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.

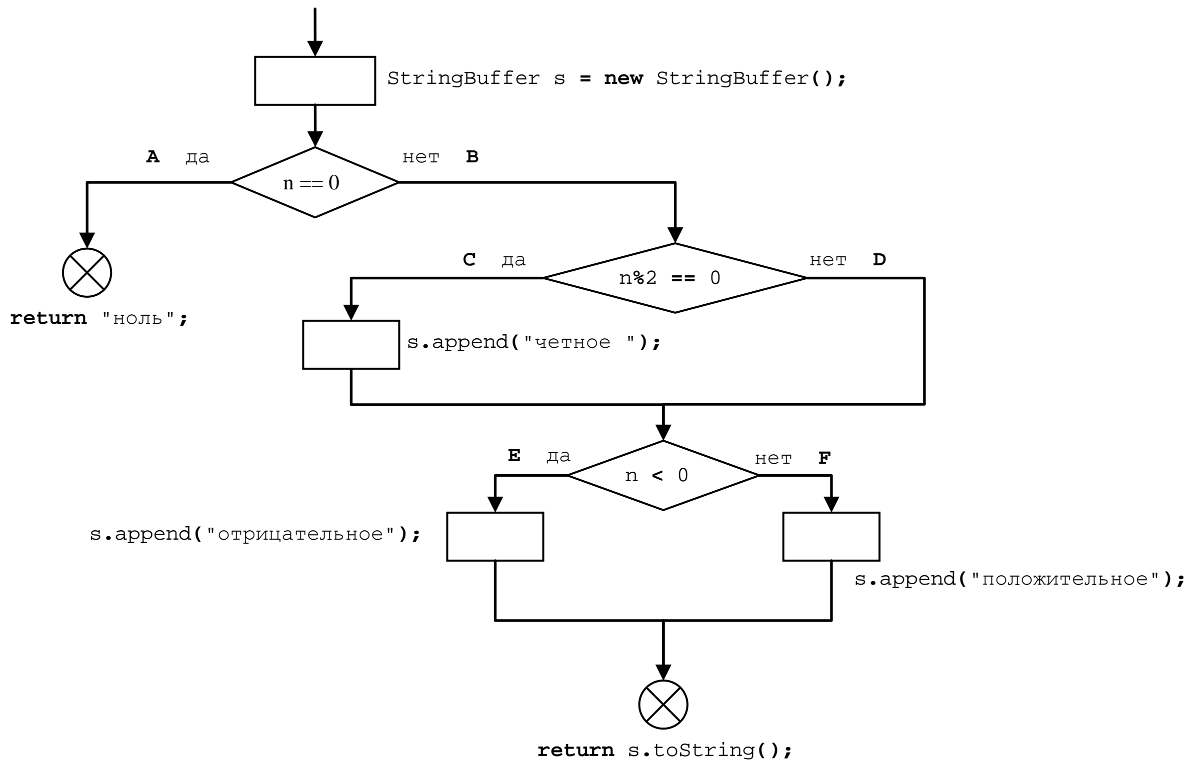
```
 1   String classifier(int n)
 2   {
 3     StringBuffer s = new StringBuffer();
 4
 5     if(n == 0)
 6       return "ноль";
 7
 8     if(n%2 == 0)
 9       s.append("четное ");
10
11     if(n < 0)
12       s.append("отрицательное");
13     else
14       s.append("положительное");
```

```
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.

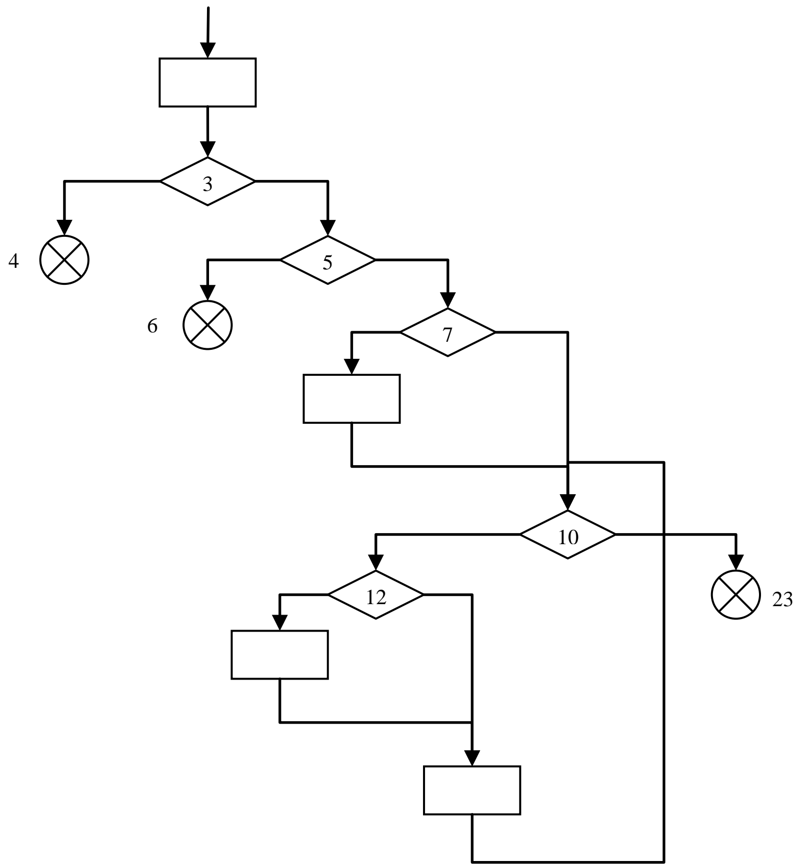
В рассматриваемом примере имеется шесть ветвей — на схеме графа они помечены латинскими буквами. Для каждой ветви, кроме D, есть соответствующий набор инструкций, который выполняется, когда покрывается эта ветвь. В нашем случае концевая вершина ветви D соответствует условному оператору, который выполнится 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.

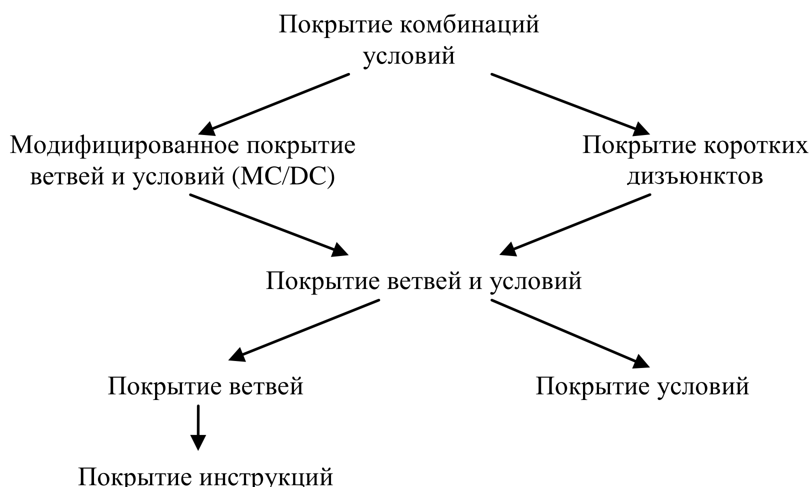
Элементарные условия, входящие в программу, выстраиваются в некоторой последовательности. Сначала всем элементарным формулам приписываются значения 0.

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.

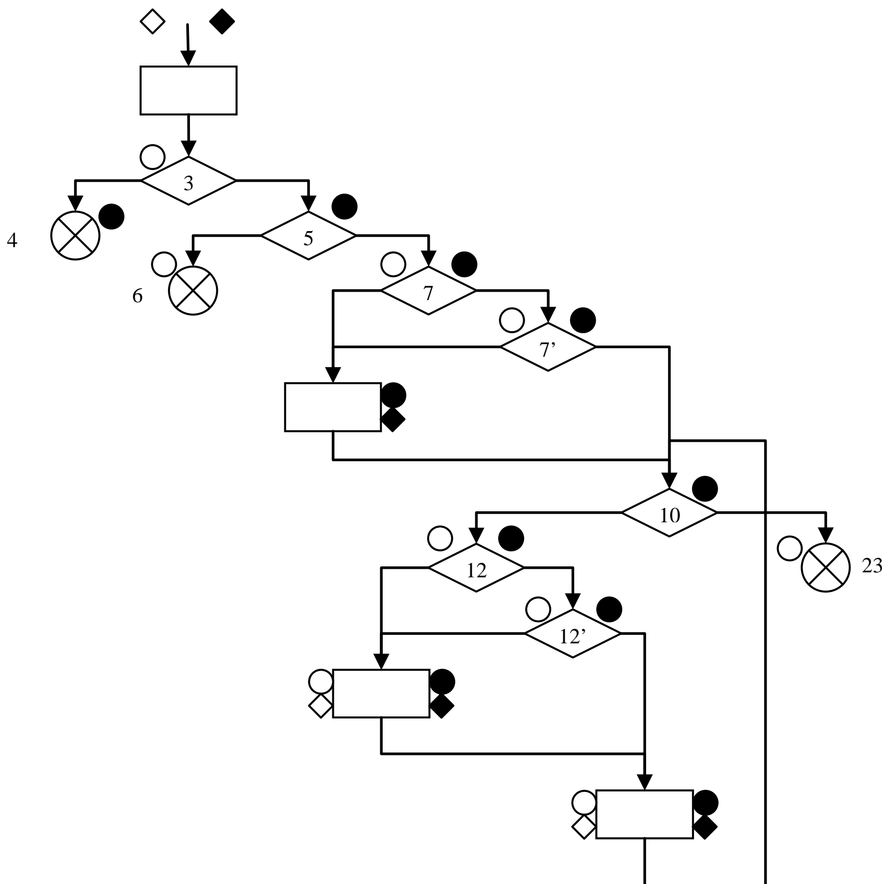
Инструкции использования переменной могут входить в du-путь много раз.

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.

<u>Замечание</u>. *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 — (<u>n</u>). 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-(<u>7’</u>)-23, 0-(7)-12, 0-(7’)-12, 0-(<u>7’</u>)-12, 0-(7)-12’, 0-(7’)-12’, 0-(<u>7’</u>)-12’, 0-(7,12)-14, 0-(7’,12)-14, 0-(<u>7’</u>, 12)-14, 0-(7,12’)-14, 0-(7’,12’)-14, 0-(<u>7’</u>, 12’)-14, 0-(7,<u>12’</u>)-20, 0-(7’,<u>12’</u>)-20, 0-(<u>7’</u>,<u>12’</u>)-20, 14-20, 20-12, 20-12’, 20-23, 20-(12)-14, 20-(12’)-14, 20-(<u>12’</u>)-20.

Для переменной b — 0-4, 0-5, 0-7, 0-7’, 0-(7)-8, 0-(7’)-8, 0-(<u>7’</u>)-10, 0-(<u>7’</u>)-12, 0-(<u>7’</u>)-12’, 0-(<u>7’</u>,12)-15, 0-(<u>7’</u>,12’)-15, 0-(<u>7’</u>,<u>12’</u>)-19, 8-10, 8-12, 8-12’, 8-(12)-15, 8-(12’)-15, 8-(<u>12’</u>)-19, 15-19, 19-10, 19-12, 19-12’, 19-(12)-15, 19-(12’)-15, 19-(<u>12’</u>)-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’-(<u>7’</u>)-10-12-12’-(<u>12’</u>)-19-20 и 20-23), <1, –1> (дает 0-7-(7)-8-10-12’-(<u>12’</u>)-19-20), <–1, 1> (дает 0-7-7’-(7’)-8-10-12’-(<u>12’</u>)-19-20), <1, 2> (дает 0-(<u>7’</u>)-12-(12)-14-15-19-20-10-12-12’-(<u>12’</u>)-19-20), <–1, –2> (0-(<u>7’</u>)-12-12’-(12’)-14-15-19-20-10-12-12’-(<u>12’</u>)-19-20), <1, –2> (0-7-(7)-8-10-12-(12)-14-15-19-20-10-12-12’-(<u>12’</u>)-19-20), <–1, 2> (0-7-7’-(7’)-8-10-12-12’-(12’)-14-15-19-20-10-12-12’-(<u>12’</u>)-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.

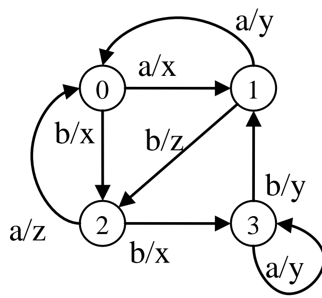
У всех видов автоматов имеются состояния и переходы. Соответственно, для всех видов автоматов можно определить следующие метрики покрытия.

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.