Критерии тестового покрытия, основанные на структуре контрактных спецификаций

В. В. Кулямин
kuliamin@ispras.ru

Аннотация. В данной статье рассказывается о критериях тестового покрытия, применяемых в технологии UniTESK и основанных на структуре контрактных спецификаций функциональных требований. Эти критерии можно использовать при тестировании с использованием контрактных спецификаций (пред- и постусловий операций), независимо от применяемой при этом технологии. Приводится алгоритм определения достижимых комбинаций элементарных условий из кода пред- и постусловий, используемый для автоматического удаления недостижимых ситуаций из отчетов о тестовом покрытии.

1. Введение

Одним из наиболее широко используемых методов контроля качества программных и аппаратных систем является тестирование. Оно определяется как проверка соответствия поведения системы требованиям к ней, выполняемая на основе результатов работы этой системы в некотором конечном наборе специально созданных ситуаций [1]. В этом определении подчеркивается, что тестирование является конечной процедурой, оно всегда включает конечное число экспериментов с тестируемой системой или тестов.

Однако практически значимые системы настолько сложны, что количество различных ситуаций, которые необходимо протестировать для полной проверки одной такой системы, превосходит возможности сколь угодно щедро финансируемого проекта и требует колоссального времени для выполнения. Поэтому исчерпывающее тестирование, хотя и возможно теоретически из-за физической конечности любой вычислительной системы, практически совершенно невыполнимо. Этот факт зафиксирован в известной фразе Дейкстры: «Тестирование может быть использовано для демонстрации наличия ошибок, но никогда — для доказательства их отсутствия» [2].

Тем не менее, на практике именно тестирование чаще всего служит основой для оценки качества сложных систем, в том числе, и для определения того, насколько много в них ошибок, включая невыявленные. Есть ли какие-либо

основания для этого? Большинство инженеров считают, что за счет аккуратного и систематичного выбора тестовых ситуаций — входных данных, сценариев выполняемых действий и конфигураций тестируемой системы, используемых в них, — можно обеспечить достаточно высокий уровень доверия к результатам тестирования. То есть, аккуратный выбор небольшого конечного набора тестов позволяет ему адекватно отражать качество системы в целом, или, другими словами, стать статистически представительной выборкой из огромного множества всех ситуаций, в которых система должна функционировать.

Набор правил, следуя которому производится систематический отбор конечного числа тестов, принято называть *критерием полноты тестирования* (или *критерием его адекватности*) [3]. Определение подходящего для тестируемой системы критерия полноты является одной из важнейших задач, которые нужно решить при организации тестирования, наряду с определением требований к тестируемой системе.

Чаще всего критерий полноты основывается на разбиении всех ситуаций, в которых система должна работать, на классы эквивалентности на основе некоторых предположений о том, что в эквивалентных ситуациях система работает примерно одинаково, и поэтому либо в обеих присутствует ошибка, либо, наоборот, в обеих ошибки нет. Такие критерии полноты называются *критериями тестового покрытия*, а полнота тестирования с их помощью измеряется как *достигнутое тестовое покрытие* — процент классов эквивалентных ситуаций, задействованных в ходе тестирования, по отношению к числу всех классов. Эта метрика основана на простой эвристике — чем больше неэквивалентных, «существенно различающихся» ситуаций проверено, тем полнее было тестирование, и тем лучше оно отражает реальное качество системы.

Предположения, на основании которых выбираются критерии покрытия, могут иметь различную природу. Обычно используются два вида предположений — различные ситуации, возникающие при работе тестируемой системы, объявляются эквивалентными либо, исходя из того, что к ним относится один и тот же набор утверждений в требованиях к системе, либо на основе того, что в них задействован один и тот же набор элементов самой системы. Критерии первого вида называются *функциональными*, второго вида — *структурными*.

Функциональные критерии покрытия удобны для проверки соответствия поведения системы требованиям — при их использовании тесты нацеливаются на проверку требований, иногда в различных нетривиальных сочетаниях, и достаточно легко можно обнаружить, что какие-то требования совсем не реализованы. Структурные критерии покрытия удобны для тестирования отдельных модулей или сценариев взаимодействия небольшого числа модулей, их использование также позволяет выявить недостижимый код и участки кода с неясным предназначением, подозрительные как с точки

зрения наличия в них ошибок, так и с точки зрения защищенности тестируемой системы и безопасности ее работы. Часто на практике используется некоторая комбинация из функциональных и структурных критериев покрытия, поскольку в сочетании они компенсируют недостатки друг друга, сохраняя достоинства.

При создании технологий и инструментов автоматизации тестирования используемые в них критерии тестового покрытия играют важную роль, определяя как отдельные элементы этих инструментов и технологий, так и границы их применимости. В разработанной в Институте системного программирования РАН технологии тестирования UniTESK [4-6] используются, в основном, функциональные критерии покрытия. Эти критерии определяются на основе структуры формальных спецификаций функциональных требований к тестируемой системе, являющихся основным источником тестов в рамках этой технологии, и поэтому, являясь функциональными по существу, по форме они очень похожи на структурные. Тестовые ситуации объявляются эквивалентными в их рамках, если в этих ситуациях задействован один и тот же набор элементов спецификаций, представляющих требования к тестируемой системе.

Данная статья представляет применяемые в UniTESK критерии покрытия, которые с равным успехом могут использоваться и в других подходах, где возникают контрактные спецификации в виде пред- и постусловий операций, использующих модельные состояния описываемой системы. В статье также описывается алгоритм определения классов эквивалентных ситуаций по отношению к самому детальному из используемых критериев — *критерию комбинаций логических условий в спецификациях*. Кроме того, обсуждаются возможные варианты развития механизмов описания и использования критериев покрытия в тестах, создаваемых с помощью UniTESK.

2. Критерии покрытия в UniTESK

Прежде, чем приступить к описанию применяемых в UniTESK критериев покрытия, стоит рассмотреть общую структуру спецификаций, используемых в этой технологии.

2.1. Структура формальных спецификаций UniTESK

В технологии UniTESK для описания требований к тестируемой системе используются *формальные контрактные спецификации* [4-6]. Это означает, что ими формально фиксируется синтаксис интерфейса тестируемой системы — названия операций и типы параметров и результатов для каждой из них, а также *программный контракт* каждой из операций [7].

Программный контракт операции состоит из *предусловия* и *постусловия*. Первое фиксирует требования к корректному использованию этой операции со стороны окружения системы — при каких ограничениях на аргументы

обращение к этой операции корректно. Второе определяет обязательства системы по отношению к результатам вызовов этой операции — какие ограничения на результаты работы операции должны быть выполнены при корректной работе системы, если обращение к ней было правильным. При нарушении предусловия операции обращение к ней может иметь любые последствия, поведение системы в этом случае не определено. Совместно пред- и постусловия строго определяют требования к работе системы.

При анализе сложной системы с большим количеством интерфейсных операций крайне неудобно рассматривать их все вместе как однородный набор, необходимо разбиение этих операций на какие-то логические группы по реализуемым ими функциям. Эти группы почти всегда соответствуют компонентам или модулям системы, ответственным за реализацию этих функций. Часто такие компоненты имеют внутреннее состояние, недоступное для непосредственного наблюдения извне системы, но влияющее на поведение вызываемых операций компонента и изменяемое ими.

Поэтому для описания ограничений в пред- и постусловиях необходимо как-то учитывать внутреннее состояние компонента, операции которого описываются. Чтобы сделать это возможным, контракты операций одного компонента объединяются в спецификацию этого компонента вместе с описанием *структуры его модельного состояния*.

Структура модельного состояния компонента не обязана совпадать со структурой его реального состояния, зафиксированной в его коде, например, компонент, реализующий список, может использовать ссылочную структуру данных, в том время, как структура его модельного состояния в спецификациях может быть основана на массиве. В структуре модельного состояния могут совсем отсутствовать некоторые части реального состояния компонента, слабо связанные с его функциональностью, например, различного рода кэши и другие данные, используемые для увеличения производительности. Важно, что структура модельного состояния компонента должна содержать данные, достаточные для полного описания функциональности в пред- и постусловиях операций этого компонента. При этом предусловие зависит от аргументов вызова операции и данных модельного состояния при этом вызове, или *пре-состояния*. Постусловие зависит как от аргументов и пре-состояния, так и от результата и данных модельного состояния после вызова, или *пост-состояния*.

Наличие сложной структуры модельного состояния приводит к необходимости учитывать в контрактах операций ограничения на его данные, которые должны выполняться в стабильных состояниях системы, когда ни одна из операций не выполняется. Такие ограничения оформляются в виде *инвариантов* модельного состояния. Инварианты являются общими частями пред- и постусловий всех операций, как-то затрагивающих соответствующие компоненты — они должны выполняться как в пре-состоянии любого вызова, так и в его пост-состоянии.

Предусловия, постусловия и инварианты в UniTESK оформляются на расширении одного из языков программирования. Каждая из этих конструкций представляет собой блок кода, возвращающий значение булевского типа и помеченный для наглядности специальным ключевым словом, соответственно, **`pre`**, **`post`** или **`invariant`**.

2.2. Критерии тестового покрытия на основе структуры пред- и постусловий

Самым простым из используемых на практике является ***критерий покрытия операций***, в рамках которого эквивалентными считаются все обращения к одной и той же операции. Этот критерий наименее детальный, и стопроцентное покрытие согласно ему гарантирует только минимальное качество тестов — каждая тестируемая операция вызвана хотя бы один раз. Можно заметить, что определение этого критерия никак не связано с использованием спецификаций или требований.

Более детальные критерии покрытия на основе контрактных спецификаций уже опираются на их структуру. Пред- и постусловия операций представляют собой запись требований к описываемой ими системе, поэтому критерии покрытия, основанные на их структуре являются функциональными. С другой стороны, пред- и постусловия можно рассматривать как некоторый код и определять критерии его покрытия таким же образом, как это делается для кода программ.

Обычно постусловие операции описывает несколько разных режимов ее функционирования. Предусловие запрещает все комбинации значений параметров, при которых поведение операции не определено. Однако среди разрешаемых предусловием ситуаций могут остаться такие, в которых выполнить свою основную функцию операция не может. Вместо этого она должна вернуть некоторый код ошибки, создать исключительную ситуацию или как-то иначе сигнализировать о том, что ее основная задача не может быть решена в такой ситуации. Помимо режимов сигнализации о невозможности выполнить основную функцию, само ее выполнение может идти разными путями. Например, операция конкатенации двух строк может проверять, не является ли один из ее аргументов пустой строкой, и в этом случае просто возвращать копию другого аргумента, не выполняя никаких действий над ней. Если же оба аргумента не пусты, копия одного из них должна быть подвергнута модификации, чтобы получить результат операции.

При написании постусловий удобно выделять такие разные режимы, иначе постусловие превращается в большую и сложную формулу, трудную для анализа и понимания, что противоречит основной цели создания формальных спецификаций. Различные режимы работы операции описываются в виде импликаций с несовместными посылками (постусловие при этом выглядит как формула $X_1 \Rightarrow Y_1$ & … & $X_n \Rightarrow Y_n$) или в виде разных выражений для возвращаемого результата при разных исходных условиях, т.е. результат

постусловия вычисляется в разных ветвях некоторого условного оператора или оператора выбора по-разному. Поэтому эти разные режимы называют *ветвями функциональности* (или функциональными ветвями) операции.

Соответственно, следующим в сторону повышения детальности после критерия покрытия операций является ***критерий покрытия ветвей функциональности.*** Он объявляет эквивалентными все вызовы одной операции, в которых выполняется одна и та же функциональная ветвь.

В инструментах UniTESK, основанных на использовании расширений различных языков, используются несколько разные конструкции для выделения функциональных ветвей. В расширении Java вводится оператор **`branch`**, который помечает ветвь условного оператора или оператора выбора в коде постусловия, соответствующую ветви функциональности и определяет идентификатор этой ветви функциональности.

Поскольку критерии тестового покрытия в UniTESK используются во время выполнения тестов для фильтрации тестовых данных и отбора только таких вызовов, которые повышают уже достигнутое покрытие, на размещение операторов **`branch`** наложены ограничения. Они позволяют генерировать фильтры, которые без обращения к самой тестируемой операции могут определить ветвь функциональности, соответствующую произвольному набору ее аргументов. Ограничения состоят в следующем.

- Все ветвления на пути по потоку управления от начала постусловия до любого из операторов **`branch`** должны зависеть только от аргументов вызова и данных пре-состояния. При соблюдении этого условия ветвь функциональности можно вычислить, не вызывая саму операцию. По сути, данные пост-состояния могут появиться в постусловии только после одного из операторов **`branch`**, поэтому такие операторы помечают в постусловии места, в которых вызывается тестируемая операция во время работы теста.
- На любом пути по потоку управления от начала постусловия до выхода из него должен быть ровно один из операторов **`branch`**. Это позволяет отнести каждый вызов, удовлетворяющий предусловию к одной и только одной из функциональных ветвей.

В расширении языка C, используемом для разработки тестов с помощью технологии UniTESK, ветви функциональности определяются в блоке **`coverage`**, отделенном от постусловия. Такие блоки используются для вычисления классов эквивалентности тестовых ситуаций по разным критериям, в том числе и по критерию покрытия функциональных ветвей.

Все другие критерии покрытия, используемые в UniTESK, являются более детальными, чем критерий покрытия ветвей функциональности. В порядке увеличения детальности они приведены ниже.

- ***Критерий покрытия помеченных путей.***
  Операторы **`branch`**, определяющие ветви функциональности, по сути являются специальными метками, с помощью которых в коде постусловия выделяются классы эквивалентности вызовов соответствующей операции. Расстановку меток можно использовать и для более гибкого определения покрытия.
  Для этого используются метки в виде операторов **`mark`** с идентификатором данной метки. Такие метки можно использовать как в предусловии, так и в постусловии, с единственным ограничением — не ставить их внутри циклов. При следовании ему существует лишь конечное множество цепочек меток, которые могут быть пройдены при выполнении проверок предусловия и постусловия. Можно рассматривать как эквивалентные вызовы, которые приводят к прохождению одной и той же такой последовательности. Однако в UniTESK класс эквивалентности вызова должен быть вычислим без обращения к самой тестируемой операции, поэтому такие цепочки нужно обрывать при прохождении оператора **`branch`**.
  Критерий покрытия помеченных путей рассматривает как эквивалентные вызовы операции, приводящие к прохождению одной и той же цепочки операторов **`mark`** в пред- и постусловии этой операции, заканчивающейся оператором **`branch`**. Таким образом, классы вызовов по этому критерию вычислимы без обращения к операции, а сам критерий является не менее детальным, чем критерий покрытия ветвей функциональности.

- ***Критерий покрытия определяющих путей.***
  Рассмотрим граф потока управления предусловия, в котором все циклы стянуты в вершины, т.е. все вершины графа, входящие в один и тот же цикл, отождествлены в одну вершину, все ребра, входящие в них, входят в эту вершину, все выходящие — выходят из нее, а все ребра, ранее входившие в состав циклов, отброшены. В получаемом таким образом графе нет циклов, а значит количество путей, ведущих из его стартовой вершины, соответствующей началу предусловия, в одну из конечных вершин, соответствующих выходу из предусловия, конечно. То же самое верно и для графа потока управления постусловия, в котором все циклы стянуты в вершины, и все пути оборваны на операторах **`branch`**.
  Определяющим путем называется путь по графам потока управления предусловия и постусловия со стянутыми в вершины циклами, который оканчивается оператором **`branch`** и проходится при выполнении проверок при одном из вызовов операции. Из сказанного выше следует, что существует лишь конечный набор определяющих путей, и любой вызов операции соответствует одному из них, поскольку на любом пути по потоку управления от начала

постусловия до выхода из него стоит ровно один оператор **`branch`**. Можно отметить также, что каждый определяющий путь однозначно определяет лежащую на нем цепочку меток, которая заканчивается оператором **`branch`**.
Критерий покрытия определяющих путей считает эквивалентными вызовы операции, приводящие к прохождению одного и того же определяющего пути. Этот критерий покрытия не менее детален, чем критерий покрытия помеченных путей.
Ближайшим аналогом этого критерия является *критерий покрытия ветвей* исходного кода (branch coverage или decision coverage) [3,8]. В его рамках эквивалентными являются обращения к операции, при которых выполняется один и тот же набор ветвей во всех условных операторах, операторах выбора или операторах цикла. Определяющий путь тоже соответствует набору ветвей в постусловии, без рассмотрения циклов и всего кода в них, а также без учета кода, выполняемого после операторов **`branch`**.

- ***Критерий покрытия комбинаций элементарных условий.***
  Элементарным условием в данной статье будем называть логическое выражение, являющееся частью условия одного из условных операторов или операторов выбора в пред- или постусловии операции на определяющем пути, которое неразложимо на более мелкие логические выражения.
  Значения элементарных условий определяют выбор ветвей в условных операторах и операторах выбор, а значит и путь по потоку управления пред- и постусловия, проходимый при этих значениях.
  Полной комбинацией элементарных условий мы называем набор значений элементарных условий, в котором каждому условию сопоставлено либо значение true («истина»), либо false («ложь»), либо неопределенное значение, причем имеющийся набор определенных значений (true и false) однозначно задает некоторый определяющий путь в пред- и постусловии операции, и все определенные значения хотя бы один раз вычисляются при прохождении этого пути. Последнее ограничение означает, что неопределенное значение могут иметь те и только те элементарные условия, значения которых не влияют на выбор пути. Произвольный набор значений элементарных условий *накрывает* данную полную комбинацию, если значения, определенные в этой комбинации, совпадают со значениями, определенными в этом наборе.
  Критерий покрытия комбинаций элементарных условий объявляет эквивалентными вызовы одной и той же операции, при которых наборы значений элементарных условий в пред- и постусловии накрывают одну и ту же полную комбинацию. Поскольку элементарных условий конечное множество, их полных комбинаций

также конечное множество, причем каждая полная комбинация однозначно задает определяющий путь, хотя некоторые различные комбинации могут задавать один и тот же определяющий путь (из-за использования дизъюнкций нескольких элементарных условий в одном условии ветвления). Это означает, что критерий покрытия комбинаций элементарных условий не менее детален, чем критерий покрытия определяющих путей.

Аналогичный критерий покрытия кода называется *критерием покрытия комбинаций условий* (multiple condition coverage) [3,9]. Однако, согласно определению в его рамках эквивалентны такие ситуации, в которых все элементарные условия, используемые в ветвлениях, имеют в точности одни и те же значения. Использование неопределенных значений обычно игнорируется, хотя на практике оно необходимо. Например, если условие имеет вид `(object != null) && object.isCertain()`, в него входят два элементарных условия — `object != null` и `object.isCertain()` — причем, если первое имеет значение false, то значение второго не определено и не может быть определено корректно и осмысленно. Наиболее естественное расширение определения этого критерия, допускающее неопределенные значения, все-таки несколько отличается от данного выше для критерия комбинаций условий в спецификациях. Первое, в отличие от второго, использует значения всех формул, которые можно корректно определить, а не только тех, которые необходимы для вычисления пути по графу потока управления. В первом случае неопределенные значения могут оставаться только у таких условий, для которых вычисление значений требует, чтобы другие условия имели конкретные значения, как в приведенном примере с условиями `object.isCertain()` и `object != null`.

2.3. Пример использования критериев покрытия

Для пояснения определений из предыдущего раздела, рассмотрим класс, реализующий функции простого банковского счета с изменяемым минимальным разрешенным значением баланса. Такой счет должен иметь текущее значение баланса и минимальное разрешенное значение баланса; для простоты будем считать оба этих значения целыми числами, представимыми типом **`int`** в языке Java.

У рассматриваемого класса имеются операции изменения минимального разрешенного значения баланса и изменения текущего значения баланса. Ниже приведены описание структуры модельного состояния для такого класса и контрактная спецификация второй операции на расширении языка Java, используемом в UniTESK.

В приведенном примере есть два разных режима работы операции изменения баланса. В одном режиме возможное переполнение или нарушение ограничения, требующего, чтобы текущее значение баланса было не меньше минимального разрешенного его значения, приводят к тому, что состояние счета не изменяется и возвращается результат **`false`**; проверки, соответствующие этому режиму помечены оператором **`branch`** `NoChanges`. В другом режиме никакие ограничения не нарушаются, и переполнение не происходит, поэтому текущее значение баланса изменяется на величину параметра операции и возвращается результат **`true`**; этот режим помечен оператором **`branch`** `NormalCase`.

```
specification class AccountSpecification
{
  int balance    = 0;
  int minBalance = 0;

  invariant balanceNotLessThanMinimum()
  {
    return balance >= minBalance;
  }

  specification boolean deposit(int s)
  {
    post
    {
      tautology (balance >= minBalance)
      && (s >= 0 || Integer.MIN_VALUE - s <= 0);

      if(s > 0 && Integer.MAX_VALUE - s < balance)
        mark "Overflow";
      else if(   s < 0 && minBalance < 0
      && minBalance - s > balance
              || s < 0 && minBalance >= 0
              && minBalance     > balance + s
            )
        mark "Under minimum";
      else
      {
        branch NormalCase;

        return    balance    == pre balance + s
               && minBalance == pre minBalance
               && deposit    == true;
      }
```

```
        if(s < 0 && Integer.MIN_VALUE - s > balance)
          mark "Underflow";

        branch NoChanges;

        return    balance    == pre balance
               && minBalance == pre minBalance
               && deposit    == false;
      }
    }
  }
```

Кроме того, с помощью операторов **mark** выделены особые ситуации: переполнение с переходом через максимальное значение целого числа в Java (Overflow), переполнение с переходом через минимальное значение целого числа в Java (Underflow) и возможное нарушение ограничения на минимальное текущее значение баланса (Under minimum).

Классы эквивалентности вызовов, определяемые для операции изменения текущего значения баланса критериями покрытия, описанными в предыдущем разделе, приведены в Таблице 1. Полные условия, соответствующие определяющим путям, и элементарные формулы приведены в Таблице 2. Знаки '+' и '–' в Таблице 1 обозначают значения элементарных условий true и false, соответственно. Пустая клетка в строке комбинации обозначает, что соответствующее элементарное условие в данной комбинации имеет неопределенное значение.

Ветви функцио-нальности	Помеченные пути	Определя-ющие пути	Комбинации элементарных условий						
			C1	C2	C3	C4	C5	C6	C7
NormalCase	NormalCase	P1	–	–					
			–	+	–				
			+	–		–		–	
			+	–		+	–		
NoChanges	Overflow; NoChanges	P2	–	+	+				
	Under minimum; NoChanges	P3	+	–		–		+	–
			+	–		+	+		–
	Under minimum; Underflow; NoChanges	P4	+	–		+	+		+

Таблица 1. Классы эквивалентности, выделяемые описанными критериями покрытия в примере.

В рассматриваемом примере критерии покрытия помеченных и определяющих путей дают одни и те же классы эквивалентности — каждый

определяющий путь проходит через уникальный набор меток. Однако в общем случае критерий покрытия определяющих путей оказывается более детальным, чем критерий покрытия помеченных путей.

2.4. Недостижимые ситуации

В Таблице 1 перечислены только достижимые, встречающиеся на практике классы эквивалентности по различным критериям. В принципе, из управляющих конструкций кода постусловия можно сделать вывод, что возможна последовательность меток `Overflow; Underflow; NoChanges`. Недостижимость ее следует из того, что условие прохода через метку `Overflow` включает `s > 0`, а условие прохода через `Underflow` — `s < 0`.

Условия выполнения определяющих путей	
P1	`!(0 < s && Integer.MAX_VALUE – s < balance) && !(s < 0 && minBalance < 0 && balance < minBalance – s || s < 0 && !(minBalance < 0) && balance + s < minBalance)`
P2	`0 < s && Integer.MAX_VALUE – s < balance && !(s < 0 && balance < Integer.MIN_VALUE – s)`
P3	`!(0 < s && Integer.MAX_VALUE – s < balance) && s < 0 && minBalance < 0 && balance < minBalance – s || s < 0 && !(minBalance < 0) && balance + s < minBalance && !(s < 0 && balance < Integer.MIN_VALUE – s)`
P4	`!(0 < s && Integer.MAX_VALUE – s < balance) && s < 0 && minBalance < 0 && balance < minBalance – s || s < 0 && !(minBalance < 0) && balance + s < minBalance && s < 0 && balance < Integer.MIN_VALUE – s`
Элементарные условия	
C1	`s < 0`
C2	`0 < s`
C3	`Integer.MAX_VALUE – s < balance`
C4	`minBalance < 0`
C5	`balance < minBalance – s`
C6	`balance + s < minBalance`
C7	`balance < Integer.MIN_VALUE – s`

Таблица 2. Условия определяющих путей и элементарные условия в примере.

Более тяжело определить недостижимость комбинации элементарных условий C1 = true, C2 = false, C4 = false, C6 = true, C7 = true. При этих условиях

выполнено `s < 0`, `minBalance >= 0`, `balance < Integer.MIN_VALUE - s`. Из первого условия и того, что `Integer.MIN_VALUE` является наименьшим возможным значением типа **`int`**, следует, что `Integer.MIN_VALUE - s <= 0`; из этого и условия C7 — что `balance < 0`, а отрицание условия C4 вместе с последним следствием дают `balance < minBalance`, что прямо нарушает инвариант. Таким образом, указанная комбинация значений элементарных условий действительно недостижима.

Для построения корректных тестовых отчетов необходимо выделять недостижимые среди классов эквивалентности, задаваемых критериями тестового покрытия. Эти классы не должны учитываться в отчетах, и итоговое тестовое покрытие должно вычисляться на основе только достижимых классов эквивалентности. Определить недостижимость некоторой ситуации на практике всегда возможно — можно придумать искусственные примеры, в которых это невозможно из-за связи этих примеров с неразрешенными математическими проблемами, однако в реальных системах такие ситуации не встречаются. Однако делать это вручную для сложных систем, имеющих тысячи интерфейсных операций и десятки, а то и сотни неэквивалентных видов ситуаций для каждой операции, крайне трудоемко. В то же время, если недостижимые классы будут присутствовать в отчетах о тестировании, многие тесты не будут давать 100% покрытие, хотя улучшить их не будет никакой возможности. Поэтому необходимо иметь средства для автоматического устранения недостижимых классов ситуаций из отчетов о тестовом покрытии.

В инструментах UniTESK, поддерживающих разработку тестов на расширении Java такие средства есть. В частности, для указания недостижимости указанной выше комбинации элементарных условий в нашем примере использована *тавтология* **`tautology`** `(balance >= minBalance) && (s >= 0 || Integer.MIN_VALUE - s <= 0)`. Тавтологии указывают логические выражения, которые истинны в силу смысла используемых в них переменных и операций, а не в силу своей синтаксической структуры. Они используются инструментом построения тестов, чтобы отсекать невыполнимые комбинации условий.

Помимо тавтологий, указанных пользователями, инструмент использует естественные тавтологии, отражающие базовые свойства операций сравнения для типов в Java, например, для любых выражений x и y верно $(x == y) \| (x != y)$, при любых числовых выражениях x, y, z выполнено $!(x < y \&\& y < z \&\& z < x)$. Тавтологии такого типа не нужно вписывать в спецификации.

Достижимость различных помеченных или определяющих путей в UniTESK вычисляется на основе достижимости соответствующих комбинаций значений элементарных условий. Для этого применяется алгоритм, описанный ниже.

2.5. Определение достижимых комбинаций элементарных условий

Приводимый в этом разделе алгоритм используется для вычисления набора достижимых комбинаций значений элементарных условий, извлеченных из пред- и постусловия одной операции.

Извлеченные из кода спецификаций элементарные условия считаются одинаковыми, если они совпадают текстуально. Такой способ их отождествления накладывает следующие ограничения на написание пред- и постусловий.

- Один и тот же идентификатор, используемый в условиях разных ветвлений, должен обозначать одно и то же. Это ограничение должно выполняться как для используемых идентификаторов полей классов, так и для локальных переменных. Поэтому нельзя определять в разных частях постусловия локальные переменные с одним и тем же идентификатором, и использовать их в условиях ветвлений или операторах выбора.
- Кроме того, если переменная используется в условиях нескольких различных ветвлений, ее значение между использованиями не должно изменяться. Иначе одинаковые текстуально условия ($x > 0$) будут означать разные ситуации.

Алгоритм использует список вхождений элементарных условий в пред- и постусловия, в котором вхождения перечисляются в том же порядке, что и в условиях ветвлений или операторах выбора в коде предусловия и постусловия. Заметим, что одно и то же элементарное условие может входить в него несколько раз, поскольку несколько раз может встречаться в разных местах кода. Каждому вхождению условия на каждом шаге алгоритма присваивается одно из значений true или false. Список текущих значений вхождений элементарных формул будем называть таблицей значений, номер вхождения в этом списке — индексом этого вхождения.

На каждом шаге алгоритма определяется множество существенных вхождений элементарных условий, т.е. тех, чьи значения используются для вычисления пути по графу потока управления. В графе потока управления все циклы считаются стянутыми в вершины, поэтому условия циклов и условия ветвлений внутри циклов в работе алгоритма никак не участвуют.

Результат работы алгоритма — набор достижимых комбинаций элементарных условий, удовлетворяющих всем естественным или явно описанным тавтологиям. Кроме этого, составляется таблица соответствий между этими комбинациями и определяющими путями, состоящими из пройденных ветвей, а также пройденными последовательностями операторов **`mark`**.

Введем следующие обозначения.

- Множество элементарных условий: *PC*.

- Количество вхождений элементарных условий: N.
- Множество индексов вхождений элементарного условия $f \in PC$: $I(f) \subseteq [1..N]$.
- Текущее значение вхождения условия с индексом $i \in [1..N]$: $v(i) \in \{\text{true, false}\}$.
- Множество существенных вхождений: $Ess \subseteq [1..N]$.
- Множество наборов достижимых комбинаций: *Reachable*.

1. Начало работы: сначала все значения условий равны false, множество существенных вхождений пусто, множество наборов достижимых комбинаций пусто.
$\forall i \in [1..N]\ v(i) := \text{false}$
$Ess := \varnothing$
Reachable $:= \varnothing$

2. Главный цикл.

3. Вложенный цикл устранения различных значений вхождений одного условия. Пока есть различные значения вхождений одного условия
$\exists f \in PC\ \exists i,j \in I(f)\ i < j\ \&\ v(i) \neq v(j)$.

Находим пару вхождений одного и того же условия с разными значениями, в которой второе вхождение имеет наименьший индекс среди всех таких пар для всех условий и меняем в этой паре значение false на true.
Находим $(i, j) \in [1..N]\times[1..N]$: $j = min\{k: \exists m < k\ \exists f \in PC\ m,k \in I(f)\ \&\ v(m) \neq v(k)\}\ \&\ \exists f \in PC\ i,j \in I(f)\ \&\ i < j\ \&\ v(i) \neq v(j)$
Выполняем
$v(i) = \text{false} \Rightarrow v(i) := \text{true}$
$v(j) = \text{false} \Rightarrow v(j) := \text{true}$
Возвращаемся к проверке условия вложенного цикла 3.

Если условие вложенного цикла не выполнено — нет формул с разными значениями вхождений, переходим к следующему пункту.

4. Проверка тавтологий. Проверяем выполнение всех естественных тавтологий и тавтологий, указанных в операторах **`tautology`**, на текущем наборе значений условий.

Если хотя бы одна тавтология нарушена, добавляем в множество существенных вхождений первые вхождения используемых в ней элементарных условий.

Если все значения существенных вхождений равны true, конец главного цикла (идем в п. 7).

Иначе меняем значение последнего вхождения, предшествующего последнему существенному и равное false, на true, а все следующие за ним значения — на false, делаем множество существенных вхождений пустым.
Находим $i \in [1..N]$: $i \le \max(Ess)\ \&\ v(i) = \text{false}$.
Выполняем
$v(i) := \text{true}$
$\forall j\ i < j \le N \Rightarrow v(j) := \text{false}$
$Ess := \varnothing$.

Продолжаем главный цикл (идем в п. 2).

Иначе, если все тавтологии выполнены, переходим в следующий пункт.

5. Вычисляем последовательно условия ветвлений в предусловии операции, используя имеющийся набор значений вхождений формул и правила вычислений логических выражений в используемом языке, и проходя в те ветви, условия которых выполнены. Каждое вхождение, значение которого понадобилось при этих вычислениях, добавляем в множество существенных вхождений.

Если результат предусловия равен false:

Если значения всех существенных вхождений равны true, конец главного цикла (идем в п. 7).

Иначе, меняем значение последнего существенного вхождения, равного false, на true, а значения всех вхождений, следующих за ним, приравниваем false, и делаем множество существенных вхождений пустым.

Находим $i \in Ess : v(i) = \text{false}$.
Выполняем
$v(i) := \text{true}$
$\forall j\ i < j \le N \Rightarrow v(j) := \text{false}$
$Ess := \varnothing$.

Продолжаем главный цикл (идем в п. 2).

Иначе, продолжаем работу, переходя в следующий пункт.

6. Вычисляем последовательно условия ветвлений в постусловии операции, используя имеющийся набор значений вхождений формул и правила вычислений логических выражений в используемом языке, и проходя в те ветви, условия которых выполнены. Каждое вхождение, чье значение понадобилось при этих вычислениях,

добавляем в множество существенных вхождений. Делаем так, пока не дойдем до одного из операторов **`branch`**.

Строим текущий набор значений *current*: для всех существенных вхождений заносим в него значения соответствующих элементарных условий, а для остальных условий (у которых нет существенных вхождений на данном проходе) заносим неопределенное значение.

$\forall f \in PC\ I(f) \cap Ess \neq \varnothing \Rightarrow current(f) := v(i), i \in I(f)$
$\forall f \in PC\ I(f) \cap Ess = \varnothing \Rightarrow current(f) :=$ undef
$Reachable := Reachable \cup \{current\}$.

Если значения всех существенных вхождений равны true, конец главного цикла (идем в п. 7).

Иначе, меняем значение последнего существенного вхождения, равного false, на true, а значения всех вхождений, следующих за ним, приравниваем false, и делаем множество существенных вхождений пустым

Находим $i \in Ess : v(i) =$ false.
Выполняем
$v(i) :=$ true
$\forall j\ i < j \le N \Rightarrow v(j) :=$ false
$Ess := \varnothing$.

Продолжаем главный цикл (идем в п. 2).

7. Конец главного цикла.

Возвращаем построенное множество *Reachable*.

То, что приведенный алгоритм возвращает только выполнимые комбинации значений элементарных условий, следует из действий, выполняемых в п. 3 и 4. Они гарантируют, что в дальнейших пунктах используются только такие наборы значений, в которых все вхождения одного условия имеют одно значение и все тавтологии выполнены.

Полноту выдаваемого набора комбинаций доказать несколько сложнее. Это доказательство опирается на то, что при наличии нескольких вхождений одного условия, их значения надо перебирать отдельно, чтобы не выбросить случайно достижимую комбинацию. Соответствующий перебор выполняется в п. 3.

Кроме того, оптимизация перебора за счет выбрасывания комбинаций, отличающихся по значениям несущественных вхождений, после нарушения тавтологии выполняется иначе, чем после нарушения предусловия или полного прохода по пред- и постусловию. Это также связано с возможностью пропуска достижимых комбинаций.

3. Заключение

В статье описаны критерии покрытия, используемые в технологии построения тестов UniTESK и основанные на структуре контрактных спецификаций функциональных требований к программному обеспечению. Несмотря на ряд особенностей спецификаций, используемых в UniTESK (операторы **`branch`** и связанные с ними требования об использовании пре- и пост-значений), эти критерии можно использовать и для произвольных контрактных спецификаций, в которых предусловия и постусловия являются вычисляемыми по некоторой схеме предикатами.

Аналоги критериев покрытия функциональных ветвей и комбинаций значений элементарных условий достаточно легко переформулировать для тех случаев, когда постусловие является единой логической формулой. В таких случаях постусловие обычно представляется конъюнкцией нескольких импликаций ($X_1 \Rightarrow Y_1$ & … & $X_n \Rightarrow Y_n$), посылки которых несовместны (т.е. X_1, …, X_n попарно исключают друг друга). Критерий покрытия функциональных ветвей соответствует критерию покрытия посылок — эквивалентными объявляются ситуации, в которых выполнена одна из посылок X_i. Критерий покрытия комбинаций значений элементарных условий основан на представлении каждой посылки X_i в дизъюнктивной нормальной форме, при этом эквивалентными считаются ситуации, в которых выполнен один и тот же дизъюнкт в одной и той же посылке.

На практике достаточно часто элементарные условия, входящие в посылки постусловия, не являются независимыми. Во-первых, они могут быть взаимосвязаны, если используют равенства и неравенства произвольных объектов, а также порядковые сравнения чисел. Во-вторых, они могут быть связаны благодаря специфическому смыслу входящих в них переменных и функций, пример чего приведен в разделе 2.3.

Для определения достижимых комбинаций значений элементарных условий в инструментах UniTESK был разработан алгоритм, представленный в разделе 2.5. Он позволяет автоматически удалять из отчетов о полученном тестовом покрытии недостижимые комбинации, существенно сокращая тем самым трудоемкость анализа результатов тестирования. Известны практические примеры спецификаций, в которых количество достижимых комбинаций условий исчисляется сотнями, и примерно такое же количество комбинаций недостижимо. В этих случаях разбирать вручную, какие из непокрытых комбинаций можно покрыть, а какие — нет, крайне тяжело.

Описанный алгоритм опирается на указанные пользователем тавтологии, выражающие логические зависимости между элементарными условиями. Кроме того, используются так называемые естественные тавтологии, выполненные в элементарной теории равенства и неравенства объектов (равенство является отношением эквивалентности и переносится на все атрибуты равных объектов) или в теориях порядка на целых или

действительных числах (с учетом машинного представления этих чисел). Для автоматического анализа естественных тавтологий используются специфические алгоритмы, описываемые в других работах.

Литература

[1] Guide to the Software Engineering Body of Knowledge: 2004 Edition — SWEBOK. IEEE, 2005.

[2] E. W. Dijkstra. Notes on Structured Programming. T. H. Report 70-WSK-03. Eindhoven, Netherlands: Technological University, 1970.

[3] 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.

[4] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME'2002, Kopenhagen, Denmark, LNCS 2391:77–88, Springer-Verlag, 2002.

[5] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25–43, 2003.

[6] А. В. Баранцев, И. Б. Бурдонов, А. В. Демаков, С. В. Зеленов, А. С. Косачев, В. В. Кулямин, В. А. Омельченко, Н. В. Пакулин, А. К. Петренко, А. В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. Труды ИСП РАН, 5:121–156, 2004.

[7] B. Meyer. Applying 'Design by Contract'. IEEE Computer, 25(10):40–51, 1992.

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

[9] A. P. Mathur. Foundations of Software Testing. Copymat Services, 2006.