Тестирование на основе моделей

В. В. Кулямин

Лекция 9. Основы технологии разработки тестов UniTESK. Продолжение.

Эта лекция продолжает рассказ о методе разработки тестов на основе моделей, являющемся основой технологии UniTESK.

Используемый метод построения тестов включает в себя следующие виды деятельности.

1. Определение целей и рамок проекта.
2. Определение и анализ требований к тестируемой системе.
3. Определение и анализ требований к полноте тестирования.
4. Разработка и выполнение тестов.
5. Анализ результатов тестирования.

Пункты с первого по третий были рассмотрены в предыдущей лекции.

Техники построения тестов

После построения модели поведения тестируемой системы (или ее компонента) и определения критериев полноты будущих тестов можно приступать к их разработке. При разработке тестов в рамках рассматриваемой технологии используются следующие техники.

- Нацеленное построение тестов.
  Иногда самый простой и эффективный способ построить тест — это вручную задать все используемые в нем данные, указать необходимую последовательность вызовов и оформить его в виде программы, обращающейся к модели поведения тестируемого компонента для проверки корректности его работы в описанном сценарии.
- Построение тестовых данных на основе комбинирования готовых процедур инициализации и финализации для некоторых типов данных.
  Иногда можно для построения объекта заданного типа использовать во всех тестах одну и ту же процедуру. Она может быть дополнена процедурой, осуществляющей финализацию этого объекта (освобождение всех используемых им ресурсов). Для объектов некоторых, достаточно простых типов эти процедуры можно написать заранее. Строить объекты сложных типов в этом случае можно из их составных частей, т.е. инициализация такого объекта будет комбинацией инициализаций всех его полей, а финализация — комбинацией их финализаций.
- Иерархическое построение сложных тестовых данных на основе их более простых элементов с использованием фильтрации и различных техник комбинирования.
  Если необходимо построить сложный объект, например, документ определенной структуры, текст программы на некотором языке, просто объект, имеющий несколько составных частей, можно эти части построить отдельно, а затем скомбинировать. При этом для построения частей используется такая же процедура, пока не станет нужно строить данные простых типов — целые числа, символы. Числа с плавающей точкой и строки иногда можно рассматривать как простые типы данных — если их внутренние элементы незначительно влияют на поведение тестируемых операций, — а иначе они могут также строиться из более простых составляющих.
  Например, пусть необходимо построить объект, представляющий выражение, построенное по следующей грамматике.
  Expression ::= MultExpression | AddExpression (‘+’ | ‘-’) MultExpression ;

MultExpression ::= PrimeExpression | MultExpression (‘*’ | ‘/’) PrimeExpression ;
PrimeExpression ::= Constant | Identifier | ‘(’ Expression ‘)’ ;
Constant ::= (-)? [0-9] ([0-9])* ;
Identifier ::= [_A-Za-z] ([_A-Za-z0-9])* ;

Если мы предполагаем, что конкретные значения констант и идентификаторов не важны с точки зрения корректности работы с такими выражениями, можно сразу зафиксировать их возможные значения. Например, в тестовых данных будут использоваться константы -2 и 17 и (одна из них отрицательна, что покрывает опциональный знак, а вторая состоит из нескольких цифр, что покрывает минимальным образом возможное раскрытие списка в правиле для констант) и идентификаторы x и Id. Можно оформить это решение в виде двух коллекций — первая содержит два значения констант, вторая — значения идентификаторов. Далее, можно написать программу-генератор тестовых данных, в которой есть генератор примитивных выражений (констант, идентификаторов или выражений в скобках), который использует две описанные коллекции и генератор выражений в целом (уменьшая при этом возможную глубину раскрытия на один).

Генератор выражений с использованием операций * и / должен использовать генератор примитивных выражений и сам себя (с уменьшением глубины) и, в случае раскрытия выражения по второй альтернативе, вставлять по некоторому правилу знаки операции. Аналогично можно организовать и генератор выражений в целом.

При необходимости комбинировать значения, сгенерированные двумя или более генераторами более низкого уровня, можно использовать разные техники комбинирования: все возможные получаемые комбинации, комбинации всех пар значений или просто сочетания очередных выдаваемых значений.

Если, например, нужно, чтобы все используемые константы были простыми числами, такой генератор констант можно организовать на основе простого генератора всех натуральных чисел и фильтра, отсеивающего те из них, которые не просты. Фильтры можно использовать на любом уровне построенной иерархии генераторов, например, чтобы оставлять только такие выражения, в которых перемножаются значения, имеющие разные знаки.

- Построение тестовых данных с использованием разрешения ограничений.
  Часто для достижения нужной ситуации необходимо построить тестовые данные, удовлетворяющие некоторому набору условий. Например, построить два неравных положительных 32-битных целых числа, суммирование которых вызывает переполнение, т.е. сложение их дает число, большее 2^{31}-1. Для этого можно использовать различные техники разрешения ограничений.

  Самая простая из возможных техник — использование заранее приготовленных множеств возможных значений и их перебор с фильтрацией, отсеивающей значения, не удовлетворяющие заданным ограничениям. Эта техника работает, если точно известно, что среди заранее заготовленных значений найдутся подходящие. Кроме того, для построения большого набора данных или при сложных ограничениях, такая техника не слишком эффективна.

  Другой способ — использовать имеющиеся алгоритмы разрешения ограничений. Например, если набор ограничений сводится к системе линейных уравнений и неравенств над действительными числами, можно для ее решения использовать симплекс-метод. Для нескольких специфических типов задач такого вида существуют готовые алгоритмы и инструменты. Для общих систем ограничений можно использовать инструменты логического программирования.

- Комбинаторное построение тестовых последовательностей.
  При отсутствии предусловий для тестируемых операций и наличии внутреннего состояния тестовые последовательности можно строить, комбинируя вызовы

операций в различные цепочки. При этом можно использовать все возможные цепочки какой-то длины (обычно, 2-3) или более длинные последовательности де Бройна.

- Нацеленное построение тестовых последовательностей.
  При необходимости автоматически построить тестовые последовательности так, чтобы покрыть специфические ситуации можно использовать следующие техники. Можно задать метрику близости к необходимой ситуации и генерировать нужные последовательности при помощи генетических алгоритмов, оставляя «в живых» те, которые наиболее близко подходят к заданной цели.
  Можно преобразовать нужную ситуацию в набор ограничений на состояние компонента и параметры операций, а каждую из возможных операций описать в виде трансформации состояния в зависимости от ее параметров и пытаться разрешить получаемый общий набор ограничений.
- Построение тестовой последовательности как пути по автоматной модели тестируемого компонента.
  Другой способ построения тестовых последовательностей при необходимости тестировать различные взаимодействия между операциями — описать поведение тестируемого компонента с помощью некоторого конечного автомата и строить на нем различные пути. Такие пути могут проходить по всем состояниям автомата, по всем его переходам или покрывать более сложные элементы, например, все пары или тройки смежных переходов, все простые (нециклические) пути в автомате и пр.
- Динамическая генерация обхода неявно заданного автомата.
  При описании конечного автомата большие усилия нужно тратить на аккуратное описание всех переходов. Однако можно строить обход и по более простому, неявному описанию автомата, в котором задается лишь процедура вычисления текущего состояния и для каждого состояния — набор действий (стимулов), которые в нем можно выполнить. При этом алгоритм обхода начинает работать, не зная более ничего, и в ходе работы по запоминаемым состояниям и выполняемым действиям строит полный граф переходов автомата.
- Поддержка синхронизации между модельным и реальным состоянием тестируемого компонента.
  Контрактные спецификации описывают, как проверять правильность результатов обращения к одной операции. Если операции вызываются последовательно, необходимо поддерживать текущее состояние модели поведения в соответствии с реальным состоянием проверяемого компонента. Тогда можно будет проверять корректность результатов работы очередного вызова, опираясь на его аргументы и текущее модельное состояние.
  Чтобы реализовать это, используются две основные техники: либо очередное модельное состояние строится по некоторым данным о реальном состоянии компонента, которые можно достоверно узнать, либо оно экстраполируется на основе модельного состояния до вызова операции, аргументов вызова, полученных результатов и предположений о работе системы. Во втором случае ошибки, связанные с некорректным изменением состояния обнаруживаются не сразу, а иногда после целой серии других вызовов, которая приводит к неправильным с точки зрения спецификаций результатам. Иногда используется некоторая комбинация этих двух техник — часть состояния строится по достоверным данным о реальном состоянии, а другая часть экстраполируется на основе всей известной в модели информации и этих достоверных данных.
- Использование семантики чередования для проверки корректности поведения компонента в ответ на множество параллельных воздействий.

Как уже говорилось, контрактные спецификации описывают, как проверять правильность результатов обращения к одной операции. При необходимости тестировать работу системы в ответ на множество параллельных вызовов операций встает вопрос не только об очередном состоянии компонента, но и том, как определять корректность полученных результатов.

Для этого можно использовать так называемую семантику чередования — результаты работы набора параллельных вызовов считаются корректными, если эти вызовы можно так упорядочить, что в полученной цепочке буду выполняться все пред- и постусловия соответствующих операций.

При использовании такого подхода необходимо считать каждую операцию атомарным действием, которое выполняется тестируемой системой без возможности вмешательства остальных. Если это не так, то нужно выделить такие атомарные действия и указывать их в модели в качестве возможных воздействий на систему и ее реакций. При этом может оказаться, что возвращение операцией управления нужно описать как отдельное действие, поскольку ее результат может зависеть от того, какие другие операции были вызваны во время ее работы.

Разработка и выполнение тестов

Разработка и выполнение тестов объединены в одну деятельность, поскольку собственно разработка тестов обычно происходит вперемешку с их прогонами и отладкой. При отладке тестов устраняются все обнаруживаемые ошибки в модели поведения и самих тестах, остаются только те, источник которых — некорректное поведение тестируемой системы.

Деятельность по разработке тестов может быть разделена на следующие действия.

1. **Выбор общей схемы теста для данного компонента (группы компонентов)**. Сначала на основе того, что известно о поведении компонентов и требованиях к полноте их тестирования нужно определить виды тестов, которые необходимо для них разработать. Это могут оказаться простейшие тесты, которые проверяют только, что вызываемая один раз операция возвращает результат, как-то похожий на правильный. Это могут быть тесты, проверяющие работу каждой операции в нескольких ситуациях, соответствующих различным вариантам ее использования или разным ветвям функциональности. Это могут быть более сложные тесты, проверяющие корректность работы группы операций с общим состоянием с помощью обхода автомата, моделирующего поведение этой группы операций.

   Еще более сложные тесты могут проверять поведение данной группы операций в зависимости от других операций, которые могут изменять общее состояние первых, или проверять корректность параллельной работы всех пар или троек операций из заданной группы.

   С точки зрения построения схемы теста также важно, какие операции (может быть одна) в нем будут участвовать, какого рода ситуации в этом тесте буду создаваться, будет ли использоваться внутреннее состояние тестируемого компонента, будут ли нужны нетривиальные тестовые данные, будет ли нужен параллелизм. Схема выбирается в соответствии с некоторой подходящей комбинацией перечисленных выше техник.

2. **Определение дополнительных проверок (если нужно)**.

   Иногда не все проверки корректности поведения стоит оформлять в спецификациях отдельных операций, поскольку для их выполнения необходимо знать полную историю обращений к системе или значительную ее часть. В этом случае то, что можно проверить в рамках вызова одной операции (на основе текущего состояния и аргументов вызова), проверяется в ее постусловии, а те проверки, для которых необходимо знать больше об истории обращений, записываются в тестовом сценарии.

Например, для тестирования генератора случайных чисел можно написать спецификацию, в которой проверять только принадлежность результата заданному интервалу. А в рамках сценария можно собирать все получаемые результаты и проверять общие ограничения на их распределение, используя, например, критерия Колмогорова.

3. **Разработка генераторов тестовых данных (если нужно).**
   Если необходимо использовать нетривиальные тестовые данные, для их построения разрабатывается набор генераторов с помощью подходящей комбинации техник, описанных в предыдущем разделе.
4. **Определение состояний и действий, построение автомата теста (если нужно)**.
   Для тестирования компонентов, поведение которых зависит от внутреннего состояния, тесты обычно разрабатываются с использованием автоматной модели такого компонента по одной из техник, описанных выше.
5. **Разработка адаптеров (если нужно)**.
   Если интерфейс тестируемой системы несколько отличается от того интерфейса ее модели поведения, для преобразования вызовов между этими двумя интерфейсами используются тестовые адаптеры.
   Тестовые адаптеры бывают разных видов, в зависимости от двух факторов: в какую сторону они выполняют преобразование (из модели в реализацию или обратно), и какого рода связи с моделью и с реализацией (тестируемой системой) они используют.
   Модельно-реализационные адаптеры выполняют преобразование обращений из модельного интерфейса в реализационный, а результаты операций или асинхронного возникающие события преобразую обратно — из реализационного вида в модельный.
   Реализационно-модельные адаптеры производят все преобразования только из реализационного вида в модельный.
   Модельно-реализационный адаптер может оформляться в виде класса, наследующего модельный класс и использующего классы тестируемой системы. Адаптеры, переводящие события из реализации в модель, наоборот, удобнее делать реализующими какие-то интерфейсы тестируемой системы и использующими модельные классы для сохранения информации о передаваемых ими событиях.
   Например, модельно-реализационный адаптер, реализующий описанный в предыдущей лекции класс спецификации списка через java.util.Vector может выглядеть примерно так.

```
public mediator class VectorAdapter<E>
  implements ListSpecification<E>
{
  implementation Vector target;

  public mediator void add(int i, E o)
    throws IndexOutOfBoundsException
  {
    implementation { target.insertElementAt(o, i); }
```

```
    update
    {
      E[] newItems = new E[items.length + 1];
      System.arraycopy(items, 0, newItems, 0, i);
      newItems[i] = o;
      System.arraycopy(items, i, newItems, i+1, items.length-i+1);
      items = newItems;
    }
  }
  ...
}
```

В приведенном примере в блоке update выполняется экстраполяция нового состояния модели в текущей ситуации. Если же, например, можно использовать результаты методов size и getElementAt как достоверные сведения о текущем состоянии вектора, можно вместо блоков update в каждом из методов-адаптеров написать один общий блок update для всего класса.

```
public mediator class VectorAdapter<E>
  implements ListSpecification<E>
{
  ...
  update
  {
    if(target == null) items = new E[];
    else
    {
      items = new E[target.size()];
      for(int i = 0; i < target.size(); i++)
        items[i] = target.getElementAt(i);
    }
  }
  ...
}
```

6. **Прогоны и отладка тестов**.
   Ну, и наконец, в рамках этого этапа необходимо оформить тесты и отладить их.

Разберем построение автоматного теста для списка, спецификации которого описаны в предыдущей лекции. Напомним, что для его тестирования выбран критерий полноты, требующий проверить работу списка в нормальных и исключительных ситуациях для методов add и remove, для метода indexOf — при наличии аргумента в списке и при его отсутствии. Кроме того, для всех методов нужно проверить их работу для пустого списка, а для методов remove и indexOf — для списка, содержащего только один элемент.

При построении автоматного теста, нацеленного на достижения заданного критерия полноты тестирования, используется *редукция модели по критерию полноты*. При применении этой техники по набору контрактов и критерию полноты строится автоматная модель, проход по всем переходам которой гарантирует достижение заданного критерия.

Такая редукция выполняется следующим образом.

1. Выделение целей тестирования.
   На первом шаге для каждой операции перечисляются выделяемые критерием полноты тестирования ситуации — это и есть цели тестирвоания.
   В нашем случае перечень этих ситуаций такой.
   - Метод add.
     - пустой список и корректный индекс;
     - пустой список и некорректный индекс;
     - непустой список и корректный индекс;
     - непустой список и некорректный индекс.

- Метод remove.
  - пустой список;
  - список с одним элементом и корректный индекс;
  - список с одним элементом и некорректный индекс;
  - список с двумя или более элементами и корректный индекс;
  - список с двумя или более элементами и некорректный индекс.
- Метод indexOf
  - пустой список;
  - список с одним элементом и аргумент в списке;
  - список с одним элементом и аргумента нет в списке;
  - список с двумя или более элементами и аргумент в списке;
  - список с двумя или более элементами и аргумента нет в списке.

Эти ситуации представляются как ограничения, вырезающие из области определения операции подобласти в пространстве возможных состояний системы и аргументов операции.

2. Получение обобщенных состояний и действий.
   Далее все области, соответствующие различным ситуациям, проецируются на пространство состояний. Рассматриваются все возможные пересечения подмножеств этих проекций. Полученные в итоге множества состояний для каждой проекции должны либо входить в нее, либо не пересекаться с ней. Эти множества образуют разбиение всех состояний на классы эквивалентности.

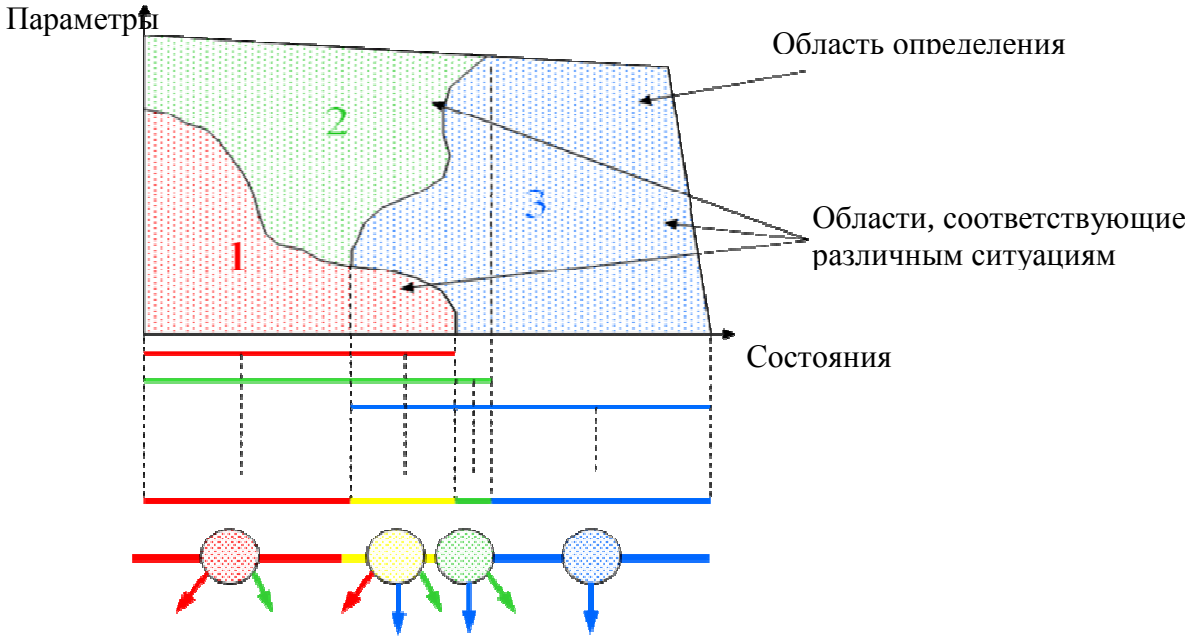


Рисунок 1. Построение обобщенных состояний и действий.

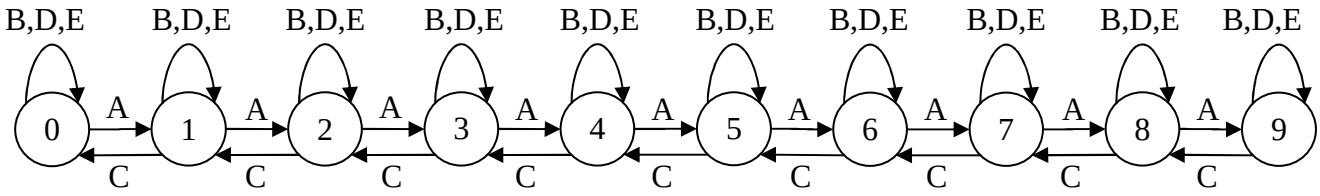
Полученные так множества состояний — обобщенные состояния — являются кандидатами на роль состояний автомата, моделирующего систему. Действия в этом автомате соответствуют всем возможным ситуациям. Проход по всем его переходам будет означать, что все исходные ситуации проверены.

В нашем примере такими пересечениями проекций ситуаций являются следующие обобщенные состояния.

- Пустой список.
- Список с одним элементом (произвольным).

- Список с двумя или более элементами.

3. Детерминизация полученного автомата.
   Полученный автомат может оказаться непригодным для контролируемого тестирования из-за высокой степени недетерминизма. Если он детерминирован — каждое действие в каждом обобщенном состоянии приводит в однозначно определяемое обобщенное состояние, то все хорошо. Если же это не так, что чтобы сделать его детерминированным, нужно расщепить те состояния, где есть недетерминированные действия. Каждый раз, когда у действия в некотором обобщенном состоянии может быть несколько возможных исходов (конечных состояний), мы разделяем это состояние на такие части, чтобы в каждой из этих частей исход действия определялся однозначно.
   Выполнение этого расщепления не всегда приводит в итоге к детерминированному автомату. В таком случае можно попробовать разделить операции на несколько групп, оставив в каждой из этих групп по одной из операций, порождающих недетерминированные действия. При таком делении часто можно построить несколько различных детерминированных автоматов (каждый соответствует только части операций, одна операция может использоваться в нескольких автоматах), обход каждого из которых дает желаемое покрытие всех ситуаций.
   В нашем примере источник недетерминизма один — при выполнении remove в списке с двумя или более элементами мы можем остаться в том же обобщенном состоянии (список будет иметь два или более элемента), а можем получить список с одним элементом.
   Чтобы удалить недетерминизм, нужно выделить в отдельное обобщенное состояние списки с двумя элементами. В оставшемся множестве — списки с тремя или более элементами — все равно останется тот же недетерминизм. Чтобы его удалить совсем, придется различать списки с различным числом элементов. Заметьте, что в полученном автомате никак не учитывается. Какие именно элементы находятся в списке — важно только их число.
   Действия в этом автомате соответствуют исходным ситуациям (аналогичные действия в различных состояниях можно обозначать одним способом).
   - Добавить элемент с корректным индексом.
   - Добавить элемент с некорректным индексом.
   - Удалить элемент с корректным индексом.
   - Удалить элемент с некорректным индексом.
   - Найти индекс первого вхождения элемента, отсутствующего в списке.
   - Найти индекс первого вхождения элемента, присутствующего в списке.
4. Ограничение полученного автомата.
   После предыдущего шага автомат может оказаться бесконечным. Тестирование является конечной процедурой, и для построения тестов нам достаточно какой-то конечной части этого автомата. Поскольку исходных целей тестирования конечное число, они достигаются при проходе по всем переходам в некотором конечном подавтомате. Чтобы выделить его, достаточно в некоторых состояния запретить действия, «уводящие дальше от начального состояния».
   Обычно такие пороговые состояния можно определить небольшим набором параметров. Эти параметры можно сделать параметрами теста, чтобы по одному и тому же описания можно было строить тесты различной сложности.
   В примере тестирования списка «уводящей от начального состояния» операцией является добавление элементов. Достаточно запретить его при достижении некоторого максимально возможного в тесте количества элементов N. Чтобы покрыть все исходные ситуации, достаточно, чтобы N было не меньше двух.



A — `add(), Normal case`
B — `add(), Exceptional case`
C — `remove(), Normal case`
D — `remove(), Exceptional case`
E — `indexOf(), все случаи`

Рисунок 2. Граф состояний автомата, моделирующего список, для значения параметра 9.

Осталось оформить неявное описание такого автомата в виде тестового сценария.

```
public scenario class ListTest
{
  ListSpecification<Object> list = mediator VectorAdapter<Object>
    (target = new Vector());  // инициализируем используемый адаптер

  int maxSize = 9;
  Object[] objPool = new Object[] { new Object(), null };

  public static void main(String[] args) // метод запуска теста
  {
    ListTest test = new ListTest();

    // устанавливаем используемый алгоритм обхода
    test.setExplorer(new jatva.exploration.DFSExplorer());

    test.run(); // выполняем тест
  }

  // блок, задающий вычисление текущего состояния
  state { return list.items.length; }

  scenario add()
  {
    if(list.items.length < maxSize) // добавляем элементы не всегда
    {
      iterate(int i = 0; i < maxSize; i++)
        iterate(Object o : objPool)
          list.add(i, o);
    }
  }

  scenario remove()
  {
    iterate(int i = 0; i < maxSize; i++)
      list.remove(i);
  }

  scenario indexOf()
  {
    iterate(Object o : objPool)
      list.indexOf(o);
  }
}
```

Анализ результатов тестирования

При анализе результатов необходимо выполнить следующие действия.

1. **Анализ обнаруженных ошибок**.
   Отчеты по результатам тестирования содержат список обнаруженных нарушений. Нарушение может иметь различную природу.

   - Это может быть ошибка в спецификациях, адаптерах или сценариях, проявляющаяся как исключительная ситуация.
     Наличие такой ошибки, означает, что спецификации написаны неправильно. Может быть, это просто описка или ошибка в коде спецификаций. Может быть, их автор рассчитывал на выполнение определенных ограничений, которые не соблюдаются. Стоит проанализировать эти ограничения — быть может, это пропущенное условие, которое нужно проверять или инвариант. В любом случае нужно внести поправки в спецификацию, хотя причиной подобной ситуации может быть неожиданное поведение тестируемой системы.
   - Другой вид ошибок — нарушения в структуре автомата теста.
     Например, он может оказаться слишком большим, и после некоторого числа переходов или состояний алгоритм обхода сообщит об этом, он неожиданно оказаться недетерминированным или несвязным — алгоритм обхода может не найти способа вернуться в некоторое обобщенное состояние, где он раньше уже был.
     Причиной таких ошибок чаще всего является неправильное построение теста — неправильно проведена детерминизация или ограничение слишком слабое (может быть, наоборот, нужно увеличить максимально возможное число переходов), забыты или неправильно определены какие-то действия.
     Гораздо реже причиной такой ошибки может стать ошибка в тестируемой системе. В этом случае, скорее всего, забыты какие-то проверки, которые помогли бы выявить ее раньше.
   - Нарушение предусловия.
     Оно означает, что нужно исправить тест, чтобы данная операция не вызывалась с некорректными аргументами.
   - Несоответствие между наблюдаемым поведением тестируемой системы и ее моделью поведения.
     Такие несоответствия проявляются как нарушения постусловий и инвариантов, невыполнение дополнительных проверяемых ограничений в сценариях, неожиданные исключения в тестируемой системе.
     В этом случае ошибка может быть как в спецификациях, адаптерах или тестах, так и в тестируемой системе. Чтобы выяснить это, нужно проанализировать создавшуюся ситуацию, еще раз проверить нарушенные ограничения — действительно они должны быть выполнены в этом случае и т.д. Часто такой анализ приводит к ошибкам не в тестируемой системе, а в тестах или моделях.
     Только в том случае, если это действительно ошибка тестируемой системы, в тесты не нужно вносить исправлений.

2. **Анализ тестового покрытия.**
   Отчеты анализируются и на предмет достигнутого покрытия, наличия непокрытых ситуаций и причин этого. Если достигнута желаемая полнота тестирования, разработку тестов можно прекратить. Если же непокрыты какие-то важные

ситуации или общее покрытие недостаточно, нужно поправить имеющиеся тесты или разработать новые, нацеленные на еще не покрытые ситуации.

Литература