УДК 681.32

В. А. Кирьянчиков, А. С. Романов

Автоматическая генерация тестов объектно-ориентированных программ по диаграммам деятельности

Рассматривается методика автоматической генерации интеграционных тестов на основе диаграмм деятельности языка UML, приводится пример диаграммы деятельности с различными типами действий и ограничений на их выполнение, предлагается алгоритм создания сценария тестирования на основе заданных в диаграмме действий и ограничений, приводится сформированный по диаграмме набор тестов, обеспечивающих проверку корректности работы моделируемой программы.

Тестирование на основе моделей, язык UML, диаграмма деятельности, генерация интеграционных тестов, язык OCL, система ПРОЛОГ

Данная статья является развитием методологии тестирования программ на основе моделей языка UML, примененной в работах [1], [2] для создания методики автоматизированной генерации модульных и интеграционных тестов. Если методика генерации модульных тестов в [1] была основана на использовании диаграмм состояний, а в работе [2] предложена генерация интеграционных тестов на основе диаграмм последовательности, то в настоящей статье предлагается методика формирования интеграционных тестов на основе диаграмм деятельности, моделирующих поведение программы в виде последовательности действий, выполняемых ее различными объектами.

Диаграммы деятельности (ДД) используются для моделирования динамических аспектов системы. Диаграммы этого класса похожи на диаграммы состояний (ДС), поскольку оба вида диаграмм представляют последовательность состояний системы во времени и условия, вызывающие переход из одного состояния в другое. Различие между этими диаграммами состоит в том, что ДС чаще используются для моделирования поведения одного объекта, тогда как ДД используются для моделирования некоторого процесса, в котором участвуют несколько объектов [3]. Кроме того, на основе ДД удобнее моделировать различные управляющие конструкции программ, такие, как ветвления и циклы. В языке UML также имеются средства для моделирования на основе ДД параллельных процессов, но в данной статье эти средства не рассматриваются. Ввиду того, что ДД используются для моделирования процессов, в которых могут принимать участие несколько объектов, сгенерированные на основе этих диаграмм тесты могут быть использованы для интеграционного тестирования. Особенно удобно их применять при регрессионном тестировании, так как при изменении кода какого-либо класса можно определить, на каких диаграммах присутствуют объекты этого класса, а следовательно, выбрать тесты, которые необходимо выполнить для проверки работоспособности системы после внесения изменений.

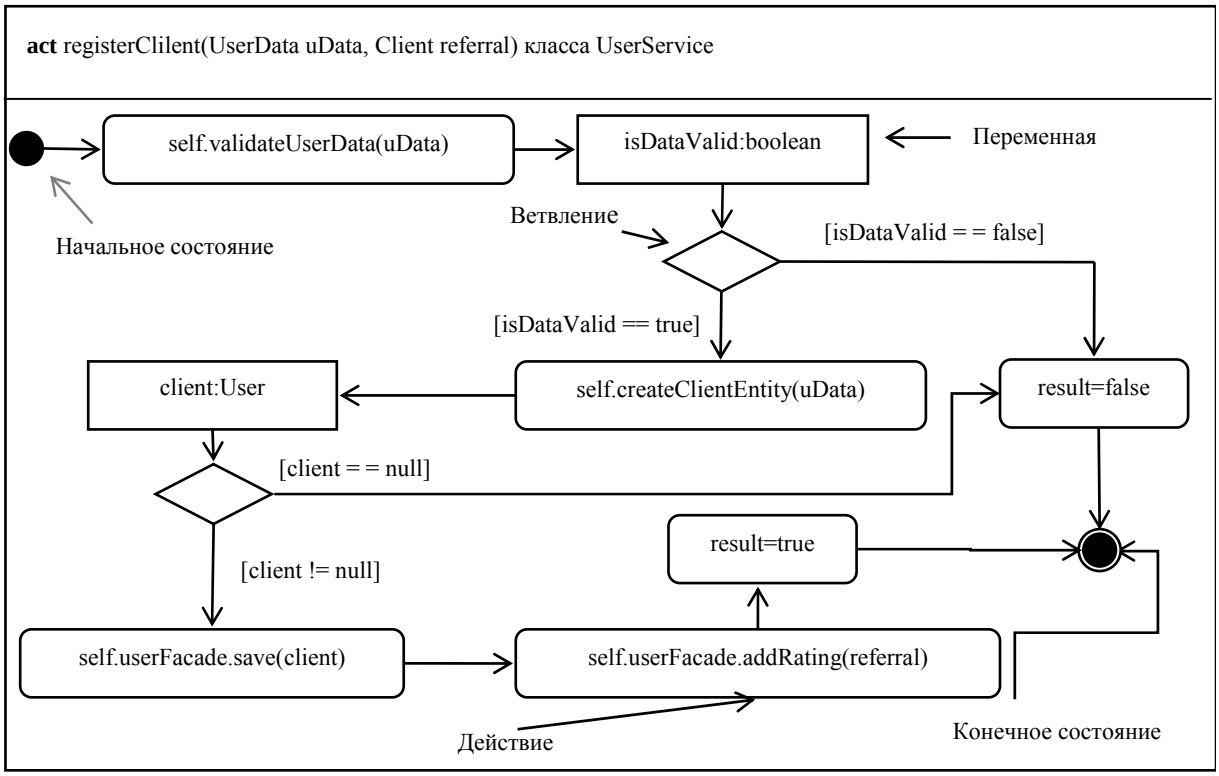
Основными элементами диаграмм деятельности, графически представляемых в виде ориентированных графов, являются узлы-деятельности. С помощью этих узлов моделируются какие-либо действия, арифметические вычисления или вызовы методов объекта в программе. Также узлами-деятельности представляются такие элементы потока управления, как распараллеливание (fork), объединение (join), ветвление (decision) и слияние (merge). Наконец, на ДД присутствуют начальный и конечный узлы, в которых соответственно начинается и заканчивается моделируемый процесс. На ДД узлы-деятельности, моделирующие действия, изображаются в виде прямоугольников со скругленными углами; узлы, моделирующие ветвление и слияние, изображаются в виде ромбов; узлы, моделирующие распараллеливание и объединение, изображаются с помощью жирных линий. Начальное состояние на ДД показано в виде закрашенного круга, а конечное состояние изображается в виде закрашенного круга внутри окружности.

Другим типом узлов, присутствующих на ДД, являются узлы-переменные. Эти узлы моделируют входные и/или выходные параметры узлов, моделирующих действие, и по существу представляют поток данных, обрабатываемых в программе. Узлы-переменные изображаются в виде обычных прямоугольников с прямыми углами.

Все узлы диаграммы деятельности связаны между собой направленными дугами, показывающими, в какой последовательности выполняются действия и какие объекты при этом порождаются и используются. Дуги могут иметь защитные условия, которые обязаны быть выполнены (принять значение «истина»), чтобы переход по дуге был осуществлен. Защитными условиями обычно нагружаются дуги, которые выходят из узлов-ветвлений.

Пример диаграммы деятельности, моделирующей процесс регистрации клиента в некоторой информационной системе, приведен на рисунке. В системе наряду с обычной регистрацией пользователей предусмотрена возможность регистрации клиентов на основе использования реферальных ссылок. При старте процесса регистрации система находится в начальном состоянии, из которого она переходит к действию self.validateUserDate(uDate). Это действие заключается в проверке введенных пользователем данных, называемой валидацией. По окончании этого действия результат проверки записывается в булевскую переменную isDataValid. Узел-ветвление разделяет маршрут дальнейшего выполнения регистрации на 2 пути. Если значение переменной isDataValid равно false, то осуществляется переход в конечное состояние и процесс регистрации завершается. Если значение переменной isDataValid равно true, то выполняется действие self.createClientEntity(uData), при котором создается объект client, представляющий запись клиента в таблице БД. Далее осуществляется сохранение этого объекта в БД посредством выполнения действия self.userFacade.save(client). Заключительным является действие self.userFacade.addRating(referral), при котором рефералу начисляется рейтинг.

По аналогии с моделями на основе диаграмм состояний и диаграмм последовательности, рассмотренными в [1], [2], модель в виде диаграммы деятельности также может быть детализирована ограничениями на языке Object Constraint Language (OCL) [4]. Эти ограничения могут отражать пред- и постусловия операций, которые представлены условиями выполнения действий, или актуальные значения переменных, присутствующих на диаграмме. Кроме того, OCL-ограничениями могут быть нагружены дуги, выходящие из узлов-ветвлений.



OCL-ограничения – это выражения на языке объектных ограничений OCL, аргументами которых являются элементы модели, а результат может быть равен true или false. Если ограничение принимает значение true, то говорят, что оно выполнено, если false – то не выполнено. Для того чтобы детализировать ДД OCL-ограничениями, необходимо иметь диаграмму классов, на которой будут смоделированы все классы, объекты которых участвуют в деятельности, моделируемой ДД. На этой диаграмме классов определяются инварианты классов, пред- и постусловия методов, которые также должны быть записаны на языке OCL.

Для проверки правильности выполнения какого-либо действия на ДД необходимо решить 3 задачи. Во-первых, привести систему в такое состояние, при котором будут выполнены инварианты объектов, участвующих в этом действии, и выполнено предусловие метода, представленного действием. Во-вторых, необходимо выполнить само действие. И в-третьих, необходимо проверить, что не нарушены инварианты объектов и выполнено постусловие метода. Как уже отмечалось, инварианты, пред- и постусловия могут быть записаны на языке OCL. Чтобы привести систему в нужное состояние, необходимо решить булевское OCL-выражение. Под решением подразумевается поиск таких значений переменных, при которых данное выражение принимает значение true. Поиск решений булевского выражения можно осуществить с помощью системы ПРОЛОГ. Для этого необходимо выражение на языке OCL транслировать в выражения языка ПРОЛОГ и запустить их выполнение с целью поиска подходящих значений переменных. Данный подход уже был использован при описании методики генерации модульных тестов по диаграммам состояний [1].

Особенность интеграционного тестирования состоит в том, что тестируется не одиночный объект, а система взаимодействующих объектов. При этом часто один тест проверяет не одиночное действие, а последовательность действий. Если какое-либо действие из этой последовательности выполняется некорректно, то правильность остальных действий не проверяется и тест считается не пройденным. Использование ДД для генерации интеграционных тестов позволяет определить все возможные маршруты выполнения программы. Для каждого маршрута можно составить последовательность выполняемых действий, которая будет основой интеграционного теста, т. е. теста, проверяющего корректность работы системы по соответствующему маршруту выполнения.

Для поиска всех маршрутов выполнения исходную диаграмму деятельности следует преобразовать в ориентированный граф следующим образом:

– каждый узел ДД заменить на вершину графа;

– каждый переход заменить на дугу графа, соединяющую вершины, соответствующие концевым узлам перехода.

Полученный в результате такого преобразования граф будет полностью повторять по структуре исходную ДД и содержать вершины, соответствующие начальному и конечному состояниям ДД, которые обозначим v_s и v_e соответственно. Для поиска возможных маршрутов выполнения программы между вершинами v_s и v_e по ее графу в литературе (например, в [5]) описаны формализованные методы, позволяющие автоматизировать процесс их получения. В этих методах выбор используемых маршрутов производится по различным критериям (минимального покрытия, цикломатического числа и т. д.) в зависимости от требований к точности тестирования.

После определения набора проверяемых маршрутов для подготовки тестов необходимо выполнить обратное преобразование маршрутов на графе в маршруты, представляющие последовательность действий в программе, выраженную в терминах ДД, следующим образом:

– каждую вершину заменить на соответствующий узел ДД;

– каждую дугу заменить на соответствующий переход между узлами ДД.

Далее разработка тестов для проверки правильности выполнения каждого маршрута складывается из следующих шагов:

1) инициализация значений входных данных;

2) выполнение действия текущего узла маршрута, если он не конечный, иначе – выход;

3) проверка постусловия и переход на п. 2, если результат проверки true, иначе – завершение тестирования.

Если все проверки постусловий выполнены успешно, то тест можно считать пройденным.

Наиболее сложной проблемой данной методики является инициализация входных данных в соответствии с начальными условиями. Следует отметить, что эта задача не всегда может быть решена. Возможность решения зависит от полно-

ты UML-модели. Однако существующие подходы к написанию интеграционных тестов позволяют обходить это ограничение, например используя заглушки вместо объектов реальных классов.

Можно выделить 2 вида данных, которые используются в программе при выполнении определенного маршрута:

1) входные данные, которые поставляются объектом класса;

2) порожденные данные, т. е. данные, полученные в результате выполнения действия.

Так, для изображенной на рисунке модели входными данными являются объекты uData класса UserData и referral класса Client, а порожденным данным является значение переменной isDataValid типа boolean. Это значение является результатом выполнения метода validateUserData(uData). Язык OCL позволяет на уровне модели описать результирующее значение метода с помощью секции def*, что дает возможность подготовить начальные данные таким образом, чтобы получить требуемые значения порожденных данных. Например, для модели, изображенной на рисунке, секция def метода validateUserData(uData:UserData) может быть определена следующим образом:

def: validateUserData(uData : UserData) : boolean = (false = uData.firstName.oclIsUndefined() and false = uData.lastName.oclIsUndefined())

Таким образом, результирующее значение метода validateUserData будет равно true, если данные о пользователе содержат его имя и фамилию. Это определение не только позволяет проверять объекты класса UserData, но также задает правило создания валидных и невалидных объектов. Иными словами, это выражение можно использовать для того, чтобы сгенерировать требуемый объект uData, который поступает на вход функции registerClient.

Для того чтобы подготовить входные данные для тестирования определенного маршрута, необходимо найти решение OCL-ограничения, которое является конъюнкцией следующих более мелких ограничений:

1) инварианты всех классов, объекты которых участвуют в тестируемом взаимодействии;

2) предусловия всех методов, которые будут вызваны в процессе выполнения теста;

3) предусловие моделируемого взаимодействия;

4) для каждого конкретного значения порожденного данного должно выполняться условие получения этого значения.

Областью решения этого OCL-ограничения будет набор конкретных значений или интервалов возможных значений каждой переменной, используемых в этом выражении. Каждый набор конкретных значений всех переменных будет удовлетворять предусловию теста.

Как уже отмечалось, после инициализации маршрута входными значениями необходимо выполнить каждый метод из тестируемого маршрута и проверить постусловие выполненного метода. Если в результате все постусловия будут выполнены, то данный маршрут пройден успешно. Для модели, изображенной на рисунке, тест будет проверять правильность выполнения следующих маршрутов:

1) self.validateUserData(uData), result=false;

2) self.validateUserData(uData), self.createClientEntity(uData); result=false;

3) self.validateUserData(uData), self.createClientEntity(uData), self.userFacade.save(client), self.userFacade.addRating(referral); result= true.

Для простоты и ясности примера будем использовать только проверки на null и операцию сравнения чисел. Инвариантом класса UserService является выражение **self.userService.OclIsUndefined()=false**, означающее, что объект self.userService определен и его значение не null. Предусловием метода UserServ ice.registerClient является выражение uData.oclIs Undefined=false and referral.oclIsUndefined(). Предусловия методов, записанные на языке OCL, приведены в таблице.

Метод	Предусловие на языке OCL
UserServ-ice.validateUserD ata(uData)	uData.oclIsUndefined() = false Это условие покрывается предусловием всего моделируемого взаимодействия
UserServ-ice.createClientEn tity(uData)	uData.oclIsUndefined() = false and uData.firstName .oclIsUndefined() = = false and uData.lastName .oclIsUndefined() = false
userFa-cade.save(client)	client.oclIsUndefined()=false
UserFa-cade.addRating(re ferral)	referral.oclIsUndefined()=false Это условие покрывается предусловием всего моделируемого взаимодействия

Постусловием метода UserFacade.addRating (referral) является выражение referral.rating > re-

* Object Management Group, Object Constraint Language version 2.3.1. http://www.omg.org/spec/OCL/2.3.1.

ferral.rating@pre, показывающее, что значение поля referral.rating после выполнения метода больше, чем до его выполнения.

Для первого маршрута требуется тест, проверяющий значение переменной result при условии, что объект uData некорректный, например атрибут name этого объекта равен null. В результате выполнения теста значение result должно быть равно false.

Для второго маршрута нужно составить тест, который проверят значение переменной result при условии, что объект client равен null. В результате выполнения теста значение result также должно быть равно false.

Для третьего маршрута нужно составить тест, который проверяет значение переменной result при выполнении всех пред- и постусловий. В результате выполнения теста значение переменной result должно быть равно true.

Следует обратить внимание, что для маршрутов 2 и 3 объект client создается в результате работы объекта UserService.userFacade. При этом в двух тестах необходимо иметь разные значения объекта client при наличии одинакового входного условия. Чтобы провести полноценное тестирование класса UserService, можно использовать 2 разных mock-объекта, т. е. 2 фиктивных объекта, имитирующих работу объекта реального класса исключительно для целей тестирования. Первый mock-объект будет возвращать в результат выполнения метода createClientEntry(UserData) значение null, а второй – любой объект класса Client, но не null.

В заключение хотелось бы отетить, что диаграммы деятельности языка UML в совокупности с диаграммами классов позволяют моделировать как структуру системы, так и ее поведение, а мощность языка позволяет применять его к моделированию систем любой сложности и уровня абстракции. В свою очередь язык OCL, предназначенный для записи объектных ограничений, является хорошо формализованным языком, позволяющим широко известными методами [6] транслировать записанные на нем выражения в программы на другом языке, в том числе и на языке ПРОЛОГ. Это позволяет автоматически получать решения OCL-выражений и использовать их для подготовки входных данных при генерации тестов для проверки требуемых маршрутов. Следовательно, можно утверждать, что совместное использование языков UML и OCL позволяет создавать модели программ, с помощью которых можно средствами автоматизации генерировать наборы тестов, проверяющих правильность работы моделируемых программ.

СПИСОК ЛИТЕРАТУРЫ

1. Кирьянчиков В. А., Романов А. С. Автоматизированная генерация модульных тестов по диаграммам состояний // Изв. СПбГЭТУ «ЛЭТИ». 2011. Вып. 9. С. 55–60.

2. Кирьянчиков В. А., Романов А. С. Автоматизация генерации тестов программ по диаграммам последовательности // Изв. СПбГЭТУ «ЛЭТИ». 2012. Вып. 9. С. 53–59.

3. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. СПб.: ДМК-Пресс, 2004.

4. Warmer J., Kleppe A. Object Constraint Language, The: Getting Your Models Ready for MDA. Second Edition. Addison Wesley, 2003.

5. Липаев В. В. Качество программного обеспечения. М.: Финансы и статистика, 1983.

6. Опалева Э. А., Самойленко В. П. Языки программирования и методы трансляции. СПб.: БХВ-Петербург, 2005.

V. A. Kiryanchikov, A. S. Romanov

THE AUTOMATED OBJECT-ORIENTED PROGRAMS TEST GENERATION BASED ON THE ACTIVITY DIAGRAMS

The article describes technique of the automated integration test generation based on UML activity diagrams. An example of the activity diagram is considered and the method of the testing scenario development is suggested. In conclusion the appropriate sets of test descriptions are given.

Model-Based Testing, UML, activity diagram, generation of integration tests, OCL, Prolog