На правах рукописи

Шамгунов Никита Назимович

РАЗРАБОТКА МЕТОДОВ ПРОЕКТИРОВАНИЯ И РЕАЛИЗАЦИИ ПОВЕДЕНИЯ ПРОГРАММНЫХ СИСТЕМ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА

Специальность 05.13.13 — Телекоммуникационные системы и компьютерные сети

АВТОРЕФЕРАТ

диссертации на соискание ученой степени кандидата технических наук

Санкт-Петербург 2004

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
Марлей Владимир Евгеньевич,

кандидат физ.-мат. наук
Новиков Федор Александрович

Ведущая организация: Ленинградский отраслевой научно-исследовательский институт связи (Санкт-Петербург)

Защита диссертации состоится «___» декабря 2004 года в ___ часов на заседании диссертационного совета Д.212.227.05 в Санкт-Петербургском государственном университете информационных технологий, механики и оптики, 197101, Санкт-Петербург, ул. Саблинская 14, СПбГУ ИТМО.

С диссертацией можно ознакомиться в библиотеке СПбГУ ИТМО.

Автореферат разослан «___» ноября 2004 г.

Ученый секретарь
Совета Д.212.227.05,
кандидат технических наук,
доцент — Поляков Владимир Иванович

Общая характеристика работы

Актуальность проблемы. При разработке телекоммуникационных систем и компьютерных сетей весьма актуальной является задача формализации описаний их поведения. При этом наиболее известным является графический язык описаний и спецификаций *SDL* (Specification and Description Language), разработанный Международным союзом электросвязи (ITU-T). Этот язык входит в Рекомендации ITU-T серии Z.100.

Диаграммы, являющиеся основой этого языка, в отличие от схем алгоритмов, содержат состояния в явном виде. Поэтому язык *SDL* является автоматным. Однако *SDL*-диаграммы обладают рядом недостатков, к которым можно отнести, например, громоздкость. С другой стороны, при разработке систем рассматриваемого класса все шире используется объектно-ориентированное программы, для проектирования которых применяется унифицированный язык моделирования (*UML* —Unified Modeling Language). В этом языке для описания поведения также используется автоматная модель — диаграммы состояний *Statecharts*, предложенные Д. Харелом. Эти диаграммы из-за использования словесных обозначений также являются весьма громоздкими.

Поэтому в последние годы были выполнены исследования, направленные на объединение языков *SDL* и *UML* (Рекомендации Z.109 ITU-T, 2000). Однако из изложенного выше следует, что применительно к описанию поведения, даже совместное применение указанных выше языков не делает диаграммы менее громоздкими.

Для устранения этого недостатка с 1991 года в России разрабатывается *SWITCH*-технология, предназначенная для алгоритмизации и программирования систем со сложным поведением. Эта технология была названа также автоматным программированием. Графы переходов, используемые для описания поведения в рамках предлагаемого подхода, достаточно компактны, так как они применяются совместно со схемами связей, подробно описывающими интерфейс автоматов.

Поэтому в настоящее время весьма актуальны исследования, направленные на обеспечение компактного и формального описания поведения программных систем.

Не менее актуальной является также решение задачи о формальном переходе от спецификации задачи к ее реализации.

Целью диссертационной работы является разработка методов проектирования и реализации поведения программных систем на основе автоматного подхода.

В работе рассматриваются два типа задач: классические вычислительные алгоритмы (в основном рекурсивные) и задачи управления. В первом случае реализация осуществляется на основе процедурного подхода, а во втором — на основе объектно-ориентированного.

Основные задачи исследования состоят в следующем.

1. Разработка метода преобразования классических вычислительных алгоритмов с явной рекурсией в автоматные, что, в частности, позволяет формализовать процесс их визуализации.
2. Разработка образца проектирования (паттерна) объектов, с изменяющимся в зависимости от состояния поведением, в котором устранены недостатки известного паттерна *State*.
3. Разработка языка автоматного программирования, основанного на непосредственной поддержке предложенного паттерна.

Методы исследования. В работе использованы методы дискретной математики, построения и анализа алгоритмов, теории автоматов, построения компиляторов, паттерны проектирования объектно-ориентированных программ.

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

1. Для ряда вычислительных алгоритмов (например, обход деревьев) предложена их автоматная реализация, которая более наглядна по сравнению с классическими решениями.
2. Предложен метод, позволяющий формально выполнять преобразования процедурных программ с явной рекурсией в автоматные программы, что делает естественной их визуализацию.
3. Для реализации объектов, поведение которых варьируется от состояния, разработан паттерн проектирования *State Machine*, обеспечивающий по сравнению с применяемым для этой цели паттерном *State*, возможность повторного использования классов состояний, централизацию логики переходов и независимость классов состояний друг от друга.
4. На базе предложенного паттерна, за счет введения дополнительных синтаксических конструкций в язык *Java*, разработан автоматный язык *State Machine*, позволяющий писать программы непосредственно в терминах автоматного программирования.

Результаты диссертации были получены в ходе выполнения научно-исследовательских работ «Разработка технологии программного обеспечения систем управления на основе автоматного подхода», выполненной по заказу Министерства образования РФ в 2001 – 2004 гг., и «Разработка технологии автоматного программирования», выполненной по гаранту РФФИ по проекту № 02-07-90114 в 2002 – 2003 гг. (http://is.ifmo.ru, раздел «Наука»).

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

Практическое значение работы состоит в том, что все полученные результаты могут быть использованы, а некоторые уже используются на практике. Предложенные методы позволяют повысить наглядность программ, упростить их визуализацию, а также упростить внесение изменений в них. При этом за счет преобразования условной логики в автоматную упрощается структура программ. Предложенный паттерн обеспечивает повторное использование классов состояний, а разработанный язык упрощает применение этого паттерна за счет непосредственного отображения его состояний в код программы.

Реализация результатов работы. Результаты, полученные в диссертации, используются на практике.

1. В компании *eVelopers* (Санкт-Петербург) при создании системы автозавершения ввода в пакете автоматно-ориентированного программирования *Unimod*.
2. В компании *Транзас* (Санкт-Петербург) при создании телекоммуникационной системы управления движением судов *Navi Harbour*.
3. В учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсам «Теория автоматов в программировании» и «Программирование на языке *Java*».

Апробация диссертации. Основные положения диссертационной работы докладывались на XXXIII конференции профессорско-преподавательского состава СПбГУ ИТМО (Санкт-Петербург, 2004), научно-методических конференциях «Телематика-2002»,

«Телематика-2004» (Санкт-Петербург) и на конференции *Microsoft Research Academic Days 2004* (Санкт-Петербург).

Публикации. По теме диссертации опубликовано 9 печатных работ, в том числе в журналах «Информационно-управляющие системы», «Программист», «Компьютерные инструменты в образовании», «Телекоммуникации и информатизация образования» и «Мир ПК».

Структура диссертации. Диссертация состоит из введения, пяти глав и заключения. Список литературы содержит 83 наименования. Работа иллюстрирована 42 рисунками и содержит две таблицы.

Содержание работы

Во введении описывается объект и предмет исследования, ставится цель и задачи исследования, обосновывается актуальность темы диссертационного исследования. Дана оценка новизны полученных результатов, сформулированы положения, выносимые на защиту.

Первая глава содержит обзор существующих методов, паттернов и языков, применяющихся для описания поведения программных систем применительно к цели и задачам диссертационной работы.

В программировании конечные автоматы применяются весьма ограниченно. Основная область их использования — компиляторы. Они применяются также при решении некоторых вычислительных задач, например, при поиске подстрок. Это, возможно, связано с тем, что в программировании обычно не разделяются управляющие и вычислительные состояния.

Известны ***методы обхода деревьев***, которые недостаточно универсальны. В настоящей работе разработан универсальный метод обхода деревьев на основе автоматного подхода, базирующегося на указанном разделении состояний.

Известен ***метод преобразования итеративных программ в автоматные***, предложенный Н.И. Туккелем и А.А. Шалыто. Этот метод не позволяет выполнять аналогичные преобразования для программ с явной рекурсией. В настоящей работе такой метод разработан.

В работе рассмотрены также ***паттерны объектно-ориентированного проектирования***. Паттерн проектирования — это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерны представляют собой квинтэссенцию опыта удачных разработок архитектуры объектно-ориентированных программ.

Для описания объектов, варьирующих поведения в зависимости от состояния, в книге Э. Гаммы и др. предложен паттерн *State*, недостатками которого являются сложность повторного использования, децентрализация логики переходов и зависимость классов состояний друг от друга. В настоящей работе предлагается паттерн, устраняющий эти недостатки.

Переходя к анализу языков, отметим, что они могут быть разделены на два типа: текстовые и графические.

К языкам первого типа относится, например, ***язык ASML*** (*Abstract State Machine Language*), предложенный Ю. Гуревичем (компания *Microsoft*). Этот язык применим в ситуациях, когда необходимо точное, непротиворечивое описания компьютерной системы. В языке вводится понятие *абстрактное состояние*. Он обеспечивает компактное

высокоуровневое описание состояния системы. Однако этот язык еще не нашел широкого применения на практике.

Ко второму типу языков относятся, например, ***языки SDL и UML***. Их особенности применительно к описанию поведения программных системы описаны выше. Отметим, что язык *UML* является нотацией, но не методом разработки программ. Существуют различные реализации языка *UML*: *Rational Rose* (*IBM*), *Together Control Center* (*Borland*), *REAL* (*СПбГУ*) и т.д. Как отмечено выше, недостатком этих языков, применительно к описанию поведения, является громоздкость.

В последнее время имеет место тенденция создания ***языков, ориентированных на предметную область***. При этом зарекомендовавшим себя способом расширения языков программирования является встраивание в них поддержки паттернов проектирования. Например, в язык *C#* встроены ключевые слова *delegate* и *event*, позволяющие эффективно реализовать паттерн *Observer*, широко используемый, например, для реализации графического интерфейса пользователя. В настоящей работе предлагается язык, на основе языка *Java*, который ориентирован на область автоматного программирования.

Подходы, использованные в настоящей работе, основаны на ***SWITCH-технологии***. В этой технологии поведение программ предлагается проектировать на основе конечных автоматов. Качество программ, построенных с применением этой технологии, достигается за счет выразительных возможностей графов переходов и изоморфного перехода от этих графов к программам. Эта технология успешно зарекомендовала себя для создания систем логического управления. С использованием этой технологии студентами и аспирантами *СПбГУ ИТМО* было создано более пятидесяти учебных проектов в самых разных областях разработки программного обеспечения (http://is.ifmo.ru, раздел проекты). Это позволило рассмотреть технологию с самых разных сторон и проанализировать ее сильные и слабые стороны. Настоящая работа продолжает исследования в этом направлении.

Из изложенного следует, что сформулированные во введении задачи актуальны, и в настоящее время эффективно не решены.

Вторая глава. Эта глава посвящена применению конечных автоматов при реализации ряда вычислительных алгоритмов.

Известно рекурсивное решение ***задачи о Ханойских башнях***. Менее известно итеративное решение. В работе предложено автоматное решение, основанное, как и в машине Тьюринга, на выделении управляющих и вычислительных состояний. В рассматриваемом случае автомат строится непосредственно по условиям задачи.

При этом автомат с двумя состояниями управляет «объектом управления» с 2^N состояниями, возникающими в процессе перекладывания дисков.

Известна ***задача о ходе коня***, которая состоит в том, чтобы найти обход доски размером `NxM` конем, перемещающимся по правилам шахматной игры.

Известны рекурсивное и итеративное решения этой задачи, реализующие *алгоритм с возвратом* (*backtracking*). В работе предлагается автоматное решение задачи. Также как и в предыдущем случае, автомат построен непосредственно по условиям задачи.

Для сокращения количества возвратов рассматриваются различные методы оптимизации.

Обход деревьев. Известна задача об обходе деревьев, в частности двоичных. Данная задача может быть решена несколькими способами: рекурсивно, итеративно с применением стека и итеративно без использования стека. Однако в известных решениях этой задачи не используется понятие состояние, что усложняет их визуализацию. Поэтому в настоящей работе для этой цели предложен автоматный подход. При этом задача решена, как с использованием стека, так и без него. Автоматы, как и в предыдущих случаях, строятся

непосредственно по условиям задачи. На рис. 1 в качестве примера приведен граф переходов автомата для нерекурсивного обхода двоичного дерева без использования стека.

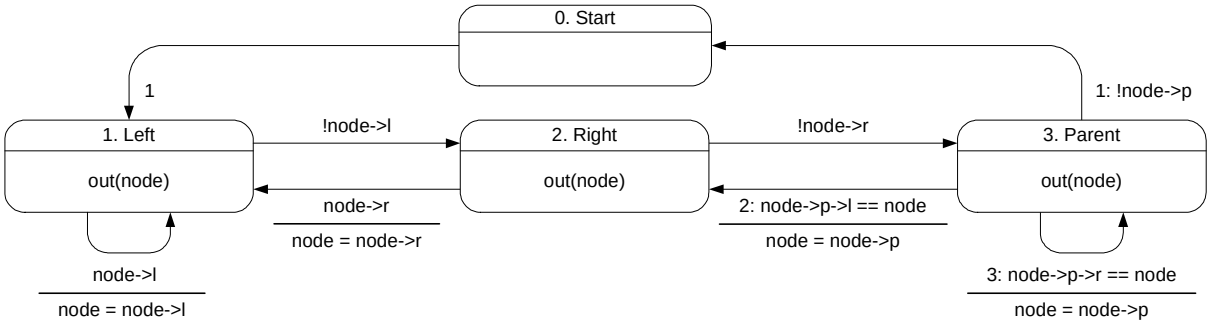

Рис. 1. Граф переходов автомата для реализации обхода двоичного дерева без использования стека

На этом рисунке используются следующие обозначения: `node` — текущий узел дерева, `node->l` — левое поддерево, `node->r` — правое поддерево, `node->p` — указатель на родителя. Единица на переходе `0-1` соответствует безусловному переходу, а `k:` — приоритету дуги с номером `k`.

Как отмечено выше, предложен также автоматный подход к обходу двоичных деревьев с использованием стека. Этот подход обобщен на случай обхода *k*-ичных деревьев.

Использование автоматного подхода позволяет получить наглядные и универсальные алгоритмы решения рассматриваемых задач, которые также весьма экономны по памяти по сравнению с классическими алгоритмами.

Преобразование программ с явной рекурсией в автоматные. Известен метод преобразования произвольных итеративных программ в автоматные программы, что позволяет реализовать произвольный итеративный алгоритм структурированной программой, содержащей один оператор `do-while`, телом которого является один оператор `switch`.

Известны также примеры преобразований рекурсивных программ в итеративные. Однако, в связи с отсутствием соответствующего метода, преобразования выполнялись неформально. В данной главе такой метод предлагается.

Изложим метод преобразования программ с явной рекурсией в автоматные.

1. Каждый рекурсивный вызов в программе выделяется как отдельный оператор. По преобразованной рекурсивной программе строится ее схема, в которой применяются символьные обозначения условий переходов (`x`), действий (`z`) и рекурсивных вызовов (`R`). Схема строится таким образом, чтобы каждому рекурсивному вызову соответствовала отдельная операторная вершина. Отметим, что здесь и далее под термином «схема программы» понимается схема ее рекурсивной функции.
2. Для определения состояний эквивалентного построенной схеме автомата Мили в нее вводятся пометки, по аналогии с тем, как это выполнялось в указанном выше методе преобразования итеративных программ в автоматные. При этом начальная и конечная вершины помечаются номером `0`. Точка, следующая за начальной вершиной, помечается номером `1`, а точка, предшествующая конечной вершине — номером `2`. Остальным состояниям автомата соответствуют точки, следующие за операторными вершинами.
3. Используя пометки в качестве состояний автомата Мили, строится его граф переходов.

4. Выделяются действия, совершаемые над параметрами рекурсивной функции при ее вызове.
5. Составляется перечень параметров и других локальных переменных рекурсивной функции, определяющий структуру ячейки стека. Если рекурсивная функция содержит более одного оператора рекурсивного вызова, то в стеке также запоминается значение переменной состояния автомата.
6. Выполняется преобразование графа переходов для моделирования работы рекурсивной функции, состоящее из трех этапов.
   6.1. Дуги, содержащие рекурсивные вызовы (`R`), направляются в вершину с номером `1`. В качестве действий на этих дугах указываются: операция запоминания значений локальных переменных `push(s)`, где `s` — номер вершины графа переходов, в которую была направлена рассматриваемая дуга до преобразования; соответствующее действие, выполняемое над параметрами рекурсивной функции.
   6.2. Безусловный переход на дуге, направленной из вершины с номером `2` в вершину с номером `0`, заменяется условием «стек пуст».
   6.3. К вершине с номером `2` добавляются дуги, направленные в вершины, в которые до преобразования графа входили дуги с рекурсией. На каждой из введеных дуг выполняется операция `pop`, извлекающая из стека верхний элемент. Условия переходов на этих дугах имеют вид `stack[top].y == s`, где `stack[top]` — верхний элемент стека, `y` — значение переменной состояния автомата, запомненное в верхнем элементе стека, а `s` — номер вершины, в которую направлена рассматриваемая дуга. Таким образом, в рассматриваемом автомате имеет место *зависимость от глубокой предыстории* — в отличие от классических автоматов, переходы из состояния с номером `2` зависят также и от ранее запомненного в стеке номера следующего состояния.
7. Граф переходов может быть упрощен за счет исключения неустойчивых вершин (кроме вершины с номером `0`).
8. Строится автоматная программа, содержащая функции для работы со стеком и цикл `do-while`, телом которого является оператор `switch`, формально и изоморфно построенный по графу переходов.

Метод иллюстрируется примерами преобразований классических рекурсивных программ в автоматные, которые приведены в порядке их усложнения (факториал, числа Фибоначчи, ханойские башни, дискретная задача о ранце).

Третья глава. Важной задачей проектирования является реализация объектов, поведение которых варьируется в зависимости от внутреннего состояния. Известен паттерн проектирования *State*, предназначенный для реализации таких объектов. Предлагается новый паттерн объектно-ориентированного проектирования, названный *State Machine*. Этот паттерн расширяет возможности паттерна *State*. В данном паттерне предложено использовать события для уведомления об изменении состояния. Это позволяет проектировать объекты такого рода из независимых друг от друга классов. Приведенный паттерн по сравнению с паттерном *State* лучше приспособлен для повторного использования входящих в него классов.

Назначение. Паттерн *State Machine* предназначен для создания объектов, поведение которых варьируется в зависимости от состояния. При этом для клиента создается впечатление, что изменился класс объекта. Таким образом, назначение предлагаемого

паттерна фактически совпадает с таковым для паттерна *State*, однако, как будет показано ниже, область применимости последнего уже.

Мотивация. Предположим, что требуется спроектировать класс `Connection`, представляющий сетевое соединение. Простейшее сетевое соединение имеет два управляющих состояния: *соединено* и *разъединено*. Переход между этими состояниями происходит или при возникновении ошибки или посредством вызовов методов *установить соединение* (`connect`) и *разорвать соединение* (`disconnect`). В состоянии *соединено* может производиться получение (метод `receive`) и отправка (метод `send`) данных по соединению. В случае возникновения ошибки при передаче данных генерируется исключительная ситуация (`IOException`) и сетевое соединение разъединяется. В состоянии *разъединено* прием и отправка данных невозможны. При попытке осуществить передачу данных в этом состоянии объект также генерирует исключительную ситуацию.

Таким образом, интерфейс, который необходимо реализовать в классе `Connection`, должен выглядеть следующим образом (пример приводится на языке *Java*):

```
package connection;

import java.io.IOException;

public interface IConnection {
  public void connect() throws IOException;
  public void disconnect() throws IOException;
  public int receive() throws IOException;
  public void send(int value) throws IOException;
}
```

Основная идея паттерна *State Machine* заключается в разделении классов, реализующих логику переходов (*контекст*), конкретных состояний и модели данных. Для осуществления взаимодействия конкретных состояний с контекстом используются *события*, представляющие собой объекты, передаваемые состояниями контексту. Отличие от паттерна *State* состоит в методе определения следующего состояния при осуществлении перехода. Если в паттерне *State* следующее состояние указывается текущим состоянием, то в предлагаемом паттерне это выполняется путем уведомления класса контекста о наступлении события. После этого, в зависимости от события и текущего состояния, контекст устанавливает следующее состояние в соответствии с графом переходов.

Преимуществом такого подхода является то, что классам, реализующим состояния, не требуется «знать» друг о друге, так как выбор состояния, в которое производится переход, осуществляется контекстом в зависимости от текущего состояния и события.

Отметим, что графы переходов, применяемые для описания логики переходов при проектировании с использованием паттерна *State Machine*, отличаются от графов переходов, рассмотренных в других работах. Применяемые графы переходов состоят только из состояний и переходов, помеченных событиями. Переход из текущего состояния S в следующее состояние S^* осуществляется по событию E, если на графе переходов существует дуга из S в S^*, помеченная событием E. При этом из одного состояния не могут выходить две дуги, помеченные одним и тем же событием. Отметим, что на графе переходов не отображаются ни методы, входящие в интерфейс реализуемого объекта, ни условия порождения событий.

Граф переходов для описания поведения класса `Connection` приведен на рис. 2. В нем классы, реализующие состояния *соединено* и *разъединено*, называются соответственно `ConnectedState` и `DisconnectedState`, тогда как событие `CONNECT` обозначает установление связи, `DISCONNECT` — обрыв связи, а `ERROR` — ошибку передачи данных.

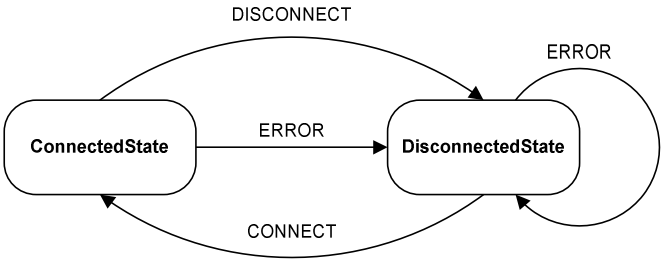

Рис. 2. Граф переходов для класса **`Connection`**

Рассмотрим в качестве примера обработку разрыва соединения при ошибке передачи данных. При реализации с использованием паттерна *State* состояние `ConnectedState` укажет контексту, что следует перейти в состояние `DisconnectedState`. В случае же паттерна *State Machine* контекст будет уведомлен о наступлении события `ERROR`, а тот осуществит переход в состояние `DisconnectedState`. Таким образом, классы `ConnectedState` и `DisconnectedState` не знают о существовании друг друга.

Преимуществом такого подхода является то, что классам, реализующим состояния, не требуется «знать» друг о друге, так как выбор состояния, в которое производится переход, осуществляется контекстом в зависимости от текущего состояния и события.

Применимость. Паттерн *State Machine* может быть использован в следующих случаях.

1. *Поведение объекта существенно зависит от управляющего состояния. При этом реализация поведения объекта в каждом состоянии будет сконцентрирована в одном классе*. Этот вариант использования иллюстрируется в данной работе на примере класса, реализующего сетевое соединение.
2. *Рефакторинг (метод преобразования исходного кода, предложенный М. Фаулером).* Примером использования может служить рефакторинг кода, проверяющего права доступа к тем или иным функциям программного обеспечения в зависимости от текущего пользователя или приобретенной лицензии.
3. *Повторное использование классов, входящих в паттерн, в том числе посредством создания иерархии классов состояний.*
4. *Эмуляции абстрактных и структурных автоматов.*

Таким образом, область применимости паттерна *State Machine* шире, чем у паттерна *State*.

Структура. На рис. 3 изображена структура паттерна *State Machine.*

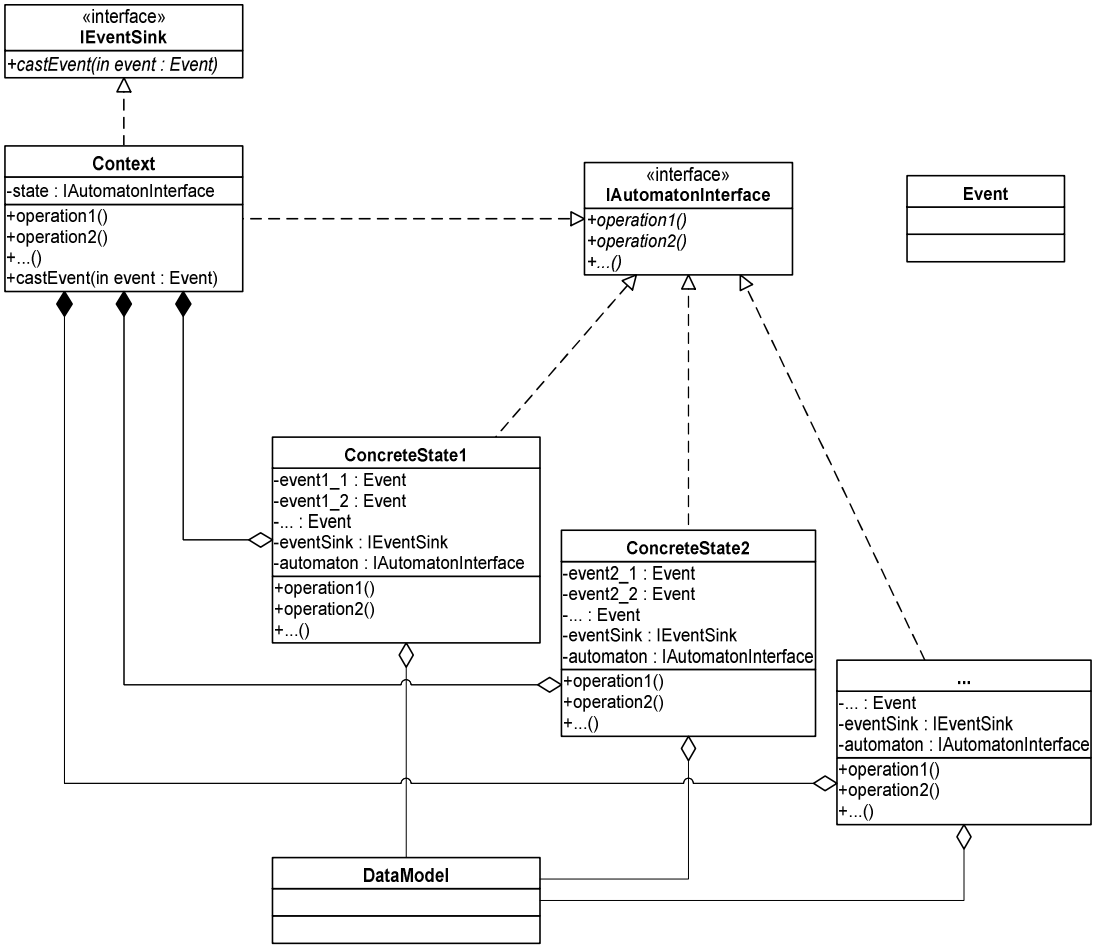

Рис. 3. Структура паттерна *State Machine*

Здесь `IAutomatonInterface` — интерфейс реализуемого объекта, а `operation1`, `operation2`, … — методы этого интерфейса. Этот интерфейс реализуется основным классом `Context` и классами состояний `ConcreteState1`, `ConcreteState2`, … . Для смены состояния объекта используются события `event1_1`, `event2_1`, …, `event2_1`, `event2_2`, …, являющиеся объектами класса `Event`. Класс `Context` содержит ссылки на все состояния объекта (`ConcreteState1` и `ConcreteState2`), а также на текущее состояние (`state`). В свою очередь, классы состояний содержат ссылку на модель данных (`dataModel`) и интерфейс уведомления о событиях (`eventSink`). Для того чтобы не загромождать рисунок, на нем не отражены связи классов состояний и класса `Event`.

Классы `Context`, `ConcreteState1`, `ConcreteState2`, … реализуют интерфейс `IAutomatonInterface`. Класс `Context` содержит переменные типа `IAutomatonInterface`. Одна из них — текущее состояние автомата, а остальные хранят ссылки на классы состояний автомата. Отметим, что стрелки, соответствующие ссылкам на классы состояний, ведут к интерфейсу, а не к этим классам. Это следствие того, что все взаимодействие между контекстом и классами состояний производится через интерфейс автомата. Связи между контекстом и классами состояний отмечены стрелками с ромбом — используется агрегация.

Участники. Паттерн *State Machine* состоит из следующих частей.

1. *Интерфейс автомата* (`IAutomatonInterface`) — реализуется контекстом и является единственным способом взаимодействия клиента с автоматом. Этот же интерфейс реализуется классами состояний.

2. *Контекст* (`Context`) — класс, в котором инкапсулирована логика переходов. Он реализует интерфейс автомата, хранит экземпляры модели данных и текущего состояния.
3. *Классы состояний* (`ConcreteState1`, `ConcreteState2`, …) — определяют поведение в конкретном состоянии. Реализуют интерфейс автомата.
4. *События* (`event1_1`, `event1_2`, ...) — инициируются состояниями и передаются контексту, который осуществляет переход в соответствии с текущим состоянием и событием.
5. *Интерфейс уведомления о событиях* (`IEventSink`) — реализуется контекстом и является единственный способом взаимодействия объектов состояний с контекстом.
6. *Модель данных* (`DataModel`) — класс предназначен для хранения и обмена данными между состояниями.

Отметим, что в предлагаемом паттерне, интерфейс автомата реализуется как контекстом, так и классами состояний. Это позволяет добиться проверки целостности еще на этапе компиляции. В паттерне ***State*** такая проверка невозможна из-за различия интерфейсов контекста и классов состояний.

Результаты. Сформулируем результаты, получаемые при использовании паттерна *State Machine*.

1. Также как и в паттерне ***State***, поведение, зависящее от состояния, локализовано в отдельных классах состояний.
2. В отличие от паттерна ***State*** в предлагаемом паттерне логика переходов (сконцентрированная в классе контекста) отделена от реализации поведения в конкретных состояниях. В свою очередь, классы состояний обязаны только уведомить контекст о наступлении события (например, о разрыве соединения).
3. Реализация интерфейса автомата в классе контекста может быть сгенерирована автоматически, а реализация логики переходов — по графу переходов.
4. Для повышения скорости смены состояний логика переходов может быть реализована специальной таблицей таким образом, что каждый переход осуществляется за одну операцию доступа по индексу.
5. Паттерн ***State Machine*** предоставляет «чистый» (без лишних методов) интерфейс для пользователя. Для того чтобы клиенты не имели доступа к интерфейсу `IEventSink`, реализуемого классом контекста, следует использовать или *закрытое* (***private***) наследование (например, в языке *C++*) или определить закрытый конструктор и статический метод, создающий экземпляр контекста, возвращающий интерфейс автомата. Соответствующий фрагмент кода имеет вид:

```
class Automaton implements IAutomatonInterface {
  private Automaton() {}
  public static IAutomatonInterface CreateAutomaton() {
    return new Automaton();
  }
}
```

6. Паттерн ***State Machine***, в отличие от паттерна ***State***, не содержит дублирующих интерфейсов для контекста и классов состояний — контекст и классы состояний реализуют один и тот же интерфейс.
7. Возможно повторное использование классов состояний, в том числе посредством создания их иерархии. В книге Э. Гаммы и др. сказано «поскольку зависящий от состояния код целиком находится в соответствующем подклассе класса `State` (базовый класс для классов состояний в паттерне ***State***), то добавлять новые

состояния и переходы можно просто путем порождения новых подклассов». На самом же деле, добавление нового состояния зачастую влечет за собой модификацию остальных классов состояний, так как иначе переход в данное состояние не может быть осуществлен. Таким образом, расширение автомата, построенного на основе паттерна *State*, является проблематичным. Более того, при реализации наследования в паттерне *State* также затруднено и расширение интерфейса автомата. Скорее всего, именно по этим причинам указанной книге Э. Гаммы и др. не описано наследование автоматов.

8. Известны работы, посвященные реализации объектов, часть методов которых не зависит от состояния. Для реализации таких объектов предложен паттерн *Three Level FSM*. Данная задача может быть также решена и при помощи паттерна *State Machine*. Для этого следует разделить интерфейс реализуемого объекта и интерфейс автомата. При этом последний реализуется контекстом в соответствии с описываемым паттерном. Для реализации полного интерфейса объекта создается наследник контекста, в котором определяются методы, не зависящие от состояния (рис. 4).

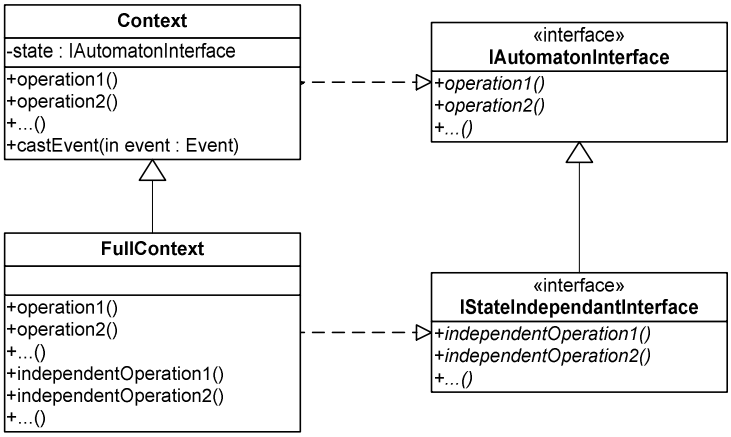

Рис. 4. Реализация методов, не зависящих от состояния

Четвертая глава. В данной главе предлагается новый язык объектно-ориентированного программирования *State Machine*, являющийся расширением языка программирования *Java*. В язык вводятся синтаксические конструкции, позволяющие программировать с использованием понятий автоматного программирования. Для обработки новых синтаксических конструкций разработан препроцессор, преобразующий код на языке *State Machine* в код на языке *Java*. При этом новые синтаксические конструкции преобразуются в код на языке *Java* в соответствии с паттерном *State Machine*.

Особенности языка State Machine. В предлагаемом языке, как и в паттерне *State Machine*, основной идеей является описание объектов, варьирующих свое поведение, в виде автоматов. В предложенном в третьей главе подходе разделяются классы, реализующие логику переходов (контексты), и классы состояний. В автомате переходы инициируются состояниями путем уведомления контекста о наступлении событий. При этом в зависимости от события и текущего состояния, контекст устанавливает следующее состояние в соответствии с графом переходов.

Логика переходов задается в терминах состояний и событий. При этом в языке *State Machine* полностью скрывается природа событий. Для пользователя они представляют собой

сущности, принадлежащие классу состояния и участвующие в формировании функции переходов.

По сравнению с языком *Java* в язык *State Machine* введены дополнительные конструкции, позволяющие описывать объекты с варьирующимся поведением в терминах автоматного программирования: *автоматов*, *состояний* и *событий*. Для описания автоматов и состояний в язык введены ключевые слова `automaton` и `state` соответственно, а для событий — ключевое слово `events`.

Отметим, что в предлагаемом языке, также как и паттерне *State Machine*, события являются основным способом воздействия объекта состояния на контекст. В указанном паттерне программист должен самостоятельно создавать объекты, представляющие события в виде статических переменных класса `Event`, в то время как в языке *State Machine* события введены как часть описания состояний. Это сделано для того, чтобы подчеркнуть их важность.

В паттерне *State Machine* реализация интерфейса автомата в контексте делегирует вызовы методов интерфейса текущему экземпляру состояния, причем делегирование реализуется вручную. В программе на языке *State Machine* это делать не требуется, так как препроцессор автоматически построит соответствующий код.

Так же как в паттерне *State Machine*, в предлагаемом языке состояние может делегировать дальнейшее выполнение действия автомату. Для этого, также как и для задания автоматов, используется ключевое слово `automaton`. При этом состояние, обрабатывающее ошибку, осуществляет действия по восстановлению и, в случае успеха, передает управление новому состоянию автомата.

Описание автоматов и состояний на языке *State Machine* помещаются в файлы с расширением `.sm`. Автором разработан препроцессор, преобразующий код, написанный на предлагаемом языке, в код на языке *Java* (в файлы с расширением `.java`). При этом новые синтаксические конструкции преобразуются в соответствии с паттерном *State Machine*. Препроцессор генерирует код, содержащий параметры типа (`generics`), что позволяет осуществлять проверку типов во время компиляции. Полученный код компилируется при помощи *Java*-компилятора, поддерживающего параметры типа .

Пример использования языка State Machine. В данном разделе особенности новых синтаксических конструкций языка *State Machine* рассматриваются на примере проектирования и реализации класса `Connection`. Этот класс описан в третьей главе.

Описание состояний. Для описания состояния используется ключевое слово `state`. Приведем код состояния `ConnectedState` на языке *State Machine*.

```
package connection;

import java.io.IOException;

public state ConnectedState implements IConnection events
        DISCONNECT, ERROR {
  protected final Socket socket;

  public ConnectedState(Socket socket) {
    this.socket = socket;
  }

  public void connect() throws IOException {
  }

  public void disconnect() throws IOException {
```

```
    try {
      socket.disconnect();
    } finally {
      castEvent(DISCONNECT);
    }
  }
  public int receive() throws IOException {
    try {
      return socket.receive();
    } catch (IOException e) {
      castEvent(ERROR);
      throw e;
    }
  }
  public void send(int value) throws IOException {
    try {
      socket.send(value);
    } catch (IOException e) {
      castEvent(ERROR);
      throw e;
    }
  }
}
```

Состояния на языке *State Machine* описываются аналогично классам на языке *Java*, за исключением того, что при описании состояния обязательно указывается интерфейс автомата (`IConnection`) и список событий, которые это состояние может сгенерировать (`ERROR`, `DISCONNECT`). Также как в паттерне *State Machine* контекст уведомляется о наступлении события вызовом метода `castEvent`.

В предлагаемом языке состояние может реализовывать несколько интерфейсов. При этом первый из реализуемых состоянием интерфейсов будет считаться интерфейсом автомата.

Описание автомата. В языке *State Machine* автомат предназначен для определения набора состояний и переходов.

Для описания автомата применяется ключевое слово `automaton`. Приведем код на предлагаемом языке для автомата `Connection`, реализующий граф переходов (рис. 2).

```
package connection;

public automaton Connection implements IConnection {
  state DisconnectedState disconnected(CONNECT -> connected, ERROR
        -> disconnected);
  state ConnectedState connected(ERROR -> disconnected, DISCONNECT
        -> disconnected);
  public Connection(Socket socket) {
    disconnected @= new DisconnectedState(socket);
    connected @= new ConnectedState(socket);
  }
}
```

Обратим внимание, что класс автомата должен реализовывать ровно один интерфейс, который и считается интерфейсом автомата. В данном примере — это интерфейс `IConnection`.

Состояния (`connected` и `disconnected` классов `ConnectedState` и `DisconnectedState` соответственно) описываются при помощи ключевого слова `state`.

Первое из состояний, описанных в автомате, является стартовым. В данном примере это состояние `disconnected`.

Переходы по событиям описываются в круглых скобках, после имени состояния. Для одного состояния переходы разделяются запятыми. Например, для состояния `connected` переходами являются `DISCONNECT -> disconnected` и `ERROR -> disconnected`. Первый из них означает, что при поступлении события `DISCONNECT` в состоянии `connected` автомат переходит в состояние `disconnected`.

В конструкторе `public Connection(Socket socket)` производится создание объектов состояний. Отметим, что состояния, входящие в автомат, должны реализовать интерфейс автомата. Инициализация объектов состояний производится при помощи нового оператора `@=`, специально введенного для этой цели в язык *State Machine*. Таким образом, оператор `connected @= new ConnectedState(socket)` означает инициализацию состояния `connected` новым объектом класса `ConnectedState`.

За исключением этого, автомат описывается аналогично классу на языке *Java*.

Отметим, что состояния автомата перечисляются, но не определяются в нем. Таким образом, одни и те же состояния могут использоваться для реализации различных автоматов.

Грамматика описания автоматов и состояний

Как отмечено выше, язык программирования *State Machine* основан на языке *Java*, в который вводятся синтаксические конструкции для поддержки программирования в терминах *автомат* и *состояние*.

В данном разделе приводятся грамматики в расширенной форме Бэкуса-Наура для описания этих конструкций.

Грамматика описания состояния

```
state_decl        ::=  modifiers state
                       type
                       extends_decl?
                       implements_decl
                       events?
                       {
                       balanced
                       }
extends_decl      ::=  extends type
implements_decl   ::=  implements type (, type)*
type              ::=  id (. id)*
events            ::=  events id (, id)*
balanced          ::=  <сбалансированная по скобкам
                       последовательность>
modifiers         ::=  (abstract | final | strictfp | public)*
```

Здесь и далее терминальные символы выделены полужирным шрифтом, а нетерминальные — наклонным. Для краткости не раскрывается определение нетерминала *balanced*. Он соответствует сбалансированной относительно использования круглых и фигурных скобок последовательности терминальных и нетерминальных символов.

Состояние должно реализовывать не менее одного интерфейса. При этом первый из них считается интерфейсом автомата.

В коде состояния возможно делегирование методов текущему состоянию автомата. Для этого используется ключевое слово `automaton`, которое имеет тип интерфейса автомата.

Отметим, что в данной версии языка состояния не могут содержать параметры типа.

Грамматика описания автомата

```
automaton_decl   ::=  modifiers automaton
                      type
                      implements_decl
                      {
                      state_var_decl+ balanced
                      }
state_var_decl   ::=  state type id ( event_mapping (, event_mapping)*
                      ) ;
event_mapping    ::=  id (, id)* -> id
```

Отметим, что интерфейс автомата должен совпадать у автомата и всех состояний, которые он использует. Это семантическое правило, поэтому оно не может быть выражено грамматикой.

Для инициализации состояний в конструкторе автомата применяется оператор `@=`. Слева от него указывается имя состояния, а справа — объект, реализующий это состояние. Тип указанного объекта должен в точности совпадать с типом, указанным при описании автомата.

В конструкторе все состояния автомата должны быть проинициализированы. При этом каждое — не более одного раза (как если бы они были обыкновенными переменными, описанными с модификатором `final`).

Использование оператора `@=` вне конструктора автомата является ошибкой.

Реализация препроцессора. Для реализации препроцессора были использованы открытые инструменты создания компиляторов *JLex* и *Cup*, разработанные в Принстонском университете. Первый из них предназначен для построения лексических анализаторов, а второй — синтаксических.

В результате работы препроцессора производится преобразование переданных ему в качестве параметров файлов с расширением `.sm`, содержащих код автоматов и состояний на языке *State Machine*, в файлы с расширением `.java`. В процессе преобразования теряется исходное форматирование программы и комментарии. Это не является недостатком, поскольку получаемый код промежуточный и не предназначен для редактирования вручную.

Препроцессор (в открытых кодах) можно скачать по адресу http://is.ifmo.ru, раздел *Статьи*. Для работы препроцессора необходимо также установить инструменты создания компиляторов *JLex* и *Cup*, доступные по адресу http://www.cs.princeton.edu/~appel/modern/java/.

Пятая глава. В данной главе описывается внедрение результатов работы.

В ***компании Evelopers*** (Санкт-Петербург) разрабатывается для платформы *Eclipse* пакет *Unimod*, предназначенный для поддержки *SWITCH*-технологии в нотации *UML*. В этом пакете при создании системы автозавершения ввода использовался метод преобразования рекурсивных программ в автоматные.

Паттерн *State Machine* внедрен в *департаменте береговых систем* компании *Транзас* (*Санкт-Петербург*) в продукт *Navi Habour*, который является телекоммуникационной системой управления движением судов (*СУДС*).

Постановка задачи. Выполнение асинхронных запросов в одном из компонентов телекоммуникационной системы *Navi Harbour* — базе данных *СУДС* производится посредством создания потоков соединения (*connection threads*). Для создания и удаления потоков соединения необходимо разработать фабрику потоков — класс `ThreadFactory`. Его проектирование осуществлено на основе паттерна *State Machine*.

Проектирование автомата ThreadFactory.

Класс `ThreadFactory` должен обеспечить:

1. Корректное создание и уничтожение потоков соединений.
2. Изоляцию клиента от получения указателя на неинициализированный объект потока.
3. Создание или удаление не более одного потока соединения одновременно.

Выделим четыре управляющих состояния:

1. *Idle* — ожидание запроса на создание или удаления потока соединения.
2. *Creating* — поток создан, ожидается завершение его инициализации.
3. *Destroing* — выполнен запрос на уничтожение потока, ожидание завершения потока.
4. *Cancelling* — выполнение отмены на создание потока.

Названия состояний соответствуют именам классов состояний. Классы генерируют следующие события:

1. *Idle*: `CREATED` — объект потока создан, `STOP_SENDED` — объекту потока отправлен запрос на остановку.
2. *Creating*: `INITIALIZED` — поток инициализирован; объекту потока отправлен запрос на остановку.
3. *Destroing*: `DESTROYED` — поток уничтожен.
4. *Cancelling*: `CANCELLED` — произведена отмена создания потока.

Граф переходов, описывающий поведение класса `ThreadFactory`, приведен на рис. 5.

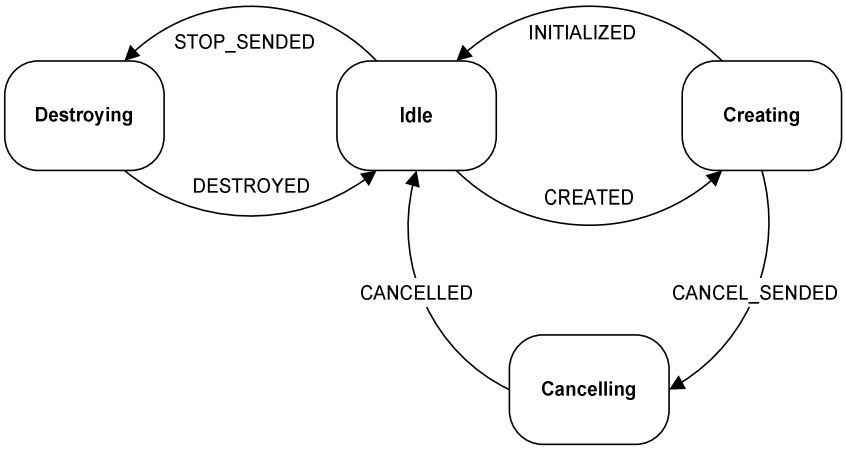

Рис. 5. Граф переходов, описывающий поведение класса *ThreadFactory*

В работе приведен код на языке *C++*, реализующий описанный класс, а также выполнено сравнение с реализациями, основанными на других подходах (реализации с использованием флагов и *SWITCH*-технологии).

Результаты работы внедрены также в ***учебный процесс*** на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсам «Теория автоматов в программировании» и «Программирование на языке *Java*».

Заключение

В диссертации получены следующие научные результаты.

1. Обоснована целесообразность разделения множества состояний на два класса: управляющие и вычислительные на примерах таких классических алгоритмов как, например, «Ханойские башни», «Обход шахматной доски ходом коня», «Обход

деревьев». Показано, что, как и в машине Тьюринга, небольшое число состояний первого типа может управлять огромных числом состояний второго типа.

2. Предложен формальный метод построения автоматных программ на основе традиционных процедурных программ с явной рекурсией.
3. Разработан паттерн проектирования *State Machine*, устраняющий такие недостатки известного паттерна *State* как сложность повторного использования классов состояний, децентрализованная логика переходов и сильная связность классов состояний друг от друга.
4. Для непосредственной поддержки паттерна *State Machine* предложен одноименный язык программирования. Этот язык является расширением языка *Java* за счет введения в него синтаксических конструкций `automaton`, `state`, `events`. Реализован препроцессор, преобразующий код, написанный на этом языке, в код на языке *Java*.
5. Результаты работы внедрены при разработке пакета *Unimod* в компании *eVelopers* и продукта *Navi Habour* в *департаменте береговых систем* компании *Транзас*, а также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсам «Теория автоматов в программировании» и «Программирование на языке *Java*».

Список публикаций

1. **Туккель Н.И., Шалыто А.А., Шамгунов Н.Н.** Реализация рекурсивных алгоритмов автоматными программами //Труды Всероссийской научно-методической конференции «Телематика-2002». СПб.: СПбГУ ИТМО. 2002, с. 181-182.
2. **Туккель Н.И., Шалыто А.А., Шамгунов Н.Н.** Реализация рекурсивных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2002. № 5. с. 72-99. http://is.ifmo.ru, раздел «Статьи»
3. **Туккель Н.И., Шалыто А.А., Шамгунов Н.Н.** Ханойские башни и автоматы //Программист. 2002. № 8, с. 82-90. (http://is.ifmo.ru, раздел «Статьи»).
4. **Туккель Н. И., Шалыто А.А., Шамгунов Н.Н.** Задача о ходе коня //Мир ПК. 2003. № 1. с. 152-155. http://is.ifmo.ru, раздел «Статьи»
5. **Шамгунов Н.Н., Шалыто А.А.**Язык автоматного программирования с компиляцией в Microsoft CLR. // Microsoft Research Academic Days in St. Petersburg, April 21-23, 2004.
6. **Шамгунов Н.Н.**, **Шалыто А.А.** Язык автоматного программирования *State* //Труды Всероссийской научно-методической конференции «Телематика-2004». СПб.: СПбГУ ИТМО. 2004, с. 180-181.
7. **Корнеев Г.А.**, **Шамгунов Н.Н.**, **Шалыто А.А**. Обход деревьев на основе автоматного подхода //Труды Всероссийской научно-методической конференции «Телематика-2004». СПб.: СПбГУ ИТМО. 2004, с. 182-183.
8. **Корнеев Г.А. Шалыто А.А. Шамгунов Н.Н.** Обход деревьев на основе автоматного подхода //Компьютерные инструменты в образовании. 2004. № 3, с. 32-37. http://is.ifmo.ru, раздел «Статьи»
9. **Корнеев Г.А., Шамгунов Н.Н., Шалыто А.А.** *State Machine*. Новый паттерн объектно-ориентированного проектирования//Информационно-управляющие системы. 2004. № 5, с. 13-25. http://is.ifmo.ru, раздел «Статьи»