Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

А. Ю. Законов

Применение генетических алгоритмов к генерации тестов для автоматных программ

Магистерская диссертация

Руководитель – О. Г. Степанов

Санкт-Петербург

2010

Оглавление

Введение

Автоматная программа состоит из автоматизированных объектов управления, каждый из которых является системой из конечного автомата и *набора объектов управления*, с которыми взаимодействует автомат [1]. Наиболее распространенным способом проверки автоматных программ является *Model Checking* [2], так как для автоматных программ высока степень автоматизации. Однако проверка моделей позволяет верифицировать только автомат, но не всю систему в целом. Поведение объектов управления и их взаимодействие с автоматом не проверяются при указанном подходе. Таким образом, в автоматной программе могут остаться невыявленные ошибки, даже если сам автомат был успешно верифицирован относительно данной спецификации.

В данной работе предлагается использовать тестирование для проверки автоматных программ в целом. Тестирование является трудоемкой и ресурсоемкой задачей. Около половины всего времени разработки проекта часто тратиться на тестирование [3]. В связи с этим в последнее время многие исследования посвящены теме автоматизации процессов создания и запуска тестов [4]. С одной стороны, важно отметить, что успешное тестирование не может гарантировать отсутствие ошибок в программе. С другой стороны, большой набор тестов может существенно помочь в обнаружении ошибок в логике работы программы и в ее реализации, и тем самым повысить вероятность того, что программа будет успешно решать поставленные задачи. В данной работе предлагается подход к тестированию автоматных программ и способ автоматизировать процесс создания тестов при помощи использования генетических алгоритмов.

В предлагаемом подходе тестирование используется для проверки соответствия спецификации системы ее реализации. Спецификация, заданная на естественном языке, пригодна только для ручного тестирования. Для

автоматизации процесса тестирования спецификация должна быть записана на формальном языке. По этой причине предлагается внести максимально возможное число требований спецификации в описание автомата. Это позволит расширить программу так, что она будет содержать в себе требования для проверки. Часто для описания автоматной модели используются конечные автоматы. Однако конечный автомат позволяет смоделировать только простую реакцию системы на события. Для моделирования поведения сложных систем, взаимодействующих со средой, в работе предлагается использовать расширенный конечный автомат, который позволяет задавать переменные и охранные условия на переходах. Зависимость поведения системы от значений этих переменных позволяют описать сложную логику работы и тем самым внести значительную часть спецификации в модель. В работе [5] предлагается использовать контракты [6] для внесения дополнительных требований спецификации в модель. Наличие исполняемой спецификации внутри автоматной модели позволяет при запуске программы автоматически проверять выполнение описанных условий. Более того, формальное описание условий позволит реализовать автоматическую генерацию тестов и искать значения, которые помогут выявить несоответствия реализации и спецификации.

Большая часть программ разрабатывается для взаимодействия с некой средой: программа получает события и входные параметры; автомат реагирует на эти события и производит результирующие данные. В автоматных системах для этой цели используются объекты управления – они получают из среды события и входные параметры, которые используются в автомате в охранных условиях на переходах или передаются как аргументы вызовов методов других объектов управления. В расширенном конечном автомате все эти значения можно представить переменными. Таким образом, переменные, использованные в модели, разделяются на два типа: внутренние, заданные и определенные внутри модели, и внешние – полученные из среды посредством

объектов управления. Во время тестирования значения внешних переменных необходимо сгенерировать вместе с кодом теста.

Учитывая приведенное выше описание модели и ее спецификации, определим понятие теста для автоматной программы. В случае традиционного подхода к разработке программного обеспечения (ПО) создание тестов осуществляется путем представления требований спецификации в виде программного кода. В предложенном подходе тест определяется не в виде кода, а в терминах автоматной модели и требований спецификации: тест автоматной программы – последовательность переходов автомата. Тест также содержит требования спецификации к задействованным переходам, состояниям и объектам управления. Такое описание теста значительно понятнее, так как не требует изучения исходного кода программы и позволяет описать важные тестовые сценарии уже на этапе написания спецификации, а не кода программы. Это должно эффективно помогать преодолеть семантический разрыв между реализацией программы и ее спецификацией, так как значительно проще перейти от естественного языка, на котором создана изначально спецификация, к автоматным терминам и такому понятию, как последовательность переходов автомата, чем к написанию программного кода.

Предложенное определение теста удобно для его описания, но не пригодно для автоматического создания кода этого теста. Для того, чтобы автомат выполнил описанный в тесте сценарий, необходимо определить последовательность событий и набор значений внешних переменных, которые приводят к выполнению автоматом заданной последовательности переходов. Определение последовательности событий не вызывает затруднений, так как, зная текущее состояние и следующий переход, однозначно можно определить, какое событие может привести к этому переходу. Однако определить набор внешних переменных, удовлетворяющих описанному пути, значительно сложнее: значения должны удовлетворять всем охранным условиям на

переходах заданного пути. Для поиска значений внешних переменных в данной работе предлагается использовать генетические алгоритмы.

Целью данной работы является разработка эффективного подхода к тестированию автоматных программ.

Задачи, рассмотренные в данной работе:

- предложить способ создания тестов для автоматных программ;
- автоматизировать процесс создания тестов, разработав инструмент для поиска последовательности событий и набора значений внешних переменных для заданной последовательности переходов в автомате;
- автоматизировать проверку требований спецификации, описанной в автоматной модели, во время запуска тестов;
- предложить способ нахождения тестовых сценариев, которые приведут к нарушению спецификации и позволят обнаружить несоответствие между спецификацией и реализацией.

В главе 1 приведено описание используемых технологий. Также глава содержит обзор существующих разработок и достижений в исследуемой области.

Глава 2 посвящена детальному описанию предлагаемого подхода. В ней будет рассмотрено использование предложенных в рамках этой работы инструментов и будут сформулированы особенности создания тестов для автоматных программ.

Глава 3 иллюстрирует разрабатываемый подход на примерах и рассказывает о результатах апробации.

Глава 1. Обзор

Корректность программы [5] – соответствие реализации программы заданной спецификации. В данном разделе рассматривается вопрос корректности программ в рамках автоматного подхода.

Для разработки тестов необходима информация о корректном поведении тестируемой системы и о разнообразии ситуаций, которые могут возникать при ее взаимодействии с другими системами или пользователями. Данная глава описывает существующие методы записи требований, построения моделей и тестирования.

1.1. Проблема проверки корректности автоматных программ

В книге [1] описан метод проектирования программ с явным выделением состояний, названный «автоматное программирование» или «SWITCH-технология». При использовании автоматного подхода программой является система автоматизированных объектов управления. В свою очередь автоматизированный объект управления представляет собой совокупность автомата и объектов управления, при помощи которых происходит взаимодействие программы со средой.

Особенность автоматного подхода заключается в том, что построение программы предлагается начинать с построения схемы связей, содержащей источники информации, систему управления, объекты управления и обратные связи от объектов к системе управления. SWITCH-технология предполагает для каждого автомата два типа диаграмм: схему связей и граф переходов. При наличии нескольких автоматов строится схема их взаимодействия. Для создания таких диаграмм предложена соответствующая нотация [7].

В работе [8] процесс создания программных систем подразделяют на следующие этапы:

1. Постановка задачи. Включает в себя сбор требований и создание прототипа программы.
2. Технический дизайн. Состоит в создании на основе собранных требований технического задания, описывающего модель системы с помощью диаграмм, псевдокода и поясняющего текста.
3. Создание кода. Заключается в реализации системы для целевой программно-аппаратной платформы в соответствии с техническим дизайном.
4. Проверка корректности. Завершает цикл разработки и представляет собой отладку созданного приложения и проверку соответствия реализации собранным требованиям.

В случае автоматных программ этап создания кода начинается с построения автоматной модели. Для описания модели принято использовать либо конечные автоматы, либо расширенные конечные автоматы.

Конечный автомат (*Finite State Machine, FSM*) [10] позволяет описать состояние, события и реакцию системы на эти события. В определенном состоянии каждому событию соответствует определенный переход. Для систем со сложным поведением более эффективно использовать расширенные конечные автоматы (*Extended Finite State Machines, EFSM*) [11], так как они позволяют добавить в модель переменные и охранные условия на переходы. Это позволяет избежать избыточного числа состояний и переходов между ними, что делает модель запутанной. Формальное определение и подробное описание расширенных конечных автоматов приводится в работе [12].

Благодаря построению модели этап создания кода в значительной степени автоматизирован. Большая часть кода программы генерируется по автоматной модели. Вручную реализуются объекты управления, с которыми взаимодействует автомат. Вследствие того, что разработка начинается с

построения модели, автоматный подход позволяет эффективно разрабатывать модули со сложным поведением.

Проверка корректности – необходимый этап разработки программного обеспечения, вне зависимости от того, какой подход используется. Основные методы проверки корректности можно разделить на три группы [13]: статический анализ, формальные методы и динамические методы.

Статический анализ используется для проверки формализованных правил и поиска часто встречающихся дефектов по заранее заданным шаблонам. Этот метод легко автоматизируется и может быть осуществлен при помощи инструментов. Однако он применим только к исходному коду программы и способен обнаруживать ограниченный набор типов ошибок.

Формальные методы используют для анализа свойств ПО формальные модели требовании к поведению программы. Анализ формальных моделей выполняется с помощью таких техник, как дедуктивный анализ (*Teorem Proving*) и проверка моделей (*Model Checking*).

Формальные методы применимы только к тем свойствам , которые выражены формально в рамках некоторой математической модели . Соответственно, для использования таких методов в проекте необходимо затратить значительные усилия на построение формальных моделей. Анализ их свойств в значительной мере может быть автоматизирован , однако для этого требуется набор навыков и знаний в специфических разделах математической логики и алгебры.

Динамические методы – методы проверки корректности, в рамках которых анализ и оценка свойств программной системы делаются по результатам ее реальной работы или работы некоторых ее моделей и прототипов . Примерами такого рода методов являются обычное тестирование или имитационное тестирование, мониторинг, профилирование. Для применения динамических

методов необходимо иметь работающую систему или хотя бы некоторые ее компоненты, или же их прототипы, поэтому нельзя использовать их на первых стадиях разработки. Зато с их помощью можно контролировать характеристики работы системы в ее реальном окружении, которые иногда невозможно аккуратно проанализировать с помощью других подходов. Динамически методы позволяют обнаруживать в ПО только ошибки, проявляющиеся при его работе, а, например, дефекты удобства сопровождения найти не помогут, однако, обнаруживаемые ими ошибки обычно считаются более серьезными.

Для применения динамических методов верификации обычно требуется дополнительная подготовка – создание тестов, разработка тестовой системы, позволяющей их выполнять или системы мониторинга, позволяющей контролировать определенные характеристики поведения проверяемой системы. Системы тестирования, профилирования или мониторинга могут быть сделаны один раз и использоваться многократно для широких классов ПО, лишь сами тесты необходимо готовить заново для каждой проверяемой системы.

Создание набора тестов, позволяющих получить адекватную оценку качества сложной системы, является довольно трудоемкой задачей. Однако среди разработчиков промышленного ПО сложилось мнение [13], что тестирование является наименее ресурсоемким методом верификации. Поэтому на практике оно используется для оценки свойств ПО очень широко. При этом чаще всего применяются не слишком надежные, но достаточно дешевые техники, такие как, например, нестрогое вероятностное тестирование, при котором тестовые данные генерируются случайным образом, или же тестирование на основе базовых сценариев использования, проверяющие лишь наиболее простые ситуации.

Для автоматных программ наиболее распространенным способом проверки является верификация. Тот факт, что модель первична, позволяет

повысить уровень автоматизации при верификации с помощью метода *Model Checking*. Недостатком этого метода является то, что он не позволяет находить ошибки в объектах управления и в их взаимодействии с автоматами. Для этой цели подходит тестирование, которое позволяет проверять систему в целом. Особенности тестирования и способы записи требований спецификации для тестирования рассмотрены далее.

1.2. Требования к программному обеспечению

Требования играют первостепенную роль при тестировании – именно они и проверяются с его помощью. В работе [12] выделено несколько функций, характерных для требований:

1. Требования говорят о том, какие конкретные свойства и характеристики данная система должна иметь, чтобы ее можно было использовать для решения поставленных задач.

2. Требования определяют, какое поведение системы является правильным и желаемым, а какое должно рассматриваться как некорректное.

Спецификация системы определяется при анализе задач, стоящих перед рассматриваемой программной системой и ее окружением. Требования могут извлекаться из различных источников:

1. Стандарты, регламентирующие функции и состав систем, работающих в определенной предметной области.

2. Внутренние ограничения задач и алгоритмов, которые выделяют сами разработчики или заказчики. Чаще всего ограничения такого рода можно определить при детальном анализе решаемой задачи.

Тестирование проверяет соответствие требованиям, и поэтому чем более точно и ясно они сформулированы, тем аккуратнее и полнее можно провести тестирование. Если какие-то требования определены нечетко, проверка их тоже

может быть выполнена лишь с определенной степенью точности, и две разные системы могут быть признаны удовлетворяющими требованиям в одинаковой мере. Получаемые при этом оценки качества таких систем будут недостоверными.

Во многих случаях тестирование может оказаться полезным и в отсутствии документально зафиксированных требований. В этих случаях проверяются некие свойства, которые считаются само собой разумеющимися и, соответственно, представляющие собой неявные требования. Пример такого свойства – отсутствие исключительных ситуации и зависаний при работе программы. Однако надо учитывать, что иногда сбой может рассматриваться как корректное поведение системы, например, как реакция системы на ввод некорректных исходных данных. В этих случаях подобное тестирование неприменимо и может привести к неправильным выводам о наличии ошибок в системе.

1.3. Программные контракты для записи спецификации

Существуют разные способы записи требований: на естественном языке, записанные при помощи темпоральной логики, утверждения (*Assertions*), программный код тестов и т.д. В работе [5] обосновывается удобство использования контрактов [6, 14] для записи требований к автоматным программам.

Программирование по контрактам (*Design by Contract*) – подход к объектно-ориентированному программированию, предложенный Бертраном Мейером [15]. Традиционно, в программировании по контрактам используются три вида специальных утверждений: предусловия, постусловия и инварианты.

Как определено в работе [16], программный контракт операции состоит из предусловия и постусловия. *Предусловие* фиксирует требования к корректному использованию этой операции со стороны окружения системы – при каких ограничениях на аргументы обращение к этой операции корректно.

Постусловие определяет то, 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, и поведение системы в этом случае не определено. *Инварианты* задают требования, которые должны выполняться на протяжении всего жизненного цикла программы.

Как отмечено в работе [5], важным свойством контрактов является их исполнимость. Это значит, что контракты должны проверяться в ходе работы программы. Для этого требуется интегрировать контракты и код программы. Лучше всего такая интеграция осуществляется при использовании языков, поддерживающих программирование по контрактам. На настоящий момент самым распространенным языком программирования с поддержкой контрактов является язык *Eiffel* [17], в котором программирование по контрактам и было впервые реализовано.

Для языка *Java* наиболее распространенным способом записи контрактов является язык *Java Modeling Language* (*JML*) [18], который интегрируется в *Java*-код в виде аннотаций [19]. *JML*-контракты удобно использовать и в сочетании с автоматным подходом. В работе [20] предложен способ генерации *JML*-аннотации по автоматной модели для языка *JavaCard* [21], которые определяют корректную последовательность переходов между состояниями, вызов функции объектов управления из корректных состояний, выполнение указанных на переходах условий. Инструмент *AutoJML* [22] позволяет из автоматного описания генерировать *JML*-аннотации, используя которые можно проанализировать сгенерированный из автомата *Java*-код методами статической проверки, например, при помощи *ESC/JAVA2* [23].

1.4. Традиционное тестирование

Тестирование программного обеспечения – процесс исследования программного обеспечения с целью получения информации о качестве

продукта [24]. В работе [25] приведен краткий обзор распространенных методов тестирования. Наиболее часто используемые виды – модульное и системное тестирование.

Модульное тестирование – в данном случае тестированию подвергаются небольшие модули программы. При тестировании модуля размером 100–1000 строк имеется возможность проверить, если не все, то, по крайней мере, многие логические ветви реализации и граничные значения параметров. В соответствии с этим строятся критерии тестового покрытия (покрыты все операторы, все логические ветви, все граничные точки). Проверка корректности всех модулей не гарантирует корректности функционирования системы модулей.

Полностью реализованный программный продукт подвергается системному тестированию. На данном этапе тестировщика интересует не корректность реализации отдельных процедур и методов, а вся программа в целом, как ее видит конечный пользователь. Основой для тестов являются общие требования к программе, включающие не только корректность реализации функций, но и производительность, время отклика, устойчивость к сбоям, атакам, ошибкам пользователя. Для системного тестирования используются другие виды критериев тестового покрытия – покрыты ли все типовые сценарии работы, все сценарии с нештатными ситуациями, попарные композиции сценариев и т.п.

Для проведения тестирования разрабатываются тестовые сценарии (*test case*). В работе [13] показано, что каждый тестовый сценарий предназначен для проверки определенных свойств компонентов системы в заданной конфигурации и включает в себя:

- действия по инициализации тестируемой системы (или компонента) и приведение ее в необходимое состояние;

- набор воздействий на систему, выполнение которых должно быть проверено данным тестом;
- действия по проверке корректности поведения системы – сравнение полученных и ожидаемых результатов, проверка необходимых свойств результатов, проверка инвариантов данных системы;
- финализацию системы – освобождение захваченных при выполнении теста ресурсов.

Во время запуска тестов создаются отчеты о нарушениях: детальное описание отдельных ошибок, обнаруженных при выполнении тестов, с указанием всех условий, необходимых для их проявления, нарушаемых требований и ограничений, возможных последствий, а также предварительной локализации ошибки в одном или нескольких модулях.

По результатам тестирования создается итоговый отчет о тестировании: суммарная информация о результатах тестов, включающая достигнутое тестовое покрытие по используемым критериям и общую оценку качества компонентов системы, проверяемых тестами.

Подходы к тестированию традиционных программ неудобно использовать для автоматных программ, так как они могут тестировать только сгенерированный из автомата код, но при этом теряются все преимущества автоматного подхода.

1.5. Тестирование на основе модели

Модель [12] – некоторое отражение структуры и поведения системы. Модель может описываться в терминах состояния системы, входных воздействий на нее, конечного набора состояний, потоков данных и потоков управления, возвращаемых системой результатов. Формальная спецификация представляет собой законченное описание модели системы и требований к ее поведению в терминах того или иного формального метода. Тестирование на

основе модели (*Model-Based Testing*) [12] – это тестирование программного обеспечения, в котором тестовые сценарии (*Test Cases*) частично или целиком создаются по модели, описывающей некоторые аспекты (чаще функциональные) тестируемой системы.

В работе [26] приводятся следующие достоинства тестирования на основе моделей:

- тесты на основе спецификации функциональных требований более эффективны, так как они в большей степени нацелены на проверку функциональности, чем тесты, построенные только на знании реализации;
- на основе формальных спецификаций можно создавать самопроверяющие тесты, так как из формальных спецификаций часто можно извлечь критерии проверки результатов целевой системы.

Недостаток тестирования на основе моделей состоит в том, что в случае традиционного подхода модель системы требуется специально создавать для тестирования. Ручное создание модели трудоемко, а модели, созданные по коду программы автоматически, часто запутаны и огромны. Также отсутствие ошибок в модели не гарантирует отсутствие ошибок в программе, так как при построении модели могли быть допущены ошибки или не учтены особенности реализации. *В случае автоматного подхода тестирование на основе моделей применять значительно удобнее*, так как разработка начинается с построения модели, и код программы генерируется по этой модели автоматически.

1.6. Применение генетических алгоритмов к тестированию

Генетический алгоритм [27 – 29] – это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путем случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, напоминающих биологическую эволюцию. При

тестировании традиционных программ генетические алгоритмы эффективно используются [4, 30, 31] для поиска тестовых значений и генерации кода тестов. Для применения генетического алгоритма к решению определенной задачи необходимо один раз определить функцию приспособленности для особей популяции. Задача поиска значений для создания теста сводится к оптимизационной задаче при помощи подбора соответствующей функции приспособленности.

Функция приспособленности должна удовлетворять следующему условию: чем лучше особь, тем выше приспособленность. В задачах, связанных с применением эволюционных алгоритмов для тестирования, часто выбирают фитнес-функцию (функцию приспособленности), которая сопоставляет более хорошим особям меньшие значения. Это удобно, так как однозначно понятно, какое решение надо найти. Если предложенное решение удовлетворяет поставленной задаче, то его значение приспособленности – ноль. Если поставленная задача не решена, то функция приспособленности оценивает, насколько близок этот вариант к правильному решению задачи. В таком случае цель задачи оптимизации – найти решение с нулевым значением функции приспособленности.

Обычно выделяются следующие этапы генетического алгоритма:

1. Создать начальную популяцию (один раз).
2. Начало цикла: размножение с использованием скрещивания.
3. Мутация.
4. Вычисление значения функции приспособленности для всех особей.
5. Формирование нового поколения.
6. Если выполняются условия останова, то конец цикла, иначе переход к началу цикла.

1.7. Тестирование расширенных конечных автоматов

При тестировании моделей, построенных с использованием расширенных конечных автоматов, выделяется две подзадачи: выбор или генерация выполнимых путей (*Feasible Paths*) для тестирования и подбор входных значений для выполнения этих путей.

Вопрос о выборе выполнимых путей для тестирования рассматривался во многих работах [9, 32 – 34]. При этом отметим, что пути для тестирования можно задавать вручную, выбирая наиболее интересные сценарии для проверки. Для путей, полученных автоматически или вручную, необходимо найти значения переменных для их прохождения с учетом всех охранных условий расширенного конечного автомата. Задача поиска этих значений менее изучена, но не менее актуальна, так как без этих значений невозможно создать тест для запуска, и все найденные выполнимые пути в автомате бесполезны. Для решения задач такого типа предлагается использовать генетические алгоритмы [9].

В данной работе предлагается алгоритм поиска значений для прохождения пути в автоматной модели, с учетом охранных условий на переходах и требований спецификации. Этот подход может быть интегрирован с алгоритмами для автоматического выбора выполнимых путей или пути могут задаваться вручную.

1.8. Существующие инструменты для тестирования моделей

Существует ряд инструментов для работы с моделями и создания тестов на их основе. В работе [35] рассмотрены инструменты для тестирования на основе моделей программ, написанных на языке *C#*. Предлагается инструмент *NModel* [36], который позволяет создавать модели на текстовом языке, автоматически строить по модели конечные автоматы, описывающие переходы в системе, и генерировать тестовые сценарии для этой системы.

В работе [37] приведен обзор инструментов для тестирования на основе моделей и также предложен инструмент *ModelJUnit* [38], который позволяет описывать расширенные конечные автоматы, как классы на языке *Java*, и автоматически генерировать для них тесты.

Инструмент *Tigris MBT* [39] позволяет графически описывать расширенные конечные автоматы и, используя случайные значения, пытается найти последовательность событий и набор переменных для полного покрытия переходов или для достижения определенного состояния.

Институт системного программирования РАН предлагает семейство инструментов разработки тестов на основе моделей *UniTesK* [40, 41], которые предлагают использовать «спецификации ограничений» для построения модели системы, используя ее интерфейсы. Для написания тестов используется *SeC* – спецификационное расширение языка *C*.

Перечисленные инструменты не решают проблему тестирования автоматных программ. Принципиальное отличие автоматных программ от моделей, используемых в существующих инструментах, состоит в том, что автоматная программа включает в себя помимо автомата объекты управления. Тестирование только автомата не позволит найти ошибки в программе в целом. Также указанные инструменты предполагают ручное создание требований для проверки отдельно от модели, что делает процесс разработки более сложным.

1.9. Постановка задачи

С учетом описанных ранее особенностей предметной области и обзора существующих инструментов можно сформулировать следующую **цель работы**: предложить способ эффективного применения тестирования для проверки соответствия реализации автоматной программы заданной спецификации.

Выделим подзадачи, которые необходимо решить для тестирования автоматных программ:

1. Предложить способ перевода спецификации из естественного языка в формат, пригодный для автоматической проверки.
2. Предложить простой и удобный способ записи тестовых сценариев.
3. Автоматически создавать из описания тестового сценария код теста, пригодный для запуска.
4. Проверять соблюдение условий спецификации во время выполнения теста.

Также необходимо предложить прототип инструмента для тестирования автоматных программ, который учитывает взаимодействие автоматов с объектами управления и позволяет удобно внести требования спецификации в автоматную модель, так как существующие инструменты не удовлетворяют этим требованиям.

Выводы по главе 1

В данной главе приведен краткий обзор существующих методов тестирования, записи требований спецификации и рассмотрена применимость существующих методов и инструментов к автоматным программам. Применительно к автоматному подходу, проведенный анализ существующих инструментов и работ в этой области выявил ряд недостатков и направлений для дальнейших исследований. Сформулирована цель данной работы и задачи, которые необходимо решить для ее достижения.

Глава 2. Применение генетических алгоритмов к генерации тестов для автоматных программ

Для решения поставленной задачи предлагается использовать возможности расширенных конечных автоматов для создания моделей, генетические алгоритмы для автоматизации создания тестов и программные контракты проверки корректности автоматных программ.

Первый раздел главы посвящен формулированию подхода, описанию решений и инструментов, разработанных для проверки посредством тестирования реализации автоматной программы, относительно требований спецификации. Второй раздел главы описывает особенности применения генетических алгоритмов для автоматизации создания тестов. Третий раздел содержит анализ преимуществ, которые имеет предложенный подход, по сравнению уже существующими разработками и инструментами в этой области.

2.1. Предлагаемый подход

Данный раздел описывает подход к созданию тестов для автоматных программ и метод автоматизации процесса создания и запуска тестов.

2.1.1. Включение спецификации в модель

Спецификация, записанная на естественном языке, пригодна только для ручного тестирования. Поэтому в процессе разработки максимально возможный процент спецификации следует формализовать, включив в модель, используя расширенный конечный автомат, переменные, охранные условия на переходах и контракты.

Объекты управления, с которыми взаимодействует автомат, также имеют спецификацию на входные воздействия, результаты их работы и особенности взаимодействия с автоматом. Все эти требования спецификации должны быть выполнены во время корректной работы программы. В предложенном подходе

спецификация объектов управления включается в описание автоматной модели. Благодаря этому особенности реализации конкретных объектов управления не важны на этапе создания тестов, так как достаточно проверять, что на вход им подаются корректные значения и что автомат корректно обрабатывает любые значения, полученные из объектов управления, которые не противоречат их спецификации.

В предложенном подходе для интеграции требований спецификации в описание автоматной модели используются контракты, записанные на языке *JML* [18]. В случае автоматной модели предлагается задавать переходам пред- и постусловия, а для состояний записывать инварианты:

- предусловия перехода определяют ожидания системы относительно значений переменных при начале этого перехода и предусловия методов объектов управления, которые будут вызваны на переходе;
- постусловия перехода задают требования к значениям переменных модели при завершении перехода и содержат постусловия вызванных методов объектов управления. Эти требования должны соблюдаться после выполнения действий на переходе и вызова методов объектов управления;
- инварианты, заданные для состояния, содержат требования, которые должны выполняться на протяжении всего времени пребывания системы в указанном состоянии.

2.1.2. Создание сценариев для тестирования

Проанализировав спецификацию, заданную на естественном языке, необходимо выбрать сценарии работы, интересные для тестирования. Выбранные сценарии далее понятным образом записываются в автоматных терминах – как последовательность переходов автомата. Такое представление теста должно быть интуитивно понятно как автору спецификации,

работающему с требованиями, записанными на естественном языке, так и разработчику системы, работающему с автоматной моделью и кодом объектов управления.

Существует ряд исследований [9, 43], в которых изучается задача генерации пути в расширенном конечном автомате, который обеспечивал бы максимальное покрытие переходов и состояний. Описанные технологии автоматического создания сценариев можно успешно сочетать с подходом, представленным в данной работе, для создания эффективных наборов тестов.

2.1.3. Поиск значений переменных для заданного пути

Автомат реагирует на события и выполняет переходы в зависимости от значений переменных автоматной модели, которые используются в охранных условиях на переходах. Представление теста в виде последовательности событий и набора значений внешних переменных удобно с точки зрения автоматического создания кода теста, однако неудобно и неясно для разработчика, который основывается на спецификации, написанной на естественном языке. Поэтому разработчик записывает тест как последовательность переходов в модели. В этом случае возникает задача поиска последовательности событий и набора значений внешних переменных, соответствующих заданной последовательности переходов в модели. В работе предложен алгоритм для автоматизации решения этой задачи.

Последовательность событий однозначно определяется по последовательности переходов. Сложнее подобрать значения переменных – они должны удовлетворять ряду требований. Во-первых, охранные условия на всех переходах в описанном пути должны быть выполнены. Во-вторых, все требования спецификации объектов управления должны выполняться, так как при реальном использовании значений этих переменных будут приходить из объектов управления с данными спецификациями. В предложенном подходе генетические алгоритмы применяются для решения задачи поиска значений.

Оптимизационные алгоритмы считаются эффективными для решения задач такого типа [4]. Подробности использования генетического алгоритма описаны в разд. 2.2.

2.1.4. Генерация кода теста и оценка корректности поведения программы

Найденные значения переменных и последовательность событий позволяют автоматически сгенерировать код теста, пригодный для запуска. Код теста, запуск которого приведет к выполнению требуемой последовательности переходов, проверяет корректность системы для этого сценария относительно спецификации, добавленной в автоматную модель. Поддержка контрактов позволяет внести большую часть спецификации в модель и тем самым автоматизировать процесс проверки системы относительно этой спецификации.

Требования, записанные на языке *JML,* включаются в *Java*-код программы в виде аннотаций. Для *JML* разработан ряд инструментов для автоматической проверки требований как статически [20], так и во время выполнения программы [42].

Тесты, которые не обнаруживают никаких ошибок в автоматной программе, являются полезными для регрессионного и стесс-тестирования [24]. Однако важнее генерировать тесты, приводящие к нарушению требований спецификации для корректного набора внешних переменных, поданного на вход, так как это позволит выявить несоответствия спецификации и реализации.

2.1.5. Поиск значений внешних переменных, приводящих к нарушению спецификации

Для поиска такого набора переменных также применяется генетический алгоритм с более сложной функцией приспособленности, которая учитывает не только охранные условия и предусловия на переходах, но также требования

спецификации к автоматной программе и объектам управления, заданным для выполняемого сценария. Более подробно функция приспособленности описана в разд. 2.2.4.

2.2. Применение генетического алгоритма для поиска значений переменных

Генетические алгоритмы считаются полезными для решения оптимизационных задач. Задачу поиска набора значений, при котором будет выполнен заданный путь в расширенном конечном автомате, можно свести к задаче оптимизации.

2.2.1. Задача оптимизации

Набор внешних переменных, задействованных на выбранной последовательности переходов, можно рассматривать как вектор значений внешних переменных <x1, x2, …, xn>, где xi – значение внешней переменной, а n – число внешних переменных для этого пути. С точки зрения генетического алгоритма, набор внешних переменных является хромосомой. Для использования оптимизационных алгоритмов необходимо определить функцию приспособленности – функцию, которая оценивает, насколько хромосома решает проблему, и позволяет выбирать лучшие решения из всего поколения.

В данной задаче функция приспособленности на вход принимает вектор значений и выдает число, характеризующее приспособленность этого вектора для выполнения заданного пути. Чем меньше это число, тем больше подходит данный вектор значений. Таким образом, задача поиска подходящего набора значений внешних переменных сводится к задаче оптимизации, где требуется найти вектор, которому соответствует минимальное значение функции приспособленности.

2.2.2. Кодирование хромосом, скрещивание и мутация

Хромосома – вектор значений внешних переменных. Один ген это значение одной из внешних переменных для заданного пути. В работе используется одноточечное (классическое) скрещивание (*One-Point Crossover*), в котором две хромосомы разрезаются один раз в соответствующей точке, и производится обмен полученными частями.

Оператор мутации (*Mutation Operator*) необходим для «выбивания» популяции из локального экстремума и способствует защите от преждевременной сходимости. В классическом подходе это достигается за счет того, что инвертируется случайно выбранный бит в хромосоме. В случае с вектором значений переменных мутация выполняется заменой произвольного гена на случайно выбранное число из области допустимых значений.

2.2.3. Функция приспособленности

Одной из поставленных в работе задач является проблема поиска набора значений внешних переменных, при которых автомат, получая соответствующие события, выполнит требуемую последовательность переходов. Функция приспособленности должна удовлетворять следующему условию: чем лучше особь подходит для поставленной задачи, тем меньше значение функции. Таким образом, проблема сводится к задаче оптимизации – найти решение с нулевым значением функции приспособленности.

Однозначного способа определения функции приспособленности не существует. В работах [44, 45], описывающих применение генетических алгоритмов для тестирования структурных программ, используется такой критерий, как *расстояние до условия* (*Branch Distance*), для определения приспособленности хромосом. *Расстояние до условия* позволяет оценить, насколько близка была данная хромосома к выполнению конкретного условия, которое на практике не было выполнено. Например, для условия `A > B`, *расстояние до условия* будет вычисляться по формуле `|A - B|`. Чем меньше

значение `|A − B|`, тем ближе значение `A` к `B` и тем ближе хромосома к тому, чтобы это условие было выполнено. Если условие выполнено, то *расстояние до условия* ноль. При этом

$$\textit{расстояние до условия}(\text{"}A >= B\text{"}) = \begin{cases} 0, A >= B \\ |A - B|, A < B \end{cases}$$

В статье [9] функция приспособленности, основанная на использовании критерия *расстояние до условия*, успешно применяется для поиска значений для выполнения пути в расширенном конечном автомате. В данной работе этот подход расширяется и модифицируется для применения к автоматным программам. Как описывалось выше, для тестирования автоматной программы, необходимо учитывать помимо расширенного конечного автомата объекты управления с их спецификацией и аспекты взаимодействия автомата с его объектами управления. Более того, в данной работе поставлена другая задача применения оптимизационного алгоритма. В подходе [9] задачей является прохождение выбранного пути. Однако прохождение пути в расширенном конечном автомате недостаточно для создания теста:

1. Необходимо учитывать требования спецификации объектов управления. Если значения не удовлетворяют требованиям функций объектов управления, которые вызываются на переходах, то данный тест нельзя считать корректным, и поведение системы во время выполнения этого теста не определено.

2. Такой тест позволит проверить только неявные требования: отсутствие зависаний или исключительных ситуации (*Exceptions*) во время выполнения теста, но никак не направлен на обнаружение несоответствия реализации системы и ее спецификации.

В данной работе указанные проблемы решаются при помощи введения функции приспособленности, которая учитывает все аспекты системы – не только охранные условия на переходах, но также спецификации объектов

управления и спецификации системы в целом, заданные при помощи контрактов, как это было описано выше. Более того, это позволяет искать решения, которые не просто удовлетворяют всем необходимым условиям, но и обнаруживают несоответствия со спецификацией.

Далее описывается способ вычисления функции приспособленности для поставленной в данной работе задачи. Для вычисления приспособленности хромосомы относительно корректного выполнения заданного пути в автоматной модели следует учитывать два типа условий:

- охранные условия на переходах автомата;
- требования спецификации методов объектов управления, которые задействованы на переходах.

Все эти условия обязательны для выполнения. Хромосомы, которые нарушают любое из этих условий, не подходят для создания тестов, так как система в соответствии со своей спецификацией не должна поддерживать работу с этими значениями. В этом случае функция приспособленности должна оценить, насколько близка хромосома к тому, чтобы выполнить нарушенные условия. Это делается при помощи вычисления *расстояния до условия* этих условий.

Для последовательности переходов может быть задано большое число условий, поэтому *расстояние до условия* всего пути необходимо вычислять по отдельности, рассматривая условия на каждом переходе этого пути. Каждый переход описывается набором параметров:

1. Событие, по которому этот переход может произойти.
2. Охранное условие, которое должно быть выполнено для совершения перехода.

3. Действия на переходе: вызов методов объектов управления, получение значений внешних переменных из среды или изменение значений переменных модели.

4. Предусловия перехода, включающие в себя требования спецификации программы, которые должны быть выполнены для выполнения перехода, и требования спецификации объектов управления, которые должны быть выполнены для вызова методов объектов управления, задействованных при переходе.

5. Постусловия перехода, включающие в себя требования спецификации программы и ее объектов управления.

Таким образом, даже в рамках одного перехода может быть задействовано большое число условий. Для более точного вычисления *расстояния до условия* перехода в работе каждый переход разбивается на несколько меньших шагов:

1. Получение события, поиск возможных переходов и проверка их охранных условий.

2. Проверка предусловий перехода, выполнение перехода и заданных действий на переходе.

3. Проверка постусловий перехода.

При выполнении каждого из этих шагов могут быть обнаружены ошибки. Переход считается выполненным успешно, если все три шага выполнены успешно.

Таким образом, сценарий теста разбивается на переходы, а каждый переход разбивается на три шага. После этого исходная последовательность переходов рассматривается как последовательность шагов. Оценка приспособленности всей последовательности шагов вычисляется как сумма

оценок для каждого шага по-отдельности. Каждый шаг оценивается по формуле вычисления *расстояния для условия*.

Отметим, что шаги выполняются последовательно и что выполненность шагов в начале пути важнее, чем в его конце. Например, если выполнены условия всех шагов, кроме первого, то сумма *расстояний до условия* будет небольшой, так как для всех, кроме первого шага, это значение будет равно нолю. На практике эта хромосома не позволяет пройти ни одного шага, так как для того, чтобы успешно пройти второй шаг, необходимо выполнить все условия на первом шаге. Поэтому в предложенном подходе *расстояния до условия* шагов суммируются с учетом местоположения этих шагов в пути – используется взвешенная сумма:

$$\textit{приспособленность пути} = \sum_{i=0..m-1} f_i * d_i,$$

где m – число шагов в пути, $m = n * 3$, здесь n – число переходов в пути;

f_i – *расстояние до условия* для i-го шага.

d_i – вес i-го с шага, $d_i = (m - i)^2$.

Если для одного шага задано несколько условий, то *расстояние до условия* этого шага вычисляется как сумма *расстояний до условия* каждого из этих условий.

2.2.4. Поиск значений для нарушения требований спецификации

Функция приспособленности, описанная выше, позволяет находить набор значений для прохождения заданного пути. На практике важно находить такие значения, при которых будет выявлено несоответствие между спецификацией и реализацией. Для этой цели при поиске значений необходимо учитывать требования спецификации, записанные при помощи контрактов в модели. В этом случае целью поиска будет такой набор значений, который будет

удовлетворять всем требованиям охранных условий и объектов управления и при этом нарушать какое-либо из требований спецификации автомата.

Для решения поставленной задачи предлагается использовать итерационный подход, так как надо пытаться нарушить условия спецификации на переходах из заданного пути по очереди, а не на всех переходах одновременно. Это объясняется тем, что если требуется нарушить условие, которое находится на втором переходе, то необходимо, чтобы все условия, относящиеся к первому переходу, были выполнены. Иначе, система уже после первого перехода будет находиться в некорректном состоянии, и нарушение требований второго перехода может оказаться следствием более ранних ошибок. Таким образом, функция приспособленности будет по-разному вычисляться для поиска ошибок на разных переходах. Для поиска ошибок на *k*-ом переходе требуется, чтобы все условия на переходах с номером, меньшим *k*, были уже выполнены.

Для поиска значений, которые приводят к обнаружению ошибок, в работе предлагается использовать значение, противоположное *расстоянию до условия*. Если условие нарушено, то значение будет ноль. Чем ближе кандидат к нарушению условия, тем меньше значение приспособленности для этого условия. Таким образом, условия спецификации добавляются в вычисление функции приспособленности, аналогично с охранными условиями и условиями объектов управления.

На первом шаге рассматривается только первый переход и рассматривается путь, состоящий из одного этого перехода. Выполняется некоторое число попыток обнаружить значения, приводящие к нарушениям на этом пути. Затем переходим к следующему шагу: добавляем второй переход к пути, и рассматриваем путь из двух переходов. Таким образом, процесс продолжается пока все переходы не будут включены в путь.

2.3. Анализ предложенного подхода и сравнение с существующими инструментами

Задача тестирования автоматной программы в предложенном подходе разбита на четыре этапа:

1. Разработчик вносит максимально возможную часть спецификации в автоматную модель, используя возможности расширенного конечного автомата и *JML*-контрактов.

2. Тестовые сценарии записываются в виде последовательности переходов автоматной модели.

3. Используя предложенный алгоритм, определяются соответствующие значения переменных для выполнения заданного сценария и генерируются тесты для запуска.

4. Тесты запускаются автоматически, во время их исполнения при помощи существующего инструмента проверяется выполненность требований спецификации, записанной в виде контрактов.

Первый и второй этапы выполняются вручную разработчиком. Третий и четвертый этапы выполняются автоматически при помощи разработанных и существующих инструментов, описанных в разд. 3.1.

Первый этап в любом случае необходим для разработки автоматной программы. Только второй этап выполняется исключительно для целей тестирования. На этом этапе разработчик описывает сценарии, на которых велика вероятность появления ошибок. С целью сохранения времени этот этап также можно выполнять автоматически – используя существующие алгоритмы для обхода расширенных конечных автоматов. Наиболее эффективно сочетать в тестовых наборах как автоматически сгенерированные пути, так и записанные вручную. Автоматически сгенерированные пути могут обходить автомат

целиком или же учесть неожиданные сценарии работы, в то время как пути, записанные вручную, будут описывать ожидаемые сценарии работы системы.

Разработаны различные инструменты для тестирования структурных и объектно-ориентированных программ. Можно рассматривать автоматную программу как обычную программу, например, как некий объект с набором доступных методов. Этот подход обеспечит возможность применения существующих решений, но он в значительно меньшей степени позволит понять, насколько система соответствует спецификации. Привычные метрики тестирования, такие как процент покрытия кода, не дают информацию о том, какая часть реальной системы была проверена. Возможность описывать сценарии в автоматных терминах и получать соответствующие тесты даст значительно более полную картину о состоянии системы.

Инструментов для тестирования автоматных программ на момент написания работы не было, а инструменты, созданные для работы с расширенными конечными автоматами, только частично решают эту проблему, как это описано в разд. 1.8.

Выводы по главе 2

Предложенный подход позволяет повысить надежность процесса разработки автоматных программ и обладает рядом важных преимуществ по сравнению с существующими инструментами:

- учитывает особенности автоматного подхода к разработке программ;
- позволяет находить ошибки во всей системе в целом;
- снижает трудоемкость тестирования при помощи инструментов для автоматизации создания и запуска тестов.

Глава 3. Пример использования

Первый раздел главы описывает известные и разработанные инструментальные средства для реализации предложенного подхода. Во втором разделе приводится описание апробации описанного подхода на примере разработки упрощенной модели банкомата – устройства для снятия денег с определенного счета. Пример начинается со спецификации задачи на естественном языке, а затем для данного примера выполняются все шаги предложенного в работе подхода.

3.1. Инструментальные средства

В данном разделе рассмотрены инструменты, которые могут понадобиться на различных этапах применения описанного подхода, и позволят сделать его эффективным. Предложено использовать существующие инструменты для построения модели и проверки условий. Также в рамках работы разработаны прототипы инструментов, которые позволят автоматизировать создание тестов и их последующий запуск. Разработанные инструменты предназначены для апробации предложенного подхода и оценки его эффективности, поэтому на данном этапе содержат много ограничений.

3.1.1. Построение автоматной модели

Для создания автоматной модели предлагается применить инструмент *yEd* [46], который позволяет удобно графически описывать расширенные конечные автоматы, содержащие переменные и охранные условия. Возможность задания дополнительных меток к переходам и состояниям можно использовать для записи *JML*-контрактов. Пользователь работает с графическим представлением расширенного конечного автомата, как показано на рис. 1.

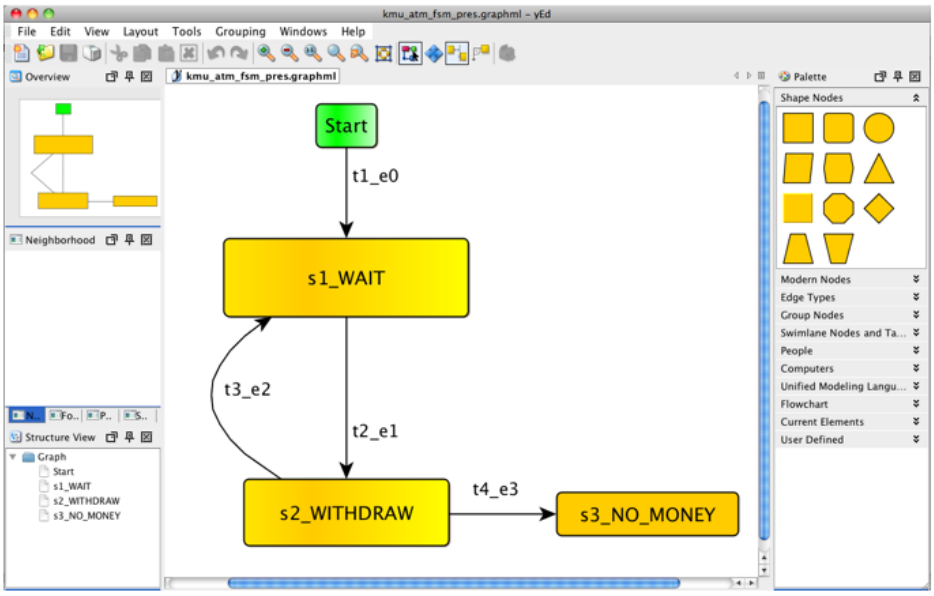

Рис. 1. Инструмент *yEd* для построения автоматной модели

Созданная модель хранится в *XML*-файле, который удобен для программной обработки и использования в последующих этапах. На рис. 2 приведен образец того, как выглядит *XML*-код модели, составленной в графическом редакторе:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <!--Created by yFiles for Java 2.7-->
  <key for="graphml" id="d0" yfiles.type="resources"/>
  <key attr.name="url" attr.type="string" for="node" id="d1"/>
  <key attr.name="description" attr.type="string" for="node" id="d2"/>
  <key for="node" id="d3" yfiles.type="nodegraphics"/>
  <key attr.name="url" attr.type="string" for="edge" id="d4"/>
  <key attr.name="description" attr.type="string" for="edge" id="d5"/>
  <key for="edge" id="d6" yfiles.type="edgegraphics"/>
  <graph edgedefault="directed" id="G">
    <node id="n0">
      <data key="d3">
        <y:ShapeNode>
          <y:Geometry height="30.0" width="42.0" x="136.0" y="0.0"/>
```

Рис. 2. Фрагмент *XML*-кода построенной модели

Существует ряд альтернативных инструментов для построения автоматных моделей: *MPS StateMachine Language* [47], *Unimod* [48] и т.д. [49]. Важно то, что как графические, так и текстовые редакторы, позволяют сохранять модель в *XML*-формате. Таким образом, на этом этапе можно использовать любой из этих инструментов, если осуществить преобразование созданного им *XML*-файла в формат, необходимый для примененения на

последующих этапах. Или, наоборот, расширить разработанные инструменты, добавив поддержку других форматов записи автоматных моделей.

Инструмент *yEd* выбран благодаря тому, что он позволяет не только описать расширенный конечный автомат, но также добавить *JML*-контракты в модель в виде меток на переходах и состояниях, в то время как *Unimod* и *MPS StateMachine Language* требуют модификации для поддержки записи контрактов.

3.1.2. Поиск значений внешних переменных для заданного пути

Разработанный инструмент *GenValueFinder* принимает на вход тестовый сценарий, записанный в виде последовательности переходов автоматной модели, и, используя генетический алгоритм, описанный в разд. 2.2, осуществляет поиск значений внешних переменных для прохождения этого пути. На вход инструмент принимает:

1. *XML*-файл с описанием автоматной модели.
2. Текстовый файл, содержащий последовательность переходов. Файл должен состоять из идентификаторов переходов, перечисленных в порядке прохождения пути:
   `t1 t2 t3 t4 t5 t6 t7`

Результаты работы *GenValueFinder*:

1. Текстовый файл `events.txt`, содержащий последовательность событий.
2. Текстовый файл `variables.txt`, содержащий значения внешних переменных в порядке их появления на заданном пути.
3. Вспомогательный файл `actions.txt`, перечисляющий действия на переходах и требования контрактов на заданном пути.

Если за определенное время значения переменных не удалось подобрать, то программа выдает ошибку. Инструмент *GenValueFinder* разработан на базе библиотеки *MBT* [39]. Библиотека *MBT* осуществляет работу с моделями, описанными при помощи инструмента *yEd*, предоставляя удобные программные интерфейсы для взаимодействия.

3.1.3. Генерация кода теста для запуска

Найденные значения переменных и последовательность событий позволяют сгенерировать код теста, пригодный для запуска. Для этого **разработан** инструмент *TestGenerator*, который принимает на вход файлы, полученные в результате работы программы *GenValueFinder*:

1. Текстовый файл, содержащий значения внешних переменных в порядке и появления на заданном пути.
2. Вспомогательный файл, содержащий действия на переходах и контракты.

Результат работы *TestGenerator*: файл `GeneratedTest.java`, содержащий *Java*-код, эмулирующий действия автомата на заданном пути. Полученный код пригоден для автоматической проверки.

3.1.4. Запуск тестов и проверка выполнения контрактов

Проверка выполнения контрактов выполняется при помощи существующего инструмента *JML Runtime Assertion Checker* [42], который входит в стандартный набор утилит при установке *JML*. Инструмент в режиме реального времени проверяет, что все контракты, добавленные в *Java*-код в виде аннотаций, выполнены. Для использования этого инструмента необходимо:

1. Скомпилировать файл `GeneratedTest.java` компилятором `jmlc`.

2. Запустить полученный файл `GeneratedTest.class` инструментом `jmlrac`.

В случае обнаружения ошибки создается исключительная ситуация (*exception*), которая позволяет проанализировать несоответствие спецификации и реализации.

3.2. Разработка устройства для снятия денег

Необходимо разработать устройство для снятия денег со счета, соответствующее приведенной в разд. 3.1.1 спецификации.

3.2.1. Спецификация системы

Спецификация на естественном языке:

1. Банкомат позволяет пользователям снимать деньги с определенного счета.
2. В начале на счету находится сумма от 0 до 100 000.
3. Пользователь может снимать деньги произвольное число раз, пока на счету неотрицательная сумма.
4. Ввод суммы для снятия происходит с клавиатуры, пользователь может ввести число от 1000 до 15000;
5. В день со счета может быть снято не более 50000.

Спецификация в таком виде удобна для разработчика и заказчика, так как написана на естественном языке. Однако, такую спецификацию можно проверять и тестировать только вручную.

3.2.2. Выделение объектов управления и их спецификации

Для формализации спецификации необходимо выделить спецификацию объектов управления и требования к логике работы системы, которые будут описаны при помощи автоматов.

Как видно из текста спецификации, в системе задействованы два объекта управления:

1. *Счет* отвечает за взаимодействие с банковским счетом: вернуть деньги на счет, снять деньги со счета.
2. *Клавитура* отвечает за взаимодействие с пользователем: ввод с клавиатуры устройства суммы для снятия.

Выделим требования спецификации, которые относятся к этим объектам управления, и отдельно запишем те требования, которые описывают логику работы системы и должны быть учтены при построении автоматной модели.

Счет:

- в начале на счету сумма от 0 до 100 000.

Клавиатура:

- пользователь может ввести число от 1000 до 15000.

Автоматная модель:

- банкомат позволяет пользователям снимать деньги с определенного счета;
- пользователь может снимать деньги произвольное число раз, пока на счету неотрицательная сумма;
- в день со счета может быть снято не более 50000.

3.2.3. Построение автомата, содержащего только состояния, переходы и события

Разработка автоматной программы начинается с построения автоматной модели. Для описанной системы можно выделить следующие состояния:

1. Инициализация.
2. Ожидание.
3. Снятие денег.
4. Отсутствие денег.

Далее необходимо определить возможные события в системе и описать реакцию системы на эти события – задать последовательность переходов. Возможные события в системе:

- *e0* – инициализация;
- *e1* – введена сумма;
- *e2* – операция выполнена;
- *e3* – ошибка.

На рис. 3 показан автомат, описывающий работу системы.

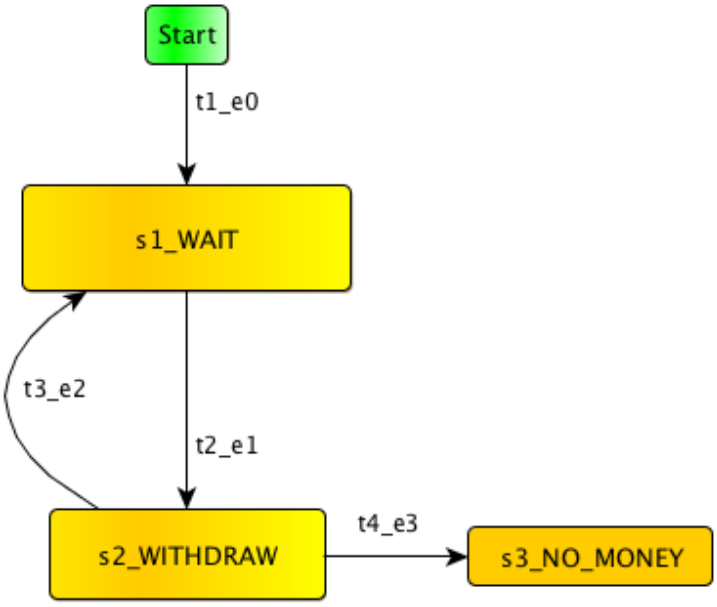

Рис. 3. Конечный автомат

Необходимо учесть, что такая модель позволяет описать только основные принципы работы, и только малая часть спецификации включена в нее:

- банкомат позволяет пользователям снимать деньги с определенного счета.

Большая часть логики включена в реализацию объектов управления. Например, из автоматной модели не ясно, когда операция снятия происходит успешно, а когда возникает ошибка. Таким образом, это требование спецификации не формализовано, а скрыто в реализации объекта управления. Включить большую часть логики в модель позволяет использование расширенного конечного автомата.

3.2.4. Построение расширенного конечного автомата и добавление контрактов

Расширенный конечный автомат позволяет задавать переменные, изменять из значения на переходах и использовать их в охранных условиях на переходах. Это дает возможность включить в модель переменные, соответствующие количеству денег на счету и введенному пользователем числу.

На рис. 4 представлен расширенный конечный автомат, описывающий логику работы системы.

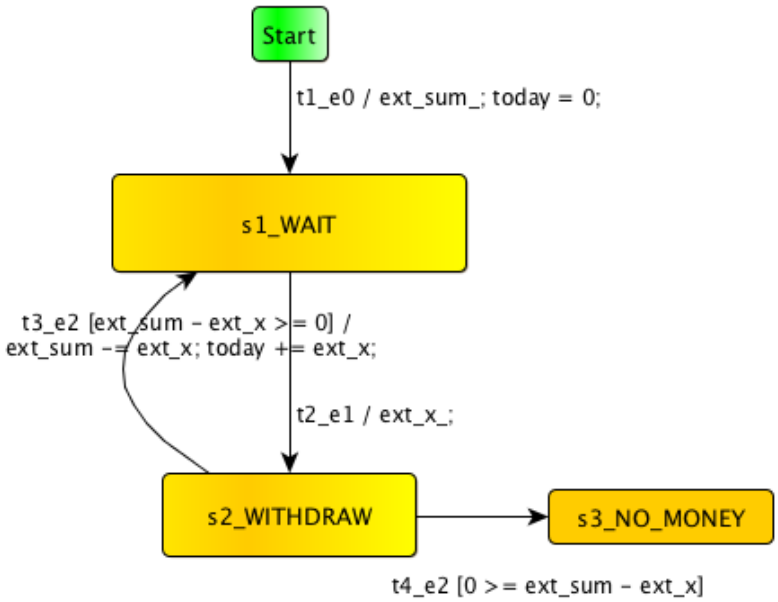

Рис. 4. Расширенный конечный автомат

Переменные в модели делятся на внутренние и внешние. Внутренние переменные определяются в самой модели, внешние – получают свои значения из среды с помощью объектов управления. В данной модели присутствуют две внешние переменные:

1. `ext_sum` – количество денег на счету. В момент начала работы начальное значение считывается со счета при помощи соответствующего объекта управления на переходе `t1`. Во время каждой транзакции это значение уменьшается на соответствующее число.

2. `ext_x` – сумма для снятия во время транзакции, которую вводит пользователь. В модель это значение попадает при помощи объекта управления *Клавиатура*. Во время работы программы это число может считываться множество раз – каждый раз, когда выполняется переход `t2`.

Внутренняя переменная `today` используется для суммирования снятых за сеанс работы денег. Именно добавление этой переменной позволяет записать условие, которое ограничивает число снятых денег за один сеанс работы.

Как было сказано ранее, это требование можно добавить либо при помощи нового состояния и охранных условий на переходах или при помощи использования контрактов. Так как данное требование спецификации относится не к логике работы, а к ограничениям системы, разумнее добавить его при помощи контрактов.

Также и требования к объектам управления можно включить в модель, создав новые состояния, но это внесет путаницу в модель и сделает ее более сложной. Поэтому лучше добавить эти требования при помощи контрактов.

На рис. 5 приведена модель, представленная в виде расширенного конечного автомата и требований, записанных при помощи контрактов.

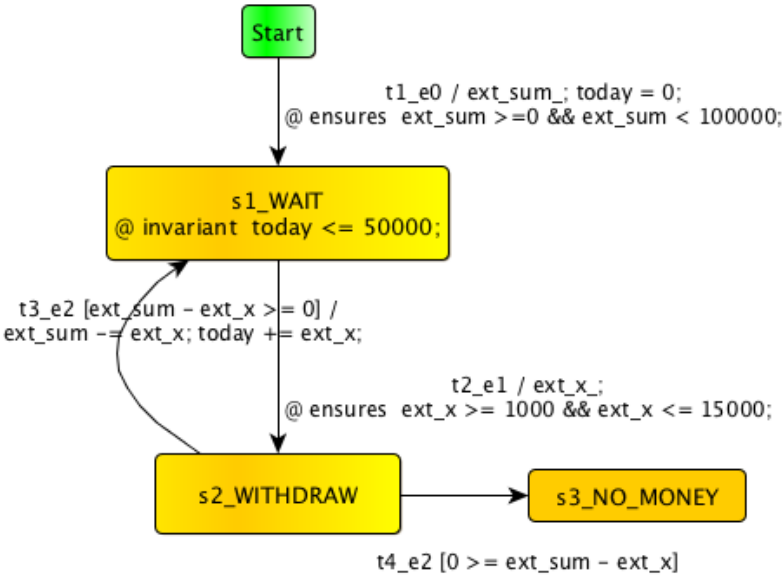

Рис. 5. Расширенный конечный автомат, содержащий контракты

В модель включены требования спецификации к объектам управления:

- *Клавиатура*:

  ```
  @ensures
  ext_x >= 1000 && ext_x <= 15000
  ```

- *Счет*:

  ```
  @ensures
  ext_sum >= 0 && ext_sum <= 100000
  ```

Требование спецификации, ограничивающее число снятых денег за сеанс работы, добавлено как контракт к состоянию:

- `@invariant today <= 50000`.

В таблице перечислены все требования системы, и то, как они формализованы в построенной модели.

Таблица. Способ формализации требований спецификации системы

Требование	**Способ записи**	**Формальное представление**
Банкомат позволяет пользователям снимать деньги с определенного счета	Конечный автомат	Рис. 3.
Изначально на счету сумма от 0 до 100 000	Контракты	`@ensures ext_sum >= 0 && ext_sum <= 100000`
Пользователь может снимать деньги произвольное количество раз, пока на счету неотрицательная сумма	Расширенный конечный автомат, охранные условия на переходах	Рис. 4.
Ввод суммы для снятия происходит с клавиатуры, пользователь может ввести число от 1000 до 15000	Контракты	`@ensures ext_x >= 1000 && ext_x <= 15000`
В день со счета может быть снято не более 50000	Контракты	`@invariant today <= 50000`

3.2.5. Описание тестовых сценариев

Проанализировав текстовую спецификацию системы, предложим вероятные сценарии использования и сделаем тесты на их основе. Сначала запишем несколько сценариев на естественном языке:

1. **Три раза** снимаются деньги со счета и на счету **заканчиваются** средства на четвертой попытке.

2. **Двадцать раз** снимаются деньги со счета и на счету **не заканчиваются** средства.

Следующий шаг – формальная запись тестов при помощи терминов автоматной модели. Тестовые сценарии, записанные как последовательность переходов:

1. `path1.txt: t1, t2, t3, t2, t3, t2, t3, t2, t4.`
2. `path2.txt: t1, t2, t3, t2, t3, t2, t3, t2, t2, t3, t2, t3, t2, t3, t2, t2, t3, t2, t3, t2, t3, t2, t2, t3, t2, t3, t2, t3, t2, t2, t3, t2, t3, t2, t3, t2, t2, t3, t2, t3, t2, t3, t2, t2, t3, t2, t3, t2, t4.`

3.2.6. Поиск значений переменных для заданных путей

В первом тестовом сценарии задействовано пять внешних переменных:

1. `ext_sum` – начальная сумма на счету;
2. `ext_x1` – пользователь ввел первый раз;
3. `ext_x2` – пользователь ввел второй раз;
4. `ext_x3` – пользователь ввел третий раз;
5. `ext_x4` – пользователь ввел четвертый раз.

Для поиска подходящих значений использован разработанный в рамках работы прототип инструмента:

```
./GenValueFinder model.xml path1.txt
```

Подходящие значения найдены за 10 секунд:

- файл `events.txt`: `e0 e1 e2 e1 e2 e1 e2 e1 e2`;
- файл `variables.txt`: `33177 13115 14485 4382 8513`

Во втором тестовом сценарии задействованы 22 внешние переменные и поиск значений для реализации описанного пути при помощи прототипа инструмента занимает 11 минут.

3.2.7. Создание кода тестов и их запуск

Код теста генерируется запуском инструмента *TestGenerator*, в результате работы которого создается файл с *Java*-кодом, эмулирующим действия системы при выполнении тестового сценария. Исходный код файла с тестом приведен в приложении.

Для проверки корректности работы системы на данном сценарии необходимо скомпилировать тест командой `jmlc` и запустить при помощи команды `jmlrac`. Запуск теста выдает следующий результат:

```
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInvariantError: by method
GeneratedTest.transition5@post<File "GeneratedTest.java",
line 34, character 7>
```

Данное описание ошибки позволяет понять, что условие `invariant` было нарушено после выполнения пятого перехода. Таким образом, при найденных значениях внешних переменных на тестируемом сценарии обнаружено несоответствие спецификации и реализации.

Выводы по главе 3

Использование предложенного подхода для разработки автоматной программы позволило проиллюстрировать и подтвердить его заявленные достоинства:

- удобство использования расширенных конечных автоматов и контрактов для построения модели;

- описание тестовых сценариев при помощи последовательности переходов;
- автоматизация поиска значений внешних переменных при помощи применения генетических алгоритмов;
- возможность проверки выполненности контрактов во время запуска программы.

В приведенном примере про помощи запуска сгенерированных тестов обнаружено несоответствие реализации программы и ее спецификации.

Заключение

В рамках данной работы были выполнены следующие задачи:

1. Изучены существующие подходы к проверке корректности автоматных программ и предложено использовать тестирование для проверки систем в целом.
2. Предложен подход к тестированию автоматных программ, предполагающий ручное или автоматическое составление тестовых сценариев и генерации тестов для них.
3. Разработан метод нахождения входных параметров для выполнения заданного сценария в автоматной модели при помощи использования генетических алгоритмов.
4. Предложено использовать контракты описания требований спецификации к взаимодействию автоматной модели с объектами управления.
5. Разработаны инструменты и проанализированы существующие инструменты, и разработан прототип инструмента для автоматизации предложенного подхода.
6. Проверена эффективность предложенного подхода на примере разработки устройства для снятия денег со счета.

Источники

1. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2009. – 176 с. http://is.ifmo.ru/books/_book.pdf

2. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ . Model Checking. М.: МЦНМО, 2002. – 416 с.

3. *Myers G.* The Art of Software Testing. NJ: John Wiley & Son. Inc, 2004.

4. *McMinn P.* “Search-based software test data generation: a survey: Research Articles /Software Testing, Verification & Reliability. 2004. 14(2), pp. 105–156.

5. *Степанов О. Г.* Методы реализации автоматных объектно-ориентированных программ. Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2009. http://is.ifmo.ru/disser/stepanov_disser.pdf

6. *Meyer B.* Applying design by contract //Computer. 25. 1992 (10) pp. 40–51.

7. *Шалыто А. А.* Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода. http://is.ifmo.ru/science/1/

8. *Шалыто А. А., Гуров В. С., Мазин М. А.* Мобильные системы. http://is.ifmo.ru/science/MD-Mobile.pdf

9. *Kalaji A., Hierons R., Swift S.* Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM) /Software Testing, Verification, and Validation (ICST). 2nd International IEEE Conference. 2009. Denver, Colorado.

10. *Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.

11. *Simon G. A.* An extended finite state machine approach to protocol specification /Proc. of 2-nd International Workshop on Protocol Specification, Testing and Verification. 1982, pp. 113–133.

12. *Кулямин В.В.* Тестирование на основе моделей. http://panda.ispras.ru/~kuliamin/lectures-mbt/

13. *Кулямин В.В.* Методы верификации программного обеспечения. http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf

14. Веб-сайт проекта *Code Contracts* компании *Microsoft*. http://research.microsoft.com/en-us/projects/contracts/

15. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.

16. *Кулямин В. В.* Критерии тестового покрытия, основанные на структуре контрактных спецификаций // Труды ИСП РАН. Подход UniTESK: итоги и перспективы. 14(1). 2008, с.89–107. http://panda.ispras.ru/~kuliamin/docs/SpecCoverageCriteria-2008-ru.pdf

17. *Standard ECMA-367.* Eiffel: Analysis, Design and Programming Language. 2nd edition (June 2006). www.ecma-international.org/publications/standards/Ecma-367.htm

18. *Leavens G. T., Baker A. L., Ruby C.* Preliminary design of JML: A behavioral interface specification language for Java. Iowa State Univ. Dept. of Comput. Sci. Tech. Rep. 98-06u, 2003.

19. *Веб-сайт проекта Contract4j* http://www.contract4j.org/contract4j

20. *Клебанов А. А.* Генерация исходного кода доказуемо-корректных *JavaCard*-программ с использованием автоматного подхода. СПбГУ ИТМО. Бакалаврская работа. 2008. http://is.ifmo.ru (раздел «Работы»).

21. *Технология* Java Card. Официальный сайт. http://java.sun.com/javacard/

22. *Poll E.* From Finite State Machines to Provably Correct Java Card Applets. www.cs.ru.nl/E.Poll/papers/

23. *Cokand D.R., Kiniry J.R.* ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Nijmegen Inst. for Computing and Inform. Sci., Tech. Rep. NIII-R0413. 2004.

24. *Тестирование программного обеспечения*. Wikipedia. http://ru.wikipedia.org/wiki/Тестирование_программного_обеспечения

25. *Дастин Э., Рэшка Д., Пол Д.* Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация. М.: Лори, 2003.

26. *Монахов А., Петренко А., Бритвина Е., Петренко О., Грошев С.* Тестирование на основе моделей. http://www.osp.ru/os/2003/09/183388/

27. *Holland J. H.* Adaptation in natural and artificial systems. University of Michigan Press. Ann Arbor, 1975.

28. *Koza J.* Genetic Programming: On the programming of Computers by Means of Natural Selection. Cambridge: MIT Press, 1992.

29. *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы. М: Физматлит, 2006. – 320 с.

30. *Wegener J., Buhr K., Pohlheim H.,* Automatic test data generation for structural testing of embedded software systems by evolutionary testing /In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002). NY: Morgan Kauf-mann. 2002, pp. 1233–1240.

31. *Pargas R. P., Harrold M. J., Peck R. R.* Test-data generation using genetic algorithms //The Journal of Software Testing, Verification and Reliability. 1999. 9 pp. 263–282.

32. *Duale Y., Ümit Uyar M.* A Method Enabling Feasible Conformance Test Sequence Generation for EFSM Models //IEEE Transactions on Computers. V.53. 2004. № 5, p. 614–627.

33. *Derderian K., Hierons R. M., Harman M., Guo Q.* Estimating the Feasibility of Transition Paths in Extended Finite State Machines // Automated Software Engineering. 17. 2010. 1, pp. 33–56.

34. *Koh, L.-S., Liu M.T.* Test path selection based on effective domains //In Proceedings International Conference «Network Protocols». 1994. Boston.

35. *Jacky J., Veanes M., Campbell C., Schulte W.* Model-Based Software Testing and Analysis with C#. Cambridge University Press. 2008.

36. *NModel web site.* http://www.codeplex.com/NModel.

37. *Mark U., Legeard B.* Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann. 2007

38. *Веб-сайт инструмента ModelJUnit.* http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

39. *Веб-сайт инструмента MBT.* http://mbt.tigris.org/

40. *Баранцев А.В., Бурдонов И.Б., Демаков А.В. и др.* Подход UniTesK к разработке тестов: достижения и перспективы. М.: Труды Института системного программирования РАН. 2004. №.5.

41. *Bourdonov I., Kossatchev A., Kuliamin V., and Petrenko A.* UniTesK Test Suite Architecture //Proc. of FME 2002. LNCS 2391. Springer-Verlag. 2002, pp. 77–88.

42. *Cheon Y., Leavens G. T.* A Runtime Assertion Checker for the Java Modeling Language (JML) /Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02). Las Vegas. Nevada. CSREA Press. 2002, pp. 322–328.

43. *Lai R.* A survey of communication protocol testing //Journal of Systems and Software. 2002. 62(1), pp. 21–46.

44. *Wegener J., Baresel A., Sthamer H.* Evolutionary test environment for automatic structural testing //Information and Software Technology. 2001. 43(14), pp. 841–854.

45. *Tracey N., Clark J., Mander K., McDermid J.* An automated framework for structural test-data generation /In Proceedings 13th IEEE International Conference Automated Software Engineering. 1998.

46. *Веб-сайт инструмента yEd.* www.yworks.com/en/products_yed_about.html

47. *MPS User's Guide.* http://www.jetbrains.net/confluence/display/MPS/MPS+User%27s+Guide

48. *Гуров В. С., Мазин М. А.* Веб-сайт проекта *UniMod*. http://unimod.sourceforge.net/

49. *Кафедра «Технологии программирования». Раздел «Проекты».* http://is.ifmo.ru/projects/

Приложение. Код сгенерированного теста

```
public class GeneratedTest {
private /*@ spec_public @*/ int today = 0;
private /*@ spec_public @*/ int ext_x = 0;
private /*@ spec_public @*/ int ext_sum = 0;
  //@ public invariant today <= 25000;

/*@ ensures  ext_sum >=0 && ext_sum < 100000;
  @*/
public void transition1() {
    ext_sum = 33177;
    today = 0;
}

/*@ ensures  ext_x >= 1000 && ext_x <= 15000;
  @*/
public void transition2() {
    ext_x = 13115;

}

public void transition3() {
    ext_sum -= ext_x;
    today += ext_x;

}

/*@ ensures  ext_x >= 1000 && ext_x <= 15000;
  @*/
public void transition4() {
    ext_x = 14485;
}

public void transition5() {
    ext_sum -= ext_x;
    today += ext_x;

}

/*@ ensures  ext_x >= 1000 && ext_x <= 15000;
  @*/
public void transition6() {
    ext_x = 4382;
}
```

```
public void transition7() {
    ext_sum -= ext_x;
    today += ext_x;
}

/*@ ensures  ext_x >= 1000 && ext_x <= 15000;
  @*/
public void transition8() {
    ext_x = 8513;
}

public static void main(String[] args) {
    GeneratedTest test = new GeneratedTest();
    test.transition1();
    test.transition2();
    test.transition3();
    test.transition4();
    test.transition5();
    test.transition6();
    test.transition7();
    test.transition8();
}


}
```