Санкт-Петербургский государственный
университет информационных технологий, механики и оптики

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

Диссертация на соискание ученой степени
кандидата технических наук

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

Санкт-Петербург – 2004

ОГЛАВЛЕНИЕ

Введение

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* [1] (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 [2] (*UML* —Unified Modeling Language). В этом языке для описания поведения также используется автоматная модель — диаграммы состояний *Statecharts* [3], 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 [4], 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* [5].
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* [6], 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 гранту Российского фонда фундаментальных исследований 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* [7] (Санкт-Петербург) при создании системы автозавершения ввода в пакете автоматно-ориентированного программирования *Unimod* [8].
2. В компании *Транзас* [9] (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 изложена на 195 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит

82 наименований. Работа иллюстрирована 46 рисунками и содержит две таблицы.

Глава 1. Обзор методов, паттернов и языков, применяющихся для описания поведения программ в телекоммуникационных системах

1.1. Методы преобразования процедурных программ в автоматные

Известна работа [10], в которой предложено преобразовывать итеративные программы в автоматные, например, с целью их визуализации. В этой работе не описывается способ преобразования рекурсивных программ в автоматные.

1.2. Паттерны проектирования. Паттерн *State*

Паттерны (образцы) проектирования представляют собой примеры наиболее удачных проектных решений в области объектно-ориентированного программирования. Применительно к теме данной работы наибольший интерес представляет паттерн ***State*** [5].

Паттерн ***State*** является наиболее известной реализацией объекта, изменяющего поведение в зависимости от состояния**.** В указанной работе данный паттерн недостаточно полно специфицирован, поэтому в разных источниках, например в работах [11, 12], он реализуется по-разному (в частности, громоздко и малопонятно). Поэтому многие программисты считают, что этот паттерн не предназначен для реализации автоматов. Другой недостаток паттерна ***State*** состоит в том, что разделение реализации состояний по различным классам приводит и к распределению логики переходов по ним, что усложняет понимание программы. При этом не обеспечивается независимость классов, реализующих состояния, друг от

друга. Таким образом, создание иерархии классов состояний и их повторное использование затруднено.

1.3. Графический язык описаний и спецификаций *SDL*

Для построения телекоммуникационных систем используется графический язык *SDL* [1], разработанный Международным институтом электросвязи (*ITU-T*). Отличительная особенность языка *SDL* состоит в том, что он не предусматривает никакой разницы между спецификацией и описанием. Основу этого языка составляет концепция взаимодействия конечных автоматов и связей между ними, называемых процессами, описываемых *SDL*-диаграммами. Для построения этих диаграмм разработан набор графических символов. На рис. 1 приведен пример [1] описания одного из процессов на языке *SDL*.

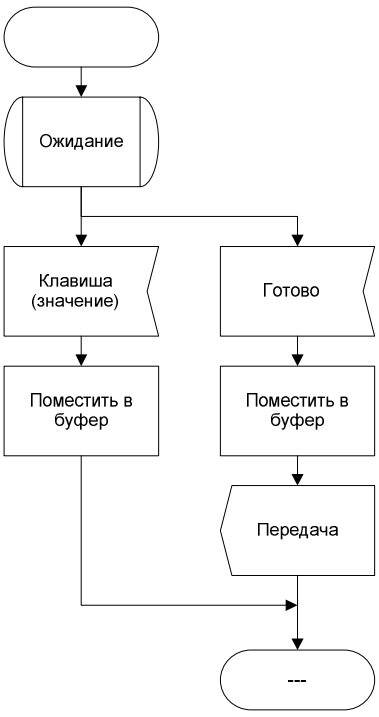

Рис. 1. Пример описания процесса на языке *SDL*

Отметим, что диаграммы в языке SDL весьма громоздки и соответствуют только одному классу автоматов — автоматам Мили [4].

1.4. Унифицированный язык моделирования *UML*

Унифицированный язык моделирования (Unified Modeling Language, *UML*) [2] является графическим языком для визуализации, спецификации, конструирования и документирования систем, в которых большая роль принадлежит программному обеспечению. С помощью *UML* можно разработать детальный план создаваемой системы, отображающий не только ее концептуальные элементы, такие как системные функции и бизнес-процессы, но и конкретные особенности реализации.

UML — это язык диаграмм и обозначений для спецификации, визуализации и документации модели объектно-ориентированных программных систем. Язык *UML* не является методом разработки — он не определяет последовательность действий при создании программного обеспечения. Язык состоит из множества модельных элементов, которые представляют различные компоненты разрабатываемой системы, и обладает следующими особенностями.

1. Предоставляет пользователям язык визуального моделирования. При этом они могут разрабатывать и обмениваться выразительными моделями.
2. Предоставляет механизмы расширения и специализации.
3. Не зависит от определенного языка программирования и процесса разработки;
4. Позволяет интегрировать лучший практический опыт разработок.

Элементы языка *UML* используются для создания диаграмм, которые описывают определенную часть системы или точку зрения на нее. Этот язык состоит из следующих типов диаграмм.

1. *Диаграммы вариантов использования* отображают действующих лиц и их взаимодействие с системой.
2. *Диаграммы классов* отображают классы и взаимодействие между ними.

3. *Диаграммы последовательностей* отображают объекты и их взаимодействие, выделяя хронологию обмена сообщениями между объектами.
4. *Диаграммы взаимодействия* отображают объекты и их взаимодействие, выделяя объекты, которые участвуют в обмене сообщениями.
5. *Диаграммы состояний* отображают состояния, переходы между ними и события, инициирующие эти переходы. Отметим, что эти диаграммы были предложены в работе Д. Харела [3].
6. *Диаграммы активности* являются развитием схем алгоритмов в части реализации параллельных процессов.
7. *Диаграммы компонентов* используется для распределения классов и объектов по компонентам при физическом проектировании системы.
8. *Диаграммы развертывания* используется для анализа аппаратных средств, на которых она будет эксплуатироваться система.

В данной работке неоднократно будут применяться диаграммы классов на языке *UML*.

Недостатки языка *UML* применительно к теме настоящей работы (для описания поведения системы) состоят в том, что диаграммы переходов весьма громоздки, и их нельзя построить по коду программы.

1.5. Язык *ASML*

В настоящее время в компании *Microsoft* развивается язык *AsmL* (Abstract State Machine Language), предложенный Ю. Гуревичем [13]. Этот язык применим в ситуациях, когда необходимо точно специфицировать компьютерную систему в части ее функциональности. Менеджеры, разработчики и тестеры могут использовать язык *AsmL* для спецификации задач.

В языке вводится понятие *абстрактное состояние*. При этом язык предоставляет возможность обеспечить компактное высокоуровневое описание состояния системы.

Описание модели системы производится по шагам. Любое состояние машины может рассматриваться как *словарь* пар (имя, значение) переменных состояния. Прогон машины состоит из серии состояний и переходов. На рис. 2 изображена диаграмма прогона, моделирующий процесс выполнения заказов.

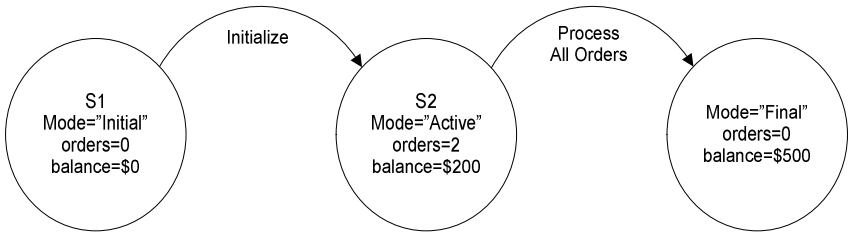

Рис. 2. Прогон из двух переходов и трех состояний

В настоящее время язык *AsmL* еще не получил достаточного распространения, особенно в сфере коммуникации.

1.6. *SWITCH*-технология

С 1991 года в России развивается *SWITCH*-технология, которая предназначена для проектирования программ на основе конечных автоматов. Качество программ, построенных с ее использованием достигается за счет выразительных средств графов переходов и изоморфного перехода от графов к программам. Эта технология успешно зарекомендовала себя при создании систем логического управления. С применением этой технологии студентами и аспирантами *СПбГУ ИТМО* было создано более пятидесяти проектов [14] в самых разных областях разработки программного обеспечения. Это позволило рассмотреть технологию с самых разных сторон и проанализировать ее сильные и слабые стороны.

Для *SWITCH*-технологии разработана графическая нотация для описания конечных автоматов. Для автоматизации построения диаграмм в этой нотации разработан инструмент проектирования *Unimod* [7], подключаемый к среде разработки *Eclipse* [15].

Отметим, что SWITCH-технология обладает рядом недостатков.

1. Монолитность — в отличие от реализации на основе паттерна *State Machine* невозможно повторно использовать составные части кода класса `ThreadFactory.`
2. При необходимости добавления входных и (или) выходных воздействий могут возникать ситуации, при которых компилятор не сможет обнаружить некоторые семантические ошибки, такие как, например, несоответствие метода интерфейса класса с вызовом автомата с соответствующим событием.

Выводы

Из изложенного выше следует.

1. В настоящее время отсутствует метод преобразования рекурсивных программ в автоматные, например, для языков, в которых нет рекурсии.
2. В настоящее время среди паттернов проектирования для описания объектов с варьирующимся поведением используется паттерн *State*, обладающий рядом недостатков.
3. Среди языков программирования отсутствуют текстовые объектно-ориентированные языки, предназначенные для эффективного программирования в рамках предметной области — автоматного программирования

Настоящая работа направлена на устранение недостатков во всех трех указанных направлениях.

Глава 2. Применение конечных автоматов для реализации вычислительных алгоритмов

2.1. Задача о Ханойских башнях

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях [16], которая формулируется следующим образом. Имеются три стержня, на первом из которых размещено N дисков. Диск наименьшего диаметра находится сверху, а ниже — диски последовательно увеличивающегося диаметра. Задача состоит в определении последовательности перекладываний по одному диску со стержня на стержень, которые должны выполняться так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы, в конце концов, все диски оказались на другом стержне.

Покажем, что применение автоматов [17] позволяет формально переходить к итеративным алгоритмам решения этой задачи, либо строить такие алгоритмы непосредственно. В работах [18, 19] приведены примеры преобразований рекурсивных программ в итеративные, однако подходы, обеспечивающие такие преобразования, не формализованы. Настоящая работа призвана устранить этот пробел.

В работе предлагаются три метода. Первый из них обеспечивает формальное построение автоматной программы (программы, построенной с использованием автоматов) на основе раскрытия рекурсии с применением стека. Второй метод также обеспечивает раскрытие рекурсии и состоит в построении автомата, осуществляющего обход дерева действий, выполняемых рекурсивной программой. Третий метод состоит в непосредственном управлении дисками и стержнями, и не использует таких абстракций как деревья и стеки. Отметим, что применение каждого из

предлагаемых методов для рассматриваемой задачи порождает автоматы с двумя или тремя состояниями, управляющие «объектом управления» с $2N$ состояниями, возникающими в процессе перекладывания дисков.

2.1.1. Классическое рекурсивное решение задачи

Известны рекурсивные алгоритмы для решения рассматриваемой задачи [16, 20]. Приведем один из таких алгоритмов, построенный по методу «разделяй и властвуй». Задача о перекладывании `N` дисков с `i`-го на `j`-ый стержень может быть декомпозирована следующим образом.

1. Переложить `N–1` диск со стержня с номером i на стержень с номером `6–i–j`.
2. Переложить диск со стержня с номером i на стержень с номером `j`.
3. Переложить `N–1` диск со стержня с номером `6–i–j` на стержень с номером `j`.

Таким образом, задача сводится к решению двух аналогичных задач размерности `N–1` и одной задачи размерности один. При этом если для решения одной задачи размерности `N–1` требуется F_N-1 перекладываний, то их общее число определяется соотношением:

$$F_N = 2F_N-1 + 1.$$

Из этого соотношения следует [21], что

$$F_N = 2_N - 1.$$

Указанный процесс декомпозиции можно изобразить с помощью дерева декомпозиции (рис. 3).

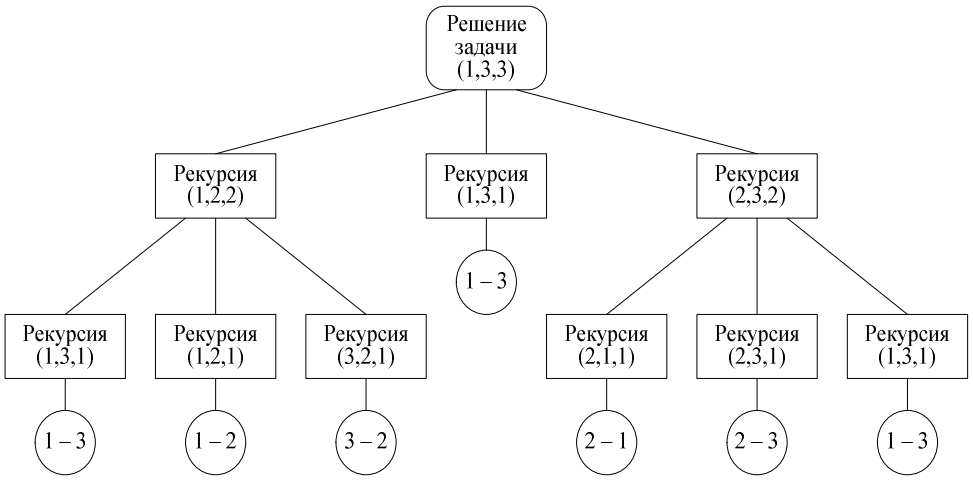

Рис. 3. Дерево декомпозиции для задачи о ханойских башнях при **`N = 3`**

В этом дереве вершины, обозначенные прямоугольниками, соответствуют подзадачам, решаемым при каждом вызове рекурсивной функции. Эти вершины помечаются номерами стержня, с которого перекладывается диск, стержня, на который перекладывается диск, и числом перекладываемых дисков. При этом для вершины с пометкой (`i,j,k`) левая вершина поддерева будет помечена (`i,6-i-j,k-1`), средняя — (`i,j,1`), а правая — (`6-i-j,j,k-1`).

Перекладывания дисков выполняются в вершинах, обозначенных кружками. Для каждой из этих вершин сохраняются первые две позиции пометки соответствующего прямоугольника — (`i,j`). Количество таких вершин равно F_N.

Приведем программу, написанную на языке *Си* и реализующую рассматриваемый рекурсивный алгоритм. Ее поведение эквивалентно обходу дерева декомпозиции слева направо, причем в вершинах обозначенных кружками производятся перекладывания.

```
#include <stdio.h>

void hanoy( int i, int j, int k, int d )
{
  int m ;
```

```
  for( m = 0 ; m < d ; m++ ) printf( "   " ) ;
  printf( "hanoy(%d,%d,%d)\n", i, j, k ) ;

  if( k == 1 )
    move( i, j, d ) ;
  else
  {
    hanoy( i, 6-i-j, k-1, d+1 ) ;
    hanoy( i, j, 1, d+1 ) ;
    hanoy( 6-i-j, j, k-1, d+1 ) ;
  }
}

void move( int i, int j, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "   " ) ;
  printf( "move(%d,%d)\n", i, j ) ;
}

void main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input, 0 ) ;
}
```

В эту программу введено протоколирование рекурсивных вызовов и четвертый параметр рекурсивной функции, используемый для определения глубины рекурсии. Ниже приводится протокол работы программы, эквивалентный дереву декомпозиции (рис. 3).

```
Ханой с 3 дисками:
hanoy(1,3,3)
  hanoy(1,2,2)
    hanoy(1,3,1)
```

```
  move(1,3)
  hanoy(1,2,1)
  move(1,2)
  hanoy(3,2,1)
  move(3,2)
hanoy(1,3,1)
move(1,3)
hanoy(2,3,2)
  hanoy(2,1,1)
  move(2,1)
  hanoy(2,3,1)
  move(2,3)
  hanoy(1,3,1)
  move(1,3)
```

2.1.2. Обход дерева действий

Дерево декомпозиции (рис. 3), вершинами которого являются рекурсивные вызовы, может быть преобразовано в дерево действий, выполняемых рассмотренной рекурсивной программой, путем исключения всех вершин кроме тех, в которых выполняется перекладывание (рис. 4).

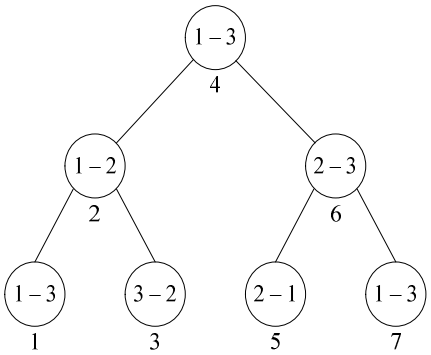

Рис. 4. Дерево действий при **N = 3**

Это дерево обладает следующим свойством: для вершины с пометкой (`i,j`) вершина левого поддерева имеет пометку (`i,6-i-j`), а вершина правого поддерева — (`6-i-j,j`).

Обход полученного дерева может быть выполнен как рекурсивно, так и итеративно. Построение итеративного алгоритма обхода может рассматриваться как способ раскрытия рекурсии. При этом необходимая последовательность перекладываний получается в результате обхода этого дерева слева направо. Такому обходу соответствуют числа, указанные рядом с каждой из вершин.

Построим итеративный алгоритм обхода этого дерева. Не храня его в памяти, будем осуществлять двоичный поиск каждой вершины, начиная с корневой, так как для нее известны номера стержней и способ определения их номеров для корневых вершин левого и правого поддеревьев. Например, для вершины с номером 5, на первом шаге алгоритма осуществляется переход от вершины 4 к ее правому поддереву — вершине 6, так как пять больше четырех. На втором шаге осуществляется переход от вершины 6 к ее левому поддереву — искомой вершине 5, так как пять меньше шести. Таким образом, несмотря на то, что обход дерева осуществляется слева направо, алгоритм работает сверху вниз.

Итеративная программа обхода дерева действий:

```
#include <stdio.h>

void hanoy( int i, int j, int k )
{
  int max_nodes = (1 << k) - 1 ;          // Всего
        вершин.
  int root = 1 << (k-1) ;                 // Номер
        корневой вершины.
  int node ;                              // Номер
        искомой вершины в дереве.

  // Определить номера стержней для каждой вершины в
        дереве.
  for( node = 1 ; node <= max_nodes ; node++ )
  {
```

```
    int a = i, b = j ;
    // Начальная позиция поиска соответствует корневой
        вершине.
    int current = root ;
    // Изменение номера вершины при переходе к следующему
        поддереву.
    int ind = root / 2 ;

    // Двоичный поиск нужной вершины.
    while( node != current )
    {
      if( node < current )
      {
        // Искомая вершина в левом поддереве.
        b = 6-a-b ;
        current -= ind ; // Переход к левому поддереву.
      }
      else
      {
        // Искомая вершина в правом поддереве.
        a = 6-a-b ;
        current += ind ; // Переход к правому поддереву.
      }
      // Разница в номерах вершин при переходе к
      // следующему поддереву уменьшается в два раза.
      ind /= 2;
    }

    // Номера стержней для рассматриваемой вершины
        определены.
    printf( "Вершина %d. %d -> %d\n", node, a, b ) ;
  }
}
```

```
void main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}
```

2.1.3. Непосредственное перекладывание дисков

Если алгоритм решения рассматриваемой задачи задавать непосредственно в терминах объекта управления (дисков и стержней), как это предлагается П. Бьюнеманом и Л. Леви [22], то его удается описать в виде графа переходов автомата с двумя состояниями (рис. 5).

Этот автомат по очереди выполняет всего два действия: перекладывает наименьший диск циклически по часовой стрелке (`z1`) и перекладывает единственно возможный диск, кроме наименьшего (`z2`). После выполнения действия `z1` автомат проверяет условие завершения алгоритма (`x1`).

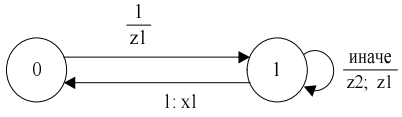

Рис. 5. Граф переходов при непосредственном перекладывании дисков

В этой программе, реализующей этот граф переходов функции `z1`, `z2` и `x1` могут быть реализованы по-разному. Например, определение номеров перекладываемого диска и участвующих в перекладывании стержней может выполняться с помощью перебора (как это имеет место ниже), либо аналитически, например, как это предложено в работе [23].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int y = 0 ;
```

```
int N ;            // Количество дисков.
int dest ;          // На какой стержень перекладываем.
int step = 0 ;      // Номер текущего шага.
int max_steps = 0 ;// Необходимое количество шагов.
int first_on = 1 ;  // На каком стержне находится первый
        диск.

// Состояние объекта управления - "содержимое" стержней.
char  s[4][100] = { "", "1", "", "" } ;

void main()
{
  int input = 14 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}

void hanoy( int from, int to, int disk_num )
{
  int i ;

  N = disk_num ;
  max_steps = (1 << N) - 1 ;
  dest = to ;

  // Заполнить первый стержень дисками.
  for( i = 2 ; i <= N ; i++ )
  {
    sprintf( s[0], "-%d", i );
    strcat( s[1], s[0] ) ;
  }

  do
    switch( y )
    {
```

```
      case 0:
        z1() ;                  y = 1 ;
      break ;

      case 1:
        if( x1() )              y = 0 ;
        else
        { z2() ; z1() ; }
      break ;
    }
  while( y != 0 ) ;

  // Вывести результат.
  printf( "\nРезультат:\n" ) ;
  for( i = 1 ; i <= 3 ; i++ )
    printf( "%d: %s\n", i, s[i] ) ;
}

// Проверить, завершено ли перекладывание.
int x1() { return step >= max_steps ; }

// Переложить первый диск по часовой стрелке.
void z1()
{
  int from = first_on ;
  int i ;

  // Определить номер следующего стержня.
  if((dest == 2 && N%2 != 0)
      || (dest == 3 && N%2 == 0))
    first_on = (from + 1)%3 ; // Порядок стержней 1-2-3.
  else
    first_on = (from + 2)%3 ; // Порядок стержней 1-3-2.
  if( first_on == 0 ) first_on = 3 ;
  move( 1, from, first_on ) ;
```

```
}

// Переложить единственный возможный диск, кроме
        наименьшего.
void z2()
{
  int i, j ;
  int disk_from, disk_to ;

  // Определить перекладываемый диск.
  for( i = 1 ; i <= 3 ; i++ )
  {
    disk_from = disk_on(i) ;
    if( disk_from > 1 )
      // Определить на какой стержень перекладывать.
      for( j = 1 ; j <= 3 ; j++ )
      {
        disk_to = disk_on(j) ;
        if( disk_to == 0 || disk_from < disk_to )
        {
          move( disk_from, i, j ) ;
          return ;
        }
      }
  }
}

// Вернуть номер диска на указанном стержне.
int disk_on( int s_num ) { return atoi( s[s_num] ) ; }

// Переложить заданный диск.
int move( int disk, int from, int to )
{
  char  *str_pos = strchr( s[from], '-' ) ;
```

```
    if( str_pos == NULL )
      s[from][0] = 0 ;
    else
      strcpy( s[from], str_pos+1 ) ;

    if( s[to][0] == 0 )
      sprintf( s[to], "%d", disk ) ;
    else
    {
      strcpy( s[0], s[to] ) ;
      sprintf( s[to], "%d-%s", disk, s[0] ) ;
    }

    step++ ;
    printf( "Шаг %d. Диск %d: %d -> %d\n", step, disk,
          from, to ) ;

    return 0 ;
  }
```

Особенность рассматриваемой программы состоит в том, что несмотря на простоту автомата, она содержит достаточно много строк, так как выполняемые в ней действия `z1` и `z2` являются сложными. При этом программа непосредственно управляет дисками, и поэтому необходимо запоминать их расположение и реализовать соответствующие вспомогательные функции.

Кроме того, в используемом алгоритме направление перекладывания первого диска (функция `z1`) зависит от четности числа дисков и номера стержня, на который их требуется переложить. При перекладывании дисков с первого на третий стержень, первый диск следует перекладывать в порядке номеров стержней `1–2–3` при четном количестве дисков, и в порядке `1–3–2` при их нечетном количестве.

В заключение раздела отметим, что обычно под состоянием программы понимается значение ячеек ее памяти [24]. Однако, такое определение не

конструктивно, так как практически в любой программе, реализующей вычислительный алгоритм, число указанных значений чрезвычайно велико. Для решения этой проблемы обратим внимание на то, что в машине Тьюринга небольшое количество состояний в автомате может управлять произвольным количеством состояний на ленте [17]. Такая же ситуация имеет место и в рассмотренных примерах, в которых автомат, содержащий два или три состояния, управляет 2^N состояниями «объекта управления», где N — число дисков.

Отметим, что приведенные выше графы переходов по структуре различны, но порождают одинаковые последовательности перекладываний.

Несмотря на то, что длина автоматных программ превышает размер традиционных, такие программы обладают тем преимуществом, что по их тексту может быть формально построен граф переходов. Он является наиболее компактной и удобной для визуализации и документирования графической формой представления программ и их алгоритмов.

2.2. Задача о ходе коня

К одному из наиболее интересных разделов программирования относятся задачи из области искусственного интеллекта, в которых решение ищется методом «проб и ошибок» [27, 29]. При этом имеет место перебор при поиске решения, продолжительность которого может быть сокращена за счет применения тех или иных эвристических правил — методов оптимизации.

Класс алгоритмов, позволяющий решать подобные задачи, в англоязычной литературе называется ***backtracking algorithms*** (*алгоритмы с возвратом*). Такие алгоритмы применяются в тех случаях, когда не подходят более «направленные» методы [25].

Исследуем одну из наиболее известных задач этого класса — задачу о ходе коня [25,26,28]. Она состоит в том, чтобы найти обход доски размером N×M конем, перемещающимся по правилам шахматной игры. Если такой

обход существует, то каждое поле посещается только один раз (выполняется N×M−1 шагов). Проанализируем методы оптимизации и исследуем работу итеративной, рекурсивной и автоматной программ.

2.2.1. Методы оптимизации

Рассмотрим следующие методы оптимизации.

1. Определение клеток, обход из которых невозможен.
2. Выявление заблокированных клеток.
3. Применение правила Варнсдорфа.
4. Использование различных массивов ходов коня.

2.2.2. Определение клеток, обход из которых невозможен

Если количество клеток доски нечетно (число белых и черных клеток отличается на единицу), то обход из некоторых клеток не существует. Отметим, что путь коня проходит по клеткам, чередующимся по цвету. Если общее число клеток доски нечетно, то первая и последняя клетки пути, пройденного конем, будут одного и того же цвета. Таким образом, обход будет существовать только тогда, когда он начнется клеткой того цвета, который имеют наибольшее число клеток.

Выполнение этого условия проверяется следующей функцией:

```
int solution_impossible()
{
  // Если произведение сторон доски нечетно и сумма
        координат начальной позиции
  // нечетна, то решения не существует.
  return ((size_x*size_y) % 2 != 0) && ((start_x+start_y)
        % 2 != 0) ;
}
```

Однако приведенное правило не охватывает всех клеток, для которых обхода не существует. Так, для доски размером 3×7, помимо тех клеток, для

которых выполняется приведенное правило, обход невозможен также из клетки $(2, 4)$.

2.2.3. Выявление заблокированных клеток

Если при очередном ходе образуется клетка, куда можно войти, но откуда нельзя выйти, то такой ход недопустим, за исключением предпоследнего в обходе. Данный метод оптимизации позволяет значительно сократить число возвратов, в том числе и при совместном использовании с правилом Варнсдорфа.

Развитием этого метода является определение групп заблокированных клеток, связанных друг с другом, но отрезанных от остальной части доски. В рассматриваемой программе определяются группы из двух заблокированных клеток, что значительно уменьшает количество возвратов для небольших досок, а при использовании вместе с правилом Варнсдорфа — и для больших (например, размером 100×100 клеток).

2.2.4. Применение правила Варнсдорфа

Среди многих эвристических методов [28], используемых для сокращения перебора, правило Варнсдорфа является наиболее простым. В соответствии с ним при обходе доски коня следует каждый раз ставить на то поле, из которого он может сделать наименьшее число ходов по еще не пройденным полям. Если таких полей несколько, то можно выбрать любое из них, что может, однако, завести коня в тупик и потребовать возврата [28]. Отметим, что наилучшим образом правило Варнсдорфа работает для угловых полей.

2.2.5. Использование различных массивов ходов коня

Ходы коня могут быть заданы, например, в виде следующего массива:

```
int possible_moves_sh[][2] = {
{-1, -2}, {-2, -1}, {-2,  1}, { 1, -2},
```

```
{-1,  2}, { 2, -1}, { 1,  2}, { 2,  1} };
```

Каждый его элемент определяет один возможный ход коня и содержит изменения его координат на доске при совершении хода. При использовании различных массивов для ходов коня количество возвратов может различаться. В программе применяются пять эвристически выбранных массивов, содержащих возможные ходы коня в различном порядке. Также задается максимальное число возвратов (GOOD_RET), и когда оно будет достигнуто, поиск пути начинается заново с использованием уже другого массива. При поиске обхода с применением последнего массива количество возвратов ограничивается значением MAX_RET. Если при совместном использовании всех предложенных методов оптимизации установить значение GOOD_RET равным единице, то для досок, близких к квадратным, можно строить обходы без единого возврата для всех клеток, из которых существует обход. Обход без единого возврата из каждой клетки не удается получить для «вытянутых» досок (например, 14×3) и для больших досок, например для доски 100×100 клеток.

2.2.6. Итеративная программа

Идея алгоритма для итеративной программы заключается в следующем.

1. На каждом шаге ищется фрагмент пути, начинающийся из текущей клетки и не включающий уже пройденные.
2. При совершении хода из массива возможных ходов извлекается очередной элемент, который приводит в незанятую клетку, находящуюся в пределах доски.
3. Если ход невозможен, то происходит возврат в предыдущую клетку (отмена хода).
4. Поиск пути считается успешным тогда, когда номер текущего хода станет равным количеству клеток на доске. Если из начальной клетки перебраны все возможные ходы, то пути не существует.

Ниже приведена структура функции, выполняющей перебор:

```
void find_path()
{
  for( move_num = 1 ; move_num <= max_moves ; )
  {
    if( make_move() )  // Сделать ход.
      move_num++ ;
    else      // Ход невозможен.
    if( move_num > 1 )
    {
      undo_move() // Отменить ход.
      move_num--;
    }
    else
      return;   // Обход невозможен.
  }
}
```

Номер использованного варианта хода для каждого из ходов запоминается в массиве, размер которого выбирается в соответствии с размером доски.

Описанный алгоритм осуществляет перебор вариантов и находит решение, если оно имеется. Отсутствие решения приводит к полному перебору всех вариантов.

Для некоторых клеток программа работает чрезвычайно медленно уже при небольших размерах доски. Например, для доски 6×6 при старте из клетки $(5,2)$ поиск пути требует более 290 миллионов возвратов.

2.2.7. Рекурсивная программа

Наиболее известное решение для задачи обхода конем — рекурсивное [29]. При этом структура функции, выполняющей перебор, имеет следующий вид:

```
int find_path( int cur_x, int cur_y, int move_num )
{
  desk_state[cur_x][cur_y] = move_num ; // Запомнить ход.
  if( move_num > max_moves ) return 1 ; // Проверить
        завершение обхода.
  // Проверить каждый возможный ход из текущей клетки.
  for( int i = 0 ; i < 8 ; i++ )
  {
    int next_x = cur_x + possible_moves[i][0] ; //
        Определить следующее поле.
    int next_y = cur_y + possible_moves[i][1] ;
    if(move_possible( next_x, next_y )
      && find_path( next_x, next_y, move_num+1 )) return
        1 ;
  }

  // Возврат.
  desk_state[cur_x][cur_y] = 0 ;
  back_ret++ ;
  return 0 ;
}
```

В этой программе могут быть использованы все виды оптимизаций, описанные выше.

2.2.8. Автоматная программа

Если первые две программы для решения этой задачи вполне традиционны [10], то автоматные к таковым не относятся.

Отметим, что автоматная программа может быть формально построена по описанной выше итеративной программе с помощью метода, изложенного в работе [10]. Автоматная программа может быть также формально построена и по приведенной рекурсивной программе с использованием метода, предложенного в работе [30].

Кроме того, можно создать автоматную программу путем непосредственного построения автомата по условиям задачи. На рис. 6 для этого автомата приведена схема связей, а на рис. 7 граф переходов автомата Мили.

Рис. 6. Схема связей автомата для задачи о ходе коня

Этот автомат использует функции и переменные, определенные в итеративной программе. Поэтому в автоматной программе применяются все рассмотренные методы оптимизации, а получаемые с ее помощью результаты совпадают с результатами работы итеративной программы.

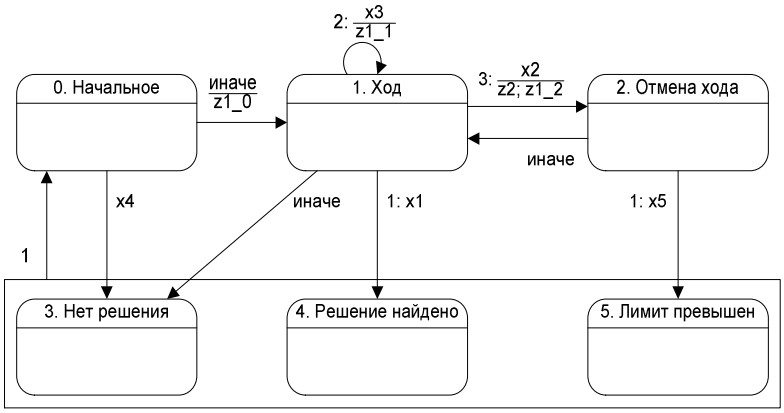

Рис. 7. Граф переходов автомата для задачи о ходе коня

Упрощенный текст функции, реализующей этот автомат, приведен ниже:

```
void find_path_switch( int limit )
{
  y = 0 ;

  do
    switch( y )
    {
      case 0:
        if( x4())              y = 3;
        else
        { z1_0();              y = 1;}
      break;
      case 1:
        if( x1())              y = 4;
        else
        if( x3()) { z1_1() ; }
        else
        if( x2()) { z2() ; z1_2();   y = 2;}
        else                   y = 3;
        break ;
      case 2:
```

```
        if( x5(limit))              y = 5;
        else
                                y = 1;
        break ;
      case 3:                   y = 0; break;
      case 4:                   y = 0; break;
      case 5:                   y = 0; break;
    }
    while( y < 3 );
  }
```

Совместное применение достаточно простых и известных методов оптимизации позволило резко сократить перебор, благодаря чему удается относительно быстро находить пути в досках весьма большой размерности. Так, на компьютере, оснащенном процессором Pentium II 400-МГц, поиск обхода из каждой клетки доски размером 200×200 занял около 20 минут (на поиск одного обхода — около $0{,}03$ с). При этом для большинства клеток обход выполняется без единого возврата назад. В программе, наряду с рассмотренными, могли бы использоваться и другие методы оптимизации [28]. Однако на досках очень большого размера, например 2000×2000 клеток, нахождение даже одного пути занимает значительное время и при применении методов оптимизации, позволяющих строить обходы без единого возврата.

2.3. Обход деревьев

Первоначально в данной рассматривается обход двоичных деревьев, а затем предлагаемое решение обобщается на случай k-ых деревьев. При этом рассматриваются алгоритмы обхода, как использованием стека, так и без его применения. Использование автоматного подхода позволяет получить наглядные и универсальные (в отличие от работы [31]) алгоритмы решения рассматриваемых задач, которые также весьма экономны по памяти по сравнению с классическими алгоритмами.

Отметим, что в работе [32] рассмотрен обход деревьев с произвольным количеством потомков. Для реализации обхода в этой работе вводится состояние, но автомат в явном виде не строится

2.3.1. Постановка задачи обхода двоичного дерева

Пусть задано двоичное дерево, например, представленное на рис. 8.

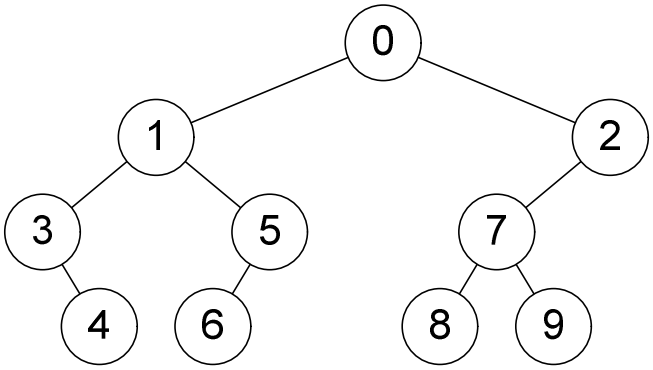

Рис. 8. Двоичное дерево

Необходимо осуществить его обход — сформировать последовательность номеров его вершин. Рассмотрим три порядка обхода дерева: прямой (preorder), обратный (postorder) и центрированный (inorder) [31, 33]. Такие порядки используются при обходе дерева в глубину. В качестве примера приведем последовательности, порождаемые обходами дерева, представленного на рис. 8:

прямой обход: 0, 1, 3, 4, 5, 6, 2, 7, 8, 9;

центрированный обход: 3, 4, 1, 6, 5, 0, 8, 7, 9, 2;

обратный обход: 4, 3, 6, 5, 1, 8, 9, 7, 2, 0.

Данная задача может быть решена несколькими способами: рекурсивно [31, 33], итеративно с применением стека и итеративно без использования стека [34].

Ни в одном из этих подходов не используются автоматы, что, по мнению авторов, не позволило обеспечить наглядность и универсальность предлагаемых решений.

2.3.2. Описание структур данных для представления двоичных деревьев

Будем представлять бинарное дерево в программе следующим образом:

```
struct Node {
int data;   // Данные в узле
  Node* l;  // Левое поддерево
  Node* r;  // Правое поддерево
  Node* p;  // Указатель на родителя
};
```

В этой структуре содержатся ссылки на правое и левое поддеревья, а также ссылка на родительскую вершину, что в дальнейшем, в частности, позволит реализовать рассматриваемый алгоритм без стека. В приводимых примерах на языке *C++* для хранения указателей на вершины дерева будем использовать переменные типа `Node*`. При этом указатель на вершину объявляется как `Node* node`. Доступ к переменным, указывающим на левое и правое поддеревья, а также на родительскую вершину осуществляется при помощи оператора `->`. Поэтому в графах переходов в условиях, помечающих дуги, будем применять следующие обозначение языка *C++*: `node->l`, `node->r` и `node->p` соответственно.

2.3.3. Ввод деревьев

Ввод деревьев в примерах будем производить в следующем виде. В первое строке записано число `N` (количество вершин в дереве). Последующие `N` строк содержат описания вершин дерева. Для `i`-ой вершины в строке с номером `i+2` (вершины нумеруются с нуля) указываются номера корней

левого и правого поддеревьев. В случае отсутствия поддерева вместо номера корня записывается число `-1`. Для рассматриваемого примера исходные данные имеют следующий вид:

```
  10
 1  2
 3  5
 7 –1
-1  4
-1 –1
 6 –1
-1 –1
 8  9
-1 –1
-1 -1
```

2.3.4. Обход двоичного дерева без использования стека

Рассмотрим обход двоичного дерева без применения стека. Идея предлагаемого алгоритма состоит в том, что при обходе двоичного дерева могут быть выделены следующие направления движения: влево, вправо, вверх. Обход начинается от корня дерева. В зависимости от номера текущей вершины и направления движения, можно принять решение о том, куда двигаться дальше. При этом стек не требуется — как уже отмечалось выше, используется указатель на родительскую вершину.

Каждому из указанных направлений может быть сопоставлено состояние автомата, реализующего алгоритм. Также вводится начальное состояние, одновременно являющееся конечным.

Таким образом, автомат имеет следующие состояния:

0. — *Start* — начальное (конечное) состояние.

1. — *Left* — движение влево.

2. — *Right* — движение вправо.

3. — *Parent*— движение вверх.

Автомат оперирует переменной `node` (имеющей тип `Node*`), являющейся указателем на текущий узел дерева.

Поясним, как строится граф переходов, описывающий поведение автомата. Переходы между состояниями определяются условиями, которыми помечаются соответствующие дуги графа переходов. Условия `node->l`, `node->r` и `node->p` обозначают наличие у текущего узла левого потомка, правого потомка и родителя соответственно. Отрицание будем обозначать восклицательным знаком (`!`). Например, переход из состояния *Parent* в состояние *Start* осуществляется при условии, что у вершины нет родителя.

Некоторые дуги графа помечены не только условиями (расположены в числителе дроби), но и действиями (расположены в знаменателе дроби). Например, переход из состояния *Right* в состояние *Left* производится при условии, что у вершины есть правое поддерево. При этом на указанном переходе выполняется действие — изменяется переменная `node`, в которой хранится указатель на текущую вершину.

Кроме действий на переходах в автомате также выполняются действия в состояниях, например, в состоянии *Left* производится вывод номера текущей вершины на экран.

Граф переходом для реализации нерекурсивного обхода двоичного дерева без использования стека приведен на (рис. 9).

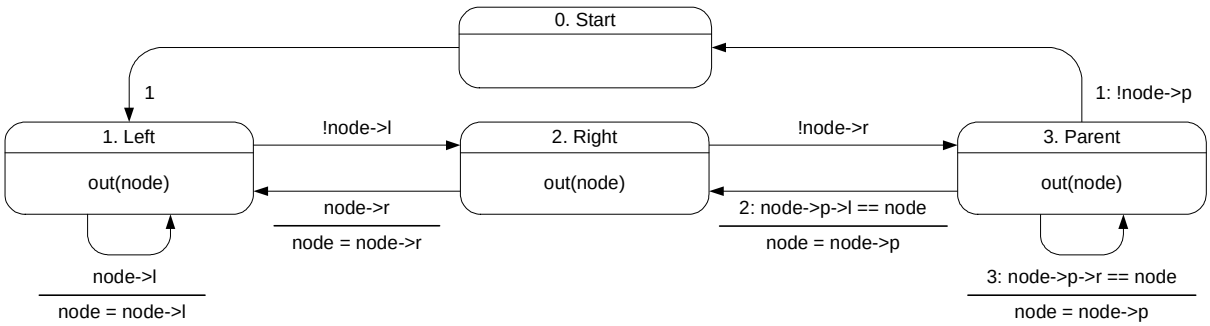

Рис. 9. Граф переходов автомата для реализации

обхода двоичного дерева без применения стека

Отметим, что в случае безусловного перехода из одного состояния в другое дуга помечается символом 1 (переход из нулевой вершины в первую). Еще одной особенностью этого графа является пометка дуг «числами с двоеточием» — приоритетами. Эти пометки указывают последовательность, в которой будут проверяться условия на дугах, исходящих из вершины. В случае если условия являются попарно ортогональными, то приоритеты не расставляются.

Отметим, что если в работах [31, 33] рассмотрено три типа обходов и для каждого из них предложен свой алгоритм, то предлагаемый подход позволяет, используя один и тот же граф переходов автомата, осуществлять все три обхода, за счет вывода номера текущей вершины только в одном из состояний: ***Left***, ***Right***, ***Parent***. При этом в зависимости от выбранного состояния получим три стандартных обхода [31, 33]:

Состояние	Обход
Left	Прямой
Right	Центрированный
Parent	Обратный

Таким образом, предложенный автомат представляет стандартные виды обхода в единой форме.

```
void traverseWithoutStack(Node const* node, int show) {
  State state = START;
  do {
    switch (state) {
      case START:
        state = LEFT;
        break;
      case LEFT:
        if (show & SHOW_LEFT) cout << node->data << " ";
        if (node->l) {
```

```
            node = node->l;
            state = LEFT;
          } else {
            state = RIGHT;
          }
          break;
        case RIGHT:
          if (show & SHOW_RIGHT) cout << node->data << " ";
          if (node->r) {
            node = node->r;
            state = LEFT;
          } else {
            state = PARENT;
          }
          break;
        case PARENT:
          if (show & SHOW_PARENT) cout << node->data << "
          ";
          if (node->p) {
            if (node->p->l == node) {
              node = node->p;
              state = RIGHT;
            } else {
              node = node->p;
              state = PARENT;
            }
          } else {
            state = START;
          }
          break;
      }
    } while(state != START);
  }
```

2.3.5. Обход двоичного дерева с использованием стека

Рассмотрим представление дерева, узлы которого не содержат ссылок на родителей. В этом случае структура `Node` выглядит следующим образом:

```
struct Node {
  int data;  // Данные в узле
  Node * l; // Левое поддерево
  Node * r; // Правое поддерево
};
```

Для обходов деревьев, представленных таким образом, необходимо использовать стек. По аналогии с предыдущим разделом для решения данной задачи может быть построен автомат, граф переходов которого представлен на рис. 10.

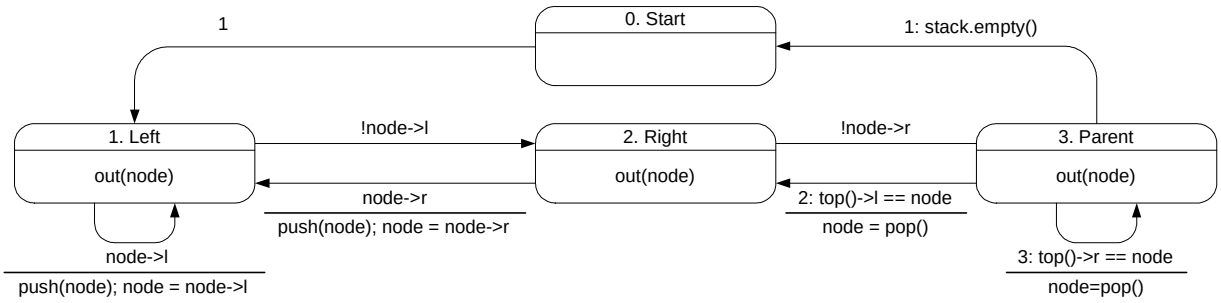

Рис. 10. Граф переходов автомата для реализации обхода двоичного дерева с использованием стека

На этом рисунке используются следующие обозначения: `push(node)` — помещение текущего узла в стек, `top()` — узел в вершине стека, `pop()` — функция удаления узла из стека, возвращающая удаленную вершину, `empty()` — проверка стека на пустоту.

Заметим, что при всех обходах дерева в стек помещается одна и та же информация. Таким образом, один стек может быть использован для реализации всех трех обходов, что позволяет экономить память.

Если вывод в состояниях *Left, Right* и *Parent* осуществлять в разные потоки, то можно получить одновременный вывод любой комбинации рассмотренных обходов.

Приведем реализацию этого автомата при помощи конструкции `switch`, построенную по графу переходов формально и изоморфно.

```
void traverseWithStack(Node const* node, int show) {
  std::stack<Node const*> stack;

  State state = START;
  do {
    switch (state) {
      case START:
        state = LEFT;
        break;
      case LEFT:
        if (show & SHOW_LEFT) cout << node->data <<  " ";
        if (node->l) {
          stack.push(node);
          node = node->l;
          state = LEFT;
        } else {
          state = RIGHT;
        }
        break;
      case RIGHT:
        if (show & SHOW_RIGHT) cout << node->data << " ";
        if (node->r) {
          stack.push(node);
          node = node->r;
          state = LEFT;
        } else {
          state = PARENT;
        }
        break;
```

```
        case PARENT:
          if (show & SHOW_PARENT) cout << node->data << "
          ";
          if (stack.empty()) {
            state = START;
          } else if (stack.top()->l == node) {
            node = stack.top(); stack.pop();
            state = RIGHT;
          } else if (stack.top()->r == node) {
            node = stack.top(); stack.pop();
            state = PARENT;
          }
          break;
      }
    } while (state != START);
  }
```

Граф переходов позволяет легко понять поведение программы, а также является тестом для ее проверки. Это объясняется тем, что состояния декомпозируют входные воздействия на группы, каждая из которых содержит условия, определяющая переходы только из этого состояния.

2.3.6. Обход *k*-ичного дерева без использования стека

Изложенный подход может быть обобщен на случай *k*-ичных деревьев [33].

В программе дерево будем представлять в следующем виде:

```
struct KNode {
  KNode * p;
  int data;
  KNode * c[k];
};
```

где k — константа, c — массив, хранящий указатели на детей. Метод, предложенный для обхода двоичных деревьев без использования стека, можно обобщить для обхода *k*-ичных деревьев (рис. 11).

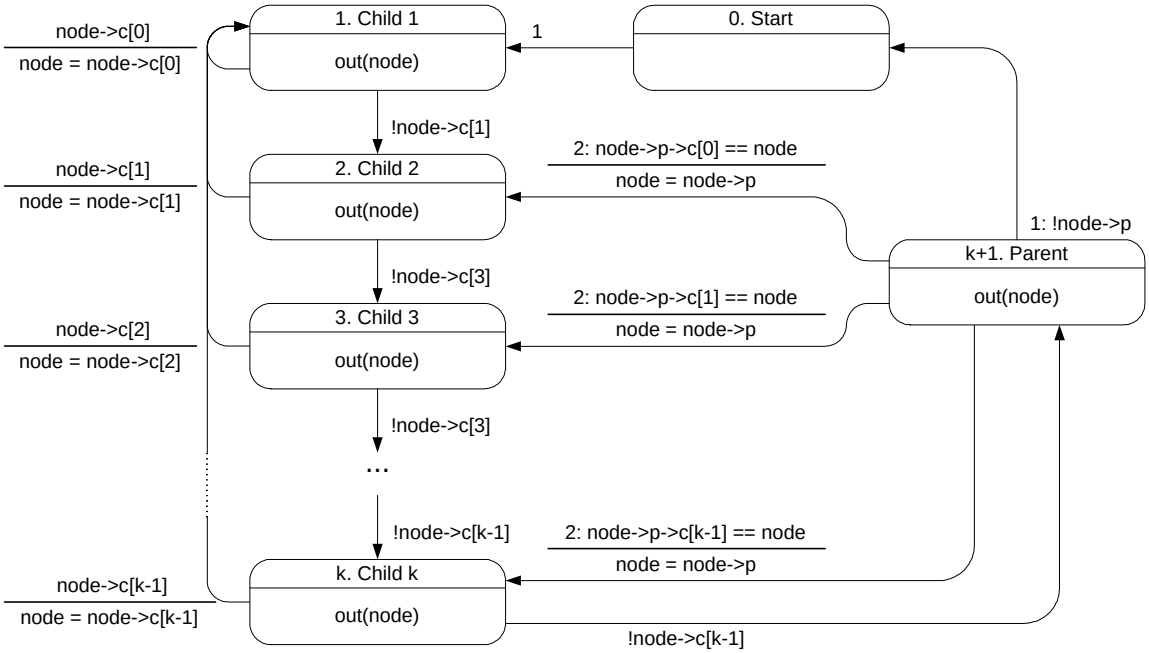

Рис. 11. Граф переходов автомата для реализации обхода

k-ичного дерева без использования стека

Отметим, что у построенного автомата $k+3$ состояний. Таким образом, число состояний автомата зависит от k.

Приведем реализацию этого автомата при помощи конструкции `switch`, построенную по графу переходов формально и изоморфно.

```
void traverseK ()
{
  enum State
  {
    Start,
    Child1,
    Child2,
    Child3,
    Parent
  };

  State state = Start;
  KNode * node = knodes;
```

```
do
{
  switch ( state )
  {
  case Start :
    state = Child1;
    break;
  case Child1 :
    cout << node -> data << ' ';
    if ( node -> c [ 0 ] != NULL )
      node = node -> c [ 0 ];
    else
      state = Child2;
    break;
  case Child2 :
    if ( node -> c [ 1 ] != NULL )
    {
      node = node -> c [ 1 ];
      state = Child1;
    }
    else
      state = Child3;
    break;
  case Child3 :
    if ( node -> c [ 2 ] != NULL )
    {
      node = node -> c [ 2 ];
      state = Child1;
    }
    else
      state = Parent;
    break;
  case Parent :
    if ( node == knodes )
      state = Start;
```

```
        else if ( node == node -> p -> c [ 0 ] )
        {
          state = Child2;
          node = node -> p;
        }
        else if ( node == node -> p -> c [ 1 ] )
        {
          state = Child3;
          node = node -> p;
        }
        else
          node = node -> p;
        break;
      }
    } while ( state != Start );
  }
```

Как отмечено выше, граф переходов построенного автомата зависит от числа *k*. Таким образом, такая реализация обхода *k*-ичного дерева не является универсальной.

Предложим универсальное решение. Из графа переходов следует, что условия перехода и действия в состояниях с первого по *k*-ое очень похожи. Поэтому логично объединить эти состояния в одно. В результате получается редуцированный автомат с четырьмя состояниями (рис. 12), который позволяет произвести обход *k*-ичного дерева при произвольном значении *k*.

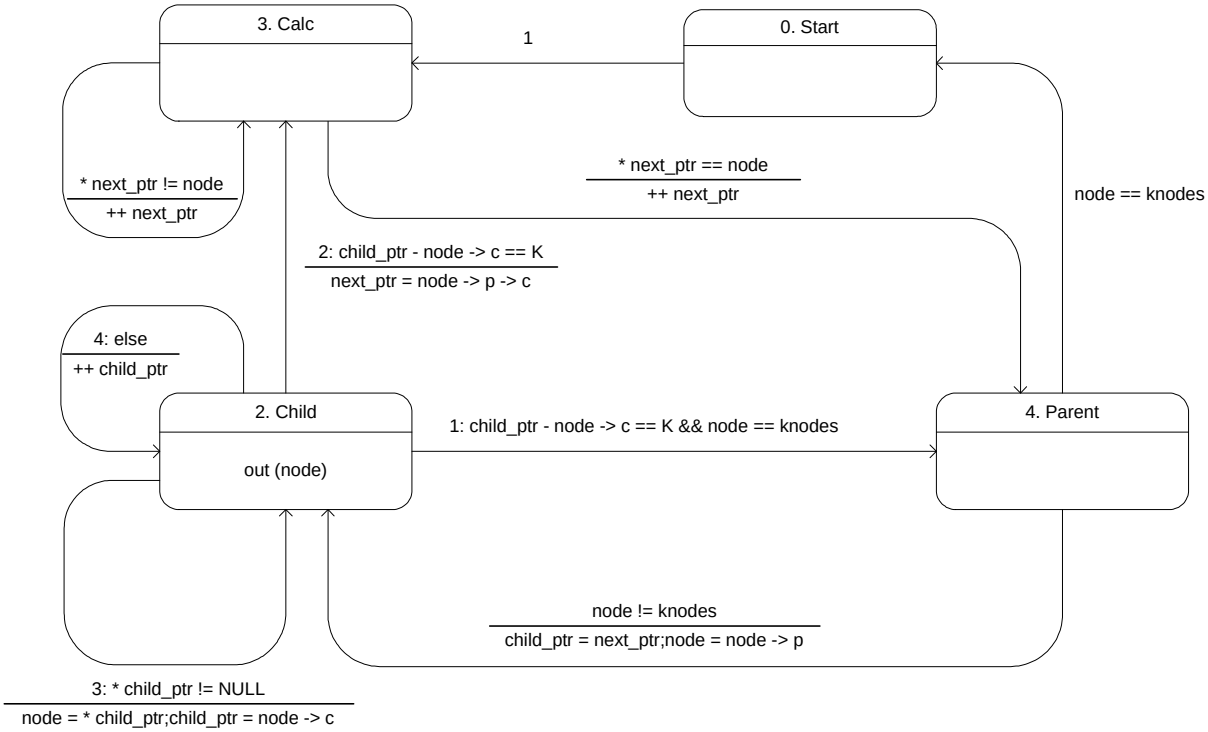

Рис. 12. Редуцированный граф переходов автомата для реализации обхода *k*-ичного дерева без использования стека

Приведем реализацию этого автомата при помощи конструкции `switch`, построенную по графу переходов формально и изоморфно.

```
void traverseKReduced ()
{
  enum State
  {
    Start,
    Child,
    Parent,
    Calc
  };

  State state = Start;
  KNode * node;
```

```
KNode * * child_ptr;
KNode * * next_ptr;

do
{
  switch ( state )
  {
  case Start :
    state = Child;
    node  = knodes;
    child_ptr = node -> c;
    break;
  case Child:
    if ( child_ptr == node -> c )
      cout << node -> data << ' ';
    if ( child_ptr - node -> c == K && node == knodes )
    {
      state = Parent;
    }
    else if ( child_ptr - node -> c == K )
    {
      next_ptr = node -> p -> c;
      state = Calc;
    }
    else if ( * child_ptr != NULL )
    {
      node = * child_ptr;
      child_ptr = node -> c;
    }
    else
    {
      ++ child_ptr;
    }
    break;
  case Calc :
```

```
        if ( * next_ptr == node )
        {
          ++ next_ptr;
          state = Parent;
        }
        else
          ++ next_ptr;
        break;
      case Parent :
        if ( node != knodes )
        {
          child_ptr = next_ptr;
          node = node -> p;
          state = Child;
        }
        else
          state = Start;
        break;
      }
    } while ( state != Start );
  }
```

2.4. Реализация рекурсивных алгоритмов на основе автоматного подхода

2.4.1. Введение

В работе [35] предложен 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`.

В работах [18, 19] приведены примеры преобразований рекурсивных программ в итеративные, однако эти преобразования выполнялись неформально, в связи с отсутствием соответствующего метода.

В настоящей работе такой метод предлагается. Он состоит в преобразовании заданной программы с явной рекурсией в итеративную программу, построенную с использованием автомата Мили. Такие программы, как и в работе [35], будем называть автоматными. Метод иллюстрируется примерами преобразований классических рекурсивных программ, которые приведены в порядке их усложнения (факториал, числа Фибоначчи, ханойские башни, задача о ранце).

2.4.2. Изложение метода

Идея метода состоит в моделировании работы рекурсивной программы автоматной программой, явно (также как и в работах [18, 19]) использующей стек. Отметим, что явное выделение стека по сравнению со "скрытым" его применением в рекурсии, позволяет программно задавать его размер, следить за его содержимым и добавлять отладочный код в функции, реализующие операции над стеком.

Перейдем к изложению метода.

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 работе [35], при преобразовании 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.

2.4.3. Факториал

Одной из простейших задач, использующих рекурсию, является вычисление факториала. Построим автоматную программу по рекурсивной программе, в которой рекурсивный вызов выделен отдельным оператором.

```
#include <stdio.h>

int    f = 0 ;

void fact( int n )
{
  if( n <= 1 )
    f = 1 ;
  else
```

```
  {
    fact( n-1 ) ;
    f = n * f ;
  }
}

int main()
{
  int input = 7 ;
  fact( input ) ;
  printf( "\nФакториал(%d) = %d\n", input, f ) ;
  return 0 ;
}
```

Построим схему этой программы (рис. 13), а по ней – граф переходов автомата Мили (рис. 14).

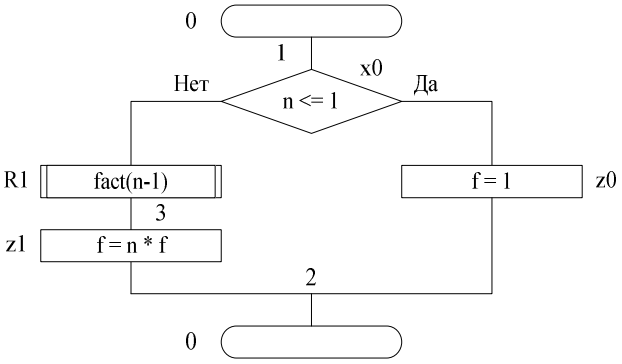

Рис. 13. Схема программы вычисления факториала

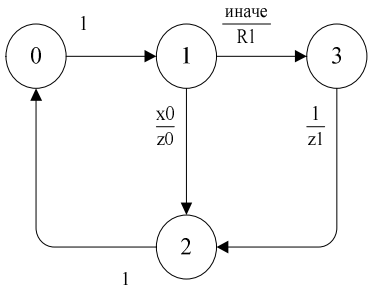

Рис. 14. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. При этом дуга 1–3 направляется в вершину с номером 1 и превращается в петлю. Условие перехода на этой петле остается неизменным, а рекурсивный вызов заменяется на операцию push и действие (n--), совершаемое над параметром рекурсивной функции при ее вызове. Условие перехода на дуге 2–0 заменяется на условие «стек пуст» и

добавляется дуга 2–3, при переходе по которой выполняется действие pop (рис. 15).

Так как исходная программа содержит только одну рекурсию, запоминать значение переменной состояния автомата в данном случае нет необходимости. Поэтому операции push и pop будут сохранять и восстанавливать только значение локальной переменной n.

Упростим этот граф за счет исключения неустойчивой вершины с номером 3 (рис. 16).

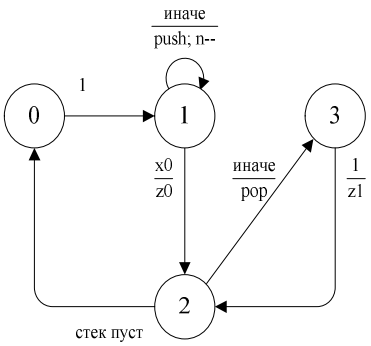

Рис. 15. Преобразованный граф переходов

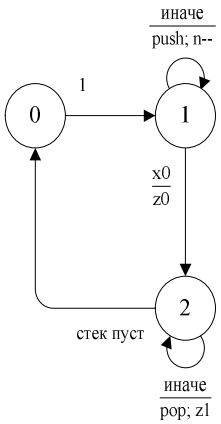

Рис. 16. Упрощенный граф переходов

Приведем автоматную программу, построенную по упрощенному графу переходов.

```
#include <stdio.h>

typedef struct
{
  int n ;
} stack_t ;

stack_t stack[200] ;
```

```
int     top = -1 ;

push( int n )
{
  top++ ;
  stack[top].n = n ;
  printf( "push {%d}: ", n ) ; show_stack() ;
}

pop( int *n )
{
  printf( "pop  {%d}: ", stack[top].n ) ;
  *n = stack[top].n ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;
  for( i = top ; i >= 0 ; i-- )
    printf( "{%d} ", stack[i].n ) ;
  printf( "\n" ) ;
}

int    f = 0 ;

void fact( int n )
{
  int y = 0, y_old ;

  do
  {
```

```
    y_old = y ;

    switch( y )
    {
      case 0:
                                                  y = 1 ;
      break ;

      case 1:
        if( n <= 1 ) { f = 1 ;   y = 2 ; }
        else
        { push( n ) ; n-- ; }
      break ;

      case 2:
        if( stack_empty() )              y = 0 ;
        else
        { pop( &n ) ; f = n * f ; }
      break ;
    }

    if( y_old != y ) printf( "Переход в состояние %d\n",
        y ) ;
  } while( y != 0 ) ;
}



int main()
{
  int input = 7 ;
  fact( input ) ;
  printf( "\nФакториал(%d) = %d\n", input, f ) ;
  return 0 ;
}
```

Приведем протокол, отражающий работу со стеком при вычислении факториала.

```
Переход в состояние 1
push {7}: {7}
push {6}: {6} {7}
push {5}: {5} {6} {7}
push {4}: {4} {5} {6} {7}
push {3}: {3} {4} {5} {6} {7}
push {2}: {2} {3} {4} {5} {6} {7}
Переход в состояние 2
pop  {2}: {3} {4} {5} {6} {7}
pop  {3}: {4} {5} {6} {7}
pop  {4}: {5} {6} {7}
pop  {5}: {6} {7}
pop  {6}: {7}
pop  {7}:
Переход в состояние 0

Факториал(7) = 5040
```

2.4.4. Числа Фибоначчи

Более сложной задачей является вычисление *чисел Фибоначчи*, программа для решения которой содержит две рекурсии, выделенные в виде отдельных операторов.

```
#include <stdio.h>

int res ;

void fibo( int n )
{
  int f ;
```

```
  if( n <= 1 )
    res = n ;
  else
  {
    fibo( n-1 ) ;
    f = res ;
    fibo( n-2 ) ;
    res += f ;
  }
}

int main()
{
  int input = 30 ;
  fibo( input ) ;
  printf( "\nФибоначи(%d) = %d\n", input, res ) ;
  return 0 ;
}
```

Отметим, что приведенная программа весьма неэффективна ввиду повторного вычисления одних и тех же значений. Эта программа может быть усовершенствована с помощью динамического программирования [16]. Построим автоматную программу по приведенной выше рекурсивной программе.

Построим схему этой программы (рис. 17), а по ней – граф переходов автомата Мили (рис. 18).

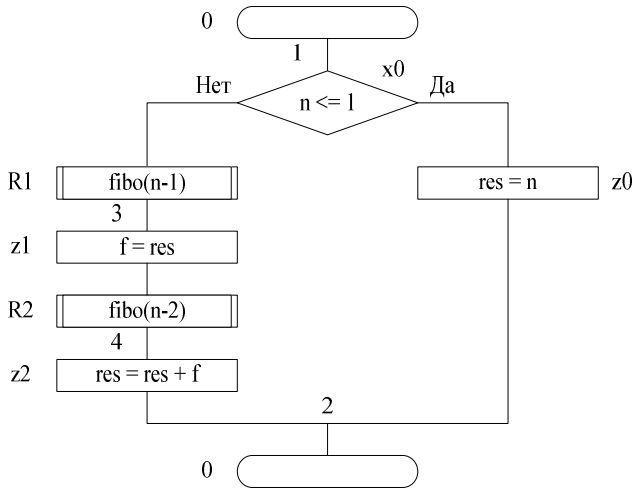

Рис. 17. Схема программы вычисления чисел Фибоначчи

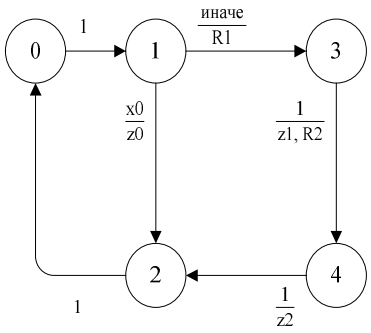

Рис. 18. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. При этом дуга `1—3` направляется в вершину с номером `1`, а рекурсивный вызов `R1` заменяется на операцию `push(3)` и действие (`n--`), выполняемое над параметром рекурсивной функции при ее вызове. Аналогичным образом корректируется дуга `3—4`, содержащая рекурсивный вызов `R2`. Эта дуга направляется в вершину с номером `1`, а рекурсивный вызов заменяется на операцию `push(4)` и действие (`n = n-2`), выполняемое над параметром рекурсивной функции при ее вызове. Безусловный переход на дуге `2—0` заменяется на условие «стек пуст», и ему присваивается первый приоритет. Добавляются дуги `2-3` и `2-4`, переход по которым зависит от значения переменной состояния автомата, запомненного в верхнем элементе стека. Если это значение равно «3», осуществляется переход по дуге `2-3`, а если это значение равно «4» — то по дуге `2-4`. При переходе по этим дугам выполняется действие `pop` (рис. 18).

Дуги `2—3` и `2—4` соответствуют возврату из рекурсивной функции. Так как исходная программа содержит две рекурсии, необходимо запоминать в

стеке не только локальные переменные n и f, но и значение переменной состояния автомата y.

Упростим этот граф за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 20).

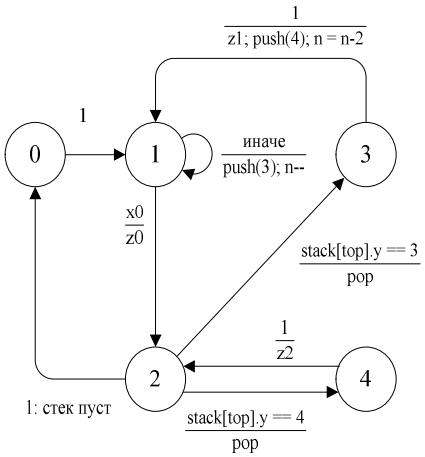

Рис. 19. Преобразованный граф переходов

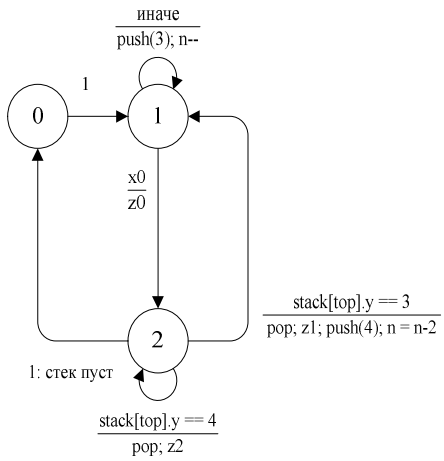

Рис. 20. Упрощенный граф переходов

Приведем автоматную программу, построенную по упрощенному графу переходов.

```
#include <stdio.h>

typedef struct
{
  int y, n, f ;
} stack_t;

stack_t stack[200] ;
int top = -1;

push( int y, int n, int f )
{
  top++ ;
```

```
  stack[top].y = y;
  stack[top].n = n;
  stack[top].f = f;
  printf( "push {%d,%d,%d}: ", y, n, f ) ; show_stack() ;
}

pop( int *y, int *n, int *f )
{
  printf( "pop  {%d,%d,%d}: ", stack[top].y,
        stack[top].n, stack[top].f ) ;
  if( y ) *y = stack[top].y;
  *n = stack[top].n;
  *f = stack[top].f;
  top--;
  show_stack();
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;

  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d} ", stack[i].y, stack[i].n,
        stack[i].f ) ;
  printf( "\n" ) ;
}

int res = 0 ;

void fibo( int n )
{
  int f ;
  int y = 0, y_old ;
```

```
do
{
  y_old = y ;

  switch( y )
  {
    case 0:
                                          y = 1 ;
    break ;

    case 1:
      if( n <= 1 ) { res = n ;              y = 2 ; }
      else
      { push( 3, n, f ) ; n-- ; }
    break ;

    case 2:
      if( stack_empty() )                   y = 0 ;
      else
      if( stack[top].y == 3 )
      {
        pop( NULL, &n, &f ) ;
        f = res ;
        push( 4, n, f ) ; n = n - 2 ;
                                          y = 1;
      }
      else
      if( stack[top].y == 4 )
      {
        pop( NULL, &n, &f ) ;
        res = res + f;
      }
    break ;
  }
```

```
    if( y_old != y ) printf( "Переход в состояние %d\n",
        y);
  } while( y != 0 ) ;
}

int main()
{
  int input = 4 ;
  fibo( input ) ;
  printf( "\nФибоначи(%d) = %d\n", input, res ) ;
  return 0 ;
}
```

Приведем протокол, отражающий работу со стеком.

```
Переход в состояние 1
push {3,4,0}: {3,4,0}
push {3,3,0}: {3,3,0} {3,4,0}
push {3,2,0}: {3,2,0} {3,3,0} {3,4,0}
Переход в состояние 2
pop  {3,2,0}: {3,3,0} {3,4,0}
push {4,2,1}: {4,2,1} {3,3,0} {3,4,0}
Переход в состояние 1
Переход в состояние 2
pop  {4,2,1}: {3,3,0} {3,4,0}
pop  {3,3,0}: {3,4,0}
push {4,3,1}: {4,3,1} {3,4,0}
Переход в состояние 1
Переход в состояние 2
pop  {4,3,1}: {3,4,0}
pop  {3,4,0}:
push {4,4,2}: {4,4,2}
Переход в состояние 1
push {3,2,2}: {3,2,2} {4,4,2}
Переход в состояние 2
pop  {3,2,2}: {4,4,2}
```

```
push {4,2,1}: {4,2,1} {4,4,2}
Переход в состояние 1
Переход в состояние 2
pop  {4,2,1}: {4,4,2}
pop  {4,4,2}:
Переход в состояние 0

Фибоначи(4) = 3
```

2.4.5. Задача о ханойских башнях

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях, описанная выше.

Эта задача является более сложной, чем приведенные выше, так как рассматриваемый вариант ее решения содержит три рекурсии. В приведенной программе содержатся вызовы функций протоколирования ее работы с отображением глубины рекурсии.

```
#include <stdio.h>

void hanoy( int i, int j, int k, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "    " ) ;
  printf( "hanoy(%d,%d,%d)\n", i, j, k ) ;

  if( k == 1 )
    move( i, j, d ) ;
  else
  {
    hanoy( i, 6-i-j, k-1, d+1 ) ;
    hanoy( i, j, 1, d+1 ) ;
    hanoy( 6-i-j, j, k-1, d+1 ) ;
  }
}
```

```
void move( int i, int j, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "    " ) ;
  printf( "move(%d,%d)\n", i, j ) ;
}

int main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input, 0 ) ;
  return 0 ;
}
```

Построим схему этой программы (рис. 21), а по ней – граф переходов автомата Мили (рис. 22).

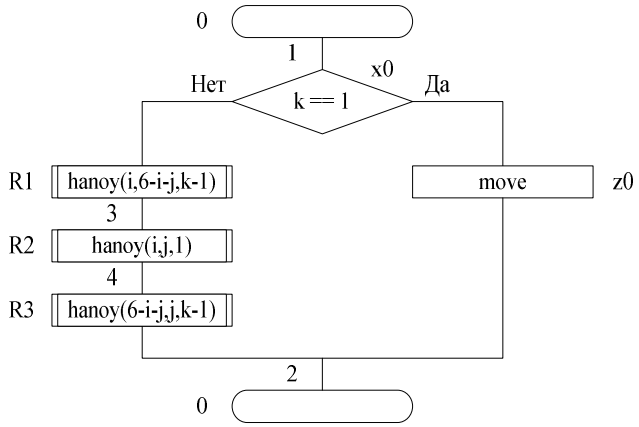

Рис. 21. Схема программы решения задачи о ханойских башнях

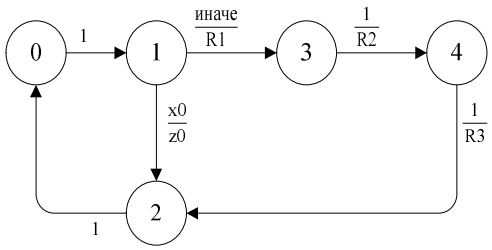

Рис. 22. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. Дуги, содержащие рекурсивные вызовы, направляются в вершину с номером 1. При этом сами рекурсивные вызовы заменяются операцией push и действием, выполняемым над параметрами функции. На рис. 23 такие действия обозначены символом z с номером, соответствующим номеру рекурсивного вызова: `z1 (j=6-i-j ; k--)` для `R1`, `z2 (k=1)` для `R2` и `z3 (i=6-i-j; k--)` для R3. Остальные преобразования выполняются по аналогии с предыдущими примерами.

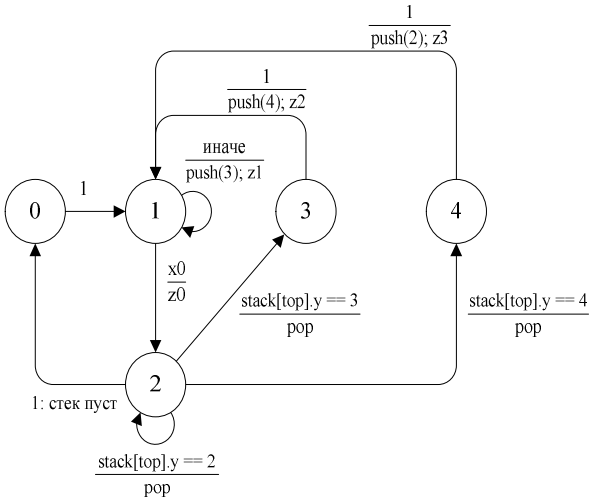

Рис. 23. Преобразованный граф переходов

Приведем программу, построенную по графу переходов автомата Мили с пятью состояниями (рис. 23):

```
#include <stdio.h>

typedef struct
{
  int y, i, j, k ;
} stack_t ;

stack_t stack[200] ;
int     top = -1 ;

push( int y, int i, int j, int k )
{
  top++ ;
  stack[top].y = y ;
  stack[top].i = i ;
  stack[top].j = j ;
  stack[top].k = k ;
  printf( "push {%d,%d,%d,%d}: ", y, i, j, k ) ;
        show_stack() ;
}
```

```
pop( int *y, int *i, int *j, int *k )
{
  printf( "pop  {%d,%d,%d,%d}: ", stack[top].y,
        stack[top].i,

          stack[top].j, stack[top].k ) ;
  if( y ) *y = stack[top].y ;


  *i = stack[top].i ;
  *j = stack[top].j ;
  *k = stack[top].k ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;
  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d,%d} ", stack[i].y, stack[i].i,
        stack[i].j, stack[i].k ) ;
  printf( "\n" ) ;
}

void hanoy( int i, int j, int k )
{
  int y = 0, y_old ;

  do
  {
    y_old = y ;
```

```
switch( y )
{
  case 0:

      y = 1 ;
  break ;

  case 1:
    if( k == 1 ) { move( i, j ) ;     y = 2 ; }
    else
    {
      push( 3, i, j, k ) ;
      j = 6 - i - j ; k-- ;
    }
  break ;

  case 2:
    if( stack_empty() )                              y
    = 0 ;
    else
      pop( &y, &i, &j, &k ) ;
    /*
    if( stack[top].y == 4 )
    {
      pop( NULL, &i, &j, &k ) ;       y = 4 ;


    }
    else
    if( stack[top].y == 3 )
    {
      pop( NULL, &i, &j, &k ) ;       y = 3 ;
    }
    else
    if( stack[top].y == 2 )
```

```
        { pop( NULL, &i, &j, &k ) ; }
        */
      break ;

      case 3:
        push( 4, i, j, k ) ;
        k = 1 ;
          y = 1 ;
      break ;

      case 4:
        push( 2, i, j, k ) ;
        i = 6 - i - j ; k-- ;                          y = 1
        ;
      break ;
    }

    if( y_old != y ) printf( "Переход в состояние %d\n",
        y ) ;
  }
  while( y != 0 ) ;
}

void move( i, j )
{
  printf( "%d -> %d\n", i, j ) ;
}

int main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
  return 0 ;
}
```

В этой программе операторы, реализующие дуги, исходящие из вершины с номером 2 (за исключением дуги 2—0), закомментированы и заменены одним оператором pop(&y, &i, &j, &k). Такое упрощение программы, однако, приводит к тому, что по указанному оператору невозможно определить, в какое состояние будет выполнен переход.

Упростим граф переходов, приведенный на рис. 23, за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 24).

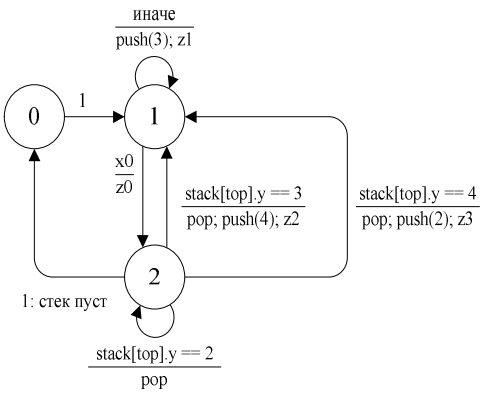

Рис. 24. Упрощенный граф переходов

При построении программы по графу переходов автомата Мили с тремя состояниями (рис. 24), функция hanoy() реализуется следующим образом:

```
void hanoy( int i, int j, int k )
{
  int y = 0, y_old ;

  do
  {
    y_old = y ;

    switch( y )
    {
```

```
case 0:

     y = 1 ;
break ;

case 1:
  if( k == 1 ) { move( i, j ) ;     y = 2 ; }
  else
  {
    push( 3, i, j, k ) ;
    j = 6 - i - j ; k-- ;
  }
break ;

case 2:
  if( stack_empty() )                    y = 0 ;
  else
  if( stack[top].y == 4 )
  {
    pop( NULL, &i, &j, &k ) ;
    push( 2, i, j, k ) ;
    i = 6 - i - j ; k-- ;                y = 1 ;
  }
  else
  if( stack[top].y == 3 )
  {
    pop( NULL, &i, &j, &k ) ;
    push( 4, i, j, k ) ;
    k = 1 ;
    y = 1 ;
  }
  else
  if( stack[top].y == 2 )
  { pop( NULL, &i, &j, &k ) ; }
break ;
```

```
    }

    if( y_old != y ) printf( "Переход в состояние %d\n",
        y ) ;
  }
  while( y != 0 ) ;
}
```

Остальная часть программы остается неизменной по сравнению с предыдущей. Отметим, что при такой реализации не удается выполнить ее дальнейшее упрощение за счет замены трех условий переходов одной функцией pop.

Приведем протокол, отражающий работу со стеком для программы, построенной по упрощенному графу переходов.

```
Ханой с 3 дисками:
Переход в состояние 1
push {3,1,3,3}: {3,1,3,3}
push {3,1,2,2}: {3,1,2,2} {3,1,3,3}
1 -> 3
Переход в состояние 2
pop  {3,1,2,2}: {3,1,3,3}
push {4,1,2,2}: {4,1,2,2} {3,1,3,3}
Переход в состояние 1
1 -> 2
Переход в состояние 2
pop  {4,1,2,2}: {3,1,3,3}
push {2,1,2,2}: {2,1,2,2} {3,1,3,3}
Переход в состояние 1
3 -> 2
Переход в состояние 2
pop  {2,1,2,2}: {3,1,3,3}
pop  {3,1,3,3}:
push {4,1,3,3}: {4,1,3,3}
Переход в состояние 1
1 -> 3
```

```
Переход в состояние 2
pop  {4,1,3,3}:
push {2,1,3,3}: {2,1,3,3}
Переход в состояние 1
push {3,2,3,2}: {3,2,3,2} {2,1,3,3}
2 -> 1
Переход в состояние 2
pop  {3,2,3,2}: {2,1,3,3}
push {4,2,3,2}: {4,2,3,2} {2,1,3,3}
Переход в состояние 1
2 -> 3
Переход в состояние 2
pop  {4,2,3,2}: {2,1,3,3}
push {2,2,3,2}: {2,2,3,2} {2,1,3,3}
Переход в состояние 1
1 -> 3
Переход в состояние 2
pop  {2,2,3,2}: {2,1,3,3}
pop  {2,1,3,3}:
Переход в состояние 0
```

2.4.6. Задача о ранце

Задача о ранце может быть сформулирована следующим образом. Имеется N типов предметов, для каждого из которых известны его объем и цена, причем количество предметов каждого типа не ограничено. Необходимо уложить предметы в ранец объема M таким образом, чтобы стоимость его содержимого была максимальна.

Как и в случае задачи о вычислении чисел Фибоначчи, непосредственное рекурсивное решение этой задачи является крайне неэффективным из-за повторного решения одних и тех же подзадач [16]. Этот недостаток может быть устранен путем применения динамического программирования. При этом промежуточные результаты решения задачи запоминаются и используются в дальнейшем.

Несмотря на то, что решение задачи содержит только одну рекурсию, эта задача рассматривается в настоящей работе для иллюстрации предлагаемого метода, так как соответствующая ей программа, построенная на основе программы, предложенной в работе [16], отличается весьма сложной логикой по сравнению с приведенными выше.

```
#include <stdio.h>

typedef struct
{
  int size, val ;
} item_t ;

item_t  items[] = { 3,4, 4,5, 7,10, 8,11, 9,13 } ; //
        Типы предметов.
int N = sizeof(items)/sizeof(item_t) ;// Количество типов
        предметов.
enum    { knap_cap = 17 } ;          // Объем ранца.
int    maxKnown[knap_cap+1];
item_t  itemKnown[knap_cap+1];
int    knap_ret = 0;

void knap( int cap )
{
  int i, space, max, maxi = 0, t;

  if( maxKnown[cap] != -1 )
  {
    knap_ret = maxKnown[cap];
    return ;
  }

  for( i = 0, max = 0 ; i < N ; i++ )
  {
    space = cap - items[i].size ;
```

```
    if( space >= 0 )
    {
      knap( space ) ;
      t = knap_ret + items[i].val;
      if( t > max )
      {
        max = t;
        maxi = i;
      }
    }
  }

  maxKnown[cap] = max ; itemKnown[cap] = items[maxi];
  knap_ret = max;
}

void show_knap( int cap, int value )
{
  int i ;
  int total_size = 0 ;

  printf( "\nВиды предметов {объем, цена} (%d):\n", N );
  for ( i = 0 ; i < N ; i++ )
    printf("{%d,%d} ", items[i].size, items[i].val);

  printf("\n\nОбъем ранца: %d.\n", cap);
  printf("Содержимое ранца:\n");

  i = cap;
  while(i > 0 && i - itemKnown[i].size >= 0)
  {
    printf( "{%d,%d} ", itemKnown[i].size,
        itemKnown[i].val );
    total_size += itemKnown[i].size;
    i -= itemKnown[i].size;
```

```
  }
  printf( "\nЗанятый объем: %d. Ценность: %d\n",
        total_size, value );
}

void init()
{
  int i ;

  for ( i = 0 ; i < knap_cap+1 ; i++ )
    maxKnown[i] = -1 ;
}

void main()
{
  init() ;
  knap( knap_cap ) ;
  show_knap( knap_cap, knap_ret ) ;
}
```

В приведенной программе массив `maxKnown` используется для запоминания результатов решения подзадач, а массив `itemKnown` применяется в функции `show_knap()` для определения содержимого ранца, полученного в результате решения задачи.

Построим схему этой программы (рис. 25).

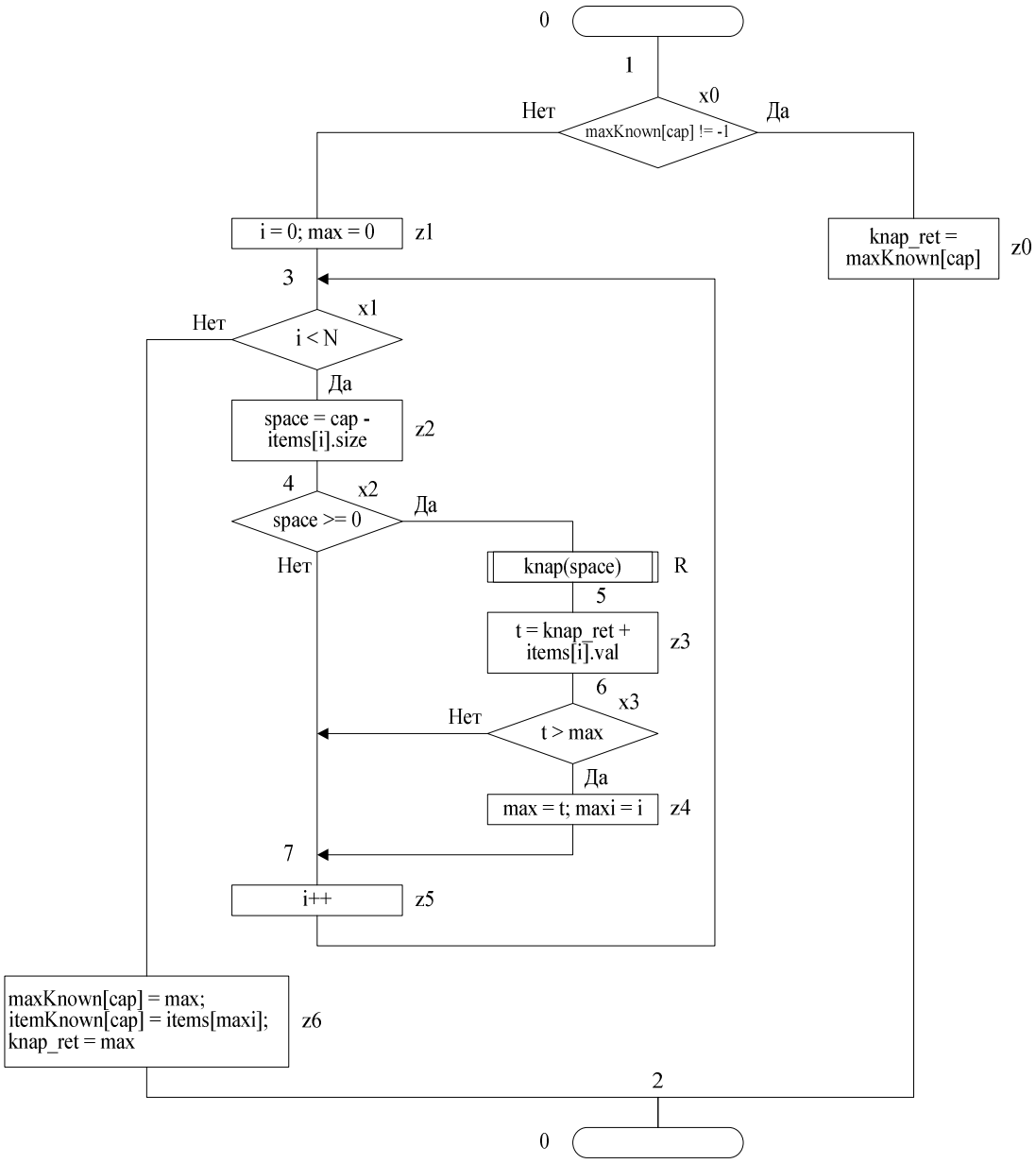

Рис. 25. Схема программы решения задачи о ранце

По схеме программы построим граф переходов автомата Мили (рис. 26).

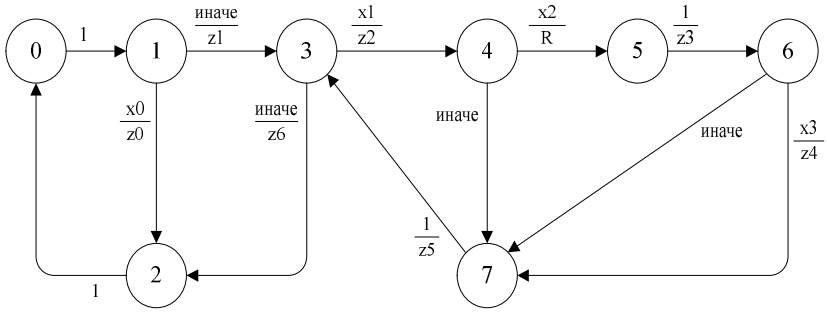

Рис. 26. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. При этом дуга `4–5` направляется в вершину с номером `1`, а выполняемый при переходе по этой дуге рекурсивный вызов заменяется на операцию `push` и действие (`cap = space`), выполняемое над параметром рекурсивной функции при ее вызове. Безусловный переход на дуге `2–0` заменяется на условие «*стек пуст*». Добавляется дуга `2–5`, при переходе по которой выполняется действие pop (рис. 27).

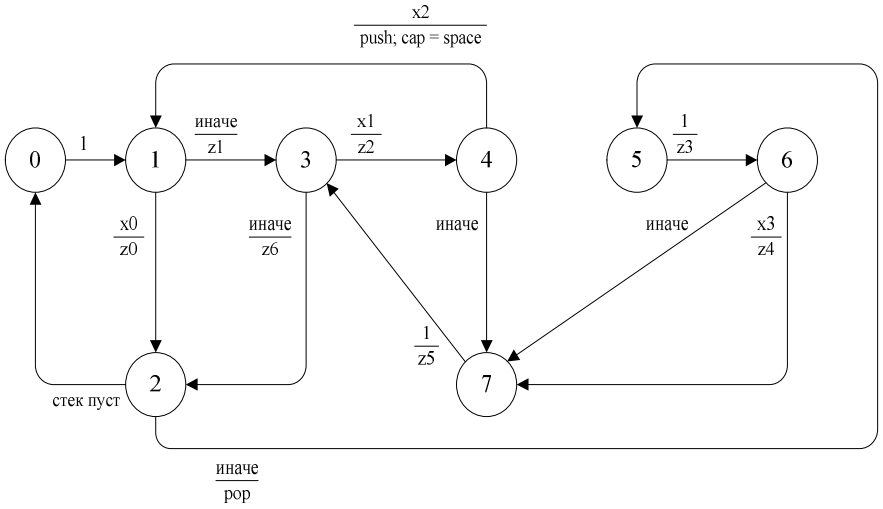

Рис. 27. Преобразованный граф переходов

Упростим этот граф за счет исключения неустойчивой вершины с номером 5 (рис. 28).

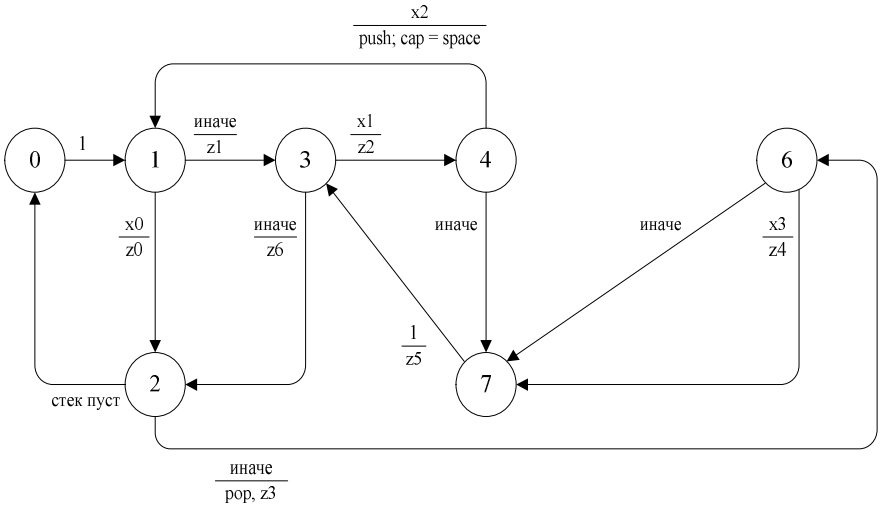

Рис. 28. Упрощенный граф переходов

Приведем автоматную программу, построенную по графу переходов автомата Мили с семью состояниями.

```
#include <stdio.h>

typedef struct
{
  int cap, i, space, max, maxi, t ;
} stack_t ;

stack_t stack[200] ;
int     top = -1 ;

push( int cap, int i, int space, int max, int maxi, int t
        )
{
  top++ ;
  stack[top].cap = cap ;
  stack[top].i = i ;
  stack[top].space = space ;
  stack[top].max = max ;
```

```
  stack[top].maxi = maxi ;
  stack[top].t = t ;
  printf( "push {%d,%d,%d,%d,%d,%d}: ", cap, i,
          space, max, maxi, t ) ;
  show_stack() ;
}

pop( int *cap, int *i, int *space, int *max, int *maxi,
        int *t )
{
  printf( "pop  {%d,%d,%d,%d,%d,%d}: ", stack[top].cap,
        stack[top].i,
          stack[top].space, stack[top].max,
        stack[top].maxi, stack[top].t ) ;
  *cap = stack[top].cap ;
  *i = stack[top].i ;
  *space = stack[top].space ;
  *max = stack[top].max ;
  *maxi = stack[top].maxi ;
  *t = stack[top].t ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;

  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d,%d,%d,%d} ", stack[i].cap,
        stack[i].i,
            stack[i].space, stack[i].max, stack[i].maxi,
        stack[i].t ) ;
```

```
  printf( "\n" ) ;
}

typedef struct
{
  int size, val ;
} item_t ;

item_t  items[] = { 3,4, 4,5, 7,10, 8,11, 9,13 } ;
int    N = sizeof(items)/sizeof(item_t) ;
enum    { knap_cap = 17 } ;
int    maxKnown[knap_cap+1] ;
item_t  itemKnown[knap_cap+1] ;
int    knap_ret = 0 ;

void knap( int cap )
{
  int y = 0, y_old ;
  int i, space, max, maxi = 0, t ;

  do
  {
    y_old = y ;

    switch( y )
    {
      case 0:
                                      y = 1 ;
      break ;

      case 1:
        if( maxKnown[cap] != -1 )
        { knap_ret = maxKnown[cap] ;     y = 2 ; }
        else
        { i = 0 ; max = 0 ;              y = 3 ; }
```

```
break ;

case 2:
  if( stack_empty() )              y = 0 ;
  else
  {
    pop( &cap, &i, &space, &max, &maxi, &t ) ;
    t = knap_ret + items[i].val ;   y = 6 ;
  }
break ;

case 3:
  if( i < N )
  { space = cap - items[i].size ;   y = 4 ; }
  else
  {
    maxKnown[cap] = max ;
    itemKnown[cap] = items[maxi] ;
    knap_ret = max ;           y = 2 ;
  }
break ;

case 4:
  if( space >= 0 )
  {
    push( cap, i, space, max, maxi, t ) ;
    cap = space ;
                               y = 1 ;
  }
  else
                               y = 7 ;
break ;

case 6:
```

```
        //if( t >= max ) // Фиксируется последнее из
        оптимальных решений.
        if( t > max ) // Фиксируется первое из
        оптимальных решений.
        { max = t ; maxi = i ;           y = 7 ; }
        else
                                    y = 7 ;
      break ;

      case 7:
        i++ ;                       y = 3 ;
      break ;
    } ;

    if( y_old != y ) printf( "Переход в состояние %d\n",
        y ) ;
  } while( y != 0 ) ;
}

void show_knap( int cap, int value )
{
  int i ;
  int total_size = 0 ;

  printf( "\nВиды предметов {объем, цена} (%d):\n", N ) ;
  for ( i = 0 ; i < N ; i++ )
    printf( "{%d,%d} ", items[i].size, items[i].val ) ;

  printf( "\n\nОбъем ранца: %d.\n", cap ) ;
  printf( "Содержимое ранца:\n" ) ;

  i = cap ;
  while( i > 0 && i - itemKnown[i].size >= 0 )
  {
```

```
    printf( "{%d,%d} ", itemKnown[i].size,
        itemKnown[i].val ) ;
    total_size += itemKnown[i].size ;
    i -= itemKnown[i].size ;
  }
  printf( "\nЗанятый объем: %d. Ценность: %d\n",
        total_size, value ) ;
}

void init()
{
  int i ;

  for ( i = 0 ; i < knap_cap+1 ; i++ )
    maxKnown[i] = -1 ;
}

void main()
{
  init() ;

  knap(knap_cap) ;
  show_knap( knap_cap, knap_ret ) ;
}
```

Приведем результат работы этой программы, идентичный результату работы рекурсивной программы.

```
Виды предметов {объем, цена} (5):
{3,4} {4,5} {7,10} {8,11} {9,13}

Объем ранца: 17.
Содержимое ранца:
{3,4} {7,10} {7,10}
Занятый объем: 17. Ценность: 24
```

Выводы

Показано, что использование автоматного подхода позволило получить наглядные и, в отличие от классических, универсальные алгоритмы решения задач обхода деревьев, которые весьма экономны по памяти.

Итеративная, рекурсивная и автоматная программы позволяют получать одинаковые результаты, однако автоматная более понятна за счет явного выделения состояний.

В работе [36] было отмечено, что итеративные алгоритмы по вычислительной мощности эквивалентны рекурсивным. Однако, в известной авторам литературе, формальный метод преобразования рекурсивных алгоритмов в итеративные отсутствует. Как отмечалось во введении, в работах [18, 19] приведены примеры такого преобразования, которые, однако, не являются формализованными. Настоящая работа устраняет указанный пробел. При этом развивается основанный на использовании конечных автоматов подход, предложенный в работе [10].

Отметим, что автоматы, которые строятся с помощью предлагаемого подхода, зависят от глубокой предыстории – в отличие от классических автоматов, переходы из одного из состояний (с номером 2) определяются ранее запомненным в стеке номером следующего состояния.

Глава 3. Паттерн проектирования *State Machine*

Наиболее известной реализацией объекта, изменяющего поведение в зависимости от состояния, является паттерн ***State*** [5]**.** В указанной работе данный паттерн недостаточно полно специфицирован, поэтому в разных источниках, например в работах [11, 12], он реализуется по-разному (в частности, громоздко и малопонятно). Поэтому многие программисты считают, что этот паттерн не предназначен для реализации автоматов. Другой недостаток паттерна ***State*** состоит в том, что разделение реализации состояний по различным классам приводит и к распределению логики переходов по ним, что усложняет понимание программы. При этом не обеспечивается независимость классов, реализующих состояния, друг от друга. Таким образом, создание иерархии классов состояний и их повторное использование затруднено. Несмотря на эти недостатки, паттерн ***State*** достаточно широко применяется в программировании, например, при реализации синхронизации с источником данных в библиотеке ***Java Data Objects*** (***JDO***) [37].

Заметим, что проблемы с паттерном ***State*** возникают не только при его описании, но даже в определении, в том числе, из-за неправильного перевода. Так в работе [38] приведено следующее определение: «шаблон ***State*** — состояние объекта контролирует его поведение», в то время как более правильным было бы следующее: «шаблон ***State*** — состояния объекта управляют его поведением». Эти определения имеют разный смысл, так как в английском языке слово ***control*** переводится как *управление*, а в русском языке управление — это действие на объект, а контроль — мониторинг и проверка выполняемых действий на соответствие заданному поведению. Не случайно часто говорят «система управления и контроля».

Кроме работы [5], паттерн ***State***, как отмечалось выше, описывается в работах [11, 12]. В них авторы рассматривают реализацию графов переходов

автоматов с помощью этого паттерна. В работе [11] код реализации графа переходов с двумя вершинами занимает более десяти страниц текста. Такая же странная ситуация и в работе [12]. Автор этой книги легко справился с описанием и примерами к сорока шести паттернам, и только для паттерна *State* ему понадобилась помощь рецензента, который предоставил ему пример. Однако приведенный в этом примере граф переходов с четырьмя вершинами реализован таким образом, будто бы автор недостаточно разобрался в паттерне.

В настоящей работе, предлагается новый паттерн, объединяющий достоинства реализации автоматов в *SWITCH-технологии* [4] (централизация логики переходов) и паттерна *State* (локализация кода, зависящего от состояния в отдельных классах).

Новый паттерн назван *State Machine*. Обратим внимание, что в работе [39] уже был предложен паттерн с аналогичным названием, предназначенный для программирования параллельных систем реального времени на языке *Ada 95*. Тем не менее, авторы выбрали именно это название, как наиболее точно отражающее суть предлагаемого паттерна.

Для обеспечения повторного использования классов состояний, входящих в паттерн, предложено применять механизм *событий*, которые используются состояниями для уведомления автомата о необходимости смены состояния. Это позволяет централизовать логику переходов автомата и устранить «осведомленность» классов состояний друг о друге. При этом реализация логики переходов может осуществляться различными способами, например с использованием таблицы переходов или оператора выбора (оператор `switch` в языке *C++*).

Более двадцати методов реализации объектов, изменяющих поведение в зависимости от состояния, рассмотрены в работе [40]. Паттерн *State Machine* мог бы продолжить этот список. Наиболее близким к новому паттерну является объединение паттернов *State* и *Observer*, рассмотренное в работе [41]. Однако, предложенный в этой работе подход достаточно сложен,

так как добавляет новый уровень абстракции — класс `ChangeManager`. В паттерне *State Machine* используется более простая модель событий, не привлекающая относительно тяжелую реализацию паттерна *Observer*. В работе [42] предложена реализация паттерна *State*, позволяющая создавать иерархии классов состояний. Зависимость между классами состояний снижается за счет того, что переход в новое состояние осуществляется по его имени. Такая реализация, тем не менее, не снимает семантической зависимости между классами состояний.

3.1. Описание паттерна

В названиях разделов будем придерживаться соглашений, введенных в работе [5].

3.1.1. 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* [5], однако, как будет показано ниже, область применимости последнего уже.

Отметим, что в этом определении имеются в виду так называемые *управляющие*, а не *вычислительные* состояния [17]. Их различие может быть проиллюстрировано на следующем примере. При создании вычислительной системы для банка имеет смысл выделить режимы нормальной работы и банкротства в разные управляющие состояния, так как в этих режимах поведение банка может существенно отличаться. В то же время, конкретные суммы денег и другие характеристики в банковском балансе будут представлять вычислительное состояние.

3.1.2. 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 невозможна. 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 (здесь и далее примеры приводятся 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 работах [3, 4, 43]. 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. 29. 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.

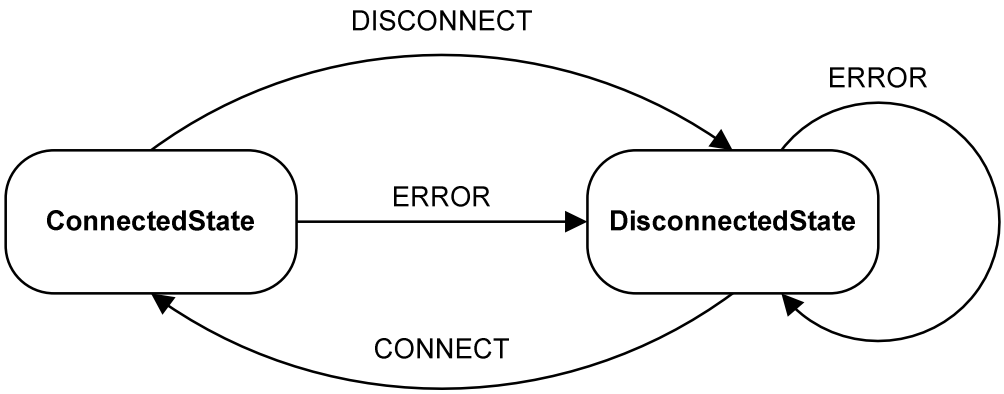

Рис. 29. Граф переходов для класса **`Connection`**

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.

3.1.3. 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 кода* [44], *зависящего от состояния объекта.* 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.* Пример такого использования будет рассмотрен в разд. 3.2.
4. *Эмуляции абстрактных и структурных автоматов.*

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

3.1.4. Структура

На рис. 30 изображена структура паттерна ***State Machine***.

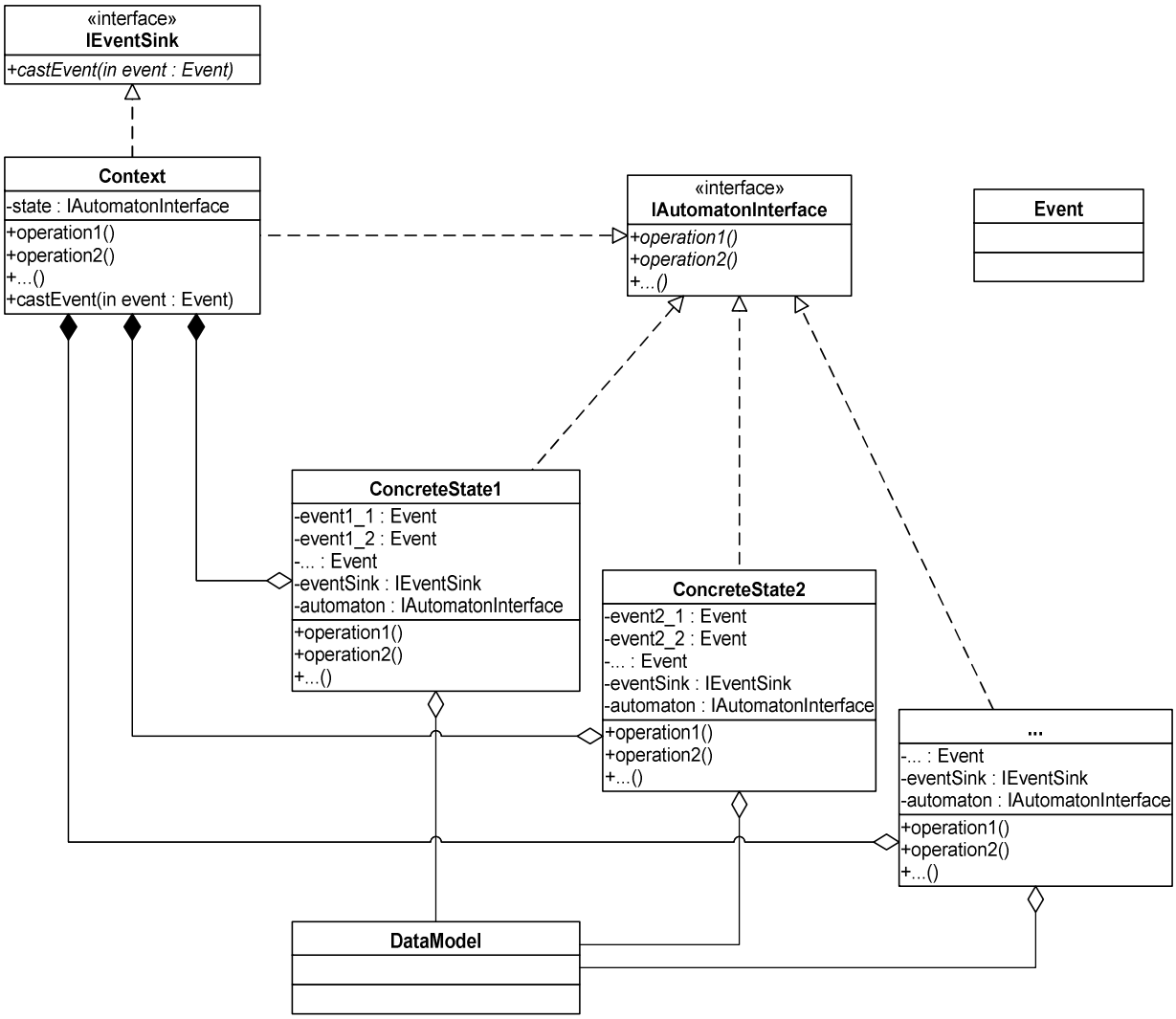

Рис. 30. Структура паттерна *State Machine*

Здесь `IAutomatonInterface` — интерфейс, реализуемого объекта, а `operation1`, `operation2`, … — его методы. 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.

3.1.5. 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.

3.1.6. Отношения

При инициализации контекст создает экземпляр модели данных и использует его при конструировании экземпляров состояний. Кроме модели данных, в конструктор класса состояния также передается интерфейс уведомления о событии (ссылка на контекст).

В процессе работы контекст делегирует вызовы методов интерфейса текущему экземпляру состояния. При исполнении делегированного метода объект, реализующий состояние, может сгенерировать событие — уведомить об этом контекст по интерфейсу уведомления о событиях.

Решение о смене состояния принимает контекст на основе события, пришедшего от конкретного объекта состояния.

3.1.7. 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. Логика переходов может быть реализована табличным способом, что повышает скорость смены состояний.
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. В отличие от паттерна ***State*** паттерн ***State Machine*** не содержит дублирующих интерфейсов для контекста и классов состояний.
7. Возможно повторное использование классов состояний, в том числе посредством создания их иерархии. Заметим, что в работе [5] сказано, что «поскольку зависящий от состояния код целиком находится в одном из подклассов класса `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 в работе [5] не описано наследование автоматов.

8. В работе [45] рассматривается задача реализации объектов, часть методов которых не зависит от состояния, для решения которой 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. 31).

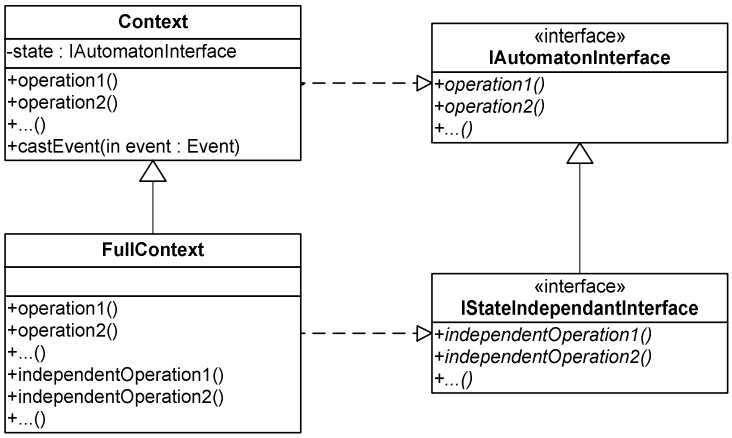

Рис. 31. Реализация методов, не зависящих от состояния

3.1.8. Реализация

Рассмотрим возможные модификации паттерна *State Machine*.

1. *Хранение модели данных*. Контекст можно реализовать таким образом, чтобы он включал в себя модель данных, как предлагается в паттерне *State*. Тогда в конструктор объекта состояния передается только один параметр. Однако при таком подходе зависимость реализации состояний от контекста увеличивается, что усложняет повторное использование классов состояний.
2. *Stateless и stateful классы состояний*. В паттерне *State Machine* контекст и классы состояний реализуют интерфейс автомата. Это достигается за счет того, что указанные классы содержат ссылки на модель данных и интерфейс уведомления о событиях. Таким образом, классы состояний являются *stateful* (зависят от предыстории). Такой подход не всегда приемлем, поскольку в некоторых ситуациях критичен расход памяти. Паттерн *State Machine* можно видоизменить так, чтобы состояния были *stateless* (не зависят от предыстории). При этом для классов состояний придется определить новый интерфейс, отличающийся от интерфейса автомата тем, что в каждый метод добавлены параметры, через которые передаются ссылки на модель данных и интерфейс уведомления о событиях. Это приводит к фактическому дублированию интерфейсов, что затрудняет модификацию кода и его повторное использование, но экономит память.
3. *Задание переходов*. Переходы между состояниями задаются в контексте. Это можно сделать, например, при помощи конструкции `switch`, как предлагается в *SWITCH-технологии* [4]. Переходы также можно задать таблицей, отображающей пару <*текущее состояние, событие*> в <*новое состояние*>. Инфраструктуру для реализации табличного подхода можно реализовать в базовом для всех контекстов классе.

4. *Протоколирование.* Вынесение логики переходов в контекст позволяет осуществлять протоколирование переходов автоматов в терминах состояний и событий.
5. *Создание модели данных*. Экземпляр модели данных может либо создаваться конструктором контекста (как описано выше), либо передаваться в параметрах конструктора контекста.

3.1.9. Пример кода

Коды всех примеров, описанных в статье, доступны по адресу [46].

В следующем примере приведен код на языке *Java*, реализующий класс *Соединение*, описанный в разд. 3.1.2. Это упрощенная модель произвольного удаленного соединения, через которое можно передавать данные.

Прежде всего, опишем интерфейсы и базовые классы, которые используются в данном примере. Эти классы вынесены в пакет `ru.ifmo.is.sm` (`sm` — сокращение от *State Machine*). Диаграмма классов этого пакета приведена на рис. 32.

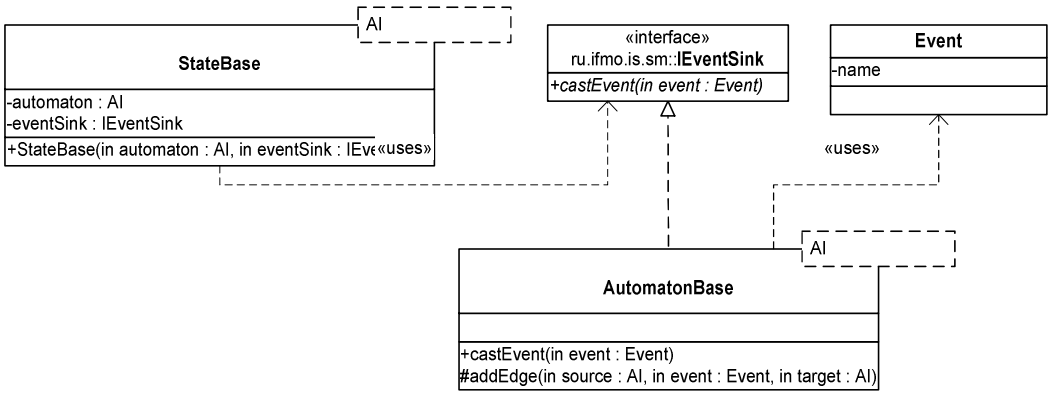

Рис. 32. Диаграмма классов пакета **`ru.ifmo.is.sm`**

Опишем классы и интерфейсы, входящие в него.

1. `IEventSink` — интерфейс уведомления о событии:

```
package ru.ifmo.is.sm;

public interface IEventSink {
  public void castEvent(Event event);
```

```
}
```

2. `Event` — класс события. Используется для уведомления контекста из классов состояний:

```
package ru.ifmo.is.sm;

public final class Event {
  private final String name;

  public Event(String name) {
    if (name == null) throw new
        NullPointerException();
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

3. `StateBase` — базовый класс для состояний. Для проверки типов во время компиляции применяются *параметры типа* (***generic, template***), появившееся в *Java 5.0*. В конструкторе запоминается интерфейс для приема событий:

```
package ru.ifmo.is.sm;

public abstract class StateBase<AI> {
  protected final AI automaton;
  protected final IEventSink eventSink;

  public StateBase(AI automaton, IEventSink
        eventSink) {
    if (automaton == null || eventSink == null) {
      throw new NullPointerException();
    }
    this.automaton = automaton;
    this.eventSink = eventSink;
  }

  protected void castEvent(Event event) {
    eventSink.castEvent(event);
  }
}
```

4. `AutomatonBase` — базовый класс для всех автоматов. Он предоставляет возможность наследнику регистрировать переходы,

используя метод addEdge. Дополнительно класс AutomatonBase реализует интерфейс уведомления о событии:

```
package ru.ifmo.is.sm;

import java.util.IdentityHashMap;
import java.util.Map;
import java.util.HashMap;

public abstract class AutomatonBase<AI>
        implements IEventSink {
  protected AI state;
  private final Map<AI, Map<Event, AI>> edges =
      new HashMap<AI, Map<Event, AI>>();

  protected void addEdge(AI source, Event event,
       AI target) {
    Map<Event, AI> row = edges.get(source);
    if (null == row) {
      row = new IdentityHashMap<Event, AI>();
      edges.put(source, row);
    }
    row.put(event, target);
  }

  public void castEvent(Event event) {
    try {
      state = edges.get(state).get(event);
    } catch (NullPointerException e) {
      throw new IllegalStateException("Edge is
        not defined");
    }
  }
}
```

Классы, созданные в соответствии с паттерном *State Machine*, образуют пакет connection. Диаграмма классов этого пакета приведена на рис. 33.

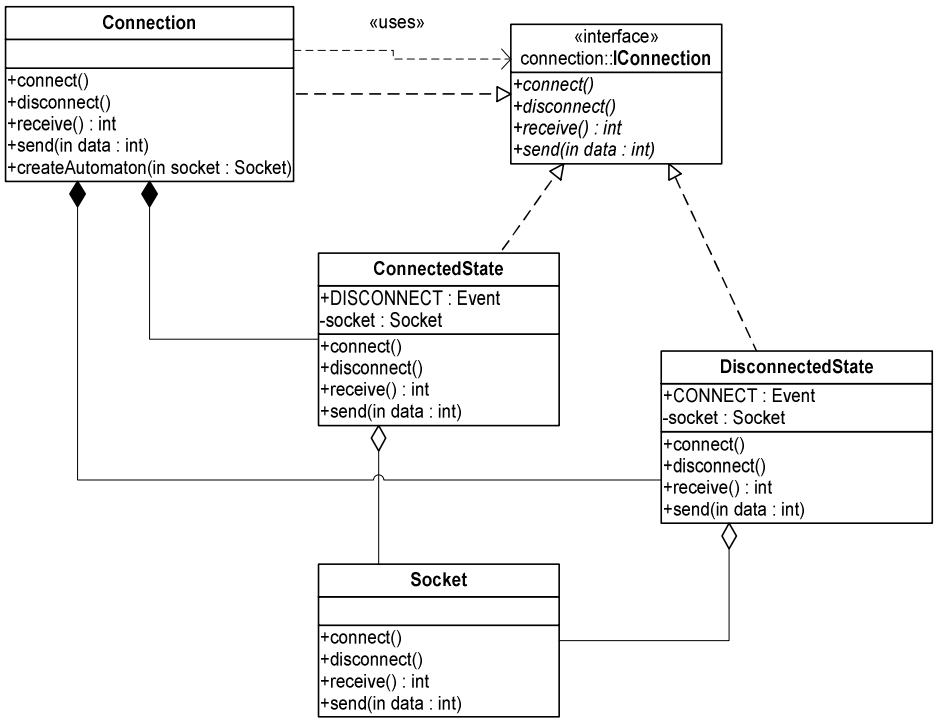

Рис. 33. Диаграмма классов пакета **`connection`**

В качестве модели данных автомата используется класс `Socket`, в рассматриваемом случае реализующий интерфейс `IConnection`.

После определения интерфейса для клиента и модели данных необходимо перейти к определению управляющих состояний автомата. Для данного примера реализуем классы `ConnectedState` и `DisconnectedState`. В состоянии `ConnectedState` могут произойти события `ERROR` и `DISCONECT`, а в состоянии `DisconnectedState` — события `CONNECT` и `ERROR` (рис. 29).

Обратим внимание, что на рис. 29 присутствуют дуги, помеченные одинаковыми событиями. В данном примере объекты событий создаются в классе состояния, из которого исходит переход. Например, событие `ERROR` на переходе из состояния `ConnectedState` в состояние `DisconnectedState` не совпадает с аналогичным событием на петле из

состояния DisconnectedState. При другой реализации для события ERROR мог бы быть создан только один объект.

Приведем код классов состояний.

Класс ConnectedState:

```
package connection;

import ru.ifmo.is.sm.*;
import java.io.IOException;

public class ConnectedState <AI extends IConnection>
    extends StateBase<AI> implements IConnection {
  public static final Event DISCONNECT = new
        Event("DISCONNECT");
  public static final Event ERROR = new Event("ERROR");

  protected final Socket socket;

  public ConnectedState(AI automaton, IEventSink
        eventSink, Socket socket) {
    super(automaton, eventSink);
    this.socket = socket;
  }

  public void connect() throws IOException {
  }

  public void disconnect() throws IOException {
    try {
      socket.disconnect();
    } finally {
      eventSink.castEvent(DISCONNECT);
    }
  }
```

```
  public int receive() throws IOException {
    try {
      return socket.receive();
    } catch (IOException e) {
      eventSink.castEvent(ERROR);
      throw e;
    }
  }

  public void send(int value) throws IOException {
    try {
      socket.send(value);
    } catch (IOException e) {
      eventSink.castEvent(ERROR);
      throw e;
    }
  }
}
```

Обратим внимание, что класс состояния только частично специализирует параметр типа класса `StateBase`. При расширении интерфейса автомата наследники класса состояния еще более специализируют этот тип. Окончательная специализация указывается при создании класса состояния в конструкторе автомата. Это обеспечивает возможность последующего расширения интерфейса автомата и классов состояний.

Класс `DisconnectedState`:

```
package connection;

import java.io.IOException;
import ru.ifmo.is.sm.*;

public class DisconnectedState <AI extends IConnection>
    extends StateBase<AI> implements IConnection {
```

```
  public static final Event CONNECT = new
        Event("CONNECT");
  public static final Event ERROR = new Event("ERROR");

  protected final Socket socket;

  public DisconnectedState(AI automaton, IEventSink
        eventSink, Socket socket) {
    super(automaton, eventSink);
    this.socket = socket;
  }

  public void connect() throws IOException {
    try {
      socket.connect();
    } catch (IOException e) {
      eventSink.castEvent(ERROR);
      throw e;
    }
    eventSink.castEvent(CONNECT);
  }

  public void disconnect() throws IOException {
  }

  public int receive() throws IOException {
    throw new IOException("Connection is closed
        (receive)");
  }

  public void send(int value) throws IOException {
    throw new IOException("Connection is closed (send)");
  }
}
```

Остается описать класс `Connection` — контекст. В этом классе реализована логика переходов, в соответствии с графом переходов на рис. 29. Заметим, что последние четыре метода этого класса — делегирование интерфейса текущему состоянию:

```
package connection;

import java.io.IOException;
import ru.ifmo.is.sm.AutomatonBase;

public class Connection extends
        AutomatonBase<IConnection>
    implements IConnection {
  private Connection() {
    Socket socket = new Socket();

    // Создание объектов состояний
    IConnection connected = new
        ConnectedState<Connection>(this, this, socket);
    IConnection disconnected = new
        DisconnectedState<Connection>(this, this,
        socket);

    // Логика переходов
    addEdge(connected, ConnectedState.DISCONNECT,
        disconnected);
    addEdge(connected, ConnectedState.ERROR,
        disconnected);
    addEdge(disconnected, DisconnectedState.CONNECT,
        connected);
    addEdge(disconnected, DisconnectedState.ERROR,
        disconnected);

    // Начальное состояние
    state = disconnected;
```

```
    }

    // Создание экземпляра автомата
    public static IConnection createAutomaton() {
      return new Connection();
    }

    // Делегирование методов интерфейса
    public void connect() throws IOException {
          state.connect(); }
    public void disconnect() throws IOException {
          state.disconnect(); }
    public int receive() throws IOException { return
          state.receive(); }
    public void send(int value) throws IOException {
          state.send(value); }
}
```

Обратим внимание, что в классах состояний определена только логика генерации событий, а логика переходов сконцентрирована в классе контекста.

3.2. Повторное использование классов состояний

Рассмотрим два расширения класса `Connection`. Первое расширение будет демонстрировать возможность добавления методов в интерфейс класса, а второе — изменение поведения за счет введения новых состояний.

3.2.1. Расширение интерфейса автомата

Проиллюстрируем возможность расширения интерфейса автомата на примере добавления возможности возврата данных в объект соединения для их последующего считывания. Введем интерфейс `IPushBackConnection`,

расширяющий интерфейс `IConnection` методом `pushBack`. Таким образом, расширенный интерфейс выглядит следующим образом:

```
package push_back_connection;

import connection.IConnection;
import java.io.IOException;

public interface IPushBackConnection extends IConnection
        {
  void pushBack(int value) throws IOException;
}
```

Отметим, что код для этого примера помещен в пакет `push_back_connection`, диаграмма классов которого представлена на рис. 34.

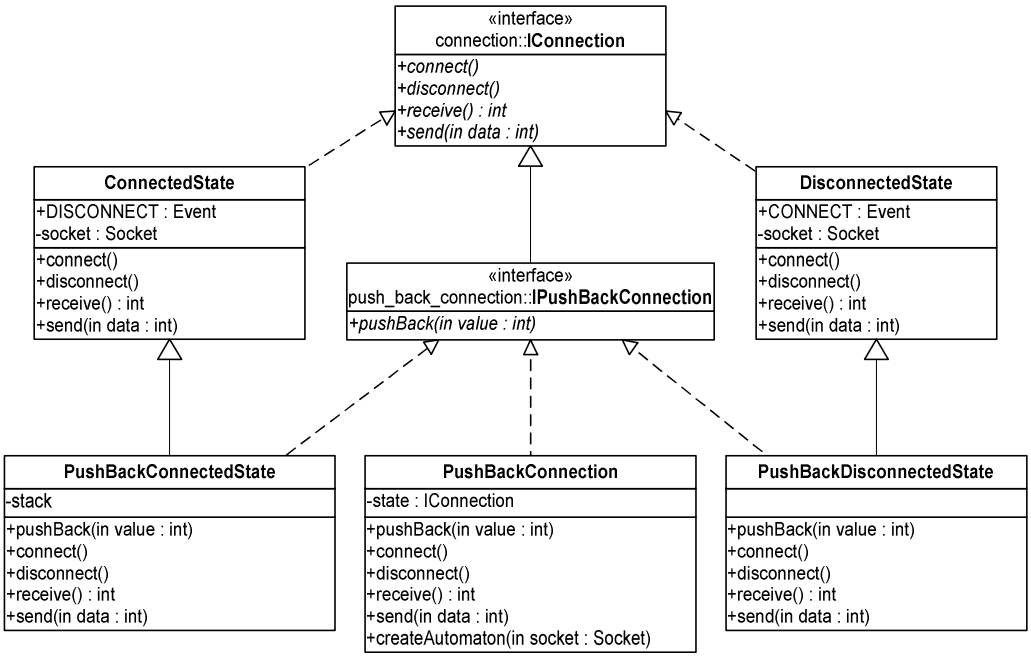

Рис. 34. Диаграмма классов пакета **`push_back_connection`**

При вызове метода `pushBack(int value)` значение, переданное в параметре этого метода, заносится в стек. При последующем вызове метода

`receive` возвращается элемент из вершины стека, а если стек пуст, то значение извлекается из объекта `socket`.

Заметим, что в рассматриваемом случае количество управляющих состояний автомата не изменится, равно как и граф переходов автомата, но контекст и классы состояний должны реализовывать более широкий интерфейс `IPushBackConnection`. Назовем контекст нового автомата `BushBackConnection`, а новые состояния — `PushBackConnectedState` и `PushBackDisconnectedState`.

Приведем реализацию класса `PushBackConnectedState` (класс `PushBackDisconnectedState` реализуется аналогично). При этом класс `PushBackConnectedState` расширяет класс `ConnectedState`, наследуя его логику:

```
package push_back_connection;

import connection.*;
import ru.ifmo.is.sm.IEventSink;
import java.util.Stack;
import java.io.IOException;

public class PushBackConnectedState <AI extends
        IPushBackConnection>
    extends ConnectedState<AI> implements
        IPushBackConnection {
  Stack<Integer> stack = new Stack<Integer>();

  public PushBackConnectedState(AI automaton, IEventSink
        eventSink, Socket socket) {
    super(automaton, eventSink, socket);
  }

  public int receive() throws IOException {
    if (stack.empty()) {
      return super.receive();
```

```
    }

    return stack.pop().intValue();
  }

  public void pushBack(int value) {
    stack.push(new Integer(value));
  }
}
```

Отметим, что в классе `PushBackConnectedState` параметр типа специализируется еще более чем в классе `ConnectedState`. Это требуется для того, чтобы поле `automaton`, определенное в классе `StateBase` имело правильный тип — `IPushBackConnection`. Таким образом, интерфейс автомата “протягивается” сквозь всю иерархию классов состояний через параметр типа.

Созданные классы состояний используются для реализации класса контекста `PushBackConnection`:

```
package push_back_connection;

import connection.Socket;
import ru.ifmo.is.sm.AutomatonBase;
import java.io.IOException;

public class PushBackConnection extends
        AutomatonBase<IPushBackConnection>
    implements IPushBackConnection {
  private PushBackConnection() {
    Socket socket = new Socket();

    // Создание объектов состояний
```

```
  IPushBackConnection connected = new
      PushBackConnectedState<PushBackConnection>(this,
      this, socket);
  IPushBackConnection disconnected = new
      PushBackDisconnectedState<PushBackConnection>(thi
      s, this, socket);

  // Логика переходов
  addEdge(connected, PushBackConnectedState.DISCONNECT,
      disconnected);
  addEdge(connected, PushBackConnectedState.ERROR,
      disconnected);
  addEdge(disconnected,
      PushBackDisconnectedState.CONNECT, connected);

  // Начальное состояние
  state = disconnected;
}

// Создание экземпляра автомата
public static IPushBackConnection createAutomaton() {
  return new PushBackConnection();
}

// Делегирование методов интерфейса
public void connect() throws IOException {
      state.connect(); }
public void disconnect() throws IOException {
      state.disconnect(); }
public int receive() throws IOException { return
      state.receive(); }
public void send(int value) throws IOException {
      state.send(value); }
public void pushBack(int value) throws IOException {
      state.pushBack(value); }
```

```
}
```

Приведенный пример иллюстрирует, что классы состояний могут использоваться повторно в случае расширения интерфейса автомата.

3.2.2. Расширение логики введением новых состояний

Расширение логики поведения будем рассматривать на примере сетевого соединения, которое в случае возникновения ошибки при передаче данных закрывает канал и бросает исключение. При очередной попытке передачи данных производится попытка восстановить соединение и передать данные.

Для описания такого поведения требуется ввести новое состояние — *ошибка*, переход в которое означает, что канал передачи данных, закрыт из-за ошибки. Вызов любого метода передачи данных в этом состоянии будет приводить к установке нового соединения.

Таким образом, требуется реализовать класс `ResumableConnection`. Для этого необходимо дополнительно реализовать класс `ErrorState`, определяющий поведение в состоянии *ошибка*. Для состояний *соединено* и *разъединено* используются классы `ConnectedState` и `DisconnectedState`, уже разработанные в разд. 3.1.9. Граф переходов для класса `ResumableConnection` изображен на рис. 35.

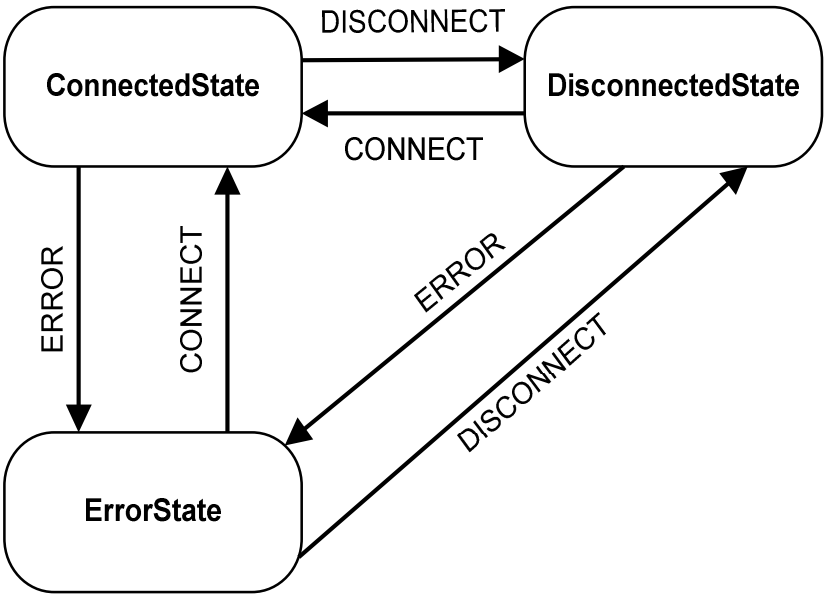

Рис. 35. Граф переходов класса **`ResumableConnection`**

Класс `ErrorState` реализуется следующим образом:

```
package resumable_connection;

import connection.*;
import ru.ifmo.is.sm.*;
import java.io.IOException;

public class ErrorState <AI extends IConnection>
    extends StateBase<AI> implements IConnection {
  public static final Event CONNECT = new
        Event("CONNECT");
  public static final Event DISCONNECT = new
        Event("DISCONNECT");

  protected final Socket socket;

  public ErrorState(AI automata, IEventSink eventSink,
        Socket socket) {
    super(automata, eventSink);
    this.socket = socket;
```

```
  }

  public void connect() throws IOException {
    socket.connect();
    castEvent(CONNECT);
  }

  public void disconnect() throws IOException {
    castEvent(DISCONNECT);
  }

  public int receive() throws IOException {
    connect();
    return automaton.receive();
  }

  public void send(int value) throws IOException {
    connect();
    automaton.send(value);
  }
}
```

Класс `ResumableConnection` реализуется следующим образом:

```
package resumable_connection;

import connection.*;
import ru.ifmo.is.sm.AutomatonBase;
import java.io.IOException;

public class ResumableConnection extends
        AutomatonBase<IConnection>
    implements IConnection {
  private ResumableConnection() {
    Socket socket = new Socket();
```

```
  // Создание объектов состояний
  IConnection connected = new
      ConnectedState<ResumableConnection>(this, this,
      socket);
  IConnection disconnected = new
      DisconnectedState<ResumableConnection>(this,
      this, socket);
  IConnection error = new
      ErrorState<ResumableConnection>(this, this,
      socket);

  // Логика переходов
  addEdge(connected, ConnectedState.DISCONNECT,
      disconnected);
  addEdge(connected, ConnectedState.ERROR, error);
  addEdge(disconnected, DisconnectedState.CONNECT,
      connected);
  addEdge(disconnected, DisconnectedState.ERROR ,
      error);
  addEdge(error, ErrorState.CONNECT, connected);
  addEdge(error, ErrorState.DISCONNECT, disconnected);

  // Начальное состояние
  state = disconnected;
}

// Создание экземпляра автомата
public static IConnection createAutomaton() {
  return new ResumableConnection();
}

// Делегирование методов интерфейса
public void connect() throws IOException {
      state.connect(); }
```

```
    public void disconnect() throws IOException {
          state.disconnect(); }
    public int receive() throws IOException { return
          state.receive(); }
    public void send(int value) throws IOException {
          state.send(value); }
}
```

Из приведенного примера видно, что классы состояний могут быть использованы повторно при реализации других автоматов.

Выводы

Паттерн *State Machine* является усовершенствованием паттерна *State*. Он заимствует основную идею паттерна *State* — локализацию поведения, зависящего от состояния, в различных классах.

Новый паттерн устраняет ряд недостатков, присущих паттерну *State*.

1. Паттерн *State Machine* позволяет разрабатывать отдельные классы независимыми друг от друга. Поэтому один и тот же класс состояния можно использовать в нескольких автоматах, каждый со своим набором состояний. Таким образом, устраняется главный недостаток паттерна *State* — сложность повторного использования.
2. В паттерне *State* не описано каким образом обеспечивается чистота интерфейса, предназначенного для клиента. Предлагаемый паттерн устраняет эту проблему.
3. В паттерне *State* логика переходов распределена по классам состояний, что порождает зависимости между классами и сложность восприятия логики переходов в целом. В паттерне *State Machine* логика переходов реализуется в контексте. Это позволяет разделить логику переходов и поведение в конкретном состоянии.
4. Использование паттерна *State Machine* не приводит к дублированию интерфейсов.

Тем не менее, паттерн *State Machine* не устраняет такой недостаток паттерна *State*, как необходимость производить делегирование интерфейса автомата текущему объекту состояния вручную. Это ограничение можно обойти, используя автоматическую генерацию кода контекста или языки программирования, поддерживающие динамическое делегирование. Автоматическая генерация кода обычно используется в *CASE*-средствах. Второй подход можно реализовать, например, на языке *Self* [47], в котором описанное выше делегирование можно выполнить при помощи динамического наследования. Это обеспечивается тем, что язык позволяет непосредственно изменять класс объекта во время выполнения, а объекты в этом языке могут делегировать операции другим объектам.

Глава 4. Язык программирования *State Machine*

В программировании часто возникает потребность в объектах, изменяющих свое поведение в зависимости от состояния. Обычно поведение таких объектов описывается при помощи конечных автоматов. Существуют различные паттерны проектирования для реализации указанных объектов, приведенные, например, в работах [5, 40]. В большинстве из этих паттернов или автоматы реализуются неэффективно, или сильно затруднено повторное использование компонентов автомата. Эти недостатки устранены в предложенном авторами паттерне ***State Machine*** [48].

В последнее время имеет место тенденция создания языков, ориентированных на предметную область [49]. В данном случае такой областью является автоматное программирование.

Одним из способов создания предметно-ориентированных языков является расширение существующих, например, за счет введения в них автоматных конструкций [50, 51].

В работе [52] был предложен язык ***State*** (расширяющий язык *C#* [53]), который предназначен для реализации объектов с изменяющимся поведением. Однако, конструкции, вводимые в этом языке, невозможно реализовать эффективно, поскольку в нем вызов метода объекта влечет за собой вычисление набора предикатов.

Зарекомендовавшим себя способом расширения языков программирования является встраивание в них поддержки паттернов проектирования. Например, в язык *C#* встроен паттерн ***Observer*** [5], широко используемый, например, для реализации графического интерфейса пользователя.

В данной работе предлагается язык программирования ***State Machine***, расширяющий язык ***Java*** [54], который основывается на одноименном паттерне. В качестве основного языка был выбран язык программирования

Java, так как для него существуют современные инструменты создания компиляторов, для которых есть не только документация, но и книга [55].

Глава состоит из пяти частей. В первой части описываются особенности предлагаемого языка *State Machine*. Во второй — приведен пример класса, реализующего сетевое соединение, и его реализация на предлагаемом языке. Третья часть содержит описание грамматики вводимых синтаксических конструкций. В четвертой части рассматривается повторное использование кода, написанного на предлагаемом языке. Пятая часть содержит описание реализации препроцессора.

4.1. Особенности языка *State Machine*

В предлагаемом языке, как и в паттерне *State Machine*, основной идей является описание объектов, варьирующих свое поведение, в виде автоматов. В предложенном в работе [48] подходе разделяются классы, реализующие логику переходов (контексты), и классы состояний. 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.

Отметим, что если в паттерне *State* [5] следующее состояние указывается текущим, то в паттерне *State Machine* это выполняется путем уведомления класса контекста о наступлении события.

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 таблицы переходов. Для описания логики переходов на этапе проектирования используются специальные графы переходов, предложенные в работе [48]. Эти графы состоят только из состояний и переходов, помеченных событиями.

По сравнению с языком *Java* в язык *State Machine* введены дополнительные конструкции, позволяющие описывать объекты с

варьирующимся поведением в терминах автоматного программирования, определенных в работе [48]: *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 реализовывалось вручную. В программе на языке *State Machine* это делать не требуется, так как препроцессор автоматически сгенерирует соответствующий код. Для этого используется технология *Reflection* [56]. Поэтому важно, чтобы препроцессор и генерируемый им код были реализованы на одном языке программирования. В частности, если бы язык расширял язык *C#* (как язык *State*), то и сам препроцессор необходимо было бы написать на языке *C#*.

Также как в паттерне *State Machine*, в предлагаемом языке состояние может делегировать дальнейшее выполнение действия автомату. При этом для ссылки на автомат используется ключевое слово `automaton` (также как и при описании автоматов).

Делегирование действия автомату требуется, например, при восстановлении после ошибки (соответствующий пример рассмотрен в разд. 4.4). В этом случае 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`. Авторами 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`) [57], что позволяет осуществлять проверку типов во время компиляции. Полученный код компилируется при помощи *Java*-компилятора, поддерживающего параметры типа.

4.2. Пример использования языка *State Machine*

В данном разделе особенности новых синтаксических конструкций языка *State Machine* рассматриваются на примере проектирования и реализации класса `Connection`, описанного в работе [48]. Приведем его краткое описание.

4.2.1. Описание примера

Требуется спроектировать класс `Connection`, представляющий сетевое соединение, имеющее два управляющих состояния: *соединено* и *разъединено*. Переход между ними 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. В случае возникновения ошибки при передаче данных генерируется исключительная ситуация (`IOException`) и сетевое соединение переходит в состояние *разъединено*, в котором прием и отправка данных невозможны. При попытке осуществить передачу данных в этом состоянии объект также генерирует исключительную ситуацию.

Интерфейс, который требуется реализовать в классе `Connection`, выглядит следующим образом.

```
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;
}
```

В работе [48] для реализации состояний *соединено* и *разъединено*, предложено использовать классы `ConnectedState` и `DisconnectedState` соответственно. Состояния уведомляют контекст о событиях: класс `ConnectedState` — о событиях `DISCONNECT` и `ERROR`, а класс `DisconctedState` — о событиях `CONNECT` и `ERROR`.

Граф переходов для рассматриваемого примера представлен на рис. 29.

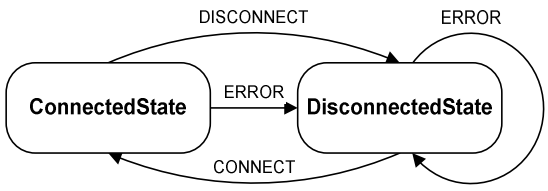

Рис. 36. Граф переходов для класса **`Connection`**

4.2.2. 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;
      }
    }
  }
```

При описании состояния указан 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`. За исключением этого, состояния описываются аналогично классу на языке *Java*.

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

Для реализации автомата `Connection`, необходимо также описать состояние `DisconnectedState`:

```
package connection;

import java.io.IOException;

public state DisconnectedState implements IConnection
        events CONNECT, ERROR {
  protected final Socket socket;

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

  public void connect() throws IOException {
    try {
      socket.connect();
      castEvent(CONNECT);
    } catch (IOException e) {
      castEvent(ERROR);
```

```
      throw e;
    }
  }

  public void disconnect() throws IOException {
  }

  public int receive() throws IOException {
    throw new IOException("Connection is closed");
  }

  public void send(int value) throws IOException {
    throw new IOException("Connection is closed");
  }
}
```

4.2.3. 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. 29):

```
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.

4.2.4. Компиляция примера

Для генерации *Java*-кода из файлов с расширением `.sm` необходимо выполнить команду `java ru.ifmo.is.sml.Main <имя файла1> [,имя файла2,…,имя файлаN]`.

В результате будет сформирован одноименный файл с расширением `.java`. Отметим, что для генерации класса `Connection` необходимо предварительно скомпилировать *Java*-компилятором интерфейс `IConnection`. Это требуется для генерации реализации этого интерфейса в соответствующем классе.

Для полной компиляции данного примера необходимо выполнить следующую последовательность команд:

```
rem Компиляция интерфейса автомата
javac IConnection.java

rem Преобразование состояний
java ru.ifmo.is.sml.Main ConnectedState.sm
        DisconnectedState.sm

rem Компиляция классов состояний
javac ConnectedState.java DisconnectedState.java

rem Преобразование автомата Connection
java ru.ifmo.is.sml.Main Connection.sm

rem Компиляция автомата
javac Connection.java
```

В результате будут сформированы соответствующие *Java*-файлы, которые будут скомпилированы *Java*-компилятором `javac`.

Отметим, что для компиляции и работы полученных классов требуется только класс `AutomatonBase`, определенный в пакете `ru.ifmo.is.sml.runtime`.

4.3. 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 [58] для описания этих конструкций.

4.3.1. Грамматика описания состояния

Ниже приведена грамматика описания состояния (табл.1).

Таблица 1. 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 [43].

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.

4.3.2. Грамматика описания автомата

Ниже приведена грамматика описания автомата (табл. 2).

Таблица 2. 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 является ошибкой.

4.4. Повторное использование

Одним из преимуществ объектно-ориентированного программирования является возможность повторного использования кода. Эта возможность также поддерживается и в предлагаемом языке.

4.4.1. Допустимые способы повторного использования

В *Java* объектно-ориентированном программировании интерфейс объекта представляет собой некоторый контракт [59]. При этом возможно наследование класса, основываясь только на его контракте. Наследование от автомата как класса допустимо, но для расширения его поведения в наследнике необходимо изменять набор состояний и функцию переходов. Таким образом, пользователь не может воспринимать базовый автомат как черный ящик, поскольку доступ к реализации функции переходов базового автомата нарушает инкапсуляцию. Поэтому наследование автомата от класса или другого автомата в языке *State Machine* запрещено.

При этом предлагаемый язык допускает повторное использование классов состояний. Также как и в паттерне *State Machine*, это может быть сделано двумя способами.

1. Наследование состояний.
2. Использование классов состояний в нескольких автоматах.

В первом способе создаются наследники состояний, реализующие более широкий интерфейс по сравнению с базовым состоянием. Из полученных состояний конструируется новый автомат. Во втором способе одни и те же классы состояний используются для конструирования различных автоматов. В обоих случаях определяется новый автомат со своим набором состояний и переходов в нем. Также возможна комбинация этих подходов.

Ниже оба способа повторного использования классов состояний будут рассмотрены на примерах.

4.4.2. Описание примеров

Следуя работе [48], рассмотрим две модификации автомата `Connection`.

Автомат `PushBackConnection` предоставляет возможность возврата данных в объект соединения и их последующего считывания. Интерфейс этого автомата — `IPushBackConnection` расширяет интерфейс `IConnection`:

```
package push_back_connection;

import connection.IConnection;
import java.io.IOException;

public interface IPushBackConnection extends IConnection
        {
  void pushBack(int value) throws IOException;
}
```

Автомат `ResumableConnection` реализует следующую логику поведения класса, представляющего сетевое соединение. В случае возникновения ошибки при передаче данных автомат закрывает канал связи и генерирует исключительную ситуацию. При очередном вызове метода передачи данных производится попытка восстановить соединение.

Для описания такого поведения требуется ввести новое состояние — *ошибка*, переход в которое означает, что канал передачи данных закрыт из-за ошибки. Полученный граф переходов, используемого в предлагаемом подходе вида, изображен на рис. 35.

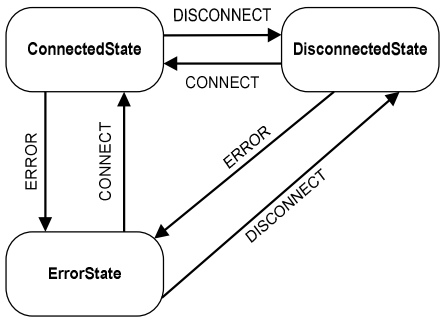

Рис. 37. Граф переходов автомата **`ResumableConnection`**

4.4.3. Наследование состояний

Для реализации автомата `PushBackConnection` необходимо описать два новых состояния `PushBackConnectedState` и `PushBackDisconnectedState`, унаследованных от состояний `ConnectedState` и `DisconnectedState` соответственно. Это позволяет обеспечить повторное использование кода состояний.

Приведем код состояния `PushBackConnectedState`:

```
package push_back_connection;

import connection.*;
import java.util.Stack;
import java.io.IOException;

public state PushBackConnectedState extends
        ConnectedState implements IPushBackConnection {
  private final Stack<Integer> stack = new
        Stack<Integer>();

  public PushBackConnectedState(Socket socket) {
    super(socket);
  }
```

```
  public int receive() throws IOException {
    return stack.empty() ? super.receive() : stack.pop();
  }

  public void pushBack(int value) {
    stack.push(value);
  }
}
```

Состояние `PushBackDisconnectedState` реализуется аналогично:

```
package push_back_connection;

import connection.*;
import java.io.IOException;

public state PushBackDisconnectedState extends
        DisconnectedState implements IPushBackConnection
        {
  public PushBackDisconnectedState(Socket socket) {
    super(socket);
  }

  public void pushBack(int value) throws IOException {
    throw new IOException("Connection is closed
        (pushBack)");
  }
}
```

Автомат `PushBackConnection` не наследует автомат `Connection`. Повторное использование кода достигается за счет применения наследников состояний `ConnectedState` и `DisconnectedState`.

```
package push_back_connection;

import connection.Socket;
```

```
public automaton PushBackConnection
  implements IPushBackConnection
  states
    PushBackConnectedState connected {
      ERROR -> disconnected,
      DISCONNECT -> disconnected
    },
    PushBackDisconnectedState disconnected {
      CONNECT -> connected,
      ERROR -> disconnected
    }
{
  public PushBackConnection(Socket socket) {
    connected @= new PushBackConnectedState(socket);
    disconnected @= new
        PushBackDisconnectedState(socket);
  }
}
```

4.4.4. Использование одного состояния в различных автоматах

Для реализации автомата `ResumableConnection` необходимо дополнительно реализовать состояние `ErrorState`, определяющее поведение в состоянии *ошибка*. В автомате также будут использованы состояния `ConnectedState` и `DisconnectedState`, разработанные для автомата `Connection`.

Приведем код состояния `ErrorState`:

```
package resumable_connection;

import connection.*;
import java.io.IOException;
```

```
public state ErrorState implements IConnection events
        CONNECT, DISCONNECT {
  protected final Socket socket;

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

  public void connect() throws IOException {
    socket.connect();
    castEvent(CONNECT);
  }

  public void disconnect() throws IOException {
    castEvent(DISCONNECT);
  }

  public int receive() throws IOException {
    connect();
    return automaton.receive();
  }

  public void send(int value) throws IOException {
    connect();
    automaton.send(value);
  }
}
```

Теперь можно определить автомат `ResumableConnection`:

```
package resumable_connection;

import connection.*;

public automaton ResumableConnection implements
        IConnection {
```

```
    state DisconnectedState disconnected(CONNECT ->
          connected, ERROR -> error);
    state ConnectedState connected(DISCONNECT ->
          disconnected, ERROR -> error);
    state ErrorState error(DISCONNECT -> disconnected,
          CONNECT -> connected);

    private ResumableConnection() {
      Socket socket = new Socket();

      connected @= new ConnectedState(socket);
      disconnected @= new DisconnectedState(socket);
      error @= new ErrorState(socket);
    }
  }
```

Из приведенных примеров следует, что состояния могут быть использованы повторно.

4.5. Реализация препроцессора

Для реализации препроцессора были использованы инструменты создания компиляторов *JLex* и *Cup*, описанные в работе [55], которые распространяются по лицензии *open source license* [60]. Первый из них предназначен для построения лексических анализаторов, а второй — синтаксических [43].

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. Однако это не является недостатком, поскольку получаемый код является промежуточным 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 [61].

4.5.1. Генерация *Java*-классов по описанию состояний

Каждое состояние, описанное на языке *State Machine*, преобразуется в одноименный класс на языке *Java*. При этом все методы и их реализация переходят в генерируемый класс.

В случае если одно состояние расширяет другое, то генерируемый класс будет расширять соответствующий ему *Java*-класс. В противном случае, генерируемый класс будет расширять класс `AutomatonBase.StateBase`, входящий в пакет `ru.ifmo.is.sml.runtime`.

Указанный класс является базовым для всех классов состояний. В нем определено поле `automaton`, позволяющее вызывать методы автомата, в котором содержится данный экземпляр состояния. Отметим, что в языке *StateMachine* `automaton` является ключевым словом. Таким образом, конфликтов с полями, определенными пользователем, не возникает.

В классе `StateBase` также реализован метод `castEvent`, который состояния вызывают для уведомления автомата о наступлении события. Для реализации событий используется класс `AutomatonBase.StateBase.Event`.

Рассмотрим код, сгенерированный препроцессором для состояния `ConnectedState`. Отметим, что здесь и ниже форматирование сгенерированного кода и вставка комментариев выполнены вручную:

```
package connection;

import java.io.IOException;
```

```
public class ConnectedState<AI extends IConnection>
  // Базовый класс, для всех состояний
  extends ru.ifmo.is.language.AutomatonBase.StateBase<AI>
  implements IConnection {
  // События преобразуются в набор статических переменных
  public final static Event DISCONNECT = new
        Event(ConnectedState.class, "DISCONNECT", 0);
  public final static Event ERROR = new
        Event(ConnectedState.class, "ERROR", 1);

  // В остальном -- без изменений

  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;
      }
    }
  }
```

Обратим внимание, что в этом коде, также как в паттерне ***State Machine***, для представления событий используются статические переменные. При создании экземпляров событий (класса `Event`) первыми двумя параметрами конструктора являются ссылка на класс, определивший событие и имя события, которые могут быть использованы для автоматического протоколирования [62]. Третий параметр необходим для эффективной реализации графов переходов. Он представляет собой порядковый номер события в состоянии, отсчитываемый от нуля. При этом учитываются события, определенные в базовом состоянии, которым присваиваются меньшие номера.

4.5.2. Генерация *Java*-классов по описанию автоматов

Автомат в языке ***State Machine*** также преобразуется в одноименный класс, унаследованный от класса `AutomatonBase`, определенного в пакете `ru.ifmo.is.sml.runtime`.

Препроцессор автоматически генерирует методы, реализующие интерфейс автомата. В каждом из них вызывается соответствующий метод текущего состояния. Такой метод гарантированно присутствует в состоянии, поскольку автомат и состояние реализуют интерфейс автомата.

Для автомата `Connection` препроцессор генерирует следующий *Java*-класс:

```
package connection;

import java.io.IOException;
import ru.ifmo.is.sml.runtime.AutomatonBase;

public class Connection
  extends AutomatonBase<IConnection>
  implements IConnection
{
  private final static int[][] $TRANSITION_TABLE
    = new int[][] {
      {
        /* CONNECT   */ 0, /* connected */
        /* ERROR     */ 1, /* disconnected */
      },
      {
        /* DISCONNECT  */ 1, /* disconnected */
        /* ERROR     */ 1, /* disconnected */
      },
    };

  public Connection(Socket socket) {
    super(new IConnection[2/*количество состояний*/]);
    {
      DisconnectedState<IConnection> $state = new
        DisconnectedState<IConnection>(socket);
      state(0/*disconnected*/, $state, $state, this,
        $TRANSITION_TABLE[0/*disconnected*/]);
    }
    {
      ConnectedState<IConnection> $state = new
        ConnectedState<IConnection>(socket);
```

```
            state(1/*connected*/, $state, $state, this,
              $TRANSITION_TABLE[1/*connected*/]);
          }
        }

        // Делегирование методов интерфейса автомата
        public void connect() throws IOException {
              state.connect(); }
        public void disconnect() throws IOException {
              state.disconnect(); }
        public int receive() throws IOException { return
              state.receive(); }
        public void send(int value) throws IOException {
              state.send(value); }
    }
```

Отметим, что использование имени `state` для поля, хранящего текущее состояние, не приводит к неоднозначности, так как в предлагаемом языке оно является ключевым словом. По этой же причине метод, связывающий состояние с автоматом, также называется `state`. Интересной особенностью является необходимость дважды передавать в этот метод ссылку на состояние (`state`), а так же ссылку на сам автомат (`this`). Это связано с невозможностью на языке *Java* определить класс, унаследованный от своего параметра типа.

На основе переходов, заданных в описании автомата, препроцессор строит таблицу переходов, хранящуюся в статическом поле `$TRANSITION_TABLE`. Таблица представляет собой массив массивов целых чисел. В приведенном коде таблица переходов является матрицей `2x2`. Однако, в общем случае, она матрицей не является, поскольку состояния могут порождать разное количество событий.

Строки таблицы переходов передаются объектам состояний при помощи вызова метода `init`, входящего в класс `AutomatonBase`. Это

позволяет представить функцию уведомления о событии (`castEvent`) в компактном виде.

При такой реализации для осуществления каждого перехода требуется фиксированное время, существенно меньшее, чем в реализации, предложенной в работе [48]. Таким образом, базовый класс для всех автоматов выглядит следующим образом:

```
package ru.ifmo.is.sml.runtime;

public abstract class AutomatonBase<AI> {
  private final AI[] states;
  protected AI state;

  public AutomatonBase(AI[] states) {
    this.states = states;
  }

  protected void state(int index, StateBase<AI> state, AI
        stateI, AI automaton, int[] transitions) {
    states[index] = stateI;
    state.base = this;
    state.automaton = automaton;
    state.transitions = transitions;
    if (index == 0) this.state = states[index];
  }

  public static abstract class StateBase<AI> {
    protected AI automaton;
    private AutomatonBase<AI> base;
    private int[] transitions;

    protected void castEvent(Event event) {
      base.state = base.states[transitions[event.index]];
    }
```

```
    public final static class Event {
      private final Class state;
      private final String name;
      private final int index;

      public Event(Class state, String name, int index) {
        this.state = state;
        this.name = name;
        this.index = index;
      }
    }
  }
}
```

Обратим внимание, что класс `AutomatonBase` содержит класс `StateBase` как вложенный.

Выводы

Предложенный язык программирования *State Machine* обладает следующими достоинствами.

1. Позволяет писать программы в терминах автоматного программирования.
2. Непосредственно поддерживает одноименный паттерн.
3. Повышает компактность и облегчает восприятие кода, по сравнению с реализацией паттерна *State Machine* непосредственно на языке *Java*.
4. Обеспечивает более быстрое выполнение переходов по сравнению с реализацией паттерна *State Machine*, приведенного в работе [48].
5. Предложенный синтаксис позволяет распознавать паттерн в коде, в том числе и автоматически, что может быть использовано при построении диаграмм и документации.

Глава 5. Внедрение результатов работы

Описывается внедрение паттерна *State Machine*, предложенного авторами, при проектировании системы управления потоками (thread), осуществляющими асинхронные запросы к базе данных. Выполнено сравнение реализации с использованием предлагаемого паттерна с реализациями на основе флагов и *SWITCH*-технологии.

В программировании часто возникает потребность в объектах, изменяющих свое поведение в зависимости от состояния. Обычно поведение таких объектов описывается при помощи конечных автоматов. Существуют различные паттерны проектирования для реализации указанных объектов, описанные, например, в работах [5, 40]. В большинстве из этих паттернов или автоматы реализуются неэффективно, или сильно затруднено повторное использование их компонентов. Эти недостатки устранены в предложенном авторами паттерне *State Machine* [48].

В процессе разработки программного обеспечения в компании *Транзас* [9] используются паттерны проектирования [5]. При этом до последнего времени для объектов, изменяющих свое поведение в зависимости от состояния, применялся паттерн *State*.

Кроме того, отметим, что для успешной разработки крупных проектов необходимо систематически производить улучшение существующего кода — рефакторинг [44], который требуется для того, чтобы целостность программной системы всегда находилась на приемлемом уровне. Одним из методов рефакторинга является метод, названный «*Replace Type Code with State*», идея которого состоит в замене условной логики паттерном *State*.

В настоящей работе предлагается условную логику заменять паттерном *State Machine*, который, как было показано в работе [48], обладает определенными преимуществами по сравнению с паттерном *State*. Это продемонстрировано при проектировании системы управления потоками,

осуществляющими асинхронные запросы к базе данных. Выполнено сравнение предложенной реализации с другими подходами — программированием с использованием флагов и *SWITCH*-технологии.

5.1. Область внедрения

Группа компаний *Transas* была основана в 1990 году в Санкт-Петербурге. Название *Transas* означает TRANsport SAfety Systems (Системы Безопасности на Транспорте). Успешно работая уже более 10 лет, *Transas* является одним из ведущих производителей высокотехнологичных продуктов, пользующихся спросом во всем мире.

5.1.1. Система *Navi Harbour*

Система *Navi Habour* [63] разрабатывается в *департаменте береговых систем* [64] компании *Transas* и является системой управления движением судов (*СУДС*). Этот продукт установлен в портах многих стран мира, включая Россию. На рис. 38 приведена структурная схема системы *Navi Harbour*.

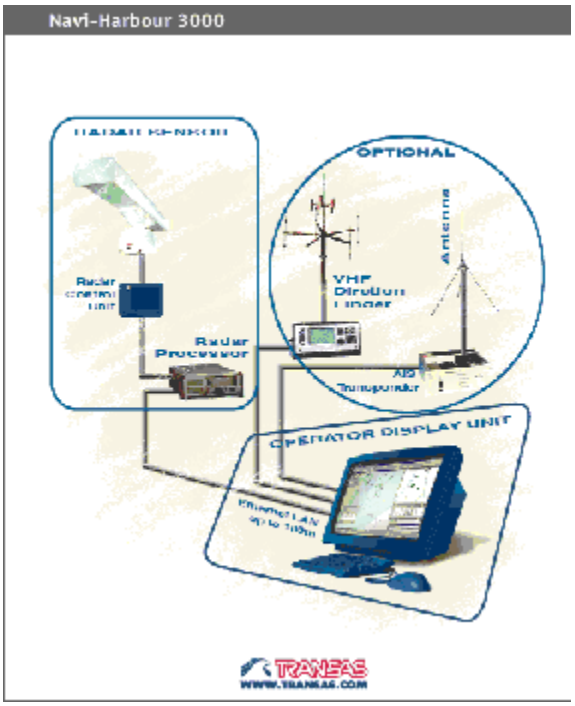

Рис. 38. Структурная схема системы *Navi Harbour*

На этом рисунке показано, что к операторскому дисплейному модулю (*ОДМ*) подключается радар и ряд дополнительных устройств (необязательных — *optional*), таких как, например, телекамеры и сенсоры погодных условий. На каждом дисплейном модуле отображается морская карта акватории, собираются данные устройств, выводятся различные тревоги (например, предупреждения столкновений, сообщения об ошибках устройств), а также другая информация, имеющая отношение к безопасности движения.

Одна из составляющих системы *Navi Harbour* — база данных *СУДС*, которая не представлена на рис. 38. В работе демонстрируется эффективность применения паттерна *State Machine* на примере использования его при разработке менеджера потоков в указанной базе данных.

5.1.2. База данных *СУДС*

Рассматриваемая база данных предназначена для учета судов и связанной с ними информации, такой как, например, судозаходы, журнал погоды, вахтенный журнал. База данных *СУДС* представляет собой набор устанавливаемых на различных компьютерах и взаимодействующих между собой компонентов. Схема развертывания базы данных *СУДС* приведена на рис. 39. Для упрощения схемы не показаны связи между дисплейными модулями.

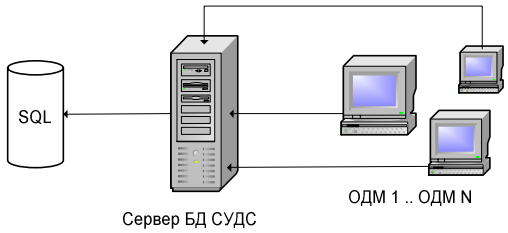

Рис. 39. Схема развертывания базы данных СУДС

В качестве хранилища данных в системе используется *Microsoft SQL Server*, который может быть размещен отдельно или на одном компьютере с сервером базы данных *СУДС*. На компьютерах *ОДМ1 … ОДМN* установлена клиентская часть базы данных, которая подключается к системе посредством технологии *COM*. Клиентская часть взаимодействует с серверной посредством технологии *DCOM*.

Типичное окно клиента базы данных приведено на рис. 40.

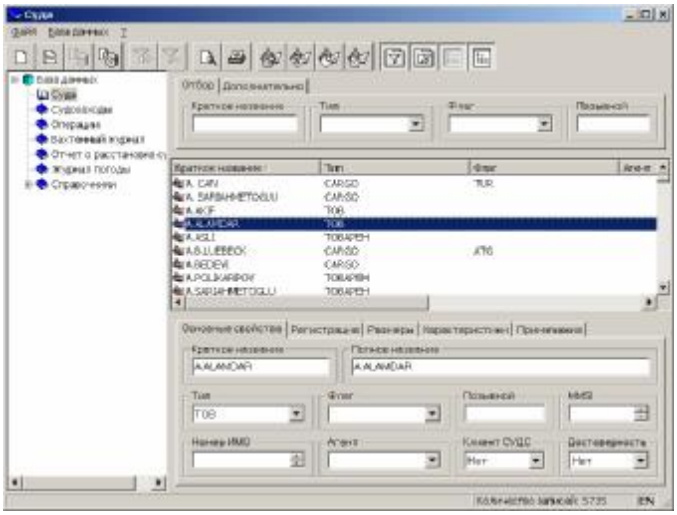

Рис. 40. Окно клиента базы данных

В этом окне производится просмотр, фильтрация и редактирование данных. При этом работа с сервером базы данных выполняется асинхронно — используются потоки (*threads*), управление которыми и является предметом внедрения предложенного паттерна.

5.2. Постановка задачи

Одним из основных требований к клиентской части базы данных *СУДС* является способность выполнять асинхронные запросы к ее серверной части. Это позволяет не блокировать работу системы *Navi Harbour* во время выполнения запроса, что очень важно для непрерывного мониторинга перемещений судов. Схема взаимодействия клиентской и серверной части приведена на рис. 41.

Серверная часть содержит компонент `VTSDB`, реализующий интерфейс `VTSDB.Application`, через который производится взаимодействие клиентской части с серверной. Компонент `DBWindow` создает потоки, каждый из которых содержит указатель на этот интерфейс.

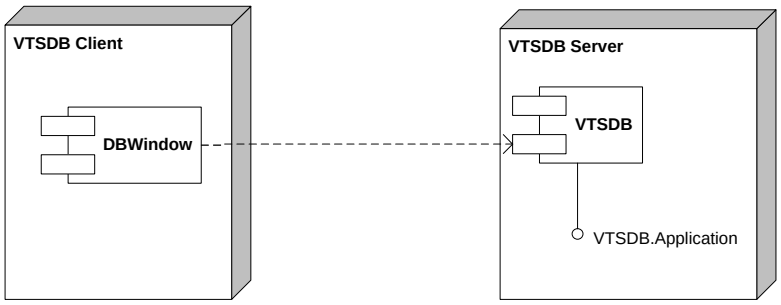

Рис. 41. Схема взаимодействия клиентской и серверной частей

Выполнение асинхронных запросов реализовано в клиентской части базы данных, посредством потоков соединения (*connection threads*). Потоки, хранящие соединение с серверной частью базы данных, создаются для каждого окна. В настоящей работе ставится задача разработки фабрики потоков — класса `ThreadFactory`, предназначенного для создания и уничтожения потоков соединения. Как будет показано в разд. 5.3, поведение этого класса варьируется в зависимости от состояния. Поэтому для его реализации будем применять паттерн *State Machine* [48]. Отметим, что в этой работе примеры, иллюстрирующие применение предложенного паттерна, реализованы на языке *Java*. Ввиду того, что структура паттерна не зависит от языка программирования, а также в связи с тем, что в компании *Транзас* в основном используется язык программирования *C++* [65], разрабатываемая фабрика потоков также реализован на этом языке использованием библиотеки *Boost* [66].

5.3. Применение паттерна *State Machine* для проектирования класса *ThreadFactory*

Покажем, как применять предлагаемый паттерн на примере проектирования класса `ThreadFactory`, введенного в разд. 5.2.

Реализация класса `ThreadFactory` и связанных с ним классов и интерфейсов произведена в пространстве имен `ThreadManagement`.

5.3.1. Формализация постановки задачи

Класс `ThreadFactory` управляет созданием и уничтожением потоков соединения. Опишем класс, инкапсулирующий поток соединения, который назовем `ConnectionThread`. Этот класс уведомляет о своей успешной инициализации (создании *DCOM*-объекта) или о завершении потока через интерфейс `IThreadNotify`. Указатель на этот интерфейс передается в конструктор класса `ConnectionThread`. Этот интерфейс, выглядит следующим образом:

```
#pragma once

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadNotify
{
  virtual void Started(ConnectionThread * threadPtr) = 0;
  virtual void Stopped(ConnectionThread * threadPtr) = 0;
};

}
```

При инициализации объект `ConnectionThread` вызывает метод `Started`, а при завершении — метод `Stopped` и передает в них указатель на себя.

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.

Интерфейс этого класса имеет следующий вид:

```
#pragma once

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadFactory
{
  virtual void Request() = 0;
  virtual void Destroy(ConnectionThread * threadPtr) = 0;
  virtual void Cancel() = 0;
};

}
```

Метод `Request` запрашивает поток соединения, метод `Cancel` отменяет запрос на создание потока соединения, а метод `Destroy` — удаляет поток соединения. Уведомления клиента класса `ThreadFactory` о создании или уничтожении потока соединения производится через интерфейс `IThreadRequest`:

```
#pragma once

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadRequest
{
  virtual void Created(ConnectionThread * threadPtr) = 0;
  virtual void Deleted() = 0;
  virtual void Canceled() = 0;
};

}
```

Метод `Created` вызывается при создании объекта соединения, в него передается указатель на инициализированный поток. Метод `Canceled` вызывается при успешной отмене запроса на создание соединения, а метод `Deleted` — при удалении объекта соединения.

Поведение класса `ThreadFactory` зависит от внутреннего состояния. Например, метод `Cancel` приводит к отмене создания потока только в том случае, если объект класса `ThreadFactory` создал объект потока и ожидает завершения его инициализации. Поэтому поведение этого класса можно описывать с помощью конечного автомата. Для его реализации решено применить паттерн *State Machine*, в котором класс `ThreadFactory` является контекстом. Контекст является единственным классом, входящим в паттерн, который доступен пользователю.

5.3.2. Проектирование автомата *ThreadFactory*

Выделим четыре управляющих состояния:

1. *Idle* — ожидание запроса на создание или удаление потока соединения;
2. *Creating* — поток создан и ожидается завершение его инициализации;
3. *Destroing* — выполнен запрос на уничтожение потока и ожидается завершения потока;
4. *Cancelling* — инициализация потока отменена и ожидается завершения потока.

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

5. *Idle*: `CREATED` — объект потока создан, `STOP_SENDED` — объекту потока отправлен запрос на остановку;

6. *Creating*: `INITIALIZED` — поток инициализирован; `CANCEL_SENDED` — объекту потока отправлен запрос на отмену инициализации и остановку;
7. *Destroing*: `DESTROYED` — поток уничтожен;
8. *Cancelling*: `CANCELLED` — отмена создания потока.

На рис. 42 приведен описывающий поведение класса `ThreadFactory` граф переходов вида, используемого в предлагаемом подходе [48].

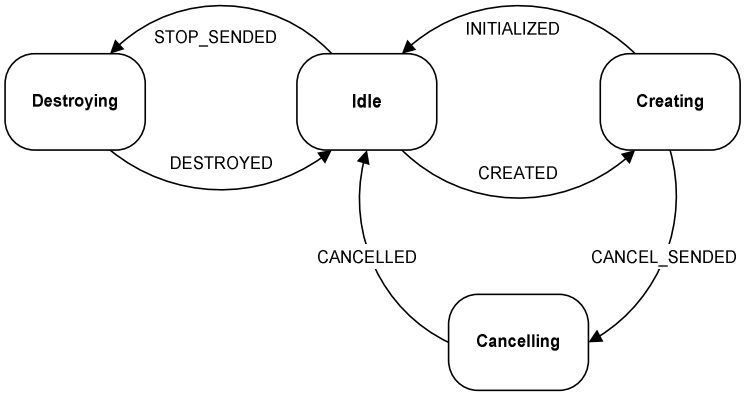

Рис. 42. Граф переходов, описывающий поведение класса **`ThreadFactory`**

5.3.3. Диаграмма классов реализации автомата *ThreadFactory*

Как было описано в разд. 5.3.1, класс `ThreadFactory` должен реализовывать интерфейс `IThreadFactory`. Для получения уведомлений от создаваемого потока класс `ThreadFactory` должен реализовывать интерфейс `IThreadNotify`.

Таким образом, интерфейсом автомата является интерфейс, расширяющий интерфейсы `IThreadFactory` и `IThreadNotify`:

```
#pragma once

#include ".\IThreadFactory.h"
#include ".\IThreadNotify.h"
```

```
namespace ThreadManagement
{

struct IThreadFactoryAI : public IThreadFactory, public
        IThreadNotify
{
};

}
```

Таким образом, в пространстве имен `ThreadManagement` находятся следующие классы и интерфейсы:

1. `ConnectionThread` — класс потока соединения;
2. `IThreadRequest` — интерфейс уведомления о создании и уничтожении объектов потока;
3. `IThreadFactory` — интерфейс фабрики потоков;
4. `IThreadNotify` — интерфейс для уведомления о создании/уничтожении потока;
5. `IThreadFactoryAI` — интерфейс автомата. Наследуется от интерфейсов `IThreadFactory` и `IThreadNotify`;
6. `ThreadFactory` — контекст. Реализуют интерфейс автомата `IThreadFactoryAI`, наследуясь от класса `StateMachine::AutomatonBase<IThreadFactoryAI>`;
7. `Idle`, `Creating`, `Destroying`, `Cancelling` — классы состояний. Реализуют интерфейс автомата `IThreadFactoryAI`, наследуясь от класса `StateMachine::StateBase<IThreadFactoryAI>`.

Отметим, что в пространство имен `ThreadManagement`, наряду с классами паттерна *State Machine*, входят также связанные с ними классы и интерфейсы, такие как `ConnectionThread` и `IThreadRequest`.

Диаграмма классов реализации автомата `ThreadFactory`, приведена на рис. 43.

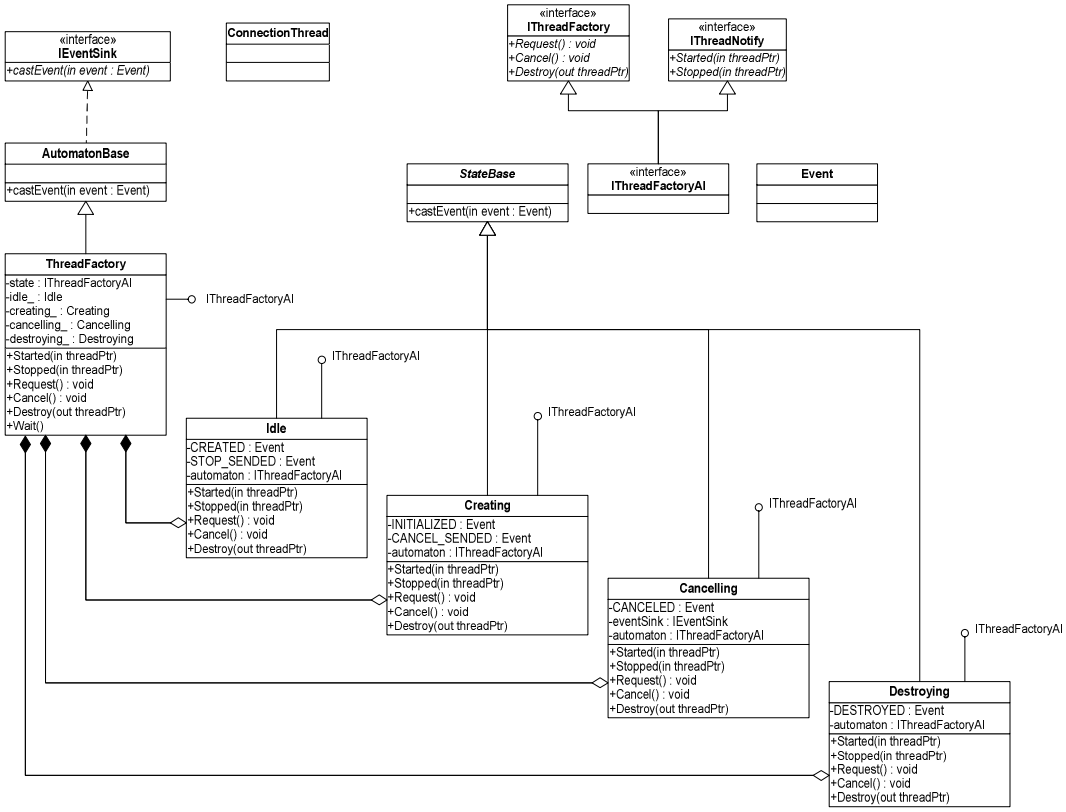

Рис. 43. Диаграмма классов реализации автомата **`ThreadFactory`**

Отметим, что для того, чтобы не загромождать рисунок, тот факт, что контекст и классы состояний реализуют интерфейс `IThreadFactoryAI`, изображен на рис. 43 в виде линий с кружками. Это заменяет пунктирные линии, которыми реализация интерфейса изображалась в структуре паттерна *State Machine*, приведенной в работе [48].

5.3.4. Реализация контекста автомата *ThreadFactory*

В соответствии с паттерном *State Machine* логика переходов задается в конструкторе контекста — классе `ThreadFactory`. В этот класс также добавлен метод `Wait`, предназначенный для ожидания завершения текущей операции (например, ожидания удаления потока).

Все методы, входящие в интерфейс автомата, реализуется однотипно: защита от многопоточного доступа (`CSingleLock locker(&lock_, TRUE)`) и делегирование операции текущему объекту состояния (например, для метода `Request` — `statePtr_->Request()`).

Приведем реализацию класса `ThreadFactory` — контекста. Отметим, что этот класс является наследником класса `AutomatonBase<IThreadFactoryAI>`, в котором реализована инфраструктура для выполнения переходов и протоколирования.

При инициализации контекст передает в конструктор своего базового класса `AutomatonBase<IThreadFactoryAI>` ссылку на начальное состояние — `idle_`. При этом вместо имени типа `AutomatonBase<IThreadFactoryAI>` используется имя `AB`, которое является его синонимом.

Переходы заданы в конструкторе класса `ThreadFactory` при помощи вызовов метода `AddTransion`. Этот метод также реализован в классе `AutomatonBase` с использованием структуры данных `map`, входящей стандартную библиотеку *C++*.

Заголовочный файл для класса `ThreadFactory`:

```
#pragma once

#include <afxmt.h>

#include "StateMachine\AutomatonBase.h"
#include ".\IThreadFactoryAI.h"
#include ".\Idle.h"
#include ".\Creating.h"
#include ".\Canceling.h"
#include ".\Destroying.h"

namespace ThreadManagement
{
```

```
class ConnectionThread;
struct IThreadRequest;
class ThreadFactory : public
        StateMachine::AutomatonBase<IThreadFactoryAI>
{
public:
  ThreadFactory(IThreadRequest & request);
  ~ThreadFactory(void);

  // Public Interface
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
  virtual void Started(ConnectionThread * threadPtr);
  virtual void Stopped(ConnectionThread * threadPtr);

  // Wait for current action finish
  void Wait();

private:
  CCriticalSection lock_;
  CEvent requestEvent_;
  IThreadRequest & request_;
  ConnectionThread * threadPtr_;

  // States
  Idle idle_;
  Creating creating_;
  Canceling canceling_;
  Destroying destroying_;
 };
```

```
}
```

Исполняемый файл для класса `ThreadFactory`:

```
#include "stdafx.h"

#include "boost\bind.hpp"
#include "StateMachine\Event.h"
#include "Logger.h"
#include ".\ThreadFactory.h"
#include ".\ConnectionThread.h"
#include ".\IThreadRequest.h"

namespace ThreadManagement
{

// This warning is "'this' : used in base member
        initializer list"
// There is no error here cause state classes demand
        reference to
// AutomatonBase<AI> which is already initialized.
#pragma warning(disable:4355)

ThreadFactory::ThreadFactory(IThreadRequest & request)
: AB(idle_),
  requestEvent_(TRUE, TRUE),
  request_(request),
  threadPtr_(NULL),
  idle_(*this, request, threadPtr_, requestEvent_),
  creating_(*this, request, threadPtr_, requestEvent_),
  canceling_(*this, request, threadPtr_, requestEvent_),
  destroying_(*this, request, threadPtr_, requestEvent_)
{
  AddTransition(idle_, Idle::CREATED, creating_);
  AddTransition(idle_, Idle::STOP_SENDED, destroying_);
  AddTransition(creating_, Creating::INITIALIZED, idle_);
```

```
  AddTransition(creating_, Creating::CANCEL_SENDED,
        canceling_);
  AddTransition(canceling_, Canceling::CANCELED, idle_);
  AddTransition(destroying_, Destroying::DESTROYED,
        idle_);
}

#pragma warning(default:4355)

ThreadFactory::~ThreadFactory(void)
{
  ASSERT(threadPtr_ == NULL);
}

void ThreadFactory::Request()
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Request();
}

void ThreadFactory::Cancel()
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Cancel();
}

void ThreadFactory::Destroy(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Destroy(threadPtr);
}

void ThreadFactory::Started(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
```

```
  statePtr_->Started(threadPtr);
}

void ThreadFactory::Stopped(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Stopped(threadPtr);
}

void ThreadFactory::Wait()
{
  ::WaitForSingleObject(requestEvent_, INFINITE);
}

}
```

Методы, образующие интерфейс автомата, могут вызываться из различных потоков. При этом методы интерфейса `IThreadFactory` вызываются из основного потока, а методы интерфейса `IThreadNotify` — из потоков, создаваемых фабрикой. Поскольку при вызове метода автомата состояние может измениться, для обеспечения потоковой безопасности в контексте производится блокировка на основе критической секции `CSingleLock locker(&lock_, TRUE)`. Отметим, что такое использование критических секций позволяет не задумываться о потоковой безопасности в классах состояний.

Макросы утверждений (`ASSERT` и `VERIFY`) [65] используются в программе для проверки ее целостности в режиме отладки.

5.3.5. Пример реализации класса состояния автомата *ThreadFactory*

Приведем в качестве примера реализацию одного из классов состояний `Idle`. Этот класс наследуется от класса `StateBase<IThreadFactoryAI>`. При инициализации в него передается ссылка на контекст и имя состояния — «`Idle`». При этом вместо имени типа

StateBase<IThreadFactoryAI> используется имя SB, которое являющееся его синонимом.

Заголовочный файл для класса Idle:

```
#pragma once

#include "StateMachine\StateBase.h"
#include "StateMachine\Event.h"
#include ".\IThreadFactoryAI.h"
#include ".\IThreadRequest.h"

#include <afxmt.h>

namespace ThreadManagement
{

class ConnectionThread;

class Idle : public
        StateMachine::StateBase<IThreadFactoryAI>
{
public:
  // Events
  static StateMachine::Event CREATED;
  static StateMachine::Event STOP_SENDED;

  Idle(AB & automaton, IThreadRequest & request,
        ConnectionThread * & threadPtr, CEvent &
        requestEvent);

  // IThreadFactory
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
```

```
  void Started(ConnectionThread * threadPtr);
  void Stopped(ConnectionThread * threadPtr);

private:
  IThreadRequest & request_;
  ConnectionThread * & threadPtr_;
  CEvent & requestEvent_;
};

}
```

Исполняемый файл для класса `Idle`:

```
#include "stdafx.h"

#include "ConnectionThread.h"
#include "Logger.h"

#include ".\Idle.h"
#include ".\exceptions.h"

#include "StateMachine\AutomatonBase.h"

namespace ThreadManagement
{

Idle::Idle(AB & automaton, IThreadRequest & request,
        ConnectionThread * & threadPtr, CEvent &
        requestEvent)
: SB(automaton, "Idle"),
  request_(request),
  threadPtr_(threadPtr),
  requestEvent_(requestEvent)
{
}

// IThreadFactory
```

```
void Idle::Request()
{
  ASSERT(threadPtr_ == NULL);
  Logger::Instance().Log("state Idle", "Trying to create
        new thread");
  threadPtr_ = new ConnectionThread(automaton_);
  VERIFY(threadPtr_->CreateThread());
  automaton_.CastEvent(CREATED);
  VERIFY(requestEvent_.ResetEvent());
}

void Idle::Cancel()
{
  throw CancelFailedException();
}

void Idle::Destroy(ConnectionThread * threadPtr)
{
  ASSERT(threadPtr_ == NULL);
  Logger::Instance().Log("state Idle", "Trying to destroy
        thread");
  threadPtr_ = threadPtr;
  threadPtr->Stop();
  automaton_.CastEvent(STOP_SENDED);
  VERIFY(requestEvent_.ResetEvent());
}

// IThreadNotify
void Idle::Started(ConnectionThread * threadPtr)
{
  ASSERT(FALSE);
}

void Idle::Stopped(ConnectionThread * threadPtr)
{
```

```
  ASSERT(FALSE);
}

StateMachine::Event Idle::CREATED("CREATED");
StateMachine::Event Idle::STOP_SENDED("STOP_SENDED");

}
```

Отметим, что для использования объекта синхронизации `CEvent` подключается заголовочный файл `<afxmt.h>`.

Завершая описание реализации класса `ThreadFactory`, отметим, что предложенный подход позволяет реализовать этот класс без условных операторов, что является признаком хорошего проектирования [44].

Отметим, что классы состояний не «знают» друг о друге. Это дает возможность, в общем случае, производить наследование от этих классов для последующего использования в других автоматах.

5.4. Приложение, визуализирующее работу класса *ThreadFactory*

Для визуализации и тестирования класса `ThreadFactory` разработано приложение ***Thread Manager***, которое позволяет производить запросы к классу `ThreadFactory`. Главное окно этого приложения приведено на рис. 43.

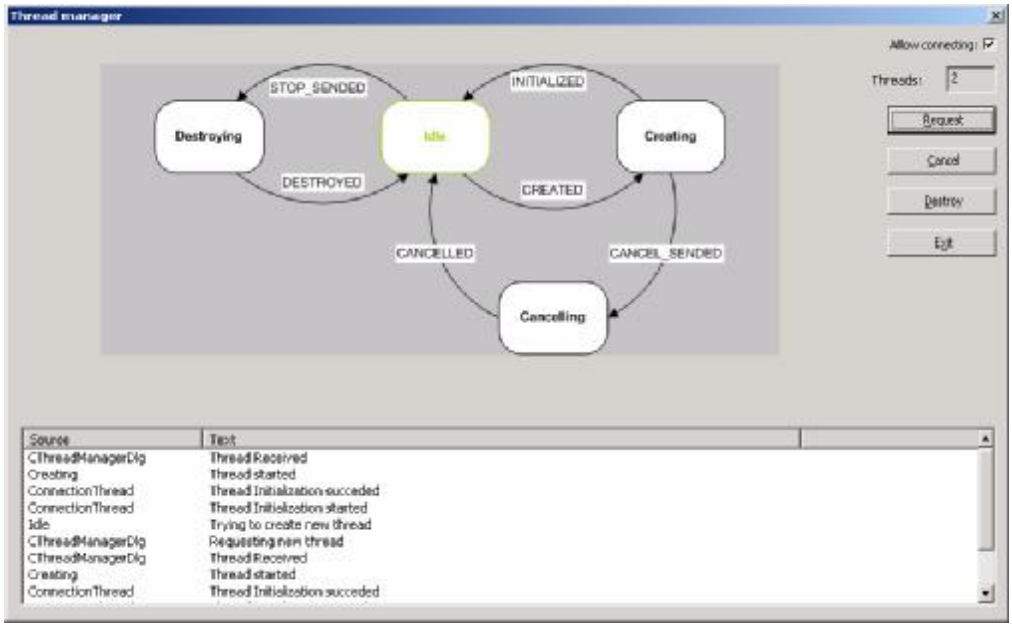

Рис. 44. Главное окно приложения *Thread Manager*

При нажатии на кнопки *Request*, *Cancel* и *Destroy* вызываются соответствующие методы класса `ThreadFactory`. Флажок *Allow connecting* эмулирует наличие/отсутствие сетевого соединения с серверной частью базы данных. В центральной части окна отображается граф переходов, визуализирующий текущее состояние автомата. В нижней части главного окна приложения отображается протокол работы программы. Программа завершается при нажатии на кнопку *Exit*.

5.5. Сравнение реализации класса *ThreadFactory* на основе паттерна *State Machine* и традиционного подхода

В разд. 5.3 описано проектирование и реализация класса `ThreadFactory` на основе паттерна *State Machine*.

При этом отметим, что обычно программисты выбирают другой способ реализации подобного класса — с использованием флагов. В этом случае код класса `ThreadFactory` выглядит иначе.

Заголовочный файл для класса `ThreadFactory`:

```
#pragma once

#include <set>
#include <afxmt.h>

#include "StateMachine\AutomatonBase.h"
#include ".\IThreadFactory.h"
#include ".\IThreadNotify.h"

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadRequest;
class ThreadFactory : public IThreadFactory, public
        IThreadNotify
{
public:
  ThreadFactory(IThreadRequest & request);
  ~ThreadFactory(void);

  // IThreadFactory
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
  virtual void Started(ConnectionThread * threadPtr);
  virtual void Stopped(ConnectionThread * threadPtr);

  // Wait for current action finish
  void Wait();

private:
  CCriticalSection lock_;
```

```
  CEvent requestEvent_;
  IThreadRequest & request_;
  ConnectionThread * threadPtr_;

  bool cancel_;
  bool destroy_;
};

}
```

Исполняемый файл для класса `ThreadFactory`:

```
#include "stdafx.h"

#include "boost\bind.hpp"
#include "StateMachine\Event.h"
#include "Logger.h"
#include ".\FactoryOld.h"
#include ".\ConnectionThread.h"
#include ".\IThreadRequest.h"
#include ".\Exceptions.h"

namespace ThreadManagement
{

ThreadFactory::ThreadFactory(IThreadRequest & request)
: requestEvent_(TRUE, TRUE),
  request_(request),
  threadPtr_(NULL),
  cancel_(false),
  destroy_(false)
{
}

ThreadFactory::~ThreadFactory(void)
{
  ASSERT(threadPtr_ == NULL);
```

```
}

void ThreadFactory::Request()
{
  CSingleLock locker(&lock_, TRUE);
  if (threadPtr_ != NULL) throw BusyException();

  Logger::Instance().Log("ThreadFactory", "Trying to
        create new thread");
  threadPtr_ = new ConnectionThread(*this);
  VERIFY(threadPtr_->CreateThread());
  VERIFY(requestEvent_.ResetEvent());
}

void ThreadFactory::Cancel()
{
  CSingleLock locker(&lock_, TRUE);
  if (cancel_ || destroy_ || threadPtr_ == NULL) throw
        CancelFailedException();

  cancel_ = true;
  threadPtr_->Stop();
}

void ThreadFactory::Destroy(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  if (threadPtr_ != NULL) throw BusyException();
  threadPtr_ = threadPtr;
  destroy_ = true;
  threadPtr_->Stop();
}

void ThreadFactory::Started(ConnectionThread * threadPtr)
{
```

```
  CSingleLock locker(&lock_, TRUE);
  if (cancel_) threadPtr_->Stop();
  else
  {
    ConnectionThread * tPtr = threadPtr_;
    threadPtr_ = NULL;
    request_.Created(tPtr);
    requestEvent_.SetEvent();
  }
}

void ThreadFactory::Stopped(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = NULL;
  if (cancel_)
  {
    Logger::Instance().Log("ThreadFactory", "Thread
        stopped. Mode cancelling");
    cancel_ = false;
    request_.Canceled();
  }
  else if (destroy_)
  {
    Logger::Instance().Log("ThreadFactory", "Thread
        stopped. Mode destroying");

    destroy_ = false;
    request_.Deleted();
  }
  VERIFY(requestEvent_.SetEvent());
}

void ThreadFactory::Wait()
{
```

```
    ::WaitForSingleObject(requestEvent_, INFINITE);
  }

  }
```

В приведенном коде присутствуют условные операторы, анализирующие значения флагов (логические переменные `cancel_` и `destroy_`). Такой код в работе [44] назван плохим, поскольку в разных методах класса эти флаги по-разному влияют на его поведение.

5.6. Сравнение реализации класса *ThreadFactory* на основе паттерна *State Machine* и *SWITCH*-технологии

В этом разделе опишем еще один способ проектирования класса `ThreadFactory` — на основе *SWITCH*-технологии. При этом отметим, что используется, так называемое, *оборачивание* автомата классом, примененное, например, в работе [67].

Схема связей автомата `ThreadFactory`, определяющая его интерфейс, приведена на рис. 45.

Рис. 45. Схема связей автомата **`ThreadFactory`**

Отметим, что для каждого метода интерфейса `IThreadFactoryAI` формируется определенное входное воздействие — событие. Например, методу `Request` соответствует событие `e0`.

Выходные воздействия реализованы с помощью закрытых методов класса `ThreadFactory`. Например, выходное воздействие `z0` создает поток соединения.

На рис. 46 приведен граф переходов структурного автомата, построенного на основе *SWITCH*-технологии.

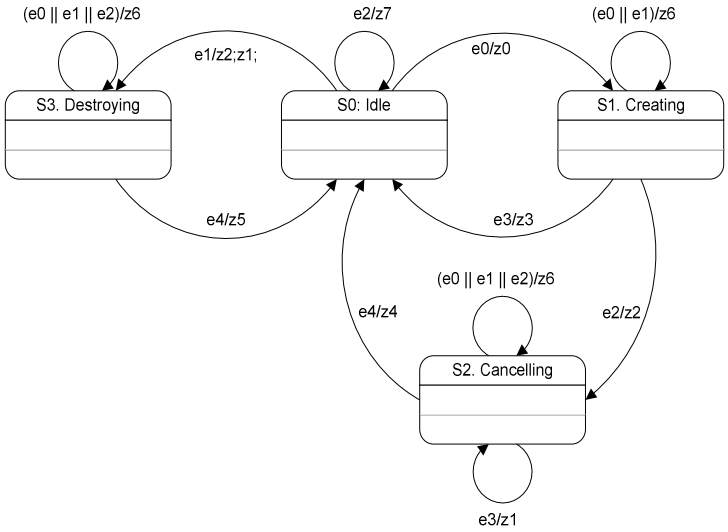

Рис. 46. Граф переходов структурного автомата

Приведем код класса `ThreadFactory`, реализующий описанный граф переходов.

Заголовочный файл для класса `ThreadFactory`:

```
#pragma once

#include <map>
#include <afxmt.h>

#include "boost\signal.hpp"
#include ".\IThreadFactory.h"
#include ".\IThreadNotify.h"
```

```
namespace ThreadManagement
{

class ConnectionThread;
struct IThreadRequest;
class ThreadFactory : public IThreadFactory, public
        IThreadNotify
{
public:
  ThreadFactory(IThreadRequest & request);
  ~ThreadFactory(void);

  // Add log listener
  template <class Func>
  void AddLogFunction(Func const & t)
  {
    log_.connect(t);
  }

  // Add State Change listener
  template <class Func>
  void AddStateChangeListener(Func const & t)
  {
    stateChange_.connect(t);
  }

  // IThreadFactory
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
  virtual void Started(ConnectionThread * threadPtr);
  virtual void Stopped(ConnectionThread * threadPtr);
```

```
  // Wait for current action finish
  void Wait();

private:
  CCriticalSection lock_;
  CEvent requestEvent_;
  IThreadRequest & request_;
  ConnectionThread * threadPtr_;

  enum States
  {
    s0, // Idle
    s1, // Creating
    s2, // Cancelling
    s3  // Destroying
  };

  States y;

  enum Events
  {
    e0, // Request
    e1, // Destroy
    e2, // Cancel
    e3, // Started
    e4  // Stopped
  };

  void A(Events e);

  void z0(); // Create new thread
  void z1(); // Destroy thread
  void z2(); // Reset event
```

```
  void z3(); // Notify created
  void z4(); // Notify cancelled
  void z5(); // Notify destroyed

  void z6(); // Throwing BusyException
  void z7(); // Throwing CancelFailedException

  std::map<States, std::string> stateNames_;
  boost::signal<void (char const *)> log_;
  boost::signal<void (char const *)> stateChange_;

  void Log(char const * message);
};
}
```

Исполняемый файл для класса `ThreadFactory`:

```
#include "StdAfx.h"

#include "Logger.h"
#include ".\FactorySwitch.h"
#include ".\ConnectionThread.h"
#include ".\IThreadRequest.h"
#include ".\Exceptions.h"

namespace ThreadManagement
{

ThreadFactory::ThreadFactory(IThreadRequest & request)
: requestEvent_(TRUE, TRUE),
  request_(request),
  threadPtr_(NULL),
  y(s0)
{
  stateNames_.insert(std::make_pair(s0, "Idle"));
  stateNames_.insert(std::make_pair(s1, "Creating"));
  stateNames_.insert(std::make_pair(s2, "Canceling"));
```

```
  stateNames_.insert(std::make_pair(s3, "Destroying"));
}

ThreadFactory::~ThreadFactory(void)
{
  ASSERT(threadPtr_ == NULL);
}

void ThreadFactory::Request()
{
  CSingleLock locker(&lock_, TRUE);
  A(e0);
}

void ThreadFactory::Destroy(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = threadPtr;
  A(e1);
}

void ThreadFactory::Cancel()
{
  CSingleLock locker(&lock_, TRUE);
  A(e2);
}

void ThreadFactory::Started(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = threadPtr;
  A(e3);
}

void ThreadFactory::Stopped(ConnectionThread * threadPtr)
```

```
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = threadPtr;
  A(e4);
}

void ThreadFactory::A(Events e)
{
  States yOld = y;
  switch (y)
  {
  case s0: // Idle
    {
      if (e == e0) {z0(); y = s1;}
      else if (e == e1) {z2(); z1(); y = s3;}
      else if (e == e2) z7();
      else ASSERT(FALSE);
    }
    break;
  case s1: // Requesting
    {
      if (e == e0 || e == e1) z6();
      else if (e == e2) {z2(); y = s2;}
      else if (e == e3) {z3(); y = s0;}
      else ASSERT(FALSE);
    }
    break;
  case s2: // Cancelling
    {
      if (e == e0 || e == e1 || e == e2) z6();
      else if (e == e3) z1();
      else if (e == e4) {z4(); y = s0;}
    }
    break;
  case s3: // Destroying
```

```
    {
      if (e == e0 || e == e1 || e == e2) z6();
      else if (e == e4) {z5(); y = s0;}
      else ASSERT(FALSE);
    }
    break;
  default:
    ASSERT(FALSE);
  };
  if (yOld != y) stateChange_(stateNames_[y].c_str());
}

void ThreadFactory::z0()
{
  ASSERT(threadPtr_ == NULL);
  Log("Trying to create new thread");
  threadPtr_ = new ConnectionThread(*this);
  VERIFY(threadPtr_->CreateThread());
  VERIFY(requestEvent_.ResetEvent());
}

void ThreadFactory::z1()
{
  Log("Trying to destroy thread");
  threadPtr_->Stop();
}

void ThreadFactory::z2()
{
  VERIFY(requestEvent_.ResetEvent());
}

void ThreadFactory::z3()
{
  Log("Thread started");
```

```
  ConnectionThread * tPtr = threadPtr_;
  threadPtr_ = NULL;
  request_.Created(tPtr);
  requestEvent_.SetEvent();
}

void ThreadFactory::z4()
{
  Log("Thread stopped");
  threadPtr_ = NULL;
  request_.Canceled();
  requestEvent_.SetEvent();
}

void ThreadFactory::z5()
{
  Log("Thread stopped");
  request_.Deleted();
  threadPtr_ = NULL;
  VERIFY(requestEvent_.SetEvent());
}

void ThreadFactory::z6()
{
  throw BusyException();
}

void ThreadFactory::z7()
{
  throw CancelFailedException();
}

void ThreadFactory::Log(char const * message)
{
```

```
  Logger::Instance().Log(stateNames_[y].c_str(),
        message);
}

void ThreadFactory::Wait()
{
  ::WaitForSingleObject(requestEvent_, INFINITE);
}
}
```

Обратим внимание, что при реализации класса `ThreadFactory` не используются глобальные переменные. В качестве возможных значений состояния используются не их номера, как в работе [67], а значения перечисляемого типа `States`, что повышает типобезопасность.

Сравним реализации на основе *SWITCH*-технологии и паттерна *State Machine*.

Достоинства реализации на основе *SWITCH*-технологии.

1. В графах переходов, используемых в этой технологии, наряду с состояниями и событиями применяются также входные переменные и выходные воздействия.
2. Код, реализующий граф переходов, строится формально и изоморфно.
3. Автомат реализуется в одном классе.
4. Код более компактен.
   4.1. В случае реализации автомата с больш*и*м количеством состояний для паттерна *State Machine* необходимо создавать по классу на каждое состояние, что может быть громоздко и трудоемко.
   4.2. Недостатками реализации на основе *SWITCH*-технологии.
5. Монолитность — в отличие от реализации на основе паттерна *State Machine* невозможно повторно использовать составные части кода класса `ThreadFactory`.

6. При необходимости добавления входных и (или) выходных воздействий могут возникать ситуации, при которых компилятор не сможет обнаружить некоторые семантические ошибки, такие как, например, несоответствие метода интерфейса класса с вызовом автомата с соответствующим событием.

Выводы

Проектирование и реализация управления потоками на основе паттерна *State Machine* были выполнены в процессе работы над проектом *Navi Harbour* в компании *Транзас*. Применение паттерна *State Machine* для разработки класса `ThreadFactory` позволило обеспечить хороший [44] дизайн программы.

1. Исключить условные операторы из текста программы.
2. Разработать классы состояний независимо друг от друга.
3. Обеспечить при необходимости возможность наследования от классов состояний.
4. Обеспечить читабельность кода, сохранив преимущества автоматного проектирования.

Приложение *Thread Manager*, предназначенное для визуализации и тестирования класса `ThreadFactory`, также разработано на языке программирования *C++* с использованием библиотек *Boost* и *MFC* [68].

Исходные и бинарные коды построенных программ доступны по адресу http://is.ifmo.ru, раздел «Статьи».

Отметим, что другие результаты работы также внедрены.

1. 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.

2. В разд. 4.2 приведены примеры применения языка *State Machine*. В настоящее время автором этот язык используется в учебном процессе кафедры «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсам «Программирование на языке *Java*» и «Применение автоматов в программировании». Во втором из указанных курсов излагаются паттерн *State Machine* и одноименный язык.

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.
    5.1. При разработке пакета *UniMod* в компании *eVelopers*.
    5.2. При разработке продукта *Navi Habour* в *департаменте береговых систем* компании *Транзас*
    5.3. В учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсам

«Теория автоматов в программировании» и «Программирование на языке *Java*».

Акты внедрения прилагаются.

Все основные результаты, полученные в диссертации, опубликованы. Кроме работ автора, на которые даны ссылки в тексте диссертации, им опубликованы также и работы [69-74]. Приведем информацию о личном вкладе автора.

В работах [69, 70] автором предложена идея метода преобразования рекурсивных программ в автоматные. Выполнена реализация ряда примеров на основе этого метода.

В работе [71] автор предложил один из методов оптимизации алгоритмов обхода доски. Он также выполнил ряд реализаций этого алгоритма.

В работах [52, 72] автором предложен язык автоматного программирования и его реализация.

В работах [73, 74] автором предложен обход двоичных деревьев на основе автоматного подхода. Окончательная версия этого обхода и обобщение на случай обхода *k*-ичных деревьев была произведена в соавторстве.

В работе [48] при разработке паттерна *State Machine* автором был предложен ряд реализаций, устраняющих недостатки паттерна *State*. Автором продемонстрирована эффективность этого паттерна по сравнению другими объектно-ориентированными реализациями.

Литература

1. **Гольдштейн Б. С.** Сигнализация в сетях связи. Том 1. М.: Радио и связь, 2001. — 439 с.
2. UML™ Resource Page. http://www.uml.org/
3. **Harel D.** Statecharts: A visual formalism for complex systems //Sci. Comput. Program. 1987. Vol.8. — P. 231-274
4. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. — 628 с.
5. **Gamma E., Helm R., Johnson R., Vlissides J.** Design Patterns. MA: Addison-Wesley Professional. 2001. — 395 (**Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. — 368 с).
6. Java technology. http://java.com/en/index.jsp
7. eVelopers Corporation. http://www.evelopers.com/about.html
8. UniMod. Eclipse plugin. http://unimod.sourceforge.net/.
9. Transas — a world-leading developer and supplier of a wide range of software and integrated solutions for the transportation industry (http://www.transas.com).
10. **Шалыто А.А. Туккель Н.И.** Реализация вычислительных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2001. № 6. http://is.ifmo.ru, раздел «Статьи».
11. **Steling S., Maassen O.** Applied Java Patterns. Pearson Higher Education. 2001, P. 608 (**Стелтинг С., Массен О**. Применение шаблонов Java. Библиотека профессионала. М.: Вильямс, 2002. — 564 с).

12. **Grand M.** Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. Wiley, 2002. — 544 p. (**Гранд М.** Шаблоны проектирования в Java. М.: Новое знание, 2004. — 560 с).
13. Abstract State Machine Language. http://research.microsoft.com/fse/asml/
14. **Кафедра «Технологии программирования»**. http://is.ifmo.ru.
15. **Eclipse**. http://www.eclipse.org/.
16. **Седжвик Р.** Фундаментальные алгоритмы на C++. Киев: ДиаСофт, 2001.
17. **Шалыто А.А., Туккель Н.И.** От тьюрингова программирования к автоматному //Мир ПК. 2002. №2. http://is.ifmo.ru, раздел «Статьи».
18. **Стивенс Р.** Delphi. Готовые алгоритмы. М.: ДМК, 2001.
19. **Ахо А., Хопкрофт Д., Ульман Д.** Структуры данных и алгоритмы. М.: Вильямс, 2000.
20. **Бобак И.** Алгоритмы: «возврат назад» и «разделяй и властвуй» //Программист. 2002. №3.
21. **Грэхем Р.**, Кнут Д., Поташник О. Конкретная математика. М.: Мир, 1998.
22. **Анисимов А.В.** Информатика. Творчество. Рекурсия. Киев: Наукова думка, 1988.
23. **Быстрицкий В.Д.** Ханойские башни. http://alglib.chat.ru/paper/hanoy.html
24. **Буч Г.** Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, СПб.: Невский диалект, 1998.
25. **Бобак И.** Алгоритмы: «возврат назад» и «разделяй и властвуй» //Программист. 2002. №3.
26. **Гарднер М.** Математические новеллы. М.: Мир, 1974.
27. **Бобак И.** Алгоритмы: AI-поиск //Программист. 2002. №7.
28. **Гик Е.** Шахматы и математика. М.: Наука, 1983.

29. **Вирт Н.** Алгоритмы + структуры данных = программы. М.: Мир: Мир, 1985.
30. **Туккель Н. И., Шамгунов Н. Н., Шалыто А. А.** Ханойские башни и автоматы //Программист. 2002 № 8. http://is.ifmo.ru, раздел «Статьи».
31. **Кормен Т., Лейзерсон Ч., Ривест Р.** Алгоритмы: построение и анализ. М.: Центр непрер.матем. образования, 2000.
32. **Шень.** А. Программирование. Теоремы и задачи. М.: Центр непрер.матем. образования, 2004.
33. **Кнут Д.** Искусство программирования. Т. 1. Основные алгоритмы. М:.: Вильямс, 2003.
34. **Касьянов В. Н., Евстигнеев В. А.** Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003.
35. **Шалыто А.А., Туккель Н.И.** Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. №5. с. 12-26. http://is.ifmo.ru, раздел «Статьи»
36. **Брукшир Дж.** Введение в компьютерные науки. М.: Вильямс, 2001.
37. Java Data Objects (JDO). http://java.sun.com/products/jdo/index.jsp.
38. **Eliens A.** Principles of Object-Oriented Software Development. MA.: Addison-Wesley, 2000. — 502 p. (**Элиенс А.** Принципы объектно-ориентированной разработки программ. М.: Вильямс, 2002. — 496 с).
39. **Sandén B.** The state-machine pattern // Proceedings of the conference on TRI-Ada '96 http://java.sun.com/products/jdo/index.jsp.
40. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns. http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf
41. **Odrowski J., Sogaard P.** Pattern Integration — Variations of State // Proceedings of PLoP96. http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz.

42. **Sane A., Campbell R.**. Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity // OOPSLA ’95. http://choices.cs.uiuc.edu/sane/home.html.
43. **Aho A., Sethi R., Ullman J.** Compilers: Principles, Techniques and Tools. MA: Addison-Wesley, 1985, 500 p. (**Ахо А., Сети Р., Ульман Дж.** Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001. —768 с).
44. **Fawler M.** Refactoring. Improving the Design of Existing Code. MA: Addison-Wesley. — 1999. — 431 p. (**Фаулер М.** Рефакторинг. Улучшение существующего кода. — М.: Символ-плюс, 2003. — 432 с).
45. **Martin R.** Three Level FSM // PLoPD, 1995. http://cpptips.hyperformix.com/cpptips/fsm5.
46. Раздел «Статьи» сайта кафедры «Технологии программирования» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (http://is.ifmo.ru/articles).
47. The Self Language. (http://research.sun.com/self/language.html).
48. **Шамгунов Н. Н., Корнеев Г А. Шалыто А. А.** Паттерн *State Machine* для объектно-ориентированного проектирования автоматов // Информационно-управляющие системы. 2004. № 5, с. 13 — 25
49. **Эндрю Х., Дэвид Т.** Программист-прагматик. М.: Лори, 2004. — 288 с.
50. Язык прикладного программирования STL 1.0. Руководство пользователя (v1.0.13b). http://lmt-automation.ifmo.ru/pdfs/stlguide_1_0_13b.pdf
51. **Ваганов С. А.** FloraWare — ускорить разработку приложений. http://www.softcraft.ru/paradigm/oop/flora/index.shtml
52. **Шамгунов Н. Н., Шалыто А. А.** Язык автоматного программирования с компиляцией в *Microsoft CLR*. // Microsoft Research Academic Days in St. Petersburg, 2004.

53. http://msdn.microsoft.com/vcsharp/team/language/default.aspx C# Language.
54. **Gosling J., Joy B., Steele G., Bracha G.** The Java Language Specification. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.
55. **Appel A. W.** Modern Compiler Implementation in Java. — NY, Cambridge, 1998. — 512 c.
56. **Green A.** Trail: The Reflection API. http://java.sun.com/d1ocs/books/tutorial/reflect/
57. Generics. http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html
58. **Naur P. et al**. Revised Report on the Algorithmic Language ALGOL 60 //Communications of the ACM. 1960. Vol. 3. No.5, pp. 299-314.
59. *Object-Oriented Programming Concepts* http://java.sun.com/docs/books/tutorial/java/concepts/
60. *Open Source License.* http://www.opensource.org/licenses/historical.php
61. http://www.cs.princeton.edu/~appel/modern/java/ Modern Compiler Implementation in Java.
62. **Шалыто А. А., Туккель Н. И.** SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5. С. 45-62. (http://is.ifmo.ru, раздел «Статьи»).
63. **Vessel Traffic Services/ Navi-Harbour** (http://www.transas.com/vts/navi_harbour/index.asp).
64. Shore-base systems department (http://www.transas.com/vts/index.asp).
65. **Straustrup B.** The C++ Programming Language. MA: Addison-Wesley, 2000, 957p. (**Страуструп Б**. Язык программирования C++. СПб.: Бином, 2001. — 1099 с).
66. Boost (http://www.boost.org)
67. **Туккель Н.И., Шалыто А.А.** Система управления танком для игры "Robocode". Вариант 1. Объектно-ориентированное программирование с явным выделением состояний. (http://is.ifmo.ru/projects/tanks/)

68. Microsoft Foundation Classes **(**http://microsoft.com)
69. **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
70. **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»)
71. **Туккель Н. И., Шамгунов Н. Н, Шалыто А. А.** Задача о ходе коня //Мир ПК 2003. №1. http://is.ifmo.ru, раздел «Статьи».
72. **Шамгунов Н.Н.**, **Шалыто А.А.** Язык автоматного программирования *State* //Труды Всероссийской научно-методической конференции «Телематика-2004». СПб.: СПбГУ ИТМО. 2004, с. 180-181.
73. **Корнеев Г. А., Шамгунов Н. Н., Шалыто А. А.** Обход деревьев на основе автоматного подхода. Полная версия статьи с приложением, опубликованная на сайте http://is.ifmo.ru, раздел «Статьи».
74. **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.
75. **Mealy G.** A Method for Synthesizing Sequential Circuits //Bell System Technical Journal. 1955. V.34. № 5. — P.1045-1079.
76. **Moore E.** Gedanken Experiments on Sequential Machines //В [6]. — P. 129-153.
77. Automata Studies //Ed. Shannon C.E., McCarthy J. Princeton Univ. Press, 1956. — P. 400 (Автоматы //Ред. Шеннона К.Э., МакКарти Дж. М.: Изд-во иностр. лит., 1956. — 451 с).

78. **Rubin M.**, **Scott D.** Finite automata and their decision problem //IBM J. Research and Development. 1959. V.3. № 2. — P. 115-125 (Кибернетический сборник. Вып.4. М.: Изд-во иностр. лит., 1962).
79. **Глушков В. М.** Синтез цифровых автоматов. М.: Изд-во физ.-мат. лит., 1962. — 476 с.
80. **Kleene S. C.** Representation of Events in Nerve Nets and Finite Automata //В работе [77]. — P. 3–41.
81. **Thompson K.** Regular expression Search Algorithm //Communications of the ACM. 1966. V.11. № 6. — P. 419-422.
82. **Hopcroft J., Motwani R., Ullman J.** Introduction to Automata Theory, Languages and Computation. MA: Addison-Wesley, 2001. — 521 p. (**Хопкрофт Д., Мотвани Р., Ульман Дж.** Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2001. — 528 с).