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*;

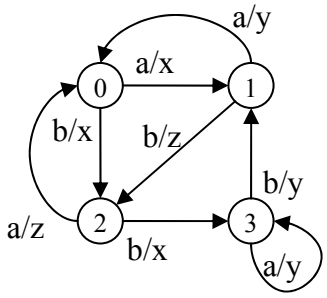
I — конечное множество, элементы которого называются *входными символами*, *входами* или *стимулами*, само I называют *входным алфавитом* автомата;

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*;

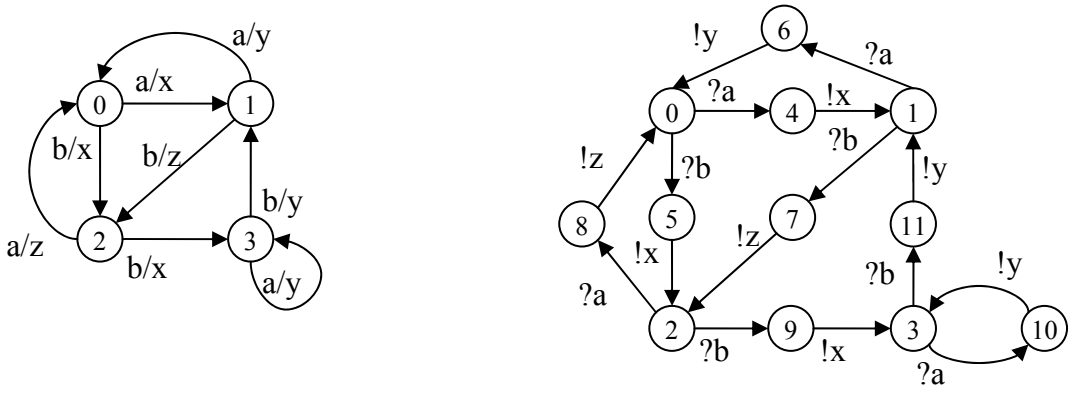
$T \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ — множество *переходов* системы. Каждый переход — тройка (s_1, a, s_2) — *выходит из* s_1, *ведет в* s_2, и помечен действием a, которое может быть стимулом, реакцией или внутренним действием. Этот переход изображают стрелкой, ведущей из s_1 и 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;

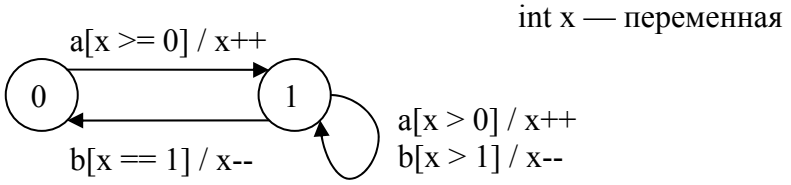
O — конечное множество, элементы которого называются *реакциями*, само O называют *выходным алфавитом* автомата;

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

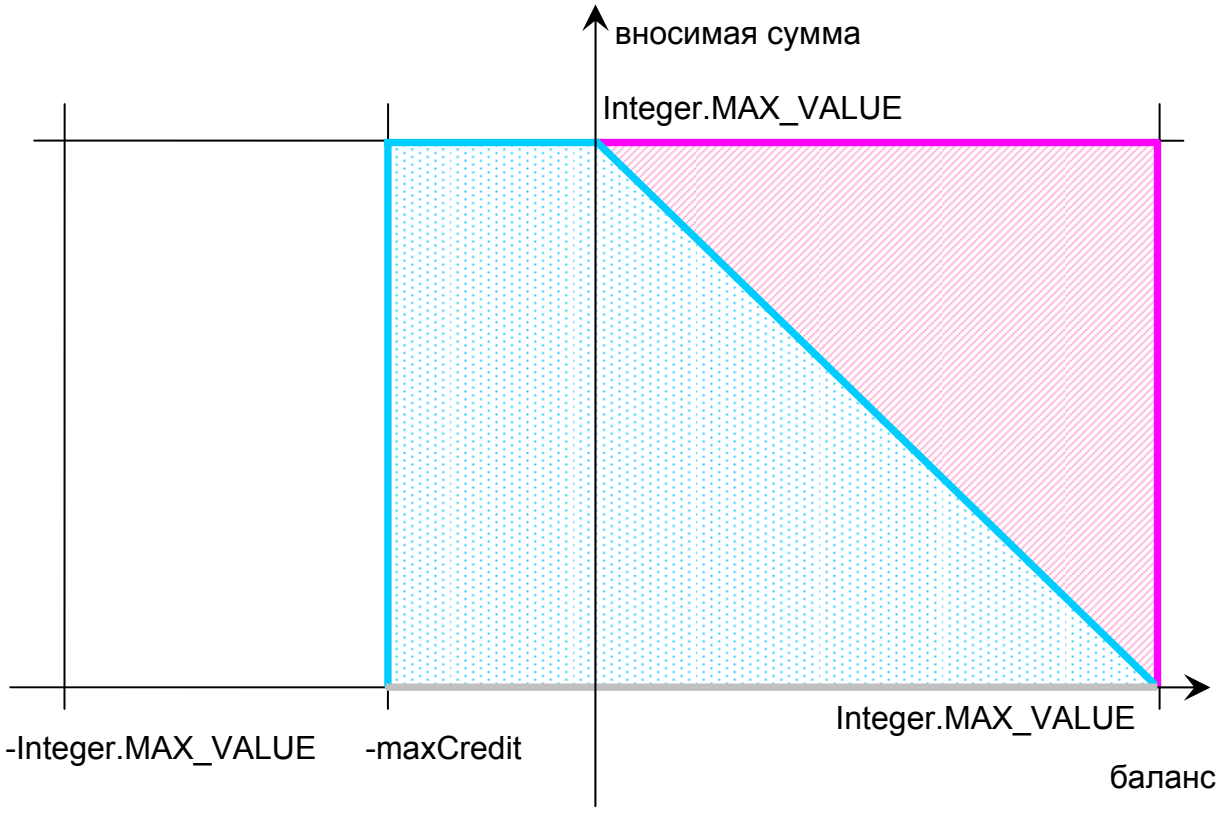
содержащих ее кривых-границах, входящих в область, на всех поверхностях, входящих в область, чьи границы не все входят в нее, и в самой области, если есть ее поверхность-граница, проходящая через эту точку и не входящая в область.

Пример.

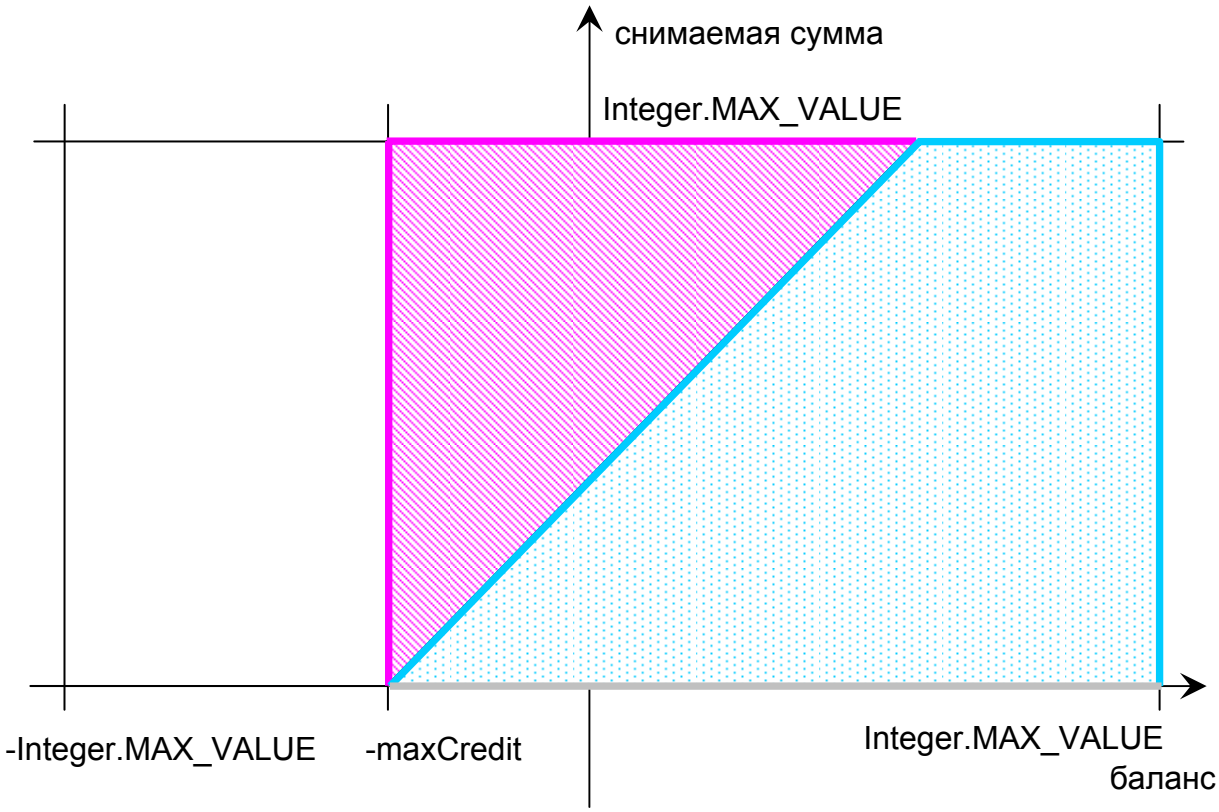
Пусть тестируется реализация банковского счета с ограниченным кредитом. В тестируемом компоненте две операции — положить деньги на счет и снять деньги со счета. Пусть снимаемая/заносимая сумма и текущий баланс счета представлены 32-битными целыми числами.

- 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.

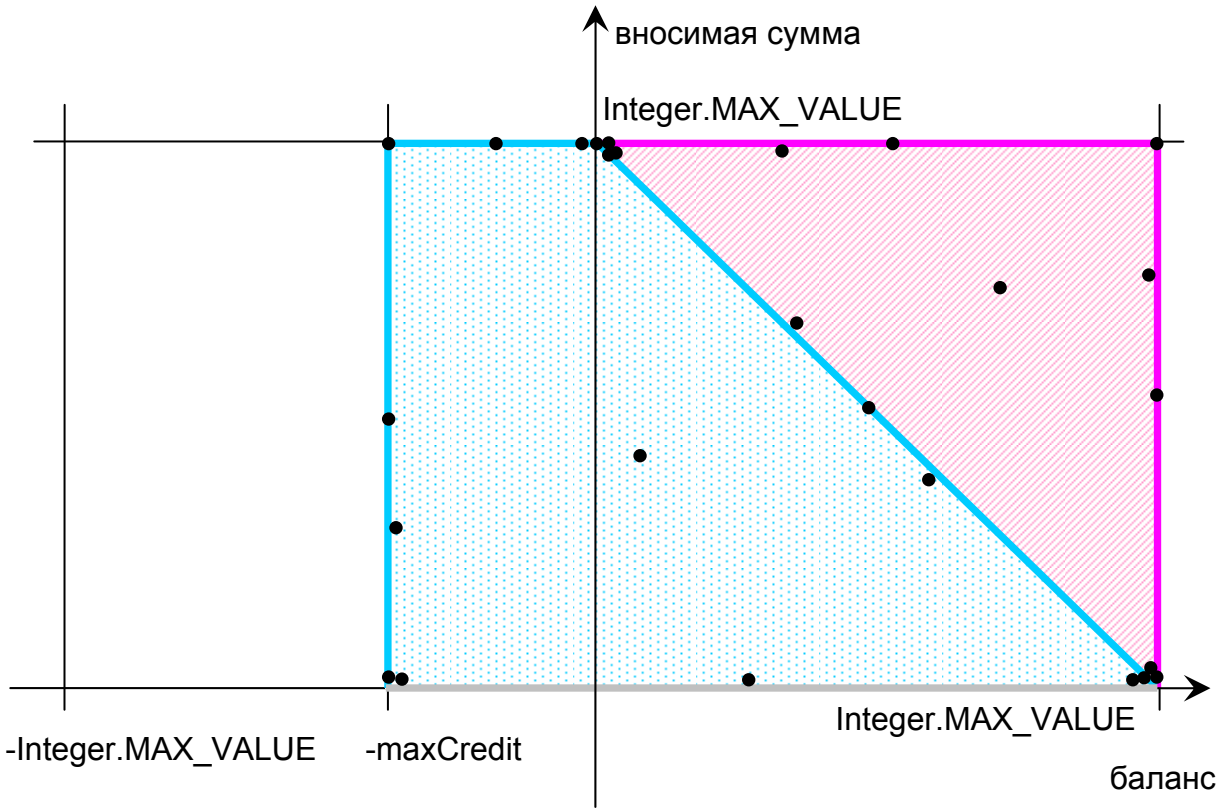


Разбиение области определения для операции внесения денег на счет. Граница голубого или красного цвета означает, что она включается в голубую или, соответственно, красную область



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.

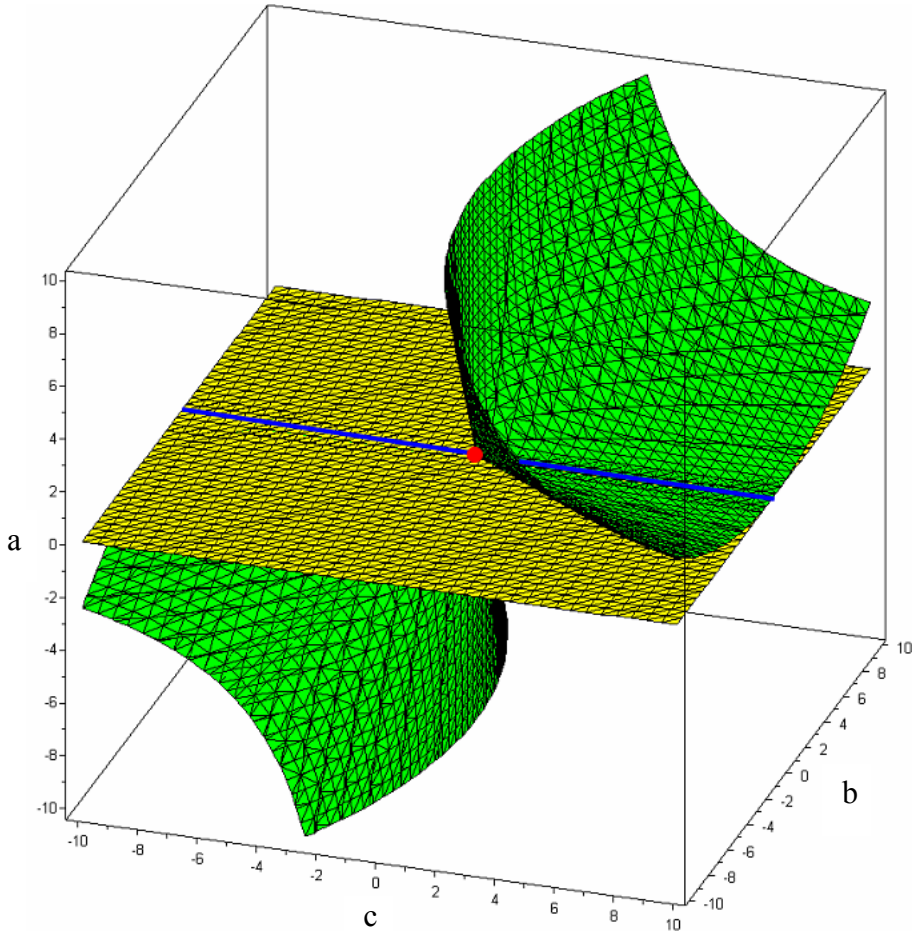


Рисунок 1. Подобласти параметров для программы, решающей квадратные уравнения.

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