

Санкт-Петербургский государственный университет информационных
технологий, механики и оптики

На правах рукописи

Степанов Олег Георгиевич

**Методы реализации автоматных
объектно-ориентированных программ**

Специальность 05.13.11. Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени кандидата технических наук

Научный руководитель –
доктор технических наук,
профессор А.А. Шалыто

Санкт-Петербург

2009

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ.....	2
ВВЕДЕНИЕ	5
ГЛАВА 1. МЕТОДЫ РЕАЛИЗАЦИИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ И АВТОМАТНЫХ ПРОГРАММ.....	11
1.1. Структура объектно-ориентированных программ.....	11
1.2. Методы повышения качества объектно-ориентированных программ	12
1.3. Автоматное объектно-ориентированное программирование ..	20
1.4. Методы повышения качества автоматных объектно-ориентированных программ	26
Выводы по главе 1	30
ГЛАВА 2. ФОРМАЛИЗАЦИЯ ТРЕБОВАНИЙ К АВТОМАТНЫМ ОБЪЕКТНО-ОРИЕНТИРОВАННЫМ ПРОГРАММАМ.....	32
2.1. Формализация требований к автоматным программам	32
2.2. Интерфейсы автоматных объектов.....	35
2.3. Автоматное программирование по контрактам.....	38
2.4. Способы проверки автоматных контрактов.....	43
2.5. Динамический метод проверки спецификаций автоматных программ	44
2.6. Формализация требований к автоматным программам	52
Выводы по главе 2.....	54
ГЛАВА 3. ВНЕСЕНИЕ ИЗМЕНЕНИЙ В АВТОМАТНЫЕ ПРОГРАММЫ ..	55
3.1. Внесение изменений в автоматные программы.....	55
3.2. Классификация изменений автоматных программ.....	57
3.3. Описание базовых изменений автоматов.....	57

3.3.1. Добавление состояния	58
3.3.2. Удаление состояния.....	58
3.3.3. Установка начального состояния.....	58
3.3.4. Снятие начального состояния	58
3.3.5. Добавление конечного состояния	58
3.3.6. Удаление конечного состояния.....	59
3.3.7. Добавление перехода.....	59
3.3.8. Изменение события на переходе.....	59
3.3.9. Изменение условия на переходе	59
3.3.10. Удаление перехода	59
3.3.11. Перемещение перехода	60
3.4. Рефакторинг автоматных программ	60
3.4.1. Группировка состояний.....	62
3.4.2. Удаление группы состояний	66
3.4.3. Слияние состояний	66
3.4.4. Выделение автомата	70
3.4.5. Встраивание вызываемого автомата.....	74
3.4.6. Переименование состояния.....	77
3.4.7. Перемещение воздействия из состояния в переходы	78
3.4.8. Перемещение воздействия из переходов в состояние	80
3.5. Метод внесения изменений в автоматные программы.....	81
3.6. Пример использования метода	83
Выводы по главе 3.....	90

ГЛАВА 4. ИНТЕГРАЦИЯ СПЕЦИФИКАЦИИ И КОДА АВТОМАТНЫХ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ.....	91
4.1. Автоматное программирование в динамических языках.....	94
4.1.1. Определение общих свойств автомата	96

4.1.2. Описание графов переходов	98
4.1.3. Пример декларации автомата.....	99
4.1.4. Использование библиотеки	103
4.1.5. Реализация библиотеки	104
4.1.6. Отладка программ.....	111
4.1.7. Взаимодействие с окружением	111
4.1.8. Формализация спецификаций автоматов	115
4.2. Интеграция спецификации и кода в мультязыковых средах	115
4.2.1. Описание языков в системе <i>JetBrains MPS</i>	115
4.2.2. Автоматное программирование в среде <i>JetBrains MPS</i>	120
4.2.3. Язык спецификации автоматов.....	123
4.2.4. Реализация языка спецификации автоматов	126
4.2.5. Пример использования языка спецификации автоматов ...	129
Выводы по главе 4.....	130
ГЛАВА 5. ВНЕДРЕНИЕ РЕЗУЛЬТАТОВ РАБОТЫ В ПРАКТИКУ	
РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ.....	131
5.1. Область внедрения	131
5.2. Использование системы <i>JetBrains MPS</i> в программе <i>YouTrack</i>	131
5.3. Автомат управления списком дефектов <i>IssueList.js</i>	133
5.4. Автомат управления выпадающей подсказкой <i>Suggest.js</i>	137
Выводы по главе 5.....	144
ЗАКЛЮЧЕНИЕ	145
ИСТОЧНИКИ.....	147

ВВЕДЕНИЕ

Актуальность темы. В настоящее время актуальной является проблема разработки качественного программного обеспечения (ПО). При разработке такого ПО важно учитывать различные аспекты качества, одним из которых является *корректность*: соответствие реализации программы заданной спецификации. Важно отметить, что в современных проектах по разработке ПО изменение спецификации даже внутри цикла разработки – обычное явление, вызванное уточнением понимания предметной области и изменением внешних условий. Другой особенностью современных программных проектов является регулярная перестройка исходного кода программы для облегчения поддержки изменений в спецификации и улучшения читаемости. Такая перестройка, при которой сохраняется поведение программы, называется рефакторингом.

Существующие технологии поддержания качества программных продуктов в таких условиях обладают рядом недостатков, основным из которых является то, что эти технологии основаны на тестировании. Тестирование не может дать гарантию отсутствия ошибок в программе. Технологии, основанные на верификации (формализованной проверке соответствия программы спецификации), в настоящее время недостаточно эффективны для использования в программах, написанных традиционным способом.

Эти недостатки сказываются на качестве модулей программ со сложным поведением, для которых тестирование оказывается наименее эффективным. В модулях со сложным поведением реакция на вызов зависит от внутреннего состояния модуля. При этом в России с 1991 года разрабатывается концепция автоматного программирования, которая позволяет эффективно разрабатывать модули со сложным поведением [1]. Частью этой концепции являются технологии совмещения автоматного и

объектно-ориентированного программирования. Это позволяет реализовывать модули со сложным поведением с использованием автоматного программирования в различных проектах, включая уже существующие.

Автоматные программы, в отличие от программ традиционного типа, могут быть эффективно верифицированы с помощью метода *Model Checking*, так как в таких программах первичными являются модели, а в традиционных – код. Однако в настоящий момент не разработаны методы, позволяющие интегрировать технологии реализации автоматных программ и их верификации в современные процессы разработки объектно-ориентированного ПО. В частности, недостаточно проработаны следующие вопросы:

- механизм эффективной формализации требований к автоматным программам;
- интеграция спецификации и исполняемого кода автоматных программ с учетом его верификации;
- синхронизация требований и реализации автоматных программ (односторонняя).

Исходя из этого, можно утверждать, что исследования, направленные на разработку методов реализации качественных автоматных объектно-ориентированных программ, весьма актуальны.

Целью диссертационной работы является создание методов реализации автоматных объектно-ориентированных программ, позволяющих интегрировать процессы разработки автоматных и объектно-ориентированных программ.

Основными задачами исследования является создание:

- метода формализации требований к автоматным объектно-ориентированным программам с использованием контрактов;
- метода интеграции спецификации и реализации автоматных объектно-ориентированных программ;
- метода модификации автоматных программ, уменьшающего число изменений, которые могут привести к появлению ошибок;
- каталога рефакторингов.

Методы исследования. В работе использованы методы теории автоматов, верификации, контрактного программирования и объектно-ориентированного проектирования.

Научная новизна. В работе получены следующие научные результаты, которые выносятся на защиту:

- метод формализации требований к автоматным объектно-ориентированным программам в виде формул темпоральной логики и контрактов, применяемых при статической и динамической верификации;
- метод интеграции формализованных требований и кода автоматных объектно-ориентированных программ на основе использования возможностей мультязыковой среды программирования;
- метод модификации автоматных программ, основанный на рефакторинге, уменьшающий число изменений, которые могут привести к появлению ошибок;
- каталог рефакторингов автоматных программ, и доказательство эквивалентности программ до и после рефакторинга.

Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами внедрения методов, предложенных в диссертации, на практике.

Практическое значение работы состоит в том, что все полученные результаты могут быть использованы, а некоторые уже используются, в практической деятельности компании *ООО «ИнтеллиДжей Лабс»* (Санкт-Петербург). Предложенные методы реализации упрощают разработку, поддержку и сопровождения автоматных объектно-ориентированных программ за счет интеграции спецификации и кода программы.

Внедрение результатов работы. Результаты, полученные в диссертации, используются на практике:

- в компании *ООО «ИнтеллиДжей Лабс»* при разработке системы учета дефектов *YouTrack*.
- в учебном процессе по курсу «Применение автоматов в программировании» кафедры «Компьютерные технологии» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СПбГУ ИТМО).

Апробация диссертации. Основные положения диссертационной работы докладывались на III Межвузовской конференции молодых ученых (СПб., 2006 г.), XXXVI научной и учебно-методической конференции профессорско-преподавательского и научного состава СПбГУ ИТМО (2007 г.), конференциях Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE) (СПбГУ, 2008 г.), 5th Central and Eastern European Software Engineering Conference in Russia (SECR) (М., 2009 г.).

Публикации. По теме диссертации опубликовано четыре статьи, в том числе две в журналах из перечня ВАК. В работах, выполненных в соавторстве, автором получено не менее половины результатов.

Гранты. Диссертация выполнялась в ходе работы над грантом для студентов, аспирантов вузов и академических институтов, находящихся на территории Санкт-Петербурга, который проводился Администрацией Санкт-Петербурга (2008 г.). Материалы диссертации используются в научно-исследовательской работе по теме «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования», которая победила в конкурсе НК-421П «Проведение научных исследований научными группами под руководством кандидатов наук» по направлению «Информатика», который был объявлен в 2009 году Федеральным агентством по образованию по Федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы.

Структура диссертации. Диссертация изложена на 150 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит 69 наименований. Работа содержит 24 рисунка и одну таблицу. В первой главе приведен обзор современных методов реализации объектно-ориентированных и автоматных программ. Во второй главе предложен метод формализации требований к автоматным программам на основе использования контрактов, темпоральных спецификаций, а также статической и динамической верификации. В третьей главе предлагаются метод рефакторинга автоматных программ и метод их модификации, уменьшающий число изменений, которые могут привести к появлению ошибок. В четвертой главе предложен метод интеграции спецификации и кода автоматных программ на основе использования возможностей мультязыковых сред. В пятой главе излагаются результаты внедрения

предложенных методов разработки программ в практику разработки автоматных объектно-ориентированных программ.

ГЛАВА 1. МЕТОДЫ РЕАЛИЗАЦИИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ И АВТОМАТНЫХ ПРОГРАММ

1.1. Структура объектно-ориентированных программ

Большинство разрабатываемых в настоящее время программ строится на основе объектно-ориентированного подхода. Известны различные методы объектно-ориентированного программирования [2]. В настоящей диссертации автор придерживается модели, описанной Б. Мейером в книге [3], основные концепции которой перечислены ниже.

Объектная декомпозиция. Объектно-ориентированные программы декомпозированы на объекты – *сущности*, реализующие определенные роли в программе. Эти роли определяются набором сервисов, которые они предоставляют в программе, и называются *абстрактными типами данных* (АТД). Формально АТД определяет множество операций, которые можно применить к объекту данного типа. С объектной декомпозицией связано понятие *инкапсуляции*: объект содержит все необходимое для исполнения его роли: данные и исполняемый код. При этом они являются скрытыми для других объектов программы, которые могут взаимодействовать с этим объектом только через операции, определенные АТД.

Классы. Классы одной стороны представляют собой единственный вид модуля в объектно-ориентированных программах, а с другой – тип, реализующий АТД. Классы могут быть *абстрактными*. При этом АТД реализуется частично. Класс определяет конкретные действия, которые объект выполнит при применении к нему одной из операций, описанных в АТД класса.

Объекты. Объекты являются экземплярами неабстрактных классов. Класс является типом и определяет набор операций, соответствующих

объекту. При объектно-ориентированном программировании вся программа или ее часть представляет собой набор объектов, взаимодействующих между собой путем выполнения операций (в некоторых других описаниях парадигмы объектно-ориентированного программирования принято говорить, что объекты *обмениваются сообщениями*).

Наследование. Этот механизм в объектно-ориентированных программах позволяет повторно использовать класс в различных сценариях за счет переопределения части его операций. С одной стороны, наследование обеспечивает возможность создания модифицированных копий модулей без изменения их исходных определений, а с другой стороны – возможность создания *подтипов*, реализующих *полиморфизм* (взаимозаменяемость объектов с одинаковым интерфейсом) в объектно-ориентированных программах.

При проектировании объектно-ориентированных программ определяются роли объектов в ней и структура их взаимодействия. Затем на основании этой информации строятся классы. Часто при проектировании объектно-ориентированных программ строятся ее модели, которые часто описываются на языке моделирования *UML* [4]. В хорошо спроектированных программах даже самая сложная реализация скрыта от использующих ее объектов простым интерфейсом.

1.2. Методы повышения качества объектно-ориентированных программ

При создании объектно-ориентированных программ методы повышения качества используются на всех этапах: от проектирования до внедрения. Эти методы могут быть различны, и направлены на уменьшение числа ошибок как при написании программы, так при ее изменениях.

Одной из причин частого изменения и уточнения требований к программным продуктам в процессе разработки является сложность

современных программ. Другой важной причиной является высокая скорость изменения технологий [5]. Технологии, используемые при разработке современного ПО, рассчитаны на высокую степень изменения требований. Рассмотрим некоторые из них.

Гибкие методологии разработки. В настоящее время широкое распространение получили *гибкие (agile)* методологии разработки ПО (например, *XP* [6], *Scrum* [7] и *MSF for Agile Software Development Process* [8]). Такие методологии ориентированы на постепенное уточнение требований. В процессах, построенных на основе этих методологий, цикл разработки программного продукта обычно делится на множество коротких *итераций*, в рамках каждой из которых разрабатывается часть программного продукта, реализующая небольшой набор связанных требований. Анализ функциональности, полученной в результате итерации, может привести к изменению требований. Важной чертой итераций является то, что каждая из них включает фазы анализа, разработки, тестирования и внедрения.

Объектно-ориентированное проектирование ПО с учетом изменений требований. Одним из требований к архитектуре объектно-ориентированной программы является ее устойчивость к изменениям. Программа должна быть спроектирована так, чтобы большинство изменений в требованиях приводило к минимальным модификациям ее реализации. Другим важным требованием к объектно-ориентированной архитектуре является возможность повторного использования классов.

Хорошие решения в объектно-ориентированных программах фиксируются в виде *паттернов* [9, 10].

Среди паттернов объектно-ориентированного проектирования, в применении к теме данной диссертации, особое значение представляют следующие [4]:

- *информационный эксперт* определяет базовый принцип распределения ролей в программе. Этот паттерн устанавливает, что роли должны быть назначены объекту, который владеет максимумом необходимой информации для выполнения роли;
- *слабая связанность* устанавливает следующие свойства: малое число зависимостей между классами и подсистемами, слабую зависимость одного класса (подсистемы) от изменений в другом классе (подсистеме), высокую степень повторного использования подсистем;
- *сильное сцепление* требует, чтобы сервисы, предоставляемые одним классом (подсистемой) были связаны друг с другом.

Таким образом, при проектировании объектно-ориентированной программы схожие обязанности группируются в одном классе. При этом связи между различными классами стараются уменьшить.

Рефакторинг. Какой бы продуманной ни была архитектура, при изменении требований часто приходится модифицировать ее код. При произвольном изменении программы велик риск внесения ошибок. Одним из распространенных приемов, используемых при модификации кода, является *рефакторинг* – полное или частичное преобразование структуры программы с сохранением ее поведения [11]. Примерами рефакторингов в объектно-ориентированных программах являются переименование класса, выделение интерфейса, перемещение метода и т.д.

Каждый вид рефакторинга реализует определенное изменение структуры программы, состоящее из набора небольших и технически простых шагов, позволяющих сделать процесс внесения изменений контролируемым. Следование правилам выполнения рефакторинга обеспечивает неизменность поведения программы, в чем можно убедиться, проведя процедуру тестирования. В последнее время широкое

распространение получили средства автоматического выполнения рефакторингов. Поддержка рефакторинга в каком-либо виде является частью практически любой современной интегрированной среды разработки. Автоматизация рефакторинга позволяет свести вероятность изменения поведения программы при его выполнении практически к нулю.

Использование рефакторинга позволяет организовать процесс изменения поведения программы следующим образом: сначала производится рефакторинг для подготовки к изменению, затем выполняется само изменение поведения. При этом рефакторинг выбирается таким образом, чтобы минимизировать модификации кода, изменяющие поведение программы.

Тестирование. Тестированием программного обеспечения называется процесс выявления в нем ошибок. Важно отметить, что успешное тестирование не может гарантировать отсутствие ошибок в программе («Тестирование программ, – отмечал Эдсгер Дейкстра, – можно использовать для того, чтобы показать наличие ошибок и *никогда* – для того чтобы показать их отсутствие!»). Существует много видов тестирования программного обеспечения [12]. Ниже рассматриваются несколько основных видов.

При тестировании ПО специалист по качеству может либо учитывать информацию о реализации тестируемого кода (так называемый метод «стеклянного» (*glass box*) или «белого» (*white box*) ящика), либо не учитывать – метод «черного ящика» (*black box*). При использовании метода «черного ящика» выделяют *функциональное тестирование*, при котором каждая функция программы тестируется путем ввода ее входных данных и анализа выходных. При этом структура функции учитывается редко. В противоположность функциональному тестированию для метода «стеклянного ящика» выделяют *структурное тестирование*, главной идеей которого является правильный выбор тестируемого программного пути. Так

как все программные пути протестировать обычно невозможно, специалисты по тестированию выделяют среди них группы, которые требуется протестировать обязательно. Для отбора таких групп применяются специальные критерии, называемые *критериями охвата (coverage criteria)*. Чаще всего используются критерии охвата *строк, ветвлений и условий*. Эти критерии устанавливают, что набор тестов должен покрывать все строки, ветвления или комбинации ветвлений программы соответственно.

Еще одним важным видом тестирования является *регрессионное тестирование*. Этот вид тестирования направлен на то, чтобы после исправления ошибки в программе проверить следующие два условия:

- ошибка действительно исправлена: проводится тест по сценарию, выявлявшему ошибку;
- исправление ошибки не привело к появлению других: после исправления ошибки тестируется вся программа.

Описанные виды тестирования требуют запуска программы (*динамическое тестирование*). Однако часть ошибок можно выявить при анализе кода программы, не запуская его. Такое тестирование называется *статическим*. Значительная часть такого тестирования выполняется компилятором при сборке программы, особенно если речь идет о языках со статической типизацией [13], в которых система типов накладывает значительные ограничения на код программы.

В современных проектах по разработке программного обеспечения особое место занимает *модульное тестирование (unit testing)* [6]. Оно используется для автоматического тестирования отдельных модулей программы. Для каждой нетривиальной функции создается модульный тест, проверяющий основные сценарии работы с ней. При этом используется структурный подход: учитываются критерии охвата модульных тестов [14]. Для запуска и написания таких тестов применяются специальные

библиотеки, существующие для всех популярных языков программирования, например, *JUnit* [15], *NUnit* [16], *PyUnit* [17] и тому подобные.

Модульные тесты обычно запускаются программистами после каждого значительного изменения кода или рефакторинга. Удобство и широкая распространенность модульных тестов связаны с тем, что тесты пишутся на том же языке, что и сама программа, а средства запуска таких тестов реализованы во всех популярных интегрированных средах разработки. Часто все модульные тесты в программе запускаются автоматически после каждого внесения изменений. Такой подход называется *непрерывной интеграцией* (*continuous integration*) [18].

Программирование по контрактам (design by contract). Этот подход к объектно-ориентированному программированию был предложен Бертраном Мейером [19]. Основной чертой этого подхода является использование *утверждений* (*assertions*) – условий, которые должны выполняться, если код программы верен. Утверждения представляют собой булевы выражения и делятся на два типа:

- утверждения, устанавливающие *контракт* между классом и его клиентами. Утверждения этого типа являются реализацией *спецификации поведения интерфейса* (*behaviorial interface specification, BIS*) [20];
- утверждения, выполняющие промежуточные проверки в коде реализации алгоритмов, фактически являются исполнимыми комментариями.

Программирование по контракту используют утверждения первого типа.

Подходы с использованием утверждений существовали и до появления программирования по контрактам для проверки корректности входных и выходных данных, но в них утверждения были доступны только самому

классу, в котором они были написаны. Клиенты класса получали информацию о требованиях к значениям входных параметров и результата работы методов класса только через неисполняемые комментарии к классу.

В программировании по контрактам используются три вида специальных утверждений:

- *предусловия* определяют ожидания метода объекта относительно значений входных параметров и состояния класса при вызове этого метода;
- *постусловия* задают обязательства метода при завершении его работы;
- также класс может специфицировать набор *инвариантов*, которые должны выполняться на протяжении всего жизненного цикла экземпляров класса.

Эти утверждения становятся частью интерфейса класса и доступны всем его клиентам. При программировании по контрактам разработка архитектуры программы включает в себя написание контрактов классов как неотъемлемой части их интерфейсов. Затем классы реализуются таким образом, чтобы удовлетворять написанным контрактам.

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* [3], в котором программирование по контрактам и было впервые реализовано. Существуют также библиотеки расширений, реализующие полную или частичную поддержку программирования по контрактам в распространенных языках программирования [21, 22]. Однако,

так как эти библиотеки не являются частью соответствующего языка, они не получили широкого распространения.

Реализация программирования по контрактам в языке *Eiffel* предусматривает три способа использования контрактов:

- использовать контракты как неисполнимые комментарии с проверкой корректности выражений. При этом проверка соответствия реализации программы описанным контрактам не осуществляется;
- статически верифицировать программу на соответствие контрактам, однако этот подход весьма сложен и его практическая реализация обычно требует введения дополнительных языковых конструкций [23];
- скомпилировать программу таким образом, что компилятор вставит код, проверяющий соответствие контрактам во время выполнения.

Наиболее распространенной и полезной является третья возможность, которая связана с выполнением проверок во время работы программы. Для эффективного использования таких проверок программу требуется регулярно запускать на определенных тестовых сценариях. Такие сценарии обычно реализуются в виде модульных тестов. Преимуществом использования контрактов в модульном тестировании является раннее оповещение об ошибке: если контракты описаны достаточно подробно, то нарушение произойдет очень скоро после выполнения ошибочной инструкции, в то время как при традиционном модульном тестировании оповещение происходит только при проверке утверждения в коде теста или возникновении исключения.

Выводы по разделу 1.2. Общей чертой рассмотренных выше методов реализации программного обеспечения является стремление хранить и

поддерживать спецификацию программы вместе с ее исходным кодом. Это связано тем, что при частом изменении требований спецификация, хранящаяся отдельно быстро теряет актуальность.

Хранение спецификации совместно с исходным кодом преследует три основные цели: *локальность*, *формализованность* и *исполнимость*. Под локальностью понимается совместное представление кода программы и его спецификации.

Формализованность – представление спецификации в формальном виде. Это свойство повышает возможности автоматической обработки спецификации, особенно если для представления кода и спецификации используется один и тот же подход. Примером, иллюстрирующим преимущества такого подхода, является рефакторинг программы с модульными тестами. Эти тесты являются формальным представлением спецификации в виде сценариев, описанных на языке программирования. Поэтому при автоматическом рефакторинге такой программы происходит автоматическое обновление тестов. Таким образом обеспечивается согласованность спецификации (тестов) и кода программы.

Исполнимость означает возможность автоматической *верификации программы* – проверки соответствия программы формализованной спецификации. Это позволяет значительно ускорить верификацию.

1.3. Автоматное объектно-ориентированное программирование

Одним из подходов, используемых при проектировании программ со сложным поведением, является автоматное программирование (программирование с явным выделением состояний) [24]. В соответствии с этим подходом компоненты программы, обладающие сложным поведением, следует представлять в виде систем автоматов, взаимодействующих друг с другом, а также с неавтоматной частью программы. Первоначально этот подход был разработан для использования в совокупности с процедурным

программированием, а затем был расширен и для объектно-ориентированного программирования [25].

В настоящее время предложены, исследованы и с успехом применяются различные автоматные модели [26]. В настоящей работе рассматриваются автоматы, поведение которых описывается графами переходов со следующими свойствами [27, 28]:

- граф описывает одно или несколько состояний автомата и переходы между ними;
- состояния автомата могут быть объединены в группы, которые могут быть вложены друг в друга. Состояния внутри группы равноправны;
- переходы могут начинаться в состоянии или в группе состояний, а заканчиваться только в состоянии (переходы, начинающиеся в группе состояний, называются *групповыми переходами*). Переходы могут начинаться и заканчиваться в одном и том же состоянии;
- каждое состояние помечено следующими атрибутами:
 - имя состояния;
 - действия при входе в состояние;
- переход может быть помечен следующими атрибутами:
 - условие перехода;
 - события;
 - действия на переходе.

Обработка события происходит следующим образом: перебираются переходы, выходящие из текущего состояния или содержащих его групп, и помеченные обрабатываемым событием. Для каждого перехода вычисляется условие, которое является булевой формулой. Эта формула может содержать события и входные переменные.

Выполняется переход, для которого значение условия истинно. Выполнение перехода состоит из следующих шагов:

- выполняются действия на переходе (вызываются указанные выходные воздействия в порядке их следования);
- текущим становится состояние, в котором заканчивается переход;
- если произошла смена состояния (текущее состояние до начала обработки события отличается от состояния, в котором заканчивается переход), то вызываются действия при входе в состояние.

Заметим, что в вышеизложенном описании отсутствует взаимодействие нескольких автоматов. Это связано с тем, что при подходе «автоматизированные объекты управления как классы» [26], которого автор придерживается в настоящей работе, взаимодействие автоматов осуществляется так же, как и взаимодействие автомата и объекта управления: через входные и выходные воздействия. Однако существуют подходы к построению автоматных объектно-ориентированных программ, в которых используются системы автоматов. Такой подход, например, применяется в инструментальном средстве *UniMod*. Поэтому предложенный во второй главе метод динамической верификации ориентирован на систему автоматов, а автоматы, используемые в подходе «автоматизированные объекты управления как классы», являются частным случаем такой системы.

Одной из основных проблем программирования с явным выделением состояний является организация взаимодействия автоматного и неавтоматного кода. В настоящее время разработан ряд подходов [29] к реализации автоматной части объектно-ориентированных программ. Эти подходы можно условно разделить на два класса: *статические* и *динамические*.

При статическом подходе выделение состояний происходит на этапе проектирования, и структура автоматной части программы жестко фиксируется в коде. При динамическом подходе поведение автоматной части может определяться во время выполнения программы. Это достигается использованием специальных библиотек, которые строят автоматную часть программы динамически на основе описания ее структуры в виде *XML* [30] или последовательности сообщений, посланных неавтоматной частью программы [31].

Смешанным можно считать декларативный подход к объявлению структуры автоматов в объектно-ориентированных программах [32]: при этом подходе используется как статическая типизация, так и позднее связывание на основе атрибутов.

Из теории управления известно понятие *автоматизированного объекта управления* (в дальнейшем – *автоматизированные объекты*), являющегося совокупностью управляющего автомата и объекта управления. В книге [26] показано, что классы, являющиеся автоматизированными объектами, являются удобным способом организации автоматного кода в объектно-ориентированной программе. В соответствии с этим подходом, для реализации сущности со сложным поведением в программе вводится автоматный класс, неотличимый для его клиентов от обычного неавтоматного класса. Этот класс используется для доступа к объекту управления. При этом сам объект управления может быть реализован внутри автоматного класса или выделен в отдельный класс. Особенностью автоматного класса является то, что семантика связи его интерфейса с объектом управления реализована в виде автомата. Это позволяет скрыть сложность поведения этого класса и заниматься вопросами его проектирования и качества отдельно от остальной программы.

Существующие подходы к программированию с явным выделением состояний, применимые в статических языках программирования и

позволяющие описывать автоматную часть программы, имеют ряд недостатков:

- невозможно совместить в одном классе, реализованном на статическом языке, автомат и объект управления;
- для текстовых языков автоматного программирования [33] описание автоматной части программы оказывается чересчур громоздким, а синтаксис этих языков иногда вынуждает программиста оперировать понятиями более низкоуровневыми, чем «состояние»;
- для графических языков автоматного программирования [34] ввод диаграмм состояний с помощью графического редактора трудоемок, и поэтому многие программисты предпочитают работать с текстовым представлением программы, достигая тем самым однородности представления кода всей программы.

В работе [33] для решения этих проблем предлагается использовать текстовый предметно-ориентированный язык (*DSL*) *stateMachine*, реализованный в системе метапрограммирования *JetBrains MPS (Meta Programming System)* [35, 36]. Этот язык позволяет присоединить автоматную часть в виде аспекта к любому классу в программе. Каждый автомат в языке *stateMachine* связан с некоторым классом и описывает его поведение. Для того чтобы задать поведение класса с помощью автомата, необходимо в этом классе определить события, на которые будет реагировать автомат. Особенностью языка *stateMachine* является то, что события в нем – это методы, реализация которых находится в автоматной части класса и зависит от его текущего состояния. Такой подход позволяет автоматизировать произвольный класс программы – обеспечить автоматное поведение класса. При этом автоматная логика класса остается скрытой даже для неавтоматной части того же класса.

В настоящее время широкое распространение получили динамические объектно-ориентированные языки программирования. Одной из их особенностей является возможность отправить любому объекту в программе произвольное сообщение (в то время как в статических языках набор возможных сообщений определяется *абстрактным типом данных* объекта [26]). Другой особенностью динамических языков является возможность замены реализаций методов во время работы программы.

Эти особенности в совокупности со спецификой синтаксиса некоторых динамических языков программирования позволяют описывать автоматную часть программ в терминах состояний и переходов между ними. Таким образом, становится возможным разработать автоматный *внутренний предметно-ориентированный язык* [36]. Одной из первых реализаций такого языка является библиотека *STROBE* [37] для языка программирования *Ruby*. Это направление получило развитие в работе [38], в которой введена поддержка наследования и вложения автоматов.

Таким образом, в настоящее время существует несколько языков и библиотек, позволяющих описывать автоматную часть программы, используя не языковую, а автоматную терминологию. В некоторых языках и библиотеках сохраняется изоморфизм между кодом программы и графами переходов.

Еще одной проблемой, недостаточно исследованной в существующих методах проектирования и реализации автоматных программ, является модификация программ при изменении требований. Предполагается, что при изменении требований автоматная программа должна быть спроектирована «с нуля» на основе новых требований. Этот процесс является неоправданно трудоемким, особенно в случае, когда изменения требований незначительны. Альтернативой является бессистемное внесение требований в разработанную программу, что может привести к возникновению ошибок.

Как было отмечено выше, для решения этой проблемы в объектно-ориентированном программировании используются рефакторинги. В настоящее время вопрос рефакторинга автоматных программ не исследован.

1.4. Методы повышения качества автоматных объектно-ориентированных программ

Рассмотрим методы, используемые для повышения качества автоматных объектно-ориентированных программ. С одной стороны, к автоматным классам, являющимся частью объектно-ориентированной программы, могут применяться все методы, описанные в разд. 1.2, а с другой – в автоматном программировании сложилась своя методика обеспечения и повышения качества.

В основе этой методики лежит ряд особенностей автоматных программ:

- при описании сложного поведения с помощью системы автоматов, оно разделяется между автоматами таким образом, чтобы поведение каждого автомата было обзримым и относительно простым, причем в соответствии с подходом «автоматизированные объекты управления как классы» различные автоматы являются независимыми сущностями;
- поведение каждого автомата формализовано;
- управляющие состояния выделены;
- условия переходов основаны на значениях небольшого числа переменных.

Основным достоинством автоматных программ является возможность повышения уровня автоматизации их верификации по сравнению с программами других классов. Это связано с тем, что автоматная программа по своей структуре изоморфна структуре Крипке, используемой в

Model Checking – одном из наиболее распространенных подходов к верификации программ.

Методы анализа автоматных программ можно разделить на два класса: *статические* и *динамические*.

При использовании *динамического подхода* работа программы анализируется во время исполнения. В этой категории наиболее распространенным является тестирование автоматных программ. Оно во многом аналогично тестированию объектно-ориентированных программ, подробно рассмотренному в разд. 1.2.

Важной особенностью тестирования автоматных программ является возможность автоматической генерации тестов по структуре автомата и формальной спецификации [39]. Такие тесты позволяют достичь значительного покрытия основных сценариев работы программы.

Статический подход основан на анализе исходного кода или его модели и призван гарантировать правильность работы программы. Статическую проверку можно проводить на нескольких уровнях.

Первым из них является проверка синтаксиса программ. Применительно к автоматным программам – это проверка корректности описания автоматов: существование всех использованных идентификаторов переменных, отсутствие переходов «в никуда» и т. д.

Вторым уровнем статической проверки является валидация – проверка корректности программы без учета ее семантики. Для автоматных программ валидация состоит в проверке для графов переходов таких свойств как *достижимость*, *непротиворечивость* и *полнота*.

Достижимость. Автомат должен иметь возможность перейти в любое состояние из начального состояния, следуя по переходам. В противном случае состояние является избыточным, его удаление не повлияет на работу автомата. Существование подобных избыточных артефактов обычно является признаком ошибки проектирования.

Непротиворечивость. Переходы из любого состояния программы должны быть непротиворечивы – не должно существовать такого входного воздействия, при котором выполняются охранные условия более одного перехода из одного и того же состояния (ортогональность переходов). При существовании таких переходов фактически возникает недетерминизм: автомат может выбрать любой переход и его поведение в таком состоянии окажется непредсказуемым, что свидетельствует об ошибке.

Полнота. В каждом конечном состоянии дизъюнкция пометок переходов, исходящих из состояния, должна быть истинна. Если это свойство не выполняется, переходы при некоторых наборах входных воздействий не определены. Обычно считается, что на входных воздействиях, переходы для которых не определены, автомат по умолчанию сохраняет состояние.

Существует ряд инструментальных средств, выполняющих валидацию автоматных программ, например, *UniMod* [34, 40].

Наибольший интерес представляет *третий уровень* статической проверки – верификация (формальная проверка соответствия автомата спецификации). В рамках верификации выделяются два основных подхода: *доказательная верификация* [41, 42] и *верификация на модели* [43].

Первый подход к верификации основан на доказательстве теоремы о соответствии программы спецификации. Этот подход трудно автоматизируется, и поэтому практически не применим для верификации больших программ.

Верификация на модели (*Model Checking*) основана на анализе соответствия модели программы с конечным числом состояний (*структуры Крипке*) формальной спецификации, заданной в виде набора формул темпоральной логики [44]. Эти данные подаются на вход верификатора, который в автоматическом режиме выполняет верификацию – в каждом состоянии модели проверяется выполнимость темпоральной формулы. В результате либо модель признается соответствующей спецификации, либо

строится контрпример – путь в модели, не удовлетворяющий спецификации. На основе анализа контрпримера принимается решение о причине несоответствия спецификации: неверное построение модели, неверная формализация спецификации или ошибка в программе. В последнем случае важно перенести контрпример из модели в исходную программу. Одной из основных причин, по которым верификация на основе метода *Model Checking* достаточно редко используется для повышения качества программ общего вида, является трудность автоматизации перехода от программы к модели и обратно.

Для автоматных программ такая автоматизация практически осуществима [45, 46], так как в этом случае программа строится по модели поведения, а при традиционном подходе – наоборот. Это утверждение также основано на том, что в автоматных программах, как и в машине Тьюринга, состояния делятся на два класса: управляющие и вычислительные. При этом управляющих состояний, в отличие от вычислительных, сравнительно немного, а верификацию в автоматных программах предлагается проводить именно для управляющих состояний.

В настоящее время существует ряд инструментальных средств, эффективно реализующих верификацию автоматных программ [47–49]. При этом, однако, отсутствуют инструментальные средства, которые позволяют *интегрировать процесс верификации с процессом разработки программ*. В частности, в известных подходах спецификация и автоматная программа существуют отдельно и обрабатываются различными инструментальными средствами. Это означает, что при модификации автоматной программы (например, при переименовании состояния) спецификацию требуется приводить в соответствие программе вручную, и наоборот.

Важно отметить, что существующие статические и динамические методы проверки качества автоматных программ требуют спецификации требований в различной форме: сценарии тестирования для динамического

метода тестирования и темпоральные формулы для статической верификации. Это требует использования различных подходов к формализации требований в зависимости от используемого метода проверки качества.

Темпоральные спецификации как метод формализации требований также имеют недостатки, один из которых – громоздкость и сложная структура формул. Например, требование «Во время движения двери лифта закрыты» для автомата управления лифтом, подробно рассмотренного во второй главе, выражается следующей темпоральной формулой: $[(z_1 \vee z_2) \Rightarrow \overline{z_4} \mathbb{U}(z_3 \vee \overline{z_4})] \wedge [z_4 \Rightarrow \overline{(z_1 \vee z_2)} \mathbb{U}(z_5 \vee \overline{(z_1 \vee z_2)})]$.

Выводы по главе 1

1. Существующие методы проверки качества объектно-ориентированных программ основаны на тестировании и не могут гарантировать отсутствия ошибок в программе. Это снижает качество объектно-ориентированных программ со сложным поведением. Автоматное объектно-ориентированное программирование является эффективным средством реализации программ со сложным поведением и позволяет производить формальную верификацию таких программ.
2. Методы формализации требований к автоматным программам разнородны и часто неэффективны. Например, при формализации простых требований в некоторых случаях получаются сложные формулы. Поэтому в работе будет предложен метод, упрощающий формальное представление требований.
3. Код и спецификация автоматных программ не интегрированы, работа с ними осуществляется с помощью различных инструментальных средств. В четвертой главе предлагается метод интеграции реализаций и формализованных спецификаций.

4. При изменении требований к автоматной программе обычно требуется заново спроектировать программу либо вносить изменения в реализацию, что обычно выполняется бессистемно. Успешно применяемый в объектно-ориентированном программировании метод рефакторинга программ недостаточно исследован в применении к автоматным программам. Этот недостаток будет, по крайней мере, частично, устранен в настоящей работе.

ГЛАВА 2. ФОРМАЛИЗАЦИЯ ТРЕБОВАНИЙ К АВТОМАТНЫМ ОБЪЕКТНО-ОРИЕНТИРОВАННЫМ ПРОГРАММАМ

2.1. Формализация требований к автоматным программам

Как было показано в первой главе, одной из основных проблем, возникающих при попытке использования автоматов в объектно-ориентированных программах, является невозможность интегрировать реализацию автомата и его спецификацию. Для автоматизации работы со спецификацией, ее требуется представить формально. Ниже предлагается метод, который позволяет представить требования в виде набора формальных спецификаций различного вида. Как было показано в первой главе, формализация спецификаций позволяет автоматизировать проверку соответствия программы спецификации. Например, для модульных тестов – распространенного метода формализации спецификаций объектно-ориентированного ПО, проверка реализуется путем исполнения тестов. По аналогии проверку любой формальной спецификации будем называть ее *исполнением*, а саму такую спецификацию – *исполняемой*.

В настоящее время для формализации требований к автоматным программам чаще всего используют темпоральные спецификации и модульные тесты. Основное достоинство методов, использующих темпоральные спецификации – то, что эти методы основаны на верификации. Достоинством методов, основанных на тестировании – возможность проверки правильности работы управляющих автоматов совместно с управляемыми объектами.

Рассмотрим использование различных видов формальных спецификаций на примере холодильника. Зададим список требований, предъявляемых к разрабатываемой программе.

1. При выходе показаний датчика напряжения за допустимый диапазон управляющее реле выключит питание.
2. В момент отключения охлаждающего элемента показания термостата не должны превышать допустимые.
3. Непосредственно после открытия двери загорается лампочка.

Формализуем эти требования, представив их в виде исполняемых спецификаций. Так как в большинстве случаев предпочтительно применять статические методы верификации, а наиболее распространенным из таких методов является верификация на модели (*Model Checking*), представим указанные выше требования в виде формул темпоральной логики.

Воспользуемся языком *линейной темпоральной логики* (*Linear Temporal Logic, LTL*) [50]. Формулы этого языка построены на множестве *атомарных высказываний* (*Prop*) с применением булевых операторов, унарного темпорального оператора \mathbb{N} («на следующем шаге») и бинарного темпорального оператора \mathbb{U} («до тех пор, как»).

Моделью для *LTL*-формулы является *вычисление* – функция $\pi: \omega \rightarrow 2^{Prop}$, которая задает значения истинности высказываний из множества *Prop* в каждый момент времени, определяемый натуральным числом. Вычисление π в момент времени $i \in \omega$ удовлетворяет *LTL*-формуле φ (обозначается $\pi, i \triangleright \varphi$) при выполнении следующих условий:

- $\pi, i \triangleright p$ для $p \in Prop \Leftrightarrow p \in \pi(i)$;
- $\pi, i \triangleright \xi \wedge \psi \Leftrightarrow (\pi, i \triangleright \xi \wedge \pi, i \triangleright \psi)$;
- $\pi, i \triangleright \xi \vee \psi \Leftrightarrow (\pi, i \triangleright \xi \vee \pi, i \triangleright \psi)$;
- $\pi, i \triangleright \overline{\varphi} \Leftrightarrow \overline{\pi, i \triangleright \varphi}$;
- $\pi, i \triangleright \mathbb{N}\varphi \Leftrightarrow \pi, i + 1 \triangleright \varphi$;
- $\pi, i \triangleright \xi \mathbb{U} \psi \Leftrightarrow \exists j > i: \pi, j \triangleright \psi, \forall k: i < k < j: \pi, k \triangleright \xi$.

Также используются темпоральные операторы \mathbb{G} («всегда») и \mathbb{F} («в будущем»), которые эквивалентны следующим формулами:

- $\mathbb{F}\varphi \Leftrightarrow true\mathbb{U}\varphi$;
- $\mathbb{G}\varphi \Leftrightarrow \overline{\mathbb{F}\overline{\varphi}}$.

Говорят, что π удовлетворяет формуле φ (обозначается $\pi \triangleright \varphi$), если и только если $\pi, 1 \triangleright \varphi$.

Для верификации автомата с использованием метода динамической верификации требуется определить:

- набор высказываний, которые разрешено использовать в *LTTL*-формулах спецификации;
- способ построения протокола выполнения автомата;
- правила вычисления значений атомарных высказываний в записях протокола.

Опишем набор высказываний, на которых можно основывать спецификации. Обработка каждого события системой автоматов состоит из следующих шагов [51]:

- получение события;
- вычисление условий на переходах;
- выполнение действий на переходе. Одним из таких действий является вызов другого автомата. При выполнении вызова управление передается вызываемому автомату с соответствующим событием. Этот автомат совершает переход по указанному событию и затем возвращает управление вызвавшему автомату;
- переход в новое состояние.

В соответствии с этой схемой необходимо предоставить возможность установить факт совершения того или иного этапа обработки события. Таким образом, достаточным представляется следующий набор базовых высказываний:

- $e_{i,j}$: автомат A_i обрабатывает событие e_j ;

- x_i : при обработке текущего события значение входной переменной x_i истинно;
- z_i : выполняется выходное воздействие z_i ;
- $s_{i,j}$: автомат A_i находится в состоянии $s_{i,j}$.

2.2. Интерфейсы автоматных объектов

При формализации спецификаций в объектно-ориентированных программах общего вида спецификации чаще всего записываются в терминах интерфейсов объектов. Например, модульные тесты имеют доступ только к интерфейсам объектов.

Контракты – другой вид спецификаций объектно-ориентированных программ. Они являются частью интерфейса объекта, его поведенческой спецификацией.

При формализации контрактов автоматов их также следует выражать в терминах интерфейса автомата. При использовании подхода «автоматизированные объекты как классы» [26] к объектно-ориентированному программированию с явным выделением состояний, у автоматного класса можно выделить два интерфейса: *объектный* и *автоматный*. Объектным является интерфейс класса объекта управления, через который происходит взаимодействие автоматного модуля с остальной частью объектно-ориентированной программы. Он представляет собой набор внешних операций, доступных другим классам программы. Этим классам неизвестна автоматная сущность реализации интерфейса, они не имеют информации о текущем внутреннем состоянии объекта.

Автоматным интерфейсом является внешняя информация об автомате: его текущее состояние, обрабатываемое событие, набор входных и выходных воздействий. При проектировании систем совместно работающих автоматов этот интерфейс применяется для их взаимодействия. Автоматы могут использовать информацию о текущем состоянии других автоматов. Входные

и выходные воздействия различных автоматов программы связаны друг с другом.

В то же время в корректно спроектированных объектно-ориентированных программах автоматы не должны иметь информацию о структуре переходов других автоматов, эта часть реализации скрыта. В рассматриваемом подходе различные автоматы программы взаимодействуют исключительно через объектные интерфейсы. Связь объектного и автоматного интерфейсов может быть произвольной и зависит от потребностей конкретного приложения.

Темпоральные спецификации предназначены для тестирования автомата. Следовательно, формулы должны иметь доступ исключительно к автоматному интерфейсу и предложенный набор высказываний покрывает автоматный интерфейс полностью.

На основании описанных требований для примера, рассмотренного выше (холодильник), можно выделить следующий автоматный интерфейс:

- x_1 : дверца открыта;
- x_2 : напряжение превысило пределы диапазона допустимых значений;
- x_3 : показания термостатов превысили пределы диапазона допустимой температуры;
- z_1 : включить лампу;
- z_2 : выключить лампу;
- z_3 : включить охлаждение;
- z_4 : выключить охлаждение;
- e_1 : открытие дверцы холодильника.

Можно выделить два подхода к формализации спецификации автомата, отличающиеся объектом верификации. При использовании первого подхода (назовем его *спецификации белого ящика*) в темпоральных спецификациях

используется полный автоматный интерфейс, включая информацию о текущем состоянии. Такая верификация может быть использована только при наличии конкретной реализации автомата. Второй подход (назовем его, соответственно, *спецификацией черного ящика*) позволяет ссылаться только на события, входные и выходные воздействия. Такие спецификации позволяют верифицировать взаимодействие автомата и объекта управления независимо от реализации автомата. Для описания указанных требований к программе управления холодильником не требуется информация о наборе состояний автомата. Это позволяет использовать верификацию методом черного ящика.

С применением введенных атомарных утверждений первые два пункта требований к программе управления холодильником могут быть записаны следующим образом:

$$x_2 \Rightarrow \mathbb{F}x_4; \quad (1)$$

$$z_4 \Rightarrow \overline{x_3}. \quad (2)$$

При формализации третьего пункта требований возникает проблема с интерпретацией наречия «непосредственно». Так как рассматриваются нетаймированные автоматы, длительность такта определяется появлением события. Таким образом, фразу «непосредственно после открытия дверцы» можно интерпретировать как «непосредственно после обработки события об открытии дверцы холодильника». При такой интерпретации третий пункт требований записывается следующим образом:

$$e_1 \mathbb{U} z_1 \quad (3)$$

Таким образом, приведенные выше вербальные требования были формализованы с использованием только информации о входных переменных и выходных воздействиях. Это означает, что был использован метод спецификации «черного ящика». Поэтому полученной формальной спецификации может удовлетворять управляющий автомат с произвольным

набором состояний, важно только чтобы его взаимодействие с объектом управления отвечало соотношениям 1–3.

Однако часто при формализации требований необходимо обратиться к структуре конкретного автомата, реализующего заданное поведение. В этом случае требуется воспользоваться методом спецификации «белого ящика».

2.3. Автоматное программирование по контрактам

Целью настоящего раздела является расширение объектно-ориентированного контрактного программирования на класс автоматных объектно-ориентированных программ. Для демонстрации недостатков существующего подхода к формализации требований рассмотрим еще один пример вербальных требований – к системе управления лифтом, которую обозначим «лифт 1»:

1. Во время движения двери лифта закрыты.
2. Пока двери лифта открыты, свет в кабине включен.

Для формализации этих требований необходимо определить автоматный интерфейс программы управления:

- z_1 : включить мотор привода кабины в направлении «вверх»;
- z_2 : включить мотор привода кабины в направлении «вниз»;
- z_3 : остановить мотор привода кабины;
- z_4 : включить механизм открывания дверей;
- z_5 : включить механизм закрывания дверей;
- z_6 : включить свет в кабине;
- z_7 : выключить свет в кабине;
- x_{11} : лифт на первом этаже;
- x_{12} : лифт на втором этаже;
- x_{13} : лифт на третьем этаже;
- x_4 : свет в кабине включен;

- e_1 : лифт прибыл на этаж;
- e_2 : двери открыты;
- e_3 : двери закрыты;
- e_{41} : вызов лифта на первый этаж;
- e_{42} : вызов лифта на второй этаж;
- e_{43} : вызов лифта на третий этаж.

Выделим следующие состояния автомата:

1. «Движение».
2. «Открывание дверей».
3. «Закрывание дверей».
4. «Ожидание с открытыми дверьми».
5. «Ожидание с закрытыми дверьми».

Формализуем первый пункт требований («Во время движения двери лифта закрыты»). Так как датчик движения в лифте отсутствует, то о том, что лифт движется, можно судить лишь по порядку подачи управляющих команд приводу кабины лифта.

Для формализации первого пункта требований, с учетом изложенного в предыдущем абзаце, требуется записать темпоральной формулу, эквивалентную утверждению: «отрезок между подачей команды запуска привода кабины и команды останова этого привода, и отрезок между открытием и закрытием дверей лифта не пересекаются». Утверждение «между двумя событиями u и v не происходит событие p » выражается формулой $u \Rightarrow \bar{p}U(v \vee \bar{p})$. Утверждение о том, что отрезки не пересекаются, эквивалентно утверждению, что между событиями, выражающими начало и конец каждого отрезка, не происходит события, выражающего начало другого отрезка.

Команда о запуске привода кабины выражается формулой $z_1 \vee z_2$, об остановке привода – формулой z_3 . Открытие дверей выражается формулой z_4 , закрытие – формулой z_5 . Таким образом, эквивалентное утверждение выражается следующей конъюнкцией двух условий: $[(z_1 \vee z_2) \Rightarrow \overline{z_4} \mathbb{U} (z_3 \vee \overline{z_4})] \wedge [z_4 \Rightarrow \overline{(z_1 \vee z_2)} \mathbb{U} (z_5 \vee \overline{(z_1 \vee z_2)})]$.

Эта формула достаточно сложна для понимания, поэтому при ее использовании дальнейшая поддержка автоматной программы и ее спецификации будет значительно затруднена и может привести к возникновению ошибок.

Альтернативным подходом может стать использование информации о структуре автомата – переход к спецификации «белого ящика». Во время движения лифта автомат должен находиться в состоянии «Движение», эта информация может быть использована для упрощения указанной выше формулы. Заменяем сложные соотношения между утверждениями z_1 , z_2 и z_3 простым утверждением s_1 , эквивалентным утверждению «автомат находится в состоянии «Движение»». После такой замены формула принимает вид: $\overline{s_1 \wedge ((z_4 \mathbb{U} z_5) \vee \overline{true \mathbb{U} z_4})}$. Она значительно проще предыдущей.

Дополнительно формулу можно упростить, воспользовавшись информацией о том, что двери могут быть открыты только при выходе из состояний «Открытие дверей» и «Ожидание с открытыми дверями». В этом случае исходное требование можно сформулировать в виде комбинации двух более простых требований: «Во время движения двери не открываются» и «Движение начинается только после состояния, при выходе из которого двери закрыты». На языке *LTL* эти требования записываются следующим образом:

- $\overline{s_1 \wedge z_4}$;
- $\overline{(s_2 \vee s_4) \Rightarrow (s_2 \vee s_4 \wedge \overline{s_3 \vee s_5}) \mathbb{U} s_1}$.

Второе соотношение делает спецификацию достаточно сложной. Однако, как следует из формул, новая спецификация выражает требования в терминах конкретных состояний, в то время как исходная – в терминах переменных и событий. Другими словами, каждая формула новой спецификации описывает состояние s_1 верифицируемого автомата. Первая из формул утверждает, что в состоянии s_1 не должно происходить выходного воздействия z_4 , а вторая – что в состоянии s_1 можно перейти только из состояний s_2 и s_4 .

Большая часть ошибок при автоматном программировании возникает из-за неточного специфицирования состояний. Разделение требований на части, относящиеся к конкретным состояниям, помогает более четко специфицировать эти состояния. Задача спецификации конкретных состояний автомата аналогична задаче, решаемой *контрактами* в объектно-ориентированном программировании, рассмотренными в первой главе. Напомним, что контракты в объектно-ориентированных программах специфицируют компоненты интерфейса. Аналогичным образом спецификации, описывающие поведение автомата в определенных состояниях, являются контрактами этих состояний.

Традиционно контракты выражаются в форме пред- и постусловий, а также инвариантов. Применительно к состояниям автоматов эта классификация оказывается вполне естественной. Предусловиями будем считать выражения, которые должны быть истинными при входе в состояние. Постусловиями назовем выражения, которые должны быть истинными при выходе из состояния. Инварианты должны выполняться в течение всего времени, пока автомат находится в соответствующем состоянии.

Требования для лифта, сформулированные выше, можно представить в виде контрактов. Первое условие является инвариантом $\overline{z_4}$, второе – предусловием $s_2 \vee s_4$.

Кроме контрактов состояний предлагается ввести также контракты групп состояний, входных и выходных воздействий. Примером контракта для группы состояний является инвариант «лампа включена» для группы состояний «двери лифта открыты» (объединяющей состояния «ожидание с открытыми дверьми», «двери открываются» и «двери закрываются»).

Так как для автомата выполнение выходного воздействия – атомарное событие, то имеет смысл только предусловие на выходное воздействие – условие, которое должно выполняться при вызове этого воздействия. Например, для выходных воздействий z_1 и z_2 имеет смысл предусловие s_1 , которое устанавливает, что при включении привода лифта управляющий автомат должен находиться в состоянии «Движение».

В связи с тем, что при использовании подхода «автоматизированные объекты управления как классы» входные воздействия становятся частью объектного интерфейса, контракты на входные воздействия являются частью интерфейса объекта, и рассматривать их отдельно как часть автоматных контрактов не требуется.

Второй пункт требований к автомату управления лифтом («Пока двери лифта открыты, свет в кабине включен») может быть сформулирован как в виде инвариантов x_4 для состояний s_2 и s_4 .

Исходя из изложенного, наравне с автоматным и объектным интерфейсами можно говорить об автоматных и объектных контрактах автоматизированного объекта управления. Объектные контракты применяются к объектному интерфейсу и используются клиентами автоматизированного объекта, а также накладывают ограничения на реализацию автомата. Автоматные контракты применяются к автоматному интерфейсу и накладывают ограничения на реализацию, но не используются клиентами объекта. Последнее свойство является отличительным для автоматных контрактов, делая их похожими на инварианты циклов, которые используются в объектно-ориентированном программировании по

контрактам. Такие инварианты так же, как и автоматные контракты, используются только внутри реализации метода и недоступны его клиентам. Это, однако, не уменьшает их полезность: контракты компонентов автомата (состояний, входных и выходных воздействий) облегчают проектирование взаимодействия этих компонентов.

Таким образом, преимуществами задания требований в виде контрактов являются более компактная форма и привязка к конкретному компоненту автомата. Это позволяет использовать контракты для уточнения семантики этого компонента. В четвертой главе будет показано, как предложенные автоматные контракты могут быть «встроены» в текст программы.

2.4. Способы проверки автоматных контрактов

Проверка выполнения контрактов может, как и в объектно-ориентированном программировании, выполняться различными способами – статическим и динамическим.

При *статической проверке* контракты используются в качестве спецификации для верификации на модели. Для этого контракты необходимо преобразовать в темпоральные формулы согласно правилам, приведенным в таблице (символ p – условие контракта).

Таблица. Правила преобразования контрактов в темпоральные спецификации

	Предусловие	Постусловие	Инвариант
Состояние, группа состояний	$\bar{s}_i \Rightarrow N(\bar{s}_i \vee p)$	$s_i \Rightarrow N(s_i \vee p)$	$s_i \Rightarrow p$
Входное воздействие	$\bar{e}_i \Rightarrow N(\bar{e}_i \vee p)$	$e_i \Rightarrow N(e_i \vee p)$	
Выходное воздействие	$z_i \Rightarrow p$		

При *динамической проверке* контрактов код автомата дополняется проверками условий контрактов. Этот код выполняется динамически при работе программы. Подробнее динамические методы проверки спецификаций автоматных программ рассматриваются в следующем разделе.

2.5. Динамический метод проверки спецификаций автоматных программ

Рассмотренные методы верификации автоматных программ (верификация темпоральных спецификаций и контрактов на модели) позволяют проанализировать ожидаемое поведение автомата на основе его математической модели. Однако в реальной программе автомат работает совместно с объектом управления и другими элементами программы, которые не являются частью модели автоматизированного объекта управления.

Так как объектно-ориентированная программа в целом является программой общего вида, то попытки ее верификации сопряжены с проблемами, рассмотренными в первой главе.

Альтернативным способом проверки работы программы является применение динамической верификации. Как было описано выше, такая верификация основана на анализе поведения программы во время выполнения. Такой анализ сводится к верификации протоколов запуска программы, которые строятся в автоматных терминах. Эти протоколы получаются в результате запуска программы в различных сценариях, которые бывают трех видов.

Первый из них основан на ее запуске в *рабочем режиме*. При его использовании ошибка обнаруживается уже после запуска программы. Однако этот подход позволяет заблаговременно узнать о проблеме в программе и завершить ее работу в безопасном режиме.

Второй подход основан на использовании набора тестов, описывающих основные сценарии работы программы. Этот набор обычно создается вручную. Этот подход фактически реализует принципы модульного тестирования объектно-ориентированного программного обеспечения, описанные в первой главе. Более того, при использовании подхода «автоматизированные объекты управления как классы», описанного в той же главе, такие тесты для автоматных частей программы могут быть реализованы с помощью тех же инструментов, что и модульные тесты для объектно-ориентированной части программы. Это обеспечивается тем, что входные данные для тестов могут задаваться через объектный интерфейс автоматизированного объекта управления.

Третий подход основан на генерации сценариев, обеспечивающих максимальное покрытие путей в автомате. Алгоритмы генерации сценариев, как правило, основаны на использовании комбинаторных методов, эвристик или генетических алгоритмов [52].

Как было отмечено выше, наиболее часто используемым методом верификации автоматных программ является верификация на модели, в которой в качестве спецификации применяются формулы темпоральной логики. Поэтому для динамической верификации в качестве языка спецификации предлагается также использовать такие формулы.

Важно отметить, что исходным представлением для этих формул могут также служить контракты, описанные в предыдущем разделе.

Методы динамической верификации рассматривают работу программы на конечном отрезке времени. Поэтому для применения *LTL*-логики в динамической верификации требуется расширить определение *LTL* на конечные вычисления, в которых момент времени i является значением из отрезка $[1, n]$. Это сделано в работе [53].

Для данного множества X определим $B^+(X)$ как множество положительных булевых формул над X (булевых формул, построенных из элементов X , которые соединены операторами \vee и \wedge) и формул *true* и *false*.

Альтернирующим автоматом Бюхи называется набор $A = (\Sigma, S, s^0, \rho, F)$ [54], где Σ – непустой конечный алфавит, S – непустое конечное множество состояний, $s^0 \in S$ – начальное состояние, F – множество принимающих состояний, а $\rho: S \times \Sigma \rightarrow B^+(S)$ – функция перехода.

Запуском альтернирующего автомата Бюхи на бесконечном слове $\omega = \{a_0, a_1, \dots\}$ называется S -помеченное дерево, функцию пометок которого обозначим r , такое, что его корень помечен значением s^0 и справедливо: если вершина дерева x глубины i помечена значением s и $\rho(s, a_i) = \theta$, то эта вершина имеет k детей x_1, x_2, \dots, x_k где $k \leq |S|$, и множество их пометок $\{r(x_1), r(x_2), \dots, r(x_k)\}$ обращает формулу θ в истину. *Запуск называется принимающим*, если на каждой последовательности переходов (ветви) принимающие состояния или переход $\rho(s, a_i) = \text{true}$ встречаются бесконечно часто.

Таким образом, каждому альтернирующему автомату A соответствует множество бесконечных слов, для которых существует принимающий запуск. Это множество называется *языком автомата A* и обозначается $L(A)$.

Можно построить аналог альтернирующего автомата Бюхи для конечных слов. Такой автомат будет называться *альтернирующим автоматом*.

В работе [55] показано, что для любой *LTL*-формулы φ можно построить альтернирующий автомат Бюхи A такой, что $L(A) = L(\varphi)$, причем число состояний автомата A линейно зависит от размера формулы φ .

Таким образом, для верификации соответствия протокола работы программы и *LTL*-спецификации, достаточно по этой спецификации построить альтернирующий автомат и проверить, является ли данный

протокол элементом языка альтернирующего автомата. В работе [53] предложены три алгоритма построения принимающего запуска для данных альтернирующего автомата и протокола.

Первый алгоритм (*обход в глубину*) пытается построить принимающий запуск, обходя альтернирующий автомат рекурсивно в глубину из начального состояния. Этот алгоритм наиболее прост в реализации, но часто требует анализа «хвоста» протокола при обходе каждой ветви автомата. Например, для спецификации вида $\mathbb{GF}\phi$, «хвост» протокола должен быть повторно проанализирован на каждом шаге. Поэтому для длинных протоколов алгоритм обхода в глубину работает неприемлемо медленно.

Второй алгоритм (*обход в ширину*) анализирует протокол путем «просмотра автомата в ширину». При этом на каждом шаге поддерживается список всех возможных комбинаций записей протокола, которые могут являться элементами принимающего запуска в данный момент. Алгоритм обхода в ширину анализирует протокол лишь однажды, но вычислительное состояние алгоритма имеет мощность, экспоненциально зависящую от размера спецификации. Для небольших формул множества состояний, допустимых в данный момент, невелики, но с ростом сложности формул мощность этих множеств может резко увеличиваться.

Экспоненциальный рост мощности вычислительного состояния алгоритма вызван недетерминизмом альтернирующего автомата. Эту проблему можно решить, анализируя протокол «от хвоста к голове» [56].

Третий из предложенных в работе [53] алгоритмов (алгоритм обратного обхода в глубину) похож на обход в глубину, но вместо рекурсивных вызовов использует состояние, уже вычисленное для «хвоста» протокола.

Опишем метод динамической верификации системы автоматов. Он состоит из следующих простых шагов:

- запись отрицания верифицируемой спецификации в виде *LTL*-формул. Это преобразование производится вручную в соответствии с рекомендациями, изложенными в работе [57];
- автоматическое построение по полученным формулам альтернирующего автомата;
- запуск системы автоматов и автоматическое построение протокола ее работы;
- автоматический обход альтернирующего автомата в соответствии с полученным протоколом с помощью одного из алгоритмов, описанных выше;
- автоматическое построение контрпримера в модели при обнаружении нарушения спецификации.

Заметим, что все шаги метода, кроме первого, выполняются автоматически, а первый шаг в том или ином виде присутствует во всех подходах к верификации автоматов по методу верификации на модели. Таким образом, предлагаемый метод не увеличивает объем ручной работы по сравнению с другими существующими автоматическими методами.

Для динамической верификации системы автоматов с использованием предлагаемого метода требуется определить:

- набор высказываний, которые разрешено использовать в *LTL*-формулах спецификации;
- способ построения протокола выполнения автомата;
- правила вычисления значений высказываний в записях протокола.

Набор высказываний, необходимых для формализации требований к автоматным программам в виде формул темпоральной логики, был рассмотрен в разд. 2.1.

Для пояснения способа построения протокола и правил вычислений значений высказываний будем рассматривать верификацию системы

автоматов управления лифтом этим методом. Граф переходов этой системы (назовем ее «лифт 2») представлены на рис. 1.

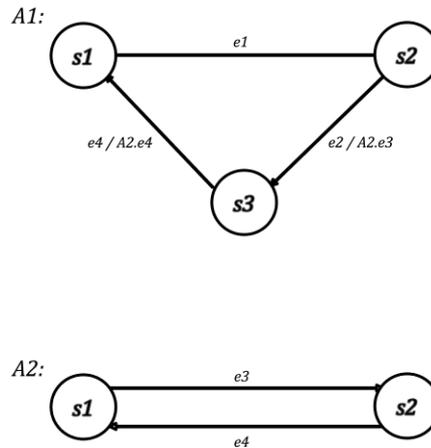


Рис. 1. Графы переходов автоматов управления лифтом

Автомат $A1$ моделирует поведение лифта ($s1$ – «Ожидание, двери закрыты», $s2$ – «Движение», $s3$ – «Ожидание, двери открыты»). Автомат $A2$ моделирует лампу в кабине лифта ($s1$ – «Выключена», $s2$ – «Включена»). Когда автомат $A1$ получает событие $e1$ («Вызов»), он переходит в состояние $s2$, в котором он ожидает событие $e2$ («Прибытие»). При его получении автомат $A1$ переходит в состояние $s3$ и вызывает автомат $A2$ с событием $e3$, включая тем самым лампу. При получении события $e4$ («Двери закрыты») автомат $A1$ пересылает это событие автомату $A2$, который в ответ выключает лампу.

В качестве спецификации, для которой производится верификация, выберем условие «через некоторое время после вызова у лифта открываются двери». В виде LTL -формулы эта спецификация имеет вид: $e_{1,1} \Rightarrow \mathbb{F}s_{1,3}$.

Определим структуру протокола, позволяющую однозначно вычислять значения предложенных высказываний в каждой записи. Эта структура базируется на шагах, выполняемых системой автоматов во время работы [51]. Протокол является конечным словом над конечным алфавитом. В

предложенном методе алфавитом протокола системы автоматов из n автоматов (по $S_i|_{i=1}^n$ состояний в каждом) с m событиями, k входными переменными и l выходными воздействиями является множество со следующими элементами: $e_{1,1}, \dots, e_{n,m}$, $s_{1,1}, \dots, s_{n,s_n}$, x_1, \dots, x_k , $\overline{x_1}, \dots, \overline{x_k}$, $z_1 \dots z_k$. Протоколирование может быть начато лишь при условии, что ни один автомат не производит обработку события. В момент начала протоколирования делаются записи $s_{i,j}$ о текущих состояниях автоматов (эту последовательность записей назовем *заголовком протокола*). При получении автоматом A_i входного события e_j в протокол делается запись $e_{i,j}$. Затем происходит вычисление входных переменных и для каждой переменной, значение которой – *true*, в протокол делается запись x_i , а для переменных со значением *false* – $\overline{x_i}$. При выполнении выходного воздействия в протокол делается запись z_i . Наконец, после обработки автоматом A_i события, в протокол делается запись $s_{i,j}$, указывающая новое состояние автомата, даже в том случае, когда произошел переход по петле.

Введем несколько определений. Назовем *секцией обработки события* автоматом A_i последовательность записей $e_{i,j} \dots s_{i,k}$. *Заголовком секции* назовем начальную подпоследовательность записей длины $1 + k$: $e_{i,j} \dots x_k$. Секции могут быть вложены. При этом вложенные секции обязательно соответствуют разным автоматам.

Определим значения описанных высказываний для каждой записи в протоколе. Высказывание $e_{i,j}$ имеет значение *true* во всех записях секции обработки события, начинающейся с записи $e_{i,j}$. Высказывание x_k имеет значение *true* для всех записей секции, если в заголовке секции встречается запись x_i и *false*, если встречается запись $\overline{x_i}$. Отметим, что в каждом заголовке для каждой входной переменной обязана присутствовать ровно одна из этих записей. При вызове вложенного автомата значения x_k определяются на основании заголовка секции обработки события этим

автоматом. Высказывание z_i имеет значение *true* только в записи z_i . Высказывание $s_{i,k}$ принимает значение *true* в записи $s_{i,k}$ и сохраняет это значение во всех записях протокола до следующей записи вида $s_{i,k}$, не включая саму эту запись. Во всех остальных случаях высказывания имеют значение *false*.

Например, при одиночном вызове лифта и его прибытии на этаж, рассматриваемая система управления лифтом произведет действия, описываемые следующим *протоколом* (в квадратные скобки взяты секции обработки событий): $s_{1,1}s_{2,1}[e_{1,1}s_{1,2}][e_{1,2}[e_{2,3}s_{2,3}]s_{1,3}]$.

Проверифицируем этот протокол на соответствие вышеприведенной спецификации $e_{1,1} \Rightarrow Fs_{1,3}$. Так как спецификация должна выполняться в каждой точке протокола, то альтернирующий автомат необходимо строить по формуле $\mathbb{G}(e_{1,1} \Rightarrow Fs_{1,3})$. Такой автомат имеет вид, показанный на рис. 2.

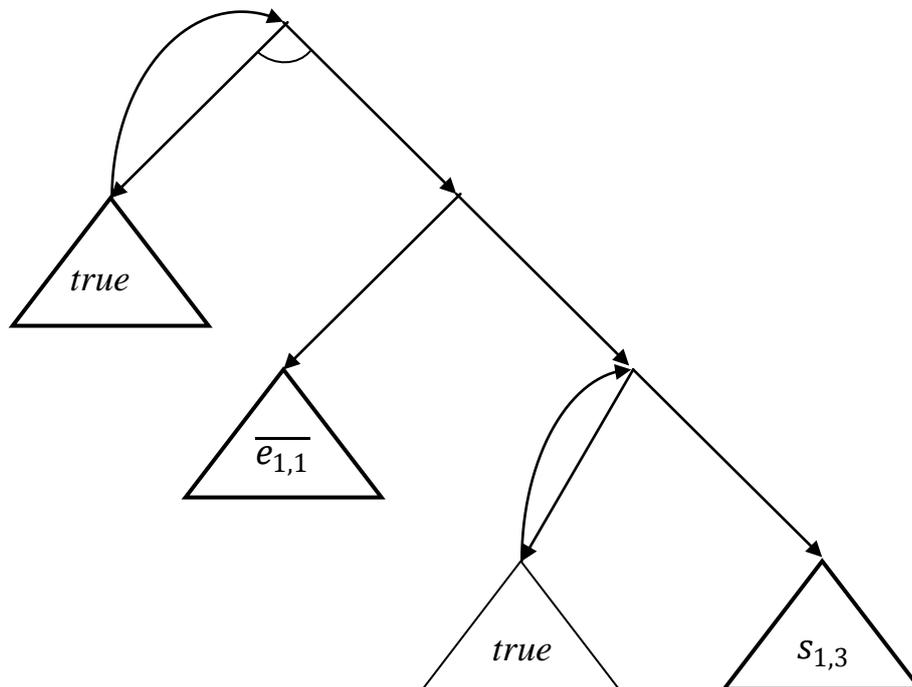


Рис. 2. Альтернирующий автомат, соответствующий требованию «После вызова двери открываются»

Жирно обведены принимающие состояния, дугой объединены «и»-переходы.

По заданному протоколу можно построить принимающий запуск в полученном альтернирующем автомате.

Отметим, что в результате разделения записей заголовка протокола $(@_{i,j} \dots X_k)$, некоторые шаги работы автомата, в которых производится обход альтернирующего автомата, могут быть представлены в нескольких записях протокола. Так как всему заголовку каждой секции соответствует один шаг, то для определения значений высказываний в этот момент времени требуется наличие информации обо всех записях заголовка. Эту особенность необходимо учитывать при реализации предложенного метода на практике, но, так как заголовок всегда имеет фиксированную длину, то при получении последней записи заголовка возможно немедленно выяснить значения всех высказываний, описывающих состояние программы, в момент начала обработки события.

При обнаружении несоответствия записей протокола заданной спецификации требуется построить контрпример: путь в программе автоматов, который приводит к нарушению условий спецификации. При использовании предложенного метода для построения контрпримера достаточно взять заголовки секций протокола от начала до момента обнаружения нарушения спецификации. Эта последовательность записей в совокупности с заголовком протокола и даст путь в автоматной программе, являющийся контрпримером.

2.6. Формализация требований к автоматным программам

В предыдущих разделах настоящей главы были описаны три вида формализации требований к автоматным программам: темпоральные спецификации, контракты и тесты. Также были описаны статический и динамический подходы к верификации автоматных программ. В настоящем

разделе предлагается метод, позволяющий выбрать способ формализации требований к автоматной программе. Предложенный метод обладает рядом преимуществ перед традиционным подходом, при котором все требования формализуются в виде темпоральных формул для верификации на модели.

Как было отмечено выше, основными ограничениями традиционного подхода являются высокая трудоемкость верификации на модели, а также невозможность верификации интеграции объекта управления и управляющего автомата.

В разд. 2.3 был предложен подход к автоматному программированию по контрактам, устраняющий *первое из приведенных ограничений*. Этот подход позволяет записывать ряд требований более компактно, чем это удается сделать с помощью формул темпоральной логики. Также запись требований в виде контрактов упрощает поддержку автоматных программ, так как спецификация оказывается более локализованной. Верификация при использовании контрактов выполняется теми же методами, что и при использовании темпоральных формул. Таким образом, контракты так же эффективны в верификации, как и темпоральные формулы, но они удобнее в использовании и проще. В связи с этим предлагается использовать контракты во всех случаях, когда это возможно.

Второе ограничение частично устраняется использованием метода динамической верификации. Этот метод позволяет проверять соответствие спецификации во время работы программы. Его недостатком является то, что он основан на тестировании, а значит, не может гарантировать соответствие программы спецификации во всех случаях. Однако этот метод позволяет повысить качество автоматных программ тогда, когда статическая верификация неприменима. В связи с этим рекомендуется использовать динамическую верификацию в таких случаях.

Резюмируем вышесказанное. Если формализуемое требование является предусловием, постусловием или инвариантом для состояния, входного или

выходного воздействия, его следует записывать в виде контракта. Если верификация требования не может быть выполнена статически, следует использовать динамическую верификацию.

Выводы по главе 2

1. Существующий подход к формализации требований к автоматным программам имеет ряд недостатков, одним из которых является сложность получаемых в некоторых случаях формул.
2. Разработан подход к автоматному программированию по контрактам, который позволяет более компактно записывать ряд требований, при этом привязывая их к отдельным элементам автоматного интерфейса.
3. Предложен подход к динамической верификации автоматных программ, позволяющий проверить, работает ли такая программа в соответствии с заданной темпоральной спецификацией.
4. На основе предложенных подходов разработан метод формализации требований к автоматным программам, позволяющий формализовать более широкий спектр требований по сравнению с существующим подходом, а также получать более компактные формальные спецификации.

ГЛАВА 3. ВНЕСЕНИЕ ИЗМЕНЕНИЙ В АВТОМАТНЫЕ ПРОГРАММЫ

3.1. Внесение изменений в автоматные программы

Любая длительное время используемая программа подвергается модификации. Это связано с рядом естественных причин: в ходе эксплуатации программы могут выявиться требования, которые не были очевидны изначально, а также могут обнаружиться ошибки в работе программы. Наконец, для проведения описанных изменений может возникнуть потребность изменить структуру программы с целью ее упрощения. Опытные программисты знают, что хорошую структуру удастся создать не сразу – она должна развиваться по мере накопления опыта [5]. Поэтому почти любая программная программа рано или поздно подвергается изменениям.

С появлением большого числа программ с явным выделением состояний возникает необходимость в поддержке таких программ. Из сказанного выше следует, что в существующую программу вносят изменения следующих типов:

- изменения в программе в соответствии с изменившимися требованиями к ней;
- исправление ошибок;
- изменения в программе, имеющие целью облегчить понимание ее работы и упростить модификацию, не затрагивая наблюдаемого поведения.

По аналогии с существующим понятием *рефакторинга* [58] объектно-ориентированных программ, будем называть изменения последнего типа, применяемые к программам с явным выделением состояний, *рефакторингом автоматов*.

Внесение любых изменений в работающую программу сопряжено с риском появления ошибок и грозит потерей надежности программы. Вместе с тем, надежность часто является важнейшим требованием, предъявляемым к программе с явным выделением состояний. Поэтому исследования, направленные на разработку подхода к безопасному внесению изменений в такие программы, являются актуальными. Эти исследования позволяют упростить процесс разработки и повысить качество создаваемых программ.

В главе 1 выше введены понятия валидации и верификации автоматных программ. Будем автомат называть *синтаксически корректным*, если он проходит валидацию. Важно отметить, что синтаксически корректный автомат удовлетворяет требованиям полноты и непротиворечивости: множество исходящих переходов для любого состояния полно и непротиворечиво. Это означает, что при обработке любого события выполняется ровно один переход.

Будем называть автомат *семантически корректным*, если его выполнение согласовано со спецификацией. Заметим, что спецификация может быть как неформальной – например, задавать требования словесно, так и формальной – например, задаваться с помощью темпоральной логики. При этом выполнение спецификации, заданной формально, в некоторых случаях можно *верифицировать*.

Синтаксически и семантически корректный автомат будем называть *корректным*. Безопасными изменениями будем называть такие изменения автоматов, которые сохраняют их корректность.

К сожалению, далеко не все изменения являются безопасными. Часто разработчики, столкнувшиеся с необходимостью внесения изменений в программу с явным выделением состояний, вынуждены проектировать ее заново или бессистемно вносить изменения, руководствуясь только собственным пониманием. Такие подходы являются неэффективными и ненадежными, так как даже простейшие изменения влияют на корректность

автоматов и могут привести к появлению трудно находимых ошибок. Например, добавление одного перехода между двумя состояниями автомата может нарушить непротиворечивость множества переходов автомата.

3.2. Классификация изменений автоматных программ

Часто изменения, вносимые в программу с явным выделением состояний, достаточно сложны, а потому порождают массу проблем, плохо поддающихся анализу. С другой стороны, существует набор *базовых* изменений, которые являются «примитивными» изменениями какой-то одной составляющей графа переходов автомата. Такие изменения достаточно хорошо поддаются описанию. Остальные, более сложные изменения, назовем *составными*. Составные изменения можно представить в виде набора базовых изменений.

В отдельный класс выделим *рефакторинги автоматов*. Как было сказано выше, рефакторинги не меняют поведение программы и применяются для улучшения ее структуры.

3.3. Описание базовых изменений автоматов

В данном разделе приводится каталог базовых изменений автоматов. На основе этих изменений строятся более сложные сценарии реконфигурации автомата. Базовые изменения могут быть деструктивными и нарушать полноту и непротиворечивость графа переходов, а также семантические свойства. Для каждого базового изменения приводятся рекомендации по проведению такого изменения.

При описании каждого базового изменения графа переходов автомата будем придерживаться определенного формата, приведенного ниже:

- *название* изменения;
- *неформальное описание* приводимого изменения;
- *рекомендации* по проведению такого изменения.

3.3.1. Добавление состояния

Описание. В граф переходов добавляется новое состояние.

Рекомендации.

- Обеспечить достижимость состояния.

3.3.2. Удаление состояния

Описание. Из графа переходов удаляется состояние и все связанные с этим состоянием переходы.

Рекомендации.

- Если удаляемое состояние является начальным, необходимо выбрать новое начальное состояние.
- Если удаляемое состояние является конечным, необходимо пересмотреть набор конечных состояний.
- Следует также учитывать описанные ниже рекомендации при удалении перехода.

3.3.3. Установка начального состояния

Описание. Состояние объявляется начальным в автомате.

Рекомендации

- Необходимо проверить, что нет других начальных состояний.

3.3.4. Снятие начального состояния

Описание. Начальное состояние объявляется нормальным.

Рекомендации

- Необходимо объявить начальным другое состояние.

3.3.5. Добавление конечного состояния

Нормальное состояние объявляется конечным в автомате.

Рекомендации

- Если из этого состояния выходили какие-то переходы, то их необходимо удалить (следуя рекомендациям при удалении перехода).

3.3.6. Удаление конечного состояния

Конечное состояние объявляется нормальным.

Рекомендации

- Необходимо пересмотреть набор конечных состояний в автомате.

3.3.7. Добавление перехода

Описание. Добавление перехода между двумя состояниями. На переходе указывается событие, по которому данный переход осуществляется. Также может указываться условие и выходное воздействие.

Рекомендации

- Необходимо проверить полноту и непротиворечивость условий на переходах.

3.3.8. Изменение события на переходе

Описание. Изменяется событие, по которому активизируется переход.

Рекомендации

- Проверить полноту и непротиворечивость условий на переходах.

3.3.9. Изменение условия на переходе

Описание. Изменяется условие, при котором осуществляется переход.

Рекомендации

- Проверить полноту и непротиворечивость условий на переходах.

3.3.10. Удаление перехода

Описание. В автомате удаляется переход.

Рекомендации

- Проверить полноту и непротиворечивость условий на переходах.

- Проверить достижимость состояний.

3.3.11. Перемещение перехода

Описание. Переход имеет начальное и конечное состояния. При перемещении перехода возможно:

- изменение начального состояния;
- изменение конечного состояния.

Данное изменение формально не является базовым, так как его можно разбить на более простые изменения. Тем не менее, оно рассматривается нами, поскольку перемещение перехода – это регулярно используемая модификация.

Рекомендации

- Проверить полноту и непротиворечивость условий на переходах.
- Проверить достижимость состояний в автомате.

3.4. Рефакторинг автоматных программ

В данном разделе описывается каталог рефакторингов автоматных программ – изменений программ, не меняющих их поведение, но улучшающих их структуру.

Выбор рефакторингов для каталога основан на наборе экспериментов. В качестве основы были взяты несколько автоматов, созданных студентами кафедры «Компьютерные технологии» СПбГУ ИТМО в ходе выполнения курсовых работ по курсу «Применение автоматов в программировании» [59]. Далее производились изменения требований к автомату, основанные на анализе предметной области. В соответствии с изменениями требовалось изменить структуру автомата. При этом часто даже простые изменения спецификации требовали значительной модификации графа переходов, так как его исходная структура оказывалась неприспособленной к внедряемому изменению. Для обеспечения совместимости структуры автомата и изменения спецификации требовалось произвести рефакторинг:

модифицировать структуру автомата таким образом, чтобы изменения, вызванные изменившейся спецификацией, естественно вписались в новую структуру. На основе таких изменений и был разработан нижеприведенный каталог рефакторингов автоматных программ.

Описание каждого рефакторинга имеет следующую структуру:

- сначала следует *название* рефакторинга;
- за названием следует *неформальное описание* изменения;
- *мотивация* описывает, почему следует пользоваться этим методом рефакторинга;
- *пример* применения рефакторинга;
- *техника* содержит описание пошагового выполнения рефакторинга;
- *доказательство* корректности рефакторинга.

В конце описания каждого рефакторинга приводится доказательство его корректности: формально доказывается, что выполнение рефакторинга не меняет логику работы автомата. Это означает, что после рефакторинга должен получиться автомат, эквивалентный исходному. Эквивалентными называются такие автоматы, которые на любую последовательность входных воздействий реагируют одинаковыми последовательностями выходных воздействий.

Будем рассматривать последовательность событий $e_{i_1}, e_{i_2}, \dots, e_{i_k}$. При доказательстве эквивалентности автоматов будем придерживаться следующих обозначений: A – изначальный автомат, A' – автомат с внесенными изменениями. Пусть при этом *цепочка элементов* – последовательность состояний, событий и выходных воздействий автомата A имеет вид: $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots$, а аналогичная последовательность для автомата A' – $s'_{j_0}, z'_{k_0}, e_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, \dots$, где:

- s_{j_t} и s'_{j_t} – состояния,
- z_{k_t} и z'_{k_t} – выходные воздействия, совершаемые при входе в состояния s_{j_t} и s'_{j_t} соответственно,
- z_{l_t} и z'_{l_t} – выходные воздействия, совершаемые на переходах после события e_{i_t} .

Если автоматы A и A' не эквивалентны, то найдется такое $t > 0$, что $z_{k_t} \neq z'_{k_t}$ или $z_{l_t} \neq z'_{l_t}$. При доказательстве эквивалентности автоматов будем опираться на технику выполнения рефакторинга.

3.4.1. Группировка состояний

Описание. Несколько простых состояний объединяются в группу состояний. При этом добавляются групповые переходы, заменяющие одинаковые переходы, исходящие из всех группируемых состояний (рис. 3).

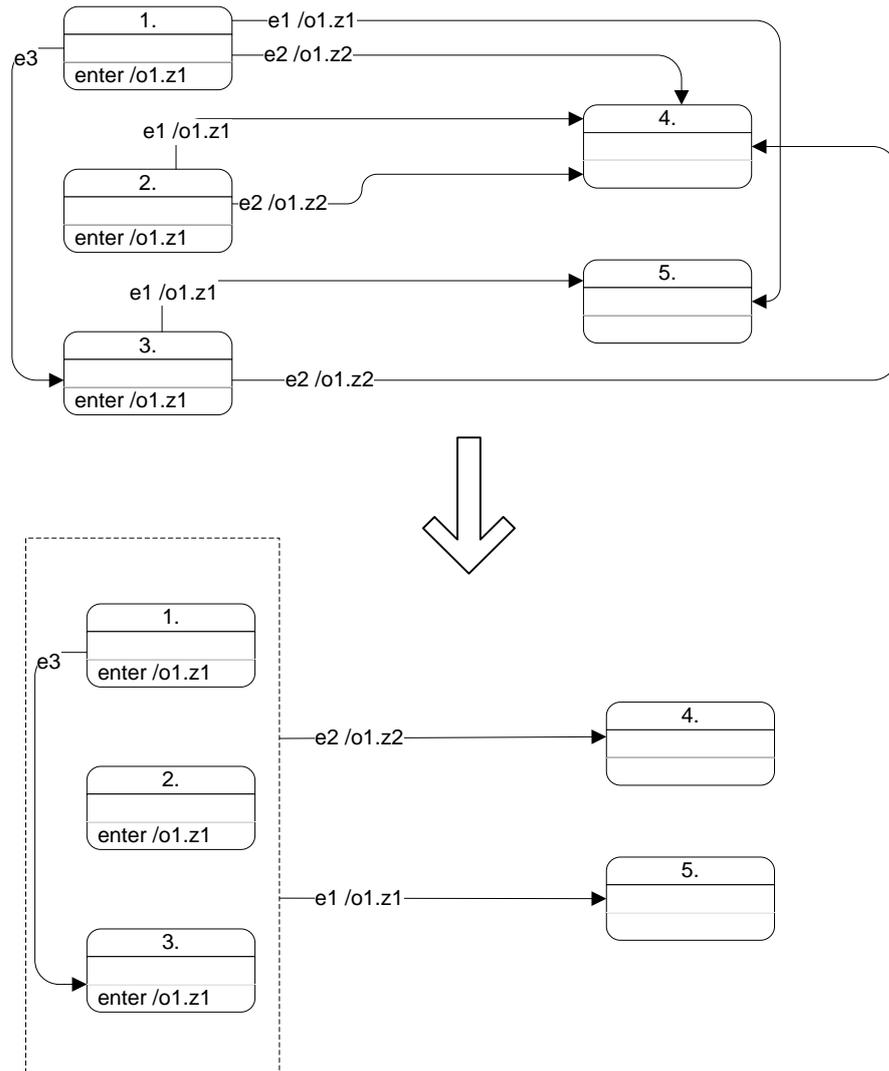


Рис. 3. Пример применения рефакторинга «группировка состояний»

Состояния, объединяемые в группу, могут иметь по несколько одинаковых переходов, в этом случае добавляется несколько групповых переходов.

Мотивация

Часто автомат имеет несколько состояний, которые имеют одинаковые переходы в другие состояния автомата. В этом случае можно выделить группу состояний, упростив структуру автомата.

Пример

Рассмотрим фрагмент графа переходов автомата «Панель в кабине лифта», отвечающий за выключение ламп в кнопках. Автомат имеет следующие пять состояний:

0. Кнопки погашены.
1. Светится «1».
2. Светится «2».
3. Светится «3».
4. Светится «S».

При наступлении события e («Выключение лампы в кнопке») в каждом из состояний 1–4 автомат должен перейти в состояние 0. Такое поведение реализуется следующим графом переходов (рис. 4).

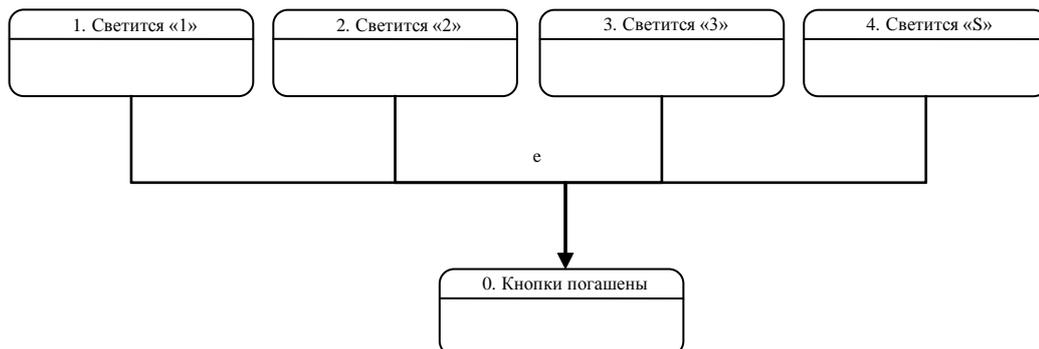


Рис. 4. Граф переходов без группировки состояний

Так как переходы из состояний 1–4 идентичны, их можно сгруппировать и соответствующим образом изменить конфигурацию автомата (рис. 5).

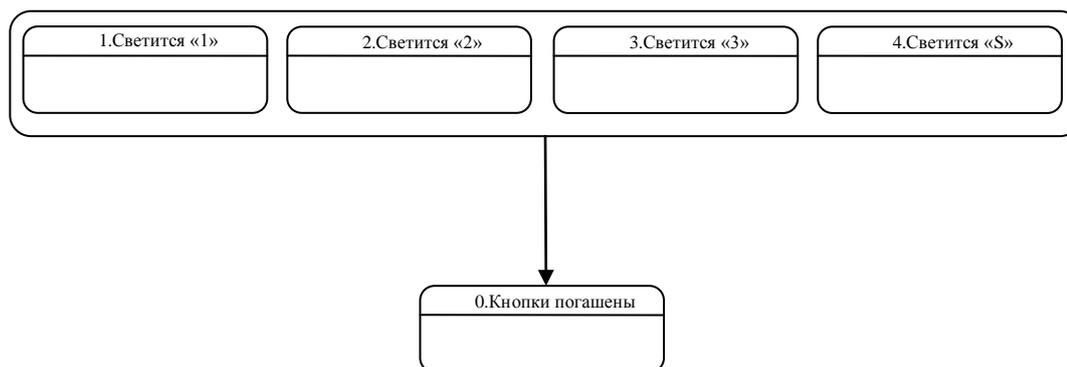


Рис. 5. Граф переходов с группировкой состояний

Техника

Для объединения состояний s_1, s_2, \dots, s_k в группу необходимо выполнить следующие действия:

1. Добавить группу g , объединяющую состояния s_1, s_2, \dots, s_k .
2. Выбрать один переход t , исходящий из s_1 , который требуется заменить групповым.
3. Убедиться, что каждое из состояний s_2, \dots, s_k имеет переход с такими же атрибутами, что и t (под атрибутами понимаются конечное состояние, событие, условие и выходные воздействия).
4. Добавить переход, исходящий из группы g , и имеющий те же атрибуты, что и t .
5. Удалить переходы, отмеченные в пунктах 2, 3.
6. Для оставшихся переходов, которые нужно заменить групповыми, повторить шаги 2–6.

Доказательство корректности. Так как групповой переход идентичен переходу из каждого состояния, измененный автомат A' будет иметь такую же цепочку элементов $s_{j_1}, z_{k_1}, e_{i_1}, z_{l_1}, s_{j_2}, z_{k_2}, e_{i_2}, z_{l_2}, \dots$, что и автомат A . Поэтому они будут эквивалентны.

3.4.2. Удаление группы состояний

Описание. При удалении группы состояний все исходящие из него переходы добавляются в состояния, находящиеся в группе, после чего сгруппированные состояния выносятся из группы.

Мотивация

Такое изменение полезно для последующей модификации автомата, если одно из состояний, входящее в удаляемую группу, изменяет логику поведения.

Техника

Для удаления группы g , объединяющей состояния s_1, s_2, \dots, s_k , необходимо выполнить следующие действия:

1. Выбрать переход t , исходящий из группы g .
2. Добавить переходы, исходящие из состояний s_i ($i = 1..k$), с атрибутами перехода t .
3. Удалить переход t .
4. Для оставшихся переходов, исходящих из группы g , повторить шаги 1–4.
5. Удалить группу g .

Доказательство корректности. Аналогично предыдущему рефакторингу.

3.4.3. Слияние состояний

Описание. Несколько состояний с одинаковыми исходящими переходами сливаются в одно состояние. При этом сохраняются переходы, соединяющие эти состояния с другими состояниями автомата.

Производить слияние состояний можно только в том случае, если одинаковы атрибуты переходов, исходящих из этих состояний, и одинаковы воздействия, вызываемые при входе в сливаемые состояния.

Мотивация

В процессе изменения программы может оказаться, что некоторые состояния дублируют логику поведения программы. В этом случае эти состояния могут быть заменены одним состоянием.

Пример

Рассмотрим пример, иллюстрирующий применение рефакторинга «слияние состояний». Для этого рассмотрим модификацию примера «Панель в кабине лифта», предложенного в разделе 3.4.1, с добавлением следующих состояний:

5. Перегорела лампа в кнопке «1»;
6. Перегорела лампа в кнопке «2»;
7. Перегорела лампа в кнопке «3»;
8. Перегорела лампа в кнопке «S»;
9. Неисправность.

Измененный граф переходов представлен на рис. 6:

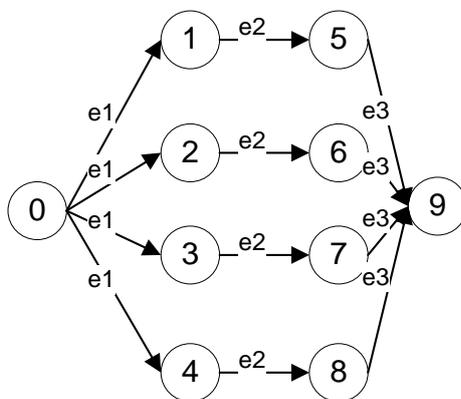


Рис. 6. Граф переходов автомата управления лифтом до слияния состояний 5–8

Граф переходов автомата после слияния состояний 5–8 в одно состояние представлен на рис. 7.

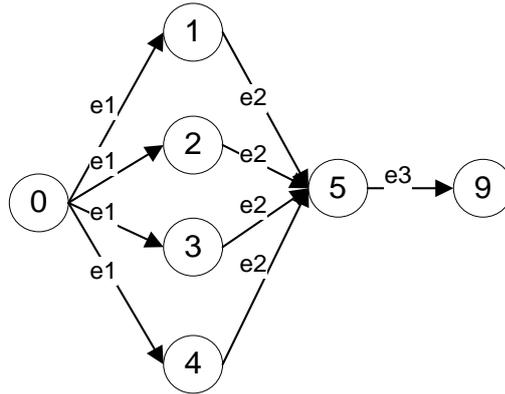


Рис. 7. Граф переходов после слияния состояний

Техника

Для слияния состояний s_1, s_2, \dots, s_k необходимо выполнить следующие действия:

1. Убедиться, что состояния s_1, s_2, \dots, s_k имеют одинаковые воздействия, вызываемые при входе в состояние.
2. Убедиться, что состояния s_1, s_2, \dots, s_k имеют исходящие переходы с одинаковыми атрибутами, и эти переходы заканчиваются в одном и том же состоянии.
3. Изменить конечные состояния переходов, которые ведут в состояния s_2, \dots, s_k , на состояние s_1 .
4. Удалить состояния s_2, \dots, s_k .

Доказательство корректности. Между состояниями автоматов A и A' строится соответствие $f: S \rightarrow S'$. При этом все сливаемые состояния переходят в одно состояние s'_1 : $f(s_1) = s'_1, f(s_2) = s'_1, \dots, f(s_k) = s'_1$.

Начальное состояние автомата A соответствует начальному состоянию автомата A' . Тогда в цепочках элементов $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots$ и $s'_{j_0}, z'_{k_0}, e'_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e'_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, \dots$ первые состояния s_{j_0} и s'_{j_0} соответствуют друг другу: $f(s_{j_1}) = s'_{j_1}$ и $z_{k_0} = z'_{k_0}$.

Докажем, что если в цепочках совпадают первые t ($t > 0$) элементов, то $(t+1)$ -е элементы также совпадают. Другими словами, если выполняется: $f(s_{j_x}) = s'_{j_x}$, $z_{k_x} = z'_{k_x}$, $z_{l_x} = z'_{l_x}$ для всех $x \leq t$, то также верно, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$. Состояние s_{j_t} может входить или не входить в множество сливаемых состояний (s_1, s_2, \dots, s_k) . Рассмотрим оба случая:

- 1) s_{j_t} не входит в множество сливаемых состояний:
 - а) $s_{j_{t+1}}$ входит в множество сливаемых состояний s_1, s_2, \dots, s_k . Тогда $s_{j_{t+1}} = s_i$ для некоторого i . Получаем $f(s_{j_{t+1}}) = f(s_i) = s'_i = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ (см. пункт 3 техники), $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 1 техники).
 - б) $s_{j_{t+1}}$ не входит в множество сливаемых состояний s_1, s_2, \dots, s_k . Тогда $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$, так как соответствующий переход и состояние не претерпевали никаких изменений.
- 2) s_{j_t} входит в множество сливаемых состояний:
 - а) $s_{j_{t+1}}$ также входит в множество сливаемых состояний s_1, s_2, \dots, s_k . Тогда $s_{j_{t+1}} = s_i$ для некоторого i . Получаем $f(s_{j_{t+1}}) = f(s_i) = s'_i = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ (см. п. 3 техники выполнения рассматриваемого рефакторинга), $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 1 техники).
 - б) $s_{j_{t+1}}$ не входит в множество сливаемых состояний s_1, s_2, \dots, s_k . Но так как s_1, s_2, \dots, s_k имеют исходящие переходы с одинаковыми атрибутами, в том числе с одинаковыми конечными состояниями (см. п. 2 техники), то после события $e_{i_{t+1}}$ автомат A' перейдет в состояние $s'_{j_{t+1}} = f(s_{j_{t+1}})$. Тогда $z'_{l_{t+1}} = z_{l_{t+1}}$ (выходное воздействие в состоянии $s_{j_{t+1}}$ осталось неизменным) и $z'_{k_{t+1}} = z_{k_{t+1}}$ (согласно п. 2 техники).

Корректность рассмотренного рефакторинга доказана.

3.4.4. Выделение автомата

Описание. Часть логики программы переносится в отдельный вызываемый автомат (рис. 8).

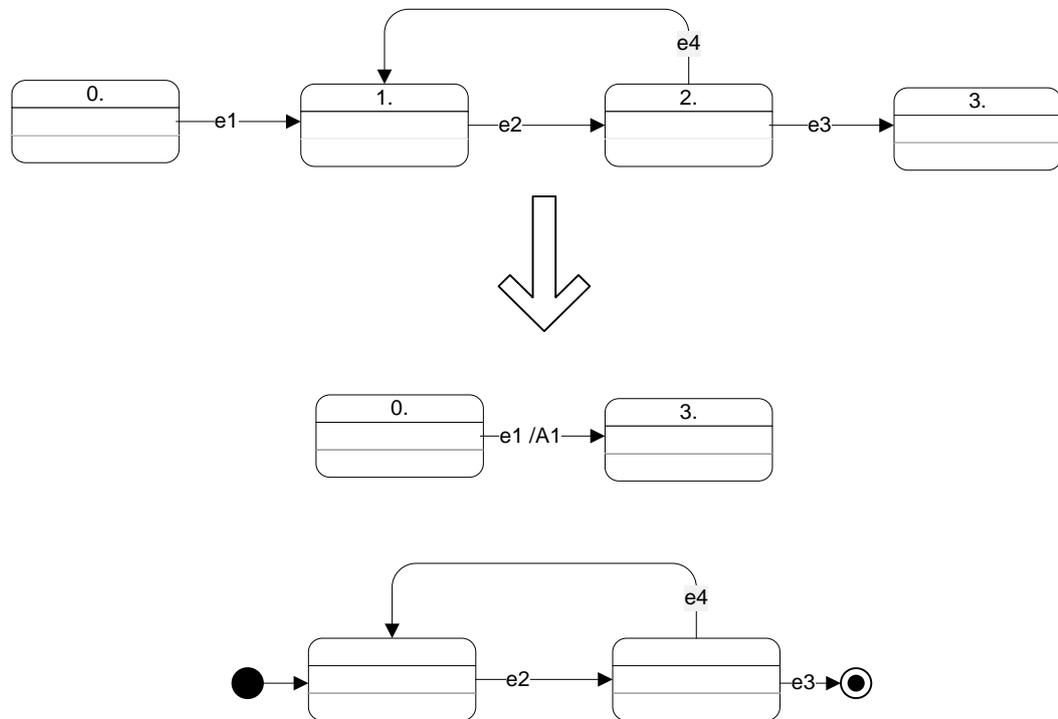


Рис. 8. Выделение автомата

Мотивация

Большие автоматы с множеством состояний, реализующие сразу несколько функций программы сложны в поддержке. При внесении изменений в работу одной из функций необходимо учитывать реализацию других функций. Понимание работы такого автомата усложнено тем, что протоколы взаимодействия частей автомата, отвечающих за реализацию различных функций, не определены четко. Соответственно, поддержка сложных автоматов может привести к постоянному возникновению ошибок, обнаружить которые и исправить гораздо сложнее, чем в программах, в которых различные функции реализованы отдельными автоматами.

Для решения этой проблемы сложный автомат разделяют на несколько более простых, для каждого из которых четко определены обязанности и

протоколы взаимодействия с другими автоматами и остальной программой. Такое разделение называют *автоматной декомпозицией* [26]. Автоматная декомпозиция обычно производится на этапе анализа требований, когда происходит построение изначального набора автоматов, которые требуется реализовать в программе. Однако часто в ходе эволюции программной программы один из автоматов накапливает реализацию чрезмерно большой функциональности. В этом случае требуется выполнить автоматную декомпозицию в уже существующей программе. Существует несколько критериев автоматной декомпозиции, следуя которым в большинстве случаев можно получить логичную архитектуру программы.

Декомпозиция *по режимам* уместна тогда, когда в поведении программы можно выделить несколько качественно различных режимов (каждый из которых, при необходимости, можно конкретизировать, выделив режимы более низкого уровня абстракции). В этом случае логично сопоставить автомат каждому из режимов, в которых поведение программы является сложным, или, иными словами, сопоставить отдельный автомат каждому абстрактному действию.

Декомпозиция *по объектам управления* применима в том случае, когда в программе присутствует несколько объектов управления. В этом случае логично поручить управление каждым из объектов отдельному автомату или поддереву в иерархии автоматов.

Следование такому критерию декомпозиции приводит к выделению в архитектуре программы пар автомат – объект управления (или группа автоматов – объект управления) и значительно приближает ее к «идеалу» парадигмы автоматного программирования – множеству взаимодействующих автоматизированных объектов управления.

Следует отметить, что техника реализации большинства рефакторингов не зависит от характера взаимодействия автоматов в программе. В некоторых случаях различные автоматы программы могут взаимодействовать

непосредственно (через автоматные интерфейсы). В других, как, например, при следовании парадигме *автоматизированные объекты управления как классы*, автоматы взаимодействуют друг с другом так же, как с объектами управления – через входные и выходные переменные. Однако, техника выполнения рефакторингов по выделению и встраиванию автомата определенно зависит от того каким образом взаимодействуют автоматы в программе. Для решения этой проблемы техника описанных рефакторингов приводится в двух вариантах: для программ, в которых автоматы взаимодействуют через вызовы соседних автоматов и для программ, в которых автоматы взаимодействуют друг с другом так же, как и с объектами управления.

Техника

Для выделения состояний s_1, s_2, \dots, s_k и исходящих из них переходов t_1, t_2, \dots, t_l в вызываемый автомат необходимо выполнить следующие действия:

1. Убедиться, что среди состояний s_1, s_2, \dots, s_k есть ровно одно такое состояние s_i , в которое ведет хотя бы один переход из состояния, не входящего в множество s_1, s_2, \dots, s_k . Обозначим отмеченный переход как t' , а его начальное состояние – s' .
2. Убедиться, что из отмеченного состояния s_i достижимы все остальные состояния выбранного подмножества.
3. Убедиться, что переходы t_1, t_2, \dots, t_l имеют в качестве конечных состояний только состояния из множества s_1, s_2, \dots, s_k и ровно одно состояние s'' , не входящее в это множество.
4. Создать новый автомат.
5. Создать в новом автомате состояния s'_1, s'_2, \dots, s'_k соответствующие состояниям s_1, s_2, \dots, s_k с сохранением выходных воздействий.

6. Добавить переходы между состояниями s'_1, s'_2, \dots, s'_k с такими же атрибутами, что и переходы между состояниями s_1, s_2, \dots, s_k .
7. Объявить состояние s'_i начальным в созданном автомате.
8. Выбрать среди состояний s_1, s_2, \dots, s_k такие, из которых есть переходы в s'' . Пусть в созданном автомате им соответствуют состояния $s'_{i_1}, s'_{i_2}, \dots, s'_{i_m}$. Добавьте из состояний $s'_{i_1}, s'_{i_2}, \dots, s'_{i_m}$ переходы, ведущие в конечное состояние созданного автомата. Присвойте этим переходам такие же атрибуты, как у соответствующих переходов изначального автомата.
9. Добавить переход между состояниями s' и s'' с атрибутами перехода t' и вызовом созданного автомата.
10. Удалить состояния s_1, s_2, \dots, s_k .

Доказательство корректности. Между состояниями автоматов A и A' естественным образом строится взаимно однозначное соответствие $f: S \rightarrow S'$.

При этом начальное состояние автомата A соответствует начальному состоянию автомата A' . Тогда в цепочках элементов $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots$ и $s'_{j_0}, z'_{k_0}, e'_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e'_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, \dots$ первые состояния s_{j_0} и s'_{j_0} соответствуют друг другу: $f(s_{j_1}) = s'_{j_1}$ и $z_{k_0} = z'_{k_0}$.

Докажем, что если в цепочках совпадают первые t ($t > 0$) элементов, то $(t+1)$ -е элементы также совпадают. Другими словами, если выполняется: $f(s_{j_x}) = s'_{j_x}, z_{k_x} = z'_{k_x}, z_{l_x} = z'_{l_x}$ для всех $x \leq t$, то также верно, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}, z_{l_{t+1}} = z'_{l_{t+1}}$. Рассмотрим несколько случаев:

- 3) s_{j_t} не входит в множество выделяемых в отдельный автомат состояний s_1, s_2, \dots, s_k :
 - а) $s_{j_t} = s'$. Тогда $s_{j_{t+1}} = s_i$ (см. п. 1 техники) или $s_{j_{t+1}}$ не входит в множество выделяемых в отдельный автомат состояний $s_1, s_2, \dots,$

s_k . В первом случае $f(s_{j_{t+1}}) = f(s_i) = s'_i = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ (см. п. 5 техники) и $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 9 техники). Во втором же случае сразу получаем $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$, так как соответствующий переход и состояние не претерпевали никаких изменений.

- b) $s_{j_t} \neq s'$. Тогда $s_{j_{t+1}}$ не входит в множество выделяемых в отдельный автомат состояний s_1, s_2, \dots, s_k , и переходы, ведущие из s_{j_t} , остались неизменными. Следовательно, $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.
- 4) s_{j_t} входит в множество выделяемых в отдельный автомат состояний s_1, s_2, \dots, s_k . Тогда либо $s_{j_{t+1}}$ также входит в множество состояний, выделяемых в отдельный автомат, либо $s_{j_{t+1}} = s''$ (см. пп. 3, 6, 8 техники):
- a) $s_{j_{t+1}}$ входит в множество выделяемых в отдельный автомат состояний s_1, s_2, \dots, s_k . Тогда согласно пп. 5, 6 техники $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.
- b) $s_{j_{t+1}} = s''$. Тогда из п. 8 техники получаем, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.

Корректность рефакторинга доказана.

Пример

Выделение автомата подробно рассматривается в разд. 3.6.

3.4.5. Встраивание вызываемого автомата

Описание. Вызываемый автомат встраивается в места своего вызова на переходах (рис. 9). Рефакторинг является обратным к предыдущему.

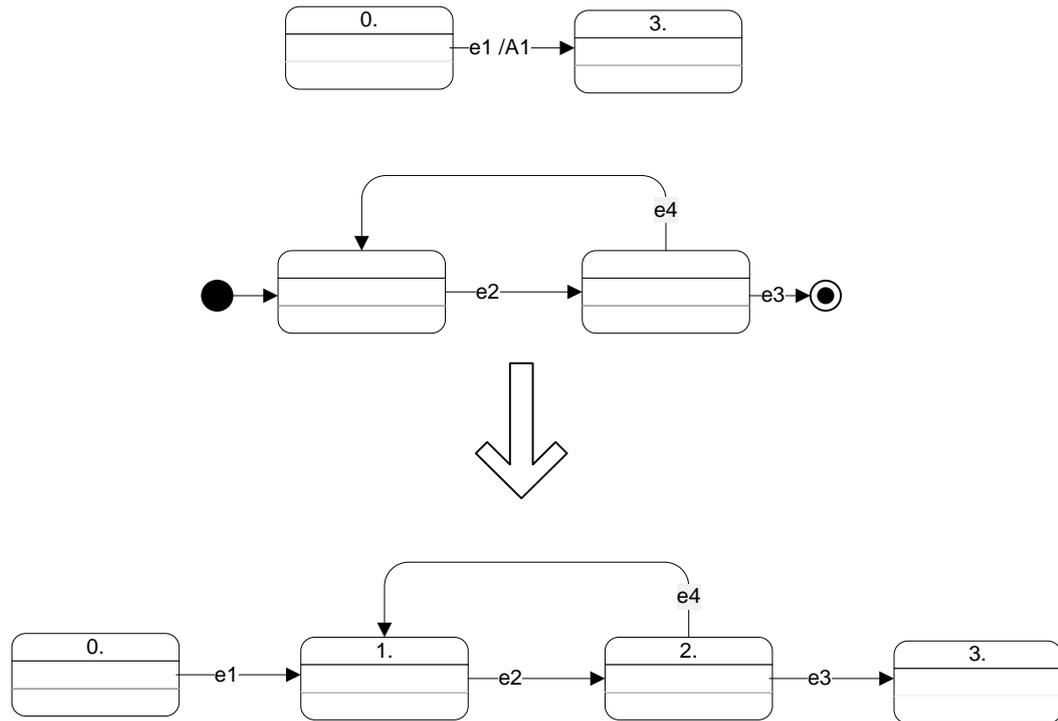


Рис. 9. Встраивание вызываемого автомата

Мотивация

Размещением всей логики поведения программы в одном графе переходов можно добиться большей наглядности, так как такой граф можно охватить «одним взглядом».

Тем не менее, следует данный рефакторинг применять с осторожностью, потому как излишне «разросшийся» граф переходов ничуть не понятнее программы более простых. Более того, встраивание автомата нарушает автоматную декомпозицию, и поэтому потенциально усложняет последующую модификацию программы.

Техника

Пусть встраиваемый автомат содержит состояния s_1, s_2, \dots, s_k (s_1 – начальное), а его вызов совершается на переходе t , соединяющем состояния s' и s'' автомата A . Для встраивания автомата необходимо выполнить следующие действия:

1. Добавить в автомат A состояния s'_1, s'_2, \dots, s'_k , соответствующие состояниям s_1, s_2, \dots, s_k с сохранением выходных воздействий.
2. Для каждого перехода t_i , соединяющего состояния s_i и s_j встраиваемого автомата добавить соответствующий переход с теми же атрибутами между парой состояний s'_i и s'_j .
3. Изменить конечное состояние перехода t на состояние s'_1 .
4. Для каждого перехода, ведущего из некоторого состояния s_i в конечное состояние встраиваемого автомата, добавить переход с такими же атрибутами, ведущий из состояния s'_i в состояние s'' .
5. Если вызовов автомата A больше нет, то удалить состояния s_1, s_2, \dots, s_k .

Доказательство корректности. По аналогии с предыдущим доказательством рассмотрим взаимно однозначное соответствие между состояниями изначального и измененного автоматов $f: S \rightarrow S'$. При этом начальное состояние автомата A соответствует начальному состоянию автомата A' . Тогда в цепочках элементов $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots$ и $s'_{j_0}, z'_{k_0}, e_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, \dots$ первые состояния s_{j_0} и s'_{j_0} соответствуют друг другу: $f(s_{j_1}) = s'_{j_1}$ и $z_{k_0} = z'_{k_0}$.

Докажем, что если в цепочках совпадают первые t ($t > 0$) элементов, то $(t+1)$ -ые элементы также совпадают. Другими словами, если выполняется: $f(s_{j_x}) = s'_{j_x}, z_{k_x} = z'_{k_x}, z_{l_x} = z'_{l_x}$ для всех $x \leq t$, то также верно, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}, z_{l_{t+1}} = z'_{l_{t+1}}$. Рассмотрим несколько случаев:

1) s_{j_t} не принадлежит автомату $A1$:

- а) $s_{j_t} = s'$. Тогда $s_{j_{t+1}} = s_1$ или $s_{j_{t+1}}$ также не принадлежит автомату $A1$. В первом случае $f(s_{j_{t+1}}) = f(s_1) = s'_1 = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}$ и $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 3 техники). Во втором же случае сразу получаем

$f(s_{j_{t+1}}) = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}, z_{l_{t+1}} = z'_{l_{t+1}}$, так как соответствующий переход и состояние не претерпевали никаких изменений.

- b) $s_{j_t} \neq s'$. Тогда $s_{j_{t+1}}$ не принадлежит автомату AI , и переходы, ведущие из s_{j_t} , остались неизменными. Следовательно, $f(s_{j_{t+1}}) = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}, z_{l_{t+1}} = z'_{l_{t+1}}$.
- 2) s_{j_t} принадлежит автомату AI . Тогда либо $s_{j_{t+1}}$ также принадлежит AI , либо $s_{j_{t+1}} = s''$ (см. пп. 2, 4 техники):
- a) $s_{j_{t+1}}$ принадлежит AI , то есть входит в множество состояний s_1, s_2, \dots, s_k . Тогда согласно пункту 2 техники $f(s_{j_{t+1}}) = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}, z_{l_{t+1}} = z'_{l_{t+1}}$.
- b) $s_{j_{t+1}} = s''$. Тогда из пункта 4 техники получаем, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}, z_{k_{t+1}} = z'_{k_{t+1}}, z_{l_{t+1}} = z'_{l_{t+1}}$.

Корректность рефакторинга доказана.

3.4.6. Переименование состояния

Описание. Изменение имени состояния.

Мотивация

Описанные до сих пор рефакторинги изменяли структуру автомата, адаптировали ее для лучшей реализации текущей или изменившейся спецификации. Между тем, часто большей прозрачности и понятности структуры автомата можно добиться простой сменой имени одного или нескольких состояний. Часто при анализе автоматов и их рефакторинге требуется изменить имена состояний так, чтобы они лучше отражали их семантику. Четкое и полное выражение смысла состояния одним-двумя словами часто может оказаться нетривиальной задачей, которая не поддается решению с первого раза. Со временем, когда программист работает с задачей уже несколько дней или недель, в голову часто приходят более удачные метафоры.

Наименования являются важной частью информационной составляющей описания автомата и в значительной степени определяют скорость восприятия его семантики. В некоторых случаях стоит выбрать более короткое имя для состояния, так как короткие имена делают граф переходов более компактным.

Переименование состояния является очень простым рефакторингом, но не стоит им пренебрегать: оно может существенно упростить восприятие логики поведения программы.

Техника

Для изменения имени состояния необходимо выполнить следующие действия:

1. Убедиться, что новое имя является уникальным для графа переходов.
2. Изменить имя состояния.

Доказательство корректности. Так как поведение автомата не зависит от имен состояний, переименование состояния не повлияет на работу автомата.

3.4.7. Перемещение воздействия из состояния в переходы

Описание. Вызов выходного воздействия, совершаемого при входе в состояние, перемещается в переходы, входящие в рассматриваемое состояние (рис. 10).

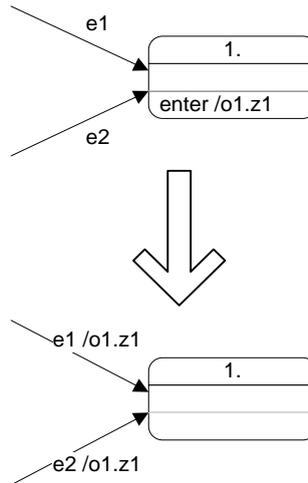


Рис. 10. Перемещение воздействия из состояния в переходы

Если при входе в состояние совершается несколько воздействий, то перемещается только первое. При необходимости рефакторинг можно повторить для остальных воздействий.

Техника

Для переноса воздействия из состояния в переходы необходимо выполнить следующие действия:

1. На каждом переходе, входящем в состояние, добавляется вызов воздействия. Если на переходах уже были выходные воздействия, то добавляемое воздействие становится последним.
2. Вызов воздействия удаляется из состояния.

Доказательство корректности. Пусть для некоторого $t : z_{k_t} \neq z'_{k_t}$ или $z_{l_t} \neq z'_{l_t}$. Рассмотрим среди всех таких t минимальное значение.

Тогда $s_{j_t} = s$. После внесения изменения цепочка элементов $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots, e_{i_t}, z_{l_t}, s_{j_t}, z_{k_t}, \dots$ будет иметь вид $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots, e_{i_t}, z_{l_t}, z_{k_t}, s_{j_t}, \dots$ ($z = z_{k_t}$). Легко видеть, что при этом общая последовательность выходных воздействий не изменяется. Аналогично для остальных t получаем, что последовательность выходных воздействий не изменится.

3.4.8. Перемещение воздействия из переходов в состояние

Описание. Вызовы одинаковых выходных воздействий, совершаемых на переходах, входящих в одно состояние, заменяются одним воздействием, выполняемым при входе в это состояние (рис. 11).

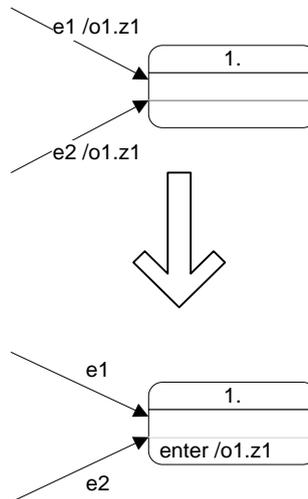


Рис. 11. Перемещение воздействия из переходов в состояние

Описываемое изменение будет являться рефакторингом, если одинаковое выходное воздействие имеют все переходы, входящие в состояние.

Пример

Перемещение воздействия из переходов в состояние подробно рассматривается в разд. 3.6.

Техника

Для перемещения воздействия z внутрь состояния s необходимо выполнить следующие действия:

1. Убедитесь, что воздействие z имеется на всех переходах, входящих в состояние s . Если на одном из переходов вызываются несколько воздействий, проверьте, что воздействие z вызывается последним.

2. Добавьте вызов воздействия z в конец списка воздействий, выполняемых при входе в состояние s .
3. Удалите вызовы воздействия z из всех переходов, входящих в состояние s .

Доказательство корректности. Пусть для некоторого $t : z_{k_t} \neq z'_{k_t}$ или $z_{l_t} \neq z'_{l_t}$. Рассмотрим среди всех таких t минимальное значение.

Тогда $s_{j_t} = s$. После внесения изменения цепочка элементов $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots, e_{i_t}, z_{l_t}, s_{j_t}, z_{k_t}, \dots$ будет иметь вид $s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, \dots, e_{i_t}, s_{j_t}, z_{l_t}, z_{k_t}, \dots$ ($z = z_{l_t}$). Легко видеть, что при этом общая последовательность выходных воздействий не меняется. Аналогично для остальных t получаем, что последовательность выходных воздействий не изменится.

3.5. Метод внесения изменений в автоматные программы

Как было показано в разд. 3.4, только рефакторинги могут обеспечить безопасность изменения – сохранение корректности автомата. Соответственно, изменения, совершаемые исключительно в целях изменения структуры автомата без изменения его поведения могут быть произведены совершенно безопасно, если они реализованы путем комбинирования нескольких рефакторингов.

Однако, согласно классификации изменений, введенной в разд. 3.1, два из трех типов изменений призваны модифицировать логику работы автомата для исправления ошибок или адаптации автомата к изменившимся требованиям. При выполнении таких изменений нельзя, не учитывая новой или старой спецификации, невозможно добиться гарантированной корректности измененного автомата.

Для проверки корректности изменения потребуется верифицировать получившийся автомат на соответствие оригинальной (в случае исправления

ошибки) или новой (в случае адаптации автомата к изменившимся требованиям) спецификации. Во второй главе указано, что ошибки, найденные при верификации, представлены в виде сценария, прохождение которого автоматом не соответствует спецификации. Результатом анализа такого сценария является коррекция либо автомата, либо спецификации. В нашем случае предполагается, что спецификация верна, и в коррекции нуждается автомат. Точнее, в коррекции нуждается набор действий по изменению автомата, после которого он перестал (или не начал) удовлетворять спецификации.

Из рассмотрения заведомо можно исключить рефакторинги: их безопасность формально доказана в разд. 3.4. Таким образом, анализу должен подвергнуться набор базовых изменений, не являющихся частью рефакторингов. К сожалению, полностью автоматизировать такой анализ не удается и его потребуется, как минимум частично, выполнять вручную. Так как ручной анализ изменений – достаточно трудоемкий процесс, чем меньший набор изменений требуется анализировать, тем лучше.

Вышесказанное приводит к следующему методу внесения изменений в графы переходов автоматных программ.

Основа метода заключается в разделении любого сложного изменения графа переходов на две фазы:

1. Рефакторинг автомата.
2. Набор модификаций, приводящих к изменению поведения автомата.

В ходе первой фазы автомат «подготавливают» к изменениям, модифицируя его структуру, не затрагивая при этом поведение (корректность этой фазы можно проверить автоматически, если спецификация исходного автомата была формализована). Целью рефакторинга является минимизация числа модификаций, выполняемых во второй фазе.

Модификации второй фазы, изменяющие поведение, делают максимально простыми, чтобы облегчить их анализ. Дополнительную помощь оказывают указанные в разд. 3.3 для каждого базового изменения списки потенциальных проблем, которые могут возникнуть при внесении соответствующего изменения. Эти списки помогут выполнять дополнительные проверки после каждого такого изменения. При этом валидация автомата после основных изменений может быть проведена автоматически, так как она не связана с семантической корректностью.

Предложенный метод позволяет избежать внесения в программу сложных изменений, с трудом поддающихся анализу. Наиболее сложные изменения доказуемо безопасны, потенциально опасные изменения просты.

Заключительным шагом должна стать верификация получившегося автомата на соответствие измененной формальной спецификации. В случае обнаружения несоответствия спецификации, использование предложенного метода значительно упрощает процедуру поиска ошибок, так как набор изменений, направленный на отражение новых требований, минимален.

3.6. Пример использования метода

Рассмотрим использование предложенного подхода на примере внесения изменений в автомат, отвечающий за работу банкомата. Моделирование работы банкомата с применением автоматного подхода осуществлено в работах [60, 61]. Приведем неформальное описание поведения программы.

Модель общения банкомата с сервером банка построена на основе транзакций: в ходе взаимодействия с пользователем устройство банкомата накапливает вводимую информацию в специальном внутреннем списке, отправляя серверу по требованию совершения операции полный набор информации, описывающий транзакцию. Так, в начале работы пользователь вводит номер карты. После этого банкомат запрашивает у него *PIN-код*. Как

номер карты, так и введенный *PIN*-код запоминаются во внутреннем списке. Банкомат запрашивает у сервера авторизацию для карты. В случае неверного ввода *PIN*-кода, карта возвращается.

После успешной проверки *PIN*-кода пользователю становится доступно основное меню, в котором он может выбрать одно из следующих действий:

- забрать карту;
- посмотреть остаток счета;
- снять деньги со счета.

При выборе первого пункта пользователю возвращается карта, и работа с банкоматом завершается. При выборе последнего пункта пользователю необходимо дополнительно выбрать одну из predetermined сумм для снятия или ввести сумму с клавиатуры. После этого, как и в случае выбора второго варианта, автомат отправляет серверу накопленные данные. При получении ответа от сервера клиенту отображается информация о результате операции. После этого банкомат вновь отображает главное меню, или пользователь забирает карту.

Управление осуществляется системой взаимодействующих автоматов. Формальное описание, содержащее в себе объекты управления, источники событий и систему автоматов, приведено ниже. На рис. 12 приведена схема связей автоматов.

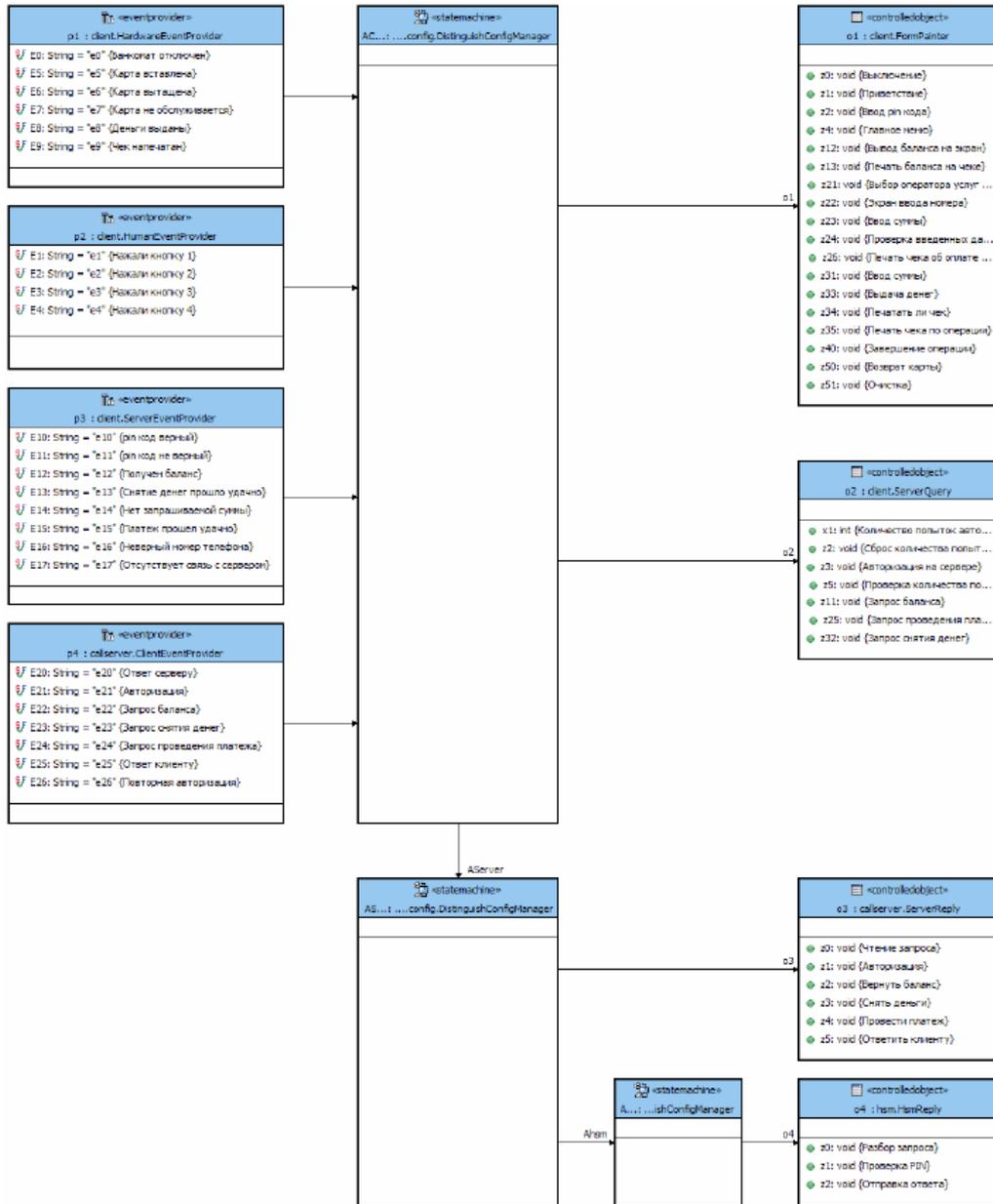


Рис. 12. Схема связей автоматов с поставщиками событий и объектами управления

На рис. 13 представлен граф переходов автомата, управляющего банкоматом.

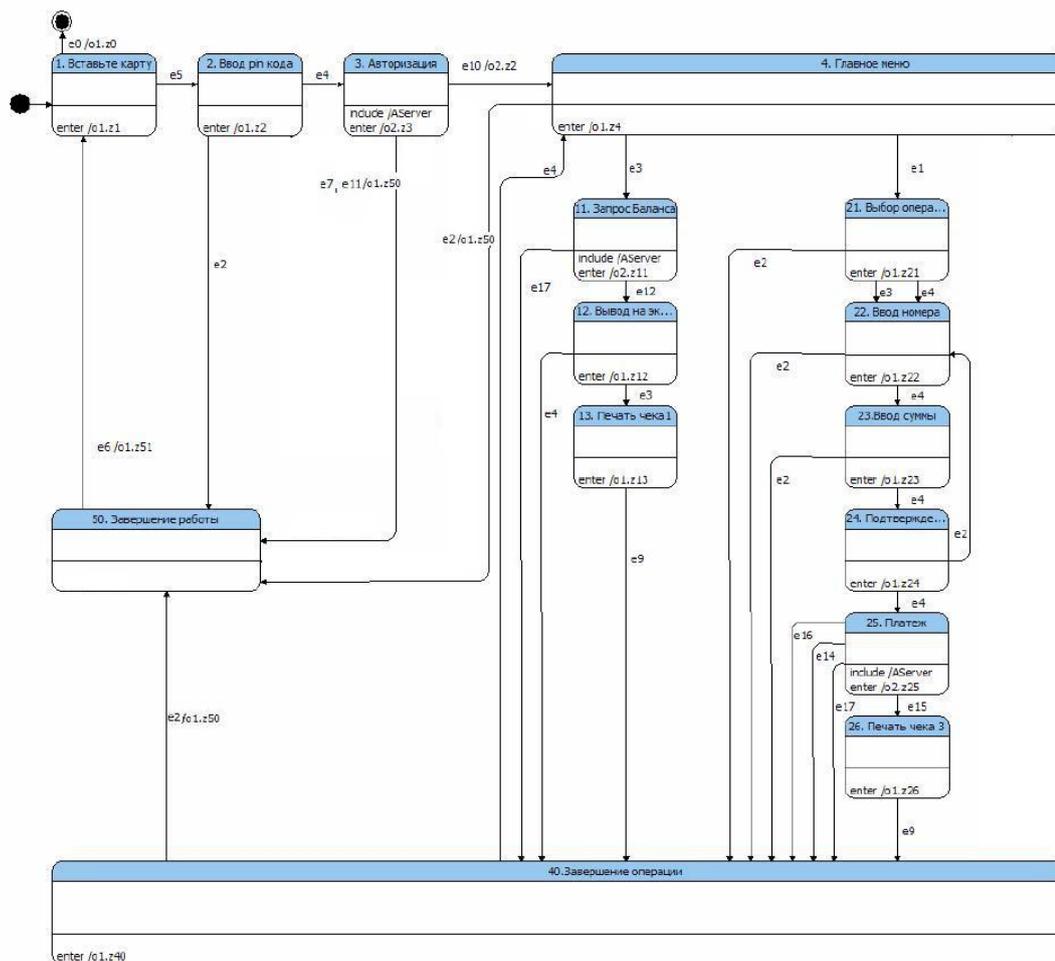


Рис. 13. Граф переходов автомата, отвечающего за работу банкомата

Рассмотрим ситуацию, когда к созданной программе изменяются требования. Пусть сценарий работы с банкоматом изменился следующим образом: «В случае неверного ввода *PIN*-кода он запрашивается повторно. В случае неверного ввода *PIN*-кода три раза подряд работа с банкоматом принудительно завершается».

Граф переходов автомата, отвечающего за работу банкомата, достаточно громоздок. Упростим его, выделив два вызываемых автомата: один для режима просмотра остатка счета, второй для режима снятия денег со счета. Граф переходов автомата после применения таких рефакторингов представлен на рис. 14.

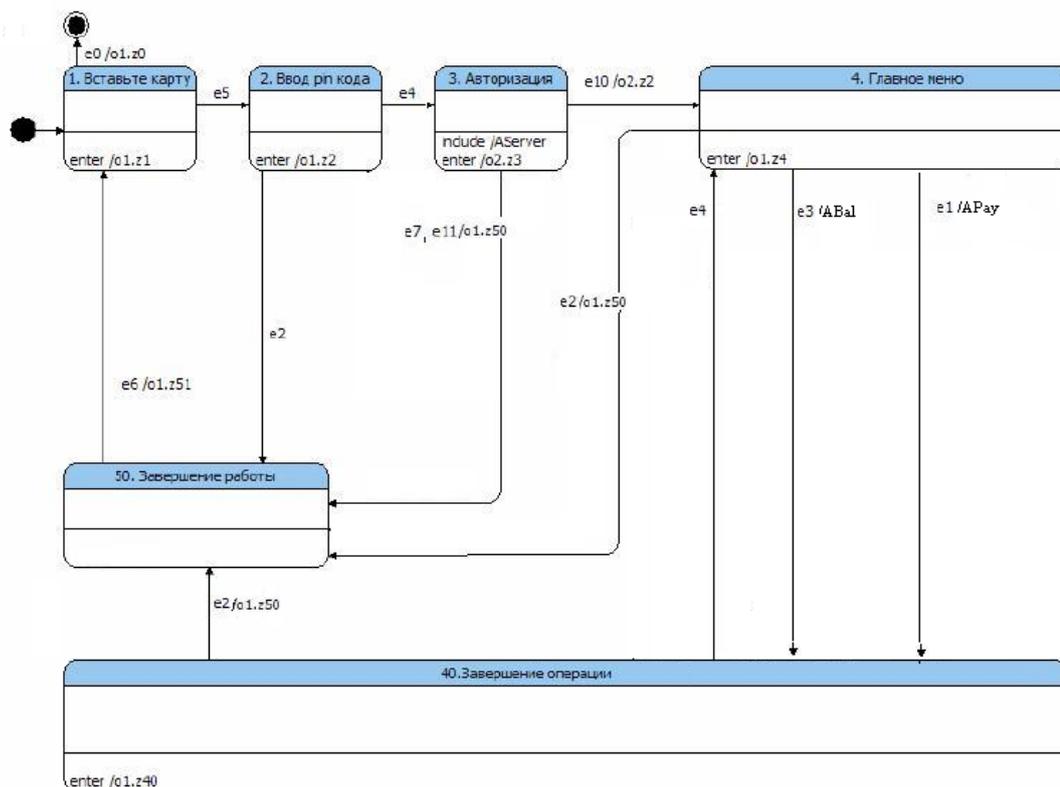


Рис. 14. Граф переходов автомата, отвечающего за работу банкомата, после выделения вызываемых автоматов

Граф переходов стал существенно проще для восприятия. В новом графе существует фрагмент, который нуждается в изменении: из состояния 3 («Авторизация») по событию $e11$ («pin код неверный») совершается действие $o1.z50$ («Возврат карты») и переход в состояние 50 («Завершение работы»). Необходимо, чтобы по событию $e11$ карта не возвращалась, а производился выбор действия в зависимости от числа неверных попыток ввода PIN -кода. Получается, что следует некоторым образом разделить связь события $e11$ и выходного воздействия $o1.z50$. Самый простой способ сделать это – осуществить перенос воздействий с переходов, ведущих в состояние 50, внутрь этого состояния. Граф переходов автомата, получающийся после применения такого рефакторинга, представлен на рис. 15.

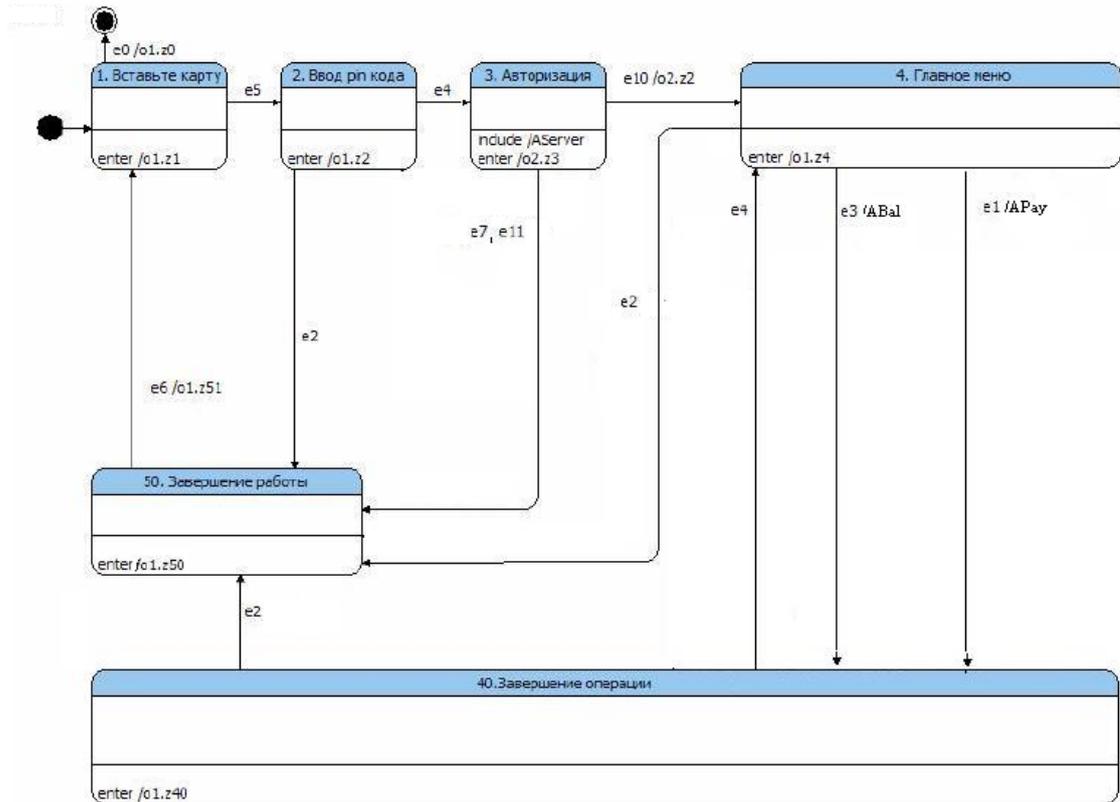


Рис. 15. Граф переходов автомата, отвечающего за работу банкомата, после переноса воздействия внутрь состояния

По событию $e11$ должно осуществляться ветвление в зависимости от числа попыток. Поэтому добавим в граф переходов состояние 5 («Неверный *PIN*-код»), в котором будет это ветвление осуществляться. Теперь установим конечным состоянием перехода по событию $e11$ добавленное состояние. Полученный граф переходов представлен на рис. 16.

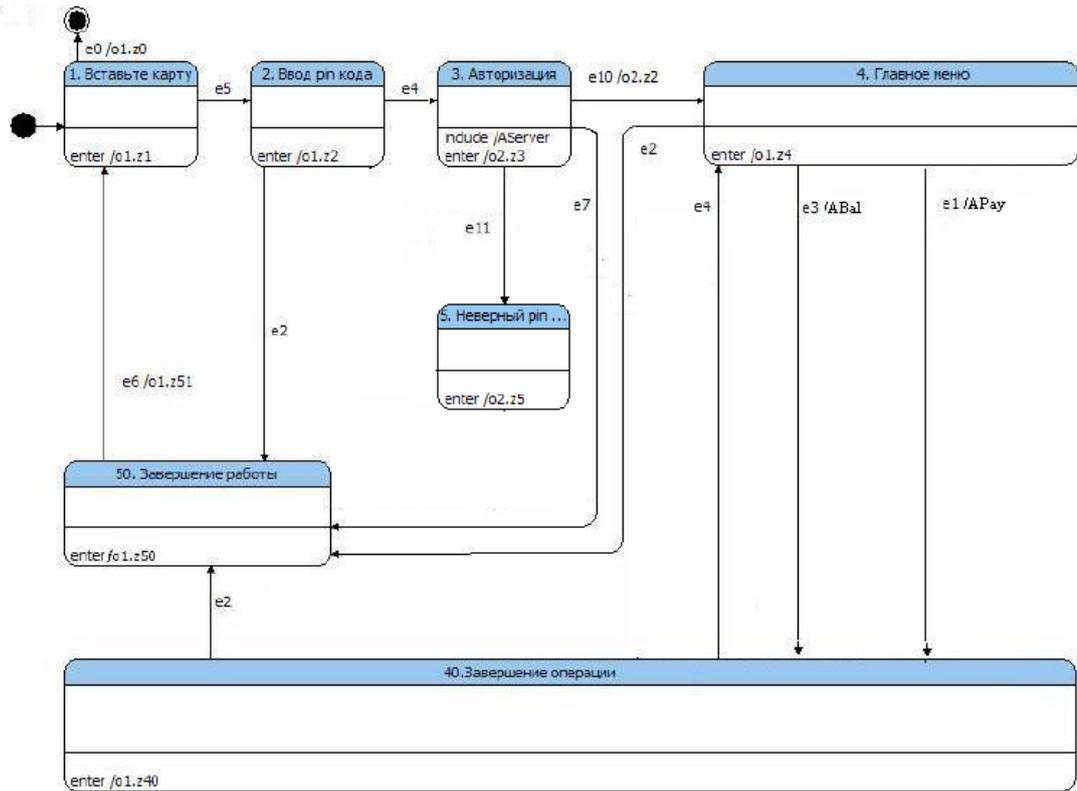


Рис. 16. Граф переходов автомата, отвечающего за работу банкомата, после добавления нового состояния

При входе в состояние 5 выполняется выходное воздействие $o2.z5$ («Проверка количества попыток»). Воздействие $o2.z5$ активизирует одно из двух событий: $e26$ («Повторная авторизация») или $e7$ («Карта заблокирована»). В первом случае, автомат должен перейти в состояние 2, во втором – в состояние 50. Соответствующие изменения графа переходов легко осуществить (рис. 17).

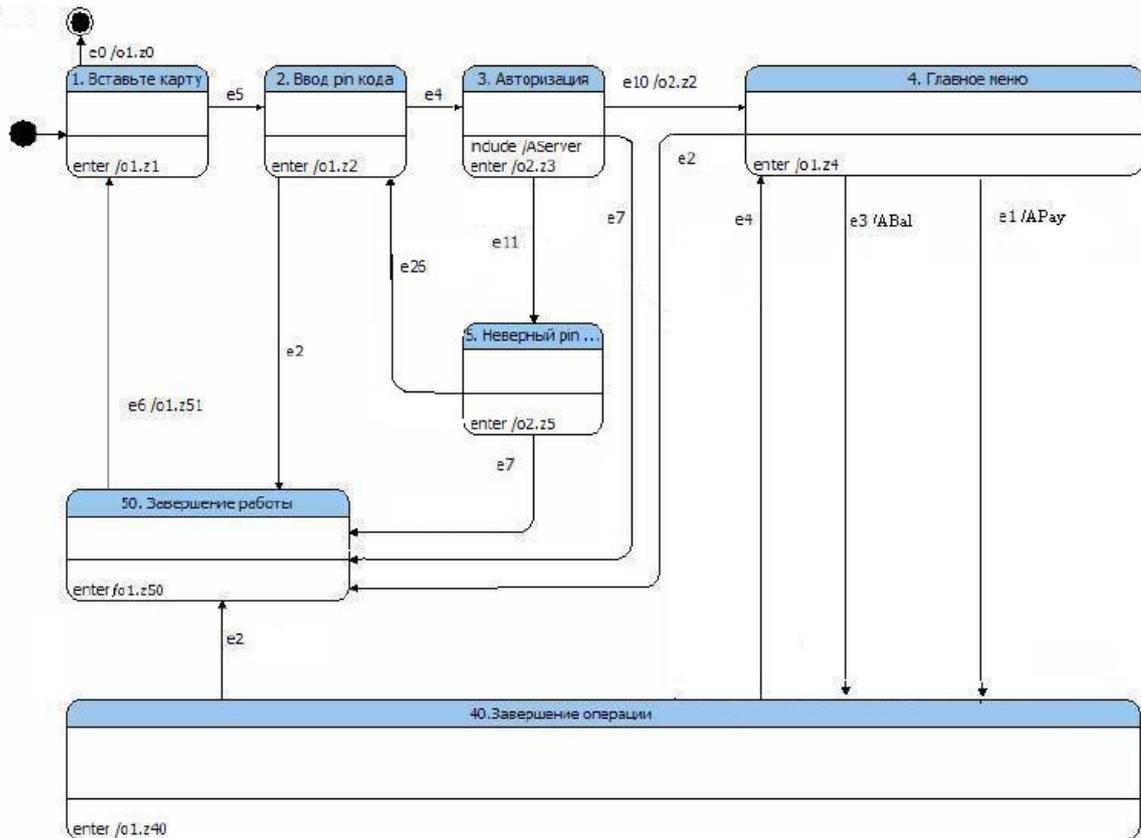


Рис. 17. Граф переходов автомата, отвечающего за работу банкомата

Полученный граф переходов удовлетворяет всем поставленным требованиям. Стоит отметить, что при модификации программы применялись только рефакторинги и базовые изменения, что позволило избежать анализа графа переходов после каждого проведенного изменения.

Выводы по главе 3

1. Введено понятие рефакторинга для автоматных программ – такой модификации программ, при которой ее поведение сохраняется.
2. Выделен набор рефакторингов автоматных программ, для каждого рефакторинга дано описание и доказано, что выполнение рефакторинга не изменяет поведение программы.
3. Предложен метод произведения изменений в автоматных программах, уменьшающий число модификаций, которые могут привести к появлению ошибок.

ГЛАВА 4. ИНТЕГРАЦИЯ СПЕЦИФИКАЦИИ И КОДА АВТОМАТНЫХ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

В первой главе были отмечены основные черты современных процессов разработки программного обеспечения: возможность регулярного изменения требований, комбинирование тестов и формальных спецификаций, единство представления кода программы и спецификации, и автоматический рефакторинг кода.

Контрактное автоматное программирование и динамическая верификация позволяют специалистам в объектно-ориентированном программировании использовать имеющиеся навыки для повышения качества автоматных компонентов со сложным поведением. Вместо составления нетривиальных темпоральных формул, возможно задать контракты для определенных частей автомата. Динамическая верификация позволяет использовать навыки модульного тестирования для повышения качества автоматных компонент в тех областях, в которых верификация неприменима. Проблема частого изменения требований была рассмотрена в третьей главе, в которой предложен метод рефакторинга автоматных программ. В этой главе рассматривается интеграция спецификации и кода автоматных объектно-ориентированных программ.

Как было отмечено выше, при традиционной разработке программного обеспечения интеграция кода программы и спецификации достигается путем представления спецификации в виде программного кода либо путем расширения языка программирования. Первый подход используется в модульном тестировании: тесты фактически являются спецификацией программы, формализованной в виде исполняемого сценария. Примером использования второго подхода является контрактное программирование. В

языке *Eiffel* [3], в котором использование контрактного программирования наиболее распространено, конструкции для описания контрактов являются частью языка.

Такая интеграция позволяет работать в условиях изменяющихся требований и кода: спецификация и код однородны, автоматические инструменты работы с кодом также изменяют спецификацию, проверка соответствия программы спецификации может быть запущена непосредственно из среды разработки.

Для получения той же степени интеграции при разработке автоматных программ требуется представить код автоматной программы и его спецификацию в виде однородном с кодом основной программы. Как было отмечено выше, этого можно достичь, либо описав спецификацию на используемом языке программирования либо расширив язык программирования специальными конструкциями для описания спецификации. Автор предлагает использовать оба подхода.

Как отмечалось выше, одним из компонентов динамической верификации является запуск верифицируемой программы на наборе тестовых сценариев. Эти сценарии, строго говоря, не являются частью спецификации, так как спецификация должна выполняться на любом сценарии. Однако они являются важным компонентом верификации и по своей структуре аналогичны модульным тестам, за исключением того что проверки не являются их частью. Соответственно, для реализации сценариев динамической верификации возможно использование основного языка программирования, применяемого в программе. При использовании подхода к организации взаимодействия автоматного и объектно-ориентированного кода «автоматизированные объекты управления как классы», автоматный компонент является обычным классом программы и может тестироваться по тем же правилам, что и остальная программа. Это позволяет использовать системы запуска модульных тестов для динамической верификации и

внедрить этот тип верификации модулей со сложным поведением в стандартные процедуры проверки качества объектно-ориентированных программ.

Так как автоматизированные объекты являются для клиентов обычными объектами программы, возможно применение к ним и традиционных в объектно-ориентированном программировании методов обеспечения качества. Если фрагмент спецификации описывается в терминах объектного интерфейса автомата, то он может быть формализован в виде обычного модульного теста для автоматизированного объекта.

Использование второго подхода к интеграции кода программы и спецификации – адаптации языка программирования для поддержки конструкций описания спецификации, сопряжено с рядом трудностей. Основные трудности связаны с текстовой природой современных языков программирования. Обычно текст программы представлен в виде строки символов, которая преобразуется компилятором в структуру программы путем синтаксического анализа текста [13]. Расширение текстовых языков может происходить путем расширения синтаксического анализатора либо предварительным преобразованием текста на расширенном языке в текст на исходном языке программирования. Расширение синтаксического анализатора современного языка программирования является сложной задачей: в большинстве случаев код анализатора не является расширяемым, а поддержка расширенного компилятора трудоемка. В некоторых современных языках программирования, например *Boo* [62] и *Nermerle* [63], предусмотрены инструкции для добавления новых конструкций в язык, однако такие языки не получили достаточного распространения. В языке *C#*, начиная с версии 3.0, существует возможность работы с деревом разбора выражения в коде программы, что позволяет определенным образом расширять семантику языковых конструкций. Однако на настоящий момент область действия этих возможностей ограничена выражениями, и поэтому не

позволяет реализовать полноценное расширение языка для работы с автоматами. Перспективным является применение динамических языков программирования, использование которых для описания автоматных программ рассмотрено в следующем разделе.

Основной проблемой использования второго подхода, а также расширения синтаксических анализаторов является неоднозначность синтаксиса при комбинировании нескольких таких расширений, например расширения для поддержки автоматного программирования и расширения для написания темпоральных спецификаций. Нет гарантии, что даже набор ключевых слов в этих расширениях не будет пересекаться, не говоря уже и о более сложных неоднозначностях. Например, один и тот же символ разными расширениями может трактоваться как оператор и модификатор.

4.1. Автоматное программирование в динамических языках

В последнее время становятся все более популярными динамические объектно-ориентированные языки программирования. Одной из их особенностей является возможность отправить любому объекту в программе произвольное сообщение (в то время как в статических языках набор возможных сообщений определяется *абстрактным типом данных* объекта [26]). Другой особенностью динамических языков является возможность заменять реализации методов, определяющих реакцию на сообщения, во время работы программы. Эти особенности в совокупности со спецификой синтаксиса некоторых динамических языков программирования позволяют описывать автоматную часть программ в терминах состояний и переходов между ними. Таким образом, возможно разработать автоматный *внутренний предметно-ориентированный язык (Domain-Specific Language, DSL)* [36]. Одной из первых реализаций такого *DSL* стала разработанная автором библиотека *STROBE* для языка программирования *Ruby*, позволяющая

описывать автоматы, спроектированные на основе автоматного программирования.

Для программиста эта библиотека предоставляет дополнительный набор инструкций (реализованных в виде методов), позволяющих поэлементно определять графы переходов (в число инструкций входят, например, `state` для определения состояния и `transition` для определения перехода). Методы используют технологию именованных параметров (при вызове инструкции значение каждого параметра связывается с параметром явно по имени), что позволяет увеличить читаемость кода.

Библиотека позволяет перенести в код на языке *Ruby* любую синтаксически верный граф переходов. Также имеется возможность перенести несколько автоматов и связать их. Возможна также интеграция с модулями на других языках программирования, в том числе модулями управления физическими объектами. При этом описание автомата изоморфно графу переходов и понятно без дополнительных инструкций и описаний.

При использовании *STROBE* для каждого автомата (в данном разделе понятие автомат не включает в себя конкретное вычислительное и управляющее состояние, для обозначения автомата в совокупности со своим состоянием используется понятие экземпляр автомата) создается класс. Это позволяет естественным образом применять возможности языка *Ruby*, такие как наследование и повторное использование классов. Это также позволяет применять стандартный механизм разрешения ссылок при использовании одного автомата из другого – ссылки между автоматами это просто ссылки между классами.

Для того, чтобы объявить автомат, требуется создать класс, унаследованный от класса `Strobe::Automaton`:

```

require `strobe/automaton'
class A0 < Strobe::Automaton
end

```

Далее в теле класса через вызовы методов библиотеки описывается структура автомата.

4.1.1. Определение общих свойств автомата

В начале декларации класса определяются такие свойства автомата как его внутренние переменные, входные и выходные воздействия, а также ссылки на номера состояний других автоматов.

Входные переменные определяются как обычные свойства класса с помощью конструкции `variables`. Например:

```

variables :x1, :x2, :x3

```

События и выходные воздействия описываются путем вызовов специальных инструкций библиотеки `inputs` и `outputs`, например:

```

inputs :e1, :e2
outputs :z20, :z21

```

Каждый автомат по умолчанию имеет стандартное входное воздействие `e0`: это воздействие используется при вызове автомата в качестве вложенного, если на графе не было явно указано воздействие, с которым необходимо производить вызов.

Согласно стандартной нотации графов переходов, в условиях переходов могут фигурировать текущие номера состояний других автоматов (через переменные y_i , соответствующие номерам текущих состояний автоматов A_i). Так как в одной программе могут работать несколько автоматных программ, и допускается повторное использование автоматов, такие связи (имен переменных и автоматов) указываются явно с помощью инструкции `alias_state`, например:

```
alias_state :y1 => A1, :y2 => A2
```

Для разрешения ссылок на другие автоматы используется стандартный механизм разрешения ссылок языка *Ruby* в сочетании с концепцией доменов. Домен – множество экземпляров автоматов, которые могут ссылаться друг на друга по именам. Внутри одного домена может быть только один экземпляр автомата с данным именем. По умолчанию именем автомата является полностью квалифицированное имя его класса. Таким образом, для того чтобы в одном домене иметь два экземпляра одного класса автоматов, необходимо вручную назначить им имена. Это можно сделать при создании экземпляра автомата:

```
a1 = A1.new(:a1)
```

или

```
domain = AutomataDomain.new  
a1 = A1.new(:a1, domain).
```

В первом случае создается экземпляр автомата *A1* с именем `:a1` (имя по умолчанию – полностью квалифицированное имя класса *A1*). Во втором – такой же автомат создается в домене, определяемым значением переменной `domain`.

Таким образом, инструкция `alias_state` связывает переменную с номером состояния автомата с указанным именем, находящегося в том же домене, что и ссылающийся автомат. Если в качестве имени автомата указано имя класса, то ищется автомат с именем, являющимся полностью квалифицированным именем этого класса – при использовании имен по умолчанию для ссылок на автоматы можно использовать стандартный механизм разрешения ссылок языка *Ruby*.

4.1.2. Описание графов переходов

Графы переходов переносятся в код последовательно состояние за состоянием. Разумнее всего после декларации состояния декларировать все переходы, выходящие из него. Состояние декларируется с помощью инструкции `state`, например, следующая конструкция декларирует состояние, вызывающее вложенный автомат A1 с событием e9 и выходным воздействием z701:

```
state :loaded_doors_closed_elevator_up,
      :subautomatons => [[A1, 9]],
      :output_actions => :z701
```

Каждое состояние имеет идентификатор для ссылок в декларациях переходов, а также номер, который либо присваивается автоматически (последовательно начиная с нуля) либо явно с помощью параметра `:number`. Этот номер должен соответствовать номеру состояния на графе – он будет использоваться для проверки номера состояния в условиях переходов.

Начальным состоянием по умолчанию считается нулевое состояние, но его также можно указать явно при помощи инструкции `initial_state`, передав ей в качестве параметра идентификатор начального состояния.

Для описания переходов между состояниями предусмотрено несколько инструкций. Для описания перехода из одного состояния в другое введена инструкция `transition`, которой в качестве параметров передаются начальное и конечное состояния, условие перехода, его приоритет и действия на переходе. Например:

```
transition :from => :on,
          :to => :off,
          :if => lambda { e3 && x61 && y1 == 2 },
          :priority => 3
```

Одним из ключевых достоинств использования *STROBE* является то, что условия переходов переносятся один-в-один с графа (с поправкой на синтаксис булевых операторов в *Ruby*). Это позволяет избежать значительной доли ошибок при ручном переносе графов в текст программы.

При декларации перехода начальное состояние можно не указывать. В этом случае начальным считается последнее декларированное состояние.

Для описания циклов предусмотрена специальная инструкция `loop` с теми же параметрами, что и у `transition`, кроме `:from` и `:to` – она декларирует петлю с началом и концом в последнем декларированном состоянии.

В *SWITCH*-технологии [1] также предусмотрена такая конструкция как группы состояний. Для описания групп и групповых переходов введены специальные инструкции `begin_group`, `end_group` и `group_transition`. Первые две определяют начало и конец декларации группы – все состояния, задекларированные между этими инструкциями, попадут в общую группу. Группы, в соответствии со *SWITCH*-технологией, могут быть вложенными.

Для декларации групповых переходов используется инструкция `group_transition`, которая имеет те же параметры, что и `transition`, но в качестве начала перехода использует текущую группу.

4.1.3. Пример декларации автомата

Для примера рассмотрим автомат «Панель в кабине лифта» из работы [64]. Он имеет граф переходов, изображенный на рис. 18.

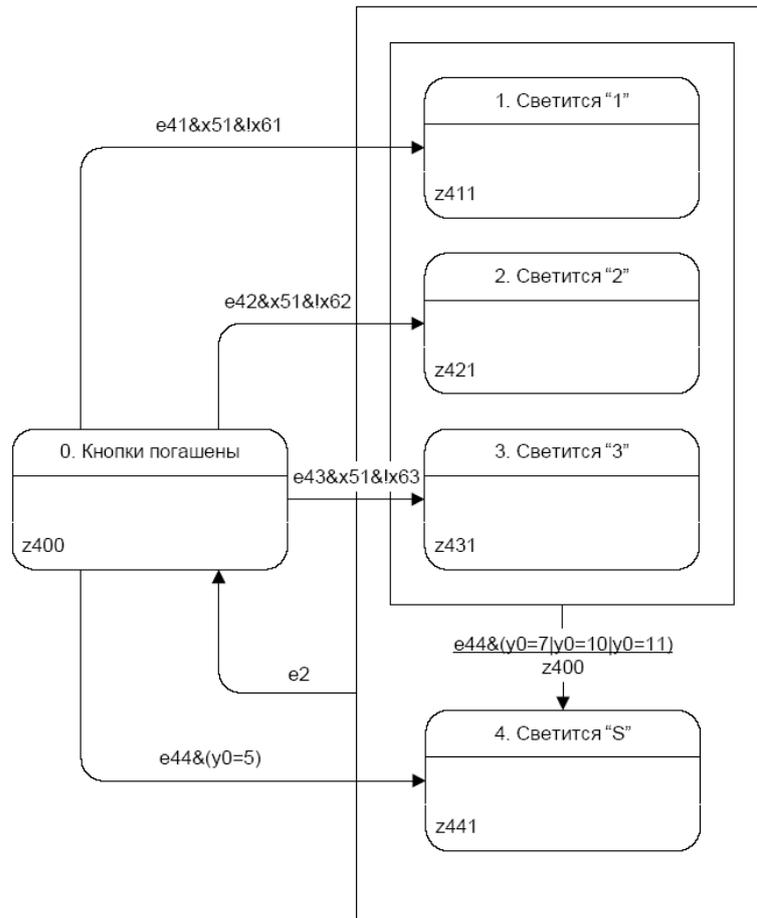


Рис. 18. Граф переходов автомата «Панель в кабине лифта»

Следующий пример кода на языке *Ruby* декларирует этот автомат, используя библиотеку *STROBE*:

```

# Подключение библиотеки STROBE
require 'strobe/automaton'

# Декларация модуля Elevator
module Elevator
  # Декларация класса A0 для ссылок на него
  class A0 < Strobe::Automaton
  end

  # Декларация класса автоматов A2

```

```

class A2 < Strobe::Automaton
  # Декларация внутренних переменных
  variables :x51, :x61, :x62, :x63

  # Использование переменной y0 как номера текущего
  # состояния автомата A0
  alias_state :y0 => A0

  # Декларация входных воздействий
  inputs :e2, :e41, :e42, :e43, :e44

  # Декларация выходных воздействий
  outputs :z400, :z411, :z421, :z431, :z441

  # Состояние "Кнопки погашены"
  state :buttons_off,
    # Действие по входу в состояние - z400
    :output_actions => :z400

  # Переход в состояние "Светится 1"
  transition :to => :button_1,
    :if => lambda { e41 && x51 && !x61 }

  # Переход в состояние "Светится 2"
  transition :to => :button_2,
    :if => lambda { e42 && x51 && !x62 }

  # Переход в состояние "Светится 3"
  transition :to => :button_3,

```

```
        :if => lambda { e43 && x51 && !x63 }

# Переход в состояние "Светится S"
transition :to => :button_s,
            :if => lambda { e44 && (y0 == 5) }

# Начало группы, включающей в себя все состояния
# кроме "Кнопки погашены"
begin_group

# Начало группы, включающей в себя состояния
# свечения кнопок этажей
begin_group

# Состояние "Светится 1"
state :button_1,
      # Действие по входу в состояние - z411
      :output_actions => :z411

# Состояние "Светится 2"
state :button_2,
      # Действие по входу в состояние - z421
      :output_actions => :z421

# Состояние "Светится 3"
state :button_3,
      # Действие по входу в состояние - z431
      :output_actions => :z431
```

```

# Групповой переход в состояние "Светится S"
group_transition :to => :button_s,
                  :if      =>          lambda
{ e44 && (y0 == 7 || y0 == 10 || y0 == 11) },
                  # Действие по переходу - z400
                  :output_actions => :z400
end_group # Конец группы

# Состояние "Светится S"
state :button_s,
        # Действие по входу в состояние - z441
        :output_actions => :z441

# Групповой переход в состояние "Кнопки
# погашены"
group_transition :to => :buttons_off,
                  :if => lambda { e2 }
end_group # Конец группы
end
end

```

4.1.4. Использование библиотеки

Автоматы, описанные с помощью *STROBE*, можно использовать из кода на языке *Ruby* путем создания экземпляров классов автоматов:

```
a = A1.new
```

Вызов автомата производится с помощью метода `call`. Например,
ВЫЗОВ

```
a.call(:e0)
```

вызывает автомат с событием `e0`.

Библиотека также предоставляет возможность получения текущего состояния автомата через методы `y` и `current_state`, возвращающие номер и идентификатор текущего состояния соответственно.

Реакция на выходные воздействия указывается путем добавления внешних обработчиков, которым передается управление при вызове соответствующего выходного воздействия, например:

```
a.on_output :z1 do
  p "вызвано выходное воздействие z1"
end
```

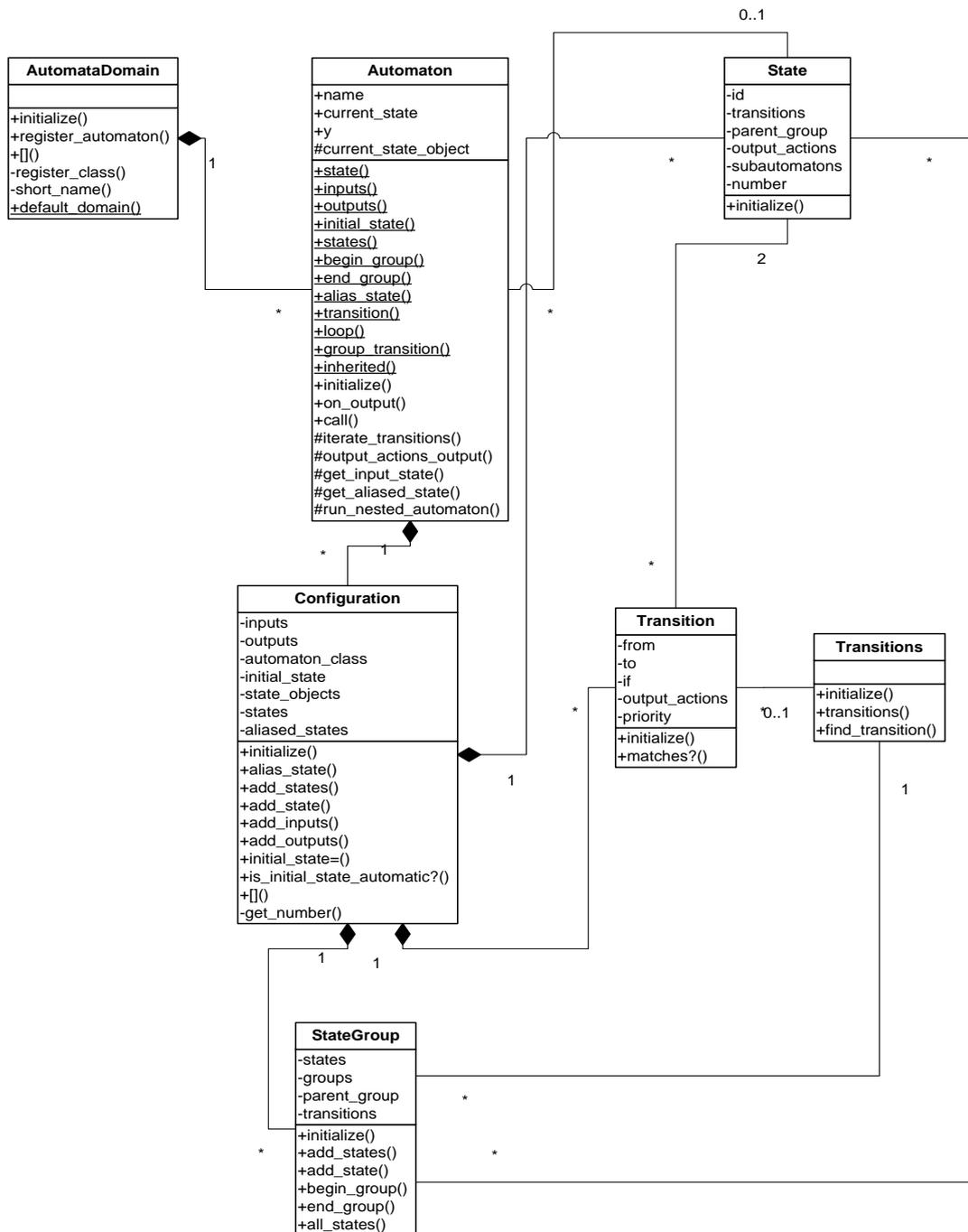
4.1.5. Реализация библиотеки

Библиотека *STROBE* реализована в виде набора классов на языке *Ruby*. Назначение каждого из них следующее:

- `Automaton` – соответствует классу автоматов и является базовым классом для конкретных классов автоматов. Этот класс содержит основные методы для работы с автоматами, а также статические методы, реализующие инструкции декларации автоматов. Экземпляр класса, являющийся потомком класса `Automaton`, является экземпляром автомата – его конфигурация, в частности, включает в себя текущее состояние и значения внутренних переменных;
- `Configuration` – класс, соответствующий конфигурации автомата – фактически, его графу переходов. Экземпляр этого класса соответствует одному из наследников класса `Automaton`;
- `AutomataDomain` – класс, соответствующий домену. Поддерживает регистрацию автоматов в домене и поиск автомата по имени;
- `State` – класс, соответствующий состоянию автомата;
- `StateGroup` – класс, соответствующий группе состояний;

- Transition – класс, соответствующий переходу (в том числе групповому).

Связи основных классов показаны на графе, изображенном на рис. 19:

Рис. 19. Схема связей классов составляющих реализацию библиотеки *STROBE*

Непосредственно программист взаимодействует только с классами `Automaton` и `AutomataDomain`. Класс `Automaton` содержит ряд статических методов, реализующих инструкции описания графов переходов:

- `state` – вводит новое состояние в граф переходов;
- `inputs` – описывает входные воздействия автомата;
- `outputs` – описывает выходные воздействия автомата;
- `initial_state` – определяет начальное состояние автомата;
- `states` – вводит несколько состояний в граф переходов;
- `begin_group` – открывает область введения группы состояний;
- `end_group` – закрывает область введения группы состояний;
- `alias_state` – определяет псевдоним для номера состояний другого автомата;
- `transition` – вводит переход в граф переходов;
- `loop` – вводит петлю в граф переходов;
- `group_transition` – вводит групповой переход в граф переходов.

Названия методов совпадают с названиями операций, описанных выше.

Фактически, интерпретатор языка *Ruby* рассматривает код внутри декларации класса как исполняемый, так что инструкции описания графа переходов являются вызовами методов класса `Automaton`. Эти методы доступны внутри декларации класса благодаря тому, что автоматные классы наследуются от класса `Automaton`.

Так как все экземпляры одного автоматного класса имеют общий граф переходов, она является свойством самого класса, а не конкретного его экземпляра. Для инкапсуляции графа переходов был введен класс `Configuration`. Экземпляр класса `Configuration` создается для каждого класса, унаследованного от класса `Automaton`. Это достигается благодаря возможности обработки события наследования класса в методе

`Automaton::inherited`, который вызывается средой выполнения при наследовании класса, в котором этот метод был определен.

При вызове методов класса `Automaton` происходит разбор параметров, и вызовы делегируются классу `Configuration`, который соответствующим образом изменяет модель графа переходов. Для повышения читаемости кода методы класса `Automaton` используют схему, эмулирующую именованные параметры. По сигнатуре все эти методы принимают на вход экземпляр класса `Hash` – хеш-таблицу, в которой хранятся пары (имя параметра, значение параметра). Синтаксис языка *Ruby* позволяет создавать экземпляры `Hash`, просто указывая пары ключ-значение, разделяя ключ от значения символами “=>”, а соседние пары – запятыми. Использование этой схемы позволяет понять смысл передаваемых значений человеку, не знакомому с библиотекой и даже с программированием на языке *Ruby*.

Конфигурация автомата определяется набором его входных и выходных воздействий, набором внутренних переменных, ссылок на другие автоматы и, собственно, графом переходов. Для описания входных и выходных воздействий используются методы `inputs` и `outputs`, которым в качестве параметров передаются имена входных и выходных воздействий автомата соответственно.

Модель графа переходов, поддерживаемая классом `Configuration`, реализована в следующих классах:

- `State` – состояние;
- `Transition` – переход (в том числе групповой);
- `StateGroup` – группа состояний;
- `Transitions` – набор переходов из одного состояния или группы состояний.

При введении нового состояния в граф переходов создается экземпляр класса `State`, который инициализируется идентификатором, номером (либо заданным вручную, либо вычисленным как $M + 1$, где M – максимальный номер состояния уже присутствующего на графе).

Для работы с группами состояний класс `Automaton` поддерживает «текущую группу состояний» – группу, в которой будут регистрироваться состояния, вводимые в граф в данный момент. Значение текущей группы изменяется методами `begin_group` и `end_group`. Первый создает новую группу состояний и делает её текущей, при этом предыдущая текущая группа устанавливается новой в качестве родительской группы. Метод `end_group` «закрывает» текущую группу, возвращая ту текущую группу, которая была установлена на момент соответствующего вызова `begin_group`.

При введении нового перехода (включая групповой) создается экземпляр класса `Transition`, инициализируемый начальным и конечным состояниями (в роли начального состояния может выступать группа состояний), условием перехода и действиями на переходе. Этот экземпляр регистрируется в экземпляре класса `Transitions` исходного состояния или группы состояний.

Класс `Transitions` позволяет регистрировать набор переходов из одного состояния или группы состояний и быстро выполнять поиск перехода с максимальным приоритетом, условие перехода которого выполняется. Для этого в классе поддерживается массив переходов, отсортированный по возрастанию приоритета, инициализация которого происходит в ленивом режиме (*lazy initialization*) – при первом обращении.

Условия переходов задаются в виде замыканий, которые являются экземплярами класса `Proc` в *Ruby*. При проверке условия перехода (это осуществляется в методе `matches?` класса `Transition`) производится выполнение кода замыкания в контексте экземпляра автоматного класса, для

которого необходимо проверить условие. При этом в теле замыкания становятся непосредственно доступны все методы этого экземпляра. При создании экземпляра автоматного класса производится динамическое порождение следующих интерфейсных методов:

- для каждого входного воздействия порождается метод с именем входного воздействия, возвращающий `true`, если это воздействие сейчас обрабатывается и `false` в других случаях;
- для каждой внутренней переменной порождаются методы чтения и записи этой переменной;
- для каждой ссылки на другой автомат порождается метод с именем ссылки, возвращающий текущий номер состояния соответствующего автомата.

Эти методы в числе прочих доступны в коде условий переходов, что позволяет переносить их с графа практически без изменений.

Для взаимодействия с другим кодом используется подписка на выходные воздействия и метод `call`, позволяющий вызвать автомат с указанным входным воздействием. Метод `call` выполняется в соответствии с семантикой, описанной выше.

Важным элементом архитектуры библиотеки является использование общего полиморфизма [65] (функций, полиморфных на потенциально бесконечном множестве типов, имеющих заданную структуру). Это позволяет избавиться от явного выделения интерфейсов в классах `State` и `StateGroup`. Ссылка из класса `Transition` на объект, являющийся началом перехода, является, как и все ссылки в динамических языках, нетипизированной, что позволяет в качестве начала перехода передавать экземпляры как класса `State`, так и класса `StateGroup`. При использовании объекта, доступного по этой ссылке вызываются только два метода: `transitions` и `parent_group`, реализованные в обоих классах.

Общий полиморфизм позволяет избежать введения специального интерфейсного типа.

Другим используемым свойством языка *Ruby* являются итераторы: метод `iterate_transitions` класса `Automaton` при серии вызовов в цикле последовательно возвращает переходы сначала из текущего состояния, затем из содержащей его группы и т. д.

Наконец, очень удобной является динамическая программа типов, позволяющая не указывать конкретные типы внутренних переменных (это соответствует семантике графов переходов, на которых конкретные типы также не указываются) и избежать приведения типов в условиях переходов.

4.1.6. Отладка программ

Среда выполнения *Ruby* поддерживает интерактивную отладку программ в диалоговом режиме. Отладчик среды выполнения языка *Ruby* поддерживает все основные команды, такие как установка точек останова, проверка значений переменных и переход внутрь вызываемого метода.

4.1.7. Взаимодействие с окружением

Важным преимуществом использования той или иной среды для автоматного программирования является возможность интегрироваться с кодом, написанным с использованием других языков и технологий. Это необходимо для работы с реальными объектами управления и обеспечения взаимодействия с библиотеками сторонних производителей.

Среда исполнения *Ruby* имеет возможность интеграции с кодом, написанным на *C* или другом языке программирования, для которого существует компилятор, позволяющий строить динамически подключаемые модули (например, *.dll* для *Windows*, *.so* для *Linux*). Для этого используется механизм расширений (*extensions*). Эти расширения являются модулями, которые при загрузке вводят дополнительные классы в среду исполнения *Ruby*. При этом реализация классов производится в расширении – на другом

языке программирования. Вот пример такого расширения, реализующего объект управления «Лампа»:

```
#include <stdio.h>

// Заголовок среды исполнения Ruby
#include "ruby.h"

// Тело метода Lamp.turn_on
static VALUE t_turn_on(VALUE self)
{
    printf("Lamp turns on\n");
    return self;
}

// Тело метода Lamp.turn_off
static VALUE t_turn_off(VALUE self)
{
    printf("Lamp turns off\n");
    return self;
}

// Идентификатор класса cLamp
VALUE cLamp;

// Функция инициализации модуля
void Init_lamp()
{
    // Объявление класса Lamp
    cLamp = rb_define_class("Lamp", rb_cObject);
}
```

```

// Объявление метода turn_on
rb_define_method(cLamp, "turn_on", t_turn_on, 0);

// Объявление метода turn_off
rb_define_method(cLamp, "turn_off", t_turn_off, 0);
}

```

Это расширение при загрузке объявляет в среде выполнения *Ruby* класс `Lamp` с двумя методами `turn_on` и `turn_off`. Для демонстрационных целей эти методы лишь выводят в консоль сообщения подтверждающие факт вызова. В реальном приложении они могут управлять настоящим реле.

Вот пример кода, в котором автомат интегрирован с этим мнимым объектом управления:

```

# Подключение модуля объекта управления
require 'integration/lamp.dll'
# Подключение библиотеки STROBE
require 'strobe/automaton'

// Автомат A0
class A0 < Strobe::Automaton
  // Выходные воздействия
  // z1 - "Включить лампу"
  // z2 - "Выключить лампу"
  outputs :z1, :z2

  // Состояние "Лампа выключена"
  state :off

```

```
// Переход в состояние "Лампа включена"  
transition :to => :on,  
            :output_actions => :z1  
  
// Состояние "Лампа включена"  
state :on  
  
// Переход в состояние "Лампа выключена"  
transition :to => :off,  
            :output_actions => :z2  
  
end  
  
// Создание экземпляра класса Lamp, управляющего  
"Лампой"  
lamp = Lamp.new  
// Создание экземпляра автомата A0  
a = A0.new  
  
// Связывание выходных воздействий с функциями  
управления "Лампой"  
a.on_output :z1 do  
  lamp.turn_on  
end  
  
a.on_output :z2 do  
  lamp.turn_off  
end
```

```
// Два вызова автомата для проверки переключения в обе
стороны
a.call(:e0)
a.call(:e0)
```

4.1.8. Формализация спецификаций автоматов

Возможности языка программирования *Ruby* технически позволяют добавить поддержку описания формализованных спецификаций. Формулы темпоральной логики и охранные условия контрактов могут быть описаны в виде выражений языка. Специальные операторы для декларации контрактов и спецификаций в виде формул темпоральной логики могут быть реализованы аналогично конструкциям описания автоматов.

Оборотной стороной свойств динамических языков программирования является сложность статического анализа кода. Так как тип объектов, участвующих в выражениях, не определен до момента вычисления этих выражений, проанализировать их семантику статически сложно. Это ограничение приводит к тому, что для автоматных программ, реализованных с помощью приведенного подхода, эффективно возможно использовать только динамические методы верификации.

4.2. Интеграция спецификации и кода в мультязыковых средах

Как было отмечено в первой главе, существуют подходы к объектно-ориентированному программированию с явным выделением состояний на основе мультязыковых сред таких, как *JetBrains MPS*. Для этих целей разработаны специальные языки автоматного программирования, например *stateMachine* [33]. Ниже дается описание системы *JetBrains MPS* и используемого в ней механизма разработки и композиции языков.

4.2.1. Описание языков в системе *JetBrains MPS*

Как было отмечено во введении к настоящей главе, одной из основных проблем расширения текстовых языков программирования является

неоднозначность при синтаксическом анализе кода, написанного и с использованием таких расширений. Однако, эта проблема затрагивает лишь нижний слой процесса синтаксического анализа. Известно, что структура программы может быть представлена в виде абстрактного и конкретного синтаксического дерева [13].

Абстрактное синтаксическое дерево (*AST*) представляет логическую структуру программы: иерархию концепций языка, их атрибуты и связи между ними. Конкретное синтаксическое дерево определяет структуру разбора текста программы. Его узлами являются терминалы и нетерминалы формальной грамматики языка. Программа однозначно представляется в виде абстрактного синтаксического дерева, но в то же время может иметь несколько альтернативных представлений в виде конкретного синтаксического дерева. Например, выражение может быть представлено как $x = y + z$ либо как $(= (x, (+, (y, z))))$.

Таким образом, проблема неоднозначности возникает на этапе получения конкретного синтаксического дерева: при комбинации нескольких расширений языка может оказаться, что одно и то же текстовое представление соответствует различным синтаксическим деревьям этих расширений. Логичным решением этой проблемы является подход, при котором программа представлена непосредственно в виде абстрактного синтаксического дерева. Этот подход был впервые предложен в работе [66]. В этом случае различные расширения предоставляют наборы дополнительных возможных узлов абстрактного синтаксического дерева, и выбор между ними происходит явно, а не посредством неоднозначной трансляции.

Именно этот подход лежит в основе системы *MPS*. В этой системе программа представлена в виде дерева, узлы которого являются концептами различных языков программирования. Процесс изменения этого дерева схож с традиционным редактированием текста программы.

Описание языка в *MPS* состоит из определений ряда *аспектов*. Наиболее важными из них являются структура языка, редактор, генератор и система типов. Структура языка определяет набор концептов – видов узлов синтаксического дерева, предоставляемых языком. Редакторы концептов описывают поведение их экземпляров в редакторе кода программы. Генератор задает семантику языка, описывая преобразования, которые необходимо применить к узлам данного языка для генерации исполняемого кода. Система типов является необязательным компонентом, но в сложных языках она играет важную роль в процессах редактирования и генерации [67].

Как было отмечено, структура языка в *MPS* задается набором концептов – типов узлов абстрактного синтаксического дерева. Каждый узел дерева в *MPS* является экземпляром какого-либо концепта. Концепты могут наследовать друг друга и образуют иерархию. Например, концепт «операция +» может наследовать концепт «бинарная операция», который в свою очередь может наследовать концепт «выражение». Множественное наследование не поддерживается.

Описание концепта содержит описание детей, ссылок и свойств. *Дети концепта* определяют то, какие узлы могут находиться в поддереве экземпляра этого концепта. Например, бинарная операция в качестве детей имеет два выражения, являющиеся ее левым и правым операндами. Описание ребенка состоит из *роли*, концепта ребенка и *кардинальности связи*. Кардинальность определяет число детей с этой ролью, которое может быть у экземпляра данного концепта. Кардинальность может принимать значения $0..1$, 1 , $0..n$ и $1..n$. Для бинарной операции набор детей будет включать две связи с ролями «левый операнд» и «правый операнд», концептом «выражение» и кардинальностью 1. Концепт «метод» имеет ребенка «тело метода» с концептом «предложение» и кардинальностью 1, и ребенка «параметр» с концептом «параметр» и кардинальностью $0..n$. Ссылки

концепта определяют набор ссылок вне собственного поддерева, которые может иметь экземпляр данного концепта. Описание ссылки аналогично описанию ребенка, за исключением того, что кардинальность может принимать только значения 0..1 и 1. Свойства концепта это описания строковых, булевских и числовых атрибутов экземпляров этого концепта. Описание свойства состоит из имени и типа его значения. Примером свойства является имя переменной у концепта «объявление переменной». Кроме объявления детей, связей и свойств в концепте определяется ряд менее важных свойств, описанных в документации к *MPS* [68].

Концепты, которые могут являться типами корней синтаксического дерева, являются корневыми. Корневой концепт соответствует файлу в традиционных языках программирования: он целиком редактируется в одном окне редактора. Несколько корневых концептов образуют модель – единицу модуляризации программы на *MPS*. Для каждой модели определяется набор используемых языков и импортированных моделей. Узел в модели может быть экземпляром только тех концептов, которые определены в используемых языках. Узлы могут ссылаться только на узлы из собственной модели, а также из импортированных моделей.

Выше было отмечено, что редактор абстрактного синтаксического дерева в *MPS* реализован таким образом, чтобы воспроизвести основные особенности процесса редактирования текстовых программ, привычного для программистов. Для этого в основу редактора была положена клеточная структура. Все редактируемое дерево представлено в виде набора вложенных клеток: корневому узлу соответствует клетка, занимающая все поле редактора. Дети узлов представлены клетками внутри клетки родительского узла. Клетки могут быть конечными и составными. Клетки второго типа представляют собой набор вложенных клеток, расположенных на экране в соответствии с выбранной политикой: вертикально, горизонтально или горизонтально с переносом на следующую строку. Атомарные клетки могут

быть константными, содержимое которых задано, и редактируемыми, содержимое которых может изменяться пользователем. Редактируемые клетки могут, например, использоваться для отображения и редактирования атрибутов узла. Кроме того, клетки могут реагировать на нажатие определенных клавиш и изменять соответствующим образом дерево.

Описание редактора концепта определяет набор клеток, которыми узлы этого концепта будут представлены в редакторе. Процесс создания редактора реализует принцип *WYSIWYG* (*What You See Is What You Get*) – позволяет оценивать результат непосредственно при редактировании.

Помимо редактирования значений свойств, редактор должен обеспечивать удобство создания новых экземпляров концептов. За это отвечает меню подстановки. При нажатии комбинации клавиш *Control+Space* активизируется меню, с помощью которого можно выбрать концепт создаваемого узла. По умолчанию список содержит все доступные конкретные концепты, расширяющие требуемый, но это поведение возможно изменить. Другой важной возможностью является механизм левых и правых трансформаций, позволяющих преобразовывать дерево при нажатии определенных клавиш. Например, при нажатии после числовой константы клавиши «+», она будет преобразована в узел операции сложения.

Генератор языка содержит набор правил, преобразующих синтаксическое дерево программы на одном языке в дерево на другом языке, для которого уже задана семантика. Обычно таким языком является *Java*, для которого используется стандартный компилятор и среда исполнения. Если один язык расширяет другой, то его генератор может преобразовывать программу в дерево на расширяемом языке. После этого стандартный генератор преобразует его в код на языке *Java*.

4.2.2. Автоматное программирование в среде *JetBrains MPS*

Для автоматного программирования в среде *JetBrains MPS* был разработан текстовый язык автоматного программирования *stateMachine* [33]. Этот язык предназначен для разработки автоматизированных объектов на *Java* и *JavaScript*. Он позволяет добавлять к произвольному *Java*-классу в проекте автоматный концепт, определяющий структуру автомата и набор событий, с которыми он может вызываться. Язык *stateMachine* поддерживает вложенные состояния, в качестве охранных условий и выходных воздействий могут использоваться произвольные конструкции языка *Java* и *JavaScript*.

Пример использования языка *stateMachine* для описания автоматной части класса, управляющего пользовательским интерфейсом списка дефектов, показан на рис. 20.

```

IssueList js program(isBootstrap: false)

requires:
  charisma.smartui.js
  jquery_extension.js
  jquery.js
  Util.js

/** no comments */
class charisma.smartui.IssueList extends Webr.component.Component {
  << constants >>
  /** no comments */
  IssueListConfig config;
  /** no comments */
  HTMLDivElement content;
  /** no comments */
  Function handler;
  /** no comments */
  Function mouseHandler;
  /** no comments */
  Function shiftUpWatcher;
  /** no comments */
  Function shiftDownWatcher;
  /** no comments */
  Boolean focused = false;
  /** no comments */
  Array<IssueListItem> items = [];
  /** no comments */
  Array<IssueListItem> divToItem = {};
  /** no comments */
  Number selected;
  /** no comments */
  Boolean ch = true;
  /** no comments */
  HTMLInputElement checkedIssuesInput;

  /** no comments */
  constructor(IssueListConfig config) {

```

```

state program for IssueList.js
state machine for charisma.smartui.IssueList {
  initial state (NOT_SHIFTED) {
    exit do (this.ch = !(this.getSelected().checkbox.checked));

    on shiftDown() do {<< statements >>} transit to SHIFTED
  }

  state (SHIFTED) {
    initial state (INITIAL) {
      on kdown() do {<< statements >>} transit to MOVE_DOWN

      on kup() do {<< statements >>} transit to MOVE_UP

      on shiftUp() do {<< statements >>} transit to NOT_SHIFTED
    }

    state (MOVE_DOWN) {
      enter do {this.getSelected().setChecked(this.ch);}

      on kdown() do {this.getSelected().setChecked(this.ch);}

      on kup() do {this.ch = !this.ch;} transit to MOVE_UP

      on shiftUp() do {<< statements >>} transit to NOT_SHIFTED
    }

    state (MOVE_UP) {
      enter do {this.getSelected().setChecked(this.ch);}

      on kup() do {this.getSelected().setChecked(this.ch);}

      on kdown() do {this.ch = !this.ch;} transit to MOVE_DOWN

      on shiftUp() do {<< statements >>} transit to NOT_SHIFTED
    }
  }
}

```

Рис. 20. Класс управления списком дефектов и его автоматная часть на языке *stateMachine*

Основными конструкциями языка *stateMachine* являются описания состояния (*state*) и перехода (*on*). Состояния могут быть вложенными, одно из состояний может быть объявлено начальным с помощью модификатора *initial*. Для состояния конструкциями *enter do* и *exit do* могут быть заданы действия при входе и при выходе из состояния соответственно.

В качестве примера использования языка *stateMachine* рассмотрим описание автомата «лифт 1», использованного во второй главе. Опишем состояния автомата: *MOVING* («Движение»), *OPENING_DOORS* («Открывание дверей»), *CLOSING_DOORS* («Закрывание дверей»), *WAITING_DOORS_OPEN* («Ожидание с открытыми дверьми»), *WAITING_DOORS_CLOSED* («Ожидание с закрытыми дверьми»). На языке *stateMachine* это описание имеет следующий вид:

```
state machine for stateSpec.samples.Lift {
  state {MOVING} {}
  state {OPENING_DOORS} {}
  state {CLOSING_DOORS} {}
  state {WAITING_DOORS_OPENED} {}
  initial state {WAITING_DOORS_CLOSED} {}
}
```

Следующим шагом опишем входные воздействия, на которые может реагировать автомат, а также выходные воздействия и входные переменные. Выходные воздействия могут быть реализованы как методы класса—автоматизированного объекта. Входные переменные являются полями этого класса. Входные воздействия описываются в императивной части автомата как методы-события и объявляются с помощью ключевого слова *event*:

```
class stateSpec.samples.Lift {
  bool on_first_floor; // x11
```

```

bool on_second_floor; // x12
bool on_third_floor; // x13
bool light_on; // x4

event arrived(); // e1
event doors_open(); // e2
event doors_closed(); // e3
event called_first_floor(); // e41
event called_second_floor(); // e42
event called_third_floor(); // e43

void move_up() {} // z1
void move_down() {} // z2
void stop() {} // z3
void open_doors() {} // z4
void close_doors() {} // z5
void light_on() {} // z6
void light_off() {} // z7
}

```

Переходы автомата описываются в автоматной части класса с помощью конструкции `on`:

```

state {MOVING} {
    on arrived() do {
        stop();
        open_doors();
    } transit to DOORS_OPENING
}

```

Программирование с явным выделением состояний в *MPS* с использованием языка *stateMachine* соответствует парадигме

«автоматизированные объекты управления как классы». Класс, в котором реализован автоматный аспект, является обычным классом на языке *Java* и используется и тестируется по тем же принципам, что и неавтоматные классы программы. Поэтому другие классы программы, частью которой является лифт, могут посылать ему события следующим образом:

```
Lift lift = new Lift();  
lift.called_first_floor();
```

4.2.3. Язык спецификации автоматов

Для формализации требований к автоматным программам автором диссертации был разработан язык *stateSpec* для *JetBrains MPS*. Этот язык расширяет язык *stateMachine* и позволяет описывать спецификации для автоматов на этом языке. В качестве спецификаций могут выступать темпоральные формулы и контракты.

Автоматизированный класс в *MPS* представлен редактором с двумя вкладками. В первой вкладке редактируется *Java*-код класса, во второй – описание его автоматной части.

Разработанный язык добавляет секцию для описания спецификаций к автоматной части класса. В этой секции программист может определить набор темпоральных формул на языке *LTL*, являющихся спецификацией класса. Язык также расширяет конструкции описания элементов автомата, добавляя возможность задания контрактов на состояния. Автомат управления кофеваркой с описанной спецификацией изображен на рис. 21.

```

state machine for CoffeeMaker {
  <listeners>

  initial state {ReadyToUse} {
    on startButtonPressed[this.voltage == 220 && this.waterLevel <= 1.0] do {<no statements>} transit to {WarmingUp}

    on unPlugged do {<no statements>} transit to {TurnedOff}

    invariant {
      voltageTreshold == 220
      currentVoltage < voltageTreshold
    }

    require for ReadyToUse
      isTurnedOn
    }
  }

  state {WarmingUp} {
    on tempIsReached(temperature)[temperature >= 100] do {<no statements>} transit to {ReadyToUse}

    on errorOccuried do {<no statements>} transit to {Broken}
  }

  state {TurnedOff} {
    on pluggedIn[this.voltage == 220] do {<no statements>} transit to {ReadyToUse}
  }

  state {Broken} {
    on unPlugged do {<no statements>} transit to {TurnedOff}
  }

  specification {
    G F !(Broken)
    ! ReadyToUse R TurnedOff
    currentVoltage < voltageTreshold U Broken
  }
}

```

Рис. 21. Описание автомата управления кофеваркой на языке *stateMachine* и его спецификации на языке *stateSpec*

Конструкции языка *stateSpec* включены в описание автомата на языке *stateMachine*. Они представлены в виде отдельной секции *specification* и специальных конструкций (например, *require* и *invariant*) внутри описания автомата.

Секция *specification* используется для описания спецификации автомата в виде набора темпоральных формул. Каждая строка этой секции содержит темпоральную формулу на языке *LTL*. В задаваемых формулах разрешается использовать булевы операторы ! («не»), && («и»), || («или»), -> (импликация), темпоральные одноместные операторы N («на следующем шаге», соответствует рассмотренному выше оператору \mathbb{N}), G («всегда»,

соответствует оператору \mathbb{G}), \mathbb{F} («в будущем», соответствует оператору \mathbb{F}), и темпоральные двуместные операторы \mathbb{U} («до тех пор, как», соответствует оператору \mathbb{U}) и \mathbb{R} («освобождает», $\psi \mathbb{R} \varphi = \overline{\overline{\psi \mathbb{U} \overline{\varphi}}}$). В качестве утверждений в формулах спецификации разрешается использовать ссылки на состояния, события и выходные воздействия. Такие утверждения истинны, когда автомат находится в соответствующем состоянии, обрабатывается соответствующее событие или производится соответствующее выходное воздействие соответственно.

Помимо секции `specification` для описания контрактов используются конструкции `invariant`, `requires` и `ensures`. Эти конструкции включаются в тела описания состояний, к которым они применяются. В настоящее время язык `stateSpec` не поддерживает контракты для входных и выходных воздействий, такая поддержка будет реализована в следующей версии. Конструкция `invariant` определяет инварианты для состояния, конструкция `requires` – предусловия, `ensures` – постусловия. Инварианты, пред- и постусловия выражаются в виде булевых формул, в которых набор утверждений эквивалентен набору, используемому в темпоральных спецификациях в секции `specification`.

При использовании разработанного расширения в среду добавляется команда «Запустить верификатор», которая по абстрактному дереву строит модель автомата и спецификацию в виде *LTL*-формул и передает их на вход верификатора *NuSMV*. Результат верификации отображается пользователю.

Для разработки данного расширения потребовалось описать структуру языка, а также реализовать логику редактирования спецификации и код запуска верификатора.

4.2.4. Реализация языка спецификации автоматов

Для реализации языка в системе *JetBrains MPS* требуется описать следующие его аспекты:

- структуру абстрактного синтаксического дерева языка;
- модель тестового редактора для каждого типа узлов абстрактного синтаксического дерева;
- модель ограничений для узлов дерева;
- систему типов.

Опишем структуру дерева. Языка состоит из специальных конструкций `specification`, `requires`, `ensures` и `invariant`, а также позволяет вводить булевы и темпоральные выражения.

Для описания выражений создан базовый концепт `BaseTemporalExpression`, наследованный от стандартного концепта `Expression`, представляющего произвольное выражение. От концепта `BaseTemporalExpression` наследованы концепты `UnaryTemporalExpression` и `BinaryTemporalExpression` для унарных и бинарных темпоральных выражений. Эти типы содержат ссылки на операнды. Для концепта `BaseTemporalExpression` также задано ограничение, которое не позволяет использовать темпоральные операторы вне секции `specification`.

Для концептов `UnaryTemporalExpression` и `BinaryTemporalExpression` описаны редакторы выражений. Для этих выражений также определена система типов, в соответствии с которой темпоральные выражения имеют булевский тип (`boolean` в системе *MPS*). Система типов задает ограничение, в соответствии с которым операндами темпоральных операторов могут выступать только выражения булевского типа.

Также введены концепты `AndOperation`, `OrOperation`, `NotOperation` и `ImplicationOperation`, реализующие булевы операторы. Эти концепты так же, как и концепт `BaseTemporalExpression`, расширяют концепт `Expression`. Собственная реализация операторов была сделана для точного контроля над набором конструкций, которые разрешено использовать в формулах спецификаций и контрактов. Для такого контроля для всех концептов языка *stateSpec* написаны ограничения, не позволяющие использовать конструкции других языков.

Для облегчения ввода двуместных операторов в редакторах соответствующих концептов реализовано так называемое «правое преобразование». Обычно при редактировании дерева в системе *MPS* задается «сверху вниз». При таком подходе для ввода двуместных операторов сначала требуется указать тип оператора. После этого в появившемся редакторе для соответствующего выражения становится возможным указать операнды. Однако в традиционных языках программирования ввод таких операторов осуществляется по схеме «левый операнд – знак оператора – правый операнд». Для реализации возможности такого ввода и используется «правое преобразование». Это преобразование состоит в том, что при вводе знака оператора справа от выражения, узел этого выражения заменяется узлом соответствующего оператора, а выражение становится его левым операндом.

Для реализации утверждений, являющихся ссылками на состояния, входные и выходные воздействия, созданы концепты `StateReference`, `FieldReference` и `MethodReference`. Особенностью реализации этих концептов является то, что они используют информацию об автомате для реализации списков автоматического завершения кода.

Автоматическое завершение кода заключается в возможности отобразить список идентификаторов, которые возможно ввести в данной точке. При выборе идентификатора из списка происходит его вставка в код программы. Автоматическое завершение кода позволяет увеличить скорость ввода программы и значительно уменьшить число ошибок в ней. Для реализации этой возможности код концептов языка *stateSpec*, выражающих утверждения, анализирует структуру автомата и составляет список состояний, входных и выходных воздействий, идентификаторы которых возможно использовать в точке ввода.

Блоками высокого уровня в языке *stateSpec* являются концепты *Specification* и *StateContract*, реализующие блоки описания спецификаций и контрактов. Внедрение этих блоков в редактор автомата осуществляется с использованием средств расширения языка *stateMachine*. Эти средства позволяют расширять набор конструкций, разрешенных к использованию в описании автомата, путем реализации интерфейса *IStateElement*.

При запуске команды верификации происходит построение модели программы и спецификации на языке *SMV*, являющемся входным языком свободно распространяемого верификатора *NuSMV*. Подход, используемый в языке *stateSpec*, аналогичен предложенному в работе [57]. Одним из основных отличий является использование языка темпоральной логики *LTL* вместо языка дерева вычислений (*Computation Tree Logic, CTL*).

Переменные модели определяются структурой автомата и набором условий на переходах. Так как в качестве условий на переходах могут выступать сложные выражения на языке *Java*, использующие вызовы других объектов программы, подробный анализ их истинности произвести сложно. В связи с этим из набора используемых выражений удаляются дубликаты, и оставшиеся выражения представляются в виде входных переменных.

4.2.5. Пример использования языка спецификации автоматов

В качестве примера использования языка спецификации автоматов, рассмотрим специфицирование требований к автомату управления лифтом, рассмотренному во второй главе и в разд. 4.2.2.

Во второй главе была произведена формализация требований к этому автомату в виде:

- инварианта \bar{z}_4 для состояния s_1 ;
- предусловия $s_2 \vee s_4$ для состояния s_1 ;
- инварианта x_4 для состояний s_2 и s_4 .

Эти контракты записываются следующим образом:

```

state {MOVING} {
  invariant {
    !doors_open
  }

  requires {
    OPENING_DOORS || WAITING_DOORS_OPENED
  }
}

state {OPENING_DOORS} {
  invariant {
    light_on
  }
}

state {WAITING_DOORS_OPENED} {
  invariant {

```

```

    light_on
  }
}

```

В качестве альтернативы два инварианта на состояния `OPENING_DOORS` и `WAITING_DOORS_OPENED` могут быть записаны в виде одной темпоральной формулы, объединяющей оба условия:

specification

```

{
  (OPENING_DOORS || WAITING_DOORS_OPENED) -> light_on
}

```

Выводы по главе 4

1. Рассмотрена проблема интеграции автоматного и объектно-ориентированного кода, а также интеграции кода и спецификации автоматных программ.
2. Показано, как проблему интеграции языков можно решить с использованием возможностей современных динамических языков программирования на примере разработанной автором библиотеки *STROBE* для языка программирования *Ruby*.
3. Показано, как проблема интеграции автоматного и объектно-ориентированного кода может быть решена с использованием мультязыковых сред на примере среды *JetBrains MPS*.
4. Разработан подход к интеграции кода и спецификации автоматных программ на основе использования возможностей мультязыковых сред.
5. Реализован язык *stateSpec* для описания спецификаций автоматных программ в виде темпоральных формул и контрактов с возможностью их верификации.

ГЛАВА 5. ВНЕДРЕНИЕ РЕЗУЛЬТАТОВ РАБОТЫ В ПРАКТИКУ РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

5.1. Область внедрения

Программа для учета дефектов *YouTrack* разработана в компании *ООО «ИнтеллиДжей Лабс»*, работающей на мировом рынке под брендом *JetBrains*. Программа *YouTrack* представляет собой интернет-приложение для работы с базами дефектов. Это приложение позволяет вводить новые дефекты, осуществлять поиск дефектов в базе данных, и осуществлять организацию записей о дефектах.

Программа *YouTrack* реализована в виде системы следующих взаимодействующих модулей:

- серверный модуль, работающий с базой данных;
- серверный модуль, реализующий логику приложения;
- клиентский модуль, работающий в браузере пользователя и реализующий пользовательский интерфейс.

Часть поведения этой программы реализована в виде автоматов. В частности, автоматы используются в серверной части для синтаксического анализа запросов и в клиентской части для реализации логики пользовательского интерфейса.

5.2. Использование системы *JetBrains MPS* в программе *YouTrack*

Технической особенностью программы *YouTrack* является то, что она реализуется на базе системы *JetBrains MPS*. В четвертой главе было приведено описание этой системы.

Для реализации программы *YouTrack* используются следующие предметно-ориентированные языки, сконструированные с помощью системы *MPS*:

- *baseLanguage* – язык, повторяющий синтаксис языка *Java*, используется для написания императивного кода;
- *dnq* – язык работы с данными, позволяющий выполнять запросы к спискам записей;
- *webr* – язык для разработки веб-приложений;
- *stateMachine* – язык для описания автоматов.

Язык *stateMachine*, прообразом которого является текстовый язык автоматного программирования [69], позволяет автоматизировать классы, написанные на языке *baseLanguage*. Особенностью использования языка *stateMachine* в программе *YouTrack* является то, что в автоматах, работающих на сервере, в качестве языка генерации используется *Java*, а в автоматах, работающих в браузере – *JavaScript*. Это потребовало выполнить две реализации языка *stateSpec* для различных языков.

Для обеспечения качества программы *YouTrack* применяются модульные тесты, использующие каркасы тестирования *JUnit* и *Selenium*.

Целью внедрения разработанного языка *stateSpec* в программу *YouTrack* являлось повышение качества автоматной части программы. Для этого была произведена формализация спецификаций автоматизированных объектов. Внедрение языка *stateSpec* также позволяет использовать верификацию вместо тестирования при разработке новых автоматизированных модулей в программе.

Опишем формализацию требований для двух автоматизированных объектов, используемых в реализации пользовательского интерфейса программы *YouTrack*.

5.3. Автомат управления списком дефектов *IssueList.js*

Одним из элементов интерфейса программы *YouTrack* является список дефектов. Этот элемент позволяет просматривать выбранный набор дефектов и выбирать подмножество дефектов из списка. Для навигации по списку используются клавиши управления кареткой. Пример работы списка дефектов изображен на рис. 22.

<input type="checkbox"/>	DTRC-1495 History doesn't get updated on taking snapshots	Nov 20	▶
<input checked="" type="checkbox"/>	DTRC-1534 The message which appears when showing empty plain list should only appear in plain list view and not be closeable	Nov 19	▶
<input checked="" type="checkbox"/>	DTRC-1544 Use softer colors for predefined coloring filters	Nov 18	▶
<input type="checkbox"/>	DTRC-1543 Source view doesn't remember path mappings	Nov 18	▶
<input type="checkbox"/>	DTRC-1541 Source view header should display source file name when available	Nov 18	▶
<input type="checkbox"/>	DTRC-1540 Source view header shouldn't be that tall in horizontal mode	Nov 18	▶
<input checked="" type="checkbox"/>	DTRC-1538 When I add annotation, the annotation sign is not added to another occurrences of the function already shown in the tree.	Nov 18	▶
<input checked="" type="checkbox"/>	DTRC-1537 Don't show old times for adjusted functions in the tree	Nov 18	▶
<input checked="" type="checkbox"/>	DTRC-1533 Selected view in the view switcher panel must not appear as disabled	Nov 18	▶
<input type="checkbox"/>	DTRC-1459 Views panel stretches icons horizontally a little when collapsed.	Nov 16	▶
<input type="checkbox"/>	DTRC-1529 Error description when opening old snapshot doesn't explain anything	Nov 16	▶

Рис. 22. Пример работы списка дефектов

Состоянием списка дефектов является набор выбранных дефектов (они имеют более темный фон на рисунке), и дефект, который в данный момент принимает команды пользователя (его идентификатор обведен рамкой на рисунке).

Логика работы списка дефектов реализована в классе `IssueList.js`. Поведение этого класса задается автоматом, реализованным на языке `stateMachine`, текст которого приведен ниже:

```
state program for IssueList.js
  state machine for charisma.smartui.IssueList {
    initial state {NOT_SHIFTED} {
```

```

    exit do {this.ch =
!(this.getSelected().checkbox.checked);}

    on shiftDown() do {<< statements >>} transit to
SHIFTED
}

```

```

state {SHIFTED} {
    initial state {INITIAL} {
        on kdown() do {<< statements >>}
        transit to MOVE_DOWN

        on kup() do {<< statements >>}
        transit to MOVE_UP

        on shiftUp() do {<< statements >>}
        transit to NOT_SHIFTED
    }
}

```

```

state {MOVE_DOWN} {
    enter do
{this.getSelected().setChecked(this.ch);}

    on kdown() do
{this.getSelected().setChecked(this.ch);}

    on kup() do {this.ch = !this.ch;}
    transit to MOVE_UP
}

```

```

    on shiftUp() do {<< statements >>}
        transit to NOT_SHIFTED
    }

state {MOVE_UP} {
    enter do
{this.getSelected().setChecked(this.ch);}

    on kup() do
{this.getSelected().setChecked(this.ch);}

    on kdown() do {this.ch = !this.ch;}
        transit to MOVE_DOWN

    on shiftUp() do {<< statements >>}
        transit to NOT_SHIFTED
    }
}
}

```

Автомат обрабатывает четыре события: `kdown` (нажата клавиша «Вниз»), `kup` (нажата клавиша «Вверх»), `shiftDown` (нажата клавиша «Shift») и `shiftUp` (отпущена клавиша «Shift»). Внутренним вычислительным состоянием является значение флага `ch`, указывающее на то, добавляется ли выбранный элемент в выделенный набор или исключается из него.

Выходными воздействиями автомата являются инвертирование флага `ch` («`this.ch = !this.ch`»), включение/исключение выбранного элемента из набора выделенных элементов

(«`this.getSelected().setChecked(this.ch)`»), а также инициализация флага `ch` («`this.ch = !(this.getSelected().checkbox.checked)`»).

Автомат содержит два состояния верхнего: `NOT_SHIFTED` («Клавиша *Shift* не нажата») и `SHIFTED` («Клавиша *Shift* нажата»). Второе состояние включает в себя три вложенных состояния: `INITIAL` («Начальное»), `MOVE_UP` («Перемещение фокуса вверх») и `MOVE_DOWN` («Перемещение фокуса вниз»).

Требования к автоматизированному объекту управления `IssueList.js` формулируются следующим образом:

- в состояние `MOVE_UP` автомат переходит только после нажатия клавиши «Вверх»;
- в состояние `MOVE_DOWN` автомат переходит только после нажатия клавиши «Вниз»;
- при движении курсором в одном направлении не должно производиться инвертирование флага `ch`.

Два первых пункта требований были формализованы в виде **контрактов-предусловий** на состояния `MOVE_UP` и `MOVE_DOWN` с формулами `kup` и `kdown` соответственно. Третий пункт был представлен в виде двух темпоральных формул:

- `kup -> !(this.ch = !this.ch) R (kdown || shiftUp || shiftDown);`
- `kdown -> !(this.ch = !this.ch) R (kup || shiftUp || shiftDown).`

Верификация автомата `IssueList.js` по этим формулам была успешно произведена.

5.4. Автомат управления выпадающей подсказкой *Suggest.js*

Программа *YouTrack* использует еще один компонент пользовательского интерфейса, реализованный с помощью автоматного подхода. Этим компонентом является всплывающая подсказка в поле поиска. Пример того, как она выглядит в работе, представлен на рис. 23:

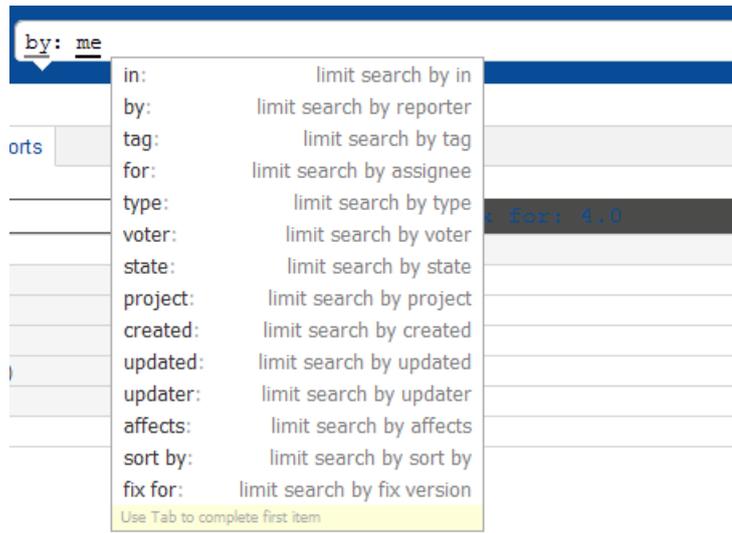


Рис. 23. Пример работы выпадающей подсказки

Всплывающая подсказка содержит список конструкций, которые можно использовать для продолжения введенного запроса. Элементы списка могут быть выбраны с клавиатуры или с помощью «мыши». Для выбора элемента с помощью клавиатуры необходимо клавишами «вверх» и «вниз» установить выделение на нужный элемент, а затем нажатием клавиши «ввод» или «табуляция» подтвердить выбор. Для выбора элемента «мышью» требуется навести курсор на нужный элемент и нажать левую клавишу «мыши». До выбора элемента с помощью клавиатуры нажатие клавиши «табуляция» приводит к выбору первого элемента в списке.

Код автомата управления выпадающей подсказкой имеет следующий вид:

```
state program for Suggest.js
  state machine for Webr.component.Suggest {
```

```

initial state {UNFOCUSED} {
    on focus() do {<< statements >>}
    transit to HIDDEN
}

state {HIDDEN} {
    enter do {this.select(-1, false);}

    on blur() do {<< statements >>}
    transit to UNFOCUSED

    on show(empty == false) do
{this.setVisible(true);}
    transit to NOTHING_SELECTED
}

state {VISIBLE} {
    on show(empty == true) do
{this.setVisible(false);}
    transit to HIDDEN
    on blur() do {this.setVisible(false);}
    transit to UNFOCUSED

    on keyPressed(event) [event.isApplicable(Close)]
do {
    this.setVisible(false);
    event.stopPropagation();
} transit to HIDDEN

```

```

    on mouseClicked(event, itemIndex)
[this.canComplete()] do {
    $(this.input).focus();
    this.complete(itemIndex);
    this.setVisible(false);
} transit to HIDDEN

initial state {NOTHING_SELECTED} {
    enter do {this.setImplicitCompleteComment();}

    on keyPressed(event)
[event.isApplicable(ImplicitComplete) &&
this.canComplete()] do {
    this.complete(0);
    this.setVisible(false);
    event.stopPropagation();
} transit to HIDDEN

    on keyPressed(event)
[event.isApplicable(PreviousItem)] do {
    this.select(this.data.items.length - 1,
false);
    event.stopPropagation();
} transit to SELECTED_WITH_KEYBOARD

    on keyPressed(event)
[event.isApplicable(NextItem)] do {
    this.select(0, false);
    event.stopPropagation();
}

```

```

    } transit to SELECTED_WITH_KEYBOARD

    on show(empty == false) do
{this.setVisible(true);}

    on mouseMove(event, itemIndex) do
{this.select(itemIndex, true);} transit to
SELECTED_WITH_MOUSE
    }

state {ITEM_SELECTED} {
    on keyPressed(event)
[event.isApplicable(PreviousItem)] do {
        this.select(this.getPrevious(), false);
        event.stopPropagation();
    } transit to SELECTED_WITH_KEYBOARD

    on keyPressed(event)
[event.isApplicable(NextItem)] do {
        this.select(this.getNext(), false);
        event.stopPropagation();
    } transit to SELECTED_WITH_KEYBOARD

initial state {SELECTED_WITH_MOUSE} {
    enter do {this.setImplicitCompleteComment();}

    on keyPressed(event)
[event.isApplicable(ImplicitComplete) &&
this.canComplete()] do {

```

```

    this.complete(0);
    this.setVisible(false);
    event.stopPropagation();
} transit to HIDDEN

on show(empty == false) do {
    this.select(-1, false);
    this.setVisible(true);
} transit to NOTHING_SELECTED

    on mouseMove(event, itemIndex) [itemIndex !=
this.selectedIndex] do {this.select(itemIndex, true);}
}

state {SELECTED_WITH_KEYBOARD} {
    enter do {this.setExplicitCompleteComment();}

    on keyPressed(event)
[event.isApplicable(ExplicitComplete) &&
this.canComplete()] do {
        this.complete(this.selectedIndex);
        this.setVisible(false);
        event.stopPropagation();
    } transit to HIDDEN

    on keyPressed(event)
[event.isApplicable(SpaceComplete) &&
this.canComplete()] do {
        this.complete(this.selectedIndex);

```

```

        this.setVisible(false);
        event.stopPropagation();
    } transit to HIDDEN

    on keyPressed(event)
[event.isApplicable(ImplicitComplete) &&
this.canComplete()] do {
    this.complete(this.selectedIndex);
    this.setVisible(false);
    event.stopPropagation();
} transit to HIDDEN

    on show(empty == false)
[this.findSelectedOption() == -1] do {
    this.select(-1, false);
    this.setVisible(true);
} transit to NOTHING_SELECTED

    on show(empty == false)
[this.findSelectedOption() != -1] do {
    this.select(this.findSelectedOption(),
false);

    this.setVisible(true);
}

    on mouseMove(event, itemIndex) [itemIndex !=
this.selectedIndex] do {this.select(itemIndex, false);}
}
}

```

```

    }
  }

```

Были определены следующие требования для этого элемента управления:

1. До нажатия клавиш «Вверх» и «Вниз» нажатие клавиши «Табуляция» приводит к выбору первого элемента.
2. Потеря фокуса приводит к переходу в состояние UNFOCUSED.
3. Переход в состояние `SELECTED_WITH_MOUSE` происходит только после движения мыши.

Эти требования были формализованы с помощью предложенного метода и представлены в виде контрактов и темпоральных спецификаций. Особенностью работы с автоматом `Suggest.js` является то, что при обработке событий производится дополнительная проверка условий. Например, в состоянии `NOTHING_SELECTED` определены следующие переходы:

```

    on keyPressed(event)
[event.isApplicable(PreviousItem)] do {
    this.select(this.data.items.length - 1,
false);
    event.stopPropagation();
} transit to SELECTED_WITH_KEYBOARD

```

```

    on keyPressed(event)
[event.isApplicable(NextItem)] do {
    this.select(0, false);
    event.stopPropagation();
} transit to SELECTED_WITH_KEYBOARD

```

Оба перехода происходят по событию `keyPressed`, но при этом должны выполняться различные условия (в первом случае проверяется, что нажата клавиша перехода к предыдущему элементу, а во втором – к следующему). Для статической обработки таких событий производится синтаксическое сравнение текстов условий и при построении модели для верификатора создаются отдельные переходы для каждой комбинации событие-условие. При определении спецификации программист также указывает конкретную комбинацию события и условия. Так, при формализации первого пункта требований, событие «нажатие клавиши «вниз» формализуется как `keyPressed(event) [event.isApplicable(NextItem)]`.

Выводы по главе 5

1. Разработана версия языка *stateSpec*, использующая в качестве языка реализации автоматизированных объектов управления *JavaScript*. Возможность такой реализации является свидетельством высокой гибкости предлагаемого метода.
2. В программе учета дефектов *YouTrack* использован язык *stateSpec* для формализации требований к автоматам, реализующим логику поведения компонентов пользовательского интерфейса. Для формализации использованы формулы темпоральной логики и контракты.

ЗАКЛЮЧЕНИЕ

В настоящей работе разработаны методы реализации автоматных объектно-ориентированных программ. Эти методы позволяют:

- формализовать требования к автоматным объектно-ориентированным программам в виде контрактов и темпоральных формул;
- использовать темпоральные формулы и контракты для статической и динамической проверки автоматных объектно-ориентированных программ;
- интегрировать код и спецификацию автоматных объектно-ориентированных программ;
- уменьшить число изменений, которые могут привести к появлению ошибок, при модификации автоматных программ.

Для реализации ряда предложенных методов был разработан язык *stateSpec*, позволяющий описывать спецификацию автоматных объектно-ориентированных программ в виде контрактов и темпоральных формул. Также разработанный язык позволяет проводить верификацию программ непосредственно в среде разработки ПО.

Созданный язык был использован для формализации спецификаций компонентов интерфейса программы *YouTrack*. Благодаря использованию языка *stateSpec* удалось представить требования к этим компонентам в виде контрактов и темпоральных спецификаций и провести верификацию.

Основными направлениями развития полученных результатов автор считает следующие:

- автоматизация рефакторингов автоматных программ;
- развитие языков формализации требований к автоматным программам.

Полученные результаты позволяют более эффективно использовать автоматное программирование при разработке программного обеспечения, а также повысить качество создаваемых программ.

ИСТОЧНИКИ

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
<http://is.ifmo.ru/books/switch/1>
2. *Грэхем И.* Объектно-ориентированные методы. М.: Вильямс, 2004.
3. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Интернет-университет информационных технологий, 2005.
4. *Ларман К.* Применение UML и шаблонов проектирования. М.: Вильямс, 2002.
5. *Kurzweil R.* The Singularity Is Near: When Humans Transcend Technology. Penguin, 2006.
6. *Beck K., Andres C.* Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004.
7. *Schwaber K.* Agile Project Management with Scrum. Microsoft Press, 2004.
8. *Turner M.* Microsoft Solutions Framework Essentials. Microsoft Press, 2006.
9. *Alexander C.* A Pattern Language: Towns, Buildings, Construction. USA: Oxford University Press, 1977.
10. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.
11. *Фаулер М.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2004.
12. *Канер С., Фолк Д., Нгуен Е.* Тестирование программного обеспечения. Киев: ДиаСофт, 2000.

13. *Ахо А., Лам М., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии и инструментарий. Вильямс, 2008.
14. *Lyu M.R., Horgan J.R., London S.* A coverage analysis tool for the effectiveness of software testing //IEEE Transactions on Reliability. 1994, № 43(4).
15. Веб-сайт каркаса JUnit. <http://junit.org/>
16. Веб-сайт каркаса NUnit. <http://www.nunit.com/>
17. Веб-сайт каркаса PyUnit. <http://pyunit.sourceforge.net/>
18. *Дюваль П., Матиас С., Гловер Э.* Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска. М.: Вильямс, 2008.
19. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.
20. *Wing J.M.* Writing Larch interface language specifications //ACM Trans. Program. Lang. Syst. 1987, № 9.
21. Веб-сайт проекта Code Contracts компании Microsoft. <http://research.microsoft.com/en-us/projects/contracts/>
22. Веб-сайт проекта Contract4j. <http://www.contract4j.org/contract4j>
23. *Barnett M., DeLine R., Fähndrich M., Leino K.Rustan M., Schulte W.* Verification of object-oriented programs with invariants //Journal of Object Technology. 2004, № 6.
24. *Шалыто А. А., Туккель Н. И.* Программирование с явным выделением состояний //Мир ПК. 2001, № 8, 9. <http://is.ifmo.ru/works/mirk/>
25. *Шопырин Д. Г., Шалыто А. А.* Объектно-ориентированный подход к автоматному программированию //Информационно-управляющие системы. 2003, № 5. <http://is.ifmo.ru/works/ooaut/>

26. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2010.
27. *Гуров В. С.* Технология проектирования и разработки объектно-ориентированных программ с явным выделением состояний (метод, инструментальное средство, верификация). Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2008.
28. *Степанов О. Г.* Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby. Магистерская диссертация. СПбГУ ИТМО, 2006.
29. *Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А.* State Machine – новый паттерн объектно-ориентированного проектирования // Информационно-управляющие системы. 2004, № 5. <http://is.ifmo.ru/works/pattern/>
30. *Наумов Л. А.* Объектно-ориентированное программирование с явным выделением состояний // Информационно-управляющие системы. 2003, № 6.
31. *Фельдман П. И.* Разработка средств для отладки автоматных программ, построенных на основе предложенной библиотеки классов. СПбГУ ИТМО, 2004. http://is.ifmo.ru/papers/aut_dlf/
32. *Астафуров А. А., Шалыто А. А.* Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования. Software Engineering Conference (Russia). М.: ТЕКАМА, 2007.
33. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования // Научно-технический вестник СПбГУ ИТМО. 2008, № 53. http://is.ifmo.ru/works/2007_10_05_mps_textual_language.pdf
34. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Инструментальное средство для поддержки автоматного программирования // Программирование. 2007, № 6.

35. *Дмитриев С.* Языково-ориентированное программирование: следующая парадигма. 2005. <http://www.rsdn.ru/article/philosophy/LOP.xml>
36. *Фаулер М.* Языковой инструментарий: новая жизнь языков предметной области. <http://www.maxkir.com/sd/languageWorkbenches.html>
37. *Степанов О. Г., Шалыто А. А., Шопырин Д. Г.* Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby. 2007. http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf
38. *Астафуров А., Тимофеев К., Шалыто А.* Наследование автоматных классов с использованием динамических языков программирования на примере Ruby. М.: ТЕКАМА, 2008. <http://www.secr.ru/?pageid=4548&submissionid=5270>
39. *Бурдонов И. Б., Косачев А. С., Кулямин В. В.* Использование конечных автоматов для тестирования программ //Программирование. 2000, № 26.
40. *Гуров В. С., Мазин М. А., Шалыто А. А.* Ядро автоматного программирования. Свидетельство о государственной регистрации программы для ЭВМ. 2006.
41. *Грис Д.* Наука программирования. М.: Мир, 1984.
42. *Непомнящий В. А., Рякин О. М.* Прикладные методы верификации программ. М.: Радио и связь, 1988.
43. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ. Model Checking. М.: МЦНМО, 2002.
44. *Pnueli A.* The Temporal Logic of Programs / Proceedings of the 18-th IEEE Symposium on Foundation of Computer Science. 1977.
45. *Кузьмин Е. В., Соколов В. А.* Моделирование, спецификация и верификация «автоматных» программ //Программирование. 2008, № 1.

46. *Вельдер С. Э., Шалыто А. А.* О верификации простых автоматных программ на основе метода Model Checking //Информационно-управляющие системы. 2007, № 3.
47. *Яминов Б. Р., Шалыто А. А.* Расширение верификатора Bagot для верификации автоматных UniMod-моделей. Свидетельство о государственной регистрации программы для ЭВМ. 2008.
48. *Лукин М. А., Шалыто А. А.* Трансляция UniMod-модели во входной язык верификатора SPIN. Свидетельство о государственной регистрации программы для ЭВМ. 2008.
49. *Kurbatsky E.* Verification of Automata-Based Programs. SPbSU, 2008.
50. *Emerson E.A.* Temporal and modal logic. MIT Press, 1990.
51. *Разработка технологии верификации* управляющих программ со сложным поведением, построенных на основе автоматного подхода. Теоретические исследования поставленных перед НИР задач. СПбГУ ИТМО, 2007. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
52. *Kalaji A., Hierons R.M., Swift S.* Automatic Generation of Test Sequences form EFSM Models Using Evolutionary Algorithms. 2008.
53. *Finkbeiner B., Sipma H.* Checking Finite Traces Using Alternating Automata //Form. Methods Syst. Des. 2004, № 24.
54. *Vardi M.* Alternating Automata and Program Verification //Computer Science Today. Recent Trends and Developments. LNCS 1000. 1995.
55. *M.Y. Vardi.* Alternating Automata: Checking Truth and Validity for Temporal Logics. Springer–Verlag, 1997.
56. *Havelund K., Roşu G.* Testing Linear Temporal Logic Formulae on Finite Execution Traces. 2001.

57. *Разработка технологии верификации* управляющих программ со сложным поведением, построенных на основе автоматного подхода. Выбор направления исследований и базовых компонентов. СПбГУ ИТМО, 2007. http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
58. Фаулер М. Рефакторинг. М.: Вильямс, 2003.
59. Кафедра «Технологии программирования». Раздел «Проекты». <http://is.ifmo.ru/projects/>
60. Козлов В. А., Комалева В. А. Моделирование работы банкомата. СПбГУ ИТМО, 2006. <http://is.ifmo.ru/unimod-projects/bankomat/>
61. Балтийский И. А., Гиндин С. И. Моделирование работы банкомата. СПбГУ ИТМО, 2008. <http://is.ifmo.ru/unimod-projects/atm/>
62. Boo language website. <http://boo.codehaus.org/>
63. Nemerle language website. <http://nemerle.org>
64. Гуров В. С., Мазин М. А., Шалыто А. А. Операционная семантика UML–диаграмм состояний в программном пакете Unimod / Телематика–2005. СПбГУ ИТМО, 2005.
65. Cardelli L., Wegner P. On Understanding Types, Data Abstraction, and Polymorphism // Computing Surveys. 1985, №12.
66. Simonyi C. The Death of Computer Languages, the Birth of Intentional Programming // The Future of Software. Univ. of Newcastle upon Tyne. England. Dept. of Computing Science, 1995.
67. Конопко К. С. HELGINS: универсальный язык для написания анализаторов типов // Компьютерные инструменты в образовании. 2007, № 4.
68. MPS User's Guide. <http://www.jetbrains.net/confluence/display/MPS/MPS+User%27s+Guide>

69. *Гуров В. С., Мазин М. А., Шальто А. А.* Текстовый язык автоматного программирования //Научно-технический вестник СПбГУ ИТМО. 2007, № 42.