Федеральное агентство по образованию

УДК 004.4’242
ГРНТИ 50.41.25
Инв. № 390174-1

ПРИНЯТО:	УТВЕРЖДЕНО:
Приемочная комиссия Государственного заказчика:	**Государственный заказчик Федеральное агентство по образованию**
От имени Приемочной комиссии **_____/Мосичева И.А. /**	**От имени Государственного заказчика** **_____/Бутко Е.Я./**

НАУЧНО-ТЕХНИЧЕСКИЙ ОТЧЕТ

о выполнении 1 этапа Государственного контракта
№ П2373 от 18 ноября 2009 г.

Исполнитель: Государственное образовательное учреждение высшего профессионального образования "Санкт-Петербургский государственный университет информационных технологий, механики и оптики"
Программа (мероприятие): Федеральная целевая программ «Научные и научно-педагогические кадры инновационной России» на 2009-2013 гг., в рамках реализации мероприятия № 1.2.2 Проведение научных исследований научными группами под руководством кандидатов наук.
Проект: Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования
Руководитель организации: Васильев Владимир Николаевич
Руководитель проекта: Шопырин Данил Геннадьевич

Согласовано: **Управление научных исследований и инновационных программ** **От имени Заказчика** **_____/Кошкин В.И./**

Санкт-Петербург
2009 г.

СПИСОК ОСНОВНЫХ ИСПОЛНИТЕЛЕЙ

по Государственному контракту П2373 от 18 ноября 2009 на выполнение поисковых научно-исследовательских работ для государственных нужд

Организация-Исполнитель: Государственное образовательное учреждение высшего профессионального образования "Санкт-Петербургский государственный университет информационных технологий, механики и оптики"

Руководитель темы:

кандидат технических наук, без ученого звания	_____ подпись, дата	Шопырин Д. Г.

Исполнители темы:

кандидат технических наук, без ученого звания	_____ подпись, дата	Гуров В. С.
кандидат технических наук, без ученого звания	_____ подпись, дата	Корнеев Г. А.
без ученой степени, без ученого звания	_____ подпись, дата	Царев Ф. Н.
без ученой степени, без ученого звания	_____ подпись, дата	Мазин М. А.
без ученой степени, без ученого звания	_____ подпись, дата	Степанов О. Г.
без ученой степени, без ученого звания	_____ подпись, дата	Лукин М. А.

без ученой степени, без ученого звания	_____ подпись, дата	Астафуров А. А.
без ученой степени, без ученого звания	_____ подпись, дата	Яминов Б. Р.
без ученой степени, без ученого звания	_____ подпись, дата	Кочелаев Д. Ю.
без ученой степени, без ученого звания	_____ подпись, дата	Тимофеев К. И.
без ученой степени, без ученого звания	_____ подпись, дата	Егоров К. В.
без ученой степени, без ученого звания	_____ подпись, дата	Царев М. Н.
без ученой степени, без ученого звания	_____ подпись, дата	Малаховски Я. М.
без ученой степени, без ученого звания	_____ подпись, дата	Буздалов М. В.
без ученой степени, без ученого звания	_____ подпись, дата	Борисенко А. А.
без ученой степени, без ученого звания	_____ подпись, дата	Федотов П. В.

РЕФЕРАТ

Отчет 87 с., четыре гл.,16 рис., одна табл., 96 источников.

Ключевые слова: автоматное программирование; валидация; верификация; объектно-ориентированное программирование; качество программ; функциональное программирование.

В настоящем отчете излагаются результаты выполнения первого этапа поисковых научно-исследовательских работ по направлениям «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» по проблеме «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования», выполняемых в рамках государственного контракта, заключенного между Федеральным агентством по образованию и государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» в соответствии с решением Единой комиссии (протокол от 29 октября 2009 г. № 3/НК–421П) по конкурсу № НК–421П «Проведение поисковых научно-исследовательских работ по направлениям: «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» в рамках мероприятия 1.2.2 Программы «Проведение научных исследований научными группами под руководством кандидатов наук» по направлению 1 «Стимулирование закрепления молодежи в сфере науки, образования и высоких технологий» федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы, утвержденной постановлением Правительства Российской Федерации от 28 июля 2008 года № 568 «О федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы».

Целями настоящего этапа являются:

1. Выполнение аналитического обзора.
2. Выбор и обоснование оптимального варианта направления исследований.
3. Подготовка плана проведения теоретических и экспериментальных исследований.
4. Разработка методов реализации автоматов на функциональных языках программирования.

При выполнении первого этапа работ использовался следующий инструментарий:

1. Статьи в ведущих зарубежных и российских журналах, монографии и патенты за период 1998–2008 гг.
2. Результаты автоматического поиска в сети Интернет по ключевым словам и шаблонам.
3. Работы членов коллектива, опубликованные в рамках НИР по схожей тематике.
4. Аналитический обзор, составленный в рамках НИР.
5. Работы членов коллектива, опубликованные в рамках НИР по схожей тематике.
6. Постановление Правительства Российской Федерации от 4 мая 2005 г. № 284 «О государственном учете результатов научно-исследовательских, опытно-конструкторских и технологических работ гражданского назначения».
7. Персональный компьютер.
8. Язык программирования *Haskell*.
9. ГОСТ 7.32–2001 «Отчет о научно-исследовательской работе. Структура и правила оформления».

Излагаются результаты выполнения аналитического обзора по следующим направлениям: качество программ, автоматное программирование, функциональное программирование, объектно-ориентированное программирование, автоматное объектно-ориентированное программирование, методы повышения качества объектно-ориентированных программ, методы повышения качества автоматных объектно-ориентированных программ, обзор методов и средств верификации на модели.

Приводится обоснование выбора оптимального варианта направления исследований, а также план проведения теоретических и экспериментальных исследований.

Описываются методы реализации автоматов на функциональных языках программирования. Приводятся примеры применения описанных методов.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящем отчете излагаются результаты выполнения первого этапа поисковых научно-исследовательских работ по направлениям «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» по проблеме «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования», выполняемых в рамках государственного контракта, заключенного между Федеральным агентством по образованию и государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» в соответствии с решением Единой комиссии (протокол от 29 октября 2009 г. № 3/НК–421П) по конкурсу № НК–421П «Проведение поисковых научно-исследовательских работ по направлениям: «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» в рамках мероприятия 1.2.2 Программы «Проведение научных исследований научными группами под руководством кандидатов наук» по направлению 1 «Стимулирование закрепления молодежи в сфере науки, образования и высоких технологий» федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы, утвержденной постановлением Правительства Российской Федерации от 28 июля 2008 года № 568 «О федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы».

Целями настоящего этапа являются:

1. Выполнение аналитического обзора.
2. Выбор и обоснование оптимального варианта направления исследований.
3. Подготовка плана проведения теоретических и экспериментальных исследований.
4. Разработка методов реализации автоматов на функциональных языках программирования.

Отчет имеет следующую структуру. В первой главе приводятся результаты выполнения аналитического обзора по направлениям:

- качество программ;
- автоматное программирование;
- функциональное программирование;
- объектно-ориентированное программирование;
- автоматное объектно-ориентированное программирование;
- методы повышения качества объектно-ориентированных программ;
- методы повышения качества автоматных объектно-ориентированных программ;
- обзор методов и средств верификации на модели.

Во второй главе приводится план проведения теоретических и экспериментальных исследований.

В третьей главе обосновывается выбор оптимального направления исследований.

В четвертой главе приводится описание методов реализации автоматов на функциональных языках программирования.

Каждая из глав снабжена выводами, кратко резюмирующими содержание главы. В заключении дается общая оценка работ по этапу.

В настоящее время при разработке ответственных систем все чаще применяются методы автоматного программирования, основной концепцией которого является представление поведения программных систем в виде взаимодействующих структурных конечных автоматов. Такое представление

программ позволяет эффективно применять известные методы верификации, такие как *Model Checking*.

Функциональное программирование в последние годы становится весьма популярной областью исследований. В свою очередь, современные средства разработки на функциональных языках предоставляют программные библиотеки, в которых можно найти практически любой популярный алгоритм, структуру данных или вспомогательные конструкции для реализации различных методов программирования (например, монады), однако в этих библиотеках отсутствуют инструменты для применения подходов автоматного программирования.

Большинство разрабатываемых сегодня программ строится на основе объектно-ориентированного подхода, для которого разработан целый рад методов повышения качества, таких как: тестирование, рефакторинг и паттерны проектирования. В тоже время, активно применяемые в ответственных системах методы объектно-ориентированного автоматного программирования также требуют разработки методов повышения качества для автоматных частей таких программ.

Изложенное позволяет утверждать, что результаты выполнения научно-исследовательской работы будут превышать мировой уровень разработок в рассматриваемой области.

1. АНАЛИТИЧЕСКИЙ ОБЗОР

В настоящей главе приводятся результаты аналитического обзора. Аналитический обзор проводился по следующим направлениям:

- качество программ;
- автоматное программирование;
- функциональное программирование;
- объектно-ориентированное программирование;
- автоматное объектно-ориентированное программирование;
- методы повышения качества объектно-ориентированных программ;
- методы повышения качества автоматных объектно-ориентированных программ;
- обзор методов и средств верификации на модели.

1.1. Качество программ

В настоящее время актуальной является проблема разработки качественного сложного программного обеспечения (ПО). При разработке такого ПО важно учитывать различные аспекты качества [1], важнейшим из которых является *корректность*: соответствие реализации программы заданной спецификации. Если система не делает того, что она должна делать, то все остальное – ее быстродействие, хороший пользовательский интерфейс – не имеет особого значения.

Все еще слишком трудно создавать ПО без ошибок, и слишком сложно исправлять ошибки, когда они появляются. Разновидности технических приемов для улучшения корректности и устойчивости одни и те же: формальные спецификации; встроенный контроль в течение всего процесса построения ПО (не просто испытания и отладка после создания); более совершенные языковые механизмы; обеспечение возможности разработчикам устанавливать требования корректности и устойчивости в сочетании с возможностью инструментов обнаруживать случаи несостоятельности до того, как они приведут к ошибкам.

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 измененной спецификации и улучшения читаемости.

В настоящее время программные системы используются повсюду: в медицине, транспорте, бытовой технике, космических полетах. Таким образом, все больше ответственности возлагается на программы, управляющие теми или иными устройствами. В результате постоянно увеличивающегося объема автоматизации в жизни человека, все более острой становится проблема обнаружения и устранения ошибок в управляющих программах. И чем раньше ошибки будут найдены, тем меньше вреда они принесут.

Для проверки работы управляющих программ используются следующие методы:

- *тестирование*;
- *имитационное моделирование*;
- *дедуктивный анализ*;
- *верификация на модели*.

Тестирование является самым простым и распространенным методом проверки работы систем. Общий принцип тестирования заключается в том, чтобы работающей системе подавать на вход определенные входные значения, и проверять, что на выходе получаются требуемые выходные значения. Положительными качествами тестирования являются, во-первых, простота, а, во-вторых, надежность, так как тестируется обычно сама работающая система. Поэтому гарантируется, что если

тесты выполнялись при тестировании, то они будут выполняться и при реальной работе. Недостатком тестирования является неполнота. Тесты проверяют функциональность системы лишь на некоторых примерах, что, естественно, не гарантирует, что система будет работать на *всех* примерах. С увеличением числа тестов можно быть достаточно уверенным в корректности работы программы. Однако гарантии корректности при этом не получить. Обычно стремятся, чтобы тесты, по крайней мере, покрывали все переходы в спроектированной системе. Например, желательно, чтобы каждая строка кода выполнялась хотя бы в одном тесте. Для эффективного тестирования создана технология программирования, называемая *Test Driven Development*, и существуют утилиты, указывающие на части кода, которые не выполняются ни при одном тесте. При этом улучшается качество тестирования, однако не решают проблему проверки правильности работы управляющей системы принципиально.

Имитационное моделирование сходно с тестированием, однако проверяется не работающая система, а модель, имитирующая ее работу. Имитационное моделирование используется в тех случаях, когда тестирование реальной системы не представляется возможным или требуется проверить корректность проекта до создания прототипа. Таким образом, при неточной модели возможно возникновение ситуации, когда некоторый тест прошел этап имитационного тестирования, но на реальной системе не выполнится. Поэтому в этом методе очень важно точно перенести логику работы системы в ее модель.

Дедуктивный анализ – это формальное доказательство свойств системы. Для управляющей системы строится набор аксиом, из которых затем с помощью формальной логики пытаются доказать выполнимость этих свойств. Преимущество дедуктивного анализа в том, что в случае успешного доказательства можно с точностью утверждать, что свойство выполняется *всегда*. При этом система может иметь даже бесконечное число состояний. Недостаток дедуктивного анализа состоит в том, что он требует большой ручной работы и высокой квалифицированности специалистов, его применяющих. При этом сложно заранее определить, сколько времени потребуется для доказательства утверждения или для его опровержения. Поэтому дедуктивный анализ применяется крайне редко и только в тех областях, в которых действительно критически важна корректность управляющей системы. В них, поочередно применяя те или иные правила вывода, пытаются из аксиом вывести заданное предположение. Однако обычно количество утверждений, выводимых из заданных аксиом, слишком велико, для того, чтобы найти среди этих утверждений требуемое. Поэтому человеку приходится «помогать» средству автоматического доказательства теорем (при его наличии), подсказывая, какое правило применить на очередном шаге. Таким образом, человек должен представлять, как доказывается заданное утверждение. Поэтому такой процесс доказательства нельзя назвать автоматическим. Необходимость в больших человеческих и временных ресурсах делает дедуктивный анализ неприменимым для большинства прикладных задач.

Верификации на модели является практически полностью автоматическим методом для проверки свойств систем с *конечным* числом состояний. Как следует из названия метода, он работает не с реальной системой, а с ее *моделью*. Для проверяемой системы сначала строится формальная модель, описывающая ее поведение. Затем для нее формулируется спецификация – утверждения, истинность которых требуется проверить. После этого выполняется автоматическая верификация, в результате которой либо доказывается, что модель удовлетворяет спецификации, либо это опровергается. Опровержение представляет собой набор действий над моделью, которые приводят нарушению спецификации.

Сравним верификацию на модели с другими методами. Верификация на модели лучше тестирования, во-первых, тем, что проверяет не просто некоторый набор пар вход-выход, а все возможные варианты входных данных. Еще одно преимущество метода верификации на модели по сравнению с тестированием состоит в том, что в случае невыполнения спецификации выводится сценарий ее нарушения, анализируя который проще найти причину ошибки, чем в случае простого ответа «нарушается». Отметим, что нарушение сценария может возникать не только в случае ошибочной модели

системы, но также в случае неверной спецификации. Оба типа ошибок легко находятся с помощью анализа сценария. Стоит напомнить, что процесс отладки систем итеративен: после обнаружения ошибки требуется ее исправить, и после этого все заново проверить. Автоматизация проверки на порядки ускоряет этот процесс.

Если сравнивать метод верификации на модели и дедуктивный анализ, то верификация на модели уступает тем, что применима только к моделям с конечным числом состояний. Тем не менее, существует большое число задач, для которых это ограничение выполняется. Кроме того, многие задачи с бесконечным числом состояний можно свести к конечным, поставив определенные ограничения, которые не сильно изменяют смысл задачи. Преимущество метода верификации на модели в том, что проверка проходит полностью автоматически и для ее выполнения не требуется особых знаний, опыта и времени. За счет этого верификация на модели применима в гораздо большем наборе областей, где используются управляющие системы, нежели дедуктивный анализ.

1.2. Автоматные модели управляющих программ

В автоматном подходе к программированию выделяются источники входных воздействий и автоматизированные объекты, каждый из которых содержит систему управления (СУ), в качестве которой применяется система взаимодействующих конечных автоматов, и объект управления (ОУ). Объект управления реализует выходные воздействия системы управления, а также формирует входные воздействия, реализующие обратную связь объекта управления с системой управления.

Входные воздействия разделяются на события, действующие кратковременно, и входные переменные, вводимые путем опроса. Входные воздействия целесообразно реализовывать в виде входных переменных, а применять события – для сокращения времени реакции системы. При этом одно и то же входное воздействие может быть одновременно представлено и событием, и входной переменной. Группы входных и выходных воздействий в общем случае связываются с состояниями, выделяемыми в каждом автомате.

На рис. 1 изображена схема автоматизированного объекта. Парадигма автоматного программирования состоит в представлении программ как систем автоматизированных объектов.

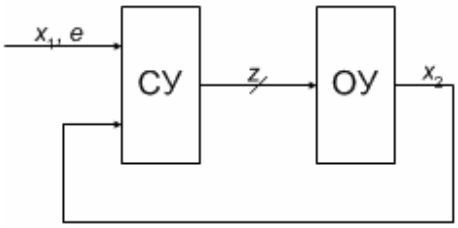
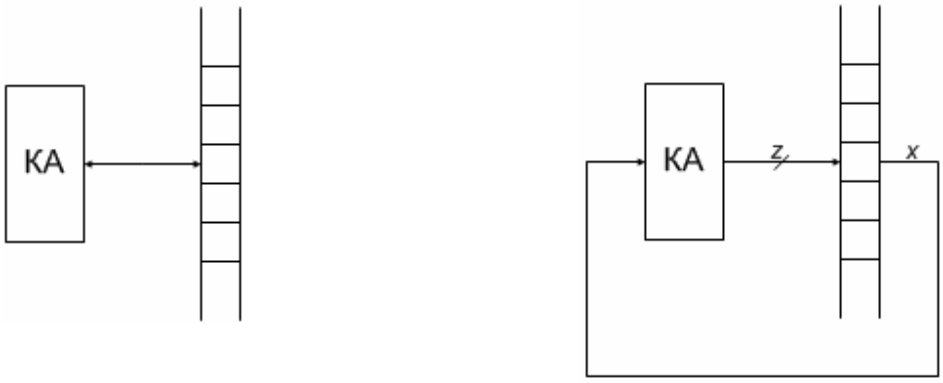


Рис. 1. Схема автоматизированного объекта

Данный подход является расширением машины Тьюринга, изображенной на рис. 2, на которой можно реализовать любой алгоритм.



а б

Рис. 2. Машина Тьюринга: классическое (а) и модифицированное (б) изображения

Базовым понятием автоматного программирования является «*состояние*». Все состояния в автоматах должны быть явно выделены. Состояния также бывают двух типов: управляющие (автоматные) и вычислительные (неавтоматные). Управляющие состояния отражают качественные особенности поведения, вычислительные – количественные.

В машине Тьюринга управляющее устройство с небольшим числом состояний, представляющее собой конечный автомат, может управлять практически бесконечным числом состояний на ленте. В технологии автоматного программирования основное внимание уделяется управляющим состояниям. Управляющие состояния выделяются и перечисляются явно, и в дальнейшем при использовании термина «состояние» понимается управляющее состояние.

Входные воздействия рассматриваются как средства для изменения состояний. Выходные воздействия автоматы могут формировать как в состояниях, так и на переходах между состояниями. Автоматы делятся на два типа: абстрактные и структурные. В абстрактных автоматах входные и выходные воздействия формируются последовательно. В структурных – «параллельно». В автоматном программировании применяются структурные автоматы.

Абстрактные автоматы обычно используются при разработке систем генерации парсеров по контекстно-свободным грамматикам и регулярным выражениям. В событийных системах управления применяются структурные автоматы (рис. 3), ранее использовавшиеся в аппаратных реализациях.

Рис. 3. Структурный конечный автомат [2]

Правила, по которым происходит смена состояний, называют *функцией переходов* автомата. Правила формирования выходных воздействий – *функцией выходов* автомата.

В общем случае, входные воздействия бывают двух типов: события и входные переменные. В настоящей работе будут рассматриваться структурные автоматы, входные воздействия которых представлены только событиями.

Для графического описания функции переходов применяются *диаграммы состояний* (*диаграммы переходов*). Пример такой диаграммы представлен на рис. 4.

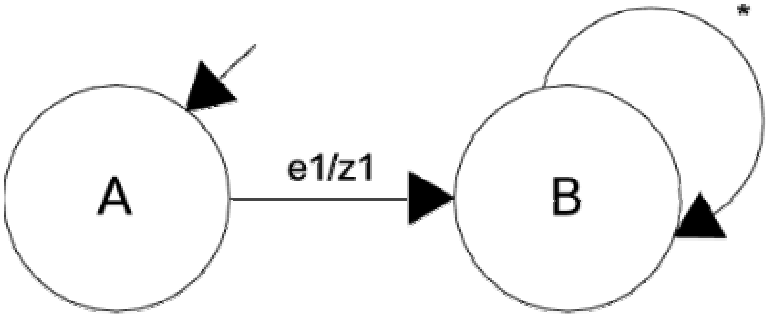


Рис. 4. Диаграмма переходов

Состояния автоматов на диаграмме переходов представлены окружностями, внутри которых находятся некоторые идентификаторы, отражающие предназначение каждого состояния. Переходы между состояниями обозначены стрелками. Рядом с каждой стрелкой перехода указано событие, по которому производится переход (например, «e1»), и, возможно, через знак «/» выходное воздействие (например, «z1»). Стрелки, события которых подписаны как «*», означают «переход по любому событию, для которого нет специальной стрелки». Кроме того, в настоящей работе будем обозначать группы похожих переходов как «*/z*», за подробным описанием таких групп следует обращаться к основному тексту настоящей работы или исходным кодам. Начальное состояние обозначается короткой стрелкой, входящей в данное состояние, но не выходящей не из какого другого.

В общем случае автоматы рассматриваются не изолированно, а как составные части взаимосвязанной системы – системы взаимосвязанных автоматов.

Автоматы между собой могут взаимодействовать тремя способами. При этом могут использоваться следующие типы взаимодействий.

- вложенность – один автомат вложен в одно или несколько состояний другого;
- вызываемость – один автомат вызывается другим автоматом;
- взаимодействие по номерам состояний – один автомат проверяет, в каком состоянии находится другой автомат.

Вложенность может рассматриваться как вызываемость с любым событием. Число автоматов, вложенных в состояние, не ограничено. Глубина вложенности также не ограничена.

Вложенные автоматы последовательно запускаются с передачей «текущего» события в соответствии с путем в схеме взаимодействия автоматов, определяемым их состояниями в момент запуска головного автомата. При этом последовательность запуска и завершения работы автоматов напоминает алгоритм поиска в глубину.

Вызываемые автоматы запускаются из выходных воздействий с передачей соответствующих «внутренних» событий. При этом автоматы могут запускаться однократно с передачей какого-либо события или многократно (в цикле) с передачей одного и того же события.

В случае, если один автомат вложен в другой, то диаграмма состояний вложенного автомата помещается в рамку внутри диаграммы внешнего автомата (рис. 5). Стрелкой из состояния внешнего автомата, указывающей на диаграмму переходов вложенного, будем обозначать передачу события.

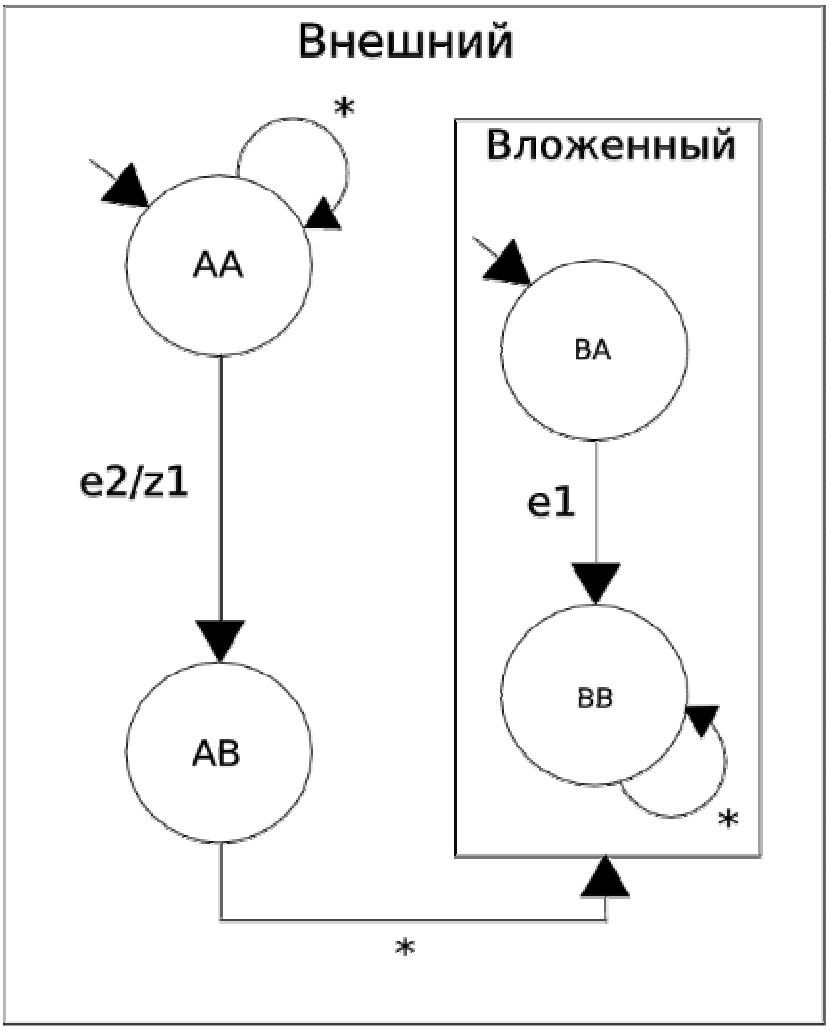


Рис. 5. Вложенные автоматы

Систему автоматов можно представить как единый автомат с помощью *прямого произведения* [3]. Состояниями *автомата-произведения C* двух автоматов *A* и *B* являются пары состояний автоматов *A* и *B*. Таким образом, число состояний автомата *C* равно произведению числа состояний автоматов *A* и *B*.

Для построения переходов в *автомате-произведении* требуется проследить «параллельную» работу автоматов *A* и *B*. Каждый из этих автоматов в зависимости от входных действий совершает переходы. При этом, если один из автоматов при получении некоторого входного воздействия не может перейти ни в какое состояние, автомат *C*, то также не сможет перейти ни в одно состояние.

1.3. Функциональное программирование

Главной моделью теории вычислительной сложности является *машина Тьюринга* [3] – вычислительный формализм, основным достоинством которого является простота описания и определенное сходство с реальными компьютерами.

Однако существуют и другие модели вычислений, одной из которых является *лямбда-исчисление* [4].

1.3.1. Лямбда-исчисление

В лямбда-исчислении имеется два примитива: *лямбда-абстракция* и *лямбда-аппликация* [4]. Все преобразования производятся при помощи единственной операции – *бета-редукции*. *Лямбда-термом* называют выражение, состоящее из лямбда-абстракций и лямбда-аппликаций, и возможно содержащее скобки. Лямбда-абстракция представляет собой выражение вида $\lambda x\,.\,y$, где x – переменная, а y – некоторый лямбда-терм. Лямбда-аппликация – выражение вида $f\,x$, где f – лямбда-абстракция, а x – переменная. Таким образом, лямбда-абстракция представляет собой функцию с одним аргументом.

Некоторое вхождение переменной v в лямбда-терм T называется *свободным*, если v не является одним из аргументов какой-либо лямбда-абстракции. Для вычисления значения лямбда-аппликации применяют операцию бета-редукции, заменяя в лямбда-абстракции все свободные вхождения переменной, имя которой стоит до точки, в лямбда-терме, который стоит после точки. Например, $(\lambda x\,.\,x\;x)\;y$ будет преобразовано в $y\;y$, но $(\lambda x\,.\,x\;(\lambda x\,.\,\mathrm{x}))\;y$ будет преобразовано в $y\;(\lambda x\,.\,\mathrm{x})$. Заметим, что функции нескольких аргументов можно представить в виде последовательности вложенных лямбда-абстракций (например, $\lambda x\,.\,\lambda \mathrm{y}\,.\,\mathrm{x}\;\mathrm{y}$). Поэтому для них принята сокращенная запись вида $\lambda x\;\mathrm{y}\,.\,\mathrm{x}\;\mathrm{y}$.

В чистом лямбда-исчислении весь процесс вычисления значения лямбда-терма заключается в произведении некоторой последовательности бета-редукций. Говорят, что лямбда-терм находится в *нормальной форме* [4], если над ним больше нельзя произвести ни одной бета-редукции.

Существуют лямбда-термы без нормальной формы (например, $(\lambda x\,.\,x\;x)\;(\lambda x\,.\,x\;x)$) и термы, достижимость нормальной формы которых зависит от последовательности редукций (например, $\$\,(\lambda x\,.\,y)\;((\lambda x\,.\,x\;x)\;(\lambda x\,.\,x\;x))$). Выражение, над которым можно произвести бета-редукцию, называется *редекс* [4].

По последовательности производимых редукций выделим два следующих порядка вычислений:

- аппликативный – первым производится редукция над самым правым из максимально вложенных редексов;
- нормальный – первым редуцируется самый левый редекс из минимально вложенных.

Теорема Чёрча-Россера, говорит о том, что если у лямбда-терма есть нормальная форма, то она единственна и достигается нормальным порядком редукций.

Известно, что лямбда-исчисление эквивалентно машине Тьюринга, а, следовательно, с его помощью можно представить все арифметические, логические и прочие операции, которые способен выполнить обычных компьютер [4].

1.3.2. Типизированное лямбда-исчисление

Если сопоставить каждому подтерму в лямбда-терме некоторый тип, то получим *типизированное лямбда-исчисление* по Черчу [4]. Тип «обычной переменной» принято обозначать одним словом (например, *int*), а тип лямбда-абстракции выражением вида $a \to b$, где a и b – тоже какие-то типы. Например, функция, принимающая два аргумента типов a и b и возвращающая значение типа c, будет иметь тип $a \to b \to c$, а функция, принимающая в качестве аргументов функцию, принимающую аргумент типа a и возвращающую значение типа b, и аргумент типа c, и возвращающая значение типа d, будет иметь тип $(a \to b) \to c \to d$.

Выражение считается корректно типизированным, если во всех лямбда-аппликациях у всех подставляемых значений типы совпадают с типами соответствующих аргументов лямбда-абстракций.

Введем возможность именования выражений при помощи конструкции *let*, определенной следующим образом: $let\ x = s\ in\ t$ равносильно $(\lambda x\,.\,t)\ s$. Например, $let\ x = 1\ in\ x + ((\lambda x\,.\,x)\ 2 * x)$ равносильно $1 + ((\lambda x\,.\,x)\ 2 * 1)$, а, следовательно, равносильно трем.

Известной особенностью типизированного лямбда-исчисления является то, что вычисление любого корректно типизованного лямбда-выражения всегда завершается, иначе говоря, нормальная форма достижима [4]. Поэтому типизированное лямбда-исчисление не является Тьюринг-полным. Для того, чтобы вернуть ему эквивалентность машине Тьюринга, вводят оператор рекурсии *Rec* с типом

$$((a \to b) \to (a \to b)) \to a \to b$$

и дополнительное правило редукции для рекурсивных функций:

$$\mathrm{Re}c\ F \mapsto F(\mathrm{Re}c\ F)\,.$$

Таким образом, расширенное рекурсией типизированное лямбда-исчисление, а, следовательно, и функциональные языки программирования, представляют собой Тьюринг-полный вычислительный формализм, в котором результат работы программы получают вычислением значения некоторой функции. Однако в таких языках отсутствуют циклы, оператор присваивания и переменные (в том смысле, в котором это слово употребляется в императивных языках программирования).

1.4. Реализация автоматов на функциональных языках программирования

Из изложенного следует, что при использовании чистых функциональных языков программирования (например, *Haskell* [5 – 7]) не удается непосредственно применять методы, используемые при реализации конечных автоматов на императивных языках программирования (например, *C++* [8]). В связи с тем, что состояние автомата, по сути – глобальная переменная, то обработка последовательностей событий в императивных языках производится при помощи циклов.

В настоящее время существуют системы, реализующие на функциональных языках программирования абстрактные конечные автоматы (например, [9, 10]), однако функции переходов в таких реализациях строятся не по изоморфным диаграммам состояний, а по множествам порождающих правил контекстно-свободных грамматик. В свою очередь, реализации на императивных языках программирования, построенные по диаграммам состояний, обычно валидируются сторонними утилитами, а не компилятором.

1.5. Структура объектно-ориентированных программ

Большинство разрабатываемых в настоящее время программ строится на основе объектно-ориентированного подхода. Известны различные методы объектно-ориентированного программирования [11]. Широко используется модель [1], 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 объединяет в себе данные и исполняемый код, предназначенные для выполнения его роли. При этом они являются скрытыми для других объектов программы, которые могут взаимодействовать с этим объектом только через операции, определенных АТД.

Классы. Классы, с одной стороны, представляют собой единственный вид модуля в объектно-ориентированных программах, а с другой – тип, реализующий АТД. Классы могут быть *абстракт-*

ными. 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. Будучи типом, класс определяет набор операций, соответствующих объекту. Вся объектно-ориентированная программа 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, это обеспечивает возможность создания модифицированных копий модулей без изменения их исходных определений, а, с другой стороны, наследование дает 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* [12]. В хорошо спроектированных программах даже самая сложная реализация скрыта от использующих ее объектов простым интерфейсом.

1.6. Методы повышения качества объектно-ориентированных программ

Компьютерные программы играют все большую роль. При этом возрастает значение качества ПО.

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

Частые изменения и уточнение требований к программным продуктам в процессе разработки вызвано сложностью современных программ, большинство деталей которых невозможно проработать заранее. Другой важной причиной является высокая скорость изменения технологий [13]. Технологии, используемые при разработке современного ПО, рассчитаны на высокую степень изменения требований. Рассмотрим некоторые из них.

Гибкие методологии разработки. В настоящее время наблюдается значительное распространение *гибких* (*agile*) методологий разработки ПО. Такие методологии (например, *XP* [14], *Scrum* [15] и *MSF for Agile Software Development Process* [16]) ориентированы на постепенную проработку 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. По результатам итерации и анализа получившейся реализации требования могут изменяться. 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 важных требований к архитектуре объектно-ориентированной программы является ее устойчивость к изменениям. Программа должна быть спроектирована так, чтобы большинство изменений в требованиях вызывало минимум модификаций ее реализации. Вторым важным требованием к хорошей объектно-ориентированной архитектуре является возможность повторного использования классов.

Хорошие решения в объектно-ориентированных программах фиксируются в виде *паттернов* [17, 18].

Среди паттернов объектно-ориентированного проектирования особое значение представляют следующие [12]:

- *информационный эксперт* определяет базовый принцип назначения обязанностей. Этот паттерн устанавливает, что обязанности должны быть назначены объекту, который владеет максимумом необходимой информации для выполнения обязанности;
- *слабая связанность* устанавливает следующие свойства: малое число зависимостей между классами и подпрограммами, слабую зависимость одного класса (подпрограммы) от изменений в другом классе (подпрограмме), высокую степень повторного использования подпрограмм;
- *сильное сцепление* определяет свойство связанности сервисов, предоставляемых одним классом (подпрограммой).

Таким образом, при проектировании объектно-ориентированной программы схожие обязанности группируются локально в 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 наиболее распространенных способов изменения кода программы является *рефакторинг* – процесс полного или частичного преобразования структуры программы при сохранении ее поведения [19]. 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 рефакторингов позволяет организовать процесс изменения поведения программы следующим образом: сначала производится серия рефакторингов для подготовки к изменению, затем выполняется само изменение поведения. При этом последовательность рефакторингов выбирается таким образом, чтобы минимизировать модификации кода, изменяющие поведение программы.

Тестирование. Тестированием программного обеспечения называется процесс выявления в нем ошибок. Важно понимать, 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!»). Виды тестирования программного обеспечения чрезвычайно многообразны [20]. Рассмотрим несколько основных.

При тестировании ПО тестер 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 из 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.

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;
- проверка, не привело ли исправление ошибки к появлению других ошибок: 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 [21], в которых программа типов накладывает значительные ограничения на код программы.

В современных проектах по разработке программного обеспечения особое место занимает *модульное тестирование* (*unit testing*) [20, 22]. Оно используется для автоматического тестирования отдельных модулей программы. При этом для каждой нетривиальной функции создается модульный тест, проверяющий основные сценарии работы с ней. При этом используется структурный подход: учитываются критерии охвата модульных тестов [23]. Для запуска и написания таких тестов применяются специальные библиотеки, существующие для всех популярных языков программирования [24], например, *JUnit* [25], *NUnit* [26], *PyUnit* [27] и т.д.

Модульные тесты запускаются программистами после каждого значительного изменения кода или рефакторинга. Удобство и широкая распространенность модульных тестов связана с тем, что 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*) [27].

Программирование по контракту (design by contract). Этот подход к объектно-ориентированному программированию был предложен Бертраном Мейером [1]. Основным компонентом этого подхода является использование *утверждений* (*assertions*) – условий, которые должны выполняться, если код программы верен. Утверждения синтаксически представляют собой булевы выражения с некоторыми расширениями и делятся на два типа:

- утверждения, устанавливающие *контракт* между классом и его клиентами. Утверждения этого типа являются реализацией *спецификации поведения интерфейса* (*behavorial interface specification, BIS*) [29];
- утверждения, выполняющие промежуточные проверки в коде реализации алгоритмов, фактически являются исполнимыми комментариями.

Программирование по контракту представляет собой развитие первого типа утверждений. Утверждения использовались и до появления контрактного программирования для проверки корректности входных и выходных данных, но эти утверждения были доступны только самому классу, в котором они были написаны. Клиенты класса получали информацию об утверждениях только через неисполняемые комментарии к классу.

В контрактном программировании предлагается ввести 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 контракту 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. Для удобства реализации этого свойства требуется расширение языка программирования конструкциями, которые позволяют записывать контракты как часть интерфейса класса. На настоящее время самым распространенным языком программирования с поддержкой контрактов является язык *Eiffel* [1], в котором программирование по контракту и было впервые реализовано. Существуют также библиотеки расширений, реализующие полную или частичную поддержку контрактного программирования в распространенных языках программирования [30, 31]. Однако, так как эти библиотеки не являются встроенной возможностью соответствующего языка, они не получили широкого распространения.

Реализация контрактного программирования в языке *Eiffel* предусматривает три возможности использования контрактов:

- контракты можно использовать как неисполнимые комментарии с проверкой корректности выражений. При этом проверка соответствия реализации программы описанным контрактам не осуществляется;
- можно статически верифицировать программу на соответствие контрактам, однако этот подход весьма сложен и его практическая реализация обычно требует введения дополнительных языковых конструкций [32];
- можно 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 проверок во время выполнения 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.

Общей чертой рассмотренных выше технологий 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 частей кода программы и относящихся к ним частей спецификации. Формализованность – представление спецификации в формальном виде. Это повышает возможности автоматической обработки спецификации. При этом дополнительным достоинством является единство представления кода программы и формализованной спецификации.

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, спецификация (тесты) автоматически синхронизируется с кодом программы.

Исполнимость означает возможность автоматической *верификации программы* – проверки соответствия программы формализованной спецификации. Это свойство позволяет значительно ускорить верификацию, а, следовательно, и упростить изменение программного кода.

1.7. Автоматное объектно-ориентированное программирование

Одним из подходов при проектировании программ со сложным поведением является автоматное программирование (программирование с явным выделением состояний) [33]. 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 с процедурным программированием, а затем был расширен и для объектно-ориентированного программирования [34].

В настоящее время предложены, исследованы и применяются различные автоматные модели [2]. Будем рассматривать автоматы, поведение которых описывается графами переходов со следующими свойствами [35, 36]:

- 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, кроме события, может содержать 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 изложенном описании отсутствует взаимодействие нескольких автоматов. Это связано с тем, что при подходе «автоматизированные объекты управления как классы» [2] 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, например, применяемый в инструментальном средстве *UniMod*, в которых используются системы автоматов.

Одной из основных проблем программирования с явным выделением состояний является организация взаимодействия автоматного и неавтоматного кода. В настоящее время разработан ряд

подходов [37] 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* [38] или последовательности сообщений, посланных неавтоматной частью программы [39].

Смешанным можно считать декларативный подход к объявлению структуры автоматов в объектно-ориентированных программах [40]: в нем 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 [2] 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 такого 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 [41] 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 [42] 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 [41] для решения этих проблем предлагается использовать текстовый предметно-ориентированный язык (*DSL*) *stateMachine*, реализованный в программе метапрограммирования *JetBrains MPS (Meta Programming System)* [42, 43]. 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 автомате 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 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 [1]). Другой особен-

ностью 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, становится возможным разработать автоматный *внутренний предметно-ориентированный язык* [44]. Одной из первых реализаций такого языка является библиотека *STROBE* [45] для языка программирования *Ruby*. Это направление получило развитие в работе [46], в которой поддержаны наследование и вложение 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 диаграммами переходов.

Еще одной проблемой, недостаточно исследованной в существующих методах проектирования и реализации автоматных программ, является модификация программ при изменении требований. Как было отмечено выше, для решения этой проблемы в объектно-ориентированном программировании используются рефакторинги. В настоящее время вопрос рефакторинга автоматных программ не исследован.

1.8. 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. 1.6, 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 автоматов, оно разделяется между ними таким образом, чтобы поведение каждого автомата было обозримым 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. 1.6.

Важной особенностью тестирования автоматных программ является возможность автоматической генерации тестов по структуре автомата и формальной спецификации [47]. 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 влияет на работу автомата. Существование подобных избыточных состояний 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 неконечном состоянии дизъюнкция пометок переходов, исходящих из состояния, должна быть истинна. Если это свойство не выполняется, то 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* [48, 49].

Наибольший интерес представляет *третий уровень* статической проверки – верификация (формальная проверка соответствия автомата спецификации). В рамках верификации выделяются два основных подхода: *доказательная верификация* [50, 51] и *верификация на модели* [52].

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 [53]. 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 [54, 55], 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 [56 – 58]. 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 раздельно 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.

Инструментальные средства, реализующие метод верификации на модели, называются *верификаторами*. Существует большое число верификаторов, у каждого из которых свои особенности. Однако, в общем случае процесс верификации можно разбить на три этапа: моделирование, спецификация и верификация.

Моделирование – это построение модели системы, или ее формализация для того, чтобы верификатор смог с ней работать. Для формализации каждый верификатор имеет свой *входной язык*. Входные языки верификаторов предназначены для того, чтобы проектирующий систему человек сам выделил важные детали поведения и абстрагировался от неважных деталей реализации. Языки верификаторов различаются по своей выразительной способности, удобству и уровням абстракции. Существуют верификаторы, которые принимают модель в виде программы на языке, похожем на обычный язык программирования. Примером такого верификатора является верификатор *SPIN*. Синтаксис его входного языка *Promela* намеренно создан похожим на синтаксис языка программирования *C*, для того, чтобы было проще переносить логику работы программы, написанной на языке *C*, в модель верификатора.

Существует верификатор *SMV*, который принимает модель в виде раскрашенной *сети Петри*. Такой входной язык удобен для моделирования, например, бизнес-процессов.

Известен также верификатор *Bogor* с гибким входным языком. В нем можно создавать собственные абстракции, а также писать код как на языке высокого уровня, похожем на язык программирования *Java*, так и на языке низкого уровня, описывая каждое состояние системы.

Такое разнообразие входных языков верификаторов связано с тем, что определенные входные языки оказываются более удобными для соответствующих типов систем. Этап моделирования системы очень важен, так как неверная или неполная модель может привести к ошибкам. В результате неточности моделирования верификатор может найти в модели ошибки, которые не существуют в исходной системе, или, что еще хуже, провести успешную верификацию, в то время, как исходная система не удовлетворяет предъявленным требованиям.

В общем случае полученную модель верификатор разбивает на элементарные состояния и строит граф переходов между ними. Этот граф называется *моделью Крипке*. В первых верификаторах он строился явно как набор вершин (элементарных состояний) и переходов между ними, но в современных верификаторах для экономии памяти такой граф не строится явно. От размеров построенного графа зависит время верификации, и в случае очень большого числа элементарных состояний модели, верификация может даже оказаться невозможной из-за нехватки памяти или огромного времени, требующегося для верификации. Поэтому верификаторы не принимают модель в виде программы на языке программирования, таком как, например, *Java*. Такие программы оказываются слишком под-

робными, и если их разбивать их элементарные состояния, то полученная модель Крипке в большинстве случаев окажется необозримой. Тем не менее, в данном направлении ведутся активные исследования.

Спецификация – это процесс формирования требований к модели, выполнимость которых верификатор должен проверить. Во-первых, требования можно формировать, в виде элементарных проверок прямо в коде модели (*assertion*). С помощью них можно проверить, например, что в неком состоянии некоторая переменная всегда принимает заданное значение. Такие требования задают ограничения на элементарные состояния системы. Большинство верификаторов позволяют формулировать утверждения в виде формул *темпоральной логики*. Такие формулы задают ограничения на поведение системы во времени, что делает ограничения гораздо более выразительными, чем элементарные проверки. Например, можно сформулировать утверждение вида «если было выполнено условие 1, то *когда-нибудь в будущем* выполнится условие 2». Такие утверждения, вообще говоря, описывают бесконечные истории работы модели, и возможность их проверки основывается на предположении о конечности числа элементарных состояний модели.

Верификация производится с помощью специальных алгоритмов, которые проверяют, что построенная модель Крипке удовлетворяет заданной темпоральной формуле. В результате работы такого алгоритма либо доказывается, что для любой истории формула выполняется, либо находится опровержение – история (набор переходов в модели Крипке), которая приводит к нарушению темпоральной формулы. Отметим, что перед верификатором стоит также задача трансляции полученного набора переходов в модели Крипке в термины исходной модели, написанной на входном языке верификатора. Только после этого пользователь верификатора сможет увидеть сценарий в построенной им модели, на котором нарушается свойство, и исправить модель или спецификацию.

1.9. Обзор методов верификации на модели

Существует несколько типов темпоральных логик. При этом в верификации наиболее распространены два из них: *CTL* – логика ветвящегося времени и *LTL* – логика линейного времени.

Алгоритм автоматической проверки темпоральных утверждений на модели был впервые предложен в 80-х годах Э. Кларком и Э. Эмерсоном. Они построили алгоритм верификации формул логики *CTL*, который имеет полиномиальную сложность относительно размеров модели Крипке и длины темпоральной формулы. Позднее этот алгоритм был улучшен до линейной сложности относительно произведения длины формулы на размер модели Крипке [59] и был реализован в верификаторе *EMC*. Этот верификатор может проверять модели Крипке, содержащие большое число состояний.

В 1985 году был создан первый алгоритм проверки формул *LTL*, основанный на табличном методе. Со временем улучшались как алгоритмы проверки темпоральных формул на модели, так и компьютерная техника. В результате размеры моделей, с которыми могли работать верификаторы, постоянно увеличивались.

Кроме темпоральных формул, было предложено задавать спецификации в виде автоматов. С помощью несложных преобразований можно получить из модели Крипке автомат, и тогда получается, что и спецификация и модель могут задаваться одной математической моделью, что дает возможность применять результаты из теории автоматов. В 1983 году М. Варди и П. Вольпером было показано, что любую формулу логики *LTL* можно преобразовать в эквивалентный автомат Бюхи – один из простейших конечных недетерминированных автоматов над бесконечными словами [60]. Эквивалентность означает, что любая история, на которой выполняется исходная темпоральная формула, допускается также и построенным автоматом Бюхи, и наоборот. Тогда для доказательства выполнения исходной формулы на исходной модели Крипке требуется доказать включение языка автомата, построенного из модели Крипке, в язык автомата Бюхи, построенного по темпоральной формуле.

Подход М. Варди и П. Вольпера к проверке на модели поставил две задачи: преобразование формулы в автомат и проверка пустоты автомата.

Первую задачу решает алгоритм преобразования *LTL*-формулы в автомат, предложенный в работе [61]. Алгоритм предназначен для осуществления проверки «на лету», следовательно, процесс генерации автомата-спецификации может идти во время генерации модели и быть в процессе скорректированным необходимым образом. Алгоритм дает на выходе обобщенный автомат Бюхи, который может быть преобразован в классический автомат Бюхи.

Для решения второй задачи группой ученых [62] был предложен алгоритм поиска цикла с допускающим состоянием, достижимым из начального, который для графа размера n требует $O(n)$ памяти с произвольным доступом. Таким образом, это позволяет делать проверку пустоты пересечения двух автоматов Бюхи с использованием линейной памяти. Предложенный алгоритм использует два поиска в глубину. Так как нет необходимости держать в памяти весь исследуемый автомат, то метод можно использовать для верификации «на лету». Позднее М. Варди и П. Вольпер доказали [60], что проверка пустоты автомата Бюхи *NLOGSPACE*-полна [63].

Для проверки формул логик ветвящегося времени используются автоматы на бесконечных деревьях (*automata on infinite trees*). Например, М. Варди и П. Вольпер в статье [64] использовали их для верификации формул логики *ADPDL* (вариант *DPDL, Deterministic Propositional Dynamic Logic*).

Главной проблемой верификации является так называемая проблема «комбинаторного взрыва». Она связана с тем, что при наличии в системе параллельных процессов, число элементарных состояний системы растет экспоненциально от числа элементарных состояний каждого процесса. Это происходит из-за того, что в каждый момент времени управление может передаться в другой процесс, который осуществит свой очередной переход. В результате, верификатору приходится перебирать все возможные цепочки переходов. Поэтому даже несложные на первый взгляд модели могут оказаться слишком сложными для верификации.

В 1986 году Р. Бриан в работе [65] предложил новое компактное представление булевых функций в виде ациклических ориентированных графов и алгоритмы для работы с ними. Это представление является канонической формой — у каждой функции есть свой уникальный граф, причем размер этого графа минимален. В работе Р. Бриана были изложены алгоритмы для работы с новым представлением, такие как композиция, подстановка значения вместо одного из аргументов и т.д.

На основе представления Р. Бриана, названного упорядоченные двоичные разрешающие диаграммы (*Ordered Binary Decision Diagram*, *OBDD*), К. МакМиллан и соавторы в работе [66] разработали методику неявного представления пространства состояний вместо списков смежности. Каждое состояние системы может быть закодировано как вектор булевых переменных. При этом отношение между двумя состояниями можно представить в виде булевой функции. *OBDD* не всегда решают проблему комбинаторного взрыва, но многие реальные системы могут быть верифицированы с их помощью. В работе [66] говорилось о верификации систем с большим числом состояний. Был описан алгоритм символьной проверки на модели для формул μ-исчисления, а также методы трансляции в μ-исчисление формулы логик *CTL* и *PTL*. Исследователи продемонстрировали применения своего алгоритма для проверки выполнимости формул логики линейного времени, сильной и слабой эквивалентности конечных систем переходов, и включения языков для конечных ω-автоматов. Они также провели верификацию простой синхронной конвейерной схемы.

К. МакМиллан разработал систему верификации *SMV*, которую описал в своей диссертации [67]. Входной язык *SMV* предназначен для описания модульных иерархических параллельных систем с конечным числом состояний, как синхронных, так и асинхронных. Язык допускает булев и перечислимые типы данных. Программа на этом языке может быть снабжена спецификациями в виде формул языка *CTL*. Верификатор строит по программе модель и, используя поиск в пространстве состояний, проверяет ее на соответствие спецификации. Если модель не соответствует спецификации, то система выдает контрпример в виде трассы выполнения. Протокол для согласования кеша распределенной памяти мультипроцессора *Encore Gigamax* был смоделирован на языке *SMV* и проверен с использованием символьной верификации в комбинации с индукцией. Система была представлена как асин-

хронная композиция синхронных конечных автоматов. Для упрощения верификации была использована абстракция. При этом не был точно воспроизведен механизм замены кеша, что добавило еще больше недетерминизма. За несколько минут было найдены ошибки, которые до этого не были обнаружены другими методами из-за большого размера пространства состояний модели.

В диссертации обсуждались методики символьной проверки на модели, часть из которых была реализована в системе *SMV*. Техника поиска в пространстве состояний при помощи *OBDD* может быть использована и при проверки на модели формул логики *CTL*. В работе также описан класс схем, для которых графы переходов могут быть эффективно представлены в виде *OBDD*. Диссертация также содержит альтернативный символьному подход к проблеме «комбинаторного взрыва», основанный на развертывании сетей Петри в специальную структуру – «сеть расширения» (*occurrence net*).

В работах [68, 69] описана улучшенная методика символьной проверки, которая позволяет верифицировать системы с большим числом состояний, чем позволяет метод, описанный в работе [70]. Она основана на так называемых *группированных отношениях перехода* (*partitioned transition relations*). Основная идея состоит в том, чтобы представлять отношение перехода не в виде одной монолитной функции, а в виде множества функций, по одной для каждой переменной состояния. Часто имеет смысл объединять некоторые из этих функций в конъюнкции или дизъюнкции. Методика подходит как для синхронных, так и для асинхронных схем. Время, затрачиваемое на верификацию, зависит полиномиально от размера схемы.

В 1992 году в университете Карнеги-Меллона [71] построили точную модель протокола согласования кэша, описанного в черновом варианте стандарта *IEEE Futurebus+* (стандарт *IEEE 896.1-1991*) и проверили ее на соответствие спецификации. Для формализации и верификации была использована система *SMV*. Модель протокола на языке *SMV* состояла из 2300 строк кода, не считая комментариев. Некоторые детали протокола были скрыты из-за того, что не были еще четко прописаны в стандарте (были помечены словом «возможно»). Исследователи отметили, что одной из самых полезных частей проекта является модель мостов шины, которые соединяют их в иерархических конфигурациях системы. Эти компоненты не были описаны в стандарте детально, но без их моделирования невозможно анализировать иерархические конфигурации, так как именно в них возникает наиболее сложное поведение. С использованием *SMV* и модели мостов стало возможным найти некоторые потенциальные ошибки в иерархическом протоколе. Также была найдена ошибка и в протоколе для одной шины.

Опишем, как с помощью этих диаграмм можно представлять модели Крипке. *OBDD* используются для кодирования функций, зависящих от набора булевых переменных. Для того, чтобы перейти к функциям от переменных, принимающих произвольный конечный набор значений, используется следующая простая идея: для такой функции можно каждое значение переменной представить в виде набора бит (двоичного числа). Затем можно соединить булевы представления всех переменных и рассматривать каждый бит как новую булеву переменную. Таким образом, будет получена функция от некоторого множества булевых переменных, которую можно будет представить при помощи указанной диаграммы. Далее, для кодирования множества элементов при помощи *OBDD* строится функция, которая в зависимости от элемента возвращает значение `true`, если элемент принадлежит множеству, и значение `false` – в противном случае. Таким образом, модель Крипке можно закодировать следующим образом: построить *OBDD*, кодирующую множество состояний модели Крипке, а также построить *OBDD*, возвращающую по паре состояний значение `true`, если существует переход в модели Крипке из первого состояния во второе, и значение `false` – в противном случае.

Проблема комбинаторного взрыва особенно остро стоит в области верификации распределенных асинхронных систем. Если рассматривать все возможные последовательности событий в такой системе, то число состояний в модели растет экспоненциально. Однако исследования в этой области показали, что рассмотрение всех последовательностей выполнения не всегда является необходимым

для верификации. Действительно, в асинхронных системах для двух событий часто не имеет значения, в каком порядке они произошли. На этом базируются методики *редукции частичных порядков* (*partial order reduction*). Впервые они независимо появились в работах А. Валмари [72, 73, 74], П. Годфруа [75] и П. Вольпера [76].

При наличии параллельных процессов в верифицируемой модели, в элементарные состояния модели входят все возможные варианты последовательного выполнения этих процессов. В результате число состояний модели Крипке возрастает на порядки. Каждый процесс задает частичный порядок на события. При этом число вариантов выполнения программы – это число способов дополнить частичный порядок. Однако можно заметить, что обычно большинство различных последовательностей выполнения не различаются с точки зрения верифицируемой формулы, и перебирать их все нет необходимости. Например, если работают два параллельных процесса, порядок выполнения которых не влияет на результат программы, а в спецификации используется лишь этот результат, то нет смысла перебирать все возможные варианты последовательного выполнения, так как для них результат будет одинаков – система все равно придет в одно и то же глобальное состояние. Однако это неверно, например, если верифицируемая спецификация затрагивает порядок выполнения процессов. В этом случае может понадобиться перебрать все возможные варианты последовательного выполнения процессов. Таким образом, можно построить отношение эквивалентности между порядками выполнения, эквивалентными с точки зрения спецификации, и для каждого класса эквивалентности использовать лишь одного его представителя. В результате сильно уменьшится число состояний в модели Крипке. Поэтому память и время, необходимые для верификации, тоже уменьшатся.

Редукция частичных порядков основывается на понятиях устойчивых и спящих множеств.

Устойчивые множества (*persistent sets*) [77, 78] сначала использовались для определения взаимных блокировок (deadlock). Множество переходов *T*, которые возможных из состояния *s*, называется устойчивым, если для всех переходов не из множества *T* из *s* или из состояний, которые достижимы из состояния *s* по переходам не из множества *T*, верно, что они независимы с переходами из множества *T*. Можно показать, что в ходе поиска по графу переходов для каждого состояния достаточно обследовать только устойчивое множество переходов.

Спящие множества (*sleep sets*) [77, 75] определяются для каждого состояния, как множество переходов, которые не будут обследованы. Например, если в каком-либо состоянии *s* возможны два независимых перехода *a* и *b* и переход *b* ведет в состояние *t*, то зная, что переход *a* из *s* обследован, можно не исследовать его из состояния *t* (он относится к спящему множеству для состояния *t*), так как обе последовательности переходов *ab* и *ba* ведут в одно и то же состояние. Таким образом, спящие множества предназначены для того, чтобы обходить все достижимые состояния системы без перебора всех переходов.

Устойчивые множества и спящие множества, хотя и различаются между собой, основаны на сходных идеях. Отличный от них метод разработан Дж. Хольцманом и Д. Пеледом [79] и используется в системе верификации *SPIN* [80].

Кроме применения *OBDD* и редукции частичных порядков, выделяют также следующие методы, позволяющие применять верификацию на модели даже для систем, не поддающихся верификации напрямую:

- декомпозиция;
- абстракция;
- симметрия;
- индукция.

Декомпозиция применяется для верификации сложных систем, разделенных на модули. Вместо того чтобы верифицировать всю систему сразу, можно верифицировать каждый модуль в отдельности. При верификации каждого модуля делается предположения относительно других модулей, с ко-

торыми он взаимодействует. Затем проверяется, что сделанные предположения действительно выполняются.

Абстракция – это, пожалуй, самый эффективный способ упрощения задачи для верификатора. Суть этого метода состоит в абстрагировании от несущественных (как кажется пользователю) деталей реализации. За счет этого число состояний модели Крипке уменьшается. Здесь важно не потерять важные детали, которые могут повлиять на результат верификации. Некоторые верификаторы, такие, как, например, верификатор *Bogor*, предоставляют пользователю возможность создавать свои типы во входном языке. За счет этого может быть создана новая абстракция. Например, если в системе используется сущность «множество», в которую можно добавлять и удалять элементы, а также проверять их наличие, то запрограммировать такое, казалось бы, простое поведение будет достаточно сложно, если пользоваться только возможностями такого языка, как, например *Promela* (входной язык верификатора *SPIN*). В верификаторе *Bogor* можно один раз создать новый тип, который будет описывать новую сущность, и после этого свободно пользоваться им при формализации модели для верификатора.

Симметрия как метод упрощения верификации состоит в следующем наблюдении. Во многих системах содержатся одинаковые элементы, например, несколько одинаковых процессов. Такая симметрия может порождать одинаковые подграфы в модели Крипке. Этот факт можно использовать для редукции графа и за счет этого можно добиться сокращения числа состояний. Иногда можно преобразовать исходную симметричную модель так, что суть ее работы останется таже, но модель Крипке окажется проще.

Индукция позволяет верифицировать целые семейства систем с конечным числом состояний. Например, протокол взаимного исключения должен работать для произвольного числа процессов, что порождает бесконечное семейство конечных систем. Для верификации строится некая инвариантная система, представляющая работу произвольного элемента семейства систем. Верификация такой системы будет означать верификацию всего семейства исходных систем.

Методы проверки на модели, основанные на проверке выполнимости булевых формул (*SAT*), являются альтернативой подходу, использующему *OBDD*. Прогресс в методах решения задачи *SAT* [81 – 83] позволяет разработать эффективные алгоритмы верификации.

Ограниченная проверка на модели (*bounded model checking, BMC*) была предложена в университете Карнеги-Меллона [84]. Основная идея метода состоит в том, чтобы попытаться найти контрпример заданной длины *k*, сгенерировав формулу, которая выполняется только если такой контрпример существует. Формула является конъюнкцией трех групп условий: условия на начальные состояния, условия отношений переходов и условия на конечные состояния. Генерация формулы из спецификации на языке *LTL* может быть выполнена за полиномиальное время. Контрпример находится достаточно быстро и при этом имеет минимальную длину. Метод требует меньше памяти, чем алгоритмы с использованием *OBDD*. Проблема ограниченной проверки состоит в том, что она может установить только отсутствие контрпримера заданной длины, но не отсутствие контрпримера вообще.

В работе [85] были скомбинированы *BDD* и *SAT*, что позволило обрабатывать большие по размеру графы.

Метод неограниченной проверки на модели (*unbounded model checking*) разработанный К. МакМилланом [86] обобщает ограниченную проверку и интерполяцию. При этом предложенный алгоритм в цикле запускает процедуру ограниченной проверки для какой-то длины контрпримера. Если формула невыполнима (контрпример не найден), то она разбивается на две части: первая соответствует начальным условиям и первому условию перехода, а вторая – остальной формуле. Для двух частей считается интерполянт, который характеризует состояния, достижимые из начальных за один шаг, но из которых нельзя дойти до конечных состояний. Это новое множество используется для следующего запуска процедуры ограниченной проверки.

Способность решателя *SAT* выдавать доказательство того, почему формула невыполнима, в работе [87] используется в комбинации с *детализацией абстракции* (*abstraction refinement*). Предложенный метод верификации формул логики *LTL* чередует ограниченную проверку с неограниченной. На каждой итерации алгоритма запускается ограниченная проверка на модели *(BMC)* для некоторого значения *k*. Если он не находит контрпример, то, основываясь на выданном решателем *SAT* доказательстве невыполнимости, строится абстрактная модель, на которой запускается процедура обычной (неограниченной) проверки.

Еще один метод, использующий детализацию абстракции, был предложен Э. Кларком и соавторами в работе [88]. Алгоритм вначале запускает неограниченную проверку на абстрактной модели. Если свойство выполнено, то алгоритм завершается, а если оно не выполнено, то на основе контрпримера генерируется формула для решателя *SAT*. В случае существования переменных, для которых формула выполняется, строится контрпример на модели *SAT*. Если формула невыполнима, то на основе доказательства, выданного решателем *SAT*, абстракция уточняется. Это происходит за счет добавления в модель некоторых переменных, которые были спрятаны.

Еще одна интересная техника проверки на модели использует конъюнктивно-нормальные формы (*КНФ*). В работе [89] было показано, что, модифицировав некоторые *SAT*-алгоритмы, можно получить способ удаления квантора всеобщности из формулы, получив формулу в *КНФ*. Становится возможным преобразование *CTL*-формулы **AX** *p*, где *p* – булева формула, в эквивалентную формулу в *КНФ*. Таким образом, любая *CTL*-формула вычисляется с использованием неподвижных точек.

Более подробные обзоры методов верификации с использованием *SAT* можно найти в работах [90, 91].

Наличие такого числа алгоритмов и приемов верификации на модели позволяет сделать вывод о перспективности данного подхода. Отметим, что если задача проверки семантических свойств программы алгоритмически неразрешима [3], то проверка свойств модели может быть произведена за конечное время. При этом благодаря появлению новых алгоритмов в области верификации на модели постоянно уменьшаются требования к объему памяти и времени, требующихся для верификации. Кроме того, возможность широкого выбора средств верификации и языков задания спецификации заметно расширяет возможности метода верификации на модели. Так, например, для описания простых свойств модели можно использовать простейшие темпоральные логики. Это позволит уменьшить время проверки данного свойства для конкретной модели. Так как автоматные модели являются весьма простыми и имеют конечное число состояний, то из изложенного следует, что верификацию таких моделей можно эффективно проводить на основе рассмотренных методов.

1.10. Обзор средств верификации

В данном разделе производится обзор инструментальных средств, позволяющих прямо или косвенно верифицировать автоматные программы.

1.10.1. Верификатор *SPIN*

В мире существует большое число верификаторов. Одним из самых известных, является верификатор *SPIN* [80].

Верификатор *SPIN* исходно был создан для верификации протоколов. Он позволяет создавать модели, содержащие несколько процессов, взаимодействующих с помощью глобальных переменных или с помощью каналов с ограниченным размером буфера.

Существует консольная версия верификатора *SPIN*. Однако обычно с ним работают через удобную для пользователя оболочку – *XSPIN*.

При использовании этого верификатора сначала разрабатывается модель программы на языке *Promela*. В моделях на этом языке создаются процессы (они характеризуются модификатором `proctype`), которые взаимодействуют между собой. В языке *Promela* существует несколько операторов недетерминированного выбора, что позволяет создавать недетерминированные модели. Затем формируется спецификация для модели при помощи формул темпоральной логики *LTL*. После этого запускается так называемый *имитационный режим* (*interactive simulation*). В нем написанная модель запускается со случайными значениями в недетерминированных условиях и проверяется выполнение спецификации. В таком режиме проверяется лишь один путь развития модели. Это предназначено не для верификации, а для проверки пользователем, что разработанная им модель делает именно то, что он хотел. Тем не менее, возможно, что принципиальные ошибки в модели или спецификации будут найдены уже на этом этапе. Для полной верификации по модели и спецификации генерируется специальная программа на языке *C*, которая производит полный перебор путей в пространстве состояний модели для поиска нарушений спецификации. В результате работы программы либо выводится сообщение об удачном завершении верификации, либо генерируется сценарий ошибки, который используется в имитационном режиме для представления истории ошибки. Схема работы верификатора изображена на рис. 6.

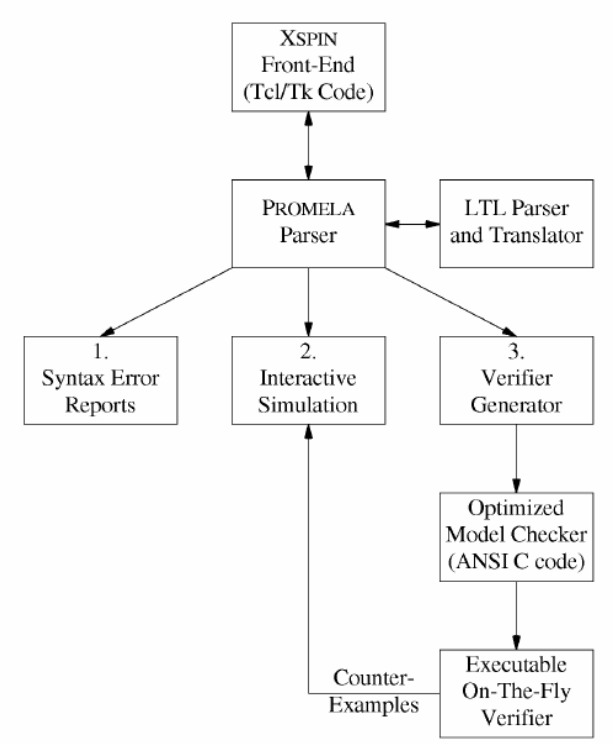


Рис. 6. Схема работы верификатора *SPIN*

Верификатор *SPIN* может работать со спецификациями заданными в виде автомата Бюхи. В случае если спецификация задана в виде *LTL*-формулы, запускается специальная утилита, которая преобразует ее в эквивалентный автомат Бюхи.

Автоматы Бюхи [92] – это одни из простейших конечных автоматов над бесконечными словами – ω-автоматов. Путь в автомате Бюхи называется допускающим, если он бесконечное число раз проходит через хотя бы одно из его допускающих состояний. Автомат Бюхи допускает слова, порождаемые допускающим путем. Оказывается, любую формулу в логике *LTL* можно преобразовать в недетерминированный автомат Бюхи [92]. Это означает, что множество путей, удовлетворяющих фор-

муле и допускаемых автоматом Бюхи, будут совпадать. К примеру, *LTL*-формула *FG p* («Существует состояние, после которого всегда будет выполняться *p*») преобразуется в автомат Бюхи, изображенный на рис. 7.

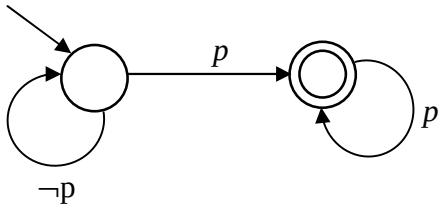


Рис. 7. Автомат Бюхи для формулы FG p

Этот автомат является недетерминированным, так как он недетерминированно выбирает момент, с которого условие *p* всегда будет выполняться, и переходит в допускающее состояние. Этот автомат не является неполным: при условии *¬p* из допускающего состояния нет перехода. Это следует трактовать как переход в сток – недопускающее состояние с циклом в себя. Таким образом, если в автомате не определен переход по очередному условию из текущего состояния, то слово считается недопускающим.

Модель Крипке, (граф элементарных переходов в исходной модели) достаточно просто можно преобразовать в ω-автомат. Языком полученного автомата будут пути в модели Крипке. Также известно, что можно построить ω-автомат, допускающий пересечение языков двух ω-автоматов [92]. Тогда возникает следующая идея: построить два автомата Бюхи: из *отрицания* исходной *LTL*-формулы и из модели Крипке и их перемножить. Тогда любое слово, допускаемое полученным автоматом, будет одновременно путем в модели Крипке, и не будет удовлетворять исходной *LTL*-формуле. Следовательно, это будет искомый контрпример – путь в модели Крипке, нарушающий свойство. Обратно, если контрпример существует, то он будет допускаться обоими автоматами Бюхи, а, следовательно, и их пересечением.

Эта идея используется во многих верификаторах *LTL*-формул, в том числе в верификаторе *SPIN*. Для ее реализации требуется:

- преобразовать отрицание *LTL*-формулы в автомат Бюхи;
- построить пересечение полученного автомата и модели Крипке (точнее, ее автоматного представления);
- найти допускающий путь в полученном пересечении или убедиться, что его не существует.

Для трансляции *LTL*-формулы в автомат Бюхи был предложен алгоритм [92], и в открытом доступе существуют программы, его реализующие (например, *LTL2ba*, который для верификатора *SPIN* строит автомат Бюхи на языке *Promela*).

Построение пересечения автомата Бюхи и модели Крипке можно делать неявно по правилам, изложенными ниже.

- Состояние пересечения – это комбинация состояния модели Крипке и состояния автомата Бюхи.
- Каждый шаг в пересечении совершается в два действия: сначала происходит переход в модели Крипке, а затем в автомате Бюхи. Таким образом, они работают «параллельно».
- Если автомат Бюхи неполный (что бывает при использовании программы *LTL2ba*), то при отсутствии активных переходов текущее состояние направляется в «сток»: недопускающее состояние с единственным переходом-петлей.

- Если активных переходов несколько, то возможные переходы «перемножаются» – создается ветвление как по активным переходам в автомате Бюхи, так и в модели Крипке. Например, если в текущем состоянии системы активны два перехода в автомате Бюхи и три перехода в модели Крипке, то для «перемноженной» системы будет ветвление из текущего состояния в шесть переходов.
- Если автомат Бюхи оказался в допускающем состоянии, то, следовательно, и пересечение автоматов оказалось в допускающем состоянии.

Перейдем к описанию алгоритма проверки на пустоту полученного пересечения (напомним, что пересечение – тоже автомат Бюхи).

Пусть p – допускающий путь автомата Бюхи. Так как множество состояний автомата конечно, то найдется суффикс p' пути p, такой что всякое состояние из него встречается бесконечное число раз. Это означает, что любое состояние этого суффикса достижимо из любого другого состояния суффикса. Следовательно, состояния из суффикса p' входят в состав некоторой сильно связной компоненты графа, описывающего автомат Бюхи. Причем, так как одно из допускающих состояний также входит в этот суффикс, то сильно связная компонента содержит допускающее состояние.

С другой стороны, если в графе автомата Бюхи существует достижимая сильно связная компонента, содержащая допускающее состояние, то можно построить допускающий путь. Он будет иметь структуру $\alpha\beta^*$, где α – префикс, ведущий к сильно связной компоненте, а β – цикл в этой компоненте, содержащий допускающее состояние. Структура допускающего пути изображена на рис. 8.

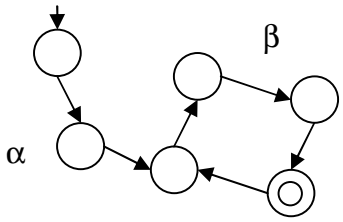


Рис. 8. Структура допускающего пути

Таким образом, проверка пустоты языка автомата Бюхи равносильна поиску сильно связной компоненты, достижимой из начального состояния и содержащей допускающее состояние. Для этого используется алгоритм *двойного поиска в глубину* [92] (*double Depth-First Search*, *DFS*).

В этом алгоритме чередуются два поиска в глубину. Первый из них может запускать второй, а второй может либо завершить работу всего алгоритма, либо передать управление обратно в первый поиск. В этом случае первый поиск продолжает свою работу. Каждый поиск использует свой флаг для пометки посещенных состояний.

Первый поиск запускает второй в тот момент, когда он готов к откату из допускающего состояния. Если второй поиск в процессе обхода попадает в состояние, находящееся в стеке первого поиска, то допускающий путь получен. Если этого не происходит, то после завершения обхода второй поиск возвращает управление в первый.

Более формально алгоритм проверки на пустоту языка автомата Бюхи с помощью двойного поиска в глубину может быть записан следующим (на псевдокоде из работы [92]) образом:

```
procedure emptiness
  for all q₀ ∈ Q₀ do
    dfs1(q₀);
  terminate(false);
end procedure

procedure dfs1(q)
  local q′;
  hash(q);
  for all последователей q′ вершины q do
    if q′ не содержится в хэш-таблице then dfs1(q′);
  if accept(q) then dfs2(q);
end procedure

procedure dfs2(q)
  local q′;
  flag(q);
  for all последователей q′ вершины q do
    if q′ в стеке dfs1 then terminate(true);
    else if q′ не является помеченной then dfs2(q′);
end procedure
```

Алгоритм возвращает значение `true`, если был найден допускающий путь, и значение `false` – в противном случае. Если алгоритм вернул значение `true`, то нетрудно восстановить допускающий путь: в стеке первого поиска хранится путь из начального состояния в некоторое допускающее состояние *q1*. Этот путь и будет искомым префиксом α. В стеке второго поиска хранится путь из состояния *q1* в некоторое состояние *q2*, содержащееся в стеке первого. Тогда, достроив этот путь состояниями, находящимися в стеке первого поиска выше *q2*, получим цикл $q1 \to q2 \to q1$, проходящий через допускающее состояние *q1*, а, следовательно, и искомый суффикс β. Таким образом, будет получен допускающий путь $\alpha\beta^*$.

Доказательство корректности алгоритма подробно описано в работе [92].

Для борьбы с комбинаторным взрывом в верификаторе *SPIN* используется эффективный алгоритм редукции частичных порядков, особенностью которого по сравнению с другими аналогичными алгоритмами является то, что он не требует дополнительной памяти для работы. Эксперименты показали [80], что на примере задачи выбора лидера среди нескольких процессов, число состояний системы без редукции растет экспоненциально от числа процессов, в то время как применение алгоритма редукции частичных порядков позволяет снизить этот рост до линейного.

Также в верификаторе *SPIN* используются эвристики для компактного хранения пройденных состояний. Это позволяет верифицировать большие системы. Для уменьшения памяти, занимаемой множеством посещенных состояний, можно было бы использовать сжатие с помощью алгоритма Лемпеля – Зива – Велча или алгоритма Хаффмана. Это приводит к сильному увеличению времени работы верификатора (до 400%), в то время как используемая память уменьшается не так сильно – 10−20% [80]. Используемый в верификаторе *SPIN* алгоритм приводит к экономии памяти на 60–80% при небольших затратах времени.

В случае если задача слишком велика и не поддается верификации по причине нехватки памяти, можно проводить неполную верификацию. При этом желательно добиться как можно большего покрытия пространства состояний верификатором, для того, чтобы иметь наиболее близкую к полной верификацию. В верификаторе *SPIN* для этого используется специальный алгоритм, кодирующий каждое посещенное состояние двумя битами, адреса которых генерируются двумя независимыми хэш-функциями. Этот алгоритм очень эффективен и позволяет добиться покрытия близкого к 100% даже для таких задач, максимальное покрытие пространства состояний которых без использования этого алгоритма равнялось 10%.

Перечисленные алгоритмы и улучшения делают верификатор *SPIN* эффективным и позволяют применять его для верификации реальных систем [80].

Так как верификатор *SPIN* создан для верификации протоколов, имеющих конечное число состояний, то можно предположить, что его применение для верификации автоматных моделей будет весьма эффективно. При этом верификатор *SPIN* поддерживает параллелизм, верифицируемых протоколов, и поэтому позволит верифицировать системы взаимодействующих автоматов.

1.10.2. Общее описание инструментального средства *Converter*

Инструментальное средство *Converter* позволяет автоматизировать верификацию визуальных автоматных программ, разработанных при помощи инструментального средства *UniMod* [35].

По автоматной программе инструментальное средство *Converter* создает модель, в которой отброшены несущественные детали. *LTL*-формула преобразуется в пригодный для верификатора *SPIN* вид.

Инструментальное средство *Converter* должно получать на вход три параметра.

1. Путь к файлу с визуальной автоматной программой, разработанной при помощи инструментального средства *UniMod* и сохраненной в *XML*-формате.
2. Имя файла, в который будет записана созданная модель.
3. Одну *LTL*-формулу с требованиями к модели.

На выходе инструментального средства *Converter* формируется файл, в котором записан полный отчет о проведенной верификации, включая автоматически построенный верификатором *SPIN* контрпример, представленный в текстовом виде, если он найден.

Автоматическая верификация автоматных программ с помощью инструментального средства *Converter* состоит из нескольких этапов (рис. 9).

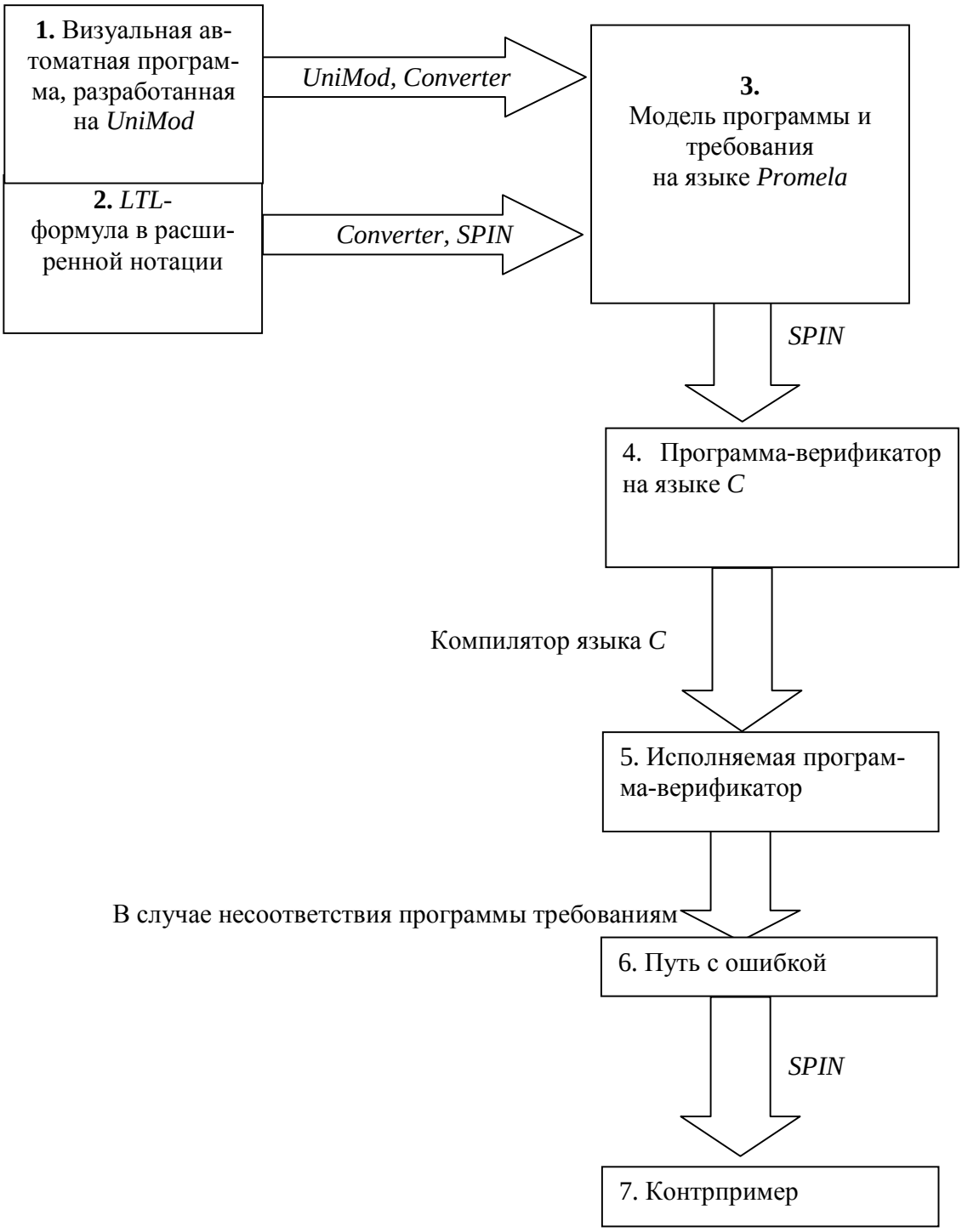


Рис. 9. Общее описание работы программного средства

1. *Converter* принимает на вход визуальную автоматную программу, разработанную при помощи инструментального средства *UniMod*, сохраненную в формате *XML*, и требования к ней, записанные на языке *LTL* в расширенной нотации (рис. 9, переход 1 – 2). **Важно: верификатору на вход подаются не требования, а их отрицание.**

2. При помощи инструментального средства *UniMod* из *XML*-файла получается автоматная модель на языке *Java* (рис. 10) – переход 1 – 3 на рис. 9.

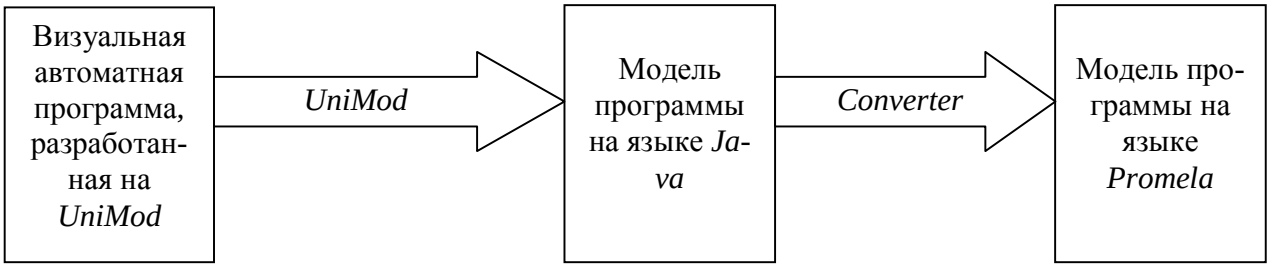


Рис. 10. Взаимодействие с *UniMod*

Converter транслирует автоматную модель с языка *Java* на язык *Promela* (рис. 10) – переход 1 –3 на рис. 9.

Converter преобразует *LTL*-формулу в нотацию верификатора *SPIN* (рис. 11).

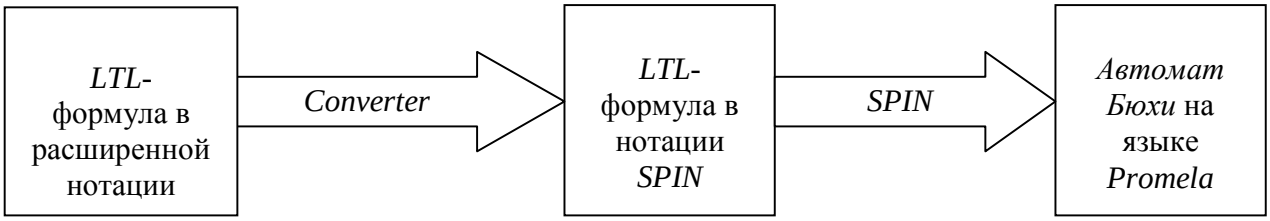


Рис. 11. Преобразование *LTL*-формул

Это преобразование выполняется следующим образом.

- Все элементарные высказывания должны быть записаны в фигурных скобках.
- Все элементарные высказывания должны удовлетворять синтаксису языка *Promela*.
- Каждому элементарному высказыванию присваивается идентификатор *pk*, где *k* – порядковый номер элементарного высказывания в формуле.
- Для каждого элементарного высказывания *Converter* генерирует макрос на языке *Promela*, закрепляющий идентификатор за элементарным высказыванием. Если *Converter* распознает формулу как неправильную, то он выводит в консоль сообщение «Wrong formula».
- Идентификаторы событий *exx*, где *xx* – номер события, преобразовываются в число *xx*.
- Пример. Формула (элементарное высказывание)

```
{lastEvent == e13}
```

преобразовывается в строку на языке *Promela*:

```
#define pk (lastEvent == 13),
```

где *k* – номер элементарного высказывания.

3. *Converter* запускает верификатор *SPIN* в режиме генерации конструкции *never claim* (с помощью команды `spin -f <формула>`). Верификатор по *LTL*-формуле генерирует конструкцию *never claim*, представляющую собой *автомат Бюхи*, записанный на языке *Promela* (рис. 11). Это также переход 2–3 на рис. 9.

4. После того, как на языке *Promela* описаны модель и требование к ней, *Converter* запускает *SPIN* на верификацию (командой `spin -a <Модель>`). Верификатор *SPIN* генерирует файл `pan.c`, представляющий собой программу-верификатор на языке *C* (переход 3 – 4 на рис. 9).
5. После компиляции файла `pan.c` получается программа-верификатор для данной конкретной модели с заданными требованиями. Программа *pan* выполняет верификацию построенной модели. При обнаружении ошибок программа *pan* выдает *trail*-файл, в котором описана трасса ошибки в формате, понятном верификатору *SPIN* (переходы 4–6 на рис. 9). Кроме того, программа-верификатор *pan* выводит отчет, в котором содержится краткая информация об ошибках (переход 5–6 на рис. 9), использованной памяти, версии верификатора *SPIN* и т.д.
6. По команде `spin –t –p <Модель>` верификатор выводит отчет, содержащий контрпример. *Converter* собирает воедино отчет, созданный *SPIN*, и отчет, созданный программой *pan* (переход 6–7 на рис. 9).

1.10.3. Верификация при помощи *UniMod.verifier*

Инструментальное средство *UniMod.verifier* предназначено для верификации автоматных программ. В ходе верификации программы возникает необходимость осуществлять следующие действия.

1. Вычислять глобальное состояние программы. Глобальное состояние должно однозначно определять поведение программы
2. Совершать элементарный шаг программы. Элементарный шаг является переходом программы из одного глобального состояния в другое без посещения иных глобальных состояний.
3. Откатывать элементарный шаг программы. При этом программа возвращается в предыдущее состояние.
4. В каждом состоянии определять возможные элементарные шаги.
5. Определять значения набора предикатов программы, используемых в требованиях.

Инструментальное средство *UniMod.verifier* использует *метод эмуляции* [93] для выполнения перечисленных действий. При этом верификация совершается путем управляемого эмулирования работы автоматной программы с одновременным наблюдением за верифицируемыми свойствами программы. Такой подход позволяет напрямую работать с верифицируемой программой, без дополнительных преобразований над ней.

В методе эмуляции пять действий, необходимых при верификации, осуществляются следующим образом.

Глобальное состояние программы складывается из набора текущих состояний каждого автомата.

Элементарный шаг работы программы – это обработка системой автоматов событий. В результате обработки может смениться набор состояний автоматов.

Поведение автоматной программы определяется набором состояний автоматов. Поэтому для отката достаточно вернуть автоматы в те состояния, в которых они находились до выполнения шага.

Для автоматной программы строго определена схема работы системы автоматов и последовательность передачи управления между автоматами. Кроме того, система работает в одном потоке. Поэтому недетерминированность в работе системы возникает лишь в результате разных последовательностей входных событий, а также в результате различных возможных значений переменных, запрашиваемых у объектов управления. Поэтому в методе эмуляции возможные элементарные шаги определяются следующим образом.

Перед совершением очередного элементарного шага определяется набор событий, которые система автоматов может обработать в текущем глобальном состоянии, и каждое из этих событий затем используется для создания одной из историй работы программы. Аналогично, при необходимо-

сти вычислить условие перехода, в выражении которого участвуют переменные объектов управления. Такое условие принимается равным `True` или `False`, и эти значения используются для создания различных историй работы программы.

Для вычисления предикатов при совершении элементарного шага сохраняется следующая информация:

- состояния автоматов до выполнения шага;
- обрабатываемое событие;
- список вычисленных значений условий на переходах;
- список вызванных действий у объектов управления.

Эта информация позволяет вычислять значения предикатов. В методе эмуляции поддерживается следующий набор предикатов:

- `wasEvent(e)` – возвращает `True`, если в последнем шаге было выбрано для обработки событие `e`, и `False` – в противном случае;
- `wasInState(sm, s)` – возвращает `True`, если перед последним шагом автомат `sm`, находился в состоянии `s`;
- `isInState(sm, s)` – возвращает `True`, если после совершения последнего шага автомат `sm` находится в состоянии `s`;
- `cameToState(sm, s)` – возвращает `True`, если после совершения последнего шага автомат `sm` сменил свое состояние на `s`. Тоже, что `(isInState(sm, s) && !wasInState(sm, s))`;
- `cameToFinalState()` – возвращает `True`, если после совершения шага корневой автомат модели вошел в свое конечное состояние. Это означает, что автоматная программа завершила работу.
- `wasAction(z)` – возвращает `True`, если в ходе выполнения шага было вызвано выходное воздействие `z`;
- `wasFirstAction(z)` – возвращает `True`, если в ходе выполнения шага первым вызванным действием было `z`;
- `wasLastAction(z)` – возвращает `True`, если в ходе выполнения шага последним вызванным действием было `z`;
- `getActionIndex(z)` – возвращает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Этот предикат предназначен для того, чтобы формулировать утверждения, задающие порядок вызова действий объектов управления в автоматной программе;
- `wasTrue(g)` – возвращает `True`, если в ходе выполнения последнего шага один из переходов был помечен условием `g`, и его значение было определено как `True`. `g` описывает целое условие, а не значение одной переменной. Например, `g = «!o1.x1 && o1.x2»`;
- `wasFalse(g)` – возвращает `True`, если в ходе выполнения последнего шага на одном из переходов было встречено условие `g`, и его значение было определено как `False`.

Особенностью метода эмуляции является то, что он не требует дополнительных преобразований над автоматной программой или над контрпримером, сгенерированным верификатором. Например, в работе [93] верификация системы автоматов производится с использованием следующих преобразований.

1. Система автоматов преобразуется в модель Крипке, записанную на входном языке верификатора *SPIN*.

2. Требования к системе автоматов переводятся в термины построенной модели.
3. Модель верифицируется верификатором *SPIN*, в случае ошибки выдается сценарий ошибки в терминах входного языка *SPIN*.
4. Сценарий ошибки переводится в термины исходной системы автоматов.

Эта схема верификации изображена на рис. 12.

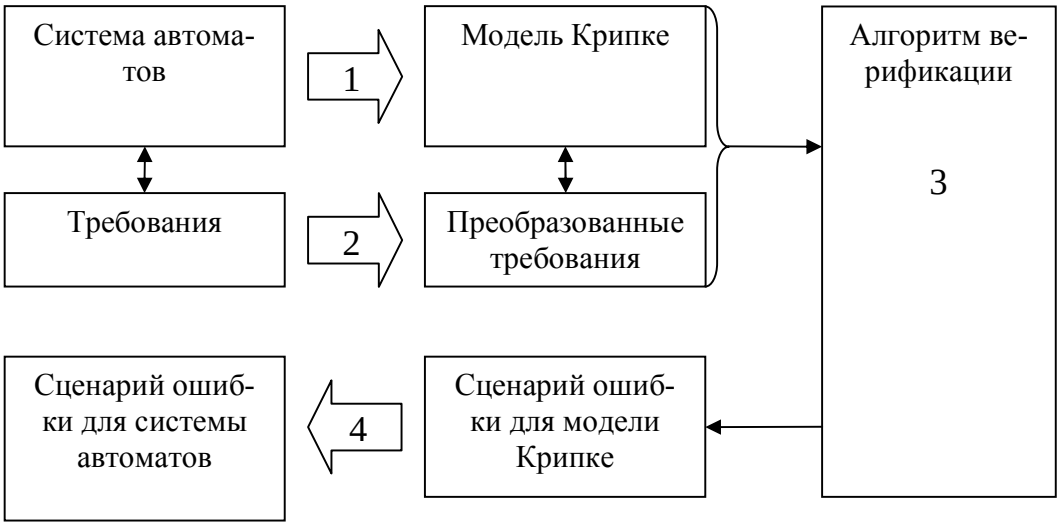


Рис. 12. Схема верификации с явным построением модели Крипке

В методе же эмуляции не строится явно модель Крипке, и алгоритм верификации работает непосредственно с системой автоматов. В результате не требуются ни преобразование системы во входной язык верификатора, ни преобразование требований, ни преобразование сценария ошибки, поскольку сценарий сразу выдается в терминах состояний и переходов в системе автоматов. Схема верификации по методу эмуляции изображена на рис. 13.

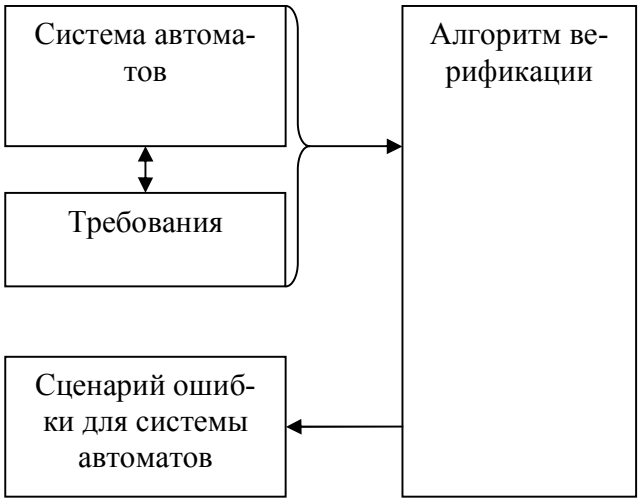


Рис. 13. Схема верификации методом эмуляции

Поскольку в методе эмуляции обработка событий в системе автоматов происходит атомарно с точки зрения верификации, возникают некоторое ограничение относительно темпоральных формул, описывающих требования к системе. Это ограничение состоит в том, что если в требовании важен порядок выполнения предикатов, то их значения не должны изменяться одновременно в пределах одной обработки события.

Инструментальное средство *UniMod.verifier* основано на верификаторе *Bogor* [94, 95] – верификаторе с модульной структурой и расширяемым входным языком, который называется *BIR – Bogor Input Language*. Благодаря расширяемости входного языка верификатора *Bogor* появилась возможность интеграции верификатора с инструментом для создания и запуска автоматных программ *UniMod*. Таким образом, было создано единое средство для создания, запуска и верификации автоматных программ.

Входной язык верификатора *Bogor* был расширен новым типом – моделью системы автоматов. Над этой моделью можно совершить лишь одно действие («`step`» – совершить один элементарный шаг работы системы автоматов) – выбрать очередное событие и обработать его в системе автоматов. Также у модели можно запросить многочисленные свойства, соответствующие предикатам, как например, `wasEvent`, `isInState` и т.д. Созданный тип был назван `AutomataModel`. Его объявление на языке *BIR* выглядит следующим образом:

```
extension AutomataModel for
    com.UniMod.verifier.Bogorextension.AutomataModelModule
{
    typedef type;

    expdef AutomataModel.type create();

    expdef boolean wasEvent(AutomataModel.type model, string
        event);
    expdef boolean wasInState(AutomataModel.type model, string
        stateMachine, string state);
    expdef boolean isInState(AutomataModel.type model, string
        stateMachine, string state);
    expdef boolean cameToState(AutomataModel.type model, string
        stateMachine, string state);
    expdef boolean cameToFinalState(AutomataModel.type model);
    expdef boolean wasAction(AutomataModel.type model, string
        action);
    expdef boolean wasFirstAction(AutomataModel.type model,
        string action);
    expdef boolean wasLastAction(AutomataModel.type model,
        string action);
    expdef int getActionIndex(AutomataModel.type model, string
        action);
    expdef boolean wasTrue(AutomataModel.type model, string
        guard);
    expdef boolean wasFalse(AutomataModel.type model, string
        guard);

    actiondef step(AutomataModel.type model);
}
```

Во многих предикатах участвует имя автомата. Поскольку в системе автоматов один автомат может быть вложен в несколько состояний своего родителя, то для идентификации автомата вводится понятие *путь автомата*, который строится по следующим правилам:

- путь корневого автомата имеет формат «/<название корневого автомата>»;
- путь вложенных автоматов имеет формат «<путь родительского автомата>:<состояние родительского автомата>/<название вложенного автомата>».

Верификатор *Bogor* верифицирует программы, написанные на языке *BIR*. За счет создания специального типа, программа для верификации системы автоматов весьма проста:

```
AutomataModel.type model;

main thread MAIN() {
  loc init:
    do invisible {
      model := AutomataModel.create();
    } goto loop;

  loc loop:
    do {
      AutomataModel.step(model);
    } goto loop;
}
```

Программа состоит из двух состояний. В первом из них (`init`) создается новая система автоматов. Реально при этом из указанного отдельно файла создается *UniMod*-модель автоматной программы. Второе состояние (`loop`) бесконечный цикл, в котором постоянно выполняется шаг работы системы автоматов. Отметим, что несмотря на то, что рассматриваемый цикл бесконечный и не имеет выхода, программа не зависнет. Программа для верификатора – это не реально исполняющаяся программа, а модель, которая подвергается верификации. Когда автоматная модель в бесконечном цикле попадет в уже посещенное состояние, верификатор остановит цикл.

Требования к системе автоматов формулируются на языке *BIR* в темпоральной логике *LTL*. Для этого в языке *BIR* есть специальное расширение. Например, свойство «автомат `A` никогда не попадет в состояние `Error`» записывается на языке *BIR* следующим образом:

```
LTL.temporalProperty(
  Property.createObservableDictionary(
  Property.createObservableKey(
    "is_Error", AutomataModel.isInState(model, "/A", "Error"))
  ),

  LTL.always(
    LTL.negation(LTL.prop("is_Error"))
  )
);
```

Сначала создается пропозиционная формула «`is_Error`», которой соответствует вызов требуемого свойства системы автоматов. Затем записывается темпоральная формула в виде вложенных темпоральных операторов. В расширении *LTL* языка *BIR* поддерживаются операторы, указанные в таблице (операторы записаны, как они объявляются на языке *BIR*).

Таблица. Операторы языка *BIR*

`expdef LTL.Formula` *always* `(LTL.Formula);`	**G** (Globally, всегда)
`expdef LTL.Formula` *eventually* `(LTL.Formula);`	**F** (Future, когда-нибудь в будущем)
`Expdef LTL.Formula` *negation* `(LTL.Formula);`	¬ (отрицание)
`expdef LTL.Formula` *next* `(LTL.Formula);`	**X** (neXt, в следующий момент времени)
`expdef LTL.Formula` *until* `(LTL.Formula, LTL.Formula);`	**U** (Until, до тех пор, пока)
`expdef LTL.Formula` *weakUntil* `(LTL.Formula, LTL.Formula);`	**W** (Weak until). Этот оператор был добавлен в *LTL*-расширение языка *BIR* для удобства. *p W q = (p U q) | G (p ∩ ¬q)*. Это – то же самое, что *p U q*, однако *q* не должно когда-либо выполниться
`expdef LTL.Formula` *release* `(LTL.Formula, LTL.Formula);`	**R** (Release, освобождение)
`expdef LTL.Formula` *equivalence* `(LTL.Formula, LTL.Formula);`	↔ (Эквивалентно). p ↔ q = (p → q) & (q → p)
`expdef LTL.Formula` *implication* `(LTL.Formula, LTL.Formula);`	→ (Следует)
`expdef LTL.Formula` *conjunction* `(LTL.Formula, LTL.Formula);`	**&** (И)
`expdef LTL.Formula` *disjunction* `(LTL.Formula, LTL.Formula);`	| (Или)

С помощью этих операторов можно задать любую *LTL*-формулу для записи требований к системе автоматов. Такие формулы во время верификации преобразуются в автомат Бюхи, как этого требует алгоритм двойного обхода в глубину [92]. Например, *LTL*-формула `G¬(is_Error)` будет преобразована в автомат следующего вида:

```
function generated$FSA()
{
    loc T0_init:
        when true do
        {
        }
        goto T0_init;
        when AutomataModel.isInState(model, "/A", "Error") do
        {
        }
```

```
            goto bad$accept_all;
        loc bad$accept_all:
            when true do
            {
            }
            goto bad$accept_all;
    }
```

Состояния сгенерированного автомата, названия которых начинаются с «bad$», являются допускающими.

Выводы по главе 1

1. Корректность является наиболее важным свойством современного программного обеспечения. Для проверки работы управляющих программ используются несколько методов, самым мощным из которых является формальная верификация.
2. В последние годы автоматное программирование все чаще используется для построения управляющих программ ответственных систем. Основной идеей автоматного программирования является представление программ в виде множества взаимодействующих автоматов.
3. Функциональное программирование является мощным вычислительным формализмом, эквивалентным машине Тьюринга, набирающим популярность в академических исследованиях и промышленном программировании.
4. До сих пор неизвестны реализации структурных конечных автоматов на функциональных языках программирования. Валидация автоматных программ обычно производится сторонними утилитами, а не компилятором.
5. Большинство разрабатываемых сегодня программ строится на основе объектно-ориентированного подхода, для которого разработан целый рад методов повышения качества, таких как: тестирование, рефакторинг и паттерны проектирования. В тоже время, активно применяемые в ответственных системах методы объектно-ориентированного автоматного программирования также требуют разработки методов повышения качества для автоматных частей таких программ.
6. Существует целый ряд способов и средств верификации на модели, однако построение моделей для обычных программ требует больших усилий, а в общем случае является неразрешимой задачей. Однако для автоматной программы переход к модели и обратно можно осуществлять автоматически. Поэтому верификация на модели может эффективно применяться для автоматных программ.

2. ВЫБОР И ОБОСНОВАНИЕ ОПТИМАЛЬНОГО ВАРИАНТА НАПРАВЛЕНИЯ ИССЛЕДОВАНИЙ

В настоящем разделе представлено обоснование оптимального варианта проведения исследований.

На основании результатов выполненного аналитического обзора предлагается следующий вариант направления исследований: будет проведена разработка методов реализации автоматов на функциональных языках программирования, методов рефакторинга автоматов при разработке автоматных программ, метода динамической верификации автоматных программ, а также методов применения контрактов при разработке объектно-ориентированных автоматных систем. Кроме того, будут разработаны виртуальные лаборатории для обучения рефакторингу автоматных программ и применению контрактов при разработке объектно-ориентированных автоматных систем, которые будут апробированы при выполнении студенческих лабораторных работ.

Экспериментальные исследования разработанных методов будут проводиться на примере. После выполнения экспериментальных исследований будет проведена корректировка разработанных методов. По итогам работ будет зарегистрировано не менее четырех программ для ЭВМ, а также будет опубликовано не менее четырех статей в журналах из перечня ВАК.

Конечные автоматы широко применяются как модель при разработке программного обеспечения ответственных систем. При этом возникает необходимость валидации построенных автоматов. Существующие методы валидации автоматов основаны на различных надстройках над используемыми языками программирования, что не является безопасным для построения ответственных систем. Одним из способов решения этой проблемы является перенос валидации автоматов на этап компиляции. С такой задачей традиционно хорошо справляются функциональные языки программирования. Подходы к совместному использованию автоматного и функционального программирования разработаны недостаточно. Для устранения этого недостатка будут разработаны методы реализации автоматов на функциональных языках программирования, обеспечивающие валидацию некоторых их свойств на этапе компиляции.

При разработке объектно-ориентированных программ применяется широкий спектр различных методов, позволяющих повысить качество разрабатываемого кода. К ним относится рефакторинг кода и использование контрактов. При проведении теоретических исследований будут разработаны аналоги данных методов, которые позволят улучшить качество процесса создания автоматных программ.

Кроме этого, при выполнении работ по настоящему государственному контракту будут разработаны виртуальные лаборатории для обучения рефакторингу автоматных программ и применению контрактов при разработке объектно-ориентированных автоматных систем. Эти программы позволят увеличить объем знаний по методам рефакторинга, а также по методам применения контрактов при разработке автоматных программ.

Выводы по главе 2

1. Выбрано и обосновано направление проведения исследований.

3. ПЛАН ПРОВЕДЕНИЯ ТЕОРЕТИЧЕСКИХ И ЭКСПЕРИМЕНТАЛЬНЫХ ИССЛЕДОВАНИЙ

3.1. План проведения первого этапа теоретических и экспериментальных исследований

На первом этапе проведения исследований планируется провести следующие работы:

- разработка методов реализации автоматов на функциональных языках программирования (25.11.2009 – 27.11.2009):
  - разработка обеспечивающего валидацию на этапе компиляции метода представления функций переходов автоматов без выходов в функциональных языках программирования на основе алгебраических типов данных;
  - разработка метода представления вложенных автоматов;
  - разработка метода применения монад для представления структурных автоматов Мили с выходными воздействиями;
  - разработка метода представления автоматов Мура и смешанных автоматов.

Результаты работ первого этапа:

- методы реализации автоматов на функциональных языках программирования;
- научно-технический отчет.

3.2. План проведения второго этапа теоретических и экспериментальных исследований

На втором этапе проведения исследований планируется провести следующие работы:

- программная реализация методов реализации автоматов на функциональных языках программирования (01.01.2010 – 15.01.2010):
  - программная реализация обеспечивающего валидацию на этапе компиляции метода представления функций переходов автоматов без выходов в функциональных языках программирования на основе алгебраических типов данных;
  - программная реализация метода представления вложенных автоматов;
  - программная реализация метода применения монад для представления структурных автоматов Мили с выходными воздействиями;
  - программная реализация метода представления автоматов Мура и смешанных автоматов.
- экспериментальное исследование методов реализации автоматов на функциональных языках программирования (16.01.2010 – 31.01.2010);
- разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ (01.02.2010 – 31.05.2010);
- разработка, программная реализация и экспериментальное исследование метода динамической верификации автоматных программ (01.02.2010 – 31.05.2010);
- разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем (01.02.2010 – 31.05.2010).

По итогам второго этапа работ планируется подготовить для публикации две статьи в журналах из перечня ВАК, а также подготовить и подать заявки на регистрацию двух программ для ЭВМ.

Результаты работ второго этапа:

- программная реализация методов реализации автоматов на функциональных языках программирования;
- методы рефакторинга автоматов при разработке автоматных программ;
- метод динамической верификации автоматных программ;
- методы применения контрактов при разработке объектно-ориентированных автоматных систем;
- две статьи в журналах из перечня ВАК;
- два свидетельства о регистрации программы для ЭВМ;
- протоколы экспериментов;
- научно-технический отчет.

3.3. План проведения третьего этапа теоретических и экспериментальных исследований

На третьем этапе проведения исследований планируется провести следующие работы:

- разработка программы, позволяющей увеличить объем знаний для более глубокого понимания изучаемого предмета исследования и пути применения новых явлений, механизмов или закономерностей (виртуальная лаборатория для обучения рефакторингу автоматных программ) (01.01.2011 – 31.01.2011);
- разработка программы, позволяющей увеличить объем знаний для более глубокого понимания изучаемого предмета исследования и пути применения новых явлений, механизмов или закономерностей (виртуальная лаборатория для обучения применению контрактов при разработке объектно-ориентированных автоматных систем) (01.02.2011 – 31.03.2011);
- апробация разработанных виртуальных лабораторий при выполнении студенческих лабораторных работ с публикацией отчетов в сети Интернет (01.02.2011 – 30.06.2011);
- обобщение и оценка результатов исследований – оценка полноты решения задачи и достижения поставленных целей (01.05.2011 – 30.06.2011);
- обобщение и оценка результатов исследований – сопоставление и обобщение результатов анализа научно-информационных источников и теоретических и экспериментальных исследований (01.05.2011 – 30.06.2011);
- обобщение и оценка результатов исследований – оценка эффективности полученных результатов в сравнении с современным научно-техническим уровнем (01.05.2011 – 30.06.2011);
- разработка рекомендаций по использованию результатов НИР при создании научно-образовательных курсов (01.05.2011 – 30.06.2011).

По итогам третьего этапа работ планируется подготовить для публикации две статьи в журналах из перечня ВАК, а также подготовить и подать заявки на регистрацию двух программ для ЭВМ.

Результаты работ третьего этапа:

- виртуальная лаборатория для обучения рефакторингу автоматных программ;
- виртуальная лаборатория для обучения применению контрактов при разработке объектно-ориентированных автоматных систем;
- две статьи в журналах из перечня ВАК, содержащие основные результаты НИР;
- два свидетельства о регистрации программы для ЭВМ;
- рекомендации по использованию результатов НИР при создании научно-образовательных курсов;
- научно-технический отчет.

Выводы по главе 3

1. Разработан план проведения теоретических и экспериментальных исследований.
2. На первом этапе теоретических исследований будет проведена разработка методов реализации автоматов на функциональных языках программирования.

4. РАЗРАБОТКА И ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ МЕТОДОВ РЕАЛИЗАЦИИ АВТОМАТОВ НА ФУНКЦИОНАЛЬНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

В данном разделе предложены методы реализации событийных структурных конечных автоматов на функциональных языках программирования с последующим обобщением на вложенные автоматы и на использование монад.

4.1. Разработка обеспечивающего валидацию на этапе компиляции метода представления функций переходов автоматов без выходов в функциональных языках программирования на основе алгебраических типов данных

В настоящем разделе предложен метод представления функций переходов автоматов без выходных воздействий, обеспечивающих валидацию на этапе компиляции.

Суть предлагаемого подхода к реализации функции переходов на функциональных языках программирования состоит в представлении событий при помощи алгебраических типов данных, а состояний автоматов – кортежами или структурами соответствующих переменных. Тогда функция переходов должна принимать в качестве параметров кортеж старого состояния и входные воздействия (события), а возвращать – кортеж нового состояния. Продемонстрируем данный подход на примерах.

4.1.1. Обработка одиночных событий

Пусть требуется реализовать счетный триггер, реагирующий на события. Другими словами, требуется построить конечный автомат с двумя состояниями, кодируемыми нулем и единицей, который управляется кнопкой. Каждое нажатие кнопки порождает событие *e*. К выходу *z* конечного автомата подключена лампа. Каждое событие *e* переводит автомат в состояние *1 - y*, где *y* – текущее состояние. При этом переменная состояния одновременно является выходной переменной *(z = y)*. Таким образом, каждое нажатие кнопки будет приводить то к включению лампы, то к ее выключению. Диаграмма состояний данного автомата представлена на рис. 14.

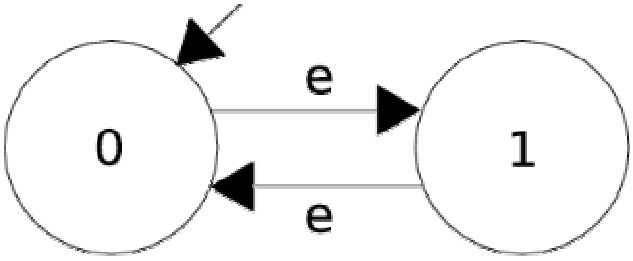


Рис. 14. Диаграмма состояний счетного триггера [2]

Исходный код, реализующий этот триггер, приведен в листинге 1.

Листинг 1. Реализация счетного триггера на *Haskell*

```
-- Событие: «Нажатие на кнопку».
data Event = ButtonClick deriving Show
-- Состояния: «Выключено» и «Включено».
data State = LampOff | LampOn deriving Show

-- Функция переходов.
gotEvent :: State -> Event -> State
gotEvent LampOff ButtonClick = LampOn
gotEvent LampOn ButtonClick = LampOff

-- Функция, вызываемая системой.
-- Начальное состояние – LampOff.
main = print $ gotEvent LampOff ButtonClick
```

Рассмотренная реализация функции переходов не является единственно возможной. Например, для этой цели можно использовать конструкции *case* так, как это сделано в листинге 2.

Листинг 2. Использование конструкции case

```
-- Исходный код предыдущего листинга
...

gotEvent :: State -> Event -> State
gotEvent state event = case state of
    LampOff -> case event of
        ButtonClick -> LampOn
    LampOn -> case event of
        ButtonClick -> LampOff
```

Таким образом, метод преобразования диаграммы переходов в код функции переходов на языке *Haskell* состоит в следующем:

- объявить функцию переходов с двумя параметрами: старым состоянием системы и событием;
- объявить оператор *case*, у которого в качестве параметра используется старое состояние;
- для каждого состояния автомата:
  - добавить ветку в операторе *case*, где слева от стрелки стоит идентификатор состояния, а справа – переходы автомата в этом состоянии;
  - если переход автомата не зависит от события, то справа от стрелки поместить этот переход (например, идентификатор нового состояния);
  - если переход автомата зависит от события, то справа от стрелки объявить оператор *case*, у которого в качестве параметра используется событие. После этого для каждого возможного события следует добавить ветку в этом операторе, где слева от стрелки стоит идентификатор события, а справа – действие автомата при его получении.

4.1.2. Последовательности событий

Для того чтобы добавить возможность применения последовательности событий к начальному состоянию введем функцию *applyEvents* (листинг 3).

Листинг 3. Функция *applyEvents* и ее использование

```
-- Исходный код предыдущего листинга за исключением функции
        main
...

-- Функция, применяющая события к начальному состоянию.
applyEvents :: State -> [Event] -> State
-- Результат = состояние, если событий больше нет.
applyEvents st [] = st
-- Иначе делаем переход и вызываемся рекурсивно.
applyEvents st (e:es) = applyEvents (gotEvent st e) es

-- Новая функция main.
main = print $ applyEvents LampOff [ButtonClick]
```

В листинге 3 функция принимает в качестве параметров: состояние, к которому необходимо применить список событий, и сам список событий. Если этот список пуст, то функция возвращает переданное ей состояние, иначе она вызывает себя рекурсивно, но в качестве нового списка событий использует хвост текущего, а в качестве нового состояния – результат применения функции переходов к голове списка и текущему состоянию.

Отметим, что при ручном программировании функций переходов конечных автоматов программистам на императивных языках часто приходится следить за структурой выражений, используемых для вычисления новых переменных состояния, поскольку смена их значений должна происходить не последовательно, а одновременно во всей системе. Поэтому в подобных реализациях применяются промежуточные переменные: сначала программа считает значения изменяющихся переменных и записывает их в промежуточные переменные, а затем следует блок присваиваний значений промежуточных переменных переменным состояния, за исключением, быть может, последней переменной. В то же время, программист на функциональном языке избавлен от необходимости следить за подобными вещами вручную. Одновременность перехода обеспечивается за счет использования кортежей или структур.

4.1.3. Обобщение на произвольные типы данных для событий и состояний

Если бы в реализуемом примере было более одного конечного автомата, то пришлось бы писать несколько функций, аналогичных *applyEvents*. Поэтому можно выделить общую их часть в библиотечный код, представленный в листинге 4.

Листинг 4. Библиотечная функция *applyEvents* и ее использование

```
-----------------------------------------------------
-- Библиотечный код.

-- Тип функции переходов.
type SwitchFunc state event = state -> event -> state

-- Функция, применяющая список событий к начальному
-- состоянию автомата при помощи функции переходов.
applyEvents :: SwitchFunc state event -> state -> [event] ->
        state
applyEvents _ state [] = state
applyEvents switchFunc state (event:eventsTail) = applyEvents
        switchFunc
        (switchFunc state event) eventsTail

-----------------------------------------------------
-- Реализация счетного триггера
-- при помощи приведенного выше библиотечного кода.

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Функция переходов для счетного триггера.
triggerSwF LampOff ButtonClick = LampOn
triggerSwF LampOn ButtonClick = LampOff

-- Функция, вызываемая системой.
main = print $ applyEvents triggerSwF LampOff [ButtonClick]
```

Новая версия функции *applyEvents* принимает дополнительный аргумент – функцию переходов. Таким образом, *applyEvents* становится полиморфной и может оперировать любыми типами данных, представляющими события и состояния. Отметим также, что новая версия функции *applyEvents*, является левой сверткой (одна из стандартных операций функционального программирования) списка событий по функции переходов. Поэтому можно заменить все определение функции *applyEvents* на *applyEvents = foldl*.

4.1.4. Преимущества функциональной реализации

При сравнении реализаций счетного триггера на *Haskell* с реализациями на языке *C* (листинги 5, 6) можно отметить, что алгебраические типы данных позволяют производить более строгие проверки на этапе компиляции. Например, при использовании алгебраических типов данных невозможно случайно проверить на равенство элемент множества состояний с элементом множества событий, а код на *C* лишен этого свойства.

Листинг 5. Реализация счетного триггера на *C*

```
// Событие: <<Нажатие на кнопку>>.
#define EVENT_BUTTONCLICK 0
// Состояния: <<Выключено>> и <<Включено>>.
#define LAMPOFF 0
#define LAMPON 1

// Переменная состояния, начальное состояние --- LAMPOFF.
int current_state = LAMPOFF;
// Функция переходов.
void got_event(int event)
{
    if (event == EVENT_BUTTONCLICK)
        switch (current_state)
        {
            case LAMPOFF: current_state = LAMPON; break;
            case LAMPON: current_state = LAMPOFF; break;
        }
}

// Функция, вызываемая системой.
int main()
{
    got_event(EVENT_BUTTONCLICK);
    printf("%i", current_state);
}
```

Листинг 6. Реализация счетного триггера на *C* с использованием конструкции *enum*

```
// Событие: <<Нажатие на кнопку>>.
enum event {EVENT_BUTTONCLICK};
// Состояния: <<Выключено>> и <<Включено>>.
enum state {LAMPOFF, LAMPON};

// Переменная current_state, функции got_event
// и main из предыдущего листинга
...
```

Однако основным достоинством данного подхода является то, что компилятор способен самостоятельно проверить полноту и непротиворечивость веток оператора *case*, тем самым, осуществляя валидацию функции переходов на этапе компиляции.

4.2. Разработка метода представления вложенных автоматов

В разд. 4.1 было рассмотрено представление конечных автоматов, которые реагируют на внешние воздействия только сменой состояний. Однако состояние системы может содержать описание нескольких автоматов. Таким образом можно реализовывать вложенные конечные автоматы.

В качестве задачи для реализации системы со вложенностью рассмотрим упрощенную модель цифрового устройства с четырьмя кнопками на корпусе и экраном. Устройство может быть включено,

может отображать меню, состоящее из некоторого списка элементов, и быть выключено, причем выключенное устройство запоминает состояние меню и при следующем входе в него отображает сохраненную позицию курсора.

Следовательно, имеется два конечных автомата (устройство и меню), причем второй вложен в первый. Автомат, описывающий устройство, имеет три состояния: «Выключено», «Включено» и «Меню», а автомат, описывающий меню, – два: «Выбран первый пункт меню» и «Выбран второй пункт меню». Панель устройства содержит четыре кнопки: «Включить-выключить», «Меню», «Вверх», «Вниз». Графы переходов системы представлены на рис. 15.

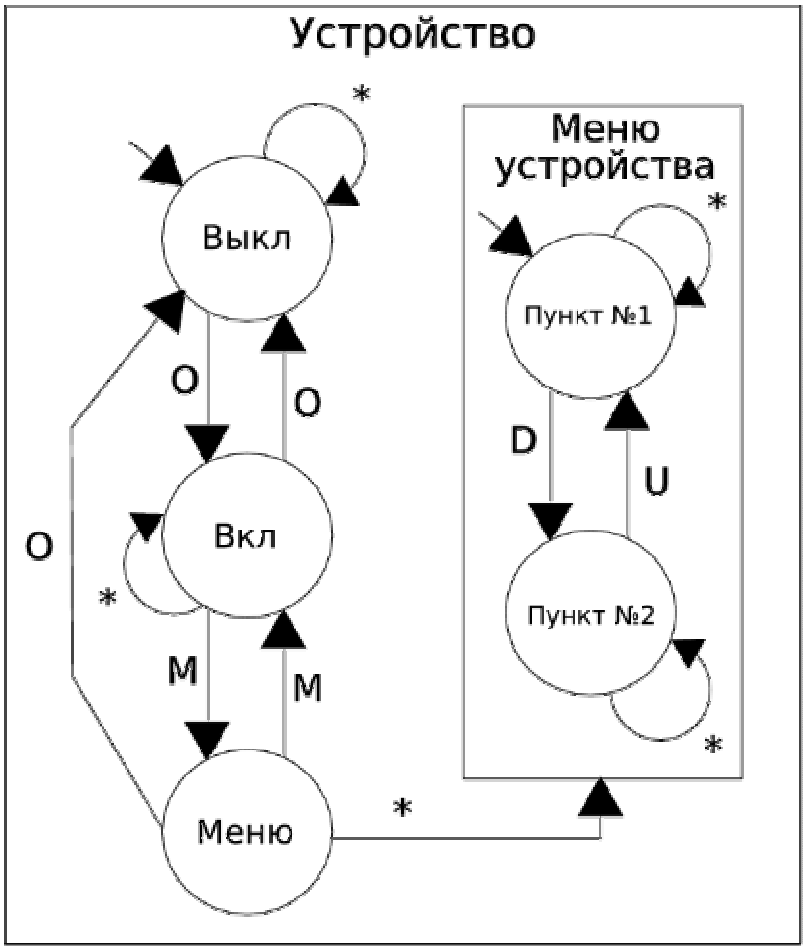


Рис. 15. Диаграммы состояний устройства и его меню. «O», «M», «U», «D» – кнопки включения-выключения, меню, вверх и вниз соответственно

Общая часть реализации примеров этой секции вынесена в листинг 7.

Листинг 7. Определения состояний и событий устройства с меню

```
-- Типы событий и состояний для устройства.
data DeviceEvent = OnOffButton
                 | MenuButton
                 | UpButton
                 | DownButton deriving Show
data DeviceState = DeviceIsOff
                 | DeviceIsOn
                 | DeviceIsInMenu deriving Show

-- Типы событий и состояний для меню.
data MenuEvent = DeviceButton DeviceEvent deriving Show
data MenuState = MenuIsInPositionOne
               | MenuIsInPositionTwo deriving Show

data SystemState = SystemState
    { dstate :: DeviceState
    , mstate :: MenuState } deriving Show

applyEvents = foldl
```

4.2.1. Реализация с зависимыми функциями переходов

Первым предлагаемым методом реализации вложенности является реализация для всех автоматов собственных функций переходов (методом из разд. 4.1) с той особенностью, что функции переходов внешних автоматов манипулируют состояниями всех внутренних автоматов.

Поскольку для структур данных компилятор автоматически генерирует функции извлечения элементов, то их использование выглядит привлекательнее при работе с большими векторами состояний, нежели использование кортежей.

У рассматриваемого устройства с меню вектор состояния системы содержит два элемента. Поэтому разница между реализацией на структурах и реализацией на кортежах практически отсутствует. В листинге 8 используются структуры.

Листинг 8. Модель устройства, реализованная при помощи структур

```
-- Определения состояний и событий
...

-- Функция переходов для устройства.
deviceSwF state event = case dstate state of
    DeviceIsOff -> case event of
        OnOffButton -> state { dstate = DeviceIsOn }
        _ -> state
    DeviceIsOn -> case event of
        OnOffButton -> state { dstate = DeviceIsOff }
        MenuButton -> state { dstate = DeviceIsInMenu }
        _ -> state
    DeviceIsInMenu -> case event of
        OnOffButton -> state { dstate = DeviceIsOff }
        MenuButton -> state { dstate = DeviceIsOn }
        _ -> state { mstate =
            (menuSwF (mstate state)
                (DeviceButton event)) }

-- Функция переходов для меню.
menuSwF mstate event = case mstate of
    MenuIsInPositionOne -> case event of
        DeviceButton DownButton -> MenuIsInPositionTwo
        _ -> mstate
    MenuIsInPositionTwo -> case event of
        DeviceButton UpButton -> MenuIsInPositionOne
        _ -> mstate

-- Функция, вызываемая системой.
main = print $ applyEvents
    deviceSwF
    ( SystemState
        { dstate = DeviceIsOff
        , mstate = MenuIsInPositionOne } )
    [OnOffButton, MenuButton, DownButton, OnOffButton]
```

4.2.2. Реализация с независимыми автоматами

В предыдущей реализации (листинг 8) функция переходов автомата устройства имела полный контроль над состоянием вложенного автомата с меню, поскольку оперировала всем состоянием системы целиком. Однако если устройство может посылать события, адресованные меню, через некоторый внешний канал, то автоматы можно сделать независимыми друг от друга. Реализация, приведенная в листинге 9, использует именно этот подход.

Листинг 9. Модель устройства с независимыми автоматами

```
-- Определение состояний и событий
...

-- Функция переходов всей системы.
systemSwF state event =
    SystemState { dstate = ndstate, mstate = nmstate }
    where
        (ndstate, act) = deviceSwF (dstate state) event
        nmstate = case act of
            Nothing -> mstate state
            Just x -> menuSwF (mstate state) x

-- Функция переходов для устройства.
-- Возвращает кортеж (новое состояние,
-- событие отправляемое вложенному автомату).
deviceSwF dstate event = case dstate of
    DeviceIsOff -> case event of
        OnOffButton -> (DeviceIsOn, Nothing)
        _ -> (dstate, Nothing)
    DeviceIsOn -> case event of
        OnOffButton -> (DeviceIsOff, Nothing)
        MenuButton -> (DeviceIsInMenu, Nothing)
        _ -> (dstate, Nothing)
    DeviceIsInMenu -> case event of
        OnOffButton -> (DeviceIsOff, Nothing)
        MenuButton -> (DeviceIsOn, Nothing)
        _ -> (dstate, Just $ DeviceButton event)

-- Функция переходов для меню.
menuSwF mstate event = case mstate of
    MenuIsInPositionOne -> case event of
        DeviceButton DownButton -> MenuIsInPositionTwo
        _ -> mstate
    MenuIsInPositionTwo -> case event of
        DeviceButton UpButton -> MenuIsInPositionOne
        _ -> mstate

-- Функция, вызываемая системой.
main = print $ applyEvents
    systemSwF
    ( SystemState
        { dstate = DeviceIsOff
        , mstate = MenuIsInPositionOne } )
    [OnOffButton, MenuButton, DownButton, OnOffButton]
```

Отметим, что это первый пример, реализующий выходные воздействия – посылка автоматом устройства события автомату меню. Функция переходов всей системы *systemSwF* реализует передачу сообщений между автоматами.

4.3. Разработка метода применения монад для представления структурных автоматов Мили с выходными воздействиями

Одной из наиболее значимых встроенных конструкций языка *Haskell* являются *монады*. С их помощью, например, осуществляется ввод-вывод, который не нарушает функциональные основы языка. О монадах можно думать, как о способе объединения последовательности действий – в некотором смысле способ «эмулировать» императивное программирование в рамках функционального. Более подробно о монадах можно прочитать в работах [7, 96]. Из сказанного не следует, что монады невозможно реализовать во многих функциональных языках программирования, однако в *Haskell* они используются повсеместно.

Поэтому логичным развитием идеи реализации конечных автоматов на *Haskell* является попытка построить монаду, удобную в использовании.

4.3.1. Монада для реализации автоматов

Снова обратимся к счетному триггеру на два состояния (рис. 14). Однако теперь реализуем этот конечный автомат с помощью монады, которую назовем *FSM*, реализовав небольшую «библиотеку». Исходный код данной реализации представлен в листинге 10.

Приведенная реализация библиотечного кода для монады *FSM* содержит следующие функции для выполнения операций над состояниями:

- *getState* – получает текущее значение состояния конечного автомата;
- *setState* – устанавливает текущее состояние конечного автомата в определенное значение;
- *applyEvent* – производит переход по событию при помощи некоторой функции переходов;
- *applyEvents* – производит переход по списку событий;
- *applyFSM* – вычисляет результат выполнения монады *FSM*.

По сравнению с листингом 4 объем библиотечного исходного кода значительно увеличился. С другой стороны, появилась возможность запоминать состояние автомата в некоторый момент времени при помощи функции *getState* и устанавливать его в другом месте при помощи функции *setState*. Таким образом, возможно, например, заставить конечный автомат перейти «назад во времени».

Листинг 10. Реализация счетного триггера при помощи монады *FSM*

```
------------------------------------------------------
-- Библиотечный код.

-- Абстрактный тип функции перехода.
type SwitchFunc state event = state -> event -> state

-- Тип монадического преобразования состояния.
newtype FSM state a = FSM ( state -> (state, a) )

-- Реализация функций класса Monad.
instance Monad (FSM state) where
    (FSM first) >>= second =
        FSM ( \s0 -> let (s1, a) = first s0
                         (FSM q) = second a in q s1 )
    return a = FSM ( \s -> (s, a) )

-- Взятие текущего состояния.
getState :: FSM state state
getState = FSM ( \was -> (was, was) )

-- Установка текущего состояния.
setState :: state -> FSM state ()
setState state = FSM ( \s -> (state, ()) )

-- Функция для применения монады.
-- Передает начальное состояние первым аргументом.
applyFSM :: s -> FSM s a -> (s, a)
applyFSM s (FSM p) = p s

-- Переход по событию.
applyEvent :: SwitchFunc state event -> event -> FSM state ()
applyEvent switchFunc event = do
    st <- getState
    setState (switchFunc st event)

-- Переход по списку событий.
applyEvents :: SwitchFunc state event -> [event] -> FSM state
       ()
applyEvents _ [] = return ()
applyEvents switchFunc (event:eventsTail) = do
    applyEvent switchFunc event
    applyEvents switchFunc eventsTail

------------------------------------------------------
-- Собственно программа.

data Event = ButtonClick deriving Show
```

```
data State = LampOff | LampOn deriving Show

gotEvent :: State -> Event -> State
gotEvent state ButtonClick = case state of
    LampOff -> LampOn
    LampOn  -> LampOff

main = print $ fst $
    applyFSM LampOff $ applyEvents gotEvent [ButtonClick,
        ButtonClick]
```

4.3.2. Реализация с использованием монады *State*

Стандартная библиотека языка *Haskell* содержит монаду *State*, которая предоставляет обобщенный механизм манипуляции состояниями. В частности, эта монада может использоваться для реализации конечных автоматов.

Монада *State* предоставляет следующие функции для манипуляции состояниями:

- *get* – получает текущее значение состояния;
- *put* – устанавливает текущее состояние;
- *runState* – вычисляет результат выполнения монады *State*.

В приведенном листинге 11 функции *getState*, *setState*, *applyEvent*, *applyEvents*, *applyFSM* (интерфейс монады *FSM*) реализованы при помощи функций, предоставляемых стандартной монадой *State*. В основной части программы реализован счетный триггер на два состояния.

Таким образом, предложенная монада *FSM* из листинга 10 представляет собой альтернативную реализацию *State*, дополненную операциями специфичными для реализации конечных автоматов.

Листинг 11. Реализация счетного триггера при помощи монады *State*

```
------------------------------------------------------
-- Библиотечный код.

import Control.Monad.State as S

-- Абстрактный тип функции перехода.
type SwitchFunc state event = state -> event -> state

-- Тип монадического преобразования сотояния.
type FSM state a = S.State state a

-- Взятие текущего состояния.
getState :: FSM state state
getState = get

-- Установка текущего состояния.
setState :: state -> FSM state ()
setState state = put state

-- Функция для применения монады.
-- Передает начальное состояние первым аргументом.
applyFSM :: s -> FSM s a -> (s, a)
applyFSM s p = (\(a,b) -> (b,a)) (runState p s)

-- Переход по событию.
applyEvent :: SwitchFunc state event -> event -> FSM state ()
applyEvent switchFunc event = do
    st <- getState
    setState (switchFunc st event)

-- Переход по списку событий.
applyEvents :: SwitchFunc state event -> [event] -> FSM state
       ()
applyEvents _ [] = return ()
applyEvents switchFunc (event:eventsTail) = do
    applyEvent switchFunc event
    applyEvents switchFunc eventsTail

------------------------------------------------------
-- Собственно программа.

data Event = ButtonClick deriving Show
data AState = LampOff | LampOn deriving Show

gotEvent :: AState -> Event -> AState
gotEvent state ButtonClick = case state of
    LampOff -> LampOn
```

```
    LampOn  -> LampOff

main = print $ fst $
    applyFSM LampOff $
        applyEvents gotEvent [ButtonClick, ButtonClick]
```

4.3.3. Применимость подхода

Проблемы реализаций с использованием монад состоят в том, что они не несут какой-либо практической пользы при реализации конечных автоматов общего вида.

Можно предположить, что их применение может быть удобно, при разработке автоматов, результат вычисления конечного состояния которых имеет всего два возможных варианта: «допустить» и «отвергнуть». Поскольку последовательность действий монады в таком случае представляет собой некоторый «путь», на каждом шаге которого можно решить, что данный путь не является искомым и, возможно, перейти к поиску другого.

Такое применение монад активно используется при разработке парсеров [9], однако реализация структурных автоматов при помощи монад, по всей видимости, не дает никаких дополнительных преимуществ по сравнению с реализациями при помощи обычных рекурсивных функций.

4.3.4. Выходные воздействия с использованием монады *IO*

Среди рассмотренных примеров только функция *systemSwF* из листинга 9 формировала выходные воздействия, передавая сообщения от внешнего автомата вложенному. В других примерах функции переходов автоматов самостоятельно не выполняли выходных воздействий, а только возвращали результирующее состояние, которое печаталось на консоль функцией *main*. На практике автоматам требуется выводить сообщения на экран, отправлять пакеты данных в сеть, обмениваться событиями и т. д. Поэтому необходимо реализовать более общие механизмы реализации выходных воздействий.

В качестве примера реализации автомата с выходными воздействиями рассмотрим счетный триггер с четырьмя состояниями. Его отличие от предыдущего триггера состоит в том, что у него два входных события: нажатие и отпускание кнопки. Лампа будет включаться или выключаться только после того, как кнопка будет отпущена, а повторное нажатие или отпускание не будет производить никакого эффекта. Диаграмма переходов данного триггера представлена на рис. 16.

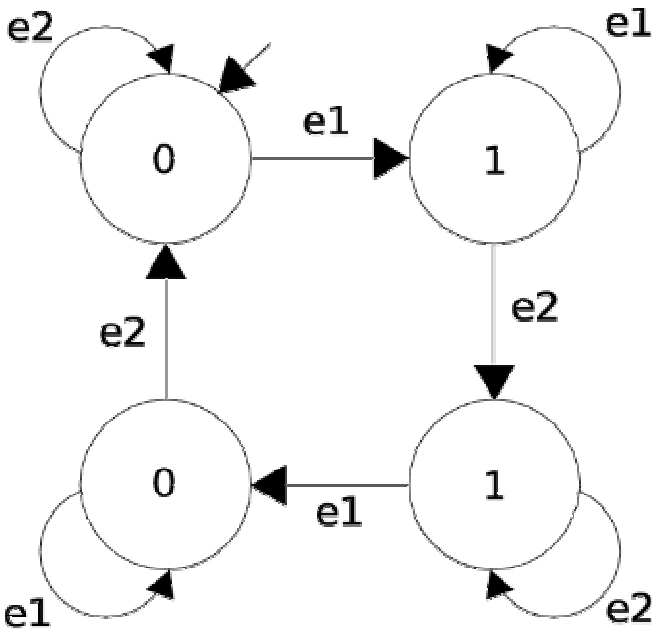


Рис. 16. Диаграмма переходов счетного триггера с четырьмя состояниями [2]. «e1», «e2» – нажатие и отпускание кнопки соответственно

Поскольку в языке *Haskell* все выходные воздействия рано или поздно становятся вычислениями монады *IO*, то наиболее простым способом реализации выходных воздействий является модификация функции переходов автомата таким образом, чтобы она в качестве типа возвращаемого значения имела *IO state*, где *state* – тип данных состояния автомата. В этом случае все выходные воздействия можно выполнять непосредственно в функции переходов. Пример такой реализации представлен в листинге 12.

Листинг 12. Монада *IO* в функции переходов счетного триггера

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show
data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show

-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и осуществляющей IO.
type SwitchFunc state input output = state -> input -> IO state

-- Функция applyEvents, реализованная через монаду IO
applyEvents :: SwitchFunc state input output ->
    state -> [input] -> IO state
applyEvents switchFunc state [] = return state
applyEvents switchFunc state (event:eventsTail) = do
    newState <- switchFunc state event
```

```
    applyEvents switchFunc newState eventsTail

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case state of
    LampOffButtonUp -> case event of
        ButtonDown -> return LampOffButtonDown
        _ -> return state
    LampOnButtonUp -> case event of
        ButtonDown -> return LampOnButtonDown
        _ -> return state
    LampOffButtonDown -> case event of
        ButtonUp -> do
            print "LampOn"
            return LampOnButtonUp
        _ -> return state
    LampOnButtonDown -> case event of
        ButtonUp -> do
            print "LampOff"
            return LampOffButtonUp
        _ -> return state

-- Функция, вызываемая системой.
main = do
    result <- applyEvents triggerSwitchFunc LampOffButtonUp
        [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
    print result
```

Такой подход предоставляет много свободы, так как в данном случае функция переходов (в отличие от предыдущих) не является чистой. Данный способ реализации является аналогом подхода, используемого в императивных языках программирования, поскольку все вычисления выполняются строго в контексте *IO*. Однако такая функция переходов не может быть использована в контексте чистых вычислений и, кроме того, несет в себе потенциальные опасности, поскольку компилятор в большинстве случаев не способен проверить корректность вычислений, производимых внутри функции переходов.

4.4. Разработка метода представления автоматов Мура и смешанных автоматов

В этом разделе предложены методы реализации автоматов с выходными воздействиями без использования монад, методы декомпозиции функций переходов таких автоматов, а также специальный синтаксис для представления автоматов Мура и смешанных автоматов. Кроме того, вводится понятие активных автоматов, и предложен метод их реализации на языке программирования *Haskell*.

4.4.1. Возврат списка выходных воздействий

Первым вариантом реализации выходных воздействий в автоматах Мили является подход, изложенный в разд. 4.3.4, однако поскольку использование монады *IO* для реализации выходных воздействий не может обеспечить уровня контроля со стороны компилятора, доступного при использо-

вании чистых функций, то необходимо разработать альтернативный метод реализации выходных воздействий без применения монады *IO*.

В качестве такого метода реализации выходных воздействий предлагается изменять функции переходов автоматов таким образом, чтобы она возвращала не только новое состояние автомата, но и список элементов некоторого типа. Это может быть как алгебраический тип данных, представляющий собой множество всех возможных выходных воздействий, так и тип *IO ()*. В этом случае ответственность за осуществление воздействий лежит на той части кода, которая вызывает функцию переходов автомата, однако сама функция переходов остается чистой функцией в не зависимости от того, какие выходные воздействия она формирует.

Отметим, что данный метод является логическим продолжением метода, предложенного в листинге 9, а общий принцип построения функции переходов в данном методе заключается в использовании метода, предложенного в разд. 4.1, с тем отличием, что в качестве описания перехода применяется не только новое состояние, но и список выходных воздействий. Листинг 13 демонстрирует данный подход на примере счетного триггера с четырьмя состояниями из разд. 4.3.4. (рис. 16). Кроме этого, данный подход используется во всех последующих реализациях, как наиболее гибкий.

Листинг 13. Список выходных воздействий в функции переходов счетного триггера

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show
data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show
-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и список выходных воздействий.
type SwitchFunc state input output = state -> input -> (state,
       [output])

-- Функция applyEvents, реализованная через свертку.
applyEvents :: SwitchFunc state input output ->
    state -> [input] -> (state, [output])
applyEvents switchFunc state events =
    foldl (switchToAcc switchFunc) (state, []) events
-- Функция «конвертирующая» функцию переходов
-- в аккумулятор выходных воздействий.
switchToAcc :: SwitchFunc state input output ->
    (state, [output]) -> input -> (state, [output])
switchToAcc switchFunc (state, output) event =
    (newState, output ++ newOutput)
    where (newState, newOutput) = switchFunc state event

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case state of
    LampOffButtonUp -> case event of
        ButtonDown -> (LampOffButtonDown, [])
        _ -> (state, [])
    LampOnButtonUp -> case event of
        ButtonDown -> (LampOnButtonDown, [])
        _ -> (state, [])
    LampOffButtonDown -> case event of
        ButtonUp -> (LampOnButtonUp, [print "LampOn"])
        _ -> (state, [])
    LampOnButtonDown -> case event of
        ButtonUp -> (LampOffButtonUp, [print "LampOff"])
        _ -> (state, [])

-- Функция, вызываемая системой.
main = do
    sequence_ o
    print s
    where
        (s, o) = applyEvents triggerSwitchFunc LampOffButtonUp
            [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
```

4.4.2. Декомпозиция функции переходов

В реальных задачах часто возникают ситуации, когда поведение автомата в двух и более состояниях отличаются только константами, известными во время компиляции. Для устранения дублирования кода в подобных ситуациях обычно реализуют *комбинаторы*, представляющие собой параметризуемые шаблоны, позволяющие лаконично записывать вычисления. Например, для автоматов с выходными воздействиями часто используемыми комбинаторами, описывающими переходы, являются: «перейти в некоторое состояние, не совершая выходных воздействий» и «остаться в текущем состоянии».

Библиотечный код в листинге 14 содержит обобщения использованных в предыдущих разделах типов данных и функций, а также три базовых комбинатора для описания переходов.

Листинг 14. Библиотечный код для выбранного представления функции переходов

```
-- Тип функции переходов.
type StateTransition state event output =
    state -> event -> (state, [output])

-- Комбинатор <<просто перейти в состояние state>>.
pure :: state -> (state, [output])
pure state = (state, [])

-- Комбинатор <<перейти в состояние state с
-- одним выходным воздействием output>>.
blot :: state -> output -> (state, [output])
blot state output = (state, [output])

-- Комбинатор <<перейти в состояние state со
-- списком выходных воздействий output>>.
dark :: state -> [output] -> (state, [output])
dark state output = (state, output)
--

-- Тип функции переходов, пригодной для использования вместе с
       foldl.
type FoldableTransition state event output =
    (state, [output]) -> event -> (state, [output])

-- Преобразователь из StateTransition в FoldableTransition
stt2foldable :: StateTransition state event output ->
    FoldableTransition state event output
stt2foldable transition (pstate, accumulator) event =
    (nstate, accumulator ++ output)
    where (nstate, output) = transition pstate event
```

4.4.3. Декомпозиция функции переходов по состояниям

Функцию переходов можно представить, как некоторый набор состояний, где каждое из них содержит внутри себя всю свою логику (реакцию на входные воздействия). Каждое состояние можно получить путем параметризации константами некоторого шаблона. Таким образом, функцию перехо-

дов автомата можно декомпозировать в набор параметризованных *состояний-шаблонов*, где тело функции переходов принимает решение только о том, какому из шаблонов передать управление.

В листинге 15 представлен пример реализации декомпозиции функции переходов счетного триггера с четырьмя состояниями (рис. 16).

Листинг 15. Пример декомпозиции функции переходов счетного триггера

```
-- Библиотечный код
...

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show
data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show

-- Шаблон состояния.
statePattern ifUp ifDown event = case event of
    ButtonUp   -> pure ifUp
    ButtonDown -> pure ifDown

-- Функция переходов. (\x -> x e) --- лямбда-функция,
-- в данном примере позволяющая не добавлять " e" в конец
       каждой
-- строки веток оператора case.
triggerSwF :: StateTransition TriggerState TriggerEvent ()
triggerSwF s e = (\x -> x e) $ case s of
    LampOffButtonUp   -> statePattern s LampOffButtonDown
    LampOffButtonDown -> statePattern LampOnButtonUp s
    LampOnButtonUp    -> statePattern s LampOnButtonDown
    LampOnButtonDown  -> statePattern LampOffButtonUp s

main = print $ foldl (stt2foldable triggerSwF)
    (pure LampOffButtonUp) [ButtonDown, ButtonDown]
```

4.4.4. Декомпозиция с переименованием событий

Ограничением состояния-шаблона *statePattern*, представленного в листинге 15, является то, что оно может быть использовано только для работы со строго определенным типом событий. Дело в том, что левая часть ветки оператора *case* (до стрелки) должна быть константой и не может быть представлена как аргумент шаблона.

Однако данное ограничение можно обойти, введя возможность переименования событий при помощи вспомогательных функций. Такая реализация представлена в листинге 16.

Листинг 16. Пример декомпозиции с переименованием событий

```
-- Исходный код предыдущего листинга, за исключением
-- функций statePattern и triggerSwF
...

-- Тип событий шаблона состояния.
data StatePatternEvent = BUp | BDown

-- Шаблон состояния.
statePattern ifUp ifDown event = case event of
    BUp   -> pure ifUp
    BDown -> pure ifDown

-- Переименование событий.
evMap :: TriggerEvent -> StatePatternEvent
evMap ButtonUp = BUp
evMap ButtonDown = BDown

-- Функция переходов. (\x -> x $ evMap e) --- лямбда-функция,
-- в данном примере позволяющая не добавлять
-- " (evMap e)" в конец каждой строки веток оператора case.
triggerSwF :: StateTransition TriggerState TriggerEvent ()
triggerSwF s e = (\x -> x $ evMap e) $ case s of
    LampOffButtonUp   -> statePattern s LampOffButtonDown
    LampOffButtonDown -> statePattern LampOnButtonUp s
    LampOnButtonUp    -> statePattern s LampOnButtonDown
    LampOnButtonDown  -> statePattern LampOffButtonUp s
```

В рассматриваемом листинге функция *evMap* применяется к каждому событию для всех веток оператора *case*, однако если убрать лямбда-функцию из тела *triggerSwF* и передавать событие каждому состоянию-шаблону непосредственно, то различные шаблоны могут работать с различными типами событий не только по отношению к функции переходов, но и по отношению друг к другу.

Таким образом, функцию переходов автомата можно составлять из абсолютно независимых частей, а элементы одной функции переходов – повторно использовать в другой.

4.4.5. Автоматы Мура и смешанные автоматы

Все предыдущие автоматы являлись автоматами Мили, поскольку функции их переходов представляли собой отображения $E \times Y \to Z$, где E – множество событий, Y – множество состояний автомата, а Z – множество выходных воздействий. Для автоматов Мура функции переходов имеют вид отображения $Y \to Z$. Иначе говоря, выходное воздействие автомата Мура зависит только от состояний, в которых находится автомат до и после перехода по событию [2].

Любой автомат Мура можно преобразовать в автомат Мили, соответствующим образом изменив список выходных воздействий. Целью данного раздела является разработка синтаксиса, который позволит записывать функцию переходов автомата в форме Мура, а преобразование в форму Мили будет производиться автоматически.

Листинги текущего раздела предполагают использование библиотечного кода из листинга 14.

4.4.6. Синтаксис для автоматов Мура

Суть предлагаемого синтаксиса основана на использовании метода, предложенного в разд. 4.1, но с тем отличием, что представление каждого состояния (в левой части оператора *case* верхнего уровня) производится тройкой элементов: выходные воздействия при входе в состояние, отображение из множества событий на множество состояний (функция переходов без выходных воздействий), выходные воздействия при выходе из состояния. Реализация данного подхода на примере счетного триггера с двумя состояниями представлена в листинге 17.

Листинг 17. Реализация счетного триггера автоматом Мура

```
-- Библиотечный код
...

-- Функция-преобразователь из автомата Мура в автомат Мили.
moore2mealy transition pstate event = (nstate, outr ++ inr)
    where
        (_, trans, outr) = transition pstate
        nstate = trans event
        (inr, _, _) = transition nstate

-------------------------------------------------------
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Функция переходов для счетного триггера.
triggerSwFMoore state = case state of
    LampOff -> ([print "In Off"], (\e -> case e of
        ButtonClick -> LampOn),  [print "Out Off"])
    LampOn  -> ([print "In On"],  (\e -> case e of
        ButtonClick -> LampOff), [print "Out On"])

triggerSwF = moore2mealy triggerSwFMoore

-- Функция, вызываемая системой.
main = do
    sequence_ o
    print s
    where
        (s, o) = foldl (stt2foldable triggerSwF)
            (pure LampOff) [ButtonClick, ButtonClick]
```

Недостатком рассматриваемого листинга является то, что если переход производится из состояния в само себя, то сначала выполняются выходные воздействия выхода из текущего состояния, а потом выходные воздействия входа в текущее состояние. Такое поведение обеспечивается не для всех автоматов Мура, поскольку существуют автоматы, для которых предполагается, что отсутствие перехода в новое состояние означает, что выходных воздействий не формируется.

Данный недостаток можно устранить двумя способами:

- модифицировать функцию *moore2mealy* так, чтобы она принимала еще один параметр – функцию, решающую следует ли выполнять выходные воздействия при переходе из текущего состояния в то, которое должно стать следующим;
- ввести возможность делать утверждения вида «автомат остается в текущем состоянии» и модифицировать функцию *moore2mealy* так, чтобы она это учитывала.

Реализация первой модификации представлена в листинге 18, а второй – в листинге 19. Для наглядности функция переходов автомата счетного триггера в этих листингах модифицирована так, чтобы после входа в состояние *LampOn* счетный триггер больше не переходил в состояние *LampOff*.

Листинг 18. Первая модификация

```
-- Библиотечный код
...

-- Модифицированная функция-преобразователь
-- из автомата Мура в автомат Мили.
moore2mealy ifss transition pstate event = (nstate, output)
    where
        (_, trans, outr) = transition pstate
        nstate = trans event
        (inr, _, _) = transition nstate
        output = if (ifss pstate nstate)
            then outr ++ inr
            else []

-------------------------------------------------------
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
-- Новый тип TriggerState, на этот раз
-- его элементы можно сравнивать на равенство.
data TriggerState = LampOff | LampOn deriving (Show, Eq)

-- Модифицированная функция переходов для счетного триггера.
triggerSwFMoore state = case state of
    LampOff -> ([print "In Off"], (\e -> case e of
        ButtonClick -> LampOn), [print "Out Off"])
    LampOn  -> ([print "In On"],  (\e -> case e of
        ButtonClick -> LampOn), [print "Out On"])

doOnlyIfChanged a b = a /= b

triggerSwF = moore2mealy doOnlyIfChanged triggerSwFMoore

-- Функция, вызываемая системой.
main = do
    sequence_ o
    print s
    where
        (s, o) = foldl (stt2foldable triggerSwF)
            (pure LampOff) [ButtonClick, ButtonClick]
```

Листинг 19. Вторая модификация

```
-- Библиотечный код
...

-- Модифицированная функция-преобразователь
-- из автомата Мура в автомат Мили.
moore2mealy transition pstate event = (nstate, output)
    where
        (_, trans, outr) = transition pstate
        xstate = trans event
        (inr, _, _) = transition nstate
        (nstate, output) = case xstate of
            Nothing -> (pstate, [])
            Just x -> (x, outr ++ inr)

------------------------------------------------------
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Модифицированная функция переходов для счетного триггера.
triggerSwFMoore state = case state of
    LampOff -> ([print "In Off"], (\e -> case e of
        ButtonClick -> Just LampOn), [print "Out Off"])
    LampOn  -> ([print "In On"],  (\e -> case e of
        ButtonClick -> Nothing),     [print "Out On"])

triggerSwF = moore2mealy triggerSwFMoore

-- Функция, вызываемая системой.
main = do
    sequence_ o
    print s
    where
        (s, o) = foldl (stt2foldable triggerSwF)
            (pure LampOff) [ButtonClick, ButtonClick]
```

Отметим, что вторая модификация является более гибкой, поскольку в ней, в частности, не требуется существования операции сравнения на типе данных событий автомата.

4.4.7. Синтаксис для смешанных автоматов

Если вторым элементом кортежа, возвращаемого функцией переходов, сделать функцию, возвращающую новое состояние и список событий, то таким образом можно представлять смешанные автоматы – автоматы, у которых явно указаны воздействия, выполняемые не только при входе и выходе из состояния, но и на каждом ребре перехода в диаграмме состояний.

Взяв за основу листинг 19, модифицируем его так, чтобы функция переходов автомата была смешанной. Полученный результат представлен в листинге 20.

Листинг 20. Реализация счетного триггера смешанным автоматом

```
-- Библиотечный код
...

-- Модифицированная функция-преобразователь
-- из смешанного автомата в автомат Мили.
mixed2mealy transition pstate event = (nstate, output)
    where
        (_, trans, outr) = transition pstate
        xstate = trans event
        (inr, _, _) = transition nstate
        (nstate, output) = case xstate of
            Nothing -> (pstate, [])
            Just (x, o) -> (x, outr ++ o ++ inr)

------------------------------------------------------
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Модифицированная функция переходов для счетного триггера.
triggerSwFMoore state = case state of
    LampOff -> ([print "In Off"], (\e -> case e of
        ButtonClick -> Just $
            blot LampOn (print "Some out")), [print "Out Off"])
    LampOn  -> ([print "In On"],  (\e -> case e of
        ButtonClick -> Nothing), [print "Out On"])

triggerSwF = mixed2mealy triggerSwFMoore

-- Функция, вызываемая системой.
main = do
    sequence_ o
    print s
    where
        (s, o) = foldl (stt2foldable triggerSwF)
            (pure LampOff) [ButtonClick, ButtonClick]
```

4.5. Пример применения разработанных методов

В настоящем разделе производится применение разработанных методов реализации конечных автоматов на функциональных языках программирования на примере активного счетного триггера.

4.5.1. Активные автоматы

В рассмотренных ранее примерах заранее была известна вся последовательность событий, которые получит автомат, а выходные воздействия всегда имели тип *IO ()*. В данном разделе будет продемонстрировано получение событий из внешнего мира, а также представление выходных воздействий специальными типами данных. Кроме того, будет представлена практическая реализации автомата как потока (*thread*) программы.

Пусть состояние автомата содержит *источники*, при помощи некоторой операции над которыми можно получить следующее событие. Источниками могут быть как числовые идентификаторы (например, файловый дескриптор *UNIX*), так и ссылки на объекты в памяти, константные списки событий и т. п.

Тогда применим операцию извлечения следующего события из источника. Полученное событие передадим функции переходов автомата. Выполним выходные воздействия. После этого новое состояние примем за текущее. Таким образом можно реализовать автомат, работающий в отдельном потоке (будем называть такие автоматы *активными*). Однако при этом нет ответа вопрос: когда этот автомат должен остановить свое исполнение?

Вариантов решения данной проблемы два:

- ввести специальное выходное воздействие, означающее «завершить исполнение»;
- ввести специальное состояние, попав в которое поток завершается («конечное состояние»).

Поскольку реализация первого решения может привести к неопределенности в том случае, если в списке выходных воздействий после воздействия «завершить исполнение» находятся еще элементы, то в настоящей работе был выбран второй вариант.

Ввести одно конечное состояние на все автоматы невозможно, так как множества их состояний представлены различными типами данных, однако можно ввести класс (в терминах языка *Haskell*) с одной операцией, которая будет определять, является ли текущее состояние конечным.

Таким образом, активный автомат оперирует кортежем из множества источников и состояния автомата, функция входов и функция выходов оперируют множеством источников, а функция переходов только состоянием автомата.

В листинге 21 представлен библиотечный код, необходимый для реализации активных автоматов, где «старый библиотечный код» – код листинга 14.

Листинг 21. Библиотечный код для реализации активных автоматов

```
-- Старый библиотечный код
...

class AutomataState state where
    lastState :: state -> Bool

-- Структура, представляющая отдельный активный автомат.
data IOAutomata source state event output = IOAutomata
    -- Получение событий из внешнего мира.
    (source -> IO (source, [event]))
    -- Функция переходов.
    (state -> event -> (state, [output]))
    -- Функция осуществления выходных воздействий.
    (source -> output -> IO source)
    -- Стартовое состояние системы.
    (source, state)

-- Исполняет активный автомат.
runIOAutomata (IOAutomata ep stt og psystem) =
    runIOAutomata' ep stt og psystem

runIOAutomata' ep stt og (psource, pstate) = do
    (xsource, events) <- ep psource
    let (nstate, outputs) = foldl (stt2foldable stt) (pure
        pstate) events
    nsource <- foldlM og xsource outputs
    if lastState nstate
        then return ()
        else runIOAutomata' ep stt og (nsource, nstate)
    where
        foldlM _ s [] = return s
        foldlM ofunc s (o:os) = do
            ns <- ofunc s o
            foldlM ofunc ns os
```

4.5.2. Активный счетный триггер

В листинге 22 при помощи предложенного выше библиотечного кода реализован счетный триггер в виде активного автомата, получающего события со стандартного ввода приложения. Поскольку у данного триггера источником является стандартный ввод, то структура, представляющая источники активного автомата, в данном примере – пустой кортеж.

Листинг 22. Библиотечный код для реализации активных автоматов

```
--- Библиотечный код
...

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Тип выходных воздействий.
data TriggerOutput = SayOn | SayOff

-- Конечное состояние.
instance AutomataState TriggerState where
    lastState LampOff = True
    lastState _ = False

-- Получение событий из внешнего мира.
eventF state = do
    print "Say: click"
    x <- getLine
    return $ if x == "click"
        then blot state ButtonClick
        else pure state

-- Функция переходов.
triggerSwF state ButtonClick = case state of
    LampOff -> blot LampOn SayOn
    LampOn  -> blot LampOff SayOff

-- Функция выходов.
outF state output = do
    case output of
        SayOn  -> print "On"
        SayOff -> print "Off"
    return state

main = runIOAutomata (IOAutomata
    eventF triggerSwF outF ((), LampOff))
```

Заметим, что в данной модели только функция переходов автомата может модифицировать состояние. Поэтому, если бы производилось рассмотрение реализации, например, клиент-серверного приложения, то в серверном автомате следовало бы хранить список клиентов в структуре источников. В функции получения события такого автомата – просматривать этот список на наличие новых сообщений от клиентов. Функции выходов также может потребоваться множество источников автомата, например, для того, чтобы найти файловый дескриптор клиента, которому следует отправить сообщение. Поэтому определение типа *IOAutomata* имеет вид, представленный в листинге 21.

Таким образом, предложенная модель реализации активных автоматов в целом пригодна для практического использования.

Выводы по главе 4

1. Разработан метод реализации на функциональных языках программирования структурных конечных автоматов, обеспечивающий валидацию на этапе компиляции.
2. Разработан метод представления вложенных автоматов
3. Разработан метод применения монад для представления структурных автоматов Мили с выходными воздействиями.
4. Разработаны методы декомпозиции функции переходов и метод представления автоматов Мура и смешанных автоматов.
5. Разработанные методы применены на примере реализации активного счетного триггера.

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на первом этапе работ по контракту, были выполнены:

- аналитический обзор;
- выбор и обоснование направления исследований;
- подготовка плана проведения теоретических и экспериментальных исследований;
- разработка методов реализации автоматов на функциональных языках программирования;

Аналитический обзор был выполнен по следующим направлениям:

- качество программ;
- автоматное программирование;
- функциональное программирование;
- реализация автоматов на функциональных и объектно-ориентированных языках программирования;
- методы повышения качества объектно-ориентированных программ;
- методы повышения качества автоматных объектно-ориентированных программ;
- обзор методов и средств верификации на модели.

В результате выполнения аналитического обзора разработан план проведения теоретических и экспериментальных исследований. В соответствии с этим планом в рамках теоретических исследований были разработаны следующие методы:

- метод представления функций переходов автоматов без выходов в функциональных языках программирования на основе алгебраических типов данных;
- метод представления вложенных автоматов;
- метод применения монад для представления структурных автоматов Мили с выходными воздействиями;
- метод представления автоматов Мура и смешанных автоматов.

Результаты выполненных работ позволяют утверждать, что научно-технический уровень исследований превышает уровень исследований в рассматриваемой области, проводимых в лучших исследовательских центрах мира.

ИСТОЧНИКИ

1. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.
2. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2009.
3. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
4. *Abelson H., Sussman J., Gerald J.* Structure and Interpretation of Computer Programs. MIT Press, 1985.
5. *Bird R.* Introduction to Functional Programming using Haskell. NY: Prentice Hall, 1998.
6. *Davie A.* Introduction to Functional Programming System Using Haskell. Cambridge: Cambridge University Press, 1992.
7. *Hudak P., Peterson J., Fasel J.* A gentle introduction to Haskell 98.
8. *Stroustrup B.* The C++ Programming Language. Boston: Addison-Wesley, 2000.
9. *Parsec.* http://www.haskell.org/haskellwiki/Parsec.
10. The parser generator for Haskell. http://www.haskell.org/happy/.
11. *Грэхем И.* Объектно-ориентированные методы. М.: Вильямс, 2004.
12. *Ларман К.* Применение UML и шаблонов проектирования. М.: Вильямс, 2002.
13. *Kurzweil R.* The Singularity Is Near: When Humans Transcend Biology. Penguin, 2006.
14. *Beck K., Andres C.* Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2004.
15. *Schwaber K.* Agile Project Management with Scrum. Microsoft Press, 2004.
16. *Turner M.* Microsoft Solutions Framework Essentials. Microsoft Press, 2006.
17. *Alexander C.* A Pattern Language: Towns, Buildings, Construction. USA: Oxford University Press, 1977.
18. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.
19. *Фаулер М.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2004.
20. *Канер С., Фолк Д., Нгуен Е.К.*. Тестирование программного обеспечения. Киев: ДиаСофт, 2000.
21. *Ахо А., Лам М., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии и инструментарий. Вильямс, 2008.
22. *Бек К.* Экстремальное программирование: разработка через тестирование. СПб: Питер, 2003.
23. *Lyu M., Horgan J., London S.* A coverage analysis tool for the effectiveness of software testing //IEEE Transactions on Reliability. 1994. № 43(4), с. 527–535.
24. *Wikipedia: List of Unit Testing Frameworks.* http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
25. *Веб-сайт каркаса JUnit.* http://junit.org/
26. *Веб-сайт каркаса NUnit.* http://www.nunit.com/
27. *Веб-сайт каркаса PyUnit.* http://pyunit.sourceforge.net/
28. *Дюваль П., Матиас С., Гловер Э.* Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска. Вильямс, 2008.
29. *Wing J. M.* Writing Larch interface language specifications //ACM Trans. Program. Lang. Syst. 1987. № 9, с. 1–24.
30. *Веб-сайт проекта Code Contracts компании Microsoft.* http://research.microsoft.com/en-us/projects/contracts/
31. *Веб-сайт проекта Contract4j.* http://www.contract4j.org/contract4j

32. *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, с. 27–56.
33. *Шалыто А. А., Туккель Н. И.* Программирование с явным выделением состояний //Мир ПК. 2001. № 8, 9, с. 116–121, 132–138. http://is.ifmo.ru/works/mirk/
34. *Шопырин Д. Г., Шалыто А. А.*. Объектно-ориентированный подход к автоматному программированию //Информационно-управляющие программы. 2003. № 5, с. 29–39. http://is.ifmo.ru/works/ooaut/
35. *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. 2008.
36. *Степанов О. Г.* Предметно-ориентированный язык автоматного программирования на базе динамического языка *Ruby*. Магистерская диссертация. 2006.
37. *Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А.* State Machine – новый паттерн объектно-ориентированного проектирования //Информационно-управляющие программы. 2004. № 5, с.13–25. http://is.ifmo.ru/works/pattern/
38. *Наумов Л. А., Шалыто А. А.* Искусство программирования лифта. Объектно-ориентированное программирование с явным выделением состояний //Информационно-управляющие программы. 2003. № 6, с. 38–49.
39. *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/
40. *Астафуров А. А., Шалыто А. А.* Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования. 2007.
41. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования. 2008. http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf
42. *Дмитриев С.* Языково-ориентированное программирование: следующая парадигма. 2005. http://www.rsdn.ru/article/philosophy/LOP.xml
43. *Фаулер М.* Языковой инструментарий: новая жизнь языков предметной области. http://www.maxkir.com/sd/languageWorkbenches.html
44. *Acts As State Machine*. http://agilewebdevelopment.com/plugins/acts_as_state_machine
45. *Степанов О. Г., Шалыто А. А., Шопырин Д. Г.* Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby. http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf
46. *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
47. *Бурдонов И. Б., Косачев А. С., Кулямин В. В.* Использование конечных автоматов для тестирования программ //Программирование. 2000. № 26, с. 61–73.
48. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Инструментальное средство для поддержки автоматного программирования //Программирование. 2007. № 6, с. 38–50.
49. *Гуров В. С., Мазин М. А., Шалыто А. А.* Ядро автоматного программирования // Свидетельство об официальной регистрации программы для ЭВМ. № 2006 613249 от 14.09.2006.
50. *Грис Д.* Наука программирования. М.: Мир, 1984.
51. *Непомнящий В. А., Рякин О. М.* Прикладные методы верификации программ. М.: Радио и связь, 1988.
52. *Кларк Э.М., Грамберг О., Пелед Д.*. Верификация моделей программ. Model Checking.. М.: МЦНМО, 2002.
53. *Pnueli A.*. The Temporal Logic of Programs. 1977.

54. *Кузьмин Е. В., Соколов В. А.* Моделирование, спецификация и верификация «автоматных» программ //Программирование. 2008. № 1, с. 22–32.
55. *Вельдер С. Э., Шалыто А. А.* О верификации простых автоматных программ на основе метода *Model Checking* //Информационно-управляющие программы. 2007. № 3, с. 27–38.
56. *Яминов Б. Р., Шалыто А. А.* Расширение верификатора *Bogor* для верификации автоматных UniMod-моделей //Свидетельство об официальной регистрации программы для ЭВМ. № 2008 611055 от 28.02.2008.
57. *Лукин М. А., Шалыто А. А.* Транслятор автоматной *UniMod*-программы во входной язык верификатора SPIN //Свидетельство об официальной регистрации программы для ЭВМ. № 2008 610473 от 25.01.2008.
58. *Kurbatsky E.* Verification of Automata-Based Programs /proceedings of the Second Spring young Researchers` Coloquium on Software Engineering. SPbSU 2008, pp. 15–17.
59. *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications: A Practical Approach / In Proceedings of the 10th Annual ACM Symposium on Principles of Programming Language. Austin. 1983, pp. 117−126.
60. *Vardi M.Y., Wolper P.* Reasoning about infinite computations // Information and Computation. 1994. V. 115, pp. 1−37.
61. *Gerth R., Peled D., Vardi M. Y., Wolper P.* Simple on-the-fly automatic verification of linear temporal logic / In Proceedings of 15th Workshop Protocol Specification, Testing, and Verification. Warsaw: Chapman & Hall. 1995, pp. 3−18.
62. *Courcoubetis C., Vardi M.Y., Wolper P., Yannakakis M.* Memory efficient algorithms for the verification of temporal properties // Formal Methods in System Design. 1992. V. 1, p. 275−288.
63. *Ахо А., Ульман Д., Хопкрофт Д.* Структуры данных и алгоритмы. М.: Вильямс. 2000.
64. *Vardi M. Y., Wolper P.* Automata-theoretic techniques for modal logics of programs // Journal of Computer and System Science. 1986. V. 32(2), pp. 182–221.
65. *Brayant R.* The complexity of propositional linear temporal logics //Journal of the ACM. 32(3). 1985, pp. 733–749.
66. *Burch J. R., Clarke E. M., Dill D. L., Hwang L. J., McMillan K. L.* Symbolic model checking: 1020 states and beyond // Information and Computation. 1990. V. 98, I. 2, pp. 142−170.
67. *McMillan K. L.* Symbolic model checking: an approach to the state explosion problem. PhD thesis. SCS. Carnegie Mellon University, 1992.
68. *Burch J. R., Clarke E. M., Long D. E.* Symbolic model checking with partitioned transition relations / International Conference on Very Large Scale Integration. Edinburgh: North-Holland. 1991, pp. 49−58.
69. *Burch J. R., Clarke E. M., Dill D. L., Long D. E., McMillan K. L.* Symbolic model checking for sequential circuit verification // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1994. V. 13, I. 14, pp. 401−424.
70. *Clarke E. M., Emerson E. A.* Design and synthesis of synchronization skeletons using branching time temporal logic // Logic of Programs. 1981. V. 131, pp. 52−71.
71. *Clarke E., Grumberg O., Hiraishi H., Jha S., Long D., McMillan K.L., Ness, L.* Verification of the Futurebus + cache coherence protocol // Formal Methods In System Design, 1995. V. 6, I. 2, pp. 217 − 232.
72. *Valmari A.* Error detection by reduced reachability graph generation / Proc. 9th International Conference On Application And Theory Of Petri Nets. Venice. 1988, pp. 95−112.
73. *Valmari A.* Heuristics for lazy state generation speeds up analysis of concurent systems /In Proc. Of the Finnish Artificial Intelligence Symposium STeP-88, V. 2, Helsinki. 1988, pp. 640−650.
74. *Valmari A.* Stubborn attack on state explosion. In Proc. 2nd Workshop on Computer Aided Verification. Lecture Notes in Computer Science. V.531. Springer-Verlag. 1990, pp. 156−165.

75. *Godefroid P.* Using partial orders to improve automatic verification methods / Proc. 2nd Workshop on Computer Aided Verification. Lecture Notes in Computer Science. V. 531. Springer-Verlag. 1990, pp. 176−185. Extended version in ACM/AMS DIMACS Series, V. 3, pp. 321−340, 1991.
76. *Godefroid P., Wolper P.* Using partial orders for efficient verification of deadlock freedom and safety properties / Proc. 3rd Workshop on Computer Aided Verification. Lecture Notes in Computer Science. V.575. Springer-Verlag. 1991, pp. 332−342.
77. *Godefroid P.* Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem. Lecture Notes in Computer Science. V.1032. Springer-Verlag. 1996, pp. 418–425.
78. *Godefroid P., Pirottin D.* Refining dependencies improves partial-order verification methods /Proc. 5nd Conference on Computer Aided Verification. Lecture Notes in Computer Science. V.697. Springer-Verlag. 1993, pp. 438−449.
79. *Holzmann G.J., Peled D.* An improvement in formal verification /Proc. FORTE'94. Bern. 1994, pp.177−191.
80. *Holzmann G.J.* The Model Checker *SPIN* // IEEE Transactions on software engineering. 1997. V. 23. I. 5.
81. *Goldberg E., Novikov Y.* Berkmin: A fast and robust *SAT*-solver /In DATA, 2002.
82. *Marques-Silva J., Sakallah K.* GRASP: A search algorithm for propositional *SAT*isfiability //IEEE Transactions on Computers. 1999. V. 48. № 5.
83. *Madigan C.F., Malik S., Moskewicz M.W., Zhang L., Zhao Y.* Chaff: engineering an efficient *SAT* solver /In DAC, 2001.
84. *Biere A., Cimatti A., Clarke E., Zhu Y.* Symbolic model checking without BDDs // Lecture Notes in Computer Science. V. 1579. Springer-Verlag. 1999, pp. 193−207.
85. *Ganai M., Krohm F., Kuehlmann A., Paruthi V.* Robust boolean reasoning for equivalence checking and functional property verification // IEEE TCAD, Vol. 21(12), 2002, pp. 1377−1394.
86. *McMillan K.L.* Interpolation and *SAT*-based model checking / CAV, 2003.
87. *Amla N., McMillan K.* Automatic abstraction without counterexamples // TACAS, 2003.
88. *Chauhan P., Clarke E., Kukula J., Sapra S., Veith H., Wang D.* Automated abstraction refinement for model checking large state spaces using *SAT* based conflict analysis / FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design. 2002. London: Springer-Verlag, pp. 33−51.
89. *McMillan K.L.* Applying *SAT* methods in unbounded symbolic model checking / CAV, 2003.
90. *Amla N., Du X., Kuehlmann A., Kurshan R., McMillan K.* An analysis of *SAT*-based model checking techniques in an industrial environment /CHARME. 2005, pp. 254−268.
91. *Biere A., Gupta A., Prasad M.* A survey of recent advances in *SAT*-based formal verification / STTT, 2005.
92. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО. 2002. 416 с.
93. *Васильева К.А., Кузьмин Е.В.* Верификация автоматных программ с использованием *LTL* // Моделирование и анализ информационных систем. 2007. Т. 14. № 1, с. 3–14.
94. *Robby, Dwyer M., Hatcliff J.* Bogor: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp. 3–22.
95. *Robby, Dwyer M., Hatcliff J.* Bogor: An Extensible and HighlyModular Model Checking Framework, March 2003. In the Proceeding soft he Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003). Technical Report, SAnToS-TR2003-3.
96. *Кирпичев Е.* Монады // RSDN Magazine. 2008. № 3.