Паттерны проектирования тестовых сценариев[1]

В.С. Мутилин, mutilin@ispras.ru

Аннотация. Рассматриваются вопросы использования типовых решений (паттернов проектирования) для построения тестовых программ, основанных на обобщенных моделях тестируемых систем в форме неявно заданных конечных автоматов. В качестве базовой технологии применяется технология UniTesK [1].

1. Введение

В тестировании на основе моделей модели используются для нескольких целей: для проверки соответствия тестируемой системы требованиям, представленным в модели; для задания покрытия; для генерации тестовых данных. Нас, в первую очередь, будет интересовать последний случай – применение моделей для генерации тестовых данных.

При тестировании на основе моделей широко распространенным подходом к генерации тестовых данных является представление системы в виде конечного автомата и генерация тестовой последовательности на основе его обхода [2, 3, 4]. Авторы, использующие конечные автоматы для тестирования, отмечают, что это позволяет повысить качество тестирования, улучшить сопровождаемость тестов [5, 6]. В работе [7] отмечается, что использование конечных автоматов дает возможность получать последовательности, которые было бы сложно получить при применении других подходов. В отличие от ручного тестирования и тестирования с автоматизированным прогоном тестов, при использовании автоматов тестовая последовательность строится автоматически. В случайных тестовых последовательностях сложно управлять тестами, направлять их на проверку определенных требований. В результате часть важных требований может быть не проверена вовсе.

В данной работе в качестве базовой технологии рассматривается технология UniTesK [1]. В UniTesK для генерации тестовых данных используются модели конечных автоматов, задаваемые в тестовых сценариях. Тестовая последовательность получается автоматически в результате обхода такого автомата. Для проверки требований и задания покрытия используются формальные спецификации. Технология UniTesK позволяет использовать

[1] Работа поддержана грантом РФФИ (05-01-999).

только тестовые сценарии без спецификаций; при этом проверка требований и оценка покрытия может производиться в тестовом сценарии. Паттерны, описываемые в статье, могут применяться для разработки тестовых сценариев и без использования спецификаций. Однако обычно применение технологии предполагает наличие спецификации системы, и поэтому мы будем считать, что для задания требований используется спецификация, определяющая модельное состояние. Заметим лишь, что в случае отсутствия спецификации в паттернах вместо модельного состояния следует оперировать состоянием реализации.

Практически для всех методов, основанных на конечных автоматах, увеличение количества состояний приводит к увеличению длины тестовой последовательности и времени работы тестов. Поэтому на практике для методов характерны ограничения на приемлемое количество состояний, при котором тесты работают не слишком долго. Так, для метода, описанного в [2], приемлемым оказывается несколько десятков состояний. Для технологии UniTesK приемлемое количество состояний гораздо больше – несколько сотен.

Опыт показывает, что размер спецификации составляет четвертую-пятую часть размера реализации, а в некоторых случаях может достигать размера реализации. Поэтому число модельных состояний, определяемых спецификацией, часто оказывается слишком большим для того, чтобы использовать их в качестве состояний конечного автомата. Кроме того, это может оказаться и вовсе невозможным, если число состояний модели бесконечно.

Для борьбы с разрастанием числа состояний в технологиях, в которых для построения тестовых последовательностей используются конечные автоматы, часто используется обобщение состояний, представляющее собой разбиение состояний модели на классы эквивалентности. В работе [6] такие классы называются гиперсостояниями. В технологии UniTesK обобщение состояний задается в тестовом сценарии с помощью функции, возвращающей обобщенное состояние на основе состояния модели.

Для использования обобщенного состояния для построения тестовой последовательности необходимо выполнение требований накладываемых обходчиком, который осуществляет обход конечного автомата. Одно из основных требований обходчиков – это присутствие детерминированности в той или иной степени: просто детерминированности, наличия детерминированного сильно связного покрывающего подавтомата или сильной дельта-связности.

Таким образом, выбор обобщенного состояния в тестовом сценарии – это поиск компромисса между количеством состояний, возможностями обходчика, использующего это состояние для построения тестовой последовательности, и разнообразием состояний, т.е. возможностью покрыть в процессе обхода требуемые тестовые ситуации.

В следующем разделе будет показано, что разработка тестового сценария по технологии UniTesK – это сложная задача, и автоматизировать ее не удается. Процесс разработки затрудняется тем обстоятельством, что количество

состояний модели велико, а порой и бесконечно. Построение автомата приемлемых размеров не гарантируется; кроме того, не для всех моделей можно построить автомат, удовлетворяющий требованиям обходчиков. Поскольку автомат задается неявно, т.е. его окончательный вид определяется только в процессе обхода, проверка требований обходчика затруднительна до запуска тестов, что еще больше усложняет задачу.

В данной статье предложены паттерны проектирования, использование которых позволяется упростить разработку тестовых сценариев. Паттерны, которые рассматриваются в статье, во многом схожи с паттернами, предложенными Кристофером Александером [8] для проектирования зданий. По его словам, «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново». Еще больше описываемые паттерны схожи с паттернами проектирования объектно-ориентированных программ [9]. Так же, как и для паттернов объектно-ориентированного проектирования, при описании паттернов проектирования тестовых сценариев упор делается на решение, а не на проблему. Паттерны описываются без использования формального представления. На данном этапе важно исследовать пространство паттернов, а не формализовать его. Так же, как и для паттернов Александера, возможно последовательное применение описываемых паттернов, и в большинстве случаев это приводит к хорошему решению. Однако, в отличие от паттернов Александера, описываемые паттерны не составляют «полный» набор, из которого можно вывести пошаговые инструкции по созданию тестового сценария.

Паттерны получены на основе изучения существующих тестовых сценариев, написанных с использованием технологий KVEST и UniTesK. Технология UniTesK появилась, как развитие технологии KVEST, разработанной для тестирования ядра операционной системы Nortel Networks [10]. UniTesK в течение многих лет успешно применяется для тестирования различного программного обеспечения. Для нахождения паттернов было проанализировано около трехсот тестовых сценариев. Статистика показывает, что найденные паттерны используются в 80% тестовых сценариев, и лишь в 20% случаев требуются дополнительные соображения.

Паттерны позволяют передать опыт, накопленный разработчиками тестовых сценариев. Опытный разработчик, вместо решения каждой задачи с нуля, старается повторно пользоваться теми решениями, которые оказались удачными в прошлом. Отыскав хорошее решение, он будет прибегать к нему снова и снова. Разработчик, знакомый с паттернами, может сразу применять их к решению новой задачи, не пытаясь каждый раз «изобретать велосипед».

2. Процесс разработки тестового сценария

В технологии UniTesK задание конечного автомата, на основе которого строится тестовая последовательность, вынесено в отдельный компонент –

тестовый сценарий [11]. В тестовом сценарии задаются обобщенные состояния и итерации параметров методов. Обобщенные состояния определяют состояния конечного автомата и задаются с помощью функции, возвращающей обобщенное состояние на основе состояния модели. Таким образом, состояния модели разбиваются на классы с одинаковым обобщенным состоянием. Обобщенные состояния позволяют уменьшить количество состояний автомата. Итерации параметров методов задают перебор параметров, с которыми следует вызвать методы, определяя тем самым возможные переходы автомата. Однако при таком задании переходов состояние после выполнения метода неизвестно до вызова метода. Поэтому окончательный вид автомата определяется только в процессе обхода автомата.

На основе описания обобщенного состояния и итераций параметров методов специальный компонент тестовой системы, называемый обходчиком, строит обход переходов автомата, получая тестовую последовательность. В процессе обхода, выполняя вызовы методов, обходчик определяет окончания переходов автомата.

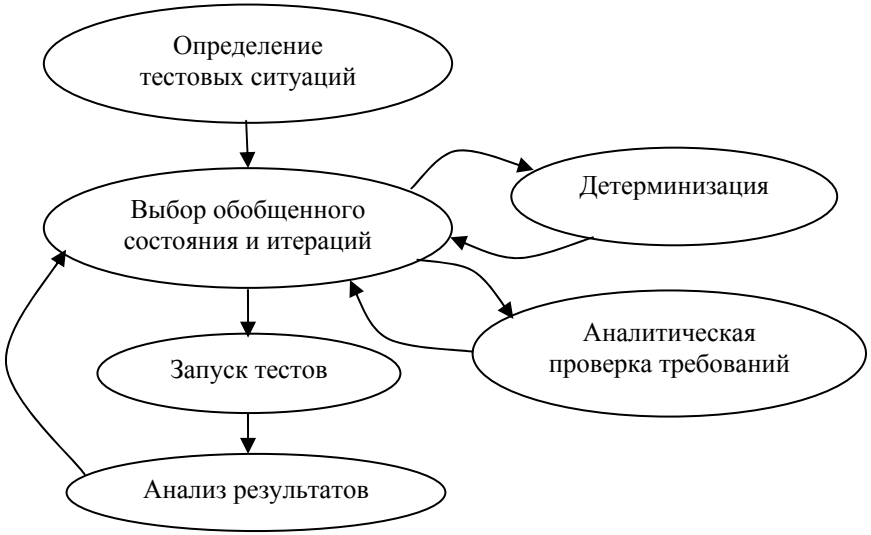

Рис. 1. Процесс разработки тестового сценария

Процесс разработки тестового сценария начинается с определения тестовых ситуаций (рис. 1). Тестовая ситуация определяет множество состояний, метод, и наборы параметров при которых она может быть покрыта. Тестовая ситуация является достижимой в данном состоянии, если в этом состоянии существует такой набор параметров, что метод, вызванный с этим набором параметров в данном состоянии, покрывает эту ситуацию. Тестовые ситуации позволяют определить, какие состояния являются «интересными» для тестирования, а

также судить о разнообразности обобщенных состояний. Обобщенные состояния являются разнообразными, если они позволяют покрыть выбранные тестовые ситуации.

Основным источником тестовых ситуаций является покрытие, определенное в спецификациях, так как именно оно является основным критерием завершенности тестирования в технологии UniTesK. При определении тестовых ситуаций могут учитываться также дополнительные соображения, не нашедшие отражения при определении покрытия в спецификации.

Например, если в качестве модельного состояния используются деревья, то дополнительным соображением может быть наличие разнообразных деревьев: высоких, широких, сбалансированных и т.д. Подобные свойства сложно отображать в спецификации.

Другой пример дополнительного соображения – перебор значений свойства, не определяющего внешнее поведение реализации и, соответственно, не отраженнного в спецификации. Однако внутреннее поведение реализации может зависеть от этого свойства, используемого для оптимизации.

На следующем шаге процесса разработки тестового сценария происходит выбор обобщенного состояния и итераций параметров методов. Выбор обобщенного состояния является непростой задачей Это поиск компромисса между количеством состояний, их разнообразием и возможностями обходчика, использующего данное состояние для построения тестовой последовательности.

Количество состояний непосредственным образом отражается на длине тестовой последовательности и времени работы тестов. Приемлемое количество состояний зависит от скорости работы системы. Опыт показывает, что для быстрых систем приемлемое количество состояний – несколько сотен, для медленных – несколько десятков.

Обобщенные состояния должны быть достаточно разнообразными, чтобы обход всех переходов автомата покрывал выбранные тестовые ситуации. Для этого достаточно в качестве обобщения взять разбиение состояний на группы, в которых достижимы одинаковые наборы тестовых ситуаций. Такое разбиение обычно не приводит к большому количеству состояний, так как количество тестовых ситуаций обычно невелико.

Рассмотрим пример обобщения. На рис. 2 в левой части показаны состояния, у каждого из которых нарисованы возможные переходы. Переходы, соответствующие разным тестовым ситуациям, показаны стрелками с разной штриховкой. Обобщением является объединение состояний с одинаковым набором штриховок выходящих стрелок. Очевидно, что обход, покрывающий все переходы автомата, покроет все возможные тестовые ситуации.

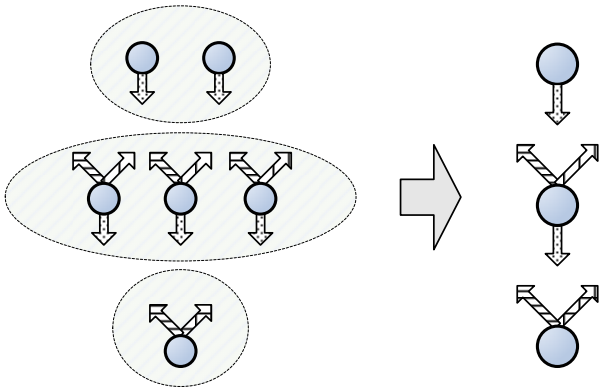

Рис. 2. Пример начального обобщения

Если количество обобщенных состояний слишком велико, то приходится жертвовать разнообразием, укрупняя обобщенные состояния. Укрупнение обобщенных состояний может приводить к тому, что в двух модельных состояниях одного обобщенного состояния, достижимыми являются разные множества тестовых ситуаций. Может получиться так, что в одном модельном состоянии тестовая ситуация достижима, а в другом – нет. Однако можно надеяться, что в процессе обхода будут получены оба состояния, и все тестовые ситуации будут все равно покрыты.

В ситуации, показанной на рис. 3, обобщение состояний приводит к тому, что не всякий обход автомата покроет переходы, заштрихованные вертикальными линиями, сеточкой и точками. На рисунке эти стрелочки обозначены пунктирной линией.

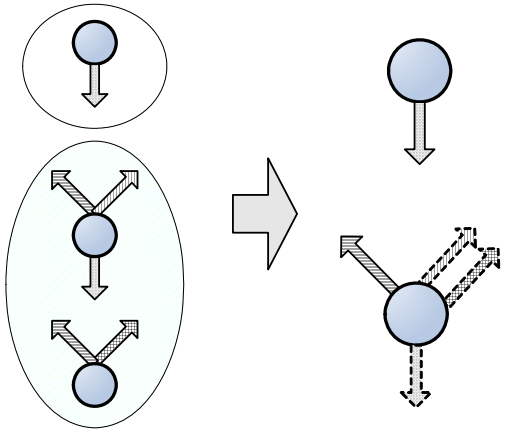

Рис.3. Укрупнение состояний

В ряде случаев достигнуть покрытия, не меняя обобщенного состояния, удается с помощью введения дополнительных сценарных методов или путем разработки дополнительного тестового сценария, возможно, с другим обобщенным состоянием.

На этом же шаге процесса разработки тестового сценария выбираются итерации параметров методов. Итерации задаются в сценарных методах тестового сценария. В общем случае одному тестируемому методу может соответствовать несколько сценарных методов, и один сценарный метод может задавать итерации сразу для нескольких методов, а также последовательность вызовов. В наиболее частом случае каждому методу соответствует один сценарный метод. Итерация параметров может быть выполнена с фильтрацией по критерию покрытия. В этом случае итерируются идентификаторы элементов покрытия (тестовых ситуаций), и для каждого элемента подбирается набор параметров, покрывающий выбранный элемент. Фильтрацию также называют обобщением переходов, так как она разбивает параметры и состояния на группы, соответствующие разным элементам тестового покрытия. Фильтрация может использоваться как для сокращения количества переходов, так и для детерминизации автомата.

После начального выбора обобщенного состояния и итераций следует проверить выполнение требований обходчика. На данный момент в инструментах UniTesK есть пять видов обходчиков:

1. базовый обходчик;
2. обходчик детерминированных автоматов;
3. обходчик детерминированных автоматов с функцией сброса;
4. обходчик автоматов, имеющих детерминированный, сильно связный покрывающий подавтомат;
5. обходчик сильно дельта-связных автоматов.

Во всех этих обходчиках требуется, чтобы число состояний и переходов было конечно. В базовом обходчике не используется обобщенное состояние; считается, что у автомата имеется одно единственное состояние, и, таким образом, не накладываются какие-либо дополнительные ограничения. В обходчике детерминированных автоматов требуется, чтобы автомат, описываемый сценарием, был детерминированным и сильно связным. В обходчике детерминированных автоматов с функцией сброса требуется детерминированность автомата, а сильная связность обеспечивается с помощью функции сброса, задаваемой разработчиком сценария. Четвертый обходчик может работать с недетерминированными автоматами, однако в нем требуется, чтобы существовал детерминированный сильно связный подавтомат, который содержит все состояния исходного автомата. В последнем обходчике требуется, чтобы для любых двух состояний автомата существовала адаптивная тестовая последовательность, ведущая из одного состояния в другое. Заметим, что этому требованию заведомо удовлетворяют автоматы, содержащие детерминированный сильно связный покрывающий подавтомат.

Для удовлетворения требований обходчиков можно использовать следующие методы:

1. дробление состояний;
2. введение связующих переходов;
3. обобщение переходов.

Метод дробления состояний описан в [12] (в статье ему соответствуют алгоритмы 1 и 2 построения дельта детерминированного и вполне определенного фактор-графа). Здесь он называется методом дробления, так как его применение для заданного начального обобщения состояний и переходов приводит к разбиению состояний, принадлежащих одному обобщенному состоянию, на несколько непересекающихся групп, которые образуют новые обобщенные состояния (рис. 4). Использование метода дробления позволяет во многих случаях достичь детерминированности автомата. Многие предлагаемые в данной статье паттерны могут быть получены применением метода дробления. Однако для применения метода требуется представление модели в виде конечного автомата. Преставление модели в виде конечного автомата зачастую бывает слишком сложным из-за слишком большого числа получающихся состояний и переходов. Кроме того метод применим не для всех автоматов.

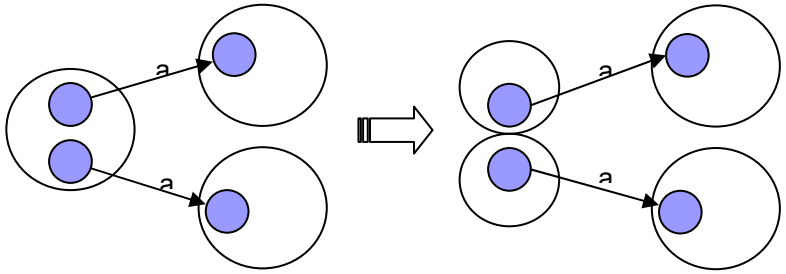

Рис. 4. Метод дробления

Метод дробления может построить автомат, удовлетворяющий требованиям обходчика, однако число его состояний может быть слишком велико. Тогда для уменьшения количества состояний можно жертвовать разнообразностью состояний.

Метод связующих переходов используется совместно с обходчиком автоматов, имеющих детерминированный, сильно связный покрывающий подавтомат. К уже имеющемуся недетерминированному переходу добавляется дополнительный детерминированный переход, гарантированно переводящий систему в заданное состояние. Например, если метод в зависимости от некоторых свойств переводит систему в два разных состояния, можно ввести дополнительный переход (сценарный метод), при выполнении которого перед вызовом метода свойства устанавливаются таким образом, чтобы перейти в требуемое состояние. На рис. 5 показан метод *add*, который в зависимости от

некоторых свойств либо изменяет, либо не изменяет состояние. Для обеспечения существования детерминированного подграфа вводится сценарный метод *add2*, гарантированно переводящий систему в новое состояние.

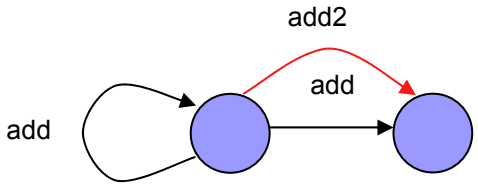

Рис. 5. Метод введения связующих переходов

Метод обобщения переходов состоит в использовании фильтрации при итерации параметров методов.

Например, если при тестировании метода добавления элемента во множество целых чисел в качестве обобщенного состояния выбирать размер множества, то простая итерация параметров приводит к недетерминированному автомату. Обобщение переходов по ветвям функциональности «добавляемый элемент есть во множестве», «добавляемого элемента нет в множестве» позволяет получить детерминированный автомат (см. рис. 6).

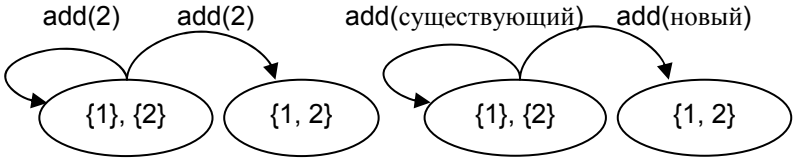

Рис. 6. Метод обобщения переходов

До запуска тестов желательно проверить выполнение требований обходчика. Однако данная проверка затруднительна, поскольку до запуска тестов сложно представить точный вид автомата. В сценарии задаются лишь состояние автомата и итерации переходов, а результирующие состояния переходов определяются только во время работы теста. С одной стороны, неявное задание автомата позволяет упростить задание автомата в тестовом сценарии, но, с другой стороны, затрудняет проверку требований обходчика без запуска тестов.

Кроме того, при неоднозначности в спецификации возможных пост-состояний тестирование разных реализаций может приводить к построению разных автоматов, отличающихся результирующими состояниями переходов.

В процессе запуска тестов определяется точный вид автомата. Результатами запуска тестового сценария является:

1. достигнутое покрытие спецификаций (тестовых ситуаций);
2. нарушение или выполнение требований обходчика;
3. количество состояний, переходов и время работы.

Таким образом, процесс построения тестового сценария оказывается сложным. Процесс затрудняется тем обстоятельством, что количество состояний модели велико, а порой и бесконечно. Построение автомата приемлемых размеров не гарантируется; кроме того, не для всех моделей можно построить автомат, удовлетворяющий требованиям обходчиков. Поскольку окончательный вид автомата определяется в процессе обхода, проверка требований обходчика до запуска тестов затруднительна, что еще больше усложняет задачу.

3. Понятие паттерна

Паттерны представляют собой удачные решения часто встречающихся задач. Там, где использование процесса разработки тестового сценария приводит к хорошему результату, паттерны позволяют повторно использовать полученные результаты. Там, где использование процесса затруднительно, знание паттернов позволяет выделить части, к которым они применимы. Паттерны позволяют использовать инструментальную поддержку.

Паттерны, описываемые в данной статье, получены на основе анализа более чем десятилетнего опыта разработки тестов ИСП РАН [13] в различных проектах:

- Nortel Networks (ядро ОС);
- Luxoft (банковское приложение);
- Intel (стандартная библиотека Java);
- Microsoft Research (протокол IPv6);
- Вымпелком (детализация по счетам);
- НИИ системных исследований РАН (ОС 2000);
- Persistent (Service Data Objects, реализация BEA).

Было проанализировано около трехсот тестовых сценариев. В результате анализа проектов выделено десять наиболее распространенных паттернов. Названия выделенных паттернов, их краткая характеристика и статистика использования показаны в таблице 1.

Наиболее широкое применение имеет группа паттернов с размером структуры данных в качестве обобщенного состояния: *длина списка*, *размер множества*, *размер отображения*, *число вершин дерева*. Эти паттерны используются в более чем половине случаев применения паттернов.

Название	Краткая характеристика	Статистика использования	
Длина списка	В качестве обобщенного состояния выбирается длина списка	13%	41%
Размер множества	В качестве обобщенного состояния выбирается размер множества	17%	
Размер отображения	В качестве обобщенного состояния выбирается размер отображения	8%	
Число вершин дерева	В качестве обобщенного состояния выбирается число элементов дерева	3%	
Декартово произведение	В качестве обобщенного состояния выбирается декартово произведение других обобщенных состояний	18%	39%
Выделение элементов	Паттерн основан на выделении элементов обладающих некоторыми свойствами	5%	
Единственное состояние	В качестве обобщенного состояния выбирается одно единственное состояние	10%	
Мульти-множество чисел детей	В качестве обобщенного состояния выбирается мультимножество, элементами которого являются числа – количество непосредственных детей для каждой вершины дерева	3%	
Код дерева	В качестве обобщенного состояния выбирается код дерева, однозначно определяющий его структуру	2%	
Среднее состояние	Все промежуточные состояния объединяются в одно обобщенное состояние	1%	
? (без паттерна)		20%	

Таблица 1. Паттерны проектирования

В большой части паттернов явным образом определяется обобщенное состояние; это такие паттерны, как длина списка, размер множества, размер отображения, число вершин дерева, единственное состояние, мультимножество чисел детей, код дерева, среднее состояние. Оставшиеся два паттерна декартово произведение и выделение элементов явным образом состояние не определяют, а используются совместно с другими паттернами.

Паттерны покрывают большинство распространенных структур данных: списки, множества, отображения, деревья. Для объединения нескольких

обобщенных состояний, соответствующих разным структурам данных, используется паттерн *декартово произведение*. В паттерне *выделение элементов* учитываются свойства элементов, так как в остальных паттернах при выборе обобщенного состояния предполагается, что тестовые ситуации от свойств элементов не зависят.

Описание каждого паттерна состоит из следующих частей:

1. название;
2. краткое описание;
3. область применения;
4. обобщенное состояние;
5. итерация параметров методов;
6. примеры;
7. совместное использование;
8. использование в проектах.

Название служит для краткого именования паттерна. Область применения описывает ситуации, в которых применим описываемый паттерн, а также известные расширения паттерна. В части примеров приводятся простые и наглядные примеры применения паттернов; в данной статье примеры приводятся на расширении языка Java [14]. В части совместного использования приводятся паттерны, с которыми можно удачно использовать описываемый паттерн. Примеры использования паттерна в проанализированных проектах описываются в части использования в проектах.

4. Описание паттернов

В этом разделе приводятся подробные описания четырех паттернов: длина списка, размер множества, декартово произведение и мультимножество чисел детей.

Паттерн *размер отображения* очень похож на паттерн *размер множества* применительно к множеству ключей отображения, отличие состоит только в дополнительной итерации значений отображения при добавлении элементов.

Паттерны *число вершин дерева* и *код дерева* – это упрощенные версии паттерна *мультимножество чисел детей*. Первый паттерн не покрывает деревья разнообразной структуры, однако позволяет тестировать деревья с большим количеством вершин. Второй паттерн позволяет протестировать деревья всевозможных структур, однако применим лишь для небольшого числа вершин.

Паттерн *среднее состояние* позволяет тестировать достижение максимального количества элементов в различных структурах данных (списках, множествах, отображениях, деревьях) за счет объединения всех промежуточных состояний между максимальным и минимальным состояниями в одно обобщенное состояние.

Паттерн *единственное состояние* применяется для тестирования методов, не зависящих от состояния, и для построения простого тестового сценария, в котором разнообразие покрываемых тестовых ситуаций полностью зависит от выбора итераций параметров методов.

Паттерн *выделение элементов* основан на выделении элементов, обладающих некоторыми свойствами. Он применяется совместно с другими паттернами: *длина списка*, *размер множества*, *размер отображения*, *число вершин дерева* и другими. Цель применения – покрытие тестовых ситуаций, зависящих от свойств элементов. Данный паттерн может применяться совместно с паттерном *декартово произведение* с целью достижения детерминизма.

4.1. Длина списка

4.1.1. Краткое описание

В качестве обобщенного состояния выбирается длина списка, присутствующего в модельном состоянии или сконструированного на его основе.

4.1.2. Область применения

Применяется для присутствующих в модельном состоянии списка, очереди, стека и любых других перечислений. Использование паттерна предполагает, что тестовые ситуации не зависят от элементов списка. Для покрытия разнообразных элементов списка предполагается использование дополнительных технических приемов: дополнительных сценарных методов, паттерна *выделение элементов*.

4.1.3. Обобщенное состояние

Используется целочисленное или натуральное состояние 0, 1, 2, … . Функция вычисления обобщенного состояния возвращает количество элементов списка. Для обеспечения конечности обобщенных состояний вводится ограничение на количество состояний, задаваемое как параметр сценария.

4.1.4. Итерация параметров методов

Может использоваться простая итерация элементов списка, их конструирование. Обязательно присутствие методов, добавляющих элементы в список, увеличивающих длину списка, а также методов, удаляющих один или несколько элементов списка.

Допустимо использовать обобщение переходов. Обобщение в простом случае существенно сокращает количество переходов. В более сложных случаях обобщение позволяет побороть недетерминизм, возникающий при зависимости методов добавления и удаления от свойств элементов списка.

4.1.5. Примеры

```
List modelList;

// Метод добавляет элемент e в список.
void add(Integer e);

// Метод удаляет элемент по индексу index из списка.
// Если индекс выходит за границы списка,
// вырабатывается исключение IndexOutOfBoundsException.
void remove(int index) throws IndexOutOfBoundsException;
```

Тестовые ситуации для метода *add*:

1. список пуст;
2. список не пуст.

Тестовые ситуации для метода *remove*:

1. индекс *index* отсутствует в списке;
2. индекс *index* есть в списке:
   a. список пуст;
   b. список содержит единственный элемент;
   c. список содержит больше одного элемента.

Обобщенное состояние – *IntGenState*, параметр конструктора – длина списка *modelList: modelList.size()*. Для ограничения количества состояний в сценарий добавляется переменная *int maxSize*.

Для метода *add* с использованием конструкции *iterate* итерируются элементы списка – целые числа; итерация происходит, только если длина списка не превышает *maxSize*. Для метода *remove* итерируются индексы списка:

```
scenario boolean add() {
    //objectUnderTest – модель, содержащая спецификационные методы
    //add и remove
    if(objectUnderTest.modelList.size()<maxSize) {
        iterate(int i=0; i<10; i++; ) {
            //вызов спецификационного метода add
            objectUnderTest.add(new Integer(i));
        }
    }
    return true;
}

scenario boolean remove() {
    iterate(int i=-1; i<=objectUnderTest.modelList.size(); i++;) {
        //вызов спецификационного метода remove
        objectUnderTest.remove(i);
    }
    return true;
}
```

4.1.6. Совместное использование

Используется совместно с паттерном *выделение элементов*. Для тестирования списков при максимальном заполнении рекомендуется использовать паттерн *среднее состояние*.

4.1.7. Использование в проектах

Количество файлов в директории; количество выделенных идентификаторов; количество выделенных семафоров; количество элементов меню; количество слушателей сообщений интерфейса (*action listeners*); количество элементов списка (*List*); размер списка ожидающих обработки операций *send*, *receive*; количество сообщений в очереди; количество синхронизированных потоков (*joined threads*); количество потоков, которые могут быть синхронизированы (*joinable*); количество отмененных (*canceled*), заблокированных (*blocked*), отсоединенных (*detached*) потоков.

4.2. Размер множества

4.2.1. Краткое описание

В качестве обобщенного состояния выбирается размер множества, присутствующего в модельном состоянии или сконструированного на его основе.

4.2.2. Область применения

Применяется для множества, присутствующего в модельном состоянии, или сконструированного на его основе. Использование паттерна предполагает, что тестовые ситуации не зависят от элементов множества. Для покрытия ситуаций, зависящих от элементов множества, предполагается использование дополнительных технических приемов: дополнительных сценарных методов, паттерна *выделение элементов*.

4.2.3. Обобщенное состояние

Используется целочисленное или натуральное состояние 0, 1, 2, … . Функция вычисления обобщенного состояния возвращает количество элементов множества. Для обеспечения конечности обобщенных состояний вводится параметр сценария, который может задавать ограничение как на количество состояний, так и на количество разнообразных элементов, итерируемых в сценарных методах.

4.2.4. Итерация параметров методов

Для итерации параметров предпочтительнее использовать обобщение параметров по ветвям функциональности, соответствующим наличию или отсутствию во множестве добавляемого или удаляемого элемента.

Для обеспечения более детального покрытия, нежели покрытие ветвей, может использоваться сочетание обобщений по ветвям и по более детальному покрытию. В случае недетерминированности обобщенных переходов по более детальному покрытию обобщение по ветвям обеспечивает существование детерминированного подавтомата и позволяет использовать обходчик детерминированных подавтоматов.

В сочетании с обобщением параметров, обеспечивающих существование детерминированного подавтомата, может использоваться и простая итерация параметров. Такая итерация, как правило, обеспечивает большее число различных переходов и позволяет покрыть более разнообразные тестовые ситуации, например, тестовые ситуации, зависящие от элементов множества.

В итерациях обязательно присутствие методов, добавляющих элементы во множество, увеличивающих размер множества, а также методов, удаляющих один или несколько элементов множества.

4.2.5. Примеры

```
Set modelSet;

// Метод добавляет элемент e в множество.
void add(Integer e);

// Метод удаляет элемент e из множества.
// Если элемент присутствовал во множестве, возвращает true.
// Иначе false.
boolean remove(Integer e);
```

Тестовые ситуации для метода *add*:

1. множество пусто;
2. множество не пусто:
    a. добавляемый элемент присутствует в множестве;
    b. добавляемый элемент отсутствует в множестве.

Тестовые ситуации для метода *remove*:

1. множество пусто;
2. множество содержит единственный элемент:
    a. удаляемый элемент присутствует во множестве;
    b. удаляемый элемент отсутствует во множестве;
3. множество содержит более одного элемента:
    a. удаляемый элемент присутствует во множестве;
    b. удаляемый элемент отсутствует во множестве.

Обобщенное состояние – *IntGenState*, параметр конструктора – размер множества *modelSet: modelSet.size()*. Для ограничения количества состояний в сценарий добавляется переменная *int maxSize*.

Для методов *add* и *remove* итерируются ветви функциональности, соответствующие отсутствию или присутствию элемента во множестве. Для каждой ветви перебираются элементы множества до тех пор, пока не будет

найден элемент, попадающий в выбранную ветвь функциональности. Для ограничения количества обобщенных состояний итерация для метода *add* происходит, только если размер множества не превышает *maxSize*.

```
scenario boolean add() {
    //objectUnderTest–модель, содержащая спецификационные
методы
    //add и remove
    if(objectUnderTest.modelSet.size()<maxSize) {
        // 0 – элемент отсутствует во множестве
        // 1 – элемент присутствует во множестве
        iterate(int b=0; b<2; b++; ) {
            //поиск элемента, удовлетворяющего заданной ветви
            for(int i=0; i<10; i++) {
                Integer e = new Integer(i);
                if(b==0 &&
!objectUnderTest.modelSet.contains(e)
                  || b==1 &&
objectUnderTest.modelSet.contains(e)) {
                    //вызов спецификационного метода add
                    objectUnderTest.add(e);
                    break;
                }
            }
        }
    }
    return true;
}

scenario boolean remove() {
    iterate(int b=0; b<2; b++; ) {
        //поиск элемента, удовлетворяющего заданной ветви
        for(int i=0; i<10; i++) {
            Integer e = new Integer(i);
            if(b==0 && !objectUnderTest.modelSet.contains(e)) {
              || b==1 && objectUnderTest.modelSet.contains(e))
{
                //вызов спецификационного метода remove
                objectUnderTest.remove(e);
                break;
            }
        }
    }
    return true;
}
```

4.2.6. Совместное использование

Используется совместно с паттерном *выделение элементов*. Для тестирования максимального заполнения множества рекомендуется использовать паттерн *среднее состояние*.

4.2.7. Использование в проектах

Размер множества свободных идентификаторов; размер множества ресурсов разделяемой памяти; размер множества выделенных буферов (*allocated buffers*); размер пула выделенных ресурсов; размер пула выделенных процессов; размер пула выделенных семафоров; размер множества активных RMI-объектов; размер множества идентификаторов активных RMI-объектов; размер множества дескрипторов очереди сообщений.

4.3. Декартово произведение

4.3.1. Краткое описание

В качестве обобщенного состояния выбирается произведение других обобщенных состояний.

4.3.2. Область применения

Применяется совместно с другими паттернами. Позволяет объединять тестовые ситуации, задаваемые элементами декартова произведения. При увеличении количества элементов декартова произведения количество состояний резко возрастает. Если между состояниями нет зависимостей, то количество состояний произведения есть произведение количеств состояний, задаваемых каждым элементом. Опыт показывает, что допустимо произведение лишь небольшого числа элементов, в пределах десяти.

4.3.3. Обобщенное состояние

Тип состояния зависит от типов элементов произведения. В общем случае можно пользоваться *PairComplexGenState* и *ListComplexGenState*, конструируемыми из пары и списка обобщенных состояний соответственно. Для произведения целочисленных состояний можно пользоваться классами обобщенных состояний *IntPairGenState*, *IntTripleGenState*, *IntListGenState*.

4.3.4. Итерация параметров методов

Вообще говоря, итерация параметров зависит от элементов произведения и в каждом случае выбирается по-разному. Однако существует два достаточно распространенных случая.

Первый случай – произведение одинаковых обобщенных состояний, являющихся обобщением одинаковых структур. В этом случае к итерации параметров для каждого метода добавляется итерация по структурам, составляющим элементы произведения.

Второй случай – произведение разных обобщений одной и той же структуры. В этом случае итерации в ряде случаев можно оставить неизменными.

4.3.5. Примеры

Пример 1. Произведение длин списков.

Спецификация описывает список, такой же, как в примере для паттерна *Длина списка*.

```
List modelList;

// Метод добавляет элемент e в список.
void add(Integer e);

// Метод удаляет элемент по индексу index из списка.
// Если индекс выходит за границы списка,
// вырабатывается исключение IndexOutOfBoundsException.
void remove(int index) throws IndexOutOfBoundsException;
```

В сценарии заводится массив *ListMediator testLists[]*, в котором хранятся списки, сконструированные для тестирования. Т.е. в этом массиве хранятся те же объекты, что используются для тестирования одного списка (*objectUnderTest*), – медиаторы списков с присоединенными оракулами.

Обобщенное состояние – *IntListGenState*. При конструировании обобщенного состояния производится итерация по элементам массива *testList* и добавляется длина каждого списка *modelList: modelList.size()*.

Так же, как и для тестирования одного списка, вводится ограничение на максимальную длину всех списков *int maxSize.*

Для тестирования методов добавления и удаления в сценарных методах итерируются тестируемые списки, а затем параметры методов, так же, как для одного списка.

```
scenario boolean add() {
 iterate(int i=0; i<=objectUnderTest.testLists.length; i++; )
{
     objectUnderTest = testLists[i];
     //objectUnderTest–модель,содержащая спецификационные методы
     //add и remove
     if(objectUnderTest.modelList.size()<maxSize) {
         iterate(int j=0; j<10; i++; ) {
             //вызов спецификационного метода add
             objectUnderTest.add(new Integer(j));
         }
     }
 }
 return true;
}

scenario boolean remove() {
 iterate(int i=0; i<=objectUnderTest.testLists.length; i++; )
{
     objectUnderTest = testLists[i];
```



```
     iterate(int j=-1;j<=objectUnderTest.modelList.size();j++;) {
         //вызов спецификационного метода remove
         objectUnderTest.remove(j);
     }
 }
 return true;
}
```

Пример 2. Активные идентификаторы.

```
   // Отображение из идентификаторов объектов в статус
объекта.
   // true – объект активный, false – объект неактивный
   Map modelMap;

   // Метод связывает ключ key со значением value.
   // Если ключ присутствовал в отображении,
   // возвращает предыдущее значение, связанное ключом,
   // иначе возвращает null.
   Object put(Integer key, Boolean value);

   // Метод удаляет ключ key из отображения.
   // Возможно удаление только неактивного идентификатора.
   // Возвращает true, если ключ успешно удален
   // или не присутствовал в отображении; иначе возвращает
false.
   boolean remove(Integer key);
```

Тестовые ситуации для метода *put*:

1. отображение пусто;
2. отображение не пусто:
    a. добавляемый идентификатор присутствует в отображении:
        i. присутствующий идентификатор активен:
            1. добавляемый идентификатор активен;
            2. добавляемый идентификатор не активен;
        ii. присутствующий идентификатор неактивен:
            1. добавляемый идентификатор активен;
            2. добавляемый идентификатор не активен;
    b. добавляемый идентификатор отсутствует в отображении:
        i. добавляемый идентификатор активен;
        ii. добавляемый идентификатор не активен.

Тестовые ситуации для метода *remove*:

1. отображение пусто;
2. отображение содержит единственный идентификатор:
    a. удаляемый идентификатор присутствует в отображении:
        i. удаляемый идентификатор активен;
        ii. удаляемый идентификатор не активен;
    b. удаляемый идентификатор отсутствует в отображении:

i. удаляемый идентификатор активен;
ii. удаляемый идентификатор не активен;

3. отображение содержит более одного идентификатора:
   a. удаляемый идентификатор присутствует в отображении:
      i. удаляемый идентификатор активен;
      ii. удаляемый идентификатор не активен;
   b. удаляемый идентификатор отсутствует в отображении:
      i. удаляемый идентификатор активен;
      ii. удаляемый идентификатор не активен.

Обобщенное состояние – *IntPairGenState*, параметры конструктора – размер отображения и количество активных идентификаторов. Количество активных идентификаторов – это обобщенное состояние, которое получено выделением из отображения элементов, обладающих свойством активности. Таким образом, в этом примере применяются три паттерна: *размер отображения*, *выделение элементов* и *декартово произведение*. Для ограничения количества состояний в сценарий добавляется переменная *int maxSize*.

Для методов *put* и *remove* итерируются ветви функциональности. Для метода *put* выделим шесть ветвей функциональности:

1. отсутствует, добавляем активный;
2. отсутствует, добавляем неактивный;
3. присутствует активный, добавляем активный;
4. присутствует активный, добавляем неактивный;
5. присутствует неактивный, добавляем активный;
6. присутствует неактивный, добавляем неактивный.

Для метода *remove*:

1. отсутствует;
2. присутствует активный;
3. присутствует неактивный.

Для каждой ветви перебираются идентификаторы до тех пор, пока не будет найден идентификатор, попадающий в выбранную ветвь функциональности. Для ограничения количества обобщенных состояний итерация для метода *put* происходит, только если размер отображения не превышает *maxSize*.

```
scenario boolean put() {
   //objectUnderTest – модель, содержащая спецификационные
методы
   //put и remove
   if(objectUnderTest.modelMap.size()<maxSize) {
       //b = 0 – 5 описанные выше
       iterate(int b=0; b<6; b++; ) {
           //поиск параметров, удовлетворяющих заданной ветви
           for(int i=0; i<10; i++) {
               Integer k = new Integer(i);
               for(int j=0; j<2; j++) {
                  boolean v = (j==0)?false:true;
```



```
if(!objectUnderTest.modelMap.containsKey(k)){
                        if(b==0 && v || b==1 && !v) {
                           objectUnderTest.put(k, new Boolean(v));
                           break;
                        }
                    } else {
                        boolean existing =
                         ((Boolean)

objectUnderTest.modelMap.get(k)).booleanValue();
                        if(b==2 && existing && v
                                ||  b==3 && existing && !v
                                ||  b==4 && !existing && v
                                ||  b==5 && !existing && !v) {
                           //вызов спецификационного метода put
                           objectUnderTest.put(k, new Boolean(v));
                            break;
                        }
                    }
                }
            }
        }
    }
    return true;
}

scenario boolean remove() {
    iterate(int b=0; b<3; b++; ) {
        //поиск идентификатора, удовлетворяющего заданной ветви
        for(int i=0; i<10; i++) {
            Integer k = new Integer(i);
            if(b==0 && !objectUnderTest.modelMap.containsKey(k)) {
                objectUnderTest.remove(k);
                break;
            }
            if(objectUnderTest.modelMap.containsKey(k)) {
                boolean existing =
                 ((Boolean)

objectUnderTest.modelMap.get(k)).booleanValue();
                if(b==1 && existing || b==2 && ! existing) {
                    objectUnderTest.remove(k);
                    break;
                }
            }
        }
    }
    return true;
}
```

4.3.6. Совместное использование

Используется совместно с паттерном *выделение элементов*. Для элементов декартова произведения используются другие паттерны.

4.3.7. Использование в проектах

Произведение количества свободных ресурсов и количества выделенных ресурсов; количество свободных элементов в каждом пуле из списка; произведение количества выделенных элементов и множества идентификаторов; произведение количества слушателей и количества уникальных слушателей; количество элементов в списке для каждого списка из набора; произведение типа упорядочивания в отображении и размера отображения; произведение зарегистрированных RMI объектов и количества активных объектов RMI; произведение количества зарегистрированных и количества пересоздаваемых при перезапуске RMI-объектов; произведение количества зарегистрированных и количества экспортированных RMI-объектов; произведение глубины дерева, количества компаний и количества активных компаний; произведение типа доступа к очереди сообщений и ее размера; произведение максимального размера очереди, максимального размера сообщения и размера очереди; произведение количества ждущих вызовов send, receive, максимального размера очереди, размера очереди и типа блокировки; произведение количества заявок, количества заявок ожидающих обработку и количества заявок находящихся в обработке.

4.4. Мультимножество чисел детей

4.4.1. Краткое описание

В качестве обобщенного состояния выбирается мультимножество, элементами которого являются числа – количество непосредственных детей для каждой вершины дерева.

4.4.2. Область применения

Применяется при тестировании программ, для описания поведения которых в качестве состояния спецификации используется дерево. Считается, что в спецификации описаны методы, позволяющие изменять структуру дерева и свойства узлов.

4.4.3. Обобщенное состояние

В качестве обобщенного состояния выбирается мультимножество, элементы которого – количество непосредственных дочерних вершин. Рассмотрим пример.

На рис. 7 показаны два дерева, в которых одна вершина имеет две дочерних, одна имеет одну дочернюю, и две вершины в каждом дереве не имеют дочерних вершин. Таким образом, обоим этим деревьям соответствует одно и тоже обобщенное состояние, мультимножество {0,0,1,2}.

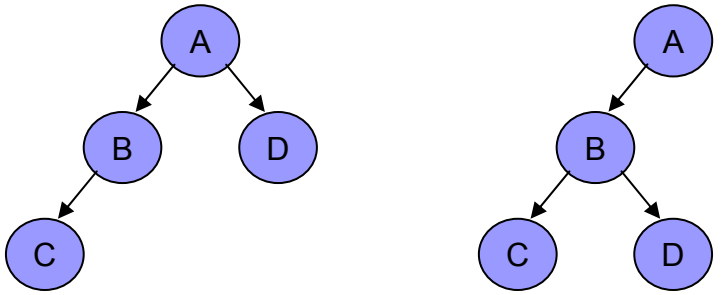

Рис. 7. Примеры деревьев

Результаты подсчета количества обобщенных состояний в зависимости от числа вершин приведены в таблице 2. Как видно, количество обобщенных состояний значительно меньше числа корневых деревьев для того же числа вершин. Это делает данное обобщенное состояние практически пригодным для использования при тестировании.

Число вершин	**1**	**5**	**10**	**15**	**20**	**25**
Число корневых деревьев	1	9	719	87811	12826228	2067174645
Число обобщенных состояний	1	5	30	135	490	1575

Таблица 2. Количество обобщенных состояний

Вместе с тем, данное обобщенное состояние определяет разнообразные виды деревьев. Мультимножества вида {0, …, 0, N}, где N – количество вершин определяют широкие деревья, а мультимножества вида {0, 1, …, 1} определяют высокие деревья.

4.4.4. Итерация параметров методов

Будем считать, что интерфейс содержит методы *add*, *delete* и *createRoot*. У метода *add* имеются два параметра: вершина, к которой нужно добавить ребенка, и добавляемая вершина. Метод требует, чтобы вершина, к которой

добавляется ребенок, существовала. У метода *delete* имеется один параметр – удаляемая вершина. Можно выделить две разновидности метода: метод удаляет только листовые вершины, метод удаляет все поддерево, корнем которого является заданная вершина. Метод *createRoot* создает корневую вершину; вершину можно создать, если дерево пусто.

Описанные таким образом методы оказываются детерминированными, так как по заданному дереву и набору параметров любого метода однозначно определяется вид результирующего дерева.

Легко видеть, что итерация вершин дерева в качестве параметров методов описывает недетерминированный автомат. На рис. 8 показано обобщенное состояние {0, 1, 1} и два соответствующих ему дерева, которые различаются порядком вершин. В этом обобщенном состоянии переход, соответствующий вызову метода *add(C, D)*, может переводить автомат как в обобщенное состояние {0, 1, 1, 1} так и в обобщенное состояние {0, 0, 1, 2} в зависимости от дерева, соответствующего исходному обобщенному состоянию.

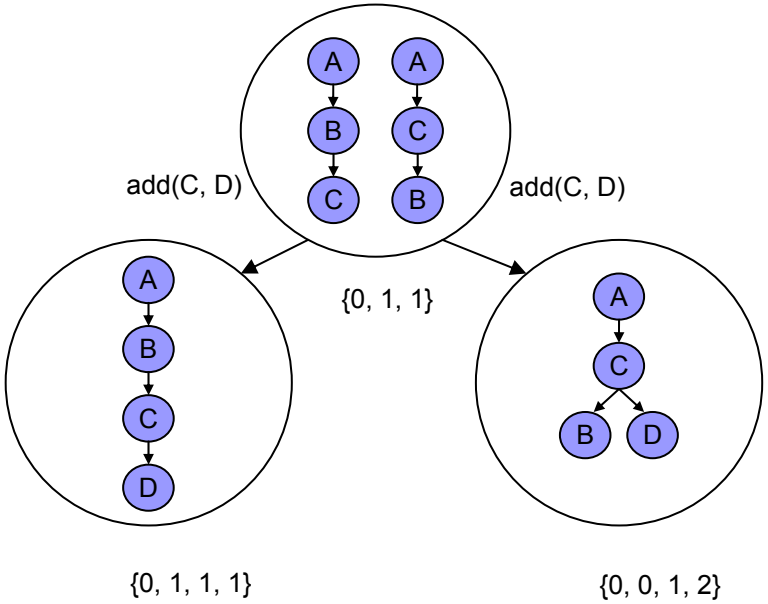

Рис. 8. Простая итерация параметров

Этот недетерминизм легко преодолеть, заменив итерацию вершин итерацией различных элементов мультимножества обобщенного состояния. Элементу мультимножества при выполнении перехода ставится в соответствие произвольная вершина дерева с заданным числом непосредственных детей.

Таким образом, имеем следующие переходы:

- *add(i, A)*;
- *delete(i)*;
- *createRoot(A)*,

где *i* – элемент мультимножества обобщенного состояния, *A* – добавляемая вершина.

Переходы по методу *add* становятся детерминированными, так как мультимножество после перехода определяется однозначно: выбранный элемент мультимножества заменяется большим на единицу, и дополнительно к мультимножеству добавляется нуль (см. рис. 9).

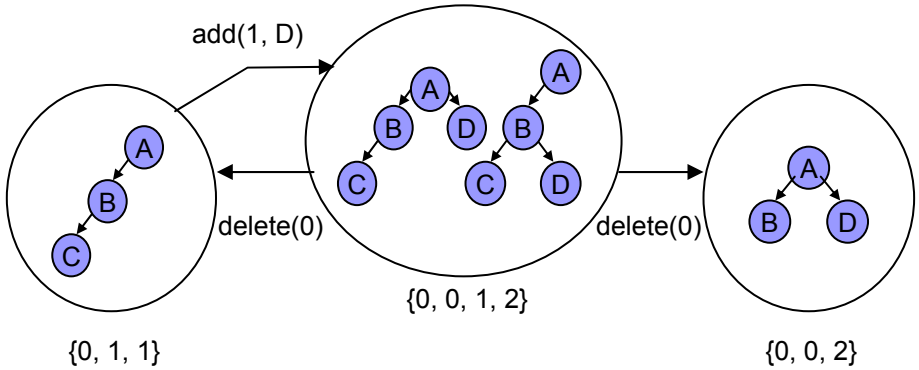

Рис. 9. Итерация элементов мультимножества

Однако переходы по методу *delete* по-прежнему оказываются недетерминированными. В примере, показанном на рис. 9, в состоянии {0, 0, 1, 2} результирующее состояние при удалении вершины без детей зависит от того, сколько было детей у родителя. Если у родителя было два ребенка, то результирующим состоянием является {0, 1, 1}, а если один, то {0, 0, 2}. Таким образом, для данного обобщенного состояния и выбора итераций оказываются неприменимыми обходчики, требующие детерминизма автомата.

Для обходчика сильно дельта-связных автоматов требуется существование адаптивных тестовых последовательностей, связывающих любые два состояния автомата. При нашем выборе состояния и итераций из любого состояния существует тестовая последовательность, ведущая в состояние, которое соответствует пустому дереву. На каждом шаге данной последовательности удаляется вершина без детей, и за число шагов, равное размеру мультимножества, данная последовательность приводит в состояние {}. Далее, используя переходы по методу *add*, можно построить тестовую последовательность, ведущую из начального состояния в любое другое.

Таким образом, к данному выбору итераций применим обходчик сильно дельта-связных автоматов. Обходчик автоматов, имеющих сильно связный детерминированный покрывающий подавтомат, также оказывается применимым при добавлении в сценарий дополнительного перехода, переводящего автомат в состояние {}. В случае, когда метод *delete* удаляет поддеревья, в дополнительном переходе следует удалить корень дерева. Если же метод удаляет только листовые вершины, то следует написать последовательность методов *delete*, в которой на каждом шаге удаляется листовая вершина до тех пор, пока дерево не станет пустым.

На практике часто оказывается, что методы *add* и *delete* зависят не только от структуры дерева. В таком случае для выполнения требований обходчика требуется введение дополнительных переходов, при которых вершина гарантированно добавляется или удаляется. Например, методы могут зависеть от свойств вершины; тогда перед добавлением или удалением свойства вершины следует поменять так, чтобы вершина гарантированно добавилась или удалилась.

4.4.5. Примеры

```
    Tree modelTree;

    // Метод добавляет вершину node к вершине parent,
    // если parent есть в дереве.
    // Если вершины parent нет в дереве, вершина не
добавляется.
    add(Node parent, Node node);

    // Метод удаляет вершину node, если таковая есть в дереве
    // и является листовой.
    // Иначе вершина не удаляется.
    delete(Node node);

    // Метод создает корневую вершину; вершину можно создать,
    // если дерево пусто.
    Node createRoot();
```

Тестовые ситуации для метода *add*:

1. родитель не существует;
2. родитель существует:
    a. родитель не имеет дочерних вершин;
    b. родитель имеет дочерние вершины.

Тестовые ситуации для метода *delete*:

1. вершина не существует;
2. вершина существует:
    a. вершина не имеет дочерних вершин;
    b. вершина имеет дочерние вершины.

Тестовые ситуации для метода *createRoot*:

1. дерево пусто;
2. дерево не пусто.

Обобщенное состояние – мультимножество целых чисел *IntMultisetGenState*. Считаем, что имеется метод, возвращающий вершину дерева по номеру от нуля до числа вершин в дереве *modelTree.getNodeByIndex(int index)*.

```
    IntMultisetGenState genstate = new IntMultisetGenState();
    for(int i=0; i<objectUnderTest.modelTree.size(); i++) {
        Node node =
objectUnderTest.modelTree.getNodeByIndex(i);
        genstate.addElement(node.children().size());
    }
```

Для ограничения количества состояний в сценарий добавляется переменная *int maxSize*.

Для метода *add* итерируем элементы мультимножества, и для каждого элемента подбираем вершину с соответствующим количеством детей. Считаем, что имеется функция, которая возвращает вершину с указанным количеством детей *getNodeWithChildrenSize(int size)*; если таких вершин несколько, возвращается произвольная. Итерация происходит, если количество вершин не превышает *maxSize*.

```
scenario boolean add() {
    //objectUnderTest-модель, содержащая спецификационные
методы
    //add и delete
    if(objectUnderTest.modelTree.size()<maxSize) {
        IntMultisetGenState ms = getGenState();
        //итерация элементов мультимножества
        iterate(IntegerIteratorInterface iter=ms.getIterator();
                !iter.stopIteration(); iter.next(); ) {
            //подбор соответствующей вершины
            Node parent =
       objectUnderTest.modelTree.getNodeWithChildrenSize(iter.value());
            //итерация добавляемых вершин
            iterate(int j=0; j<10; j++) {
                Node node = new Node(j);
                //вызов спецификационного метода add
                objectUnderTest.add(parent, node);
            }
        }
    }
    return true;
}
```

Для метода *delete* итерируются номера вершин дерева, а также вводится дополнительный сценарный метод, гарантированно удаляющий вершину.

```
scenario boolean delete() {
    //итерация вершин дерева
```

```
    iterate(int i=0; i<objectUnderTest.modelTree.size();i++;) {
        Node node =
objectUnderTest.modelTree.getNodeByIndex(i);
        //вызов спецификационного метода delete
        objectUnderTest.delete(node);
    }
    return true;
}

scenario boolean delete_aux() {
    for(int i=0; i<objectUnderTest.modelTree.size(); i++; ) {
        Node node =
objectUnderTest.modelTree.getNodeByIndex(i);
        if(objectUnderTest.modelTree.isLeaf(node)) {
            objectUnderTest.delete(node);
            break;
        }
    }
    return true;
}
```

Для покрытия тестовых ситуаций, при которых не существуют родитель в методе *add* и удаляемая вершина в методе *delete*, вводятся дополнительные сценарные методы.

```
scenario boolean add_neg() {
    if(objectUnderTest.modelTree.size()<maxSize) {
        //итерация родителей – произвольных вершин
        iterate(int i=0; i<10; i++; ) {
            //итерация добавляемых вершин
            iterate(int j=0; j<10; j++) {
                Node parent = new Node(i);
                Node r node = new Node(j);
                //вызов спецификационного метода add
                objectUnderTest.add(parent, node);
            }
        }
    }
    return true;
}

scenario boolean delete_neg() {
    //итерация произвольных вершин
    iterate(int i=0; i<10; i++; ) {
        Node node = new Node (i);
        //вызов спецификационного метода delete
        objectUnderTest.delete(node);
    }
    return true;
}
```


4.4.6. Совместное использование

Используется совместно с паттерном *выделение элементов*. Для тестирования максимального количества вершин в дереве рекомендуется использовать паттерн *среднее состояние*.

4.4.7. Использование в проектах

Паттерн использовался для деревьев, представляющих иерархию компаний, в которой дочерние компании присутствовали как в основной, так и в альтернативной иерархии. В качестве обобщенного состояния выбиралось мультимножество пар: количество дочерних компаний в основной иерархии и количество дочерних компаний в альтернативной иерархии.

Паттерн использовался для тестирования модели данных *Service Data Objects*, представляющей собой дерево со ссылками, в котором можно хранить XML-данные, реляционные данные, EJB. В качестве обобщенного состояния выбиралось мультимножество пар: количество дочерних вершин, количество ссылок на другие вершины.

5. Заключение

Методы, использующие конечные автоматы для тестирования программ, накладывают ограничения на приемлемые размеры автоматов, при которых они применимы на практике. В этом смысле не является исключением и технология UniTesK, в которой приемлемое количество состояний составляет несколько сотен. Для борьбы с разрастанием состояний в UniTesK используется обобщение состояний, задаваемое в тестовом сценарии. Выбор обобщенного состояния в тестовом сценарии – это поиск компромисса между количеством состояний, их разнообразием и возможностями обходчика, использующего это состояние для построения тестовой последовательности.

Процесс разработки тестовых сценариев по технологии UniTesK является сложной задачей. Процесс затрудняется тем обстоятельством, что количество состояний модели велико, а порой и бесконечно. Построение автомата приемлемых размеров не гарантируется; кроме того, не для всех моделей можно построить автомат, удовлетворяющий требованиям обходчиков. Поскольку окончательный вид автомата определяется в процессе обхода, проверка требований обходчика до запуска тестов затруднительна, что еще больше усложняет задачу.

В данной статье предложены паттерны проектирования, позволяющие упростить разработку тестовых сценариев. Паттерны получены в результате анализа более чем десятилетнего опыта разработки тестов ИСП РАН в семи различных проектах. Было проанализировано около трехсот тестовых сценариев. Статистика показывает, что выделенные паттерны используются в 80% тестовых сценариев и лишь в 20% требуются дополнительные соображения.

Паттерны представляют собой удачные решения часто встречающихся задач. Паттерны позволяют повторно использовать полученные результаты, передать опыт разработчиков тестовых сценариев. Знание паттернов дает возможность начинающему разработчику сценариев работать так, как работает эксперт; помогает выделить в модели части, к которым применимы паттерны. Использование паттернов позволяет опираться на библиотечные обобщенные состояния и автоматическую генерацию итераций параметров, поддерживаемые в инструментах тестирования. При использовании паттернов тестировщик может больше сосредоточиться на написании спецификаций к системе, нежели на выборе обобщенного состояния и итераций, которые должны удовлетворять требованиям обходчиков, сложно проверяемым без экспериментов.

Паттерны покрывают большинство распространенных структур данных: списки, множества, отображения, деревья. Такие структуры наиболее часто встречаются при моделировании систем. Это дает уверенность в том, что и в дальнейшем в большинстве случаев можно будет использовать выделенные паттерны.

Литература

1. В.В. Кулямин, А.К. Петренко, А.С. Косачев, И.Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6): 25-43, 2003.
2. Б. Бейзер. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и системы. Питер, 2004.
3. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE, volume 84, pp. 1090-1123, Berlin, Aug 1996.
4. H. Robinson. Graph Theory Techniques in Model-Based Testing. Proceedings of the International Conference on Testing Computer Software, 1999.
5. S. Rosaria, H. Robinson. Applying Models in your Testing Process. Information and Software Technology. Volume 42, Issue 12, Sept. 2000.
6. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. ISSTA 2002, International Symposium on Software Testing and Analysis, July 2002.
7. H. Robinson. Intelligent Test Automation. Software Testing and Quality Engineering, September/October 2000, pp. 24-32.
8. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. A Pattern Language. Oxford University Press, New York, 1977.
9. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Спб: Питер, 2001.
10. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999, pp. 608-621.
11. I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.
12. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Применение конечных автоматов для тестирования программ. Программирование, 26(2):61-73, 2000.

13. http://unitesk.ispras.ru – сайт, посвященный технологии тестирования UniTesK и реализующим ее инструментам.
14. I.B. Bourdonov, A.V. Demakov, A.A. Jarov, A.S. Kossatchev, V.V. Kuliamin, A.K. Petrenko, S.V. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'01. LNCS 2244, pp. 301-307. Springer-Verlag, 2001.