17
18    return s.toString();
19  }
```

Набор тестовых данных 0, 1, 2, –2, по-прежнему покрывающий 100% ветвей, не обнаруживает такую ошибку. Чтобы выявить ее, необходимо использовать тесты, покрывающие комбинации условий, встречающихся в операторах ветвления.

Комбинации условий определяется следующим образом. Выделим условия всех ветвлений в коде программы, определяемых условными операторами, операторами цикла или операторами выбора. Эти условия являются предикатами, составленными при помощи логических операций (отрицания «не», конъюнкции «и», дизъюнкции «или», равенства, неравенства или исключающего «или») из *элементарных условий* — логических формул,

которые уже не разлагаются на составляющие логические формулы. Выделив все элементарные условия из ветвлений в коде заданной программы, мы можем составлять комбинации из этих условий и их отрицаний, или, что то же самое, из их значений true и false (1 и 0). В приведенном примере элементарными условиями являются предикаты (n == 0), (n%2 == 0) и (n < 0). Из трех условий и их отрицаний можно составить 8 комбинаций, однако не все эти комбинации будут выполнимы. Так, не может быть одновременно выполнено (n == 0) и (n < 0), или (n == 0) и !(n%2 == 0). Выбросив все невыполнимые комбинации, мы получим в данном примере только 5 комбинаций элементарных условий и их отрицаний — если (n == 0), значения остальных двух условий определяются однозначно, иначе два оставшихся условия становятся независимыми и дают еще 4 комбинации.

Номер	n == 0	n%2 == 0	n < 0
1	1	1	0
2	0	0	0
3	0	0	1
4	0	1	0
5	0	1	1

Комбинация значений элементарных условий покрывается тестом, если во время его выполнения эти условия в некоторый момент имеют в точности эти значения.

Метрика покрытия комбинаций условий (multiple condition coverage) определяется как доля покрытых текстами комбинаций значений элементарных условий, участвующих в условиях ветвлений программы, по отношению к общему количеству выполнимых комбинаций значений элементарных условий.

В приведенном выше примере набор тестовых данных 0, 1, 2, –2 не покрывает комбинацию условий с номером 3 в таблице. Покрыв эту ситуацию, например, с помощью значения параметра, равного –1, мы обнаружим внесенную в программу ошибку.

Метрика покрытия комбинаций условий строго сильнее метрики покрытия ветвей, поскольку она сильнее, а при использовании составных предикатов в условиях ветвлений, как в нашем примере, становится неэквивалентной ей.

В рассмотренном примере нам удалось достаточно быстро определить невыполнимые комбинации, поскольку все условия зависят от одной целочисленной переменной n. В общем случае вопрос о выполнимости различных комбинаций достаточно непрост, поскольку элементарные условия могут быть связаны неявно. Например, они могут использовать глобальные переменные, значения которых подчиняются нетривиальным ограничениям, скажем, одна из переменных может представлять список каких-то объектов, а вторая — индекс одного из объектов в этом списке, либо –1, если список пуст. В общем случае определить выполнимость комбинации значений произвольных формул оказывается достаточно сложно, поскольку для этого нужно знать смысл используемых в условиях переменных и функций. Поэтому вычисление тестового покрытия по метрике покрытия комбинаций условий в общем случае не автоматизируется.

Другая трудность практического использования покрытия по этой метрике связана с возможным экспоненциальным ростом количества ситуаций, выделяемых для программы при росте ее размера. Если в программе используется n операторов ветвления, условие каждого из них элементарно и между этими условиями нет никаких связей, то возникает всего лишь 2n ветвей, но 2^n возможных комбинаций условий. Поэтому на практике используются метрики тестового покрытия, более сильные, чем покрытие ветвей, но приводящие к меньшему числу различных ситуаций, чем покрытие комбинаций условий.

Если при этом не учитывать покрытие ветвей, получится *метрика покрытия условий* (condition coverage), которая не сильнее метрики покрытия ветвей. Для каждого элементарного условия определяется, сколько значений оно может принимать при всех

возможных сценариях выполнения программы. Обычно, если нет специфических ограничений, оно может принимать два значения. Но иногда встречаются *постоянные условия*, которые либо всегда равны true, либо всегда равны false, при всех выполнениях программы. Определяется общее число возможных значений элементарных условий, в которое каждое обычное условие вносит 2 значения, а постоянное условие — 1. Метрика покрытия условий вычисляется как отношение количества значений, которые все элементарные условия принимали в ходе теста, к числу возможных значений условий. Для достижения 100% по этой метрике надо, чтобы все элементарные условия принимали все возможные значения (что еще не гарантирует, что все ветви были покрыты). Так, в предыдущем примере 3 условия, каждое из которых может принимать значения true и false, соответственно, метрика покрытия условий определяет 6 целей, которые надо покрыть. Если, скажем, использованы тестовые значения 0 и 1, условия (n == 0) и (n%2 == 0) оба принимали по два значения, а условие (n < 0) – только значение false, поэтому достигнутое покрытие условий равно (2*2+1*1)/6 = 5/6 = 83.33%.

Наиболее слабая из использующих элементарные условия метрик, уточняющая покрытие ветвей, — *метрика покрытия условий и ветвей* (condition/decision coverage или condition/branch coverage). Считается она следующим образом. Метрика покрытия условий и ветвей вычисляется как отношение общего количества значений, которые все элементарные условия принимали в ходе теста, сложенного с количеством выполненных ветвей, к сумме общего возможного числа возможных значений условий и числа достижимых ветвей.

При наличии n элементарных условий в программе метрика покрытия условий и ветвей определяет не более 2n ситуаций. Она, как легко видеть, сильнее метрики покрытия ветвей. Однако в нашем примере тесты, дающие 100% покрытия ветвей, дают и полное покрытие условий и ветвей, не обнаруживая внесенную ошибку.

Более сильная метрика покрытия — *метрика модифицированного покрытия условий и ветвей* (modified condition/decision coverage, MC/DC). Набор тестов считается достигающим 100% покрытия по этой метрике, если

- каждая достижимая ветвь покрывается этим набором;
- каждое непостоянное элементарное условие принимает оба возможных значения при выполнении этого набора;
- для каждого составного условия ветвления и каждого входящего в него элементарного условия, изменение значения которого способно изменить значение всего условия, есть два теста, в которых все остальные входящие в это составное условие элементарные условия имеют одни и те же значения, а данное элементарное условие и составное условие в целом в этих тестах имеют различные значения.

Проще говоря, в полном тесте по метрике MC/DC должно быть продемонстрировано, что каждое элементарное условие, способное влиять на результирующее значение включающего его условия ветвления, действительно изменяет его значение независимо от остальных элементарных условий.

В ошибочном коде, рассмотренном выше, значение условие третьего ветвления (n < 0) && (n%2 == 0) может быть изменено за счет изменения значения любой из двух входящих в него формул. Но сделать это можно, только изменяя значение формулы с true на false. Поэтому полный по MC/DC тестовый набор должен обеспечивать для этих формул выполнение комбинаций значений <1, 1>, <0, 1>, <1, 0>. При этом внесенная ошибка будет обнаружена.

Рассмотрим более сложный пример. Пусть в коде некоторой функции имеется два условия ветвлений: (x > 0) && (y > 0) || (x == 0) && (z != null) и (x < 0) && (z == null) || (z != null) && z.isEmpty(). Составим таблицу возможных комбинаций значений элементарных условий и соответствующих значений условий ветвлений, учитывая, что выражение z.isEmpty() определено, только когда выполнено z != null.

	x > 0	x == 0	x < 0	y > 0	z == null	z != null	z.isEmpty()	I	II
1	0	0	1	0	0	1	0	0	0
2	0	0	1	0	0	1	1	0	1
3	0	0	1	0	1	0		0	1
4	0	0	1	1	0	1	0	0	0
5	0	0	1	1	0	1	1	0	1
6	0	0	1	1	1	0		0	1
7	0	1	0	0	0	1	0	1	0
8	0	1	0	0	0	1	1	1	1
9	0	1	0	0	1	0		0	0
10	0	1	0	1	0	1	0	1	0
11	0	1	0	1	0	1	1	1	1
12	0	1	0	1	1	0		0	0
13	1	0	0	0	0	1	0	0	0
14	1	0	0	0	0	1	1	0	1
15	1	0	0	0	1	0		0	0
16	1	0	0	1	0	1	0	1	0
17	1	0	0	1	0	1	1	1	1
18	1	0	0	1	1	0		1	0

В следующей таблице представлены только комбинации, существенные для первого ветвления. Среди них найдем такие пары, что в них меняется значение только одного элементарного условия и всего условия в целом.

	x > 0	x == 0	y > 0	z == null	I	x > 0	x == 0	y > 0	z == null
	0	0	0	0	0		C		
	0	0	0	1	0				
	0	0	1	0	0	A	D		
	0	0	1	1	0	B			
	0	1	0	0	1		C		G
	0	1	0	1	0				G
	0	1	1	0	1		D		H
	0	1	1	1	0				H
	1	0	0	0	0			E	
	1	0	0	1	0			F	
	1	0	1	0	1	A		E	
	1	0	1	1	1	B		F	

Соответствующие комбинации помечены в последних четырех столбцах таблицы. Две строки помечены одной буквой, когда они входят в одну пару комбинаций. Эта буква находится в столбце, соответствующем элементарному условию, которое меняет свое значение. Чтобы обеспечить полное покрытие по MC/DC для первого условия, необходимо выбрать по паре комбинаций, соответствующих хотя бы одной букве в каждом из последних четырех столбцов. Достаточно, например, выбрать комбинации, помеченные буквами A, D, E, H. Это всего лишь 5 комбинаций. Или B, C, F, G — это 6 комбинаций.

Ниже приведена аналогичная таблица для второго условия ветвления. Условия (z == null) и (z != null) однозначно определяют значения друг друга, поэтому можно рассматривать только одно из них. Кроме того, нельзя изменить значение (z == null) и не изменить значения z.isEmpty() — это условие вычислимо только при одном значении первого. Можно, однако, считать, что неопределенное значение формулы z.isEmpty() соответствует любому ее значению из другой комбинации.

	x < 0	z == null	z != null	z.isEmpty()	II	x < 0	z == null	z.isEmpty()
	1	0	1	0	0			C
	1	0	1	1	1			C
	1	1	0		1	A		
	0	0	1	0	0			D
	0	0	1	1	1		B	D
	0	1	0		0	A	B	

Таким образом, можно выбрать набор комбинаций, помеченных буквами A, B и D — всего 4 комбинации.

Итоговый полный набор комбинаций по метрике MC/DC помечен звездочками в следующей таблице.

	x > 0	x == 0	x < 0	y > 0	z == null	z != null	z.isEmpty()	I	II
1***	0	0	1	0	0	1	0	0	0
2	0	0	1	0	0	1	1	0	1
3	0	0	1	0	1	0		0	1
4	0	0	1	1	0	1	0	0	0
5	0	0	1	1	0	1	1	0	1
6***	0	0	1	1	1	0		0	1
7***	0	1	0	0	0	1	0	1	0
8	0	1	0	0	0	1	1	1	1
9***	0	1	0	0	1	0		0	0
10	0	1	0	1	0	1	0	1	0
11	0	1	0	1	0	1	1	1	1
12	0	1	0	1	1	0		0	0
13	1	0	0	0	0	1	0	0	0
14	1	0	0	0	0	1	1	0	1
15***	1	0	0	0	1	0		0	0
16***	1	0	0	1	0	1	0	1	0
17***	1	0	0	1	0	1	1	1	1
18***	1	0	0	1	1	0		1	0

Заметим, что из 18 возможных комбинаций достаточно отобрать только 8, чтобы получить полное покрытие по MC/DC.

- Пара комбинаций 1 и 7 показывает, что изменение значения (x == 0) может изменить выполняемую ветвь в первом ветвлении.
- Пара комбинаций 6 и 18 показывает то же для условия (x > 0).
- Пара комбинаций 7 и 9 — то же для условия (z == null).
- Пара комбинаций 15 и 18 — то же для условия (y > 0).
- Пара комбинаций 16 и 17 — то же для условия z.isEmpty() и второго ветвления.

- Пара комбинаций 6 и 18 — то же для условия (x < 0) и второго ветвления.
- Пара комбинаций 17 и 18 — то же для условия (z == null) и второго ветвления.

В общем случае метрика MC/DC позволяет вместо 2^n комбинаций условий использовать 2n различных ситуаций.

Другая метрика покрытия, более сильная, чем покрытие ветвей, основана на использовании короткой логики. Различные ситуации по этой метрике соответствуют различным *коротким дизъюнктам*, то есть комбинациям значений элементарных условий однозначно, с учетом короткой логики, определяющим выполнение всех ветвлений в коде программы. Соответственно, покрытие считается как доля покрытых дизъюнктов среди всех возможных.

Снова используем пример с двумя ветвлениями, определяемыми условиями (x > 0) && (y > 0) || (x == 0) && (z != null) и (x < 0) && (z == null) || (z != null) && z.isEmpty(). Пусть , для определенности, код программы имеет примерно такой вид.

```
if((x > 0) && (y > 0) || (x == 0) && (z != null)) …
else …
…
if((x < 0) && (z == null) || (z != null) && z.isEmpty()) …
else …
```

Таблица комбинаций значений элементарных условий, которые однозначно определяют ход ее выполнения в соответствии с короткой логикой, показана ниже. Эти комбинации отличаются от всех возможных комбинаций тем, что значение (y < 0) при (x <= 0) не важно, поскольку это условие не будет оцениваться по правилам короткой логики. Вместо звездочек можно подставить произвольные значения.

	x > 0	x == 0	x < 0	y > 0	z == null	z != null	z.isEmpty()	I	II
	0	0	1	*	0	1	0	0	0
	0	0	1	*	0	1	1	0	1
	0	0	1	*	1	0		0	1
	0	1	0	*	0	1	0	1	0
	0	1	0	*	0	1	1	1	1
	0	1	0	*	1	0		0	0
	1	0	0	0	0	1	0	0	0
	1	0	0	0	0	1	1	0	1
	1	0	0	0	1	0		0	0
	1	0	0	1	0	1	0	1	0
	1	0	0	1	0	1	1	1	1
	1	0	0	1	1	0		1	0

Строится такая таблица следующим образом.

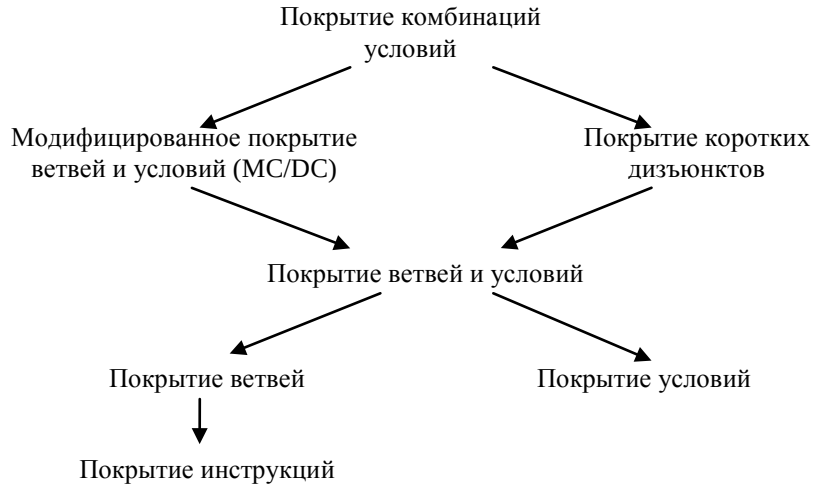
Элементарные условия, входящие в программу, выстраиваются в некоторой последовательности. Сначала всем элементарным формулам приписываются значения 0.

Для заданного набора значений определяется, выполним ли он с точки зрения взаимосвязей между условиями. Если набор значений не выполним, значение последней формулы, равной 0 изменяется на 1, давая новый набор значений. Если все формулы равны 1, таблица построена.

Если набор значений выполним, определяется соответствующий путь программы, при этом помечаются те формулы, значение которых было важно для вычисления этого пути. В результате часть формул становится помеченными. В таблицу заносится набор значений всех

помеченных формул, а для непомеченных — звездочка, означающая, что их значения несущественны для выбора пути исполнения при указанных значениях остальных формул. Далее изменяется значение последней формулы, равной 0, которая либо сама помечена, либо после которой есть помеченная. Так получается очередной набор значений. Если же все формулы равны 1, таблица построена.

Ниже приведена схема отношения «сильнее» для метрик покрытия, основанных на потоке управления.



Метрики покрытия на основе потоков данных

Метрики покрытия на основе потоков данных определяются использованием в программе различных значений данных. Поскольку задача метрики покрытия — выделение разнообразных ситуаций, основной интерес представляют переменные программы, которые могут в разных сценариях ее выполнения представлять различные значения. В качестве переменных рассматриваются и параметры функции.

Следующие два понятия нужны только для определения ряда метрик покрытий. Инструкция, в которой используется некоторая переменная, называется *ее использованием* (use). Инструкция, в которой определяется новое значение для некоторой переменной, называется *ее определением* (definition).

Точка входа в функцию считается определением для всех ее параметров.

Примеры.

Инструкция if(x > 0 && y <= z) …; использует три переменных — x, y, z.

Инструкция x = y++ – 12*(t = z+1) + t*t; использует переменные y, z, t и определяет переменные x, y, t. Для переменной y порядок определения и использования задан однозначно — она сначала используется, потом определяется. Для переменной t этот порядок зависит от языка и компилятора: для Java он задан однозначно — используется только что определенное значение t, равное z+1, для C или C++ этот порядок, а вместе с ним и результата всего выражения, зависит от конкретного компилятора и даже от того, собиралась ли содержащая эту инструкцию программа с оптимизациями или нет.

du-путь или *путь от определения к использованию* для заданной переменной — путь по графу потока управления, начинающийся с вершины, соответствующей инструкции, определяющей значение переменной, заканчивающийся вершиной, соответствующей инструкции ее использования, и не содержащий вершин для инструкций определения этой переменной, кроме первой.

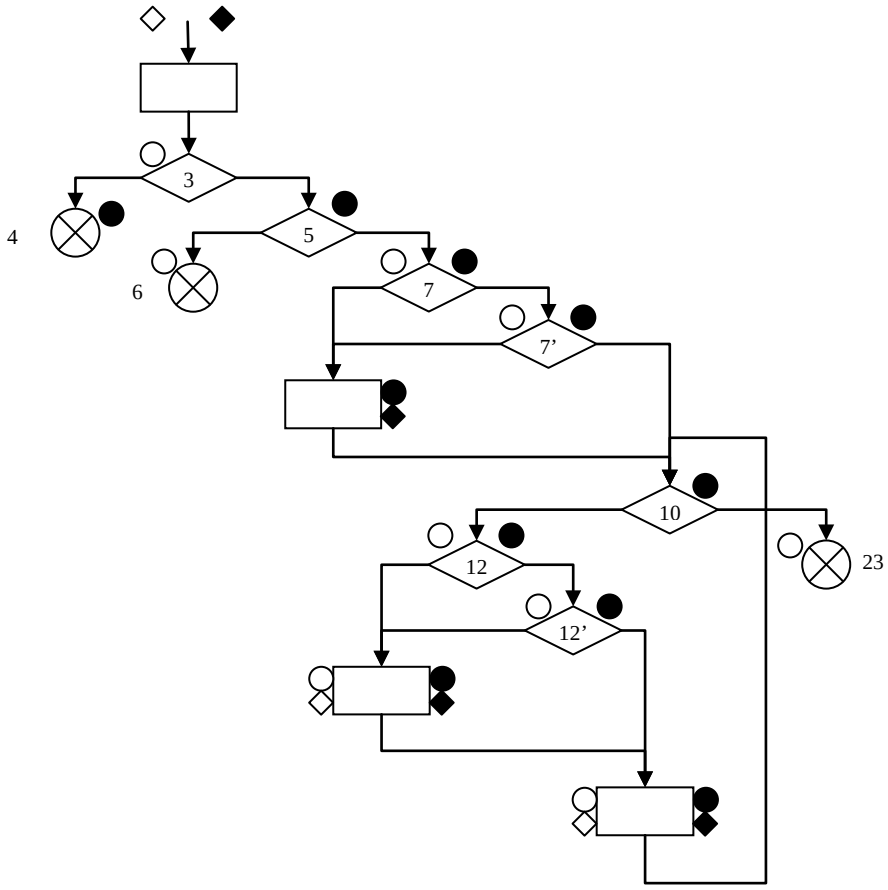
Инструкции использования переменной могут входить в du-путь много раз.

Рассмотрим снова пример функции, вычисляющей наибольший общий делитель.

```
1  int gcd(int a, int b)
```

```
 2  {
 3    if(a == 0)
 4      return b;
 5    if(b == 0)
 6      return a;
 7    if(a > 0 && b < 0 || a < 0 && b > 0)
 8      b = -b;
 9
10    while(b != 0)
11    {
12      if(b > a && a > 0 || b < a && a < 0)
13      {
14        a = b-a;
15        b = b-a;
16        a = a+b;
17      }
18
19      b = a-b;
20      a = a-b;
21    }
22
23    return a;
24  }
```

Чтобы аккуратно определять места определения и использования переменных нужно построить полный граф потока управления с учетом короткой логики. В этом графе составному условию ветвления может соответствовать несколько ветвлений — ровно столько, сколько разных сценариев вычисления этого условия с учетом короткой логики.



В нашем примере две переменных — a и b. Места определения значений a помечены на графе белыми ромбами, а места использования — белыми кружками. Для b определения помечены черными ромбами, а использования — черными кружками.

du-путь считается *покрытым*, если он был полностью пройден при выполнении программы. *Недостижимым* называется du-путь, который не может быть покрыт ни при каком выполнении программы. Использование переменной считается *покрытым*, если для каждого определения этой переменной, для которого есть хотя бы один du-путь, начинающийся в этом определении и ведущий к этому использованию, один из таких путей был покрыт.

Метрика покрытия использований (all-uses coverage) — доля покрытых использований всех переменных по отношению к количеству достижимых использований.

Метрика покрытия du-путей (du-path coverage) — доля покрытых du-путей для всех переменных программы по отношению к достижимым du-путям.

Метрика покрытия du-путей сильнее метрики покрытия использований. Она является одной из самых сильных метрик покрытия, используемых на практике. Хотя кажется, что количество тестов в полном наборе по такой метрике может быть очень большим, в большинстве случаев это не так — один тест часто покрывает сразу много du-путей.

***Замечание**. Обычно в учебниках и статьях, касающихся метрик покрытия на основе потока управления и на основе потоков данных, утверждается, что полное покрытие использований (и, тем более, du-путей) влечет полное покрытие ветвей. Это неправда, в чем можно убедиться на следующем примере.*

```
int f(int x)
{
  if(x >= 0)
    return 1;
  else
    return -1;
}
```

Эта ошибка была допущена еще в статье [4], где впервые определены метрики покрытия кода на основе потока управления, ее последующее исправление в [5] выполнено неаккуратно, так что по существу ошибка так и осталась. Но большинство авторов учебников и монографий по тестированию, по-видимому, цитируют эти статьи или обзор [2] без вникания в детали.

В примере вычисления наибольшего общего делителя имеются следующие du-пути. Здесь числами обозначаются строки программы, 0 соответствует входу, а числом с апострофом — второе использование переменной в строке с тем же номером. Путь, в котором условие ветвления n разрешается положительно, содержит (n), отрицательно — (n). Перечислены только du-пути, для которых нет однозначно определенного содержащего их du-пути.

Для переменной a — 0-3, 0-6, 0-7, 0-7’, 0-(7)-23, 0-(7’)-23, 0-(7’)-23, 0-(7)-12, 0-(7’)-12, 0-(7’)-12, 0-(7)-12’, 0-(7’)-12’, 0-(7’)-12’, 0-(7,12)-14, 0-(7’,12)-14, 0-(7’, 12)-14, 0-(7,12’)-14, 0-(7’,12’)-14, 0-(7’, 12’)-14, 0-(7,12’)-20, 0-(7’,12’)-20, 0-(7’,12’)-20, 14-20, 20-12, 20-12’, 20-23, 20-(12)-14, 20-(12’)-14, 20-(12’)-20.

Для переменной b — 0-4, 0-5, 0-7, 0-7’, 0-(7)-8, 0-(7’)-8, 0-(7’)-10, 0-(7’)-12, 0-(7’)-12’, 0-(7’,12)-15, 0-(7’,12’)-15, 0-(7’,12’)-19, 8-10, 8-12, 8-12’, 8-(12)-15, 8-(12’)-15, 8-(12’)-19, 15-19, 19-10, 19-12, 19-12’, 19-(12)-15, 19-(12’)-15, 19-(12’)-19.

Недостижимыми являются все пути вида 0-(...)-23 — поскольку в случае (b == 0) мы свернем уже в строке 5.

Для достижения всех остальных путей достаточно взять набор тестовых данных <1, 0> (дает 0-3-5-6), <0, 1> (дает 0-3-4), <1, 1> (дает 0-7-7’-(7’)-10-12-12’-(12’)-19-20 и 20-23), <1, –1> (дает 0-7-(7)-8-10-12’-(12’)-19-20), <–1, 1> (дает 0-7-7’-(7’)-8-10-12’-(12’)-19-20), <1, 2> (дает 0-(7’)-12-(12)-14-15-19-20-10-12-12’-(12’)-19-20), <–1, –2> (0-(7’)-12-12’-(12’)-14-15-19-20-10-12-12’-(12’)-19-20), <1, –2> (0-7-(7)-8-10-12-(12)-14-15-19-20-10-12-12’-(12’)-19-20), <–1, 2> (0-7-7’-(7’)-8-10-12-12’-(12’)-14-15-19-20-10-12-12’-(12’)-19-20), <2, –1>, <–2, 1>, <2, 3>, <–2, –3>.

Структурные критерии на уровне компонента

На уровне компонентов, содержащих несколько методов, метрики покрытия могут определяться с использованием метрик покрытия кода отдельных методов. Но кроме этого есть более высокоуровневая информация — граф вызовов методов и функций компонента друг из друга. Одна и та же функция может вызываться из данной несколько раз в различных ветвях, поэтому ребра этого графа соответствуют вхождениям инструкций вызова — сколько различных инструкций содержит вызов функции f(), столько и ребер будет вести из данной функции в функцию f().

Так же, как и в случае ветвей, вызовы могут быть достижимыми или недостижимыми. *Метрика покрытия вызовов* вычисляется как отношение выполненных различных инструкций вызова к общему количеству достижимых таких инструкций.

Вычислять для компонента метрики покрытия, определенные на основе кода методов, можно, но при этом часто становится еще более тяжело определить, какие из определяемых ими ситуаций являются достижимыми, а какие — нет. Для метрики покрытия вызовов решение этой задачи обычно не слишком тяжело.

Метрики на основе потоков данных на уровне компонента строятся на базе изменения и использования глобальных переменных или полей класса различными методами. Точно так же, если мы знаем, какие методы используют, а какие изменяют значения полей класса, естественно пытаться покрыть все пары определение-использование, вызывая цепочки методов.

Структурные критерии на уровне подсистемы

На уровне подсистемы графы потоков данных и потока управления во многом сливаются, превращаясь в схемы передачи управления или сообщений между компонентами. Обычно определяют множество сценариев взаимодействия компонентов, в каждом из которых реализуется некоторая часть возможных передач управления и сообщений, а затем измеряют общее покрытие как часть из этих сценариев, реализованную в ходе тестирования.

В больших системах могут использоваться метрики, основанные просто на доле затронутых тестами функций, компонентов, форм или окон, таблиц данных или других элементов данных по отношению к общему числу соответствующих элементов. Их основное достоинство в том, что они достаточно просто измеряются и не требуют для понимания определения каких-то дополнительных сущностей — сценариев, возможных вызовов, du-путей и пр.

Критерии полноты на основе структуры входных данных

Часто при тестировании нельзя воспользоваться информацией об устройстве тестируемой системы, поскольку ее исходный код недоступен. При этом вся известная информация о ее структуре — это структура входных данных — общий список доступных извне интерфейсов и структуры данных параметров каждого интерфейса.

Чтобы определить метрику покрытия входных данных, нужно разбить их на подмножества эквивалентных с какой-то точки зрения данных. Есть два основных способа определения такой эквивалентности.

- Выделение разных подтипов данных одного типа на основе практических соображений. Например, ряд соображений позволяет разделить целые числа на положительные, отрицательные и 0. Следуя другим соображениям можно разбить их на четные и нечетные числа.

  Для числа типа double можно определить такие возможности: 0, целое, с небольшим количеством цифр в дробной части (<= 2), например, 0.5 или 0.25, с большим количеством цифр в дробной части, например, $9.38752635549 \cdot 10^{-7}$. Кроме того, можно

выделить положительные и отрицательные числа, а также превосходящие и не превосходящие 1 по абсолютной величине.

- Разделение значений по определенным правилам, касающимся их структуры.
  Например, целые числа обычно представляются в двоично-дополнительном формате, все отрицательные имеют первый (последний) бит, равный 1, в отличие от неотрицательных. Поэтому разбиение на положительные и отрицательные можно обосновать структурой представления числа в машине. Можно делить целые числа на большие и небольшие по абсолютной величине в соответствии с тем, есть ли хотя бы один бит равный 1 среди первых 16. То есть, числа >= 65536 по абсолютной величине объявляются большими, а меньшие — небольшими.
  Числа с плавающей точкой имеют более сложную структуру. В представлении числа формата double используется 64 бита. Первый бит — знак S, следующие 11 бит представляют экспоненту E, оставшиеся 52 бита — мантиссу M. Само число при этом считается равным $(-1)^S \cdot (2^{52}+M)/2^{52} \cdot 2^{(E-1023)}$, если экспонента не равна 0 или 2047.
  Если экспонента равна 0, соответствующие числа называются денормализованными и вычисляются по другой формуле — $(-1)^S \cdot M/2^{52} \cdot 2^{-1023}$. Если экспонента равна 2047, то при нулевой мантиссе получаются представления для положительной и отрицательной бесконечностей, а при ненулевой — для специального значения NaN, которое обозначает неопределенный результат.
  Таким образом, как специальные классы чисел с плавающей точкой можно выделить 0, –0 (есть такое специальное число!), положительные и отрицательные денормализованные, положительные и отрицательные нормализованные числа, положительную и отрицательную бесконечность и множество значений NaN.

Для определения классов эквивалентных сложных данных обычно используют их структуру и классы эквивалентности данных простых типов, из которых они построены. Так, определяя разбиения на классы для объектов, у которых два целочисленных поля, достаточно естественно объявить эквивалентными объекты, у которых соответствующие поля имеют один знак или одновременно равны 0 — получается всего 9 классов таких объектов.

Для более сложных данных используют описание их структуры в виде грамматики, чтобы выделить классы эквивалентности или соответствующие метрики покрытия.

Например, для документов, которые описываются некоторой контекстно свободной грамматикой, можно определить следующие метрики покрытия.

Метрика покрытия правил — доля правил грамматики, использованных для построения тестовых данных, среди всех ее правил.

Метрика покрытия альтернатив — для каждого правила определяется, сколько имеется возможных альтернатив его раскрытия, и вместо 1 в определении предыдущей метрики для всех правил учитывается это число, а для покрытых — только количество реализованных альтернатив.

Пример. Пусть входной документ программы соответствует следующей грамматике.

Doc ::= A | B ;

A ::= (“X” | “Y” | “Z”) (“::”)? “!”

B ::= “O” (A)*

В этом случае один входной документ вида “OX!” покрывает все правила — для его построения используется и корневое правило, и правило для B, и правило для A.

Чтобы покрыть все альтернативы в правиле A, достаточно использовать такие входные данные: “X::!”, “Y!”, “Z!”. Здесь первая альтернатива раскрыта всеми возможными способами. Вторая альтернатива — наличие или присутствие “::” — тоже. Для правила B достаточно “O” и “OX!”, если считать, что опциональный список (A)? достаточно раскрыть на глубину 1. Чтобы список можно было отличить от опционального символа, можно

раскрыть его на глубину 2, использовав, например, “O”, “OX!” и “OX!Z!”. Таким образом, набор данных “X::!”, “Y!”, “Z!”, “O”, “OX!”, “OX!Z!” покрывает все альтернативы во всех правилах.

Можно определить более сложные метрики покрытия комбинаций альтернатив, но обычно они на практике дают слишком большое количество необходимых тестов.

Критерии полноты на основе требований

Базовая идея метрик покрытия на основе требований — если два теста проверяют выполнения одних и тех же ограничений, определенных в требованиях, скорее всего, они либо оба обнаружат ошибку, либо оба выполнятся успешно. Поэтому их можно считать эквивалентными.

Это неправда, как и базовая идея структурных критериев, но тоже позволяет ввести достаточно удобные метрики покрытия.

Обычно требования оформляются в виде неформального текста, организованного иерархически, т.е. с выделенными пунктами и подпунктами. В этом случае можно определить метрику покрытия требований как долю проверяемых тестовым набором наиболее детальных выделенных пунктов требований среди тех, которые вообще можно проверить (иногда часть таких требований невозможно проверить вообще, либо за время, ограниченное рамками проекта).

Более формальное определение можно дать *метрике покрытия утверждений*. Для ее определения из требований выделяются элементарные проверяемые утверждения, выполнение каждого которых, вообще говоря, не связано с выполнением остальных. Метрика определяется как доля проверенных в тестах таких утверждений по отношению ко всем.

Например, пусть в требованиях к системе есть такая фраза.

«Система должна поддерживать выполнение операций чтения, добавления, удаления и модификации записи о клиенте, причем, при параллельной работе нескольких операторов только один из них в своей сессии может удалять и модифицировать уже имеющуюся запись».

В ней можно выделить следующие утверждения:

- «Оператор может прочесть данные записи о клиенте»
- «Оператор может добавить запись о клиенте»
- «Оператор может удалить запись о клиенте, если работает с ней один»
- «Оператор может модифицировать запись о клиенте, если работает с ней один»
- «Если несколько операторов работает с записью о клиенте, только один может ее удалить»
- «Если несколько операторов работает с записью о клиенте, только один может ее модифицировать»

Соответственно, измерять полноту покрытия тестов можно отношением количества проверенных из этих утверждений к общему числу выделенных.

Часто встречающийся случай — оформление требований в виде набора правил (или бизнес-правил). В этом случае *метрикой покрытия правил* считают долю проверенных тестами правил среди всех имеющихся.

Например, система управления лифтами может описываться следующими правилами.

1. Если нажимается внешняя кнопка вызова лифта на каком-то этаже, этот вызов помещается в общее множество вызовов.
2. Если нажимается внутренняя кнопка вызова лифта на какой-то этаж, кабина помещает этот вызов в свое множество обслуживаемых.

3. Если препятствий к закрытию дверей кабины нет в течение двух секунд, двери закрываются.
4. Если кабина прибывает на один из этажей, вызовы с которых есть у нее в множестве обслуживаемых, она останавливается и открывает двери.
5. Если двери закрылись и обслуживаемых или общих вызовов нет, кабина стоит на месте и направление ее движения не определено.
6. Если двери закрылись и есть обслуживаемые вызовы, направления движения кабины совпадает с предыдущим, если этаж не крайний (первый или последний), и противоположно, если этаж крайний.
7. Если двери закрылись, определено направление движения, и есть общие вызовы с этажа по этому направлению, кабина выбирает ближайший из них по разнице этажей, добавляет его в свое множество обслуживаемых и удаляет из общего.
8. Если двери закрылись и направление движения не определено, но есть общие вызовы, кабина выбирает ближайший к ней по разнице этажей, добавляет его в свое множество обслуживаемых и удаляет из общего, после чего направление движение кабины определяется направлением на этот этаж.

Как видно, правила часто могут быть представлены в виде выражений «Если выполнено A, то сделать B», где A — некоторое составное условие в терминах предметной области, B — некоторое множество действий.

Это позволяет определить метрики покрытия условий правил примерно так же, как это делается для покрытия условий, используемых в коде. Можно использовать аналоги метрик покрытия элементарных условий, условий и ветвлений, комбинаций условий или MC/DC. Использовать аналог метрики покрытия коротких дизъюнктов нельзя, если только мы не уверены, что в коде эти же условия используются ровно в этом же порядке, что практически никогда не выполнено. Поэтому в аналогичных ситуациях нужно использовать «длинные» дизъюнкты, которые эквивалентны просто всем комбинациям элементарных условий.

Те же соображения помогают определить аналогичные метрики покрытия требований в тех случаях, когда требования описываются на любом формализованном языке.

В целом критерии покрытия на основе требований обладают важным преимуществом уже потому, что они позволяют учитывать требования при оценке полноты тестирования, так что непроверенное требование автоматически означает не вполне полное тестирование. Однако, в отличие от структурных метрик, для организации измерения покрытия по требованиям нужны серьезные дополнительные усилия, например, указание у каждого теста, какие требования он проверяет. Кроме того, проверка некоторого требования в определенной ситуации еще не означает, что то же требование будет выполнено в другой, если в этих ситуациях работают различные наборы компонентов и элементов кода тестируемой системы.

Структурные критерии и критерии полноты на основе требований, также иногда называемые функциональными, удачно дополняют друг друга — структурные позволяют отслеживать полноту тестов по отношению к коду, а функциональные — по отношению к требованиям. Если же в двух ситуациях задействуется один и тот же код и проверяются одни и те же требования, достаточно трудно сделать так, чтобы в них система работала по-разному, т.е. в одном случае правильно, а в другом ошибочно. Поэтому на практике для оценки полноты тестирования хорошо использовать комбинацию из структурных и функциональных критериев покрытия.

Критерии полноты на основе предположений об ошибках.

Поскольку общие предположения вида «если что-то выполняется или проверяется и содержит ошибку, то мы ее обнаружим» неверны, некоторые специалисты по тестированию предлагают использовать явное указание ошибок, на обнаружение которых нацелен набор тестов, в качестве критерия его полноты.

К сожалению, описать все возможные ошибки крайне тяжело, а если указать только некоторые, возникает законное подозрение, что эти-то ошибки тесты находят, а вот какие-то другие — нет. Если наша программа действительно застрахована от ошибок других типов, все в порядке, но в большинстве случаев это не так.

Наиболее удобный на практике способ измерения полноты тестирования на основе явных гипотез о возможных ошибках — это метод определения полноты тестов на основе обнаруженных мутантов.

В рамках этого метода для языка программирования, на котором написана тестируемая программа, определяется достаточно полный набор *операторов мутации*. Каждый такой оператор изменяет текст программ, например, удаляя определенную инструкцию, вставляя новую инструкцию, заменяя переменные в выражениях на другие переменные того же типа или на константные выражения того же типа, заменяя операторы арифметических действий +, –, *, / друг на друга, заменяя операторы логических операций друг на друга и пр. Важно, что после применения любого из операторов мутации синтаксически и семантически корректная программа остается корректной.

Программа, получаемая из тестируемой применением одного оператора мутации, называется *мутантом*. При применении большего числа операторов получаются мутанты второго и более высоких порядков, но они обычно не используются, потому что их количество даже для небольшой программы очень велико.

Те мутанты, которые эквивалентны по поведению исходной программе, т.е. ведут себя точно так же во всех ситуациях, выбрасывают из полученного множества мутантов. После этого используется метрика полноты тестов, определяемая как доля обнаруживаемых тестами мутантов среди оставшихся.

На практике наборы мутантов обычно получаются достаточно большими, и приходится затрачивать значительные усилия, чтобы отсеять из них эквивалентные исходной программе, поскольку обнаружение эквивалентности нельзя автоматизировать полностью. В результате метрика полноты на основе доли обнаруженных мутантов достаточно сильна, но ее применение очень трудоемко.

Еще одним аргументом против использования мутантов служит то, что они помогают обнаруживать только небольшие и случайные ошибки-опечатки. Серьезная ошибка в понимании требований чаще всего приводит не к изменению одного знака или пропуску одной инструкции, а к потере целой группы инструкций, которые должны были срабатывать в определенных условиях, вместе с условиями их выполнения. Обнаружить такую ошибку с помощью мутаций очень нелегко.

Критерии полноты на основе произвольных моделей

Различные другие модели поведения или устройства тестируемой системы, так же, как и описания требований к ней или графовые схемы передачи управления, могут служить для определения критериев покрытия. Для этого достаточно, чтобы в модели были некоторые элементы, которые задействуются в различных ситуациях.

Критерии полноты на базе моделей могут в ряде случаев уточнять и детализировать критерии полноты, полученные непосредственно из требований. На практике все дополнительные модели обычно являются уточненными требованиями к тестируемой системе, поэтому у определенных с их помощью критериев те же достоинства и недостатки, которые выше были указаны для критериев, основанных на требованиях.

Алгебраические модели

Один из экзотических примеров такого рода — алгебраические описания программных систем. Абстрактный тип данных, класс или компонент может быть описан алгебраически

как система с операциями, удовлетворяющими определенным соотношениям на их результаты.

Например, тип списка элементов типа T с операциями получения числа элементов, добавления элемента в заданное место списка, удаления элемента из заданного места и получения элемента, находящегося в определенном месте, описывается так.

[].size() = 0

[X.size()] ≡ [X]

(i <= X.size()) => X.add(i, o).size() = X.size()+1

(i < X.size()) => X.remove(i).size() = X.size()–1

(i < X.size()) => [X.get(i)] ≡ [X]

(i, j <= X.size() & i < j) => [X.add(i, o1).add(j, o2)] ≡ [X.add(j–1, o2).add(i, o1)]

(i <= X.size()) => [X.add(i, o1).add(i, o2)] ≡ [X.add(i, o2).add(i+1, o1)]

(i <= X.size()) => [X.add(i, o).remove(i)] ≡ [X]

(i, j <= X.size() & i < j) => [X.add(i, o).remove(j)] ≡ [X.remove(j–1).add(i, o)]

(i, j <= X.size() & i > j) => [X.add(i, o).remove(j)] ≡ [X.remove(j).add(i, o)]

(i <= X.size()) => X.add(i, o).get(i) = o

(i, j <= X.size() & i < j) => X.add(i, o).get(j) = X.get(j-1)

(i, j <= X.size() & i > j) => X.add(i, o).get(j) = X.get(j)

(i, j < X.size()-1 & i < j) => [X.remove(i).remove(j)] ≡ [X.remove(j+1).remove(i)]

(i, j < X.size() & i <= j) => X.remove(i).get(j) = X.get(j+1)

(i, j < X.size() & i > j) => X.remove(i).get(j) = X.get(j)

Терм в данной системе — любая конечная цепочка операций, примененная к []. В приведенных аксиомах X можно заменять на произвольный терм. Терм считается правильным, если выполнено следующее.

- Операция add(i, o) применяется в нем только к подтермам X, про которые можно доказать, что i <= X.size().
- Операции remove(i) и get(i) применяются в нем только к подтермам X, про которые можно доказать, что i < X.size().

В аксиомах [X] ≡ [Y] означает эквивалентность термов X и Y, так что в любом высказывании X можно заменять на Y и обратно без изменений в истинности этого высказывания. Из этих аксиом можно доказать, что любой список, т.е. правильный терм, эквивалентен терму, получаемому конечной цепочкой операций add(), в которых используются значения первого параметра, на единицу меньшие номера вызова операции в цепочке, — это канонический вид списка. Длиной терма назовем количество операций, участвующих в нем.

Тестами можно считать произвольные термы.

Метрикой покрытия аксиом можно считать долю тех аксиом, которые используются при приведении заданного набора тестов к каноническому виду, среди всех выписанных аксиом.

Метрикой покрытия термов длины <=k можно считать долю термов длины <=k, используемых в тестах или появляющихся в процессе их приведения к каноническому виду, среди всех термов длины <=k.

К сожалению, для алгебраических моделей трудно обосновать выбор k такого, что стоит пытаться добиться полного покрытия всех термов длины <=k, но не стоит пытаться покрыть все термы длины <=(k+1).

Автоматные модели

Гораздо более широко используют описания поведения программных систем в виде конечных автоматов или их расширений — расширенных, бесконечных, взаимодействующих, иерархических автоматов, систем переходов и т.д.

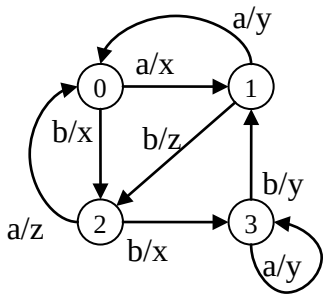
У всех видов автоматов имеются состояния и переходы. Соответственно, для всех видов автоматов можно определить следующие метрики покрытия.

Метрика покрытия состояний — доля тех состояний, в которых система побывала во время тестирования, по отношению к количеству всех достижимых состояний.

Метрика покрытия переходов — доля переходов, выполненных во время тестирования, по отношению к числу всех достижимых переходов.

Метрика покрытия цепочек переходов длины k — доля выполненных при тестировании цепочек последовательных переходов дины k по отношению к общему количеству достижимых цепочек последовательных переходов длины k.

Например, для конечного автомата, изображенного ниже, достаточно выполнить цепочку входов abb, чтобы побывать во всех состояниях. Покрытие переходов при этом будет равно только 3/8 = 37.5%.



Чтобы покрыть все переходы, можно выполнить цепочку bbabbaaa. Покрытие пар последовательных переходов при этом равно 7/16 = 43.75%.

Покрытые пары последовательных переходов: 0-aa, 0-ab, 0-ba, 0-bb, 1-aa, 1-ab, 1-ba, 1-bb, 2-aa, 2-ab, 2-ba, 2-bb, 3-aa, 3-ab, 3-ba, 3-bb.

В описании расширенных автоматов участвуют предикаты-охранные условия переходов. Поэтому при определении метрик покрытия для расширенных автоматов можно использовать те же подходы, что и для определения метрик покрытия условий, их комбинаций и MC/DC.

Литература

[1] IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004.

[2] H. Zhu, P. A. V. Hall, J. H. R. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427, Dec. 1997.

[3] B. Beizer. Software Testing Techniques. International Thomson Press, 1990.

[4] S. Rapps, E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering 11:367–375, 1985.

[5] P. G. Frankl, E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. IEEE Transactions on Software Engineering 14:1483–1498, 1988.

Тестирование на основе моделей

В. В. Кулямин

Лекция 4. Основные техники построения тестов

В этой лекции рассматриваются основные техники построения тестов и используемые для этого модели. Начнем с разных видов моделей.

Виды моделей, используемых при построении тестов

Для разработки тестов необходима информация о корректном поведении тестируемой системы и о разнообразии ситуаций, которые могут возникать при ее взаимодействии с другими системами или людьми. Чаще всего при использовании моделей для построения тестов различные тестовые ситуации стараются выделить из структуры этих моделей. Сами же модели описывают, прежде всего, поведение системы или требования к этому поведению.

Все модели, используемые для описания поведения ПО делятся на два основных вида — *исполнимые* (или *операционные*) и *логико-алгебраические*.

Исполнимые модели дают описание поведения системы в терминах ее действий, определяя ***как*** она работает: какие воздействия она получает извне, что делает в ответ, какие реакции выдает. Такое описание либо можно непосредственно выполнить, либо для его выполнения нужна некоторая виртуальная машина. Обычно из самого вида модели устройство такой машины примерно понятно, хотя многие детали ее устройства могут оказаться нетривиальны. Типичный пример исполнимой модели — конечный автомат. Он работает очень просто — сначала находится в начальном состоянии, ему на вход подается последовательность входных символов, он выдает в ответ последовательность выходных символов той же длины и вместе с выдачей каждого символа изменяет свое состояние.

Логико-алгебраические модели описывают поведение системы не в терминах ее действий, а в терминах свойств результатов ее работы. Они делают больший акцент на то, ***что*** она делает, а не как это происходит. Чаще всего они не могут быть выполнены непосредственно.

Есть, однако, промежуточные разновидности моделей, которые хотя имеют логико-алгебраическую природу, но и достаточно легко исполнимы, или совмещают элементы обоих классов моделей.

Исполнимые модели

Практически все виды исполнимых моделей являются обобщенными и расширенными автоматами разных типов.

Конечные автоматы (Finite State Machines, FSM, Finite Automata).

Конечный автомат — набор (S, s_0, I, O, T), где

S — конечное множество, элементы которого называются *состояниями* автомата;

s_0 — элемент S, называемый *начальным состоянием*;

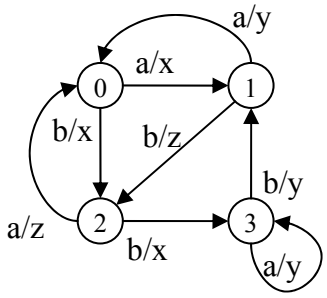
I — конечное множество, элементы которого называются *входными символами*, *входами* или *стимулами*, само I называют *входным алфавитом* автомата;

O — конечное множество, элементы которого называются *выходными символами*, *выходами* или *реакциями*, само O называют *выходным алфавитом* автомата;

$T \subseteq S \times I \times O \times S$ — множество *переходов* автомата. Каждый переход — четверка (s_1, i, o, s_2) — имеет *начальное состояние* s_1, *конечное состояние* s_2, *стимул* i и *реакцию* o. Говорят, что он *выходит из* s_1 и *ведет в* s_2, помечен стимулом i и реакцией o. Этот переход изображают стрелкой, ведущей из s_1 и в s_2 и помеченной i/o.

Автомат работает следующим образом. В начале он считается находящимся в начальном состоянии. На вход он получает извне некоторую последовательность стимулов. При получении очередного стимула в некотором состоянии недетерминированным образом выбирается один из переходов, помеченных принятым стимулом и выходящих из текущего состояния. Очередным текущим состоянием автомата становится конечное состояние выбранного перехода, и наружу выдается реакция, которой помечен выбранный переход. Выполнение автомата не определено, если в текущем состоянии нет переходов, помеченных получаемым стимулом.

Таким образом, автомат реализует некоторое соответствие последовательностей символов из I и последовательностей символов из O, $\varphi \subseteq I^* \times O^*$. Это соответствие, однако, описывается не прямо, а через возможные сценарии работы автомата. Часто его называют *поведением автомата* и рассматривают автомат как способ реализации этого соответствия.



На рисунке выше изображен конечный автомат с множеством состояний {0,1,2,3}, множеством стимулов {a,b}, множеством реакций {x,y,z} и множеством переходов {<0,a,x,1>, <0,b,x,2>, <1,a,y,0>, <1,b,z,2>, <2,a,z,0>, <2,b,x,3>, <3,a,y,3>, <3,b,y,1>}. Реализуемое им соответствие φ является отображением всех возможных конечных последовательностей {a,b}* в конечные последовательности {x,y,z}*. Например, φ(aaaaaa) = xyxyxy, φ(abaabaaba) = xzzxzzxzz, φ(bbbbbb) = xxyzxy.

Обобщением конечных автоматов являются необязательно конечные автоматы, в которых множества состояний, стимулов и реакций уже не всегда являются конечными.

Другое обобщение — системы размеченных переходов или просто системы переходов (Labeled Transition Systems, LTS). В обычных системах переходов не различают стимулы и реакции, называя их просто *действиями*. Ниже определяется система переходов со стимулами и реакциями (Input Output Labeled Transition System, IOLTS) — такие системы более естественно моделируют По с точки зрения тестирования, при котором всегда есть воздействия на систему и есть выдаваемые ей реакции на эти воздействия.

Конечная система переходов со стимулами и реакциями — набор (S, s_0, I, O, τ, T), где

S — конечное множество *состояний* системы;

s_0 — элемент S, называемый *начальным состоянием*;

I — конечное множество *входных символов*, *входов* или *стимулом*, I называют *входным алфавитом* системы;

O — конечное множество *выходных символов*, *выходов* или *реакций*, O называют *выходным алфавитом* системы; I и O не пересекаются;

τ — некоторый символ, не являющийся элементом I и O, он называется *пустым* или *внутренним действием*;

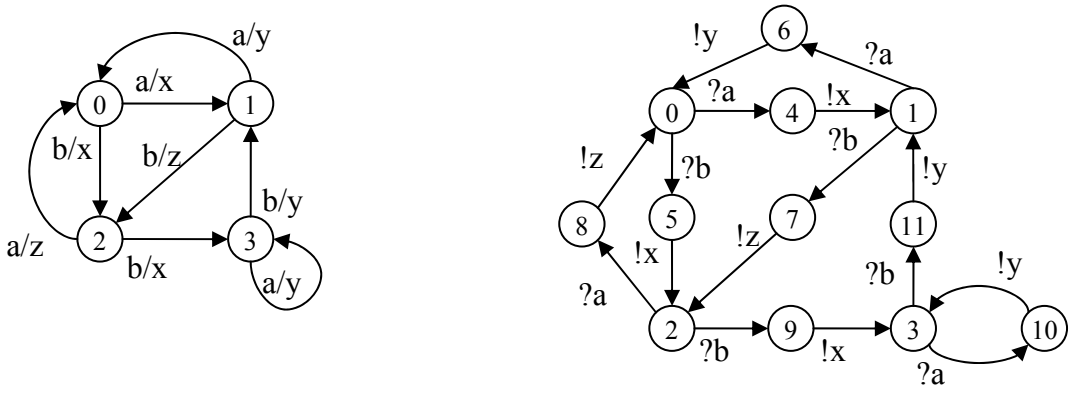
$T \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ — множество *переходов* системы. Каждый переход — тройка (s_1, a, s_2) — *выходит из* s_1, *ведет в* s_2, и помечен действием a, которое может быть стимулом, реакцией или внутренним действием. Этот переход изображают стрелкой, ведущей из s_1 и в s_2 и помеченной a. При этом, чтобы отличить стимулы от реакций, первые обычно изображают с вопросительным знаком, а вторые — с восклицательным: ?a, ?b, !x, !y.

Для систем переходов тоже определено выполнение. Сначала система находится в начальном состоянии. На вход она получает извне некоторую последовательность стимулов. При получении очередного стимула в некотором состоянии недетерминированным образом выбирается один из следующих вариантов.

- Система может выполнить переход, помеченный принятым стимулом и выходящий из текущего состояния. При этом принятый стимул выбирается из входной последовательности, и очередным стимулом становится следующий за ним.
- Система может выполнить переход, помеченный реакцией. При этом эта реакция выдается вовне.
- Система может выполнить переход, помеченный внутренним действием. При этом никаких действий, наблюдаемых извне, не происходит.

Один из подходящих в каждом случае переходов выбирается недетерминированным образом, очередным текущим состоянием становится конечное состояние выбранного перехода.

Ясно, что система переходов тоже реализует некоторое соответствие между I* и O*, которое называется ее *поведением*. Соответствия, реализуемые системами переходов, более богаты, чем реализуемые автоматами. Например, длина соответствующих последовательностей стимулов и реакций может быть различной, пустая последовательность стимулов может соответствовать непустой последовательности реакций и наоборот. На рисунке ниже показана конечная система переходов, реализующая то же поведение, что и представленный слева автомат.



Вообще верно, что любое поведение конечного автомата может быть реализовано конечной системой переходов с теми же входным и выходным алфавитами, а обратное — неверно. Некоторые конечные системы переходов не имеют конечных автоматов с таким же поведением.

Так же, как и автоматы, системы переходов могут быть бесконечными — с бесконечным множеством состояний или алфавитами.

Другие обобщения автоматов связаны с введением дополнительных структур, например, выделением части информации о состоянии в отдельные переменные, а также введением параметров стимулов и реакций. При этом получаются расширенные автоматы.

Расширенный конечный автомат — набор (S, V, P, s_0, P_0, I, n_I, X, O, n_O, Y, T), где

S — конечное множество *управляющий состояний* автомата;

V — множество, возможно бесконечное, значений внутренних данных автомата;

P — отображение конечного набора [1..n] индексов в V, P:[1..n] $\rightarrow$ V; значение P на индексе I называется значением i-й переменной автомата, которое также обозначается p_i.

s_0 — элемент S, называемый *начальным состоянием*;

P_0 — отображение [1..n] индексов в V, называемое *начальными значениями переменных*;

I — конечное множество, элементы которого называются *операциями* или *стимулами*, само I называют *входным алфавитом* автомата;

n_I — отображение I в неотрицательные числа, определяет число параметров для каждого стимула;

X — множество, возможно бесконечное, значений параметров стимулов;

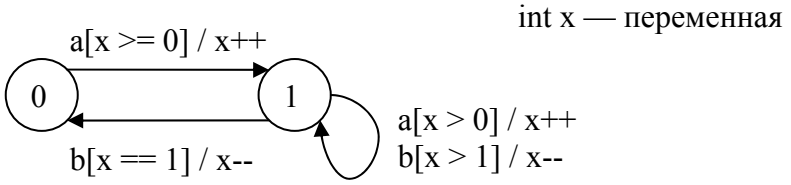
O — конечное множество, элементы которого называются *реакциями*, само O называют *выходным алфавитом* автомата;

n_O — отображение O в неотрицательные числа, определяет число параметров или данных каждой реакции;

Y — множество, возможно бесконечное, значений данных реакций;

T — множество *переходов* автомата; каждый переход t включает *начальное управляющее состояние* s_1, стимул i, *охранное условие* или *условие перехода* g_t (guard condition) — предикат на множестве $V^n \times X^n_I$, реакцию o, отображение параметров реакции $V^n \times X^n_I$ в множество Y^n_O, определяющее правила вычисления данных реакции по текущим значениям переменных и параметрам стимула, *конечное управляющее состояние* s_2, и *действие* a_t — некоторое отображение $V^n \times X^n_I$ в множество V^n, определяющее новые значения переменных.

Выполнение расширенного автомата отличается от выполнения обычного тем, что помимо текущего состояния имеются текущие значения переменных, при приходе стимула с набором аргументов охранное условие определяет, может ли быть выполнен данный переход при текущем наборе значений переменных и заданных значениях параметров стимула. Выполняемый переход выбирается недетерминировано из всех, помеченных данным стимулом, начинающихся в данном управляющем состоянии и имеющих выполненное охранное условие. При выполнении некоторого перехода новое управляющее состояние автомата равно конечному управляющему состоянию перехода, новые значения переменных определяются при помощи его действия — новое $p_i = a_t(p_1,\ldots, p_n, x_1,\ldots, x_{nI})$, значения параметров реакции — по соответствующему отображению в переходе.



Другое обобщение конечных автоматов — *взаимодействующие автоматы*.

Система взаимодействующих автоматов определяется как конечное множество автоматов, некоторые из которых связаны направленными каналами передачи данных — у каждого канала есть автомат-приемник и автомат-передатчик. Часть реакций автомата-передатчика не отдается наружу, а попадает в один из каналов, в которые он может их передать. Соответственно, принимающий автомат часть своих стимулов получает извне, а часть — из каналов, для которых он является приемником. Каждый канал устроен как очередь — посылающий автомат вставляет свою реакцию как последнее сообщение, принимающий автомат выбирает всегда первое находящееся в канале сообщение.

Иногда используются взаимодействующие расширенные автоматы, получаемые из расширенные в точности так же, как взаимодействующие получаются из обычных.

Иерархические автоматы — могут иметь структурированные состояния, одни из их состояний являются под-состояниями других. Это полезно для возможности определить переход по одному и тому же стимулу из целого множества состояний в одно. Иерархический автомат может быть преобразован в конечный при помощи процедуры уплощения — состояниями конечного автомата объявляются цепочки состояний

иерархического, в которых каждое следующее состояние является под-состоянием предыдущего, а у последнего состояния нет под-состояний.

Иерархические расширенные автоматы с дополнительной возможностью определения параллельных под-состояний — это *диаграммы переходов* (Statecharts) в рамках UML.

Помимо переменных, в автомат можно добавить таймеры — это специальные переменные, которые автономно изменяют свое значение в соответствии с ходом времени. Расширенные автоматы с таймерами называются *временными* (Timed Automata).

Поведение автомата можно определить не только для конечных, но и для бесконечных входных последовательностей. В этом случае получаются так называемые ω-*автоматы*.

В теоретической информатике используются исполнимые модели программ в виде некоторых машин — *машины Тьюринга*, *машины Поста* или другие машины, формализующие понятие универсальной или ограниченной вычислимости. Они удобны для доказательства результатов об алгоритмической сложности решения или о неразрешимости каких-либо задач, но в качестве способа моделирования сложных программных систем не используются.

Логико-алгебраические модели

Логико-алгебраические модели описывают преимущественно свойства моделируемых систем. По виду используемых формализмов их можно (несколько условно) разделить на логические и алгебраические.

Логические модели используют различные виды логических исчислений, отличающиеся операциями, которые можно применять к высказываниям для построения новых высказываний, и описывают свойства систем как набор высказываний в определенной логике.

Алгебраические модели используют алгебраические исчисления, в которых операции выполняются не над высказываниями, а над термами. Система описывается в рамках такого исчисления как множество термов (соответствующих возможным состояниям системы), на которых специальными аксиомами определяется отношение эквивалентности (какие состояния считаются одинаковыми). Часто определяются термы нескольких типов, тогда один из этих типов соответствует состояниям системы, а остальные — типам других объектов, которые можно получать с помощью операций системы.

Основные виды используемых при описании ПО алгебраических моделей следующие.

- Реляционные алгебры (см. в курсах по базам данных).
- Абстрактные типы данных (пример приведен в Лекции 3 — там описан абстрактный тип индексированного списка)
- Разнообразные алгебры процессов.
- Машины с абстрактным состоянием (Abstract State Machines, ASM, см. ниже).

Основные виды логических моделей такие.

- Исчисления высказываний и предикатов, первого и высших порядков.
- λ-исчисление первого и высших порядков.
- Временные логики, использующие операторы «раньше», «позже», «когда-нибудь в будущем» и пр.
- μ-исчисление, являющееся расширением временных логик.
- Логики явного времени, где вместо соотношений между порядком событий типа «позже»-«раньше»-«между» явно указывается время или временные интервалы, когда они имели место.

Промежуточные модели

Промежуточные модели либо определяются как исполнимые, но могут быть легко интерпретированы как логико-алгебраические или наоборот, либо сочетают в себе черты обоих основных классов моделей.

Например, алгебры процессов представляют собой алгебраические исчисления, в которых процессы представлены термами, строящимися по определенным правилам. В то же время естественная интерпретация алгебр процессов дается системами размеченных переходов, и часто между системами переходов и представляемыми ими процессами не делается различий.

Пример второго рода — *машины с абстрактным состоянием* (ASM). В качестве состояний такой машины рассматриваются различные универсальные алгебры с одной и той же сигнатурой (называемой сигнатурой машины), включающей константы true, false, undef. Переходы составляются из конечного набора элементарных действий нескольких типов.

- Смена значения операции из сигнатуры символа на некотором наборе аргументов
  $f(a_1, \ldots, a_n)$:= <выражение, составленное из операций сигнатуры>
- Условное выполнение определенных действий
  if(<условие>) <действия>
- Параллельное выполнение некоторых действий для всех объектов, обладающих некоторым свойством
  forall x : P(x) do <действия, зависящие от x>
  Приведем простой пример описания стека с помощью ASM.
- Сигнатура.
  - size : int — размер стека;
  - elements(int) : object — содержимое стека;
  - top : object — элемент в вершине стека;
  - input : {pop, push}×object — выполняемая операция.
- Начальное состояние.
  size = 0; ∀i elements(i) = undef; top = undef;
- Описание.
  if(π_1(input) = push)
  { size := size + 1; elements(size+1) := π_2(input); top := π_2(input); }
  if(π_1(input) = pop)
  { size := size - 1; elements(size) := undef; top := elements(size-1); }

Еще один вид промежуточных моделей, использующий элементы исполнимых моделей и логических — *программные контракты* (software contracts).

Программный контракт описывает некоторый компонент с определенным интерфейсом. Интерфейс представляет собой конечное множество операций. Каждая операция имеет имя и набор параметров определенных типов.

Программный контракт компонента состоит из следующих элементов.

- Описание структуры данных внутреннего состояния компонента.
  Это описание включает определение элементов данных и их типов, а также инварианты.
  Каждый инвариант — это предикат, зависящий от описанных элементов данных. Если такой предикат выполнен, данные состояния компонента корректны. Иначе — нарушены их ограничения целостности и пользоваться таким компонентом нельзя.
- Описание структур данных всех используемых типов.
  Для каждого типа также определяются элементы данных, их типы и инварианты данного типа.

- Описание контракта для каждой операции.
    - Предусловие.
      Предусловие определяет ситуации, в которых операцию можно вызывать. Оно представляет собой предикат, зависящий от внутреннего состояния компонента и аргументов операции.
    - Постусловие.
      Постусловие описывает требования к корректным результатам работы операции. Оно представляет собой предикат, зависящий от внутреннего состояния в момент вызова операции, аргументов, с которыми она была вызвана, возвращаемого ею результата и состояния компонента сразу после вызова.

В рамках программного контракта соединяется описание состояний компонента, характерное для исполнимых моделей, с декларативным описанием поведения операций, которые, вообще говоря, нельзя выполнить на какой-то машине.

Основные методы построения тестов

Основные методы построения тестов можно разделить на следующие группы.

- *Вероятностные методы*.
  Эти методы основаны на вероятностной генерации тестовых воздействий в соответствии с определенными распределениями.
  Вероятностные методы хорошо автоматизируются и являются наименее трудоемкими. Однако обеспечиваемая ими полнота тестирования варьируется случайным и плохо предсказуемым заранее образом — в одних случаях оказывается достаточно хорошей, в других очень плохой. С помощью вероятностных методов хорошо находятся случайные ошибки и опечатки, все другие виды ошибок — не очень хорошо. Используются они, когда о тестируемой системе почти ничего неизвестно, а получение дополнительной информации и построение тестов другими методами не может быть осуществлено в рамках проекта.
- *Методы, нацеленные на полное покрытие*.
  В рамках этих методов тесты строятся целенаправленно таким образом, чтобы обеспечить покрытие выделенных критерием покрытия классов ситуаций.
  Эти методы автоматизируются плохо, в основном используются при ручной разработке тестов и достаточно трудоемки. Обеспечиваемая ими полнота при адекватном выборе критерия полноты тестирования очень высока. Позволяют находить любые виды ошибок. Используются при наличии практически полной информации о системе, достаточных ресурсов и необходимости провести систематическое и аккуратное тестирование.
- *Комбинаторные методы*.
  Эти методы основаны на разбиении тестовых воздействий на некоторые элементы и составлении различных комбинаций из этих элементов по определенным правилам с целью получить достаточно систематический перебор тестовых воздействий.
  Они хорошо автоматизируются, более трудоемки, чем вероятностные, но значительно проще, чем нацеленные на покрытие методы. Обеспечиваемая ими полнота тестирования возрастает вместе с количеством получаемых тестов и может быть достаточно высокой. Позволяют находить случайные и достаточно простые ошибки. Используются при наличии лишь самой поверхностной информации о работе системы и при ограниченных ресурсах на тестирование.
- *Автоматные методы*.
  Автоматные методы построения тестов используют модели тестируемой системы в виде конечных автоматов и их различных обобщений.
  Они хорошо автоматизируются, но требуют определенных затрат на выполнение

тестов. Обеспечивают очень высокие значения полноты тестирования. Позволяют находить ошибки разных видов, в том числе и очень сложные, практически не обнаруживаемые другими методами. Используются при наличии четкой и полной информации о системе, достаточных ресурсов и повышенных требований к ее надежности и качеству.

- *Алгебраические методы*.
  Эти методы используют алгебраическое описание тестируемой системы.
  Хорошо автоматизируются, требуют средних затрат ресурсов. Обеспечивают средние показатели полноты тестирования. Обнаруживают различные виды ошибок. На практике почти никогда не используются, потому что требуют описать тестируемую систему в виде алгебраической системы или набора абстрактных типов данных с полным набором аксиом.
  Такие методы довольно экзотичны и приведены здесь для полноты информации о различных подходах к созданию тестов.

Далее разные виды методов построения тестов описаны более детально.

В этой лекции обсуждаются вероятностные, алгебраические методы и методы, нацеленные на покрытие.

Следующая лекция посвящена различным комбинаторным методам построения тестов. Следующие за ней две лекции — автоматным техникам тестирования.

Вероятностные методы

Вероятностные методы нацелены прежде всего на снижение трудоемкости разработки тестов при минимуме информации о тестируемой системе. При этом они пытаются максимизировать вероятность обнаружения достаточно серьезных ошибок.

В самом простом случае тестовые воздействия генерируются совершенно случайным образом.

При более обоснованном подходе для построения более-менее адекватных тестов делаются предположения о распределении вероятностей использования тех или иных воздействий. Обоснованность таких предположений можно проверить, только имея профиль использования системы — достаточно представительные фактические данные о том, какие операции и функции системы с какой частотой используются на практике. Если известна вероятность использования заданного воздействия при реальной работе системы, можно выполнять это воздействие в тестах ровно с такой же вероятностью.

Если известно распределение возможного ущерба от ошибок при различных воздействиях, то можно использовать и его — вероятность выполнения воздействия в тесте имеет смысл сделать пропорциональной и возможному ущербу при некорректной работе системы в результате этого воздействия. Это позволит минимизировать ущерб от необнаруженных в ходе тестирования ошибок.

Наконец, дополнительным фактором могут быть трудозатраты на устранение ошибок. Если известно распределение затрат ресурсов на исправление ошибок, возникающих при различных воздействиях, стоит сделать вероятность воздействия в тесте пропорциональной таким затратам. Теоретически, это позволит быстрее найти ошибки, для исправления которых необходимы значительные усилия.

Таким образом, распределение вероятностей воздействий во время теста формируется пропорционально распределениям вероятностей использования этих воздействий при эксплуатации системы, размеру возможного ущерба от ошибки, вызванной этими воздействиями и затратам на исправление этих ошибок. Если какие-то из этих распределений неизвестны, на их месте используются равномерные распределения.

Необходимая вероятность выполнения определенного воздействия в тесте вычисляется как произведение вероятности его использования на величину возможного ущерба при

неправильной работе системы и на величину затрат на устранение ошибки, если она будет обнаружена, деленное на сумму всех таких произведений для всех воздействий.

$$P_a = \frac{P_a^{use} \cdot V_a^{risk} \cdot V_a^{debug}}{\sum_b P_b^{use} \cdot V_b^{risk} \cdot V_b^{debug}},$$

или, в интегральной форме для плотностей вероятности

$$p = \frac{p^{use} \cdot v^{risk} \cdot v^{debug}}{\int p^{use} \cdot v^{risk} \cdot v^{debug}}$$

Стоит, однако, отметить, что в подавляющем большинстве случаев адекватно оценить вероятности использования различных воздействий, величину возможного ущерба или затрат на исправления ошибки весьма сложно. Для новых, только разрабатываемых систем, таких данных нет вообще. Для новых версий уже долгое время эксплуатировавшихся систем можно использовать оценки по аналогии со старой системой, но они также могут быть не аккуратными. Поэтому на практике вероятностное построение тестов часто основывается на слабо обоснованных предположениях о равномерности распределения вероятностей воздействий, если их различных видов не много, или о нормальном или пуассоновском законе их распределения, если воздействия включают числовые параметры. Проводимое так тестирование способно выявить некоторые ошибки, но дать сколь-либо серьезные гарантии надежности системы оно не может.

Поэтому вероятностные методы построения тестов применяются в тех случаях, когда необходимо провести тестирование с минимальными затратами на него и практически без всякой информации об особенностях реализации или требований к системе. Они позволяют получить много тестов с очень небольшими усилиями, и в этом основное достоинство таких методов. Однако, достигаемая при этом полнота тестирования случайна и часто непредсказуема заранее — если повезет, можно получить хорошо оттестированную систему, но убедиться в этом можно только при помощи других методов тестирования.

Обнаружить сложные ошибки, особенно ошибки, возникающие из непонимания специфических деталей требований, при помощи вероятностного тестирования можно только случайно. В основном, обнаруживаемые таким способом ошибки являются достаточно простыми опечатками или, наоборот, возникают из-за настолько больших пробелов в понимании задач, решаемых тестируемой программной системой, что обнаружить их достаточно просто.

Методы, нацеленные на покрытие

При использовании методов, нацеленных на покрытие, тесты строятся с основной целью — покрыть ситуации, выделяемые выбранной метрикой тестового покрытия. Такие методы тяжело автоматизировать и, в основном, с их помощью тесты разрабатываются вручную. На практике это означает, что проводится анализ достижимости различных ситуаций, которые определяются метрикой покрытия, и для тех из них, которые достижимы, соответствующие тестовые данные просто подбираются.

Основные достоинства таких методов — высокие значения полноты тестирования при аккуратном и систематичном применении и возможность нацеливать тесты на различные виды ошибок. Недостатки — довольно высокая трудоемкость разработки тестов, необходимость анализировать достижимость различных сложных ситуаций.

В качестве метрик покрытия чаще всего используются структурные метрики покрытия, метрики покрытия структуры входных данных и метрики покрытия требований (см. Лекцию 3, там же можно найти ряд примеров наборов тестовых данных, полностью покрывающих выделенные ситуации).

Доменное тестирование

Одним из специфических методов построения тестов, нацеленных на покрытие требований, является *доменное тестирование*.

Доменное тестирование используется для тестирования функций, зависящих от числовых параметров или числовых элементов состояния системы. Или эти параметры и элементы состояния сами по себе могут быть не числовыми, но достаточно естественным образом отображаться на числа и покрывать при этом большие области числовых значений. В этом случае оно позволяет обеспечить высокие значения полноты тестирования, определяемой на основе требований, с помощью небольшого числа тестов. Для нечисловых параметров, имеющих небольшие множества возможных значений, оно не столь эффективно. При доменном тестировании выполняются следующие действия.

- Область определения тестируемой функции разбивается на подобласти, в которых требования к ее результатам формулируются несколько по-разному. Обычно эта разность проявляется в различных выражениях, используемых для описания результатов функции в разных подобластях. Выделенные подобласти значений параметров обычно образуют подобласти многомерного пространства.
- Определяются все компоненты связности выделенных областей, все границы областей, компоненты связности границ, границы границ, и т.д.
  Обычные границы называются границами первого порядка, их границы — границами второго порядка, и. т.п.
  Пример: если функция имеет три числовых параметра, чаще всего ее область определения — область в трехмерном пространстве. Границы первого порядка — это поверхности, отделяющие область определения и разделяющие ее на подобласти. Границы второго порядка — кривые, по которым пересекаются или соприкасаются границы первого порядка, или которыми они разбиваются на гладкие куски. Границы третьего порядка — точки, являющиеся точками пересечения или касания границ второго порядка, а также отделяющие границы второго порядка друг от друга и разбивающие их на гладкие куски.
- Тестовые данные подбираются следующим образом.
  Областью здесь называется компонента связности одной из выделенных подобластей области определения, а также компонента связности границы любого порядка.
    - Для всякой области должен быть набор значений параметров, лежащий внутри этой области. Обычно его стараются выбирать где-то «посередине», ближе к центру масс области.
    - Для всякой компоненты связности границы любого порядка любой области, если область не включает эту компоненту границы (т.е. сама граница не входит в область определения или принадлежит другой подобласти), должен быть набор значений параметров, лежащий в этой области как можно ближе к границе.

  Для области в трехмерном пространстве должна быть выбрана точка внутри нее. Для всякой поверхности, являющейся гладким куском ее границы, если эта поверхность входит в область, выбирается точка, лежащая на ней, если нет — выбирается точка, лежащая внутри области, но как можно ближе к этой поверхности.
  Для всякой кривой, являющейся гладким куском границы 2-го порядка, если эта кривая входит в область, выбирается точка на ней, если нет — для всякого куска поверхности-границы, ограничиваемого этой кривой и входящего в область, выбирается точка, лежащая на этом куске как можно ближе к кривой, для всякого куска поверхности, не входящего в область, выбирается точка, лежащая внутри области как можно ближе к этому куску поверхности и этой кривой.
  Для всякой точки, входящей в границу третьего порядка, если она входит в область, выбирается эта точка, если нет — выбираются точки вблизи нее, лежащие на всех

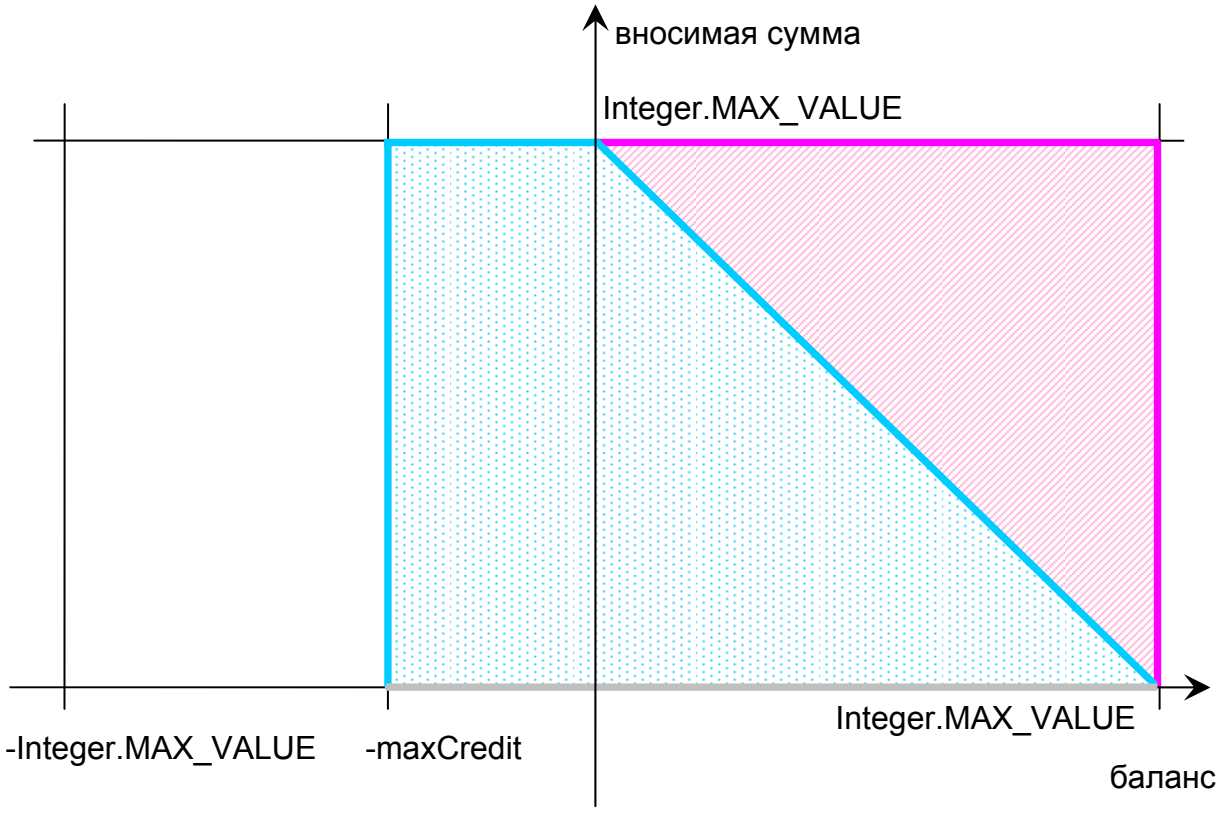
содержащих ее кривых-границах, входящих в область, на всех поверхностях, входящих в область, чьи границы не все входят в нее, и в самой области, если есть ее поверхность-граница, проходящая через эту точку и не входящая в область.

Пример.

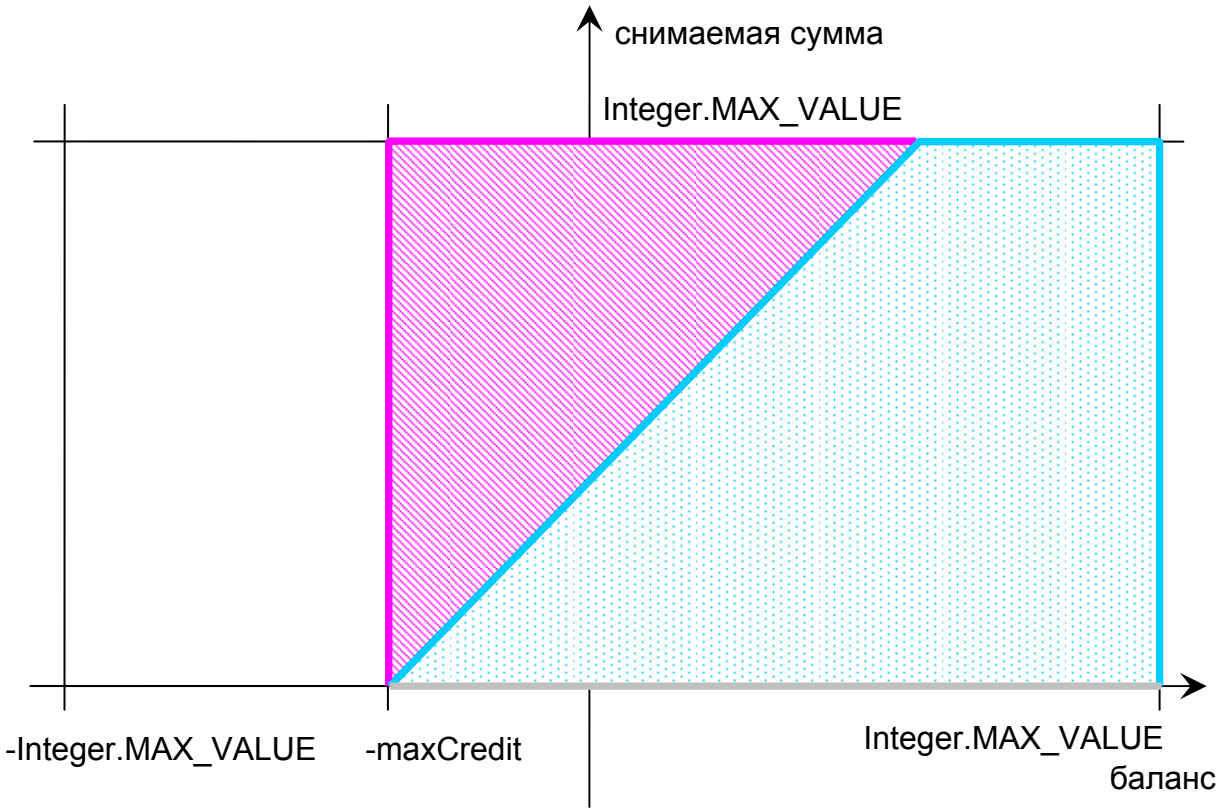
Пусть тестируется реализация банковского счета с ограниченным кредитом. В тестируемом компоненте две операции — положить деньги на счет и снять деньги со счета. Пусть снимаемая/заносимая сумма и текущий баланс счета представлены 32-битными целыми числами.

- Выделение областей определения операций и их естественных подобластей дает следующие результаты.
    - Операция внесения денег на счет добавляет вносимую сумму к балансу для тех сумм и балансов, для которых результирующая сумма не превышает максимального 32-битного числа.
      Если итоговый баланс может превысить максимум, эта операция не должна выполнять никаких действий, но должна вернуть какое-то указание на слишком большую сумму на счете для выполнения этой операции.
    - Операция снятия денег со счета вычитает снимаемую сумму из баланса, если итог не станет меньше минимально возможного баланса, соответствующего максимально возможному кредиту. Если кредит при этом превышается, баланс не изменяется, а возвращается указание на недостаточность суммы на счете для выполнения этой операции.

  Для каждой из операций выделены две подобласти различного поведения. Областью определения каждой из них служит все множество пар возможных значений баланса и вносимой/снимаемой суммы, которая должна быть положительной.
  Полученные области изображены на рисунках ниже.

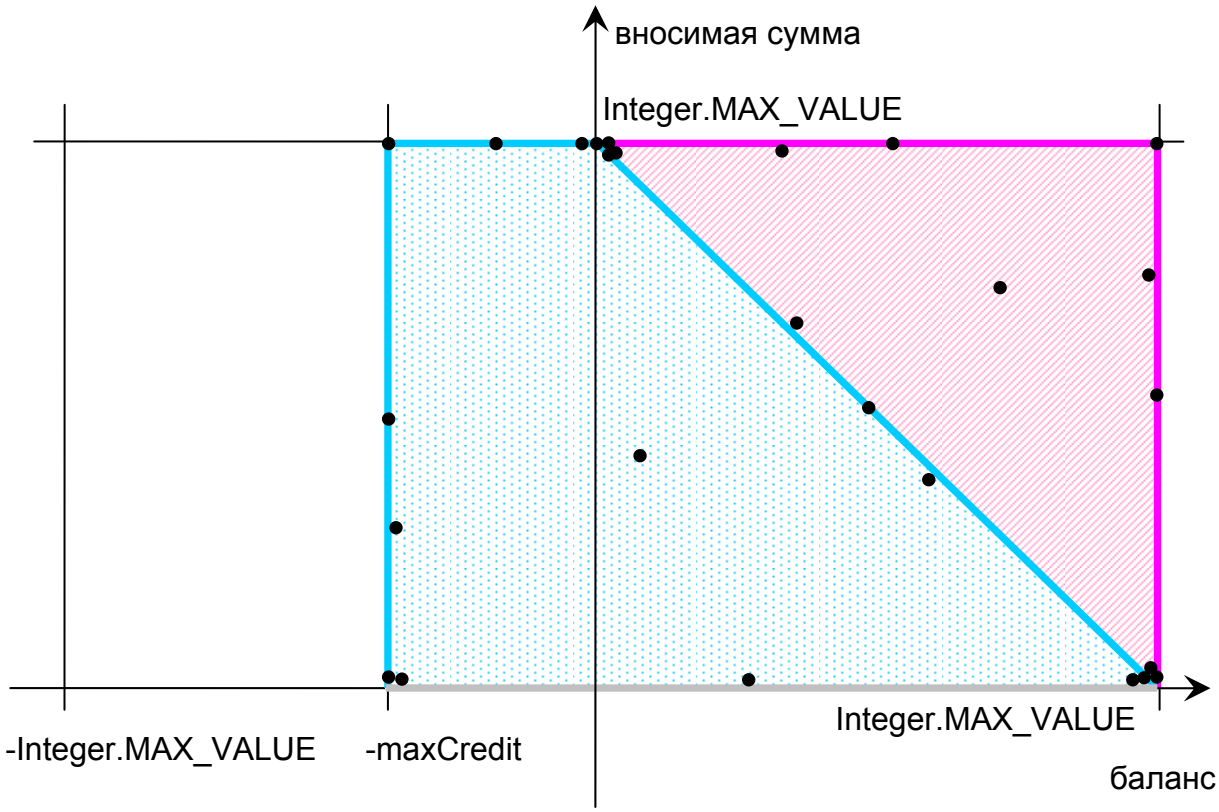


Разбиение области определения для операции внесения денег на счет. Граница голубого или красного цвета означает, что она включается в голубую или, соответственно, красную область



Разбиение области определения для операции снятия денег со счета.

- Для обеих операций получилось по две подобласти, представленные на рисунках разными цветами. Голубым цветом выделена область нормального поведения, красным — область балансов и сумм, при которых выполнение операции в нормальном режиме невозможно.
  В обоих случаях область голубая область имеет границу, состоящую из трех сегментов. Красные области в обоих случаях ограничены двумя сегментами и соприкасаются с одним из сегментов голубой области.
- Значения тестовых данных (в данном случае, включающие и значение баланса, являющееся состоянием тестируемого компонента), получаемые на основе эвристики покрытия всех подобластей, границ и окрестностей границ, изображены для операции внесения денег на счет ниже (большие черные точки). Для второй операции они получаются аналогичным образом. Заметим, что больше всего тестовыми данными должны быть покрыты окрестности границы, разделяющей выделенные подобласти.
  Некоторые выбранные точки (внутри голубой области вблизи голубой границы и внутри красной области вблизи красных границ) не являются необходимыми с точки зрения эвристик доменного тестирования, но часто такие точки добавляются (в количестве, не превосходящим число точек, выбранных по основной эвристике) для отслеживания других видов ошибок.



Более сложный пример.

Рассмотрим программу решения квадратного уравнения $ax^2+bx+c=0$. Можно считать, что программа оформлена в виде функции int solveEquation(double a, double b, double c, double *x1, double *x2). Первые три ее параметра являются коэффициентами уравнения, последние два предназначены для возвращения его решений, а целочисленный результат — это количество решений.

Условие	$a = b = c = 0$	$a = b = 0, c \neq 0$	$a = 0, b \neq 0$	$a \neq 0, b^2-4ac < 0$	$a \neq 0, b^2-4ac = 0$	$a \neq 0, b^2-4ac > 0$
Число решений	∞	0	1	0	1	2
Результат	–1	0	1	0	1	2

Таблица 1. Подобласти области определения программы, решающей квадратное уравнение.

Если решений два, возвращается 2, и указатели на оба решения. Если есть только одно решение, возвращается 1 и оба указателя x1 и x2 указывают на одно и то же значение. Если решений нет, возвращается 0, и значения по указателям произвольны. Если решений бесконечно много, возвращается –1.

Для этой функции четко выделены подмножества области определения, представленные в таблице выше. Они определяют разбиение ее пространства параметров на 4 подобласти с помощью конуса нулевого дискриминанта и плоскости нулевого первого коэффициента и еще 7 компонентов их границ. Итого выделяется 11 подобластей и элементов границ.

- внутренность конуса $b^2 = 4ac$ сверху от плоскости $a = 0$ — $a > 0$ и $b^2-4ac > 0$;
- внутренность конуса снизу от плоскости — $a < 0$ и $b^2-4ac > 0$;
- вне конуса и сверху от плоскости — $a > 0$ и $b^2-4ac < 0$;
- вне конуса и снизу от плоскости — $a < 0$ и $b^2-4ac < 0$;

- конус сверху от плоскости —$a > 0$ и $b^2-4ac = 0$;
- конус снизу от плоскости —$a < 0$ и $b^2-4ac = 0$;
- полуплоскость — $a = 0$ и $b > 0$;
- полуплоскость — $a = 0$ и $b < 0$;
- полуприямая — $a = 0$, $b = 0$ и $c > 0$;
- полуприямая — $a = 0$, $b = 0$ и $c < 0$;
- точка, вершина конуса — $a = 0$, $b = 0$ и $c = 0$.

Для каждой выделенной подобласти, каждой границы (куска поверхности конуса, полуплоскости, полупрямой или точки), а также для окрестностей границ необходимо подобрать наборы значений параметров, попадающих в соответствующее множество.

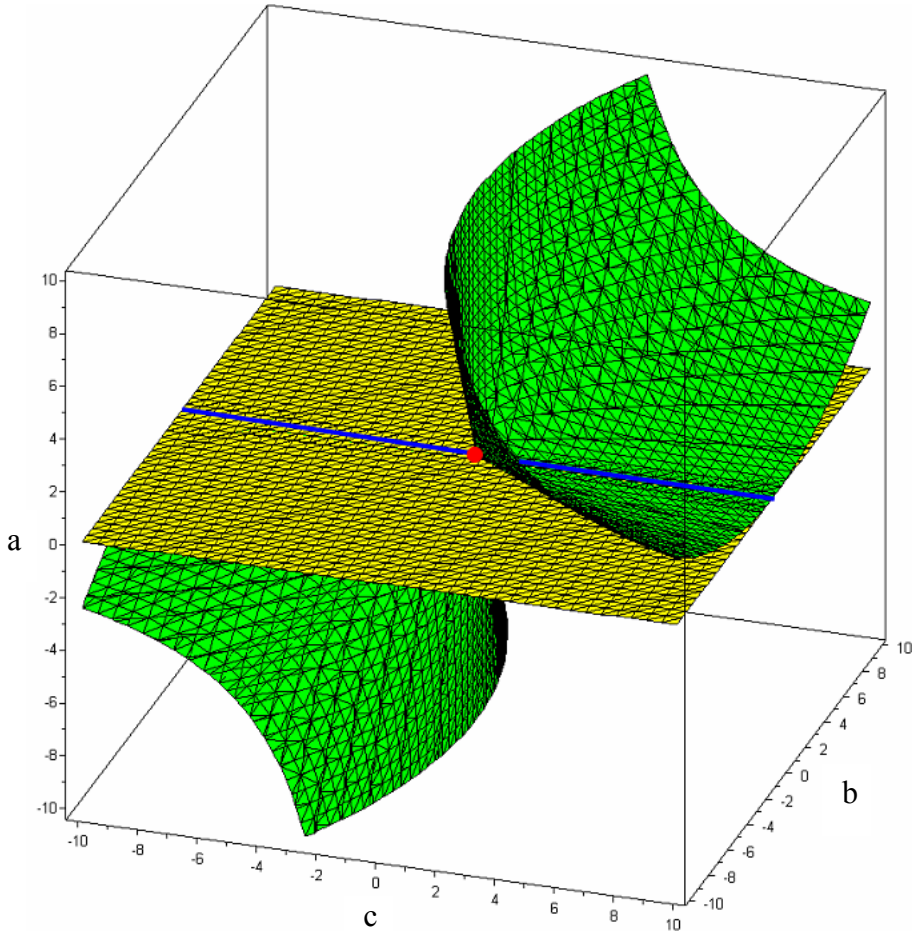


Рисунок 1. Подобласти параметров для программы, решающей квадратные уравнения.

Техники автоматизации построения тестов, нацеленных на покрытие

Как уже говорилось, методы, нацеленные на покрытие, плохо автоматизируются. Однако, есть несколько приемов, которые в ряде случаев позволяют строить такие тесты автоматически.

- Если для каждой из выделенных ситуаций написать специальную функцию, возвращающую 1, если набор значений параметров попадает в соответствующую ситуацию, и 0 иначе, можно выделять подходящие наборы значений из некоторого большого их множества, фильтруя его с помощью таких функций.
- Для предиката, определяющего условия попадания в заданную ситуацию, можно написать программу на одном из языков логического программирования, например, Prolog, выполнение которой будет находить значения параметров, удовлетворяющие этому предикату, а значит, покрывающие соответствующую ситуацию.

Помимо Prolog'a, для решения этой задачи можно использовать другие методы программирования с логическими ограничениями (Constraint Logic Programming).

- В ряде случае для построения наборов тестовых данных, покрывающих много разных ситуаций, выделяемых определенным критерием покрытия, можно использовать генетические алгоритмы.
  При этом геном считается последовательность операций, выполняемых над тестируемым компонентом и данными, и заканчивающаяся вызовом некоторой операции компонента с подготовленными данными. Оценочной функцией всего набора нужно считать получаемое набором тестов покрытие по выделенному критерию, а оценочной функцией одного теста — его близость к достижению еще не покрытых ситуаций.

Алгебраические методы

Алгебраические методы построения тестов достаточно экзотичны и очень редко используются на практике. Основной их недостаток — необходимость иметь описание тестируемых компонентов как абстрактных типов данных с полным набором аксиом, описывающих соотношения между операциями. Получить такие описания для практически значимых систем очень тяжело.

Эти методы дают средние значения полноты тестирования и позволяют находить ошибки от простых до средней сложности. Достаточно сложную и специфическую ошибку с их помощью найти тяжело.

Один из алгебраических методов построения тестов состоит в следующем.

- Выбирается некоторый набор стартовых цепочек операций.
- Для каждой стартовой цепочки просматриваются все ее начала. Если одна из начальных цепочек может быть преобразована в эквивалентный вид при помощи одной из аксиом, это преобразование выполняется над всей цепочкой.
  В результате для каждой стартовой цепочки получается множество цепочек, эквивалентных ей. Поскольку они получены с помощью однократного применения аксиом, их можно назвать цепочками первого порядка.
  Если количество полученных цепочек невелико, к каждой из них можно применить ту же технику, получив цепочки 2-го порядка, и т.д.
- Тестирование состоит в последовательном выполнении одной из стартовых цепочек и всех, эквивалентных ей, со сравнением получаемых результатов.

Пример.

Рассмотрим алгебраическое описание списка, приведенное в Лекции 3.

[].size() = 0

[X.size()] ≡ [X]

(i <= X.size()) => X.add(i, o).size() = X.size()+1

(i < X.size()) => X.remove(i).size() = X.size()–1

(i < X.size()) => [X.get(i)] ≡ [X]

(i, j <= X.size() & i < j) => [X.add(i, o1).add(j, o2)] ≡ [X.add(j–1, o2).add(i, o1)]

(i <= X.size()) => [X.add(i, o1).add(i, o2)] ≡ [X.add(i, o2).add(i+1, o1)]

(i <= X.size()) => [X.add(i, o).remove(i)] ≡ [X]

(i, j <= X.size() & i < j) => [X.add(i, o).remove(j)] ≡ [X.remove(j–1).add(i, o)]

(i, j <= X.size() & i > j) => [X.add(i, o).remove(j)] ≡ [X.remove(j).add(i, o)]

(i <= X.size()) => X.add(i, o).get(i) = o

(i, j <= X.size() & i < j) => X.add(i, o).get(j) = X.get(j-1)

(i, j <= X.size() & i > j) => X.add(i, o).get(j) = X.get(j)

(i, j < X.size()-1 & i < j) => [X.remove(i).remove(j)] ≡ [X.remove(j+1).remove(i)]

(i, j < X.size() & i <= j) => X.remove(i).get(j) = X.get(j+1)

(i, j < X.size() & i > j) => X.remove(i).get(j) = X.get(j)

В качестве стартовой возьмем одну цепочку [].add(0, o1).add(1, o2).

Эквивалентными ей цепочками будут следующие.

[].add(0, o1).add(1, o2).size()

[].add(0, o1).add(1, o2).get(0)

[].add(0, o1).add(1, o2).get(1)

[].add(0, o2).add(0, o1)

[].add(0, o1).add(1, o2).add(0, o3).remove(0)

[].add(0, o1).add(1, o2).add(1, o3).remove(1)

[].add(0, o1).add(1, o2).add(2, o3).remove(2)

Цепочек второго порядка получится уже достаточно много.

Чтобы проверить эквивалентность получаемых списков можно применить к ним все операции, возвращающие некоторые значения, и сравнить результаты. В данном случае это операции size(), get(0), get(1).

Таким образом, в нашем случае нужно взять стартовую цепочку X, каждую из полученных эквивалентных ей цепочек Y и сравнить их при помощи следующих проверок.

X.size() == Y.size() && X.get(0) == Y.get(0) && X.get(1) == Y.get(1)

Литература

Тестирование на основе моделей

В. В. Кулямин

Лекция 5. Комбинаторные методы построения тестов

Комбинаторные методы построения тестов основаны на разделении каждого тестового воздействия на ряд элементов и построении тестов как всевозможных комбинаций полученных элементов, объединяемых по определенным правилам.

Комбинаторные методы дают более высокую полноту покрытия, чем вероятностные, и при этом требуют ненамного больше ресурсов. Кроме того, они хорошо автоматизируются. Однако, с помощью комбинаторных методов трудно найти ошибки в очень специфических ситуациях, требующих учета многих факторов, а трудозатраты на их применение при учете возрастающего числа факторов растут гораздо быстрее.

Одним из примеров комбинаторных методов является *тестирование по разбиениям на категории* (category partition testing). В рамках этого подхода для построения тестов выполняются следующие действия.

- Выделяется набор операций тестируемой системы, обращения к которым должны производиться в тестах.
- Для каждой тестируемой операции анализируются требования к ней и на основе этого анализа выделяются, дополнительно к ее параметрам, некоторые факторы (внешние условия или свойства внутреннего состояния системы), от которых может зависеть ее поведение.
- Возможные значения каждого параметра операции или фактора, влияющего на ее поведения, классифицируются — разбиваются на конечное множество *категорий*. Категории выделяются таким образом, чтобы изменение значения параметра или фактора в рамках одной категории слабо изменяло требования к работе операции.
- Определяются зависимости между полученными категориями, взаимосвязи между значениями различных параметров и факторов, а также недопустимые комбинации их значений.
- Тестовые ситуации строятся как возможные комбинации категорий значений параметров и факторов. Для каждой такой комбинации определяются соответствующие конкретные значения параметров и способ достижения соответствующих значений факторов (тестовые последовательности и изменения внешних условий).

Достаточно сильно похож на эту технику и метод построения тестов *на основе дерева классификации* (classification tree method). Он используется при наличии большого количества факторов, влияющих на поведение тестируемой системы.

Сначала выделяется набор наиболее заметных факторов, влияние которых на поведение тестируемой системы достаточно сильно. Такие факторы называются в рамках этого метода *аспектами*. Для каждого из этих факторов пытаются определить разбиение его возможных значений на группы, в рамках которых требования к поведению системы меняются слабо. При этом могут быть выявлены другие факторы, чье влияние на систему становится существенным, если один из базовых аспектов зафиксирован.

Базовые аспекты образуют вершины дерева классификации, непосредственно связанные с его корнем. Их классы и вторичные аспекты привязываются к базовым аспектам, и т.д., пока все существенные факторы не будут найдены и классифицированы.

После построения дерева классификации необходимо определить зависимости между выделенными классами значений аспектов.

Тесты строятся как возможные комбинации классов значений аспектов, соответствующих листовым вершинам дерева.

Например, при тестировании встроенного программного обеспечения управления автомобилем, возможна следующая классификация ситуаций.

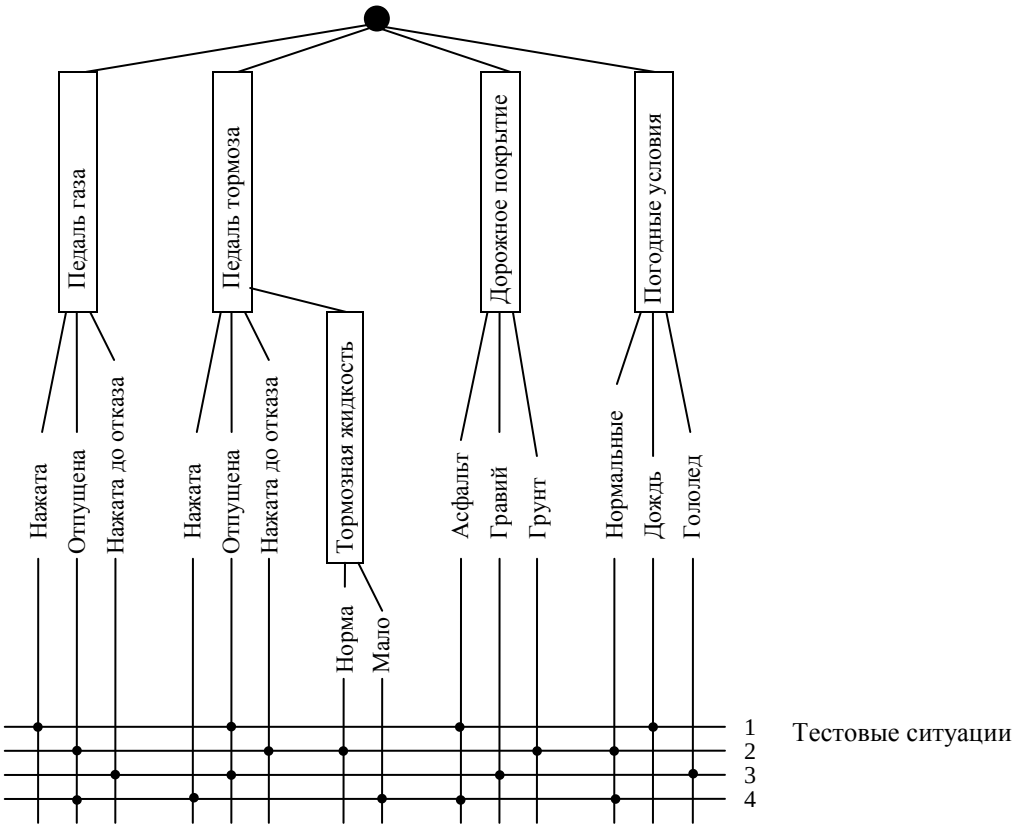


Рисунок 1. Пример использования дерева классификаций.

Тестирование на основе грамматик

Другой часто применяемой комбинаторной техникой создания тестов является *синтаксическое тестирование* или *тестирование на основе грамматик*. Оно используется в тех случаях, когда структуру множества возможных воздействий на тестируемую систему можно описать при помощи формальной грамматики.

Проще всего эту возможность заметить там, где действительно существуют некоторый специальный язык — например, в системах, работающих с языками программирования, языками запросов (SQL и др.), языками для описания структуры документов (XML, HTML и пр.) или с регулярными выражениями. Однако с помощью формальных грамматик можно описывать и структуру пользовательского интерфейса. Наиболее наглядный пример это — структура опций утилит в операционных системах типа UNIX. Чуть менее привычно использование грамматик для описания графических интерфейсов пользователя (GUI).

Рассмотрим для этого простой пример интерфейса поиска фраз и ключевых слов в стандарте POSIX. Этот интерфейс реализован при помощи разделения окна браузера на три области: основной текст, индекс и форма поиска. Форма поиска может иметь два вида, которые представлены на рисунках ниже.

Эта форма в начальном состоянии показана на Рис. 2, при выборе ссылки Word Search ее вид меняется на изображенный на Рис. 3.

Рисунок 2. Форма поиска — основной вид.

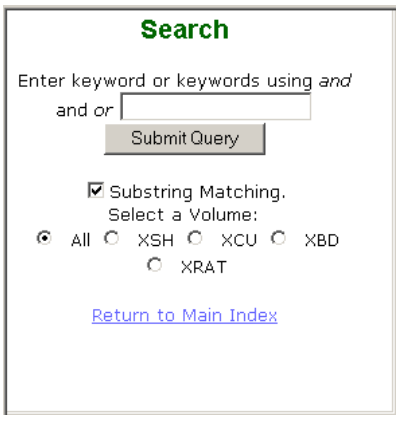


Рисунок 3. Форма поиска ключевых слов.

Все действия пользователя в рамках этой формы поиска можно описать при помощи следующей грамматики.

Chapter ::= Frontmatter | BaseDefinitions | SystemInterfaces | ShellAndUtilities | Rationale

Organization ::= Alphabetical | Topic

MainOption ::= *Chapter* | *Organization*

Section :: = All | XSH | XCU | XBD | XRAT

MainIndexAction ::= (*MainOption*)* (<Phrase> Search)* | *WordSearchAction* ReturnToMainIndex

WordSearchAction ::= *MainIndexAction* WordSearch | *WordSearchAction* (SubstringMatching)? *Section* <Keywords> SubmitQuery

<Keywords> ::= <Word> | <Keywords> "or" <Word> | <Keywords> "and" <Word>

<Phrase> ::= (<Word>)*

В приведенном описании нажатия на кнопки, ссылки или включение/выключение флажков на форме выделены таким шрифтом, как BaseDefinitions или Search. <Keywords>, <Word> и <Phrase> — это различные виды фраз, которые можно ввести в поля ввода формы.

При построении тестов на основе формальных грамматик рассматриваются тестовые ситуации, соответствующие различным вариантам построения предложений в этой грамматике. Похожие наборы вариантов рассматриваются и при определении метрик тестового покрытия на основе грамматик (см. Лекцию 3).

Каждая альтернатива дает столько вариантов, сколько в ней участвует. Опциональный элемент дает два варианта — его присутствие и его отсутствие. Список может давать несколько разных наборов вариантов — можно рассматривать только пустые и непустые списки, а можно рассматривать списки длины 0, 1, 2 и все остальные.

Например, возможные варианты раскрытия правила X ::= (A)? (B | C) D (E)*, можно перечислить так.

ABD CDE ACDEE

Выше перечислены предложения, которые покрывают только каждую из имеющихся альтернатив. Ниже — варианты раскрытия этого же правила, покрывающие все сочетания вариантов раскрытия отдельных альтернатив.

BD BDE BDEE ABD ABDE ABDEE

CD CDE CDEE ACD ACDE ACDEE

Заметим, что здесь возникает много размерностей пространства вариантов, поскольку многие альтернативы могут выбираться независимо от остальных. Поэтому возможны разные стратегии сочетания вариантов — просто раскрыть каждую альтернативу всеми возможными способами, не обращая внимания на их сочетания, перебрать все пары

сочетаний возможных раскрытий соседних альтернатив, перебрать все сочетания возможных раскрытий альтернатив в каждом правиле и пр.

Покрывающие наборы

Покрывающие наборы дают инструмент для систематического перебора различных комбинаций значений параметров или факторов.

Рассмотрим сначала небольшой пример. Предположим, мы тестируем одну операцию, на работу которой может оказать влияние несколько факторов. Примером может быть печать Web-станицы из браузера Интернет, в качестве существенных факторов для которой могут выступать объем печатаемого документа, наличие или отсутствие цветных картинок, размер бумаги, производитель используемого принтера, разновидность используемого браузера, операционная система, которой мы при этом пользуемся.

Чтобы применить комбинирование значений этих факторов, для каждого фактора должно иметься лишь небольшое число различимых значений. Предположим, что, поразмыслив, мы выбрали следующие значения.

Объем	Наличие цветных рисунков	Размер бумаги	Производитель принтера	Браузер	Операционная система
1 страница	Нет цветных рисунков	A4	HP	Internet Explorer	Windows Me
2 страницы	Есть цветные рисунки	A5	Epson	Mozilla Firefox	Windows 2000
7 страниц		B5	Canon	Opera	Windows XP
		Letter	Xerox		Linux SUSE 10.0
		Envelop C5			Linux RHEL 4.0

Таблица 1. Значения параметров для тестирования печати Web-страницы.

Если теперь попробовать составить все возможные комбинации значений факторов, получится 3·2·5·4·3·5 = 1800 тестов. Это не чересчур много, но ясно, что выполнение их всех потребует значительных затрат.

Однако можно существенно сократить эти затраты, если учесть, что подавляющее большинство ошибок в таких ситуациях (до 70%) связано с определенными комбинациями всего лишь двух факторов, то есть, если тесты будут содержать *все возможные комбинации пар значений факторов*, большая часть ошибок будет ими выявлена.

Принятие такого подхода позволяет выполнить лишь небольшое количество тестов, таких, что в них задействованы *все пары значений различных факторов*. В данном примере один из возможных минимальных наборов тестов представлен в Таблице 2.

Показанный набор минимален, поскольку для использования всех сочетаний размеров бумаги и операционных систем необходимо не менее 25 тестов.

Можно пойти еще дальше и попробовать составить тестовый набор так, чтобы он по-прежнему оставался небольшим, но содержал уже *все различные тройки значений факторов*. При этом потребуется не менее 100 = 5·5·4 тестов, что все же существенно меньше, чем 1800 (такой набор из 100 тестов действительно существует).

1	1 страница	Нет цветных рисунков	A4	HP	Internet Explorer	Linux RHEL 4.0
2	1 страница	Нет цветных рисунков	A4	HP	Opera	Windows Me
3	1 страница	Нет цветных рисунков	A5	Epson	Internet Explorer	Windows 2000
4	1 страница	Нет цветных рисунков	Envelop C5	Canon	Internet Explorer	Linux RHEL 4.0
5	1 страница	Есть цветные рисунки	B5	Epson	Opera	Windows XP
6	1 страница	Есть цветные рисунки	Letter	HP	Mozilla Firefox	Windows 2000
7	1 страница	Есть цветные рисунки	Envelop C5	Xerox	Opera	Linux SUSE 10.0
8	2 страницы	Нет цветных рисунков	A5	Xerox	Mozilla Firefox	Linux RHEL 4.0
9	2 страницы	Нет цветных рисунков	Letter	Canon	Opera	Linux SUSE 10.0
10	2 страницы	Нет цветных рисунков	Envelop C5	HP	Mozilla Firefox	Windows XP
11	2 страницы	Есть цветные рисунки	A4	Xerox	Opera	Windows XP
12	2 страницы	Есть цветные рисунки	A5	HP	Opera	Linux SUSE 10.0
13	2 страницы	Есть цветные рисунки	B5	Xerox	Opera	Windows 2000

14	2 страницы	Есть цветные рисунки	Envelop C5	Epson	Internet Explorer	Windows Me
15	7 страниц	Нет цветных рисунков	A4	Canon	Internet Explorer	Windows 2000
16	7 страниц	Нет цветных рисунков	A5	Xerox	Opera	Windows Me
17	7 страниц	Нет цветных рисунков	B5	HP	Internet Explorer	Linux SUSE 10.0
18	7 страниц	Нет цветных рисунков	B5	Canon	Mozilla Firefox	Windows Me
19	7 страниц	Нет цветных рисунков	Letter	HP	Internet Explorer	Linux RHEL 4.0
20	7 страниц	Нет цветных рисунков	Letter	Xerox	Internet Explorer	Windows XP
21	7 страниц	Есть цветные рисунки	A4	Epson	Mozilla Firefox	Linux SUSE 10.0
22	7 страниц	Есть цветные рисунки	A5	Canon	Opera	Windows XP
23	7 страниц	Есть цветные рисунки	B5	Epson	Opera	Linux RHEL 4.0
24	7 страниц	Есть цветные рисунки	Letter	Epson	Opera	Windows Me
25	7 страниц	Есть цветные рисунки	Envelop C5	Epson	Internet Explorer	Windows 2000

Таблица 2. Минимальный тестовый набор для тестирования печати Web-страницы.

Эти примеры обобщаются до понятия *покрывающего набора* глубины t. Нам не важны конкретные значения факторов или параметров, важно только, что они образуют конечное множество. Поэтому можно считать, что если некоторый фактор имеет n возможных значений, ими являются числа от 0 до (n-1). Если заданы конечные наборы значений $\{v_{ij}\}$ для k параметров, i-й параметр может принимать n_i различных значений, то *покрывающим набором* глубины t <= k является любой набор из списков значений всех параметров $\{v_{if(j)}\}$, такой что любая комбинация возможных значений любых t параметров встречается в этом наборе хотя бы один раз.

Пары значений параметров покрываются наборами глубины 2, тройки — наборами глубины 3, и т.д.

	0	0	0	0	0	4
	0	0	0	0	2	0
	0	0	1	1	0	1
	0	0	4	2	0	4
	0	1	2	1	2	2
	0	1	3	0	1	1
	0	1	4	3	2	3
	1	0	1	3	1	4
	1	0	3	2	2	3
	1	0	4	0	1	2
	1	1	0	3	2	2
	1	1	1	0	2	3
	1	1	2	3	2	1
	1	1	4	1	0	0
	2	0	0	2	0	1
	2	0	1	3	2	0
	2	0	2	0	0	3
	2	0	2	2	1	0
	2	0	3	0	0	4
	2	0	3	3	0	2
	2	1	0	1	1	3
	2	1	1	2	2	2
	2	1	2	1	2	4
	2	1	3	1	2	0
	2	1	4	1	0	1

Таблица 3. Покрывающий набор, соответствующий показанному выше тестовому набору.

Множество всех покрывающих наборов глубины t с k параметрами, принимающими n_1, …, n_k значений обозначается CA(t, n_1, …, n_k). Минимальное возможное количество рядов в покрывающем наборе обозначается CAN(t, n_1, …, n_k). Таблица 3 представляет пример набора из CA(2, 3,·2,·5,·4,·3,·5) и показывает, что CAN(2, 3,·2,·5,·4,·3,·5) = 25.

Если все параметры могут принимать одно и то же число значений, т.е. $n_1 = n_2 = \ldots = n_k = n$, соответствующий покрывающий набор называется *однородным*. Множество однородных наборов CA(t, n, …, n) также обозначается CA(t; k, n), соответствующее минимальное число рядов в таком наборе — CAN(t; k, n).

Выгода от использования покрывающих наборов определяется тем фактом, что чаще всего существуют покрывающие наборы небольшой мощности, в которых количество рядов значительно меньше, чем число всех возможных комбинаций значений параметров.

Покрывающие наборы могут эффективно использоваться в ситуациях, в которых выполнены следующие условия.

- Есть некоторый вид воздействий на тестируемую систему, имеющий довольно много параметров или факторов, влияющих на его работу.
- Значения каждого из параметров можно разбить на (небольшое) конечное число классов, таких, что все существенные изменения в поведении системы происходят только из-за изменения класса одного из параметров. Иногда просто каждый параметр может принимать лишь значения из небольшого конечного множества.
- Ошибки в поведении системы возникают в основном за счет сочетания небольшого количества факторов, определяемых значениями используемых параметров.
- Дополнительной информации о зависимости между возможными ошибками и какими-либо другими условиями на значения параметров нет.

В частности, покрывающие наборы могут использоваться для определения комбинаций учитываемых факторов при тестировании на основе разбиения на категории, на основе дерева классификации. Также можно применять покрывающие наборы для снижения количества тестов при построении различных комбинаций альтернатив на основе грамматик.

Техники построения однородных покрывающих наборов

Наиболее хорошо развиты техники построения однородных покрывающих наборов. При количестве значений всех параметров равном 2 есть даже простой алгоритм построения минимального покрывающего набора глубины 2 (см. ниже).

Для однородных наборов глубины 2, в которых число значений параметров равно степени простого числа $n = p^k$ есть метод построения, основанный на арифметике конечных полей. Известно, что для каждой степени простого числа p^k есть конечное поле с таким количеством элементов, называемое полем Галуа $GF(p^k)$. Для $k = 1$, т.е. когда число элементов само является простым, GF(p) изоморфно полю вычетов по модулю p — $\mathbb{Z}_p$.

Рассмотрим таблицу из элементов поля $GF(p^k)$, построенную следующим способом.

Первый столбец состоит из n^2 значений, сгруппированных по n одинаковых значений. Каждую такую группу значений, равных i, будем называть i-м блоком. Первому столбцу присваивается номер ∞.

Второй столбец состоит из n^2 значений, выстроенных так, что в каждом блоке встречаются все возможные n значений. Значение, стоящее во втором столбце обозначим через j. Второму столбцу присвоим номер 0.

Все остальные столбцы, с третьего по (n+1)-й, с номерами m = 1...(n-1) построим так, чтобы в блоке i в j-м ряду стояло значение, получаемое как m*i+j в арифметике $GF(p^k)$.

Построенная так таблица будет покрывающим набором из множества CA(2; n+1, n), если рассматривать каждую ее строку как набор значений n+1 параметров, соответствующих столбцам.

Пример для n = 5.

Поскольку поле GF(5) изоморфно полю вычетов по модулю 5, складывать и умножать числа в обычной целочисленной арифметике, а в конце брать вместо результата его вычет по модулю 5. Получаемый таким способом покрывающий набор представлен ниже.

NN	∞	0	1	2	3	4
	0	0	0	0	0	0
	0	1	1	1	1	1
	0	2	2	2	2	2
	0	3	3	3	3	3
	0	4	4	4	4	4
	1	0	1	2	3	4

	1	1	2	3	4	0
	1	2	3	4	0	1
	1	3	4	0	1	2
	1	4	0	1	2	3
	2	0	2	4	1	3
	2	1	3	0	2	4
	2	2	4	1	3	0
	2	3	0	2	4	1
	2	4	1	3	0	2
	3	0	3	1	4	2
	3	1	4	2	0	3
	3	2	0	3	1	4
	3	3	1	4	2	0
	3	4	2	0	3	1
	4	0	4	3	2	1
	4	1	0	4	3	2
	4	2	1	0	4	3
	4	3	2	1	0	4
	4	4	3	2	1	0

Пример для n = 4.

В GF(4) сложение и умножение устроены иначе, чем по модулю 4. Поэтому сначала приведем таблицы сложения и умножения в поле с 4-мя элементами.

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Получаемый для n = 4 по описанной конструкции покрывающий набор представлен ниже.

NN	∞	0	1	2	3
	0	0	0	0	0
	0	1	1	1	1
	0	2	2	2	2
	0	3	3	3	3
	1	0	1	2	3
	1	1	0	3	2
	1	2	3	0	1
	1	3	2	1	0
	2	0	2	3	1
	2	1	3	2	0
	2	2	0	1	3
	2	3	1	0	2
	3	0	3	1	2
	3	1	2	0	3
	3	2	1	3	0
	3	3	0	2	1

Таким образом можно строить однородные наборы глубины 2 для (n+1) параметра с n значениями при $n = p^k$. Этим показывается, что $CAN(2; p^k+1, p^k) = p^{2k}$.

Похожая конструкция существует для покрывающих наборов глубины $t > 2$ и $n = p^k > t$. Для этого надо взять таблицу из n+1-го столбца и n^t строк. Каждую строку ее можно сопоставить набору a0, a1, …, at-1 элементов из поля $GF(p^k)$. Столбцы так же обозначаются ∞, 0, 1, ..., n-1. Элемент в определенной строке и определенном столбце вычисляется по следующим правилам.

	∞	0	x
$a_0a_1...a_{t-1}$	a_0	a_{t-1}	$\Sigma a_i x^i$

Здесь снова используются сложение и умножение из поля $GF(p^k)$.

Для глубины 3 и $n = 2^k > t$ можно расширить эту таблицу на еще один столбец — два первых столбца обозначим ∞_1 и ∞_2, остальные, как раньше, — 0, 1, ..., n-1.

	∞_0	∞_1	0	x
$a_0a_1a_2$	a_0	a_1	a_2	$\Sigma a_i x^i$

Таким образом, оказывается, что $CAN(t; p^k+1, p^k) = p^{tk}$ при $t < p^k$, а также $CAN(3; 2^k+2, 2^k) = 2^{3k}$ при $k > 1$.

Пример для n = 3, t = 3.

NN	∞	0	1	2
000	0	0	0	0
001	0	1	1	1
002	0	2	2	2
010	0	0	1	2
011	0	1	2	0
012	0	2	0	1
020	0	0	2	1
021	0	1	0	2
022	0	2	1	0
100	1	0	1	1
101	1	1	2	2
102	1	2	0	0
110	1	0	2	0
111	1	1	0	1
112	1	2	1	2
120	1	0	0	2
121	1	1	1	0
122	1	2	2	1
200	2	0	2	2
201	2	1	0	0
202	2	2	1	1
210	2	0	0	1
211	2	1	1	2
212	2	2	2	0
220	2	0	1	0
221	2	1	2	1
222	2	2	0	2

Представленные выше конструкции позволяют строить однородные покрывающие наборы для небольшого числа параметров (<= n+1), принимающих n значений для n являющегося степенью простого числа.

Посмотрим теперь, как можно строить покрывающие наборы для числа значений, не являющегося степенью простого числа. Оказывается, есть общая конструкция покрывающего набора для числа значений, являющегося произведением чисел значений в уже построенных покрывающих наборов: покрывающие наборы с k параметрами глубины t для n_1 и n_2 значений дают покрывающий набор с k параметрами для глубины t для $n_1 \cdot n_2$.

Для его построения обозначим элементы двух исходных наборов через a_{ij} и b_{lj} — у этих наборов одинаковое число столбцов, и, возможно, разное число строк. В первом наборе участвуют элементы от 0 до $(n_1–1)$, во втором — от 0 до $(n_2–1)$. Любое число от 0 до $(n_1 \cdot n_2–1)$ можно однозначно представить в виде $n_2 \cdot q + r$, где q лежит от 0 до $(n_1–1)$, r — от 0 до $(n_2–1)$. Кроме того, обозначим строки новой таблицы парами индексов строк двух исходных таблиц. Тогда ее элементы могут быть построены по формуле $x_{(i,m)j} = n_2 \cdot a_{ij} + b_{mj}$. Подученная так таблица представляет покрывающий набор с k параметрами для глубины t для $n_1 \cdot n_2$.

Как следствие $CAN(t; k, n_1 \cdot n_2) <= CAN(t; k, n_1) \cdot CAN(t; k, n_2)$.

Пример для $n = 6 = 2 \cdot 3$.

		0	1
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

		0	1	2
0	0	0	0	0
1	0	1	1	1
2	0	2	2	2
3	1	0	1	2
4	1	1	2	0
5	1	2	0	1
6	2	0	2	1
7	2	1	0	2
8	2	2	1	0

		0	1
00	0	0	0
01	0	1	1
02	0	2	2
03	1	0	1
04	1	1	2
05	1	2	0
06	2	0	2
07	2	1	0
08	2	2	1
10	0	3	3
11	0	4	4
12	0	5	5
13	1	3	4
14	1	4	5
15	1	5	3
16	2	3	5
17	2	4	3
18	2	5	4
20	3	0	3
21	3	1	4
22	3	2	5
23	4	0	4
24	4	1	5
25	4	2	3
26	5	0	5
27	5	1	3
28	5	2	4
30	3	3	0
31	3	4	1
32	3	5	2
33	4	3	1
34	4	4	2
35	4	5	0
36	5	3	2
37	5	4	0
38	5	2	4

Описанные выше техники позволяют строить покрывающие наборы любой глубины с любыми значениями для небольшого количества параметров. Чтобы увеличить количество параметров, нужно использовать другие подходы.

Во-первых, для глубины 2 и произвольного числа параметров k, принимающих только 2 значения, есть алгоритм, позволяющий достаточно быстро находить минимальный возможный покрывающий набор.

Выберем наименьшее N такое, что выполнено $k \le C_{N-1}^{\lceil N/2 \rceil}$. Здесь $\lceil x \rceil$ — наименьшее целое число, большее или равное x, C_r^q — биномиальный коэффициент. Получаемые значения N для разных k сведены в следующую таблицу.

k	N		k	N		k	N
2-3	4		1717-3003	15		2496145-5200300	26
4	5		3004-6435	16		5200301-9657700	27
5-10	6		6436-11440	17		9657701-20058300	28
11-15	7		11441-24310	18		20058301-37442160	29
16-35	8		24311-43758	19		37442161-77558760	30
36-56	9		43759-92378	20		77558761-145422675	31
57-126	10		92377-167960	21		145422676-300540195	32
127-210	11		167961-352716	22		300540196-565722720	33
211-462	12		352717-646646	23		565722721-1166803110	34

463-792	13		646647-1352078	24		1166803111-2203961430	35
793-1716	14		1352079-2496144	25		2203961431-4537567650	36

Это число N равно числу строк в покрывающем наборе глубины 2 с двумя значениями для k параметров. Из таблицы видно, что небольшое число тестов может покрыть все комбинации пар значений для огромного количества параметров — 10 тестов достаточно для 126 параметров, а 20 тестов — для более чем 92000.

Первую строка набора сделаем состоящей целиком из 0. Остается N–1 строк, элементы которых строятся по столбцам. В качестве этих столбцов берутся все возможные последовательности из $\lceil N/2 \rceil$ единиц и $\lfloor N/2 \rfloor$-1 нулей.

Примеры.

CAN(2; 4, 2) = 5

0000
1110
1101
1011
0111

CAN(2; 10, 2) = 6

00000 00000
11111 10000
11100 01110
10011 01101
01010 11011
00101 10111

CAN(2; 15, 2) = 7

00000 00000 00000
11111 11111 00000
11111 10000 11110
11100 01110 11101
10011 01101 11011
01010 11011 10111
00101 10111 01111

CAN(2; 35, 2) = 8

00000 00000 00000 00000 00000 00000 00000
11111 11111 11111 11111 00000 00000 00000
11111 11111 00000 00000 11111 11111 00000
11110 00000 11111 10000 11111 10000 11110
10001 11000 11100 01110 11100 01110 11101
01001 00110 10011 01101 10011 01101 11011
00100 10101 01010 11011 01010 11011 10111
00010 01011 00101 10111 00101 10111 01111

CAN(2; 56, 2) = 9

00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 0
11111 11111 11111 11111 11111 11111 11111 00000 00000 00000 00000 0
11111 11111 11111 11111 00000 00000 00000 11111 11111 11111 00000 0
11111 11111 00000 00000 11111 11111 00000 11111 11111 00000 11111 0
11110 00000 11111 10000 11111 10000 11110 11111 10000 11110 11110 1
10001 11000 11100 01110 11100 01110 11101 11100 01110 11101 11101 1
01001 00110 10011 01101 10011 01101 11011 10011 01101 11011 11011 1
00100 10101 01010 11011 01010 11011 10111 01010 11011 10111 10111 1
00010 01011 00101 10111 00101 10111 01111 00101 10111 01111 01111 1

Доказать, что получаемый так набор действительно покрывающий достаточно просто, а вот для доказательства того, что он минимальный нужны нетривиальные комбинаторные факты, а именно — теорема Ердеша-Ко-Радо.

Для всех остальных значений параметров хороших алгоритмов построения минимальных покрывающих наборов неизвестно, более того, показано, что построение минимальных покрывающих наборов CA(2, k, n), CA(t, k, 2) — NP-полные задачи.

Для построения однородных покрывающих наборов для числа значений, не равного 2, и для большого количества параметров проще всего использовать рекурсивные конструкции, с помощью которых набор для большого количества параметров строится из наборов для меньшего количества. Ниже рассматриваются две такие конструкции — для глубины 2 и для глубины 3.

Рекурсивная конструкция для покрывающих наборов глубины 2 и $n = p^k$.

Строим набор для nm+1 параметра из набора для m параметров.

Обозначим через A_{ij} (i <= N, j <= m) элементы исходного набора из CA(2; m, n).

Обозначим также через B_{ij} элементы из нижней части (без n верхних строк) покрывающего набора, построенного с помощью самой первой конструкции, число строк в наборе B равно $z = n^2–n$.

Строим новый набор в соответствии с приведенной ниже схемой — в ней первая строка, а также первый столбец отмечают группировку элементов.

		n				n					n			
N	0	A_{11}	A_{11}	…	A_{11}	A_{12}	A_{12}	…	A_{12}	…	A_{1m}	A_{1m}	…	A_{1m}
	0	A_{21}	A_{21}		A_{21}	A_{22}	A_{22}		A_{22}		A_{2m}	A_{2m}		A_{2m}
	…	…	…		…	…	…		…		…	…		…
	0	A_{N1}	A_{N1}		A_{N1}	A_{N2}	A_{N2}		A_{N2}		A_{Nm}	A_{Nm}		A_{Nm}
	B_{11}	B_{12}	B_{13}	…	$B_{1(n+1)}$	B_{12}	B_{13}	…	$B_{1(n+1)}$	…	B_{12}	B_{13}	…	$B_{1(n+1)}$
	…	…	…		…	…	…		…		…	…		…
	B_{z1}	B_{z2}	B_{z3}		$B_{z(n+1)}$	B_{z2}	B_{z3}		$B_{z(n+1)}$		B_{z2}	B_{z3}		$B_{z(n+1)}$

Получаем соотношение $CAN(2; mp^k+1, p^k) <= CAN(2; m, p^k) + p^{2k} – p^k$.

Пример для n = 3, m = 4 Исходный набор A, набор B получается из него отбрасыванием 3-х верхних строк.

		0	1	2
0	0	0	0	0
1	0	1	1	1
2	0	2	2	2
3	1	0	1	2
4	1	1	2	0
5	1	2	0	1
6	2	0	2	1
7	2	1	0	2
8	2	2	1	0

1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

Получаемый набор размера 15 для 13 параметров выглядит так.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	1	1	1
2	0	0	0	0	2	2	2	2	2	2	2	2	2
3	0	1	1	1	0	0	0	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	0	0	0
5	0	1	1	1	2	2	2	0	0	0	1	1	1
6	0	2	2	2	0	0	0	2	2	2	1	1	1
7	0	2	2	2	1	1	1	0	0	0	2	2	2
8	0	2	2	2	2	2	2	1	1	1	0	0	0
9	1	0	1	2	0	1	2	0	1	2	0	1	2
10	1	1	2	0	1	2	0	1	2	0	1	2	0
11	1	2	0	1	2	0	1	2	0	1	2	0	1
12	2	0	2	1	0	2	1	0	2	1	0	2	1
13	2	1	0	2	1	0	2	1	0	2	1	0	2
14	2	2	1	0	2	1	0	2	1	0	2	1	0

Рекурсивная конструкция для покрывающих наборов глубины 3.

Строим набор глубины 3 для числа элементов n и 2k параметров из набора глубины 3 для того же числа элементов и k параметров и набора глубины 2 для того же числа элементов и k параметров.

Обозначим элементы исходного набора глубины 3 через A_{ij} (i <= N, j <= k), элементы исходного набора глубины 2 — через B_{ij} (i <= z, j <= k).

Строим новый набор в соответствии с приведенной ниже схемой — в ней первый столбец отмечает группировку элементов, а все сложения проводятся по модулю n.

N	A_{11}	A_{11}	A_{12}	A_{12}	A_{13}	A_{13}		A_{1k}	A_{1k}
	…	…	…	…	…	…		…	…
	A_{N1}	A_{N1}	A_{N2}	A_{N2}	A_{N3}	A_{N3}		A_{Nk}	A_{Nk}
1	B_{11}	$(B_{11}+1)$	B_{12}	$(B_{12}+1)$	B_{13}	$(B_{13}+1)$		B_{1k}	$(B_{1k}+1)$
	…	…	…	…	…	…		…	…
	B_{z1}	$(B_{z1}+1)$	B_{z2}	$(B_{z2}+1)$	B_{z3}	$(B_{z3}+1)$		B_{zk}	$(B_{zk}+1)$
2	B_{11}	$(B_{11}+2)$	B_{12}	$(B_{12}+2)$	B_{13}	$(B_{13}+2)$		B_{1k}	$(B_{1k}+2)$
	…	…	…	…	…	…		…	…
	B_{z1}	$(B_{z1}+2)$	B_{z2}	$(B_{z2}+2)$	B_{z3}	$(B_{z3}+2)$		B_{zk}	$(B_{zk}+2)$
…	…	…	…	…	…	…		…	…
n-1	B_{11}	$(B_{11}+(n-1))$	B_{12}	$(B_{12}+(n-1))$	B_{13}	$(B_{13}+(n-1))$		B_{1k}	$(B_{1k}+(n-1))$
	…	…	…	…	…	…		…	…
	B_{z1}	$(B_{z1}+(n-1))$	B_{z2}	$(B_{z2}+(n-1))$	B_{z3}	$(B_{z3}+(n-1))$		B_{zk}	$(B_{zk}+(n-1))$

Получаем соотношение CAN(3; 2k, n) <= CAN(3; k, n) + (n-1)CAN(2; k, n).

Для представленных выше в различных местах наборов с n = 3 и k = 4 можно получить набор из CA(3; 8, 3), имеющий 45 строк.

Приведенные выше конструкции позволяют строить наборы, число тестов в которых примерно пропорционально логарифму числа параметров. Доказаны следующие соотношения.

При $n\to\infty$ CAN(2, k, n) ~ (n/2)log(k)

CAN(t, k, 2) <~ $t2^t$log(k)O(log(t))

CAN(t, k, n) <~ (t-1)log(k)/log(n^t/(n^t-1)), что при $n^t\to\infty$ можно ограничить выражением $(1+\varepsilon)tn^t$log(k) для любой положительной константы ε.

Построение неоднородных покрывающих наборов

Неоднородные покрывающие наборы часто можно достаточно быстро построить из близких по конфигурации однородных.

Например, самый первый приведенный выше набор из CA(2, 3,·2,·5,·4,·3,·5) был построен так. Переставляя параметры его можно свести к CA(2, 5, 5, 4, 3, 3, 2). Уже видно, что потребуется как минимум 25 строк. Есть однородный набор CA(2, 5, 5, 5, 5, 5, 5), состоящий в точности из 25 строк — берем его и приводим все параметры с меньшим числом значений по модулю этого числа значений.

Описанный прием может не дать покрывающий набор, если есть параметры с числом значений, не взаимно простым с числом значений в исходном однородном наборе. В таких случаях часто все равно можно построить соответствующий неоднородный набор, только для «проблемных» параметров придется отдельно подбирать расположение их значений.

Еще один пример — покрывающий набор из CA(2, 10, 9, 8, 7, 6, 5, 4, 3, 2). Для него требуется не меньше 90 строк, однако близких однородных наборов с 10 значениями нет. Первые два столбца построим просто как все возможные сочетания 10 и 9 элементов. А дальше можно использовать столбцы, начиная с третьего, из однородного набора для 9

элементов из CA(2; 10, 9). Для параметров, число значений которых равно 6 и 3, придется дополнительно подбирать расстановку, но для этого достаточно несколько раз сместить вдоль столбца соответствующие значения из однородного набора, взятые по модулю 6 и 3. В итоге получается искомый покрывающий набор из 90 элементов, а значит — минимальный возможный.

В общем случае для построения покрывающих наборов любой конфигурации, однородных и неоднородных, можно использовать эвристические алгоритмы. Одна из возможных эвристик при построении набора из CA(t, n_1, …, n_k) — построить сначала все возможные комбинации из t значений параметров, а затем последовательно пытаться добавить новые столбцы, добавляя строки только при необходимости, пока не получим искомый набор. Такой алгоритм называется *жадным* и выглядит примерно так.

Упорядочим n_1, …, n_k по убыванию (IPO) и построим исходную таблицу из всех возможных комбинаций значений первых t параметров.

1. Если неиспользованных параметров нет — выдаем построенный набор в качестве результата.
   Если есть еще неиспользованные параметры, берем первый из них. Строим столбец его значений произвольным образом.
2. Вычисляем покрываемые комбинации из значений t уже имеющихся параметров, в которых последним параметром является только что добавленный. Если можно расширить это множество за счет замены значений в только что построенном столбце, делаем это.
   Если покрыты все возможные комбинации значений t параметров с использованием добавленного, идем в пункт 1.
   Иначе идем в пункт 3.
3. Построим новую строку с использованием любой из оставшихся непокрытыми комбинаций значений t параметров с использованием добавленного псоледним. Если в ней есть неопределенные значения параметров, определяем их так, чтобы при этом покрылось как можно больше других непокрытых комбинаций.
   Добавляем построенную строку к таблице и выбрасываем из множества непокрытых комбинаций все, покрываемые ею.
   Если больше нет непокрытых комбинаций, идем в пункт 1.
   Иначе возвращаемся в начало пункта 3.

Этот и другие эвристические алгоритмы построения покрывающих наборов реализованы в множестве инструментов, как коммерческих, так и доступных свободно, таких, как AETG, TestCover.com, CaseMaker, Pro-test, CATS, IBM Intelligent Test Case Handler, TConfig, TCG Jenny (см. [1]).

Комбинаторные методы построения тестовых последовательностей

Рассмотренные выше методы были, в основном, ориентированы на построение тестовых данных. Их можно использовать и для разработки более сложных тестов, вводя элементы состояния как дополнительные факторы. Однако при этом надо отдельно заботиться о преобразовании полученных комбинаторных схем в исполнимые тесты. То есть, если построенный тест определяет, что некоторый элемент состояния должен принадлежать классу X, разработчик тестов должен сам придумать способ установить этот элемент в нужное значение.

Существуют и более прямые комбинаторные методы построения тестовых последовательностей, позволяющих покрывать различные состояния тестируемой системы.

Наиболее простой такой подход основан на последовательностях де Бройна.

Предположим, что в тестируемой системе n операций без параметров, каждая из которых может изменять ее состояние. Каким образом можно построить одну последовательность их

вызовов так, чтобы при этом проверить как можно больше различных вариантов поведения системы?

Один из возможных ответов — использовать *последовательность де Бройна из n значений глубины k.* Это максимально короткая последовательность, которая содержит все возможные последовательности длины k из n значений в качестве своих подпоследовательностей. Известно, что для любых n и k такие последовательности существуют и имеют длину $n^{(k-1)}+(k-1)$. При этом все их подпоследовательности длины k различны, то есть все возможные последовательности длины k упакованы в такую последовательность максимально плотным образом.

Примеры последовательностей де Бройна.

n = 2, k = 2 — 00110

n = 2, k = 3 — 0001011100

n = 2, k = 4 — 0000100110101111000

n = 3, k = 2 — 0010211220

n = 3, k = 3 — 000101110020121120222102122 00

n = 4, k = 2 — 00102031121322330

Для построения последовательностей де Бройна используют графы де Бройна. *Граф де Бройна* с параметрами n≥1 и k≥1 B(m, k) — это ориентированный граф с n^{k-1} вершинами V(n, k) = $[0..(n-1)]^{k-1}$, являющимися всеми возможными последовательностями из n значений длины (k–1), и ребрами E(n, k) = $[0..(n-1)]^{k}$, являющимися всеми возможными последовательностями из n значений длины k. При этом ребро $x_1x_2…x_{k-1}x_k$ начинается в вершине $x_1x_2…x_{k-1}$ и заканчивается в вершине $x_2…x_{k-1}x_k$.

Примеры графов де Бройна изображены на рисунке ниже.

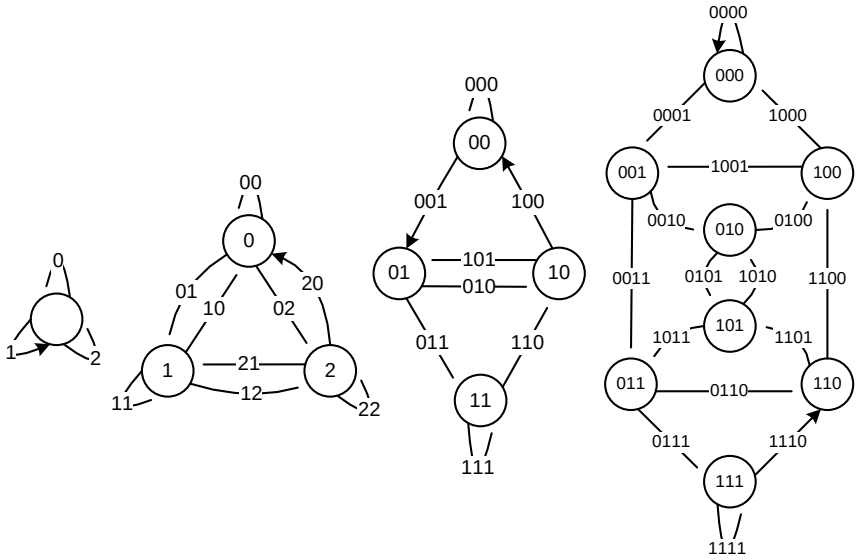


Рисунок 4. Графы B(3, 1), B(3, 2), B(2, 3), B(2, 4).

Все графы де Бройна — эйлеровы, то есть в них существует путь, содержащий все ребра по одному разу, с совпадающими началом и концом. Если выписать последовательность, являющуюся первым ребром такого пути, а для каждого следующего ребра добавлять к полученной последовательности последний символ (заметим, что начало длины k–1 этого ребра будет концом получаемой перед ним последовательсноти), получится как раз последовательность де Бройна.

Каждая последовательность де Бройна глубины k для n значений полностью покрывает все возможные ситуации в тестируемой системе с n операциями, если предположить, что ее поведение полностью определяется последними k вызванными операциями.

Литература

[1] http://www.pairwise.org/tools.asp

Тестирование на основе моделей

В. В. Кулямин

Лекция 6. Автоматные методы построения тестов. Основные понятия

Автоматные методы построения тестов основаны на предположении, что реальное поведение тестируемой системы может быть полностью описано некоторым автоматом. В виде автомата описывается также требуемое ее поведение, после чего ставится ряд экспериментов, состоящих в выполнении определенных последовательностей воздействий, с целью выяснить, отличается ли реальное поведение от требуемого.

В рамках автоматных методов предлагаются различные способы построения таких наборов тестовых последовательностей, что корректная работа тестируемой системы на них позволяет при некоторых ограничениях уверенно утверждать, что она корректно работает всегда. Основная проблема здесь — построить достаточно небольшой такой набор.

В целом автоматные методы хорошо автоматизируются, обеспечивают хорошую полноту тестирования и позволяют находить весьма сложные ошибки. Однако, они требуют наличия точного и полного описания требований к поведению тестируемой системы, определенных навыков работы с автоматными моделями ПО от разработчиков тестов и некоторых затрат на выполнение тестов. При росте сложности используемых моделей затраты на разработку тестов при помощи таких методов возрастают нелинейно.

Самой простой разновидностью автоматов являются конечные автоматы. Большинство автоматных методов построения тестов основано именно на них. На всякий случай повторим определение конечного автомата из Лекции 4.

Конечный автомат — это набор (S, s_0, I, O, T), где

S — конечное множество, элементы которого называются *состояниями* автомата;

s_0 — элемент S, называемый *начальным состоянием*;

I — конечное множество, элементы которого называются *входными символами*, *входами* или *стимулами*, само I называют *входным алфавитом* автомата;

O — конечное множество, элементы которого называются *выходными символами*, *выходами* или *реакциями*, само O называют *выходным алфавитом* автомата;

$T \subseteq S \times I \times O \times S$ — множество *переходов* автомата. Каждый переход — четверка (s_1, i, o, s_2) — имеет *начальное состояние* s_1, *конечное состояние* s_2, *стимул* i и *реакцию* o. Говорят, что он *выходит из* s_1 и *ведет в* s_2, помечен стимулом i и реакцией o. Этот переход изображают стрелкой, ведущей из s_1 и в s_2 и помеченной i/o.

Автомат может выполнять некоторую последовательность стимулов следующим образом. Сначала он считается находящимся в начальном состоянии. При получении очередного стимула в некотором состоянии недетерминированным образом выбирается один из переходов, выходящих из текущего состояния и помеченных принятым стимулом. Следующим состоянием автомата становится конечное состояние выбранного перехода, а вовне выдается реакция, которой помечен выбранный переход. Выполнение автомата не определено, если в текущем состоянии нет переходов, помеченных получаемым стимулом.

Автомат называется *полностью определенным*, если в каждом его состоянии для каждого стимула есть выходящий из этого состояния переход, помеченный этим стимулом.

Автомат *детерминирован*, если в каждом его состоянии для каждого стимула есть не более одного выходящего из этого состояния перехода, помеченного этим стимулом. Автомат *наблюдаемо детерминирован*, если в каждом его состоянии для каждого стимула и каждой реакции есть не более одного выходящего из этого состояния перехода, помеченного этим стимулом и этой реакцией.

В детерминированном автомате текущее состояние и принятый стимул однозначно определяют новое состояние автомата и выдаваемую реакцию. В наблюдаемо детерминированном новое состояние можно однозначно определить, зная предыдущее, принятый стимул и выданную реакцию.

Для детерминированного автомата (S, s_0, I, O, T) пусть δ и λ обозначают функции переходов и вывода, т.е. $(s_1, i, o, s_2) \in T$ тогда и только тогда, когда $\delta(s_1, i) = s_2$, и $\lambda(s_1, i) = o$. Оба этих отображения можно расширить до отображений состояния и принятой последовательности стимулов в последовательность реакций и итоговое состояние. Именно, для состояния автомата $s \in S$ и последовательности входных символов $\alpha \in I^*$ определим $\delta(s, \alpha) \in S$ и $\lambda(s, \alpha) \in O^*$ рекурсивно следующим образом.

Если α — пустая последовательность ε, то $\delta(s, \alpha) = s$ и $\lambda(s, \alpha) = \varepsilon$.

Если α — непустая и $\alpha = \alpha'a$, где $\alpha' \in I^*$ и $a \in I$, то $\delta(s, \alpha) = \delta(\delta(s, \alpha'), a)$ и $\lambda(s, \alpha) = \lambda(s, \alpha')\lambda(\delta(s, \alpha'), a)$.

Конечный автомат реализует некоторое соответствие последовательностей символов из I и последовательностей символов из O, $\varphi \subseteq I^* \times O^*$. Это соответствие называется его *поведением*. Для детерминированных автоматов $\varphi = \lambda(s_0, \cdot)$, то есть поведение является отображением входных последовательностей в выходные для начального состояния.

С точки зрения внешнего наблюдателя, не имеющего возможности «увидеть» текущее состояние автомата, два автомата с одинаковым поведением неотличимы. Точно также для него неотличимы состояния, в которых на одну и те же последовательности входных символов всегда может быть выдана одна и та же последовательность реакций. Такие состояния (даже в разных автоматах) называются *эквивалентными* или *неотличимыми*. Автоматы называются *эквивалентными*, если эквивалентны их начальные состояния.

Для любого автомата можно построить эквивалентный ему наблюдаемо детерминированный автомат. Этот факт доказывается примерно так же, как возможность использовать детерминированный автомат в качестве распознавателя регулярного языка.

Понятно, что внешнему наблюдателю отличить друг от друга эквивалентные автоматы невозможно. Чаще всего, это и не нужно, поскольку автомат обычно предназначен как раз для того, чтобы реализовывать некоторое соответствие между входными и выходными последовательностями, и любой автомат, делающий это правильно, считается подходящим. Поэтому с точки зрения тестирования автоматов важен только их класс эквивалентности.

В каждом классе эквивалентности автоматов есть единственный (с точностью до изоморфизма, т.е. взаимно-однозначного отображения состояний, сохраняющего начальное состояние и сопоставляющего эквивалентные) автомат с минимальным числом состояний. Такой автомат называют *минимальным* или *приведенным*. При тестировании обычно пытаются проверить соответствие как раз минимальному автомату, описывающему требуемое поведение тестируемой системы.

Специальные типы входных последовательностей автоматов

Чтобы уметь проверять соответствие поведение неизвестного автомата заданному, нужно уметь различать автоматы, у которых различия в поведении есть. Для этого используется несколько техник, основанных на специфических входных последовательностях.

Далее все определения даются для детерминированных автоматов. Для некоторых из этих понятий есть аналоги для автоматов общего вида, но их определения более сложны.

Идентификация состояний

Различающая последовательность (distinguishing sequence) конечного автомата — это такая конечная последовательность стимулов $d \in I^*$, что соответствующие ей

последовательности реакций в разных состояниях автомата разные. То есть, $\forall s_1, s_2 \in S\ s_1 \neq s_2 \Rightarrow \lambda(s_1, d) \neq \lambda(s_2, d)$.

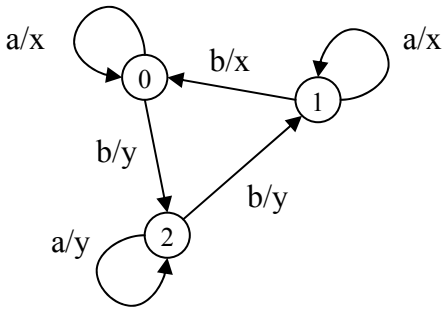


Рисунок 1. Пример конечного автомата, имеющего различающую последовательность.

Например, в автомате, изображенном на рисунке выше различающей последовательностью является последовательность ab, поскольку выполнено $\lambda(0, ab) = xy$, $\lambda(1, ab) = xx$, $\lambda(2, ab) = yy$.

Адаптивной последовательностью в алфавитах I и O называется дерево с корнем, чьи вершины помечены элементами I, а ребра помечены элементами O. Длиной адаптивной последовательности называется максимальное число вершин в цепочках ее вершин от корня до листовой.

К конечному автомату в определенном состоянии можно применять не только обычные последовательности, но и адаптивные. Результатом такого применения можно считать отображение цепочки вершин адаптивной последовательности, начинающейся от корня и продолжающейся вдоль его ребер, в выходной алфавит автомата. На это отображение накладывается следующие два ограничения.

- Если оно определено для некоторой вершины и его значение совпадает с меткой одного из ведущих из нее ребер, оно должно быть определено и для конца этого ребра.
- Если оно определено для одного из потомков некоторой вершины, оно должны быть определено и для самой вершины и значение его на этой вершине должно совпадать с пометкой ребра, ведущего к указанному потомку.

Интуитивно применение адаптивной последовательности можно описать так: мы применяем к автомату последовательно, начиная от корня, стимул, стоящий в очередной вершине, смотрим на полученную реакцию, и если она совпадает с пометкой одного из ребер, ведущих из текущей вершины, идем по нему и применяем стимул, стоящий в его конце, и т.д.

Обычная различающая последовательность называется также статической различающей последовательностью. *Адаптивной различающей последовательностью* автомата называется такая адаптивная последовательность в его алфавитах стимулов и реакций, что результаты ее применения во всех состояниях различны.

Например, для представленного выше автомата в качестве адаптивной различающей последовательности можно взять

$$a \xrightarrow{x} b$$

Длина этой адаптивной последовательности такая же, как у статической, но она позволяет быстрее определить состояние 2 — для этого достаточно получить y в ответ на a.

Различающие последовательности, как статические, так и адаптивные, позволяют однозначно определять состояние автомата, в котором они были применены. Однако не у всякого автомата они есть. Ясно, что из существования статической различающей последовательности следует существование адаптивной. Более того, есть примеры

автоматов, у которых нет статической различающей последовательности, но есть адаптивная. Ниже показан пример автомата, у которого нет даже адаптивной различающей последовательности.

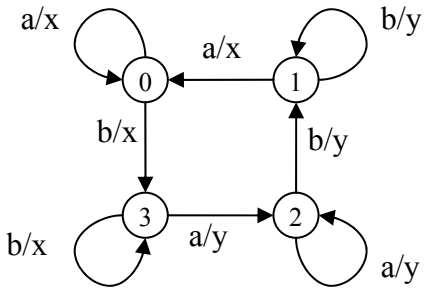


Рисунок 2. Пример автомата без различающей последовательности.

Чтобы убедиться в этом, достаточно проанализировать результаты применения a и b (по отдельности) к этому автомату. Различающая последовательность не может начинаться с a, поскольку после этого стимула перестают быть различимы пары состояний 0 и 1, 2 и 3 — после принятия a в любом из состояний пары автомат оказывается в одном и том же состоянии, выдав при этом одну и ту же реакцию. Точно так же b «слепляет» пары состояний 0 и 3, 1 и 2, поэтому различающая последовательность не может начинаться с b.

Чтобы различать состояния при отсутствии различающей последовательности, необходимы другие техники.

Последовательностью однозначных входов/выходов или *UIO-последовательностью* (unique input-output sequence) для данного состояния s называется такая последовательность стимулов u, что результат ее применения в состоянии s отличается от результатов ее применения во всех других состояниях. То есть, $\forall s_1 \in S\ s_1 \neq s \Rightarrow \lambda(s, u) \neq \lambda(s_1, u)$.

Различающая последовательность является UIO-последовательностью сразу для всех состояний автомата. В автомате, изображенном на Рис. 1, UIO-последовательностью для состояния 1 является также b, а для состояния 2 — a.

Есть автоматы, в которых нет различающих последовательностей, ни статической, ни адаптивной, но для каждого состояния есть UIO-последовательность. К сожалению, бывают и автоматы, ни одно состояние которых не имеет UIO-последовательности. Пример изображен на Рис. 2.

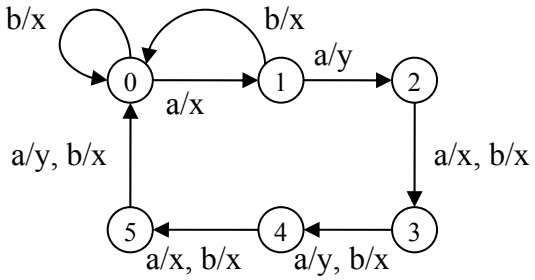
В таких случаях можно использовать *характеризующее* или *диагностическое множество последовательностей* (characterizing set) W автомата. Это такое множество входных последовательностей, что для каждых двух разных состояний автомата в нем найдется последовательность, результаты применения которой в этих состояниях различны. То есть, $\forall s_1, s_2 \in S\ s_1 \neq s_2 \Rightarrow \exists \alpha \in W\ \lambda(s_1, \alpha) \neq \lambda(s_2, \alpha)$.

Диагностическое множество всегда существует для минимального автомата, поскольку в нем поведение в разных состояниях отличается. Для автомата, имеющего различающую последовательность d, можно в качестве W взять {d}. Для автомата, в котором все состояния имеют UIO-последовательности любое множество, содержащее по UIO-последовательности для каждого состояния, является диагностическим. Для автомата, изображенного на Рис. 2, диагностическим является множество {a, b} — легко проверяется, что возможные реакции на это множество во всех его состояниях различны.

Для вычисления статической и адаптивной различающих последовательностей, UIO-последовательностей и диагностического множества можно использовать следующий алгоритм.

Упорядочим каким-либо образом множество I. Перебираем входные последовательности в лексикографическом порядке. Для каждой из них определяем возможные выходные

последовательности при ее применении в разных состояниях, разбиение множества состояний на группы состояний, для которых выходные последовательности одинаковы, а также покрытие множества состояний группами состояний, в которых можно оказаться после получения определенной выходной последовательности. Делать это можно итеративно, основываясь на результатах таких же вычислений для префиксов данной последовательности.



Рассмотрим автомат, изображенный на рисунке выше. Попробуем применить к нему этот алгоритм.

	a	b	aa	ab	ba	bb
0	x/1	x/0	xy/2	xx/0	xx/1	xx/0
1	y/2	x/0	yx/3	yx/3	xx/1	xx/0
2	x/3	x/3	xy/4	xx/4	xy/4	xx/4
3	y/4	x/4	yx/5	yx/5	xx/5	xx/5
4	x/5	x/5	xy/0	xx/0	xy/0	xx/0
5	y/0	x/0	yx/1	yx/0	xx/1	xx/0
	{0,2,4}x {1,3,5}y		{0,2,4}xy {1,3,5}yx	{0,2,4}xx {1,3,5}yx	{0,1,3,5}xx {2,4}xy	
	x{1,3,5} y{0,2,4}		yx{1,3,5} xy{0,2,4}	xx{0,4} yx{0,3,5}	xx{1,5} xy{0,4}	

После построения разбиений для всех последовательностей длины 2 видно, что многократное применение a не будет давать новой информации, просто сдвигая текущее состояние по циклу. Если же использовать b много раз, все состояния просто «слипнутся». Поэтому дальше имеет смысл пробовать только какие-то сочетания a и b.

	aab	aba	abb	baa	bab	aaba
0	xyx/3	xxx/1	xxx/0	xxy/2	xxx/0	xyxy/4
1	yxx/4	yxy/4	yxx/4	xxy/2	xxx/0	yxxx/5
2	xyx/5	xxx/5	xxx/5	xyx/5	xyx/5	xyxy/0
3	yxx/0	yxy/0	yxx/0	xxy/0	xxx/0	yxxx/1
4	xyx/0	xxx/1	xxx/0	xyx/1	xyx/0	xyxx/1
5	yxx/0	yxx/1	xxx/0	xxy/2	xxx/0	yxxx/1
	{0,2,4}xyx {1,3,5}yxx	{0,2,4}xxx {1,3}yxy {5}yxx	{0,2,4,5}xxx {1,3}yxx	{0,1,3,5}xxy {2,4}xyx	{0,1,3,5}xxx {2,4}xyx	{0,2}xyxy {1,3,5}yxxx {4}xyxx
	xyx{0,3,5} yxx{0,4}	xxx{1,5} yxy{0,4} yxx{1}	xxx{0,5} yxx{0,4}	xxy{0,2} xyx{1,5}	xxx{0} xyx{0,5}	xyxy{0,4} yxxx{1,5} xyxx{1}

Теперь мы обнаружили UIO-последовательности у двух состояний — aba для 5 и aaba для 4. Кроме того, уже видно, что baba является UIO-последовательностью для 2, а добавив ba к последовательностям aba и aaba, мы сможем разделить группу состояний {0,4}.

Далее, уже ясно, что разделяющая последовательность не может состоять только из стимулов a, поскольку такая последовательность всегда оставляет группы {0,2,4} и {1,3,5} неразличимыми, но применив b в любом месте, мы «слепим» 3 состояния, два из которых принадлежат одной из таких групп. Поэтому у этого автомата нет статической различающей последовательности.

Суммируем получаемые UIO-последовательности:

0 — aababa/xyxyxy

1 — ababa/yxyxy

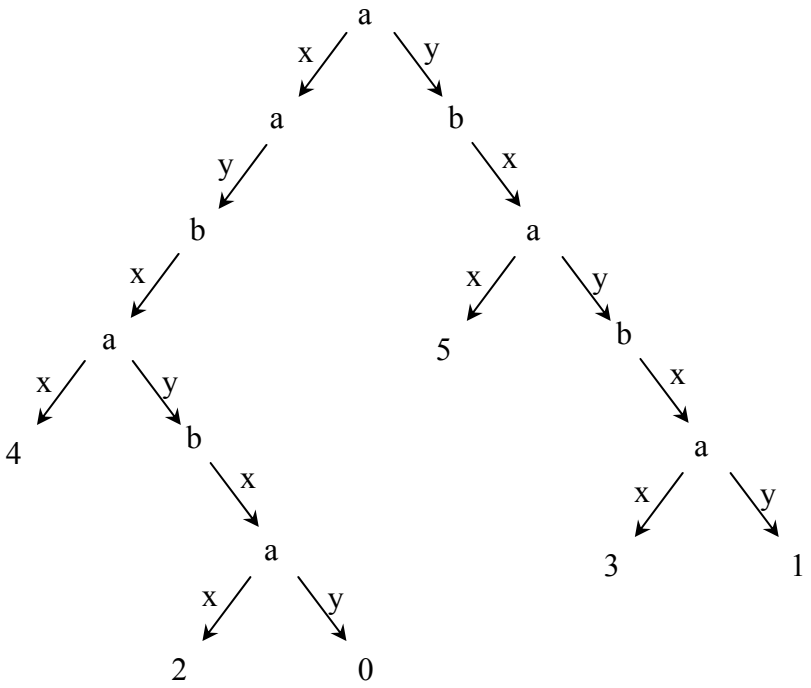
2 — baba/xyxy, aababa/xyxyx

3 — ababa/yxyxx

4 — aaba/xyxx

5 — aba/yxx

Из них можно построить адаптивную различающую последовательность. В качестве листовых вершин добавлены исходные состояния после получения соответствующих последовательностей реакций.



Если число состояний автомата равно n, а число стимулов — p, то его адаптивную различающую последовательность можно вычислить или показать, что ее не существует (при помощи несколько модифицированного алгоритма), за $O(pn^2)$ действий. Если адаптивная последовательность получается, она имеет длину не более $n(n–1)/2$

Если различающая последовательность автомата существует, она может иметь экспоненциальную от числа его состояний длину.

UIO-последовательности, если существуют, тоже могут иметь экспоненциальную длину.

Диагностическое множество всегда может быть построено за время $O(pn^2)$, при этом в нем содержится не более (n–1)-й последовательности длины не более n.

Установочные последовательности

Построение тестов

Для построения тестов на соответствие некоторому автомату нужно его знать. Такой автомат при тестировании называется *спецификацией*.

Мы предполагаем, что реальное поведение тестируемой системы, включающее все ошибки, если они есть, может быть полностью адекватно представлено некоторым конечным автоматом. Этот автомат называется *реализацией*. Мы не знаем, как он устроен, но в ходе тестирования хотим проверить, эквивалентен он спецификации или нет.

Чтобы тестирование стало возможно и реализации, и спецификация должны удовлетворять ряду требований.

В этом курсе рассматриваются методы тестирования только для детерминированных автоматов, поэтому далее будем предполагать, что спецификация и реализация детерминированы. Методы построения тестов для недетерминированных автоматов тоже есть, но формулируются существенно сложнее.

Поскольку мы хотим только проверить эквивалентность поведения, нам все равно, какой из эквивалентных автоматов брать в качестве спецификации. Поэтому можно считать, что спецификация минимальна.

Кроме того, можно предполагать, что в начале работы тестов реализация находится в начальном состоянии и алфавиты стимулов и реакций у спецификации и реализации совпадают. Если последнее не выполнено, мы любую реакцию реализации, не принадлежащую алфавиту реакций спецификации, можем рассматривать как ошибку, а стимул, не являющийся стимулом спецификации, мы просто не будем подавать.

Чтобы не возникало неопределенности при применении стимулов в неизвестных состояниях, будем считать, что и спецификация, и реализация полностью определены, то есть всегда можно применять все входные символы.

Наконец, чтобы суметь вернуться в исходное состояние, попробовав один из тестов, нужно считать, что либо спецификация и реализация сильно связны, то есть из любого их состояния можно, двигаясь по переходам, попасть в любое другое, либо что к обеим применимо специальное действие reset (далее обозначаемое R), которое переводит автомат из любого состояния в начальное и не содержит ошибок.

При наложенных ограничениях, однако, любой конечный набор тестов будет недостаточен, если не ограничить размер реализации, например, число состояний в ней. Если спецификация конечна и задан конечный набор тестов для проверки соответствия ей, мы всегда можем построить цепочку состояний, уводящую от начального, и длинную настолько, что применение всех заданных тестов закончится, не дойдя до какого-то ее состояния. Вот в этом состоянии можно сделать переход, дающий различие в поведении спецификации и реализации. Рассматриваемый тест не сможет обнаружить такую ошибку. Поэтому нужно считать, что количество состояний в реализации не превосходит некоторого заданного числа N.

Итак, от спецификации требуется

- детерминизм;
- минимальность;
- полная определенность;
- сильная связность или наличие reset.

От реализации требуется

- детерминизм;

- полная определенность;
- сильная связность или наличие reset;
- согласованность стимулов и реакций со спецификацией;
- согласованность начального состояния;
- ограниченность.

Кроме этих общих требований можно предполагать наличие вспомогательных действий, облегчающих тестирование.

- Одно из таких действий — reset (R), переводящий автоматы в начальное состояние.
- Другая возможность — наличие специального действия status (S), которое возвращает правильный идентификатор текущего состояния автомата и не изменяет состояние автомата.
- Наконец, очень полезная возможность — наличие действия set (T), которое имеет параметр-идентификатор состояния и корректно переводит автомат в это состояние.

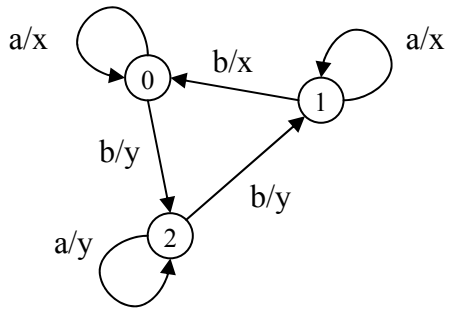
При наличии всех этих возможностей возможно *прямое тестирование*, самый простой метод тестирования автоматов. При тестировании мы хотим проверить эквивалентность автоматов, значит нужно убедиться, что любой переход реализации (определяемый своим начальным состоянием и стимулом) ведет себя так же, как и соответствующий переход в спецификации, то есть ведет в такое же состояние и возвращает ту же реакцию.

- При прямом тестировании для каждого перехода построим тест, который начинается с установки автомата в начальное состояние этого перехода с помощью set, затем выполняет сам переход и проверяет итоговое состояние с помощью status. Получаем набор тестов $\{T(s)aS\}$ для всех $s \in S$, $a \in I$. В каждом таком тесте нужно проверять, что реализация ведет себя так же, как спецификация, то есть выдает ту же реакцию и тот же идентификатор состояния в ответ на S. Если для каждого перехода это так, реализация эквивалентна спецификации.

Откажемся теперь от возможности по установке состояния, редко встречающейся на практике. В этом случае полный тест можно построить при помощи *обхода автомата*, то есть пути, проходящего по каждому его переходам по крайней мере один раз. Если спецификация сильно связна, ее обход существует.

- При тестировании на основе обхода берем одну из входных последовательностей, при применении которой выполняется обходов спецификации. В начале и после каждого стимула вставим действие S. Полученный тест проверяет каждый переход спецификации, и если реализация возвращает всегда те же реакции и те же идентификаторы состояний, она эквивалентна спецификации.

При использовании методов прямого тестирования и обхода гипотеза об ограниченности числа состояний в реализации не используется, поскольку есть мощный инструмент наблюдения реальных состояний — действие status.

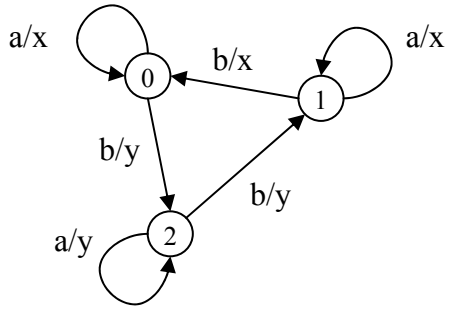


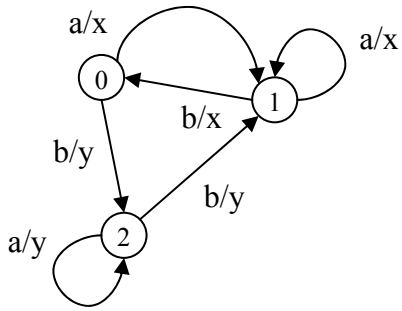
Построим полные тесты по двум описанным методам для изображенного выше автомата.

По методу прямого тестирования получаем последовательность T(0)aST(0)bST(1)aST(1)b ST(2)aST(2)bS. Выходная последовательность при этом должна быть равна x0y2x1x0y2y1.

Обход изображенного автомата выполняется, например, при применении входной последовательности ababab. По методу обхода получаем тест из входной последовательности SaSbSaSbSaSbS и корректной выходной последовательности 0x0y2y2y1x1x0.

Заметим, что при тестировании с помощью обхода действие status дает существенную информацию, без которой этот метод тестирования не может доказать эквивалентность автоматов.





Рассмотрим два автомата, изображенных выше. Автомат слева (тот же, что в разобранном выше примере) будет спецификацией, автомат справа — реализацией. Возьмем другой обход спецификации — bababa. Получаем тест SbSaSbSaSbSaS/0y2y2y1x1x0x0. На предложенной реализации этот тест дает выходную последовательность 0y2y2y1x1x0x1, которая отличается от корректной только в одном месте — последнем идентификаторе состояния.

Легко видеть, что сложность тестирования автомата с n состояниями и p стимулами методом прямого тестирования равна 3pn. Для метода обхода она не превосходит $2pn^2$ и существуют автоматы, для которых она имеет порядок $O(pn^2)$.

В следующей лекции рассматриваются более сложные методы тестирования автоматов, предназначенные для тех случаев, когда нет надежно работающих действий status и reset.

Литература

[1] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (eds.). Model Based Testing of Reactive Systems. LNCS 3472, Springer, 2005.

[2] В. Б. Кудрявцев, С. В. Алешин, А. С. Подколзин. Введение в теорию автоматов. М.: Наука, 1985.

Тестирование на основе моделей

В. В. Кулямин

Лекция 7. Автоматные методы построения тестов. Продолжение

Напомним общий контекст различных методов тестирования на основе конечных автоматов.

Рассматривается описание требований к поведению тестируемой системы, представленное в виде конечного автомата, называемого спецификацией. Реальное поведение тестируемой системы, со всеми имеющимися в ней ошибками, также может быть полностью смоделировано конечным автоматом, называемым реализацией. Реализация неизвестна, известно только, что это конечный автомат, удовлетворяющий ряду условий.

Задача состоит в построении как можно более компактного набора тестов — входных последовательностей (и соответствующих им в спецификации выходных), — позволяющих отличить реализацию от спецификации всякий раз, когда они не эквивалентны. Соответственно, если поведение реализации на этом наборе тестов не будет отличаться от поведения спецификации, можно быть уверенным, что они эквивалентны.

На спецификацию и реализацию накладываются дополнительные ограничения.

Спецификация

- детерминирована;
- минимальна;
- полностью определена;
- сильно связна или имеет действие reset(R), достоверно приводящее из любого состояния в начальное.

От реализации требуется

- детерминизм;
- полная определенность;
- сильная связность или наличие reset;
- согласованность стимулов и реакций со спецификацией — входной и выходной алфавиты реализации те же;
- согласованность начального состояния — в начале работы реализация находится в начальном состоянии;
- ограниченность — число состояний в реализации не превосходит некоторого числа N.

Методы, использующие reset

Методы построения тестов, рассматриваемые ниже, условно можно разделить на одношаговые или однофазные и многошаговые. В первых все тесты строятся по одной и той же общей процедуре, а вторые используют несколько разных процедур или шагов.

Одношаговые методы

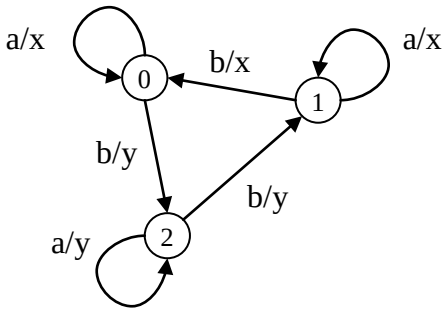
Первый метод построения тестов для автоматов без специальных действий set или status был разработан в 1973 Василевским [3]. Его более понятное изложение на английском языке приведено в статье Chow [4], поэтому большинство англоязычных авторов дают ссылки на эту статью. В статье Chow этот метод назван *W-методом* в честь Василевского.

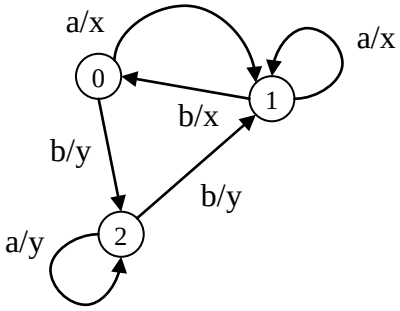
Без действия status проверить, что некоторый переход в реализации ведет в состояние, эквивалентное конечному состоянию соответствующего перехода в спецификации, становится не так просто. W-метод использует для этого диагностическое множество W, которое имеется для каждого минимального детерминированного полностью определенного автомата (см. Лекцию 6).

Как и в ранее описанных методах, необходимо для каждого перехода, однозначно определяемого своими начальным состоянием и стимулом, выполнить его, проверить корректность реакции, а затем проверить корректность его конечного состояния.

Чтобы выполнить все переходы, используется reset и покрывающее множество.

Покрывающее множество (или *базис достижимости*) C для конечного автомата — это минимальное такое множество входных последовательностей, что вместе с каждой последовательностью оно содержит все ее начала и позволят из начального состояния попасть в любое состояние автомата. То есть, #C = #S & ∀αa ∈ C α ∈ C & ∀s ∈ S ∃α ∈ C s = δ(s0, α). (Здесь # обозначает число элементов в множестве).





Для автомата, изображенного слева, покрывающим множеством является {ε, b, bb}.

Если число состояний в реализации не превосходит числа состояний в спецификации, т.е. N = n, W-метод действует так.

- В каждом тесте первым действием является reset, затем идет какая-нибудь из последовательностей из покрывающего множества (так мы оказываемся в произвольном состоянии), затем идет любой стимул (так выполняется произвольный переход), затем выполняется одна из последовательностей диагностического множества W. Чтобы проверить все переходы, нужно после каждого из них выполнить все последовательности из W.
  Другими словами этот метод можно записать так: строятся все входные последовательности из конкатенации множеств CIW, перед каждой из них вставляется R и все полученные последовательности конкатенируются.

Построим набор тестов для изображенного выше слева автомата с помощью W-метода.

В рассматриваемом автомате C = {ε, b, bb}, I = {a, b}, W = {a, b}. Значит, IW = {aa, ab, ba, bb}, и поэтому {R}CIW = RaaRabRbaRbbRbaaRbabRbbaRbbbRbbaaRbbabRbbbaRbbbb.

Корректная последовательность реакций:

xx.xy.yy.yy.yyy.yyy.yyx.yyx.yyxx.yyxx.yyxx.yyxy.

Если с помощью этого теста проверять реализацию, изображенную выше справа, будет получен следующий результат — xx.xx.yy.yy.yyy.yyy.yyx.yyx.yyxx.yyxx.yyxx.yyxy. Полученная ошибка выделена красным цветом.

- Для произвольного N >= n W-метод строит набор тестов $\{R\}CI^{N-n+1}W$. То есть, вместо однократных стимулов при построении всех возможных переходов используются все возможные последовательности стимулов длины N–n+1.

W-метод может быть применен для произвольной детерминированной полностью определенной спецификации, однако он дает достаточно большой набор тестов. Для

сокращения размеров тестов можно использовать другие способы идентификации состояний (см. Лекцию 6), например, UIO-последовательности и различающие последовательности.

D-метод применяется для спецификаций, имеющих различающую последовательность d. Он полностью аналогичен W-методу, только вместо диагностического множества W используется различающая последовательность d.

- D-метод строит набор тестов $\{R\}CI^{N-n+1}d$.

Для уже рассматриваемой ранее спецификации существует различающая последовательность d = ab. Поэтому построенный с помощью D-метода тест для N = n выглядит так: RaabRbabRbaabRbbabRbbaabRbbbab/xxy.yyy.yyyy.yyxx.yyxxx.yyxxy. Для приведенной выше ошибочной реализации результат его выполнения выглядит так: xxx.yyy.yyyy.yyxx.yyxxx.yyxxx.

В D-методе наряду со статической может также использоваться адаптивная различающая последовательность.

UIO-метод использует после каждого перехода вместо диагностического множества UIO-последовательность конечного состояния этого перехода. Однако, UIO-метод не гарантирует обнаружения всех ошибок — UIO-последовательности изменяются из-за ошибок, и поэтому может существовать ошибочная реализация, в которой для ошибочного конечного состояния одного из переходов UIO-последовательность дает корректные результаты.

Многошаговые методы

Другим методом, позволяющим сократить размер тестов, является *частичный W-метод* или *Wp-метод*. Он, в отличие от ранее представленных методов, выполняется в несколько шагов и использует идентифицирующие множества.

Идентифицирующее множество (identification set) W_s для состояния s — такое множество входных последовательностей, что для любого другого состояния автомата одна из этих последовательностей дает результат, отличающийся от результата ее применения в s. То есть, $\forall s_1 \in S\ s_1 \neq s \Rightarrow \exists \alpha \in W_s\ \lambda(s, \alpha) \neq \lambda(s_1, \alpha)$.

В качестве идентифицирующего множества состояния всегда можно выбрать некоторое подмножество диагностического множества автомата.

Шаги Wp-метода для N = n следующие.

- Сначала выполняются тесты $\{R\}CW$. Их успешное выполнение гарантирует, что все состояния реализации подобны состояниям спецификации, т.е. выдают те же результаты на все последовательности из W.
- Каждый переход, не проверенный на предыдущем шаге, т.е. не покрытый при выполнении множества C, выполняется и проверяется с помощью идентифицирующего множества своего конечного состояния.

Построим набор тестов по Wp-методу для той же спецификации, изображенной выше.

$C = \{\varepsilon, b, bb\}$, $I = \{a, b\}$, $W = \{a, b\}$, $W_0 = W$, $W_1 = \{b\}$, $W_2 = \{a\}$.

Первый шаг дает тест RaRbRbaRbbRbbaRbbb/x.y.yy.yy.yyx.yyx.

На втором шаге требуется проверить только переходы по стимулу a и переход по стимулу b в состоянии 1. Получаем RaaRabRbaaRbbabRbbbaRbbbb/xx.xy.yyy.yyxx.yyxx.yyxy.

Полный тест выглядит следующим образом: RaRbRbaRbbRbbaRbbbRaaRabRbaaRbbab RbbbaRbbbb/x.y.yy.yy.yyx.yyx.xx.xy.yyy.yyxx.yyxx.yyxy.

Результат его выполнения для приведенной выше ошибочной реализации: x.y.yy.yy.yyx.yyx.xx.xx.yyy.yyxx.yyxx.yyxy.

В том случае, когда N > n, в W_p-методе добавляются дополнительные шаги, а именно.

- На (i+1)-ом шаге для 2<= i < N-n+2 для каждой цепочки переходов длины i, которая не была проверена на одном из предудущих шагов (поскольку начало некоторых цепочек лежит на C), надо пройти по последовательности из C в начало этой цепочки, выполнить эту цепочку, а потом проверить ее конечное состояние при помощи его идентифицирующего множества.

Если у каждого состояния автомата есть UIO-последовательность, можно использовать *UIOv-метод*, гарантирующий, в отличие от UIO-метода, обнаружение всех ошибок.

UIOv-метод получается из Wp-метода заменой W на множество, содержащее UIO-последовательности всех состояний, а W_s — на UIO-последовательность состояния s.

В нашем примере полный тест по UIOv-методу выглядит так:

RaRbRabRbaRbbRbabRbbaRbbbRbbabRaabRbaaRbbabRbbbab/x.y.xy.yy.yy.yyy.yyx.yyx.yyxx.xxy.yyy.yyxx.yyxxy.

Сложность тестов в общем случае и их минимизация

Несмотря на то, что D-метод и Wp-метод обычно строят тесты, меньшие по размерам, чем W-метод, их сложность в общем случае описывается иначе.

В общем случае размер и сложность вычисления тестов по W-методу или Wp-методу одинакова и равна по порядку $O(p^{N-n+1}n^3)$ ($O(pn^3)$ для N = n). Эта оценка не может быть улучшена — существуют спецификации, которые нельзя отличить от всех ошибочных реализаций с N состояниями с помощью набора тестов, имеющего размер меньше, чем $O(p^{N-n+1}n^3)$.

Поскольку длина различающей последовательности может быть экспоненциальной от числа состояний, сложность D-метода в общем случае тоже экспоненциальная.

Для каждого конкретного случая можно, однако, сокращать размер тестов, построенных этими методами. Основное правило, которое используется при сокращении — если начало одно из элементарных тестов (последовательностей, заключенных между двумя reset'-ами) совпадает с другим элементарным тестом, то второй элементарный тест можно выбросить. Ясно, что в этом случае ошибка, обнаруживаемая выбрасываемым тестом всегда обнаруживается более длинным тестом.

Рассмотрим все полученные для нашего примера тесты.

- W-метод.
  RaaRabRbaRbbRbaaRbabRbbaRbbbRbbaaRbbabRbbbaRbbbb.
  Сокращение дает
  RaaRabRbaaRbabRbbaaRbbabRbbbaRbbbb.
- D-метод.
  RaabRbabRbaabRbbabRbbaabRbbbab.
  Сократить нельзя.
- Wp-метод.
  RaRbRbaRbbRbbaRbbbRaabRbaaRbbabRbbbab.
  Сокращение дает
  RaabRbaaRbbabRbbbab.
- UIOv-метод.
  RaRbRabRbaRbbRbabRbbaRbbbRbbabRaabRbaaRbbabRbbbab.
  Сокращение дает
  RbabRaabRbaaRbbabRbbbab.

Таким образом, в этом примере эквивалентность любой реализации с не более чем 3-мя состояниями может быть проверена с помощью теста

RaabRbaaRbbabRbbbab/xxy.yyy.yyxx.yyxxy.

Методы, не использующие reset

При отсутствии надежно работающей операции reset тестирование автоматов становится несколько сложнее. Существуют методы, позволяющие построить тесты для произвольной детерминированной полностью определенной сильно связной спецификации, но размер получаемых тестов может быть достаточно велик. Такие методы используют установочные последовательности (homing sequences).

Рассмотрим здесь только один метод, работающий без reset, и предполагающий? Что спецификация обладает различающей последовательностью d.

Поскольку спецификационный автомат сильно связен, для каждой пары его состояний s_1 и s_2 существует *переводящая последовательность* стимулов $t(s_1, s_2)$, выполнение которой в s_1 переводит автомат в состояние s_2.

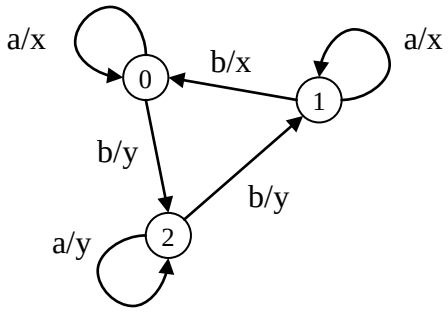
Обозначим для каждого $i >= 0$ через s_i' итоговое состоянием после выполнения d в s_i.

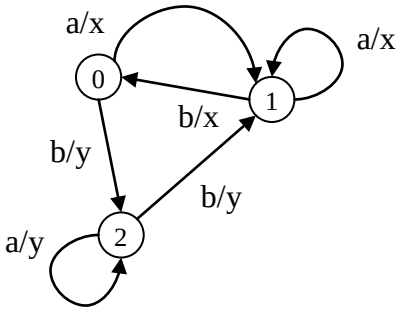
Тогда тест $d\ t(s_0', s_1)\ d\ t(s_1',s_2)\ d \ldots d\ t(s_{n-2}',s_{n-1})\ d$ проверяет, что в реализации для каждого состояния спецификации есть подобное, в котором d дает ту же последовательность реакций.

Чтобы после этого проверить, что некоторый переход работает правильно, нужно перейти в начало перехода s_i, выполнить его и выполнить d. Ошибки в реализации могут привести не в начало этого перехода, в другое место. Однако, мы уже знаем, что последовательность $dt(s_{i-1}', s_i)$, будучи применена в состоянии s_{i-1}, во-первых, проверит, что это действительно такое состояние с помощью d, а во-вторых, приведет после этого в s_i уже проверенным способом. Поэтому, попав в некоторое состояние s, для еще не проверенного перехода s_i –a-> s' выполним $t(s, s_{i-1})\ d\ t(s_{i-1}', s_i)\ a\ d$.

Получаемая таким образом последовательность обеспечит проверку всех переходов.

Построим такой тест для нашего примера спецификации.





Имеем $d = ab$, $s_0' = s_2$, $s_1' = s_0$, $s_2' = s_1$. Если мы будем обходить состояния в порядке s_0-s_2-s_1, то переводящие последовательности пусты. Поэтому первый этап дает abababab, и в его конце мы оказываемся в состоянии s_2.

Далее будем проверять переходы в следующем порядке: 1-a->1, 2-a->2, 0-a->0, 1-b->0, 0-b->2, 2-b->1. В этом случае промежуточные переводящие последовательности пусты, за исключением двух последних случаев, поэтому на втором этапе получаем такую входную последовательность: abaab.abaab.abaab.abbab.babbab.babbab.

Итоговый тест:

abababab.abaab.abaab.abaab.abbab.babbab.babbab/xyyyxxxy.yyxxx.xyyyy.xxxxy.yyxxy.yxxyy y.xxyyxx.

Результат, возвращаемый ошибочной реализацией: xxxxxxxx…, после этого тестирование можно не продолжать.

В этом примере многие ошибки могут быть найдены уже на первом этапе, поскольку в полученную на нем последовательность входят все переходы. Однако, если бы в спецификации имелись переходы по другим символам, отличным от a и b, ошибки в них обнаруживались бы только на втором этапе.

Использование других автоматных моделей

Другие автоматные модели — системы размеченных переходов, расширенные или взаимодействующие автоматы — при их использовании для тестирования чаще всего приводятся к конечным автоматам.

После этого становится можно использовать большое количество методов тестирования, разработанных для конечных автоматов.

Литература

[1] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (eds.). Model Based Testing of Reactive Systems. LNCS 3472, Springer, 2005.

[2] В. Б. Кудрявцев, С. В. Алешин, А. С. Подколзин. Введение в теорию автоматов. М.: Наука, 1985.

[3] М. П. Василевский. О распознавании неисправностей автоматов. Кибернетика, 9(4):93-108, 1973.

[4] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering, 4(3):178-187, 1978.

Тестирование на основе моделей

В. В. Кулямин

Лекция 8. Основы технологии разработки тестов UniTESK

Разработка тестов и тестирование на основе моделей предполагают, что используемые модели формулируются явно. Одной из технологий такого типа является разработанная в Институте системного программирования РАН в 1995-2002 технология UniTESK.

Назначение технологии UniTESK — разработка и развитие наборов тестов для сложных развивающихся систем. Сложность системы означает достаточно большое количество функций, сложный интерфейс (от нескольких десятков операций), достаточно большой размер кода (от $5 \cdot 10^4$ строк). Развитие системы предполагает, что время от времени появляются ее новые версии, и что набор тестов придется поддерживать в рабочем состоянии и пополнять тестами новых функций в течение значительного периода времени (начиная от 2-х лет, для более чем 2-х версий). Сказанное не означает, что при невыполнении этих условий технологию нельзя применить, просто при этом разработка тестов традиционными методами может быть более эффективной с точки зрения затрат на определенное качество тестов.

Концептуальная основа технологии UniTESK такова.

- Тесты разрабатываются исходя из некоторой модели поведения тестируемой системы. Критерии полноты тестирования определяются, чаще всего, в терминах этой же модели. Все это позволяет вести разработку тестов независимо от разработки тестируемых компонентов. Модель поведения служит источником для автоматического построения тестовых оракулов.
- Используемая модель поведения обычно создается на основе требований и желательных свойств системы, а не на основе использованных в ней проектных решений. За счет этого тестовый набор необходимо модифицировать, в основном, из-за изменений в требованиях, а не в коде системы. Полученные в результате тесты могут без модификаций или с небольшими изменениями использоваться для тестирования разных версий системы и различных систем, реализующих одни и те же функции, один и тот же стандарт. Различия в интерфейсах между разными системами при этом локализуются в небольшой части тестового набора — тестовых адаптерах.
- Для оформления моделей используются языки, насколько это возможно, близкие к языкам, используемым при разработке системы, с тем чтобы ее разработчики и архитекторы тратили как можно меньше усилий на их понимание. Обычно используются широко распространенные языки программирования или их небольшие расширения.
- Тесты строятся на основе модели поведения и критериев полноты тестирования с помощью нескольких различных техник. Выбор этих техник зависит от требуемого вида тестирования (проверка базовой функциональности, основных сценариев использования, сочетаний различных условий и ограничений и пр.), необходимой полноты, сложности входных данных и состояния системы, наличия параллелизма и асинхронности в ее поведении и других факторов. При этом сочетаются нацеленное тестирование, комбинаторное построение тестовых данных и автоматные техники для проверки работы системы в различных состояниях.

Технология UniTESK состоит из следующих частей.

- Метод разработки тестов, определяющий набор видов деятельности и решаемых ими задач, последовательность их выполнения и правила применения различных техник в зависимости от складывающейся в проекте ситуации.
- Архитектура тестового набора, определяющая основные виды компонентов тестов, связи и способы взаимодействия между ними и правила расширения и модификации созданных тестовых наборов.
- Набор техник построения и организации тестов.
- Набор языков, на которых разрабатываются модели. Большинство таких языков являются расширениями языков программирования, построенными по общим правилам.
- Инструменты, автоматизирующие работу с моделями на определенных языках и построение тестов из них.

Далее дается описание метода разработки тестов. Оно сопровождается примерами построения моделей и тестов на их основе, в основном на расширении языка Java.

Метод

Используемый метод построения тестов включает в себя следующие виды деятельности.

1. Определение целей и рамок проекта.
2. Определение и анализ требований к тестируемой системе.
3. Определение и анализ требований к полноте тестирования.
4. Разработка и выполнение тестов.
5. Анализ результатов тестирования.

Эти виды деятельности обычно выполняются примерно в той последовательности, в которой перечислены выше. Однако при необходимости используется итеративная разработка — т.е. возможны (неоднократные) возвращения от следующих видов деятельности к предыдущим для выполнения каких-то доработок или изменений. Кроме того, после проведения декомпозиции системы на отдельные компоненты в рамках первой или второй деятельности, дальнейшая разработка тестов для каждого компонента может идти независимо от остальных, поэтому разные действия для различных компонентов часто выполняются одновременно и параллельно.

Далее каждый из видов деятельности рассматривается более подробно.

Определение целей и рамок проекта

В ходе выполнения этого вида деятельности принимаются основные стратегические решения, касающиеся проекта. Выявляются границы тестируемой системы, основные проверяемые функции и свойства, интерфейс, которым можно пользоваться, основные риски, на преодоление которых нацелено тестирование. Определяются компоненты системы, которые можно тестировать независимо, набор видов создаваемых тестов и используемые при этом техники.

Принимаемые на этом этапе решения зависят от 3-х видов факторов.

- Контекст предметной области: какие требования выдвигаются к системам такого рода, какие есть документы и знания о предметной области, какие стандарты действуют в данной предметной области, что известно о возможностях использования целевой системы, о потребностях ее пользователей, заказчиков и третьих лиц, о решаемых системой задачах, возможных методах решения этих задач и возможном устройстве соответствующих компонентов системы,.
- Контекст текущего проекта: какие ресурсы (люди, время, деньги, аппаратное и программное обеспечение, другое оборудование) имеются в нашем распоряжении, какие требования к данной системе и ее тестированию выдвигают заказчик и

другие заинтересованные лица (конечные пользователи системы, ее разработчики, контролирующие и лицензирующие организации и пр.), какие другие проекты зависят или, вероятно, будут зависеть от целевой системы и от результатов этого проекта, какие требования предъявляются или, вероятно, будут предъявляться ими к целевой системе и к ее тестам.

- Архитектура целевой системы, насколько она известна на момент начала работ: разбиение системы на компоненты, задачи, решаемые различными ее компонентами, возможные сценарии взаимодействия между ними, а также правила внесения модификаций в имеющиеся компоненты и добавления новых.

Определение рамок проекта включает в себя решение следующих задач.

1. **Определение задач и рамок тестируемой системы.**
   В рамках этой задачи нужно определить, что представляет собой тестируемая система (system under test, SUT), для чего она предназначена — какие основные задачи решает. Также важно, из каких частей она состоит — что входит, а что не входит в нее, какие именно части нужно тестировать, можно ли их отделить от других частей/других систем при проведении тестирования, и как.
2. **Определение набора проверяемых функций и свойств**.
   Нужно выяснить, полный список функций (функция, feature — это некоторая услуга, предоставляемая системой), которые необходимо протестировать, а также список других свойств, которые необходимо проверять. На этом этапе выявляется только общий набор свойств и функций, без детализации.
3. **Определение используемого при тестировании интерфейса**.
   Необходимо определить набор операций или действий, с помощью которых можно воздействовать на систему, и набор ее возможных реакций и событий, создаваемых ею. Нужно стремиться найти такой интерфейс, который позволит как можно точнее оценить проверяемые свойства, с минимальными искажениями от других частей системы или других систем. Иногда, однако, можно пользоваться только таким интерфейсом, который вынуждает работать не только проверяемую функциональность, но и множество других компонентов, вносящих определенные возмущения в наблюдаемые реакции, которые нужно учитывать при тестировании.
4. **Оценка важности различных характеристик, функций и элементов системы**.
   Наборы проверяемых функций и свойств, а также компонентов системы необходимо проранжировать по их важности для основных заинтересованных лиц проекта. При этом необходимо не только использовать пожелания руководителя проекта или пользователей, но и провести аккуратный анализ зависимостей функций и анализ рисков их некорректной работы в различных ситуациях — иногда представления самих разработчиков и пользователей системы о значимости возможных ошибок для оценки работы системы в целом не являются вполне адекватными.
5. **Первичная декомпозиция интерфейса на группы связанных операций**.
   Весь набор операций, действий или событий, с помощью которых можно воздействовать на систему и получать от нее информацию и реакции, нужно разбить на группы логически связанных операций и событий. Каждая такая группа совместно реализует одну или несколько тесно связанных функций. Обычно прямые и обратные операции (создать/уничтожить, добавить/удалить, записать/стереть, войти/выйти и пр.) помещаются в одну группу.
   На этом этапе производится только первичная декомпозиция, предназначенная для предварительного планирования и оценки трудоемкости дальнейших работ. В дальнейшем эта декомпозиция может изменяться и уточняться, поскольку иногда выявляются неявные, но важные, связи и зависимости между функциями и операциями.

Помимо собственно декомпозиции на группы в рамках этой деятельности определяются зависимости между отдельными группами, позволяющие упорядочить разработку модели поведения от базовых понятий и операций к более сложным, использующим эти базовые.

6. **Выбор методов тестирования.**
   Для каждой из выделенных групп операций выявляются ее существенные с точки зрения организации тестирования свойства. К таким свойствам относятся требуемый вид тестирования (проверяется ли только базовая функциональность, основные способы использования или сложные сценарии работы, используются ли некорректные входные данные или нет), наличие или отсутствие работы с внутренним состоянием, высокая или низкая сложность входных данных и части состояния системы, связанной с данной группой, наличие или отсутствие зависимостей от других групп, важность или несущественность (отсутствие возможности) асинхронных взаимодействий с системой в рамках данной группы. В соответствии с выявленными особенностями выбирается наиболее подходящий и эффективный набор техник построения тестов в рамках технологии и видов тестов, которые будут разрабатываться для данной группы.
7. **Выбор аспектов и детальности формализации (уровня абстракции)**.
   В соответствии с выявленными особенностями данной группы и важностью проверки отдельных свойств входящих в нее операций определяется уровень абстракции выполняемой формализации требований — аспекты и детали функциональности и др. свойств, которые должны быть описаны и проверятся, и те аспекты и детали, которые будут проигнорированы.

В дальнейшем различные виды деятельности выполняются уже не по отношению ко всей тестируемой системе, а для выделенных групп интерфейсных операций в соответствии с порядком, устанавливаемым их зависимостями друг от друга — от независящих ни от чего к тем, которые зависят от ранее рассмотренных. Тесты для группы, которые не зависят друг от друга даже косвенно, могут разрабатываться параллельно. При необходимости проводится уточнение или исправление первичной декомпозиции, после чего часть уже выполненных работ во определению требований, критериев полноты или разработке тестов бывает необходимо переделать в соответствии с внесенными изменениями.

Рассмотрим небольшой пример. Пусть необходимо разработать тесты для классов, реализующих коллекции в пакете java.util библиотеки JDK. Всего в этом пакете 19 интерфейсов, 59 классов (14 из них — абстрактные, т.е. предназначенные для создания классов-наследников на их основе), 1 перечислимый тип и 21 класс исключений. Помимо коллекций, эти классы реализуют работу с датами, временем и таймерами, с географическими и национальными настройками системы, с наборами конфигурационных свойств, простейшие операции с регулярными выражениями, а также предоставляют базовые классы для образца «Подписчик». Непосредственно к коллекциям имеют отношение 16 интерфейсов, 27 классов и 3 класса исключений.

Анализ имеющейся документации (http://java.sun.com/javase/6/docs/api/) дает следующую дополнительную информацию.

- Вместо интерфейса Enumeration рекомендуется использовать Iterator, хотя последний более специфичен. По-видимому, первый интерфейс оставлен, в основном, для обеспечения совместимости со старыми программами, написанными до появления Java 1.2.
- Устаревшим также считается классы Dictionary (вместо него нужно использовать интерфейс Map).
- Классы Hastable и Vector очень похожи по выполняемым функциям на HashMap и ArrayList. Даже отличающиеся по имени их методы имеют аналогичную

функциональность. Единственное важное отличие — все методы первых двух классов синхронизованы. Это означает, что они рассчитаны на корректную работу и при асинхронных обращениях к их объектам из различных потоков. Однако, если такие свойства не важны, можно совсем не использовать эти классы.

Рассмотрим теперь отдельно разработку тестов для списков. Список — линейно упорядоченная коллекция объектов, предоставляющая доступ ко всем своим элементам по их индексам. Из классов и интерфейсов коллекций в java.util к спискам относятся один интерфейс List и 5 классов, реализующих его. Два класса — AbstractList и AbstractSequentialList — являются абстрактными, их методы доступны только через наследников (например, ArrayList и LinkedList). Класс Vector считается уже устаревшим, поэтому тесты для него могут не понадобиться.

Класс ArrayList в дополнение к методам интерфейса List предоставляет только методы управления вместимостью (capacity) массива, на основе которого построен данный список. Класс LinkedList содержит дополнительные методы работы с первым и последним элементами. Если тестировать эти функции не нужно, то для обоих классов можно разработать унифицированный набор тестов, проверяющих только методы общего интерфейса. Этот набор тестов можно будет использовать для тестирования соответствующей функциональности любого класса, реализующего интерфейс List.

Определение и анализ требований к тестируемой системе

В рамках данной деятельности определяются требования к тестируемой системе — все ограничения и свойства, которые нужно проверять в ходе тестирования. Кроме того, эти требования систематизируются и оформляются в виде модели поведения системы.

Определение требований и их формализация разбиваются на следующие задачи.

1. **Определение источников требований.**
   Прежде**,** чем начинать выделять отдельные требования, необходимо выявить все возможные их источники. К таким источникам относятся следующие.
   - Проектная и пользовательская документация на систему.
   - Архитекторы, проектировщики и разработчики данной системы.
   - Бизнес-аналитики, эксперты в данной предметной области, опытные пользователи и руководители, технологи в организациях-заказчиках и контролирующих организациях.
   - Стандарты, относящиеся к данной предметной области.
   - Другие аналогичные системы и их документация.
2. **Выделение и сбор требований.**
   После определения источников требований можно начать собирать отдельные требования — все существенные ограничения на работу системы, которые можно найти в документах или выделить из общения с экспертами в предметной области. Каждое такое утверждение должно хранить ссылку на источник, из которого оно получено.
3. **Систематизация требований.**
   После выделения набора требований их необходимо превратить в некоторую систему — каждое требование должно иметь уникальный идентификатор, чтобы на него можно было ссылаться в дальнейшем, должны быть выявлены связи между отдельными требованиями — одни из них могут уточнять другие или являться их следствием. В дальнейшем эти связи помогут аккуратно вносить изменения в систему требований, сразу выявляя нарушения ее целостности и возможные противоречия. Требования должны быть классифицированы по обязательности их выполнения — как минимум нужно отделять обязательные от опциональных.

Нужно стремиться привести требования в проверяемый вид — чтобы обладающий определенной квалификацией человек мог по некоторым результатам работы системы уверенно утверждать, выполнены они или нет. Это не всегда удается, но, по крайне мере, нужно отделить проверяемые требования от прочих.

4. **Уточнение, согласование и устранение противоречий и неполноты**.
   При систематизации и формализации требований выявляется множество дефектов — недостаточно ясные и точные формулировки, противоречия между разными требованиями или пожеланиями различных заинтересованных лиц, неполнота — отсутствие необходимой информации о свойствах системы в каком-либо аспекте. Все эти дефекты необходимо устранить, часто привлекая для этого различных заинтересованных лиц, чтобы выяснить точное значение некоторой фразы или согласовать их противоречивые интересы и привести к разумным компромиссам относительно свойств системы.
5. **Финальная декомпозиция.**
   После приведения требований в систему необходимо проанализировать первичную декомпозицию интерфейса системы и внести в нее необходимые изменения, связанные с полученной более полной информацией о работе различных функций и операций и об их зависимостях друг от друга.
6. **Формализация требований в виде модели поведения**.
   Часто параллельно систематизации и согласованию проводится формализация требований — построение формально описанной модели поведения системы, включающей все ее аспекты, важные с точки зрения данного проекта. В рамках UniTESK используются модели в виде расширенных автоматов, программных контрактов. Для описания сложных структур данных используются иерархические структурные модели (по сути — описания типов данных в виде наборов их полей, также имеющих некоторые типы или являющихся ссылками) или грамматики, пополненные дополнительными атрибутами (но не атрибутные грамматики в смысле Кнута).

Рассмотрим в качестве примера разработку модели поведения в виде программного контракта для интерфейса списка. Его методы можно условно разделить на следующие группы: работа со списком как с целым (equals, hashCode, toString, toArray, clear, size, isEmpty) , методы перебора (iterator, listIterator), методы для работы с отдельными элементами списка (add, contains, get, indexOf, lastIndexOf, remove, set), работа с коллекциями элементов (addAll, containsAll, removeAll, retainAll), работа с подсписками (subList).

Для определения требований достаточно использовать имеющуюся документацию на интерфейс List (http://java.sun.com/javase/6/docs/api/). Она дана в достаточно систематизированном виде, поэтому задачи по выделению и систематизации требований можно считать уже решенными.

Чтобы построить модель поведения списка в виде программного контракта, необходимо определить структуру состояния, содержащую информацию достаточную для описания всех ограничений на его операции, а для каждой операции определить предусловие и постусловие (которые могут зависеть от текущего состояния).

Для полного описания поведения почти всех операций списка необходимо знать полный набор его элементов и их индексы. Поэтому в качестве структуры состояния нам придется использовать тоже список. Важно только, чтобы его реализация как-то отличалась от тех, которые нам предстоит тестировать — иначе при тестах будет проверяться только то, что примененные к двум различным одинаково реализованным объектам операции дают одинаковые результаты.

Метод subList существенно отличается от других — для описания его поведения нужно иметь дополнительную структуру подсписков, построенных на основе данного. В документации сказано, что такие подсписки являются только представлением, «другим

взглядом» на данные исходного списка, т.е. при их создании не создается новая коллекция, а при их модификации модифицируется и содержимое исходного списка. Оставим пока формализацию этой функциональности за рамками рассмотрения. С учетом этого упрощения описание структуры состояния списка может быть сделано так.

```
public specification class ListSpecification<E>
{
  protected E[] items;
  ...
}
```

Теперь опишем метод add(int index, E element). В документации на этот метод сказано следующее.

> Вставляет указанный объект в список на указанную позицию. Элементы, находившиеся в списке на этой позиции или после нее, сдвигаются на одну позицию вперед.
> **Параметры:**
> index – номер позиции, на которую нужно вставить указанный объект.
> element – объект, который нужно вставить.
> **Создаваемые исключения:**
> UnsupportedOperationException – создается, если этот метод не поддерживается данной реализацией списка.
> ClassCastException – создается, если класс указанного объекты препятствует его вставке в список.
> NullPointerException – создается, если указанный объект равен null и данная реализация списка не может хранить null в качестве элемента.
> IllegalArgumentException – создается, если указанный объект не может быть вставлен в данный список.
> IndexOutOfBoundsException – создается, если указана некорректная позиция (index < 0 || index > size()).

Поскольку обращаться к этому методу можно в произвольной ситуации, его предусловие должно быть всегда выполнено.

Кроме того, для простоты предположим, что исключения первых четырех видов не возникают — т.е. тестируемая реализация поддерживает операцию add, может иметь элементы любого типа, подходящего с точки зрения типа второго параметра этого метода и может иметь null в качестве элемента. Это уже не произвольная реализация интерфейса List.

Ниже приведена спецификация метода add, использующая два вспомогательных метода: для сравнения объектов, каждый из которых может быть равен null, и для сравнения участков массивов.

```
public static boolean equalObjects(E o1, E o2)
{
  return    o1 == null && o2 == null
         || o1 != null && o1.equals(o2);
}

public static boolean equalArrays(
     E[] first,  int firstStart
   , E[] second, int secondStart, int number
  )
{
  for(int i = 0; i < number; i++)
    if(!equalObjects(first[i+firstStart], second[i+secondStart]))
      return false;
  return true;
}

public specification void add(int i, E o)
  throws IndexOutOfBoundsException
{
  post
  {
    E[] oldItems = (E[])(pre items.clone());
```

```
        if(i < 0 || i > oldItems.length)
          return    thrown != null
                 && thrown instanceof IndexOutOfBoundsException
                 && items.length == oldItems.length
                 && equalArrays(items, 0, oldItems, 0, items.length);
        else
          return    thrown   == null
                 && items[i] == o
                 && items.length == oldItems.length + 1
                 && equalArrays(items, 0,   oldItems, 0, i)
                 && equalArrays(items, i+1, oldItems, i, oldItems.length-i);
      }
    }
```

В постусловии используется *оператор пре-выражения* pre, результатом применения которого к некоторому выражению является значение этого выражения непосредственно перед началом работы описываемой операции. В данном примере он использован для получения предшествовавшего операции набора элементов списка.

Определение и анализ требований к полноте тестирования

Помимо требований к проверяемой системе, определяющих, что проверять, для построения тестов необходимы требования к полноте тестирования, определяющие, в каких ситуациях нужно выполнять проверки.

В рамках этого вида деятельности нужно решить следующие задачи.

1. **Определение критериев полноты для данного проекта**.
   Критерии полноты тестирования определяются, прежде всего, на основе структуры требований. Кроме этого, учитываются наиболее важные риски, связанные с качеством тестируемой системы. Если известны наиболее критичные, а также наиболее рискованные, ее функции и компоненты, типы наиболее вероятных ошибок, эти функции и компоненты должны проверяться более тщательно, а ситуации, соответствующие наиболее рискованным действиям, должны входить в задаваемый критерий полноты отдельно.
2. **Формализация критериев полноты**.
   Выбранные критерии полноты тестирования нужно представить в виде правил выбора структурных элементов модели поведения, которые необходимо задействовать в тестах. Используется комбинация правил из некоторого общеупотребительного набора — критерий полноты задается в терминах покрытия различных способов поведения, указанных в спецификациях, покрытия комбинаций условий, касающихся выбираемого поведения или используемых данных.
   При использовании грамматик критерий покрытия может задаваться указанием того, что нужно покрыть все правила грамматики, все альтернативы, возможные комбинации альтернатив в рамках одного или нескольких правил, если в одних их них присутствуют нетерминалы, определяемые другими. Дополнительно указываются ограничения на число раскрытий неограниченных списков, используемых в правилах.

В рассмотренном выше примере постусловие метода add списка задает два сильно отличающихся поведения, две *ветви функциональности*. При одном из них создается исключение, а содержимое списка не изменяется, при другом исключения нет, а добавляемый элемент вставляется на указанное первым аргументом место. Различие между этими поведениями отражается в различных выражениях, использованных для их описания при вычислении результата постусловия.

Чтобы явно выделить различные поведения или ветви функциональности, используется оператор оператора **`branch`**. Чтобы сделать покрытие той или иной ветви более

управляемым, условия их выполнения должны зависеть только от входных данных метода — состояния списка при его вызове и значений параметров.

После добавления указания ветвей функциональности постусловие метода add приобретает следующий вид.

```
post
{
  E oldItems[] = (E[])items.clone();

  if(i < 0 || i > oldItems.length)
  {
    branch ExceptionalCase;
    return    thrown != null
           && thrown instanceof IndexOutOfBoundsException
           && items.length == oldItems.length
           && equalArrays(items, 0, oldItems, 0, items.length);
  }
  else
  {
    branch NormalCase;
    return    thrown   == null
           && items[i] == o
           && items.length == oldItems.length + 1
           && equalArrays(items, 0,   oldItems, 0, i)
           && equalArrays(items, i+1, oldItems, i, oldItems.length-i);
  }
```

Поскольку при выборе пути до оператора **branch** могут использоваться только пре-состояние объекта и значения входных параметров, удобно считать, что положение этого оператора соответствует точке вызова тестируемой операции. Все действия, выполняемые до одного из таких операторов считаются выполненными до вызова, и, соответственно, могут использовать только пре-состояние объекта и аргументы метода. Все действия, выполняемые после такого оператора, выполняются после вызова проверяемой операции, и, соответственно, могут использовать ее результат (обозначаемый в постусловии именем самой операции) и пост-состояние объекта. Для обращения после оператора **branch** к значениям в пре-состоянии должен использоваться оператор пре-выражения.

Различные поведения задают естественный *критерий покрытия ветвей функциональности* — полнота тестирования по нему определяется процентом задействованных ветвей функциональности от их общего числа.

Другие ситуации, которые нужно задействовать в ходе тестирования, можно описать в постусловиях соответствующих операций при помощи *операторов разметки ситуаций*. Допустим, в нашем примере нужно особо проверить работу метода add для пустых списков. Включить эту ситуацию в заданный спецификациями критерий покрытия можно следующим образом.

```
post
{
  if(items.length == 0) mark “Empty list”;
  E oldItems[] = (E[])items.clone();

  if(i < 0 || i > oldItems.length)
  {
    branch ExceptionalCase;
    ...
  }
```

```
    else
    {
      branch NormalCase;
      ...
    }
```

При этом возникает более мелкое разбиение всех ситуаций согласно *критерию помеченных путей* — каждая последовательность из выполненных операторов оператора **branch** и **mark** задает помеченный путь, полнота тестирования определяется процентом покрытых помеченных путей от общего числа достижимых.

Допустим, что помимо метода add мы также описали методы remove(int index) и indexOf(E element), отметив в remove в качестве различных поведений нормальное поведение и поведение с созданием исключения, а в indexOf — поведение в ситуации, когда искомый объект находится в списке, и когда его там нет. Кроме того, с помощью меток отметим отдельно случаи пустого списка и списка с единственным элементом. Соответствующие спецификации даны ниже.

```
public static boolean arrayContains(E[] a, int start, int number, E o)
{
  for(int i = 0; i < number; i++)
    if(equalObjects(a[i + start], o)) return true;
  return false;
}

public specification int indexOf(E o)
{
  post
  {
    if     (items.length == 0) mark “Empty list”;
    else if(items.length == 1) mark “List with single element”;

    E oldItems[] = (E[])items.clone();

    if(arrayContains(items, 0, items.length, o))
    {
      branch ObjectInList;
      return    thrown == null
             && objectsAreEqual(oldItems[indexOf], o)
             && !arrayContains(oldItems, 0, indexOf, o)
             && items.length == oldItems.length
             && equalArrays(items, 0, oldItems, 0, items.length);
    }
    else
    {
      branch NoObjectInList;
      return    thrown  == null
             && indexOf == -1
             && items.length == oldItems.length
             && equalArrays(items, 0, oldItems, 0, items.length);
    }
  }
}

public specification E remove(int i)
  throws IndexOutOfBoundsException
{
  post
  {
    if     (items.length == 0) mark “Empty list”;
    else if(items.length == 1) mark “List with single element”;

    E oldItems[] = (E[])items.clone();
```

```
        if(i < 0 || i >= items.length)
        {
          branch ExceptionalCase;
          return    thrown != null
                 && thrown instanceof IndexOutOfBoundsException
                 && items.length == oldItems.length
                 && equalArrays(items, 0, oldItems, 0, items.length);
        }
        else
        {
          branch NormalCase;
          return    thrown == null
                 && remove == oldItems[i]
                 && items.length == oldItems.length - 1
                 && equalArrays(items, 0, oldItems, 0, i)
                 && equalArrays(items, i, oldItems, i+1, items.length-i);
        }
      }
    }
  }
```

Набор возможных помеченных путей в этой спецификации выглядит так.

add()	indexOf()	remove()
Empty list ExceptionalCase	Empty list ObjectInList	Empty list ExceptionalCase
Empty list NormalCase	Empty list NoObjectInList	Empty list NormalCase
ExceptionalCase	List with single element ObjectInList	List with single element ExceptionalCase
	List with single element NoObjectInList	List with single element NormalCase
NormalCase	ObjectInList	ExceptionalCase
	NoObjectInList	NormalCase

Ситуации, выделенные серым цветом в таблице, недостижимы — пустой список не может содержать элементов и из пустого списка нельзя удалить элемент по неотрицательному индексу, меньшему длины списка.

Вторая ситуации автоматически будет отброшена инструментом, поскольку она описывается невозможной комбинацией равенств и неравенств между целыми числами: `items.length == 0`, `!(i < 0)`, `!(i >= items.length)`.

Первая же ситуация описывается комбинацией условий, связь между которыми инструменту, вообще говоря, непонятна: `items.length == 0`, `arrayContains(items, 0, items.length, o)`. Поэтому, чтобы выбросить ее из обрабатываемого инструментом множества ситуаций, надо написать в спецификации *тавтологию*, явно указывающую связь между входящими в эту комбинацию условиями. После этого постусловие метода `indexOf()` выглядит так.

```
      post
      {
        if     (items.length == 0) mark “Empty list”;
        else if(items.length == 1) mark “List with single element”;

        E oldItems[] = (E[])items.clone();

        tautology items.length == 0 =>
          !arrayContains(items, 0, items.length, o);
        ...
      }
```

Для еще более детальной оценки полноты тестирования может использоваться критерий покрытия коротких дизъюнктов, составленных из условий, встречающихся в условиях

операторов ветвления или участвующих в определении ветви оператора выбора до одного из операторов **`branch`**.

Литература

Тестирование на основе моделей

В. В. Кулямин

Лекция 9. Основы технологии разработки тестов UniTESK. Продолжение.

Эта лекция продолжает рассказ о методе разработки тестов на основе моделей, являющемся основой технологии UniTESK.

Используемый метод построения тестов включает в себя следующие виды деятельности.

1. Определение целей и рамок проекта.
2. Определение и анализ требований к тестируемой системе.
3. Определение и анализ требований к полноте тестирования.
4. Разработка и выполнение тестов.
5. Анализ результатов тестирования.

Пункты с первого по третий были рассмотрены в предыдущей лекции.

Техники построения тестов

После построения модели поведения тестируемой системы (или ее компонента) и определения критериев полноты будущих тестов можно приступать к их разработке. При разработке тестов в рамках рассматриваемой технологии используются следующие техники.

- Нацеленное построение тестов.
  Иногда самый простой и эффективный способ построить тест — это вручную задать все используемые в нем данные, указать необходимую последовательность вызовов и оформить его в виде программы, обращающейся к модели поведения тестируемого компонента для проверки корректности его работы в описанном сценарии.
- Построение тестовых данных на основе комбинирования готовых процедур инициализации и финализации для некоторых типов данных.
  Иногда можно для построения объекта заданного типа использовать во всех тестах одну и ту же процедуру. Она может быть дополнена процедурой, осуществляющей финализацию этого объекта (освобождение всех используемых им ресурсов). Для объектов некоторых, достаточно простых типов эти процедуры можно написать заранее. Строить объекты сложных типов в этом случае можно из их составных частей, т.е. инициализация такого объекта будет комбинацией инициализаций всех его полей, а финализация — комбинацией их финализаций.
- Иерархическое построение сложных тестовых данных на основе их более простых элементов с использованием фильтрации и различных техник комбинирования.
  Если необходимо построить сложный объект, например, документ определенной структуры, текст программы на некотором языке, просто объект, имеющий несколько составных частей, можно эти части построить отдельно, а затем скомбинировать. При этом для построения частей используется такая же процедура, пока не станет нужно строить данные простых типов — целые числа, символы. Числа с плавающей точкой и строки иногда можно рассматривать как простые типы данных — если их внутренние элементы незначительно влияют на поведение тестируемых операций, — а иначе они могут также строиться из более простых составляющих.
  Например, пусть необходимо построить объект, представляющий выражение, построенное по следующей грамматике.
  Expression ::= MultExpression | AddExpression (‘+’ | ‘-’) MultExpression ;

MultExpression ::= PrimeExpression | MultExpression (‘*’ | ‘/’) PrimeExpression ;
PrimeExpression ::= Constant | Identifier | ‘(’ Expression ‘)’ ;
Constant ::= (-)? [0-9] ([0-9])* ;
Identifier ::= [_A-Za-z] ([_A-Za-z0-9])* ;

Если мы предполагаем, что конкретные значения констант и идентификаторов не важны с точки зрения корректности работы с такими выражениями, можно сразу зафиксировать их возможные значения. Например, в тестовых данных будут использоваться константы -2 и 17 и (одна из них отрицательна, что покрывает опциональный знак, а вторая состоит из нескольких цифр, что покрывает минимальным образом возможное раскрытие списка в правиле для констант) и идентификаторы x и Id. Можно оформить это решение в виде двух коллекций — первая содержит два значения констант, вторая — значения идентификаторов. Далее, можно написать программу-генератор тестовых данных, в которой есть генератор примитивных выражений (констант, идентификаторов или выражений в скобках), который использует две описанные коллекции и генератор выражений в целом (уменьшая при этом возможную глубину раскрытия на один).

Генератор выражений с использованием операций * и / должен использовать генератор примитивных выражений и сам себя (с уменьшением глубины) и, в случае раскрытия выражения по второй альтернативе, вставлять по некоторому правилу знаки операции. Аналогично можно организовать и генератор выражений в целом.

При необходимости комбинировать значения, сгенерированные двумя или более генераторами более низкого уровня, можно использовать разные техники комбинирования: все возможные получаемые комбинации, комбинации всех пар значений или просто сочетания очередных выдаваемых значений.

Если, например, нужно, чтобы все используемые константы были простыми числами, такой генератор констант можно организовать на основе простого генератора всех натуральных чисел и фильтра, отсеивающего те из них, которые не просты. Фильтры можно использовать на любом уровне построенной иерархии генераторов, например, чтобы оставлять только такие выражения, в которых перемножаются значения, имеющие разные знаки.

- Построение тестовых данных с использованием разрешения ограничений.
  Часто для достижения нужной ситуации необходимо построить тестовые данные, удовлетворяющие некоторому набору условий. Например, построить два неравных положительных 32-битных целых числа, суммирование которых вызывает переполнение, т.е. сложение их дает число, большее 2^{31}-1. Для этого можно использовать различные техники разрешения ограничений.

  Самая простая из возможных техник — использование заранее приготовленных множеств возможных значений и их перебор с фильтрацией, отсеивающей значения, не удовлетворяющие заданным ограничениям. Эта техника работает, если точно известно, что среди заранее заготовленных значений найдутся подходящие. Кроме того, для построения большого набора данных или при сложных ограничениях, такая техника не слишком эффективна.

  Другой способ — использовать имеющиеся алгоритмы разрешения ограничений. Например, если набор ограничений сводится к системе линейных уравнений и неравенств над действительными числами, можно для ее решения использовать симплекс-метод. Для нескольких специфических типов задач такого вида существуют готовые алгоритмы и инструменты. Для общих систем ограничений можно использовать инструменты логического программирования.

- Комбинаторное построение тестовых последовательностей.
  При отсутствии предусловий для тестируемых операций и наличии внутреннего состояния тестовые последовательности можно строить, комбинируя вызовы

операций в различные цепочки. При этом можно использовать все возможные цепочки какой-то длины (обычно, 2-3) или более длинные последовательности де Бройна.

- Нацеленное построение тестовых последовательностей.
  При необходимости автоматически построить тестовые последовательности так, чтобы покрыть специфические ситуации можно использовать следующие техники. Можно задать метрику близости к необходимой ситуации и генерировать нужные последовательности при помощи генетических алгоритмов, оставляя «в живых» те, которые наиболее близко подходят к заданной цели.
  Можно преобразовать нужную ситуацию в набор ограничений на состояние компонента и параметры операций, а каждую из возможных операций описать в виде трансформации состояния в зависимости от ее параметров и пытаться разрешить получаемый общий набор ограничений.
- Построение тестовой последовательности как пути по автоматной модели тестируемого компонента.
  Другой способ построения тестовых последовательностей при необходимости тестировать различные взаимодействия между операциями — описать поведение тестируемого компонента с помощью некоторого конечного автомата и строить на нем различные пути. Такие пути могут проходить по всем состояниям автомата, по всем его переходам или покрывать более сложные элементы, например, все пары или тройки смежных переходов, все простые (нециклические) пути в автомате и пр.
- Динамическая генерация обхода неявно заданного автомата.
  При описании конечного автомата большие усилия нужно тратить на аккуратное описание всех переходов. Однако можно строить обход и по более простому, неявному описанию автомата, в котором задается лишь процедура вычисления текущего состояния и для каждого состояния — набор действий (стимулов), которые в нем можно выполнить. При этом алгоритм обхода начинает работать, не зная более ничего, и в ходе работы по запоминаемым состояниям и выполняемым действиям строит полный граф переходов автомата.
- Поддержка синхронизации между модельным и реальным состоянием тестируемого компонента.
  Контрактные спецификации описывают, как проверять правильность результатов обращения к одной операции. Если операции вызываются последовательно, необходимо поддерживать текущее состояние модели поведения в соответствии с реальным состоянием проверяемого компонента. Тогда можно будет проверять корректность результатов работы очередного вызова, опираясь на его аргументы и текущее модельное состояние.
  Чтобы реализовать это, используются две основные техники: либо очередное модельное состояние строится по некоторым данным о реальном состоянии компонента, которые можно достоверно узнать, либо оно экстраполируется на основе модельного состояния до вызова операции, аргументов вызова, полученных результатов и предположений о работе системы. Во втором случае ошибки, связанные с некорректным изменением состояния обнаруживаются не сразу, а иногда после целой серии других вызовов, которая приводит к неправильным с точки зрения спецификаций результатам. Иногда используется некоторая комбинация этих двух техник — часть состояния строится по достоверным данным о реальном состоянии, а другая часть экстраполируется на основе всей известной в модели информации и этих достоверных данных.
- Использование семантики чередования для проверки корректности поведения компонента в ответ на множество параллельных воздействий.

Как уже говорилось, контрактные спецификации описывают, как проверять правильность результатов обращения к одной операции. При необходимости тестировать работу системы в ответ на множество параллельных вызовов операций встает вопрос не только об очередном состоянии компонента, но и том, как определять корректность полученных результатов.

Для этого можно использовать так называемую семантику чередования — результаты работы набора параллельных вызовов считаются корректными, если эти вызовы можно так упорядочить, что в полученной цепочке буду выполняться все пред- и постусловия соответствующих операций.

При использовании такого подхода необходимо считать каждую операцию атомарным действием, которое выполняется тестируемой системой без возможности вмешательства остальных. Если это не так, то нужно выделить такие атомарные действия и указывать их в модели в качестве возможных воздействий на систему и ее реакций. При этом может оказаться, что возвращение операцией управления нужно описать как отдельное действие, поскольку ее результат может зависеть от того, какие другие операции были вызваны во время ее работы.

Разработка и выполнение тестов

Разработка и выполнение тестов объединены в одну деятельность, поскольку собственно разработка тестов обычно происходит вперемешку с их прогонами и отладкой. При отладке тестов устраняются все обнаруживаемые ошибки в модели поведения и самих тестах, остаются только те, источник которых — некорректное поведение тестируемой системы.

Деятельность по разработке тестов может быть разделена на следующие действия.

1. **Выбор общей схемы теста для данного компонента (группы компонентов)**. Сначала на основе того, что известно о поведении компонентов и требованиях к полноте их тестирования нужно определить виды тестов, которые необходимо для них разработать. Это могут оказаться простейшие тесты, которые проверяют только, что вызываемая один раз операция возвращает результат, как-то похожий на правильный. Это могут быть тесты, проверяющие работу каждой операции в нескольких ситуациях, соответствующих различным вариантам ее использования или разным ветвям функциональности. Это могут быть более сложные тесты, проверяющие корректность работы группы операций с общим состоянием с помощью обхода автомата, моделирующего поведение этой группы операций.

   Еще более сложные тесты могут проверять поведение данной группы операций в зависимости от других операций, которые могут изменять общее состояние первых, или проверять корректность параллельной работы всех пар или троек операций из заданной группы.

   С точки зрения построения схемы теста также важно, какие операции (может быть одна) в нем будут участвовать, какого рода ситуации в этом тесте буду создаваться, будет ли использоваться внутреннее состояние тестируемого компонента, будут ли нужны нетривиальные тестовые данные, будет ли нужен параллелизм. Схема выбирается в соответствии с некоторой подходящей комбинацией перечисленных выше техник.

2. **Определение дополнительных проверок (если нужно)**.

   Иногда не все проверки корректности поведения стоит оформлять в спецификациях отдельных операций, поскольку для их выполнения необходимо знать полную историю обращений к системе или значительную ее часть. В этом случае то, что можно проверить в рамках вызова одной операции (на основе текущего состояния и аргументов вызова), проверяется в ее постусловии, а те проверки, для которых необходимо знать больше об истории обращений, записываются в тестовом сценарии.

Например, для тестирования генератора случайных чисел можно написать спецификацию, в которой проверять только принадлежность результата заданному интервалу. А в рамках сценария можно собирать все получаемые результаты и проверять общие ограничения на их распределение, используя, например, критерия Колмогорова.

3. **Разработка генераторов тестовых данных (если нужно).**
   Если необходимо использовать нетривиальные тестовые данные, для их построения разрабатывается набор генераторов с помощью подходящей комбинации техник, описанных в предыдущем разделе.
4. **Определение состояний и действий, построение автомата теста (если нужно)**.
   Для тестирования компонентов, поведение которых зависит от внутреннего состояния, тесты обычно разрабатываются с использованием автоматной модели такого компонента по одной из техник, описанных выше.
5. **Разработка адаптеров (если нужно)**.
   Если интерфейс тестируемой системы несколько отличается от того интерфейса ее модели поведения, для преобразования вызовов между этими двумя интерфейсами используются тестовые адаптеры.
   Тестовые адаптеры бывают разных видов, в зависимости от двух факторов: в какую сторону они выполняют преобразование (из модели в реализацию или обратно), и какого рода связи с моделью и с реализацией (тестируемой системой) они используют.
   Модельно-реализационные адаптеры выполняют преобразование обращений из модельного интерфейса в реализационный, а результаты операций или асинхронного возникающие события преобразую обратно — из реализационного вида в модельный.
   Реализационно-модельные адаптеры производят все преобразования только из реализационного вида в модельный.
   Модельно-реализационный адаптер может оформляться в виде класса, наследующего модельный класс и использующего классы тестируемой системы. Адаптеры, переводящие события из реализации в модель, наоборот, удобнее делать реализующими какие-то интерфейсы тестируемой системы и использующими модельные классы для сохранения информации о передаваемых ими событиях.
   Например, модельно-реализационный адаптер, реализующий описанный в предыдущей лекции класс спецификации списка через java.util.Vector может выглядеть примерно так.

```
public mediator class VectorAdapter<E>
  implements ListSpecification<E>
{
  implementation Vector target;

  public mediator void add(int i, E o)
    throws IndexOutOfBoundsException
  {
    implementation { target.insertElementAt(o, i); }
```

```
    update
    {
      E[] newItems = new E[items.length + 1];
      System.arraycopy(items, 0, newItems, 0, i);
      newItems[i] = o;
      System.arraycopy(items, i, newItems, i+1, items.length-i+1);
      items = newItems;
    }
  }
  ...
}
```

В приведенном примере в блоке update выполняется экстраполяция нового состояния модели в текущей ситуации. Если же, например, можно использовать результаты методов size и getElementAt как достоверные сведения о текущем состоянии вектора, можно вместо блоков update в каждом из методов-адаптеров написать один общий блок update для всего класса.

```
public mediator class VectorAdapter<E>
  implements ListSpecification<E>
{
  ...
  update
  {
    if(target == null) items = new E[];
    else
    {
      items = new E[target.size()];
      for(int i = 0; i < target.size(); i++)
        items[i] = target.getElementAt(i);
    }
  }
  ...
}
```

6. **Прогоны и отладка тестов**.
   Ну, и наконец, в рамках этого этапа необходимо оформить тесты и отладить их.

Разберем построение автоматного теста для списка, спецификации которого описаны в предыдущей лекции. Напомним, что для его тестирования выбран критерий полноты, требующий проверить работу списка в нормальных и исключительных ситуациях для методов add и remove, для метода indexOf — при наличии аргумента в списке и при его отсутствии. Кроме того, для всех методов нужно проверить их работу для пустого списка, а для методов remove и indexOf — для списка, содержащего только один элемент.

При построении автоматного теста, нацеленного на достижения заданного критерия полноты тестирования, используется *редукция модели по критерию полноты*. При применении этой техники по набору контрактов и критерию полноты строится автоматная модель, проход по всем переходам которой гарантирует достижение заданного критерия.

Такая редукция выполняется следующим образом.

1. Выделение целей тестирования.
   На первом шаге для каждой операции перечисляются выделяемые критерием полноты тестирования ситуации — это и есть цели тестирвоания.
   В нашем случае перечень этих ситуаций такой.
   - Метод add.
     - пустой список и корректный индекс;
     - пустой список и некорректный индекс;
     - непустой список и корректный индекс;
     - непустой список и некорректный индекс.

- Метод remove.
  - пустой список;
  - список с одним элементом и корректный индекс;
  - список с одним элементом и некорректный индекс;
  - список с двумя или более элементами и корректный индекс;
  - список с двумя или более элементами и некорректный индекс.
- Метод indexOf
  - пустой список;
  - список с одним элементом и аргумент в списке;
  - список с одним элементом и аргумента нет в списке;
  - список с двумя или более элементами и аргумент в списке;
  - список с двумя или более элементами и аргумента нет в списке.

Эти ситуации представляются как ограничения, вырезающие из области определения операции подобласти в пространстве возможных состояний системы и аргументов операции.

2. Получение обобщенных состояний и действий.
   Далее все области, соответствующие различным ситуациям, проецируются на пространство состояний. Рассматриваются все возможные пересечения подмножеств этих проекций. Полученные в итоге множества состояний для каждой проекции должны либо входить в нее, либо не пересекаться с ней. Эти множества образуют разбиение всех состояний на классы эквивалентности.

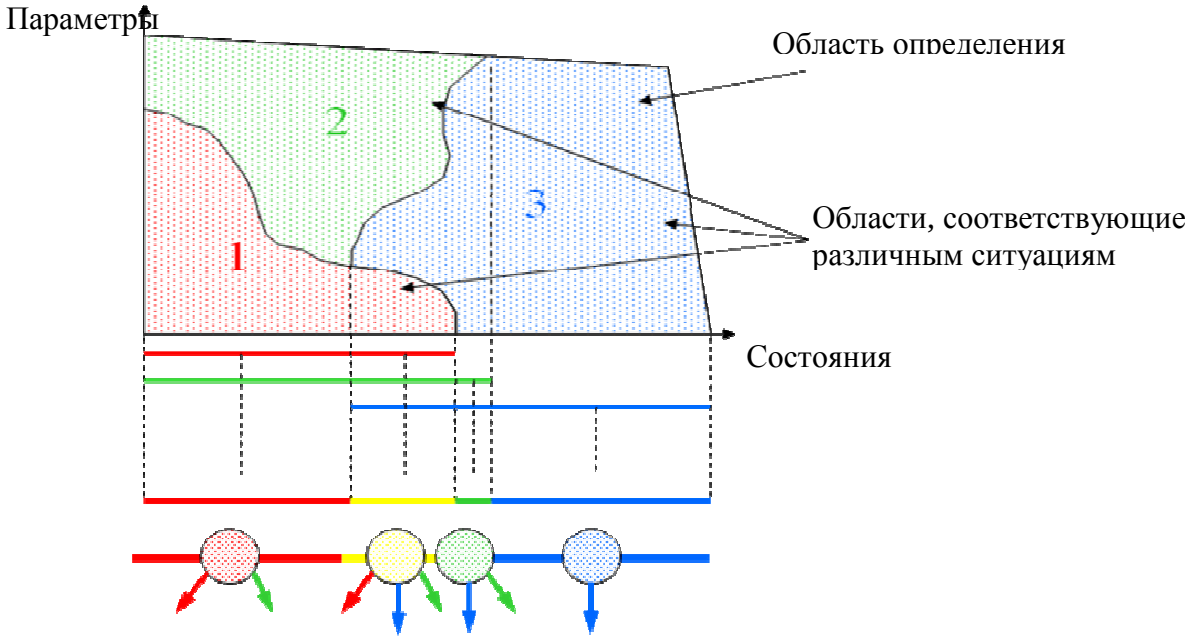


Рисунок 1. Построение обобщенных состояний и действий.

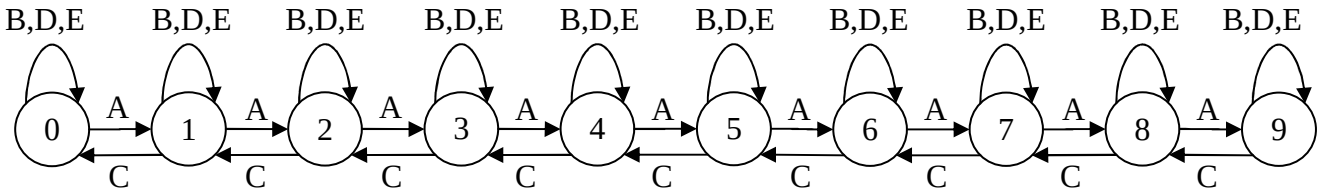
Полученные так множества состояний — обобщенные состояния — являются кандидатами на роль состояний автомата, моделирующего систему. Действия в этом автомате соответствуют всем возможным ситуациям. Проход по всем его переходам будет означать, что все исходные ситуации проверены.

В нашем примере такими пересечениями проекций ситуаций являются следующие обобщенные состояния.

- Пустой список.
- Список с одним элементом (произвольным).

- Список с двумя или более элементами.

3. Детерминизация полученного автомата.
   Полученный автомат может оказаться непригодным для контролируемого тестирования из-за высокой степени недетерминизма. Если он детерминирован — каждое действие в каждом обобщенном состоянии приводит в однозначно определяемое обобщенное состояние, то все хорошо. Если же это не так, что чтобы сделать его детерминированным, нужно расщепить те состояния, где есть недетерминированные действия. Каждый раз, когда у действия в некотором обобщенном состоянии может быть несколько возможных исходов (конечных состояний), мы разделяем это состояние на такие части, чтобы в каждой из этих частей исход действия определялся однозначно.
   Выполнение этого расщепления не всегда приводит в итоге к детерминированному автомату. В таком случае можно попробовать разделить операции на несколько групп, оставив в каждой из этих групп по одной из операций, порождающих недетерминированные действия. При таком делении часто можно построить несколько различных детерминированных автоматов (каждый соответствует только части операций, одна операция может использоваться в нескольких автоматах), обход каждого из которых дает желаемое покрытие всех ситуаций.
   В нашем примере источник недетерминизма один — при выполнении remove в списке с двумя или более элементами мы можем остаться в том же обобщенном состоянии (список будет иметь два или более элемента), а можем получить список с одним элементом.
   Чтобы удалить недетерминизм, нужно выделить в отдельное обобщенное состояние списки с двумя элементами. В оставшемся множестве — списки с тремя или более элементами — все равно останется тот же недетерминизм. Чтобы его удалить совсем, придется различать списки с различным числом элементов. Заметьте, что в полученном автомате никак не учитывается. Какие именно элементы находятся в списке — важно только их число.
   Действия в этом автомате соответствуют исходным ситуациям (аналогичные действия в различных состояниях можно обозначать одним способом).
   - Добавить элемент с корректным индексом.
   - Добавить элемент с некорректным индексом.
   - Удалить элемент с корректным индексом.
   - Удалить элемент с некорректным индексом.
   - Найти индекс первого вхождения элемента, отсутствующего в списке.
   - Найти индекс первого вхождения элемента, присутствующего в списке.
4. Ограничение полученного автомата.
   После предыдущего шага автомат может оказаться бесконечным. Тестирование является конечной процедурой, и для построения тестов нам достаточно какой-то конечной части этого автомата. Поскольку исходных целей тестирования конечное число, они достигаются при проходе по всем переходам в некотором конечном подавтомате. Чтобы выделить его, достаточно в некоторых состояния запретить действия, «уводящие дальше от начального состояния».
   Обычно такие пороговые состояния можно определить небольшим набором параметров. Эти параметры можно сделать параметрами теста, чтобы по одному и тому же описания можно было строить тесты различной сложности.
   В примере тестирования списка «уводящей от начального состояния» операцией является добавление элементов. Достаточно запретить его при достижении некоторого максимально возможного в тесте количества элементов N. Чтобы покрыть все исходные ситуации, достаточно, чтобы N было не меньше двух.



A — `add(), Normal case`
B — `add(), Exceptional case`
C — `remove(), Normal case`
D — `remove(), Exceptional case`
E — `indexOf(), все случаи`

Рисунок 2. Граф состояний автомата, моделирующего список, для значения параметра 9.

Осталось оформить неявное описание такого автомата в виде тестового сценария.

```
public scenario class ListTest
{
  ListSpecification<Object> list = mediator VectorAdapter<Object>
    (target = new Vector());  // инициализируем используемый адаптер

  int maxSize = 9;
  Object[] objPool = new Object[] { new Object(), null };

  public static void main(String[] args) // метод запуска теста
  {
    ListTest test = new ListTest();

    // устанавливаем используемый алгоритм обхода
    test.setExplorer(new jatva.exploration.DFSExplorer());

    test.run(); // выполняем тест
  }

  // блок, задающий вычисление текущего состояния
  state { return list.items.length; }

  scenario add()
  {
    if(list.items.length < maxSize) // добавляем элементы не всегда
    {
      iterate(int i = 0; i < maxSize; i++)
        iterate(Object o : objPool)
          list.add(i, o);
    }
  }

  scenario remove()
  {
    iterate(int i = 0; i < maxSize; i++)
      list.remove(i);
  }

  scenario indexOf()
  {
    iterate(Object o : objPool)
      list.indexOf(o);
  }
}
```

Анализ результатов тестирования

При анализе результатов необходимо выполнить следующие действия.

1. **Анализ обнаруженных ошибок**.
   Отчеты по результатам тестирования содержат список обнаруженных нарушений. Нарушение может иметь различную природу.

   - Это может быть ошибка в спецификациях, адаптерах или сценариях, проявляющаяся как исключительная ситуация.
     Наличие такой ошибки, означает, что спецификации написаны неправильно. Может быть, это просто описка или ошибка в коде спецификаций. Может быть, их автор рассчитывал на выполнение определенных ограничений, которые не соблюдаются. Стоит проанализировать эти ограничения — быть может, это пропущенное условие, которое нужно проверять или инвариант. В любом случае нужно внести поправки в спецификацию, хотя причиной подобной ситуации может быть неожиданное поведение тестируемой системы.
   - Другой вид ошибок — нарушения в структуре автомата теста.
     Например, он может оказаться слишком большим, и после некоторого числа переходов или состояний алгоритм обхода сообщит об этом, он неожиданно оказаться недетерминированным или несвязным — алгоритм обхода может не найти способа вернуться в некоторое обобщенное состояние, где он раньше уже был.
     Причиной таких ошибок чаще всего является неправильное построение теста — неправильно проведена детерминизация или ограничение слишком слабое (может быть, наоборот, нужно увеличить максимально возможное число переходов), забыты или неправильно определены какие-то действия.
     Гораздо реже причиной такой ошибки может стать ошибка в тестируемой системе. В этом случае, скорее всего, забыты какие-то проверки, которые помогли бы выявить ее раньше.
   - Нарушение предусловия.
     Оно означает, что нужно исправить тест, чтобы данная операция не вызывалась с некорректными аргументами.
   - Несоответствие между наблюдаемым поведением тестируемой системы и ее моделью поведения.
     Такие несоответствия проявляются как нарушения постусловий и инвариантов, невыполнение дополнительных проверяемых ограничений в сценариях, неожиданные исключения в тестируемой системе.
     В этом случае ошибка может быть как в спецификациях, адаптерах или тестах, так и в тестируемой системе. Чтобы выяснить это, нужно проанализировать создавшуюся ситуацию, еще раз проверить нарушенные ограничения — действительно они должны быть выполнены в этом случае и т.д. Часто такой анализ приводит к ошибкам не в тестируемой системе, а в тестах или моделях.
     Только в том случае, если это действительно ошибка тестируемой системы, в тесты не нужно вносить исправлений.

2. **Анализ тестового покрытия.**
   Отчеты анализируются и на предмет достигнутого покрытия, наличия непокрытых ситуаций и причин этого. Если достигнута желаемая полнота тестирования, разработку тестов можно прекратить. Если же непокрыты какие-то важные

ситуации или общее покрытие недостаточно, нужно поправить имеющиеся тесты или разработать новые, нацеленные на еще не покрытые ситуации.

Литература