На правах рукописи

Гуров Вадим Сергеевич

Технология проектирования и реализации объектно-ориентированных программ с явным выделением состояний (метод, инструментальное средство, верификация)

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

АВТОРЕФЕРАТ
диссертации на соискание ученой степени
кандидата технических наук

Санкт-Петербург
2008

Работа выполнена в Санкт-Петербургском государственном университете информационных технологий, механики и оптики

Научный руководитель:	доктор технических наук, профессор Шалыто Анатолий Абрамович
Официальные оппоненты:	доктор технических наук, профессор Тропченко Александр Ювенальевич
	кандидат физ.-мат. наук, доцент Новиков Федор Александрович
Ведущая организация	Санкт-Петербургский государственный электротехнический университет «ЛЭТИ»

Защита диссертации состоится «22» октября 2008 года в 14 часов на заседании диссертационного совета Д.212.227.06 при Санкт-Петербургском государственном университете информационных технологий, механики и оптики, 197101, Санкт-Петербург, ул. Кронверкский 49, СПбГУ ИТМО.

С диссертацией можно ознакомиться в библиотеке СПбГУ ИТМО.

Автореферат разослан 19 сентября 2008 г.

Ученый секретарь
Совета Д.212.227.06,
доктор технических наук,
профессор — Тарлыков Владимир Алексеевич

Общая характеристика работы

Актуальность проблемы. Современные программные системы, которые во многих случаях создаются с помощью объектно-ориентированных подходов, являются сложными. Для борьбы с этой сложностью непрерывно разрабатываются все новые средства, позволяющие увеличивать уровень абстракции и упрощать процесс программирования и проверки.

При создании программных систем обычно выделяют следующие фазы:

1. Постановка задачи – сбор требований и создание прототипа программы.
2. Проектирование – разработка проектной документации, отражающей структурные и поведенческие особенности создаваемой системы.
3. Реализация – создание на основе проекта кода для целевой программно-аппаратной платформы.
4. Тестирование – отладка кода и проверка соответствия реализации поставленной задаче.

Семантический разрыв при передаче знаний между проектированием и реализацией заключается в том, что разработчик обычно реализует систему в соответствии со своим пониманием проектной документации. Это приводит к ряду проблем:

1. Реализация системы не соответствует проектной документации ввиду неформальной связи фаз проектирования и реализации.
2. Проверка соответствия реализации проектной документации (верификация) может быть выполнена только вручную.
3. В случае необходимости изменений в системе, они вносятся в проектную документацию и в код программы независимо, что часто приводит к рассинхронизации документации и кода.

Причина указанных проблем кроется в том, что существуют методы проектирования объектно-ориентированных программ, которые позволяют моделировать их структуру, а также методы, позволяющие моделировать их поведение, но отсутствуют методы, которые обеспечивают связь статики и динамики в единую формальную модель.

Исследования, направленные на разработку таких методов и технологий для их поддержки, являются актуальными, так как позволят упростить процесс разработки и повысить качество создаваемых программ.

В настоящее время развивается автоматный подход к созданию программ, называемый автоматное программирование или программирование с явным выделением состояний, который обеспечивает возможность разработки указанных технологий, основанных, в том числе, и на графических языках программирования.

Цель диссертационной работы – разработка технологии проектирования и реализация объектно-ориентированных программ с явным выделением состояний.

Основные задачи исследования:

1. Создание метода проектирования объектно-ориентированных программ на основе автоматного подхода.
2. Разработка графического языка автоматного программирования.
3. Разработка методов верификации автоматных моделей программ.
4. Разработка инструментального средства для поддержки автоматного программирования.
5. Внедрение результатов работы в практику промышленной разработки программного обеспечения и в учебный процесс в СПбГУ ИТМО.

Научная новизна. На защиту выносятся следующие результаты, обладающие научной новизной:

1. Метод проектирования объектно-ориентированных программ с явным выделением состояний.

2. Графический язык для описания автоматных программ на основе *UML*-нотации.
3. Методы верификации автоматных моделей программ: метод верификации на модели (*Model Checking*), а также метод верификации полноты и непротиворечивости систем переходов автоматов.
4. Инструментальное средство для создания, верификации, отладки и запуска автоматных программ. При этом верификация на основе модели производится совместно с верификатором *Bogor*.

Перечисленные результаты получены в ходе выполнения в СПбГУ ИТМО научно-исследовательских и опытно-конструкторских работ по темам: «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода» (проводится по заказу Минобрнауки РФ с 2000 г. по настоящее время), «Разработка технологии автоматного программирования» (проводилась в 2002–2003 гг. по гранту Российского фонда фундаментальных исследований № 02-07-90114), «Разработка технологии объектно-ориентированного программирования с явным выделением состояний» (проводилась в 2005–2006 гг. по гранту Российского фонда фундаментальных исследований № 05-07-90011), «Технология автоматного программирования: применение и инструментальные средства» (государственный контракт, который выполнялся в 2005–2006 гг. в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития науки и техники»). Последняя работа вошла в список 15 наиболее перспективных проектов, выполняемых по этой программе.

Методы исследования. В работе использованы методы объектно-ориентированного проектирования, теории автоматов, теории формальных грамматик, теории графов, теории алгоритмов, теории верификации.

Достоверность научных положений и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами внедрения предложенной технологии.

Практическое значение полученных результатов состоит в том, что они успешно используются при разработке промышленных и учебных программных проектов на основе автоматного подхода.

Предложенные методы позволили устранить семантический разрыв между фазами проектирования и реализации за счет интерпретации автоматной модели программы или генерации изоморфного кода на целевом языке программирования.

За счет того, что автоматная модель системы является одновременно спецификацией и программой, процесс поддержания проектной документации в актуальном состоянии также значительно упростился.

Упрощается верификация на основе метода *Model Checking,* так как в автоматных моделях состояния явно выделены, и поэтому пространство состояний по сравнению с программами, построенными традиционным образом, резко сокращается.

Разработанная технология позволяет выявлять логические ошибки в автоматных программах на стадии проектирования, что уменьшает время разработки и время тестирования, и, как следствие, повышает качество программных продуктов.

Внедрение результатов работы. Результаты, полученные в диссертации, используются на практике в компании *eVelopers* (Санкт-Петербург) при разработке интернет-приложений для электронной коммерции и мобильных устройств, а также в компании *Intellij Labs* (Санкт-Петербург) при разработке мета-программирования *Meta Programming System*.

Полученные результаты используются также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при выполнении курсовых работ по курсу «Теория автоматов в программировании». При этом на сайте http://is.ifmo.ru в разделе

UniMod-проекты опубликовано около 30 проектов, выполненных с помощью предлагаемой технологии, которые содержат, в том числе, и проектную документацию.

Апробация диссертации. Основные положения диссертационной работы докладывались на конференциях и семинарах: II конференции молодых ученых СПбГУ ИТМО (2005 г.); XXXIII, XXXV, XXXVI научных учебно-методических конференциях СПбГУ ИТМО «Достижения ученых, аспирантов и студентов СПбГУ ИТМО в науке и образовании» (2003, 2005, 2006 гг.); «Телематика-2003», «Телематика-2004», «Телематика-2005», «Телематика-2006», «Телематика-2007» (СПб.); на семинаре «Автоматное программирование» в рамках международной конференции «International Computer Symposium in Russia (CSR 2006)» (ПОМИ им. Стеклова, 2006 г.); на конференциях «Software Engineering Conference in Russia» – SECR 2005 (Москва), «The International Scientific Conference «110-Anniversary of Radio Invention» (СПбГЭТУ, IEEE, 2005 г.); Второй Всероссийской научной конференции «Методы и средства обработки информации» (МГУ, 2005 г.); Open Source Forum (М.: Форт-Росс, 2005 г.); международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы» МВУС-2007 (Таганрог, 2007 г.); научно-технической конференции «Научно-программное обеспечение в образовании и научных исследованиях» (СПб., 2008 г.).

Публикации. По теме диссертации опубликовано 23 печатные работы, в том числе в журналах из списка ВАК «Программирование», «Информационно-управляющие системы» и «Научно-технический вестник СПбГУ ИТМО», а также в журнале «Технология клиент-сервер» и материалах указанных конференций и семинаров.

Свидетельства об официальной регистрации программ для ЭВМ. На инструментальное средство, разработанное в рамках диссертации, получены свидетельства: «Ядро автоматного программирования» №2006613249 от 14.09.2006, «Встраиваемый модуль автоматного программирования для среды разработки Eclipse» №2006613817 от 7.11.2006.

Структура диссертации. Диссертация изложена на 152 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит 114 наименований. Работа иллюстрирована 59 рисунками и содержит три таблицы.

Содержание работы

Во введении обосновывается актуальность темы диссертационной работы, описываются цель и задачи исследования. Формулируются положения, выносимые на защиту.

Первая глава содержит обзор существующих методов создания объектно-ориентированных программ на основе автоматного подхода.

В последнее время для повышения уровня абстракции средств разработки программ развивается такое направление программной инженерии как «Архитектура, управляемая моделями» (*Model-Driven Architecture, MDA*). При применении *MDA* модели программных систем представляются с помощью «Унифицированного языка моделирования» (*Unified Modeling Language, UML*). Если в течение ряда лет этот язык использовался только для представления моделей, то в последнее время все большую популярность приобретает идея *исполняемого UML*.

Моделирование динамических аспектов программ на языке *UML* затруднено в связи с отсутствием в стандарте на этот язык формального и однозначного описания правил интерпретации (операционной семантики) поведенческих диаграмм. Кроме того, ни в одном из методов проектирования объектно-ориентированных систем, «внятно» не сказано, как связывать статические диаграммы с динамическими.

Широким классом программных систем являются реактивные системы – системы, выполняющие определенные действия в ответ на внешние события. В работах Д. Харела показано, что для моделирования таких систем хорошо подходит расширение диаграмм переходов конечных автоматов, названное «диаграммы состояний» (*Statechart)*.

Для построения диаграмм состояний и генерации кода по ним создан ряд инструментальных средств, таких как, например, *I-Logix Statemate*, *XJTek AnyState*, *StateSoft ViewControl*, *SCOPE, IAR Systems visualSTATE*, *The State Machine Compiler*.

Недостаток этих инструментов состоит в том, что они позволяют строить и реализовывать только поведенческую часть модели программы, а не программу в целом. Это связано с отсутствием в данных инструментах диаграммы, отображающей в явном виде связи между конечными автоматами и объектами, поведение которых моделируется. Отметим, что спецификация *UML* позволяет задавать подобные связи с помощью диаграммы классов, однако ни в одной известной методологии не описано, как это делать.

Указанный недостаток устранен в *SWITCH*-технологии, которая вводит в процесс проектирования диаграмму связей автомата для описания его интерфейса (входных и выходных воздействий).

В настоящей работе предлагается метод на основе *SWITCH*-технологии и *UML*-нотации, позволяющий строить единую формальную модель программ. При этом диаграмма связей автомата представляется в виде *UML*-диаграммы классов.

Вторая глава содержит описание метода создания объектно-ориентированных программ на основе автоматного подхода, а также описание исполняемого графического языка автоматного программирования, его синтаксиса и операционной семантики.

Предлагается, сохранив автоматный подход, использовать *UML*-нотацию при построении диаграмм в рамках *SWITCH*-технологии, в которой программы строятся так же, как проводится автоматизация технологических процессов. При этом, используя нотацию *UML*-диаграмм классов, строятся схемы связей автоматов, которые определяют их интерфейс, а графы переходов создаются с помощью нотации *UML*-диаграммы состояний. При наличии нескольких автоматов их схема взаимодействия не строится, а все они изображаются на диаграмме классов. Эта диаграмма (как схема связей) и диаграммы состояний образуют предлагаемый графический язык для описания структуры и динамики программ.

Для проектирования программ на основе этого языка предлагается следующий метод:

1. На основе анализа предметной области в виде *UML*-диаграммы классов разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними.
2. В отличие от традиционных для объектно-ориентированного программирования подходов, из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны – они по собственной инициативе воздействуют на автоматы. Объекты управления пассивны – они выполняют действия, которые вызываются автоматами. Объекты управления также могут формировать значения входных переменных для автоматов. Автоматы активируются источниками событий и на основании значений входных переменных и текущих состояний воздействуют на объекты управления, переходя в новые состояния.
3. Используя нотацию диаграммы классов, строится схема связей автоматов, которая задает интерфейс каждого их них. На этой схеме слева изображаются источники событий, в центре – автоматы, а справа – объекты управления. Источники событий с помощью *UML*-ассоциаций связываются с автоматами, которым они поставляют события. Автоматы связываются с объектами, которыми они управляют, а также с другими автоматами, которые они вызывают или которые вложены в их состояния.
4. Схема связей, кроме задания интерфейсов автоматов, выполняет функцию, характерную для диаграммы классов – задает объектно-ориентированную структуру программы.

5. Каждый объект управления содержит два типа методов, реализующих входные переменные (`xj`) и выходные воздействия (`zk`). При этом отметим, что объект управления инкапсулирует вычислительные состояния системы, которые обычно явно не выделяются из-за большого их числа.
6. Для каждого автомата путем явного выделения управляющих состояний с помощью нотации диаграммы состояний строится граф переходов типа *Мура-Мили*, в котором дуги помечаются событием (`ei`), логической формулой из входных переменных и формируемыми на переходах выходными воздействиями.
7. В каждом состоянии могут указываться выходные воздействия, выполняемые при входе и имена вложенных автоматов, которые активны, пока активно состояние, в которое они вложены.
8. Кроме вложенности, автоматы могут взаимодействовать по вызываемости. При этом вызывающий автомат передает вызываемому событие, что и указывается на переходе или в состоянии в виде выходного воздействия. Во втором случае посылка события вызываемому автомату происходит при входе в состояние.
9. Каждый автомат имеет одно начальное и произвольное число конечных состояний.
10. Состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Основной особенностью сложных состояний является то, что дуга, исходящая из такого состояния, заменяет однотипные дуги, исходящие из каждого вложенного состояния.
11. Все сложные состояния неустойчивы, а все простые, за исключением начального – устойчивы. При наличии сложных состояний в автомате, появление события может привести к выполнению более одного перехода. Это происходит в связи с тем, что, как отмечено выше, сложное состояние является неустойчивым, и автомат выполняет переходы до тех пор, пока не достигнет первого из простых (устойчивых) состояний.
12. Каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования. Источники событий также реализуются вручную.
13. Использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем. Смысл таких символов задает схема связей. При наведении курсора на соответствующий символ на графе переходов во всплывающей подсказке отображается его текстовое описание.

Предлагаемый метод **позволяет выполнить проектирование программы в целом**.

На рис. 1 приведен пример схемы связей автомата, а на рис. 2 – его граф переходов.

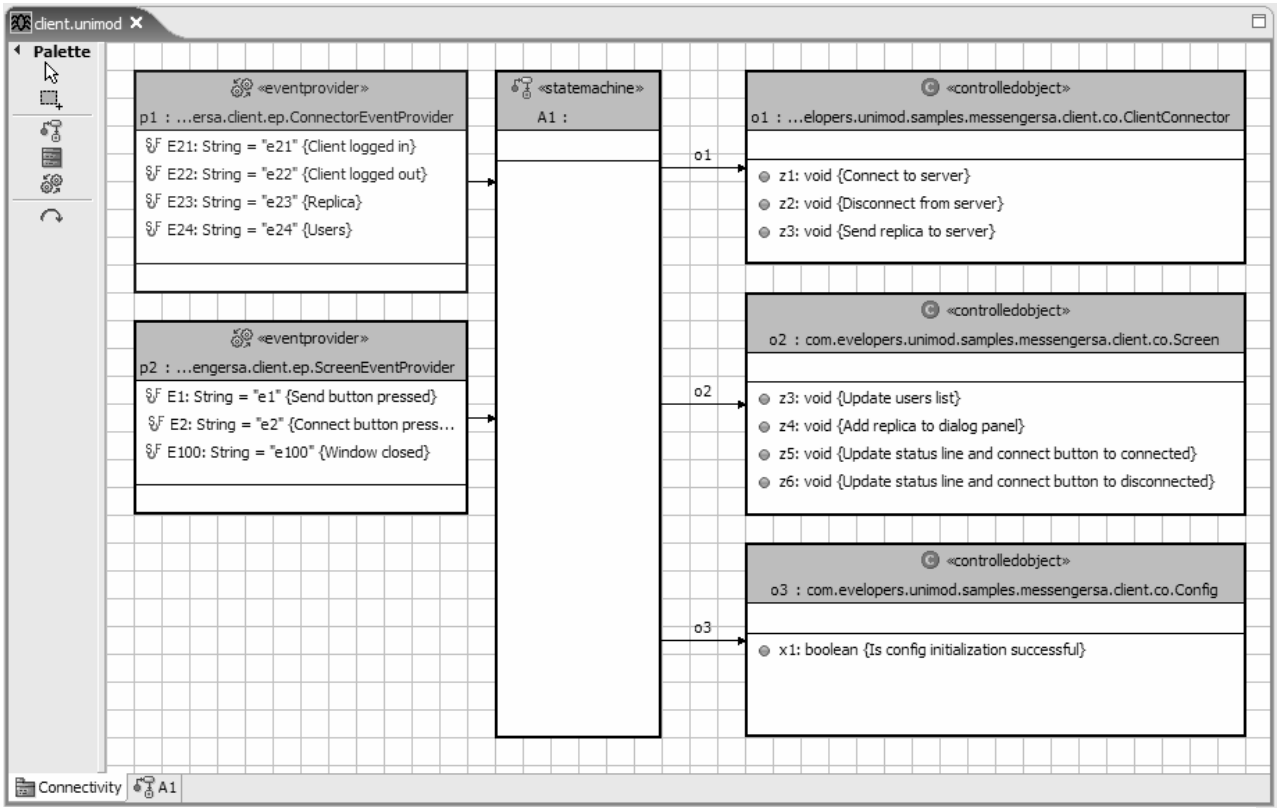

Рис. 1. Пример схемы связей автомата

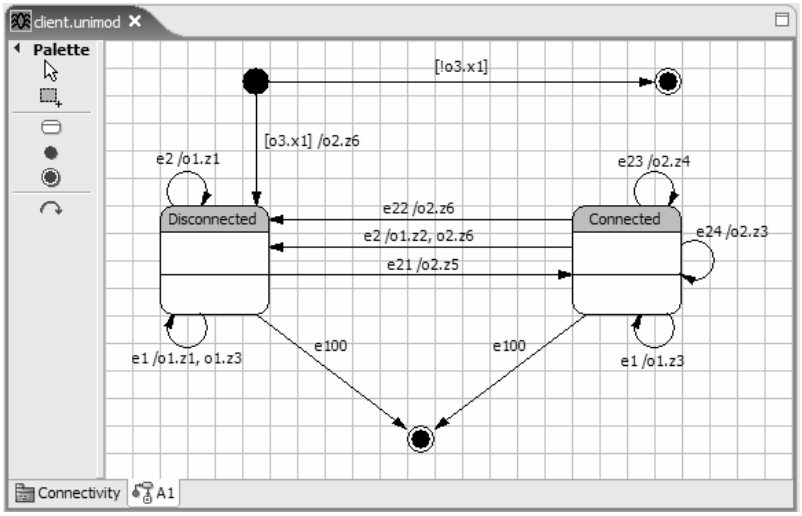

Рис. 2. Пример графа переходов автомата

Синтаксис созданного графического языка автоматного программирования основан на *UML*-нотации. При этом предлагается использовать следующие элементы *UML*-диаграммы состояний: *начальное*, *нормальное* и *конечное* состояния и переходы между ними. Состояния на диаграмме могут быть простыми и сложными. Если в нормальное состояние вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Наличие дуги, исходящей из сложного состояния, заменяет однотипные дуги из каждого вложенного состояния. Каждое сложное состояние содержит одно начальное состояние, а каждая диаграмма содержит одно *головное состояние* – сложное состояние, содержащее все остальные состояния. Для нормальных состояний могут быть определены действия при входе в состояние и вложенные

автоматы. У нормального состояния может быть произвольное число входящих и исходящих переходов. В конечное состояние может входить произвольное число переходов, а исходящих переходов быть не должно. Начальное состояние должно иметь ровно один исходящий переход.

Пометка на переходе состоит из события, инициирующего переход, условия и списка действий. На переходе может быть использовано либо событие, определенное в одном из источников, связанных с данным автоматом, либо символ «*», означающий любое событие. В качестве переменных в условиях на переходах, используются имена методов объектов управления, связанных с автоматом.

UML-состояния с параллельными регионами не поддерживаются. Это связано с тем, что проектирование объектов с одним потоком управления является достаточно простым, а для отражения параллелизма следует использовать несколько параллельно исполняемых автоматов.

Статическая модель системы состоит из одной диаграммы классов, на которой изображаются классы со следующими стереотипами: «*EventProvider*» – источник событий, «*StateMachine*» – автомат и «*ControlledObject*» – объект управления. Между такими классами возможно наличие направленных ассоциаций трех типов: от источника событий к автомату, от автомата к объекту управления и от автомата к автомату. Ассоциации должны быть помечены метками – идентификаторами. Для каждого автомата, изображенного на диаграмме классов, необходимо создать диаграмму состояний. Совокупность диаграмм состояний образуют динамическую модель системы.

Для модели системы, построенной описанным выше образом и состоящей из статической и динамической частей, задается операционная семантика:

1. При запуске модели инициализируются все источники событий и объекты управления. После этого источники событий начинают воздействовать на связанные с ними автоматы.
2. Каждый автомат начинает свою работу из начального состояния, а заканчивает – в одном из конечных.
3. При получении события автомат выбирает все исходящие из текущего состояния переходы, помеченные символом этого события.
4. Автомат перебирает выбранные переходы и вычисляет логические формулы, записанные на них, до тех пор, пока не найдет формулу со значением `true`.
5. Если переход с такой формулой найден, то автомат выполняет выходные воздействия, записанные на дуге, и переходит в новое состояние. В нем автомат выполняет выходные воздействия, а также запускает вложенные автоматы. Если новое состояние оказалось составным, осуществляется переход из начального состояния, находящегося внутри данного составного состояния.
6. Если среди выходных воздействий встречается вызываемый автомат, то он вызывается с соответствующим событием.
7. Если переход не найден, то автомат продолжает поиск перехода у родительского состояния – состояния, в которое вложено текущее состояние.
8. При переходе в конечное состояние автомат останавливает все источники событий. На этом работа системы завершается.

В третьей главе рассматриваются подходы к верификации автоматных программ на основе методов дедуктивного анализа и верификации на модели.

При использовании разработанного на основе *UML*-нотации графического языка программирования, создаваемые модели являются и формальной спецификацией и программой одновременно. Поэтому их можно подвергнуть формальной верификации.

Известны следующие подходы к верификации программ:

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

В дисертации предлагаются два метода верификации автоматных программ на базе последних двух подходов.

Верификация с помощью дедуктивного анализа. Любая диаграмма состояний должна удовлетворять следующему ограничению: множество исходящих переходов для любого состояния должно быть полно и непротиворечиво. Это означает, что при обработке любого события не должно быть альтернативных переходов и хотя бы один переход должен выполняться.

Для проверки полноты и непротиворечивости множества переходов, для каждой вершины необходимо решать задачу проверки тождественного равенства единице или нулю логической формулы. В данном случае логической формулой являются логические формулы, в которых используются следующие бинарные отношения: ‘>’, ‘<’, ‘>=’, ‘<=’, ‘=’, ‘≠’. При этом в качестве левого аргумента должна стоять целочисленная переменная, а справа – целочисленная константа.

Для решения указанной задачи можно использовать два подхода:

- вычисление значений формулы на всех возможных значениях переменных – построение таблицы истинности;
- минимизация логической формулы. Для тождественно истинных и тождественно ложных формул минимизация приводит к получению константного значения.

Второй подход обладает рядом преимуществ, так как в случае, если формула не является тождеством, то ее минимизированное представление можно использовать для облегчения процесса создания корректных диаграмм.

Алгоритм минимизации реализован для грамматики охранных условий. Построение синтаксического анализатора для данной грамматики осуществлялось с помощью библиотеки *ANTLR*, позволяющей автоматически по заданной *LL(k)*-грамматике строить транслятор входного потока в синтаксическое дерево разбора. Дерево, полученное в результате трансляции, можно использовать для вычисления значений формул во время выполнения программы, а также трансформировать в другое дерево.

Реализованный процесс минимизации логической формулы основан на трансформации синтаксического дерева разбора и состоит из следующих этапов:

1. Приведение формулы к дизъюнктивной нормальной форме (ДНФ).
2. Упрощение термов ДНФ.
3. Нахождение интервалов целочисленных переменных, входящих в терм, при которых он не обращается в ложь.

Приведение формулы к ДНФ реализуется с помощью последовательности трансформаций, сохраняющих тождественное равенство соответствующих формул.

Для приведения синтаксического дерева к дереву, соответствующему ДНФ, необходимо и достаточно удовлетворить следующие условия:

- узлы, соответствующие логическому отрицанию, должны быть родительскими только по отношению к узлам, соответствующим вхождениям переменных;

- ни один узел, соответствующий конъюнкции, не должен быть предком по отношению к узлам, соответствующим дизъюнкции.

Предлагаемая процедура проверки полноты систем переходов состоит из следующих этапов:

1. Построение дизъюнкции формул условий на переходах.
2. Оптимизация термов в дизъюнкции.
3. Приведение к ДНФ отрицания формулы, полученной на шаге (2).
4. Оптимизация термов в формуле, полученной на шаге (3).
5. Равенство нулю результирующей формулы означает полноту системы формул.

Проверка непротиворечивости выполняется аналогично.

Верификации на модели. Автоматная программа может рассматриваться как автоматизированный объект управления, состоящий из трех компонент:

- источники событий – инициируют работу системы;
- объекты управления – выполняют действия и формируют входные переменные;
- управляющая система – модуль, который принимает события и значения входных переменных и вызывает действия объектов управления. В автоматной программе данный модуль в общем случае может быть реализован, как система иерархически связанных конечных автоматов.

Верификация на модели автоматных программ состоит в проверке того, что управляющая система работает корректно, что определяется выполнением темпоральных утверждений вида «если произошло событие `e1`, то когда-нибудь будет вызвано действие `z1`» или «если всегда неверно `x1` (`x1` всегда `false`), то автомат никогда не попадет в состояние `s2`». Утверждения, которые требуется проверить, называют *требованиями*, а их совокупность – *спецификацией*. В том случае, если система автоматов удовлетворяет спецификации, считается, что верификация завершена успешно. Если хотя бы одно из требований не выполняется, то существует последовательность действий, которая приводит к нарушению этого требования. Такая последовательность называется *контрпример*.

У программ, написанных без использования автоматного подхода, отсутствует явное выделение управляющих состояний. Поэтому для таких программ возникает проблема значительного увеличения пространства состояний при верификации. Этого не происходит в автоматных программах, так как в них управляющие состояния выделяются уже на этапе проектирования. По этой причине для автоматных программ могут быть предложены эффективные методы верификации.

В работе предлагается метод верификации автоматных программ с помощью их *эмуляции*. Этот метод позволяет значительно снизить сложность преобразований исходной автоматной программы, которые необходимы для верификации.

Традиционно при верификации программы для нее строится модель *Крипке* – неявно заданный граф атомарных (вычислительных) состояний и переходов между ними. Модель *Крипке* является подробной схемой работы программы.

Требования к программе формулируются в виде формулы темпоральной логики, причем в данном случае используется *LTL* (*Linear Temporal Logic*).

LTL-формула преобразуется в автомат *Бюхи* – конечный автомат над бесконечными словами. Переходы автомата *Бюхи* помечены предикатами из исходной *LTL*-формулы. Поскольку задача верификатора – найти контрпример, если он существует, автомат *Бюхи* строится для *отрицания LTL*-формулы.

Далее модель *Крипке* преобразуется в автомат *Бюхи* для того, чтобы построить пересечение его с автоматом *Бюхи*, построенным по отрицанию *LTL*-формулы. Оно также является автоматом *Бюхи*. Для построенного пересечения автоматов запускается алгоритм

двойного поиска в глубину (модифицированный алгоритма *Тарьяна*), который находит допускающую последовательность предикатов. Если эта последовательность существует, то она является контрпримером.

В настоящей работе используется верификатор *Bogor*, который интегрируется с разработанным в рамках настоящей работы инструментальным средством *UniMod.*

В верификаторе *Bogor* явно не строятся модель *Крипке*, автомат *Бюхи* для модели *Крипке* и его пересечение с автоматом *Бюхи* для *LTL*-формулы. Верифицируемая программа записывается на входном языке верификатора. Это позволяет верификатору выделять в программе элементарные действия, поскольку в семантике этого языка определено, какие действия совершаются атомарно, как откатывать назад эти действия, возвращая систему в предыдущее состояние, как вычислять состояния объектов (переменных) и какие операторы порождают недетерминированность. При этом верификатору *Bogor* необходимо выполнять следующие действия:

1. Вычислять глобальное состояние программы.
2. Совершать элементарный шаг работы программы – переход программы из одного глобального состояния в другое без посещения иных глобальных состояний.
3. Откатывать назад элементарные шаги работы программы.
4. В каждом состоянии определять возможные дальнейшие элементарные шаги.
5. Определять значения набора предикатов программы, используемых в требованиях.

Предлагаемый в работе метод верификации автоматных программ состоит в реализации указанных действия следующим образом:

1. Глобальное состояние программы складывается из набора текущих состояний каждого автомата.
2. Элементарный шаг работы программы – обработка системой автоматов одного события, в результате которой может смениться набор состояний автоматов.
3. Поведение автоматной программы определяется набором состояний автоматов. Поэтому для отката назад достаточно вернуть автоматы в те состояния, в которых они находились до выполнения шага.
4. 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.
5. 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.

Особенностью предлагаемого метода эмуляции является то, что он не требует дополнительных преобразований автоматной программы и полученного контрпримера. Модель *Крипке* не строится явно, а 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.

Предложенный подход к верификации иллюстрируется на примере проверки автоматной программы, управляющей банкоматом. При этом за счет автоматического построения контрпримера удалось обнаружить существенную ошибку в исходной модели программы.

Предложенный метод позволяет верифицировать только автоматную часть программы. Входные и выходные воздействия, реализованные в объектах управления,

могут верифицироваться с помощью дедуктивных методов, основанных на проверке выполнения пред- и постусловий.

В четвертой главе описана реализация инструментального средства *UniMod* для поддержки разработанного графического языка и методов проектирования и верификации, изложенных в предыдущих главах. Она выполнена на языке *Java* в виде плагина для среды разработки *Eclipse*.

UniMod позволяет создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схемам связей и графам переходов автоматов.

При проектировании программ с использованием этого средства ее структура задается схемой связей в виде *UML*-диаграммы классов, а поведение – системой взаимодействующих автоматов, заданных в виде набора *UML*-диаграмм состояний Источники событий и объекты управления реализуются вручную на целевом языке программирования.

Рассматриваемое инструментальное средство поддерживает два основных типа обработки построенных диаграмм – интерпретацию и компиляцию.

Для обеспечения эффективности процесса разработки в инструментальном средстве *UniMod* также реализована следующая функциональность:

- подсветка синтаксических и семантических ошибок;
- завершение ввода и исправление ошибок ввода при редактировании пометок на переходах диаграмм состояний;
- запуск модели на исполнение;
- интерактивная отладка модели в терминах автомата.

Каждая из этих функций реализовывалась с помощью разрабатываемого средства – осуществлялась «раскрутка» (*bootstrap*), что свидетельствует о его высокой степени зрелости.

UniMod является проектом с открытым исходным кодом, который опубликован на сайте проекта http://unimod.sf.net. На момент написания диссертации было выпущено более 30 версий проекта и осуществлено более 40000 скачиваний. Ссылки на *UniMod* можно найти на десятках сайтов, посвященных автоматизации программирования.

Пятая глава содержит описание результатов внедрения предложенных методов и инструментального средства.

Метод построения программ с явным выделением состояний, а также инструментальное средство *UniMod*, описанные в данной работе, использовались:

- для разработки некоторых функций инструментального средства *UniMod*;
- в компании *eVelopers* для разработки интернет-приложений и мобильных приложений;
- в компании *JetBrains* при разработке системы мета-программирования *Meta Programming System (MPS)*;
- в учебном процессе СПбГУ ИТМО при разработке курсовых проектов.

Опишем эти области внедрения более подробно.

Реализация одной из функций инструментального средства *UniMod*

В рамках создания очередной версии пакета *UniMod* одной из важнейших является задача реализации системы автоматического завершения ввода при редактировании условий на переходах на *UML*-диаграмме состояний. Для этого в работе предлагается новая технология создания системы автоматического завершения ввода, которая позволяет, в отличие от традиционного подхода, исключить таблицу разбора нисходящего нерекурсивного синтаксического анализатора и использовать гибкий алгоритм восстановления после ошибок на уровне фразы.

Технология основывается на том, чтобы для заданной *LL(1)*-грамматике построить конечный автомат Мили, который будет синтаксическим анализатором. Автомат должен реагировать на события, поставляемые ему лексическим анализатором. Каждому событию

соответствует терминал. Предлагается для каждого правила грамматики построить диаграмму переходов, а потом свернуть все диаграммы в одну, при необходимости удаляя срединную рекурсию. Такой подход позволяет избавиться от упоминания нетерминалов на диаграммах переходов и, следовательно, разорвать связь с исходной грамматикой. Это позволяет описывать язык только с помощью диаграммы переходов и автоматически получать реализацию распознавателя для данного языка.

Внедрение в промышленное программирование. В качестве примера применения инструментального средства *UniMod* в компании *eVelopers* в диссертации описан процесс разработки автоответчика для мобильного телефона *Nokia 6600*. На данном устройстве установлена операционная система *Symbian*, а разработка приложений выполняется на языке *C++*.

Для этого языка созданы шаблоны для генерации кода, что позволило применять пакет *UniMod* при использовании языка *C++,*как целевого языка программирования. Интерпретационный подход на языке *C++* не реализован, так как для мобильных устройств требуется высокое быстродействие, обеспечиваемое при компиляции.

Разработан метод формального и изоморфного перехода от спецификации к текстам программ на языке C++. Формальность перехода позволяет для одного графа переходов использовать его в качестве полного теста для сертификации построенной программы, а изоморфность с графом переходов – проверять программу сверкой ее текста со спецификацией.

Достаточно простая адаптация инструментального средства, ориентированного на язык *Java,* для использования языка *C++* свидетельствует о переносимости предлагаемых в диссертационной работе подходов.

Текстовый язык автоматного программирования. Инструментальное средство *UniMod*, наряду с перечисленными выше достоинствами, обладает и недостатками:

- ввод диаграмм состояний с помощью графического редактора весьма трудоемок;
- многие программисты предпочитают работать с текстовым представлением программы, несмотря на то, что диаграммы позволяют представлять информацию более компактно и обозримо;
- невозможно в одном *Java*-классе совместить автомат и объект управления, что не позволяет прозрачно использовать автоматное программирование совместно с объектно-ориентированным, так как в настоящее время код, генерируемый из автоматной модели, не является в полной мере объектно-ориентированным.

Для устранения перечисленных недостатков был предложен новый подход к разработке автоматных программ и применению автоматов в объектно-ориентированных системах. В рамках этого подхода предлагается использовать систему метапрограммирования *JetBrains Meta Programming System* (*MPS*), которая позволяет создавать проблемно-ориентированные языки (*Domain Specific Language – DSL*). С помощью *MPS* созданы два варианта текстового языка для автоматного программирования. Первый язык выполнен в виде расширения языка *Java*, а второй – самостоятельным языком. Эти языки позволяют описывать состояния и логику переходов автоматов, а также события, обрабатываемые автоматами. При этом также как и в инструментальном средстве *UniMod*, функции входных и выходных переменных реализуются на другом языке программирования, таком как *Java* или *C++*.

Внедрение в учебный процесс. С помощью инструментального средства *UniMod* на кафедре «Компьютерные технологии» СПбГУ ИТМО реализовано около 30 студенческих проектов, в которых эмулировалась работа банкоматов, стиральных машин, турникетов в метро, автомобильной сигнализации и т.д. Работы опубликованы на сайте http://is.ifmo.ru/unimod-projects.

Заключение

В диссертации получены следующие научные результаты:

1. Создан метод проектирования объектно-ориентированных программ на основе автоматного подхода.
2. Разработан графический язык программирования на основе *UML*-нотации для поддержки указанного метода.
3. Разработаны методы верификации автоматных моделей программ.
4. Разработано инструментальное средство для поддержки автоматного программирования.
5. Полученные результаты внедрены в практику разработки программного обеспечения и в учебный процесс в СПбГУ ИТМО.

В настоящее время магистрантами кафедры «Компьютерные технологии» СПбГУ ИТМО проводятся работы по созданию второго поколения разработанного инструментального средства – *UniMod2*, которое основывается на новых технологиях и имеет большую функциональность.

Список публикаций

1. *Гуров В. С., Нарвский А. С., Шалыто А. А.* Автоматизация проектирования событийных объектно-ориентированных программ с явным выделением состояний // Труды X Всероссийской научно-методической конференции «Телематика-2003». СПбГИТМО (ТУ). 2003.
2. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UML. SWITCH-технология. Eclipse //Информационно-управляющие системы. 2004. № 6, с. 12–17.
3. *Гуров В. С., Нарвский А. С., Шалыто А. А.* ICQ и Автоматы // Технология клиент-сервер. 2004. № 3, с. 3–11.
4. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Разработка средств автоматизации построения объектно-ориентированных программ с явным выделением состояний // Научно-технический вестник СПбГУ ИТМО. Вып.16. Актуальные проблемы современных оптико-информационных систем и технологий. 2004, с. 88–100.
5. *Гуров В. С., Мазин М. А., Шалыто А. А.* UNIMOD – программный пакет для разработки объектно-ориентированных приложений на основе автоматного подхода //Труды XI Всероссийской научно-методической конференции «Телематика-2004». СПбГУ ИТМО. Т.1, с. 189–191.
6. *Горшкова Е. А., Новиков Б. А., Белов Д. Д., Гуров В. С., Спиридонов С. В.* Моделирование контроллера Web-приложений с использованием UML //Программирование. 2005. № 1, с. 44–51.
7. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UniMod: Метод и средство разработки реактивных объектно- ориентированных программ с явным выделением состояний /Труды Второй Всероссийской научной конференции «Методы и средства обработки информации». МГУ. 2005, с. 361–366
8. *Гуров В. С., Мазин М. А.* Создание системы автоматического завершения ввода с использованием пакета UniMod /Вестник II межвузовской конференции молодых ученых. Т.1. СПбГУ ИТМО. 2005, с. 73–87.
9. *Гуров В. С., Мазин М. А., Шалыто А. А.* Операционная семантика UML-диаграмм состояний в программном пакете UniMod //Труды XII Всероссийской научно-методической конференции «Телематика-2005». СПбГУ ИТМО. Т.1, с. 74–76.
10. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Исполняемый UML. Проект UniMod /Software Engineering Conference (Russia) – 2005. М.: РУССОФТ. Abstracts, с. 12–13.
11. *Gurov V. S., Mazin M. A., Narvsky A. S., Shalyto A. A.* UniMod: Method and Development of Reactive Object-Oriented Programs with Explicit States Emphasis

/Proceedings 2005 of St. Petersburg IEEE Chapters. International Conference «110 Anniversary of Radio Invention». SPb ETU «LETI». 2005, pp.106–110.

12. *Гуров В. С., Мазин М. А., Нарвский А. С., Парфенов В. Г., Шалыто А. А.* UniMod – инструментальное средство для поддержки автоматного програмимрования //Материалы X Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах». СПбГ ПУ. 2006, с. 481–488.
13. *Гуров В. С., Мазин М. А., Парфенов В. Г.* Графический отладчик UML-диаграмм состояний в программном пакете UniMod //Труды XIII Всероссийской научно-технической конференции «Телематика-2006». СПбГУ ИТМО. 2006, с. 106–110.
14. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Инструментальное средство для поддержки автоматного программирования //Рабочий семинар «Наукоемкое программное обеспечение» на шестой международной конференции «Перспективы систем информатики». Новосибирск: Институт проблем информатики имени А. П. Ершова. СО РАН. 2006, с. 52–54.
15. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UniMod – инструментальное средство для автоматного программирования //Материалы третьей международной конференции по проблемам управления. М.: Институт проблем управления. 2006, с. 224–227.
16. *Гуров В. С., Мазин М. А., Шалыто А. А.* UniMod – Инструментальное средство для автоматного программирования // Фундаментальные и прикладные исследования информационных систем и технологий. Научно-технический вестник СПбГУ ИТМО. Вып. 30. 2006, с. 32–44.
17. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Инструментальное средство для поддержки автоматного программирования //Программирование. 2007. № 6, с. 65–80.
18. *Гуров В. С., Мазин М. А., Зубок Д. А., Парфенов В. Г., Шалыто А. А.* Два подхода к созданию программ с использованием инструментального средства UniMod //Труды XIV Всероссийской научно-методической конференции «Телематика-2007». СПбГУ ИТМО, 2007.
19. *Гуров В. С., Шалыто А. А., Яминов Б. Р.* Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора //Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы» (МВУС-2007). Т.1, с. 198–203.
20. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования //Труды научной конференции «Компьютерные науки и технологии». Саратов: СГУ. 2007, с. 66–69.
21. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования //Фундаментальные и прикладные исследования информационных систем и технологий. Научно-технический вестник СПбГУ ИТМО. Вып. 42. 2007, с. 29–32
22. *Гуров В. С., Мазин М. А., Шалыто А. А.* Автоматическое завершение ввода условий в диаграммах состояний //Информационно-управляющие системы. 2008. № 1, с. 24–33.
23. *Гуров В. С., Яминов Б. Р.* Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора /Материалы научно-технической конференции «Научное программное обеспечение в образовании и научных исследованиях». СПбГ ПУ. 2008, с. 36–40.

Тиражирование и брошюровка выполнены в
центре «Университетские телекоммуникации»
Санкт-Петербург, Саблинская ул. 14; тел: (812)233-4669
Тираж 100 экз.