

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

УДК 681.3.06: 62-507

ВГК ОКП

№ регистрационный 01.20.03 00 802

Инв. №

"УТВЕРЖДАЮ"

Ректор СПбГУИТМО

докт. техн. наук, профессор

_____ Васильев В.Н.

ИТОГОВЫЙ ОТЧЕТ

по научно-исследовательской работе №30117, выполненной по
гранту Российского фонда фундаментальных исследований по
проекту 02-07-90114

"Разработка технологии автоматного программирования"

Этап 2

"Расширение области использования
автоматного программирования"

Руководитель НИР
докт. техн. наук,
профессор,
заведующий кафедры
"Информационные системы"

Шалыто А.А.

Санкт-Петербург
2003

Список основных исполнителей

Руководитель НИР

Доктор технических наук,
профессор,
заведующий кафедры
"Информационные системы"

Шалыто А.А.

Исполнители

Инженер-программист

Туккель Н.И.

Студент

Наумов Л.А.

Аспирант

Шамгунов Н.А.

Аспирант

Шопырин Д.

Аспирант

Гуров В.С.

Аспирант

Казаков М.А.

Студент

Макаров Н.С.

Студент

Крылов Р.А.

Студент

Григорьев А.Э.

Студент

Мазин М.А.

Студент

Корнеев Г.А.

Студент

Станкевич А.С.

Студент

Наумов А.С.

Студент

Штучкин А.Н.

Реферат

Отчет содержит 162 страницу, 70 рисунков, 4 таблицы, 90 источников литературы.

Система управления, вычислительный алгоритм, схема алгоритма, состояние, автомат, граф переходов, протокол, алгоритмизация, автоматное программирование, проектирование программ, объектно-ориентированное программирование, итеративные программы, рекурсивные программы, автоматные программы

Целью настоящего этапа работы является разработка подходов к совместному использованию автоматного и объектно-ориентированного стилей программирования. Эти подходы могут быть названы «объектно-ориентированное программирование с явным выделением состояний». Кроме того, в работе предлагаются методы преобразования итеративных и рекурсивных программ в автоматные.

В первой части работы подробно рассмотрены два подхода:

- реализация автоматов, как методов классов;
- реализация автоматов, как классов.

Показано, что первый из этих подходов позволяет:

- локализовать в методах классов их поведение;
- проводить автоматическое протоколирование, описывающее работу всех автоматов в терминах состояний, переходов, входных и выходных воздействий с указанием соответствующих объектов;
- упрощает понимание проектной документации.

Второй подход позволяет в **полной мере** совместить гибкость объектно-ориентированного программирования с наглядностью и ясностью автоматного подхода.

Предложена методика преобразования объектной программы, написанной в рамках второго подхода, в процедурную программу для выполнения ее на микроконтроллере.

Кроме указанных подходов в отчете перечислен также ряд других методов совместного использования автоматного и объектно-ориентированного стилей программирования. Эти методы подробно рассмотрены в рамках соответствующих проектов, опубликованных на сайте <http://is.ifmo.ru> в разделе «Проекты».

Во второй части отчета предложены методы преобразования итеративных и рекурсивных программ в автоматные.

Содержание

| | |
|--|----------|
| Введение | 7 |
| ЧАСТЬ I. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С ЯВНЫМ ВЫДЕЛЕНИЕМ СОСТОЯНИЙ | 8 |
| 1. Реализация автоматов, как методов классов | 8 |
| 1.1. Автоматы | 8 |
| 1.2. Особенности технологии | 13 |
| 1.3. Предлагаемый подход | 15 |
| 1.4. Танки | 18 |
| 1.5. Результаты сражений | 24 |
| 1.6. Сравнение с UML | 25 |
| 1.7. Листинги | 29 |
| 1.7.1. Фрагмент программы, реализующей класс «Стрелок» | 29 |
| 1.7.2. Фрагмент протокола боя двух танков | 32 |
| 1.8. Рефакторинг | 35 |
| 1.9. Заключительные замечания к главе 1 | 36 |
| 2. Реализация автоматов, как классов | 38 |
| 2.1. Особенности объектно-ориентированного программирования с явным выделением состояний | 40 |
| 2.2. Формулировка задачи о лифте | 43 |
| 2.3. Проектирование автоматов и классов | 49 |
| 2.4. Описание базового класса <code>Automat</code> и вспомогательных макроопределений | 54 |
| 2.5. Разработка автоматов – потомков класса <code>Automat</code> | 58 |
| 2.6. Интерфейс программы | 59 |
| 2.7. Переход от объектного варианта программы к процедурному | 65 |
| 2.8. Выводы по главе 2 | 68 |
| 2.9. Листинги реализации автоматов | 70 |
| 2.9.1. Файл <code>Automat.h</code> – Заголовочный файл базового класса | 70 |
| 2.9.2. Файл <code>Automat.cpp</code> – Файл реализации базового класса | 72 |
| 2.9.3. Файл <code>A0.h</code> – Заголовочный файл класса автомата <code>A0</code> | 73 |
| 2.9.4. Файл <code>A0.cpp</code> – Файл реализации класса автомата <code>A0</code> | 74 |
| 2.9.5. Файл <code>A1.h</code> – Заголовочный файл класса автомата <code>A1</code> | 80 |
| 2.9.6. Файл <code>A1.cpp</code> – Файл реализации класса автомата <code>A1</code> | 81 |
| 2.9.7. Файл <code>A2.h</code> – Заголовочный файл класса автомата <code>A2</code> | 84 |
| 2.9.8. Файл <code>A2.cpp</code> – Файл реализации класса автомата <code>A2</code> | 85 |
| 3. Другие подходы к объектно-ориентированному программированию с явным выделением состояний | 87 |

ЧАСТЬ II. ПРЕОБРАЗОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ В АВТОМАТНЫЕ .93

| | | |
|--------|--|-----|
| 4. | Преобразование итеративных алгоритмов в автоматные программы | 93 |
| 4.1. | Изложение метода | 95 |
| 4.2. | Примеры преобразования итеративных программ в автоматные | 97 |
| 4.3. | Преобразование неструктурированных алгоритмов в автоматные программы | 105 |
| 5. | Преобразование программ с явной рекурсией в автоматные | 112 |
| 5.1. | Изложение метода | 112 |
| 5.2. | Вычисление факториала | 115 |
| 5.3. | Вычисление чисел Фибоначчи | 119 |
| 5.4. | Задача о ранце | 123 |
| 5.5. | Задача о ханойских башнях | 131 |
| 5.5.1. | Классическое рекурсивное решение задачи | 132 |
| 5.5.2. | Моделирование рекурсии автоматной программой | 135 |
| 5.5.3. | Реализация рекурсивной программы автоматнo-рекурсивной программой | 141 |
| 5.5.4. | Реализация автоматной программы с использованием классов | 142 |
| 5.5.5. | Обход дерева действий | 145 |
| 5.5.6. | Решение задачи в терминах "объекта управления" | 151 |
| 6. | Выводы по II части | 155 |
| 7. | Публикации по результатам этапа | 157 |
| 8. | Список литературы | 161 |

Введение

В первой части настоящего отчета излагаются основы объектно-ориентированного программирования с явным выделением состояний.

В отчете подробно рассмотрены два подхода:

- реализация автоматов, как методов классов;
- реализация автоматов, как классов.

Показано, что первый из этих подходов позволяет локализовать в методах классов их поведение, проводить автоматическое протоколирование, описывающее работу всех автоматов в терминах состояний, переходов, входных и выходных воздействий с указанием соответствующих объектов, а также упрощает понимание проектной документации.

Второй подход позволяет в **полной мере** совместить гибкость объектно-ориентированного программирования с наглядностью и ясностью автоматного подхода.

Предложена методика преобразования объектной программы, написанной в рамках второго подхода, в процедурную программу для выполнения ее на микроконтроллере.

Кроме указанных подходов в отчете перечислен также ряд других методов совместного использования автоматного и объектно-ориентированного стилей программирования. Эти методы подробно рассмотрены в рамках соответствующих проектов, опубликованных на сайте <http://is.ifmo.ru> в разделе «Проекты».

Во второй части отчета предложены методы преобразования итеративных и рекурсивных программ в автоматные.

ЧАСТЬ I. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С ЯВНЫМ ВЫДЕЛЕНИЕМ СОСТОЯНИЙ

1. Реализация автоматов, как методов классов

При создании объектно-ориентированных программ [1,2] в литературе все большее внимание уделяется их проектированию [3–5]. Однако на практике с проектированием программы обычно «мучаются не долго, и поэтому мучаются всю жизнь» на остальных этапах жизненного цикла программы. Это во многом связано с недостаточной эффективностью известных технологий проектирования. В настоящем разделе делается попытка разработки технологии проектирования программ рассматриваемого класса на основе минимального набора документов, описывающих как структурные, так и поведенческие стороны программ.

1.1. Автоматы

При объектном проектировании для описания поведения объектов наряду с другими моделями [6,7] используется модель конечного детерминированного автомата, которую в дальнейшем будем называть «автоматом».

Эта модель, в отличие от применяемой в трансляторах [8], содержит выходы, на каждом из которых реализуется функция (подпрограмма, выполняющая определенные действия) [9,10]. Это резко расширяет класс задач [11], в котором могут быть применены автоматы. Более того, в работе [12] для описания поведения управляющих «устройств» было предложено использовать не одиночные автоматы, а системы

взаимосвязанных автоматов, которые обладают еще более широкими функциональными возможностями, например, в части реализации параллельных процессов. Автоматы в системе могут взаимодействовать на основе вложенности, вызываемости или обмена номерами состояний.

В автоматах в качестве входных воздействий могут использоваться:

- переменные, являющиеся функциями, возвращаемые значения которых и есть значения этих переменных;
- предикаты, проверяющие номера состояний других автоматов, взаимодействующих с рассматриваемым автоматом;
- события, обработчики которых вызывают функции, реализующие автоматы, передавая им номера этих событий, что кратко можно сформулировано как «события, с которыми вызываются автоматы» [9,10].

При этом отметим, что в общем случае события могут быть как внешними, так и внутренними.

Из изложенного следует, что в рамках предлагаемого подхода объединяются такие стили программирования, как «программирование от состояний» и «программирование от событий» (Непейвода Н.Н., Скопин И.Н. Основания программирования. Москва – Ижевск: Институт компьютерных исследований, 2003).

Изложенное опровергает сложившееся мнение о том, что автоматы имеют ограниченную область применения, например такую, как распознавание регулярных языков [8]. Даже если область применения автоматов и ограничена, то при изложенной выше их трактовке, она может простираться на неограниченный класс задач, связанных, по крайней мере, с управлением, понимаемым весьма широко. К такому классу принадлежит и задача, рассматриваемая в настоящей главе.

На практике автоматы обычно задаются визуально в виде графов переходов (диаграмм состояний), причем существуют различные их модификации, например, предложенная в работе [13] и применяемая в работе [7]. В настоящей главе используются графы переходов смешанных автоматов (С-автоматов), нотация для построения которых предложена в работах [10,11].

В работах [3–7] вопрос о реализации автоматов в рамках применяемых методологий в должной мере не рассматривался. В работах [14,15] приведены примеры реализации автоматов в рамках объектной парадигмы, причем в первом случае используется компилятивный подход, на основе которого для каждого события записывается оператор `switch`, реализующий инициируемые этим событием переходы, что, по нашему мнению, делать нецелесообразно, а во втором – интерпретационный, применение которого ограничено избыточностью, присущей интерпретации [16].

В работе [17] было предложено, используя компилятивный подход, выполнять реализацию не для отдельных событий, а для графов переходов в целом. При этом граф переходов предлагается реализовывать не только формально, но и изоморфно, что достигается применением одного или двух операторов `switch` [12] или аналогов этого оператора. Указанный подход обеспечивает изобразительную эквивалентность спецификации и реализации, что позволяет начинать проверку этой части программы со сравнения ее текста с графом переходов.

Для событийных («реактивных») систем в работах [9,10] был предложен образец (шаблон), состоящий из двух операторов `switch`, предназначенный для стандартной реализации графов переходов смешанных автоматов, которые строятся на основе указанной выше нотации. В этой связи следует обратить внимание на развиваемый в настоящее

время в рамках объектной парадигмы подход — проектирование по образцам. Так, в частности, в работе [18] описан паттерн «State», обеспечивающий построение программ с децентрализованной логикой. При этом отметим, что нами развивается в некотором смысле противоположный подход [19], который направлен на повышение централизации логики в программах. Это связано с тем, что при «сильно» распределенном управлении понять поведение программы по ее тексту практически невозможно.

На базе изложенного, используя процедурный (неклассовый) подход, в работе [12] был предложен вариант технологии создания программного обеспечения для систем логического управления. Эта технология была названа «SWITCH-технология» [20]. В дальнейшем был создан ее второй вариант, который предназначен для построения событийных систем [10] и назван в работе [9] «программирование с явным выделением состояний». Этот вариант охватывает все этапы создания программного обеспечения, включая сертификацию и документирование.

В настоящей главе предлагается еще один вариант указанной технологии, предназначенный для объектно-ориентированного программирования (проектирования), который может быть назван «объектно-ориентированное программирование с **явным** выделением состояний».

Такое название, несмотря на использование автоматов в указанных выше и в некоторых других методологиях объектной разработки программ [21], не характерно для объектной парадигмы. Так в работе [22], одним из критериев объектно-ориентированного языка является «поддержание им объектов как абстракции данных с определенным интерфейсом поименованных операций и **скрытым** состоянием». Да и само определение объекта, предложенное одним из «трех друзей» (являющихся создателями объектно-

ориентированного **проектирования**) А. Джекобсоном, в этом смысле мало что проясняет: «Объект — это сущность, способная сохранять свое состояние (информацию) и обеспечивающая набор операций (поведение) для проверки и изменения этого состояния» [23]. Кроме того, Г. Буч (другой «пророк-коммерсант» [24]) пишет: «... поведение объекта определяется его историей: важна последовательность совершаемых над объектом действий. Это объясняется тем, что у объекта есть внутреннее состояние, которое характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущим (обычно динамическими) значениями каждого из этих свойств» [4].

Определение поведения как набора операций (методов) является традиционным в объектно-ориентированном программировании [21]. Оно сдерживает применение автоматов в рамках этой парадигмы программирования, несмотря на приведенное выше мнение Г. Буча о поведении объекта как **последовательности** действий. Это определение не позволяет конструктивно использовать автоматы, так как при большом числе свойств (атрибутов) количество состояний объекта может быть огромным, что делает невозможным выделение состояний в явном виде. Не лучше складывается ситуация с использованием автоматов и при определении состояния программы как совокупности значений ее переменных [25].

В заключение раздела отметим, что различие между алгоритмической и объектной декомпозициями имеет место, но оно не столь принципиально, как утверждает Г. Буч в [3,4]. В работах [12,26] показано, что возможно такое построение схем алгоритмов (за счет введения переменных состояния), что они будут изоморфны конструкции `switch` и графам переходов автоматов. Автоматы, в соответствии с приведенным выше определением объекта, сами могут

рассматриваться как объекты, либо использоваться для описания их поведения.

1.2. Особенности технологии

Указанные в предыдущем разделе проблемы снимаются, если:

- атрибуты объекта также, как и память в машине Тьюринга или в машине фон Неймана, разделить на управляющие и остальные;
- явно выделять состояния в управляющей части объекта, так как число «управляющих» (автоматных) состояний обычно значительно меньше числа остальных состояний, например, «вычислительных». При этом находясь в одном из «управляющих» состояний, объект может пройти множество «вычислительных» состояний;
- рассматривать процессы [27] как последовательности смены состояний, разделяя их на управляющие и остальные. При этом в последних состояниях обычно задаются неявно;
- ввести в программирование понятие «пространство состояний», понимая под ним множество состояний, описывающих поведение по крайней мере одного автомата. Ориентация в этом пространстве обеспечивает существенно более понятное поведение по сравнению со случаем, когда такое пространство или ориентация в нем отсутствует;
- при спецификации задачи рассматривать автоматные состояния в качестве абстракций, переходя к переменным, им соответствующим, только на этапе реализации. Это позволяет Программисту перестать быть Фокусником, отказавшись от неупорядоченного

введения «флагов» при программировании. Программы с флагами, а не с явно выделенными состояниями, также «неустойчивы», как и слоны с тонкими ножками, изображенные, например, на картине Сальвадор Дали «Искушение святого Антония», по сравнению с нормальными слонами. Однако слоны с тонкими ножками, в отличие от программ с флагами, не встречаются повсеместно;

- программировать «через состояния», а не «через переменные» (флаги), что позволяет лучше понять и специфировать задачу и ее составные части;
- ввести в этап реализации программ, называемый традиционно «кодирование», новый подэтап, называемый в теории автоматов «кодирование состояний». Кодирование состояний необходимо для их различия, особенно в тех случаях, когда в разных состояниях формируются одинаковые выходные воздействия;
- ввести многозначное кодирование для каждого автомата с целью различия его состояний по значениям **одной** переменной;
- ввести в программирование понятие «**наблюдаемость**», обеспечивая возможность слежения за переходами каждого автомата только по одной многозначной переменной;
- «привязать» выходные воздействия к переходам, петлям или состояниям автомата с целью представления в компактном виде большого числа последовательностей действий, которые являются реакциями на соответствующие входные воздействия;
- обеспечить протоколирование **в терминах автоматов** с целью проверки корректности построенной программы,

представляющей собой систему взаимосвязанных объектов.

Необходимо отметить, что единственное ограничение, накладываемое на подход, разработанный в работах [9,10] для событийных систем, при его применении в объектно-ориентированном программировании, заключается в том, что для сохранения принципа инкапсуляции обмен номерами состояний должен производиться только между автоматами, принадлежащими одному классу.

1.3. Предлагаемый подход

Также как и при использовании любого другого подхода, применение предлагаемого связано со множеством эвристик, возвратов назад, уточнений и параллельно выполняемых работ. Однако, после завершения создания программы, предлагаемый подход может быть сформулирован (по крайней мере, для полного ее документирования) как «идеальная» технология, фиксирующая принятые решения.

1. На основе анализа предметной области выделяются классы и строится диаграмма классов, отражающая, в основном, наследование и агрегирование (например, вложенность).

2. Для каждого класса разрабатывается словесное описание, по крайней мере, в форме перечня решаемых задач.

3. Для каждого класса создается его структурная схема, несколько напоминающая карту CRC (Class – Responsibility – Collaboration, Класс – Ответственность – Кооперация) [28]. Нотация, используемая при построении структурных схем классов, приведена на рис. 1. Отметим, что в общем случае классу нет необходимости знать о вызывающих его объектах, а все вызываемые объекты не

обязательно должны быть указаны, как, впрочем, и стрелки на концах линий.

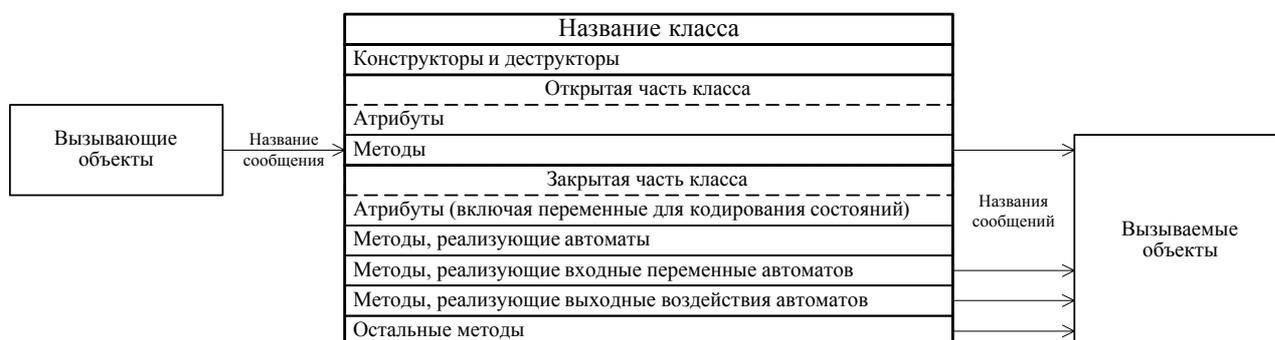


Рис. 1. Нотация, используемая при построении структурных схем классов

Интерфейс класса образуют открытые (`public`) атрибуты и методы, к которым обращаются другие объекты. Эти методы, в свою очередь, вызывают закрытые (`private`) методы рассматриваемого класса.

Как открытые, так и закрытые методы, в общем случае могут передавать сообщения другим объектам.

Закрытые методы могут быть разделены на две части: автоматные и остальные. Автоматные методы в общем случае делятся на три разновидности:

- методы, реализующие автоматы;
- методы, реализующие входные переменные автоматов;
- методы, реализующие выходные воздействия автоматов.

4. При наличии в классе нескольких автоматов строится схема их взаимодействия [10].

5. Для каждого класса составляются перечни событий, входных переменных и выходных воздействий.

6. Для каждого автомата разрабатывается словесное описание.

7. Для каждого автомата, используя правила, изложенные в работе [9,10], строится схема связей, определяющая его интерфейс, входные воздействия которого состоят из входных переменных, предикатов, проверяющих номера состояний и событий. В этой схеме в качестве событий (наряду с другими) могут быть указаны сообщения, получаемые объектом и приведенные в структурной схеме класса.

В схеме связей показываются все события, с которыми автомат запускается (вызывается), вне зависимости от того, используются ли они в пометках дуг и петель графа переходов.

В отличие от структурной схемы класса, на входах и выходах в схеме связей автомата указываются не названия сообщений, а названия соответствующих входных и выходных воздействий.

8. Для каждого автомата на основе нотации, приведенной в работах [9, 10], строится граф переходов.

9. Для каждого класса разрабатывается соответствующая ему часть программы в целом. Ее структура должна быть изоморфна структурной схеме класса, а методы, соответствующие автоматам, реализованы по шаблону, приведенному в работе [9, 10]. При этом методы, соответствующие входным переменным и выходным воздействиям автомата, располагаются отдельно от метода, реализующего автомат, из которого они только вызываются.

Благодаря такой реализации, методы, соответствующие входным переменным и выходным воздействиям автомата, могут быть абстрактными или полиморфными, и переопределяться в порождаемых классах. Таким образом, имеется возможность создать базовый класс для управления некоторой группой устройств, в котором реализуются только автоматы, а используемые ими входные переменные и

выходные воздействия переопределяются на уровне порождаемых классов, управляющих конкретными устройствами.

10. Для изучения поведения программы, определяемого, в том числе, и взаимодействием объектов, а в ходе разработки – для ее отладки, **автоматически** строятся протоколы, описывающие работу всех автоматов в терминах состояний, переходов, входных и выходных воздействий с указанием соответствующих объектов. Это обеспечивается за счет включения функций протоколирования в соответствующие методы. При необходимости могут протоколироваться также и аналоговые параметры. Тот факт, что входные и выходные воздействия «привязаны» к объектам, автоматам и состояниям упрощает понимание таких протоколов по сравнению с протоколами, которые строятся традиционно. Из рассмотрения протоколов следует, что автоматы (в отличие от классов) абстракциями не являются. При этом можно утверждать, что автоматы структурируют поведение, которое с их помощью описывается весьма компактно и строго.

11. Выпускается проектная документация.

1.4. Танки

Предлагаемая технология иллюстрируется на примере игры для программистов «Robocode», целью которой является создание виртуальных роботов, обладающих средствами обнаружения и уничтожения противника. Каждая из подобных игр имеет специфику, определяемую средой исполнения, правилами проведения боев и используемым языком программирования.

Исходя из целей настоящей главы, нас интересовала такая игра, которая в качестве языка программирования использовала бы не специализированный язык, а один из широко распространенных объектно-ориентированных языков.

Единственной известной авторам игрой, отвечающей этому требованию, является недавно (в 2001 году) разработанная компанией «Alphaworks» игра «Robocode» [29].

Эта игра позиционируется фирмой IBM как средство для обучения программированию на языке Java на примере создания программы, являющейся системой управления танком, предоставляемым средой исполнения. Отметим, что использование интерпретируемого языка в играх данного жанра дает возможность легко и быстро внедрять разработанную программу. На рис. 2 приведен внешний вид среды исполнения в процессе боя двух танков.

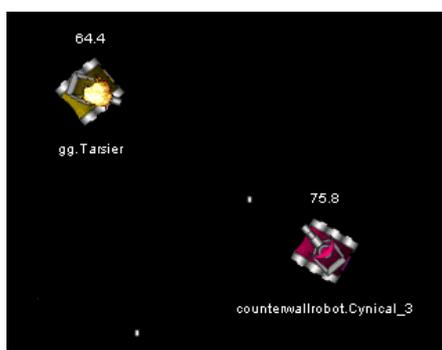


Рис. 2. Фрагмент боя

Перейдем к описанию процесса создания программы в соответствии с предлагаемой технологией.

1. Первым разрабатываемым документом является диаграмма классов, в верхней части которой расположены предоставляемые разработчику классы (в том числе из стандартной библиотеки) и среда исполнения (рис. 3).

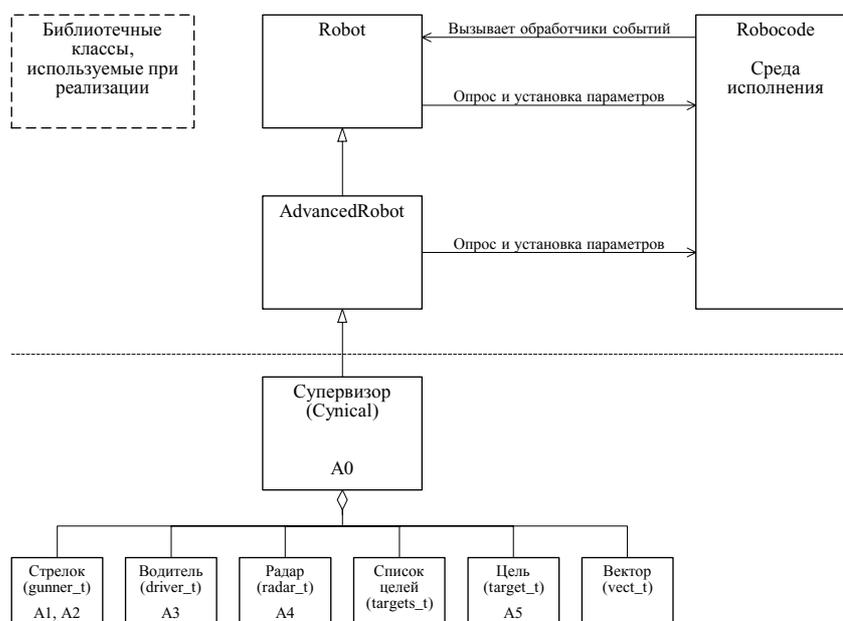


Рис. 3. Диаграмма классов

Среда формирует определенные правилами игры события и вызывает их обработчики, объявленные в классе «Robot». Эти обработчики наследуются в классе «AdvancedRobot». Указанные классы реализуют вызываемые из разрабатываемой системы управления методы, обеспечивающие непосредственное управление танком за счет опроса и установки его параметров в среде исполнения. К такой разновидности методов относится, например, метод `turnRight()`, который поворачивает танк направо.

В соответствии с требованиями программного интерфейса среды исполнения к структуре программы, она должна представлять собой один класс, порожденный от класса «Robot» или, как в данном случае, от класса «AdvancedRobot». Название содержащего программу пакета «counterwallrobot» вместе с названием головного класса образуют название танка. Головной класс «Супервизор» в программе имеет имя «Cynical», и содержит шесть внутренних (вложенных) классов, сформированных на основе анализа задачи, которые носят следующие названия: «Стрелок», «Водитель», «Радар», «Список целей», «Цель» и

«Вектор». Класс «Стрелок» является системой управления стрельбой, класс «Водитель» — системой управления маневрированием, а класс «Радар» — системой управления радаром.

Все эти шесть классов реализованы как внутренние для того, чтобы они не воспринимались средой исполнения как самостоятельные системы управления танками.

На диаграмме классы, содержащие автоматы, имеют соответствующие пометки.

Проектирование классов, рассмотрим на примере класса «Стрелок», содержащего два автомата A1 и A2.

2. Этот класс предназначен для управления стрельбой и решает следующие задачи:

- выбор цели;
- анализ параметров цели и расчет упреждения;
- управление пушкой;
- определение скорости охлаждения пушки.

3. Структурная схема класса «Стрелок» приведена на рис. 4.

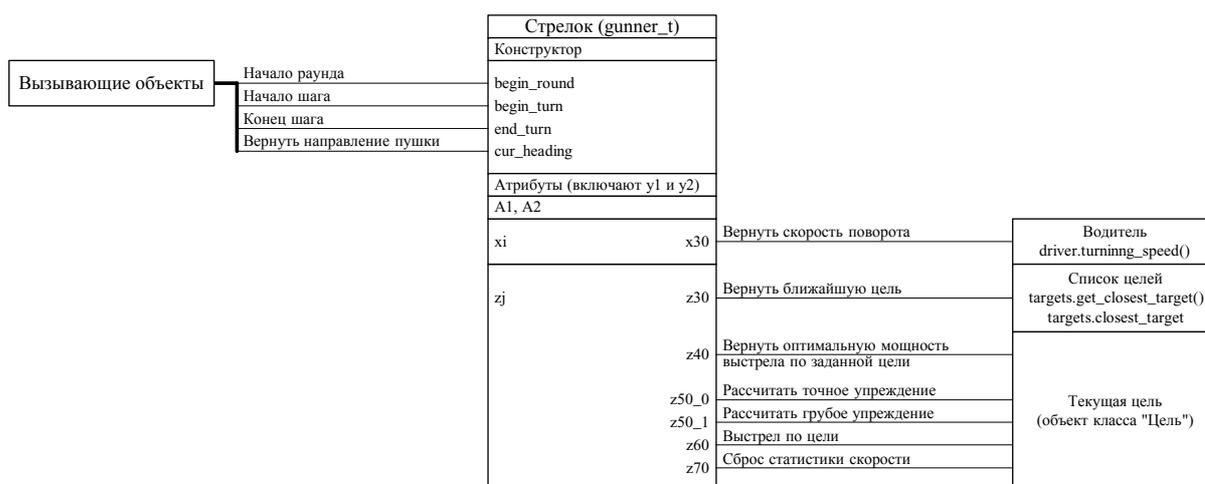


Рис. 4. Структурная схема класса «Стрелок»

Из рассмотрения структурной схемы следует, что в этом классе закрытые методы, отличные от автоматных, отсутствуют.

Интерфейс этого класса состоит из трех обработчиков событий и метода, возвращающего вызвавшему его объекту текущее положение пушки.

Закрытая часть класса состоит из атрибутов и методов, реализующих автоматы A1, A2 и их входные переменные и выходные воздействия. В этой схеме указаны обозначения только тех методов, реализующих входные переменные и выходные воздействия, которые передают сообщения другим объектам. Остальные автоматные методы указаны ниже на схеме связей автомата A1.

4. Так как класс содержит более одного автомата, то строится схема их взаимодействия (рис. 5), являющаяся, в отличие от приведенной в работе [10], весьма простой.



Рис. 5. Схема взаимодействия автоматов класса «Стрелок»

Проектирование автоматов, входящих в класс, рассмотрим на примере автомата A1, участвующего в решении первых трех из указанных задач.

В этапах 5 и 6 в данном случае нет необходимости.

7. Схема связей автомата A1 приведена на рис. 6. Отметим, что в этой схеме, в отличие от структурной схемы класса, на линиях указываются не названия сообщений, а названия входных и выходных воздействий. Например, выходное воздействие z30, названное на рис. 6 «Выбрать цель», осуществляя выбор цели, передает объекту класса

«Список целей» сообщение «Вернуть ближайшую цель» (рис. 4).

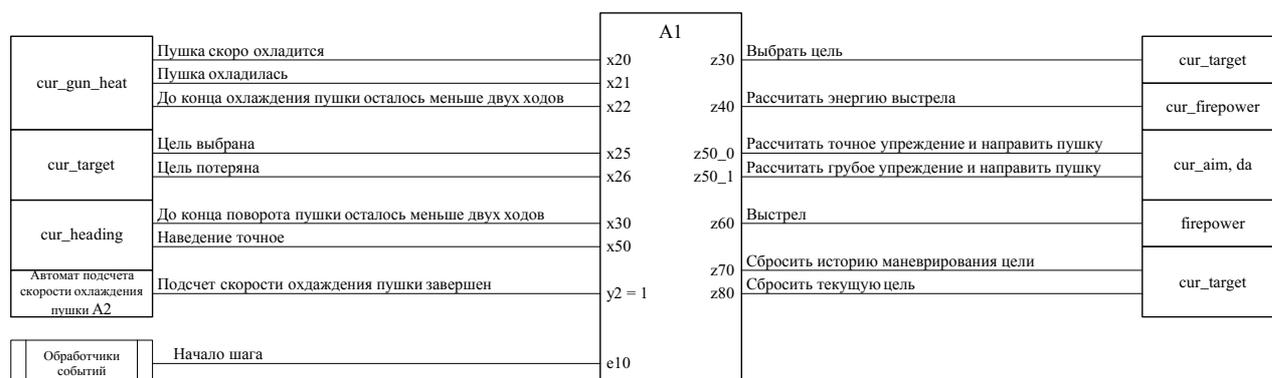


Рис. 6. Схема связей автомата управления стрельбой

8. На рис. 7 приведен граф переходов автомата A1.

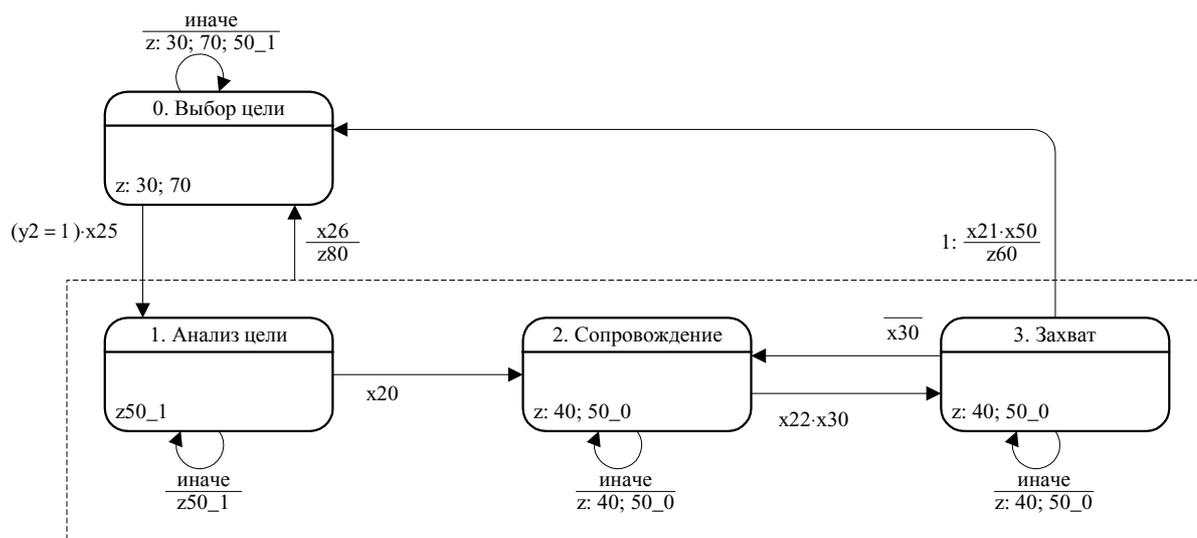


Рис. 7. Граф переходов автомат управления стрельбой

9. С целью экономии листинг 1 содержит фрагмент программы, реализующий рассматриваемый класс. Структура этого фрагмента определяется схемой, приведенной на рис. 4.

Аналогично разрабатываются остальные классы, образуя программу в целом.

10. Листинг 2 содержит фрагмент протокола функционирования разработанной системы управления для одного раунда боя двух танков.

Как было отмечено выше, при необходимости протоколы могут быть дополнены информацией о значениях аналоговых параметров. Например, при выполнении функции, реализующей выходное воздействие, определяющее необходимое направление пушки (z50_0), дополнительно может выводиться полученный результат.

Отметим также, что если в игре участвуют два «своих» танка, то может быть запротоколирован весь бой, включая все попадания снарядами и гибель, по крайней мере, одного из них. Это позволяет полностью восстановить всю историю боя, а при необходимости понять причины принятия тех или иных решений.

11. Документация на программу в целом, разработанная на основе предлагаемого подхода, приведена на сайте [30].

В заключение раздела отметим, что использование состояний, кроме указанных выше преимуществ, уменьшает вероятность того, что программа не успеет выполнить необходимые расчеты в отведенный ей интервал времени. Так, например, из рис. 7 следует, что точный расчет упреждения, занимающий значительное время, выполняется только в состояниях два и три, в которых автомат находится гораздо меньшее время, чем в остальных состояниях, в которых расчет упреждения выполняется грубо.

1.5. Результаты сражений

История создания танка состоит из двух частей. Сначала был создан **не стреляющий** танк (counterwallrobot.Cynical_1), который в течение нескольких дней (с 28.09.01 до 02.10.01) **был лучшим** на

открытом турнире, практически ежедневно проводимом создателем сайта [31]. Начиная с 15.10.01, он стал **стреляющим** (counterwallrobot.Cynical_2), и занимает 5-6 место в мире. Разработка следующей версии описана в разд. 1.8.

1.6. Сравнение с UML

Перечислим имеющиеся, **по нашему мнению**, недостатки Унифицированного Языка Моделирования (Unified Modeling Language), являющегося в настоящее время наиболее известным и распространенным в мире графическим языком для визуализации, спецификации, конструирования и документирования систем, в которых программное обеспечение разрабатывается на основе объектного подхода:

- неудачное название языка, так как он предназначен «не для моделирования как такового» [7], а для создания моделей в виде диаграмм, позволяющих описывать различные «стороны» программного обеспечения;
- не рассмотрены особенности реализации предложенных моделей, например, в части их разработки с использованием шаблонов и образцов;
- применяются весьма странные термины, например, «событие-триггер» и «сторожевое условие», в то время как в вычислительной технике термины «событие», «триггер» и «условие» имеют вполне определенный и традиционный смысл;
- необоснованно много внимания уделяется диаграммам прецедентов (диаграммам использования), в которых «прецедент (Use case) специфицирует поведение системы или ее части и представляют собой описание множества

последовательностей действий (включая варианты), выполняемых системой (совокупностью образующих ее объектов) для того, чтобы актер мог получить определенный результат» [7]. При **сложном** поведении рассматриваемых компонент построить на этапе проектирования исчерпывающие диаграммы такого типа **практически** невозможно. Обратим внимание, что в рассмотренном в настоящей главе примере вообще нет актера (пользователя), который хотел бы получить результат, отличный от простого выполнения танком требований среды разработки, и поэтому в данном случае отсутствует возможность построения рассматриваемой разновидности диаграмм;

- диаграммы кооперации при **большом** числе сообщений у взаимодействующих объектов становятся необозримыми;
- при **сложном** поведении объектов построить исчерпывающие диаграммы последовательностей на этапе проектирования **практически** невозможно;
- каждая диаграмма состояний предназначена «для моделирования **жизненного цикла** объекта» [7]. Поэтому при построении таких диаграмм могут использоваться вложенные состояния, но не вложенные автоматы. Действительно, если применять термин «жизненный цикл», то без учета реинкарнации одному объекту может соответствовать только одна диаграмма состояний. Делая акцент на поведении объекта, а не на его жизненном цикле, вопрос об единственности автомата для каждого объекта снимается. У объекта может быть много методов, часть из

которых (а не только один) могут быть автоматными. При такой трактовке использование автоматов в объектном программировании становится более понятным по сравнению с приведенными выше определениями «пророков»;

- использование словесных, а не символьных пометок дуг, петель и вершин при построении диаграмм состояний, не позволяет получать эти диаграммы в строгом и компактном виде, особенно для объектов, обладающих сложным поведением;
- если в некоторых работах, например в работе [7], упоминаются такие термины, как «автомат Мура» или «автомат Мили», то уже в стандарте [32], эти термины исключены, в то время как в предлагаемом подходе [12] используемый тип модели автомата определяет его реализацию;
- ввиду того, что в работах [12,33] показано, что параллельные процессы могут быть описаны системой взаимосвязанных графов переходов, то в применении диаграмм деятельности нет необходимости;
- вычислительные алгоритмы могут быть описаны не с помощью диаграмм деятельности, а на основе традиционных схем алгоритмов, которые при необходимости формально могут быть преобразованы в графы переходов [12].

Из изложенного следует, что при создании языка UML был нарушен «принцип Оккама», так как в него введено много сущностей и терминов без необходимости. Кстати, такого же мнения придерживаемся не только мы. Говоря о книге «трех друзей» [34], автор [2] пишет: «Перед тем как начать читать эту книгу я приготовился к самому худшему.

Я был приятно удивлен — лишь некоторые места книги содержали объяснения, которые были написаны так, будто сами авторы книги не совсем понимали, о чем речь». Также он пишет: «когда вы в первый раз сталкиваетесь с языком UML, кажется, что его невозможно понять — такое там множество диаграмм и мелочей. Во втором издании книги [28] ее авторы утверждают, что большая часть всего этого попросту никому не нужна, и поэтому они отбрасывают малозначительные детали и говорят только о самом главном из имеющегося в языке».

В настоящей работе мы пошли дальше:

- исключили диаграммы прецедентов, кооперации, последовательностей и действий;
- для каждого класса ввели его структурную схему и нотацию для ее построения;
- для каждого класса, содержащего более одного автомата, ввели схему взаимодействия автоматов;
- для каждого автомата ввели схему связей;
- изменили нотацию диаграмм состояний, например, в части введения в их вершины вложенных автоматов;
- ввели шаблон для стандартной реализации каждого автомата;
- вместо построения сценариев (прецедентов, последовательностей) при создании программы, ввели исчерпывающее **автоматическое** протоколирование, выполняемое в терминах автоматов при ее отладке и сертификации. По нашему мнению, такое протоколирование может быть полезно при построении **черных ящиков**, обеспечивая при необходимости фиксацию **всего** поведения «танка» в период боя;

- предложили **технологию** создания программного обеспечения, которая является вариантом SWITCH-технологии, предназначенным для использования при объектном проектировании.

В настоящей работе рассматривается случай однопроцессорной реализации программ, и поэтому вопрос об использовании таких диаграмм, как, например, диаграмма развертывания или ее аналоги, не ставится.

Программный проект и документация на него приведены на сайте <http://is.ifmo.ru> в разделе «Проекты» (Туккель Н.И., Шалыто А.А. Система управления танком для игры «Robocode». Вариант 1. СПб., 2001).

1.7. Листинги

Ниже в качестве примера приведены фрагмент программы, реализующий один из «автоматных» классов системы управления танком (разд. 1.7.1), и фрагмент протокола боя двух танков (разд. 1.7.2).

1.7.1. Фрагмент программы, реализующей класс «Стрелок»

```
public class gunner_t extends Object
{
    // gunner_t: Конструктор.

    // begin_round: Начало раунда.

    // Начало шага.
    public void
    begin_turn()
    {
        old_heading = cur_heading ;
        cur_heading = getGunHeadingRadians() ;
        old_gun_heat = cur_gun_heat ;
        cur_gun_heat = getGunHeat() ;

        da = 0 ;
        firepower = 0 ;

        A1(10) ;
        A2(10) ;
    }

    // end_turn: Конец шага.

    // cur_heading: Вернуть направление пушки.

    // Закрытые атрибуты.
```

```

...
private int          y1 = 0 ;
private int          y2 = 0 ;

// Реализация автомата A1.
private void
A1( int e )
{
    int    y_old = y1 ;

    if( OBJECTS_LOGGING )
        log( "Для объекта 'Стрелок':" ) ;

    if( A1_BEGIN_LOGGING )
        log_begin( "A1", y1, e ) ;

    switch( y1 )
    {
        case 0:
            if( (y2 == 1) && x25() )          y1 = 1 ;
            else
                { z30() ; z70() ; z50_1() ; }
            break ;

        case 1:
            if( x26() )          { z80() ;          y1 = 0 ; }
            else
                if( x20() )      y1 = 2 ;
                else
                    { z50_1() ; }
            break ;

        case 2:
            if( x26() )          { z80() ;          y1 = 0 ; }
            else
                if( x22() && x30() ) y1 = 3 ;
                else
                    { z40() ; z50_0() ; }
            break ;

        case 3:
            if( x26() )          { z80() ;          y1 = 0 ; }
            else
                if( x21() && x50() ) { z60() ;          y1 = 0 ; }
                else
                    if( !x30() )   y1 = 2 ;
                    else
                        { z40() ; z50_0() ; }
            break ;

        default :
            if( A1_ERROR_LOGGING )
                log_error( "A1", y1 ) ;
    }

    if( y1 != y_old )
    {

        if( A1_TRANS_LOGGING )
            log_trans( "A1", y1, y_old ) ;

        switch( y1 )
        {
            case 0:
                z30() ; z70() ;
                break ;

            case 1:
                z50_1() ;
                break ;
        }
    }
}

```

```

    case 2:
        z40() ; z50_0() ;
        break ;

    case 3:
        z40() ; z50_0() ;
        break ;
    }
}

if( A1_END_LOGGING )
    log_end( "A1", y1 ) ;
}

// Реализация автомата A2.

/** Реализация входных переменных.

// x10: Подсчет скорости охлаждения пушки завершен. Автомат A2.

private boolean
x20()
{
    boolean result = cur_gun_heat/gun_heat_decrement <= 3 ;

    if( INPUTS_LOGGING )
        log_input( "x20", "Пушка скоро охладится", result ) ;
    return result ;
}

// x21: Пушка охладилась.

// x22: До конца охлаждения пушки меньше двух ходов.

// x25: Цель выбрана.

// x26: Цель потеряна.

private boolean
x30()
{
    boolean result = true ;

    double gun_to_go = get_angle_diff( cur_heading, cur_aim.a ) ;
    double turn_direction = gun_to_go >= 0 ? 1 : -1 ;
                                // Обращение к объекту класса "Водитель".
    double gun_turning_speed = turn_direction * gun_rotation_speed
                                + driver.turning_speed() ;

    result = Math.abs( gun_to_go / gun_turning_speed ) <= 1 ;

    if( INPUTS_LOGGING )
        log_input( "x30", "До конца поворота пушки меньше двух ходов", result ) ;
    return result ;
}

// x50: Наводение точное.

/** Реализация выходных воздействий.
private void
z30()
{
    if( OUTPUTS_LOGGING )
        log_output( "z30", "Выбрать цель" ) ;

    cur_target = targets.get_closest_target( 8 ) ;
    if( cur_target == null )
        cur_target = targets.closest_target ;
}

```

```

// z35: Подсчет скорости охлаждения пушки. Автомат А2.
// z36: Вывести скорость охлаждения пушки. Автомат А2.
// z40: Рассчитать мощность выстрела.
// z50_0: Рассчитать точное упреждение и направить пушку.
// z50_1: Рассчитать грубое упреждение и направить пушку.
// z60: Выстрел.
// z70: Сбросить историю маневрирования цели.
// z80: Сбросить текущую цель.
} // gunner_t

```

1.7.2. Фрагмент протокола боя двух танков

Для объекта 'Супервизор':

```

{ A0: Автомат А0 запущен в состоянии 0 с событием e9
  * z10_0: Инициализация при запуске.
  T A0: Автомат А0 перешел из состояния 0 в состояние 1
  * z10_1: Инициализация в начале раунда.

```

*** Раунд 1

```

  * z10_2: Инициализация в начале шага.
} A0: Автомат А0 завершил свою работу в состоянии 1
----- 0 ----- Начальный шаг (e9)

```

Для объекта 'Супервизор':

```

{ A0: Автомат А0 запущен в состоянии 1 с событием e10
  * z10_2: Инициализация в начале шага.
} A0: Автомат А0 завершил свою работу в состоянии 1

```

Для объекта 'Стрелок':

```

{ A1: Автомат А1 запущен в состоянии 0 с событием e10
  * z30: Выбрать цель.
  * z70: Сбросить историю маневрирования цели.
  * z50_1: Рассчитать грубое упреждение и направить пушку.
} A1: Автомат А1 завершил свою работу в состоянии 0
{ A2: Автомат А2 запущен в состоянии 0 с событием e10
  i x10: Подсчет скорости охлаждения пушки завершен? - НЕТ.
  * z35: Подсчет скорости охлаждения пушки.
} A2: Автомат А2 завершил свою работу в состоянии 0

```

Для объекта 'Радар':

```

{ A4: Автомат А4 запущен в состоянии 0 с событием e10
  i x70: Цикл сканирования завершен? - НЕТ.
  * z100_0: Повернуть радар влево.
} A4: Автомат А4 завершил свою работу в состоянии 0

```

Для объекта 'Водитель':

```

{ A3: Автомат А3 запущен в состоянии 0 с событием e10
  i x100: Враг близко? - ДА.
  i x110: Сработал таймер T110? - ДА.
  * z200_0: Инициализация движения по траектории 'Маятник'.
  * z200_1: Добавить случайную составляющую к траектории 'Маятник'.
  * z200_2: Определить направление и скорость движения 'Маятник'.
} A3: Автомат А3 завершил свою работу в состоянии 0

```

----- 30 ----- **Выстрел по цели**

Для объекта 'Цель' (gg.Tarsier):

```

{ A5: Автомат А5 запущен в состоянии 2 с событием e100
  * z1001: Обновить параметры цели.
} A5: Автомат А5 завершил свою работу в состоянии 2

```

Для объекта 'Супервизор':

```

{ A0: Автомат А0 запущен в состоянии 1 с событием e10
  * z10_2: Инициализация в начале шага.

```

```

Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e140
  i x1000: Информация о цели устарела? - НЕТ.
} A5: Автомат A5 завершил свою работу в состоянии 2
} A0: Автомат A0 завершил свою работу в состоянии 1
Для объекта 'Стрелок':
{ A1: Автомат A1 запущен в состоянии 3 с событием e10
  i x26: Цель потеряна? - НЕТ.
  i x21: Пушка охладилась? - ДА.
  i x50: Наведение точное? - ДА.
  * z60: Выстрел.
T A1: Автомат A1 перешел из состояния 3 в состояние 0
  * z30: Выбрать цель.
  * z70: Сбросить историю маневрирования цели.
} A1: Автомат A1 завершил свою работу в состоянии 0
{ A2: Автомат A2 запущен в состоянии 1 с событием e10
} A2: Автомат A2 завершил свою работу в состоянии 1
Для объекта 'Радар':
{ A4: Автомат A4 запущен в состоянии 0 с событием e10
  i x70: Цикл сканирования завершен? - ДА.
  i x80: Пройденный радаром путь меньше 180 градусов? - ДА.
  * z101_0: Сбросить память пройденного радаром пути.
T A4: Автомат A4 перешел из состояния 0 в состояние 1
  * z100_1: Повернуть радар вправо.
} A4: Автомат A4 завершил свою работу в состоянии 1
Для объекта 'Водитель':
{ A3: Автомат A3 запущен в состоянии 0 с событием e10
  i x100: Враг близко? - ДА.
  i x110: Сработал таймер T110? - ДА.
  * z200_0: Инициализация движения по траектории 'Маятник'.
  * z200_1: Добавить случайную составляющую к траектории 'Маятник'.
  * z200_2: Определить направление и скорость движения 'Маятник'.
} A3: Автомат A3 завершил свою работу в состоянии 0

----- 42 ----- Попадание в цель (e130)
Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e100
  * z1001: Обновить параметры цели.
} A5: Автомат A5 завершил свою работу в состоянии 2
Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e130
  * z1010: Обновить статистику попаданий в цель.
} A5: Автомат A5 завершил свою работу в состоянии 2
Для объекта 'Супервизор':
{ A0: Автомат A0 запущен в состоянии 1 с событием e10
  * z10_2: Инициализация в начале шага.
Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e140
  i x1000: Информация о цели устарела? - НЕТ.
} A5: Автомат A5 завершил свою работу в состоянии 2
} A0: Автомат A0 завершил свою работу в состоянии 1
Для объекта 'Стрелок':
{ A1: Автомат A1 запущен в состоянии 1 с событием e10
  i x26: Цель потеряна? - НЕТ.
  i x20: Пушка скоро охладится? - НЕТ.
  * z50_1: Рассчитать грубое упреждение и направить пушку.
} A1: Автомат A1 завершил свою работу в состоянии 1
{ A2: Автомат A2 запущен в состоянии 1 с событием e10
} A2: Автомат A2 завершил свою работу в состоянии 1
Для объекта 'Радар':
{ A4: Автомат A4 запущен в состоянии 1 с событием e10
  i x70: Цикл сканирования завершен? - ДА.
  i x80: Пройденный радаром путь меньше 180 градусов? - ДА.
  * z101_0: Сбросить память пройденного радаром пути.
T A4: Автомат A4 перешел из состояния 1 в состояние 0
  * z100_0: Повернуть радар влево.
} A4: Автомат A4 завершил свою работу в состоянии 0
Для объекта 'Водитель':
{ A3: Автомат A3 запущен в состоянии 0 с событием e10
  i x100: Враг близко? - ДА.

```

```

i x110: Сработал таймер T110? - ДА.
* z200_0: Инициализация движения по траектории 'Маятник'.
* z200_1: Добавить случайную составляющую к траектории 'Маятник'.
* z200_2: Определить направление и скорость движения 'Маятник'.
} A3: Автомат A3 завершил свою работу в состоянии 0

```

----- 59 ----- **Попадание в нас (e45)**

```

Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e100
* z1001: Обновить параметры цели.
} A5: Автомат A5 завершил свою работу в состоянии 2
Для объекта 'Водитель':
{ A3: Автомат A3 запущен в состоянии 0 с событием e45
i x100: Враг близко? - ДА.
i x110: Сработал таймер T110? - ДА.
* z200_0: Инициализация движения по траектории 'Маятник'.
* z200_1: Добавить случайную составляющую к траектории 'Маятник'.
* z200_2: Определить направление и скорость движения 'Маятник'.
} A3: Автомат A3 завершил свою работу в состоянии 0
Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e136
} A5: Автомат A5 завершил свою работу в состоянии 2
Для объекта 'Супервизор':
{ A0: Автомат A0 запущен в состоянии 1 с событием e10
* z10_2: Инициализация в начале шага.
Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e140
i x1000: Информация о цели устарела? - НЕТ.
} A5: Автомат A5 завершил свою работу в состоянии 2
} A0: Автомат A0 завершил свою работу в состоянии 1
Для объекта 'Стрелок':
{ A1: Автомат A1 запущен в состоянии 2 с событием e10
i x26: Цель потеряна? - НЕТ.
i x30: До конца поворота пушки меньше двух ходов? - ДА.
i x22: До конца охлаждения пушки меньше двух ходов? - ДА.
T A1: Автомат A1 перешел из состояния 2 в состояние 3
* z40: Рассчитать мощность выстрела.
* z50_0: Рассчитать точное упреждение и направить пушку.
} A1: Автомат A1 завершил свою работу в состоянии 3
{ A2: Автомат A2 запущен в состоянии 1 с событием e10
} A2: Автомат A2 завершил свою работу в состоянии 1
Для объекта 'Радар':
{ A4: Автомат A4 запущен в состоянии 0 с событием e10
i x70: Цикл сканирования завершен? - ДА.
i x80: Пройденный радаром путь меньше 180 градусов? - ДА.
* z101_0: Сбросить память пройденного радаром пути.
T A4: Автомат A4 перешел из состояния 0 в состояние 1
* z100_1: Повернуть радар вправо.
} A4: Автомат A4 завершил свою работу в состоянии 1
Для объекта 'Водитель':
{ A3: Автомат A3 запущен в состоянии 0 с событием e10
i x100: Враг близко? - ДА.
i x110: Сработал таймер T110? - ДА.
* z200_0: Инициализация движения по траектории 'Маятник'.
* z200_1: Добавить случайную составляющую к траектории 'Маятник'.
* z200_2: Определить направление и скорость движения 'Маятник'.
} A3: Автомат A3 завершил свою работу в состоянии 0

```

----- 332 ----- **Конец раунда (e20)**

```

Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e100
* z1001: Обновить параметры цели.
} A5: Автомат A5 завершил свою работу в состоянии 2
Для объекта 'Супервизор':
{ A0: Автомат A0 запущен в состоянии 1 с событием e10
* z10_2: Инициализация в начале шага.
Для объекта 'Цель' (gg.Tarsier):
{ A5: Автомат A5 запущен в состоянии 2 с событием e140
i x1000: Информация о цели устарела? - НЕТ.
} A5: Автомат A5 завершил свою работу в состоянии 2

```

```

} A0: Автомат A0 завершил свою работу в состоянии 1
Для объекта 'Стрелок':
{ A1: Автомат A1 запущен в состоянии 0 с событием e10
  i x25: Цель выбрана? - ДА.
  T A1: Автомат A1 перешел из состояния 0 в состояние 1
  * z50_1: Рассчитать грубое упреждение и направить пушку.
} A1: Автомат A1 завершил свою работу в состоянии 1
{ A2: Автомат A2 запущен в состоянии 1 с событием e10
} A2: Автомат A2 завершил свою работу в состоянии 1
Для объекта 'Радар':
{ A4: Автомат A4 запущен в состоянии 1 с событием e10
  i x70: Цикл сканирования завершен? - ДА.
  i x80: Пройденный радаром путь меньше 180 градусов? - ДА.
  * z101_0: Сбросить память пройденного радаром пути.
  T A4: Автомат A4 перешел из состояния 1 в состояние 0
  * z100_0: Повернуть радар влево.
} A4: Автомат A4 завершил свою работу в состоянии 0
Для объекта 'Водитель':
{ A3: Автомат A3 запущен в состоянии 0 с событием e10
  i x100: Враг близко? - ДА.
  i x110: Сработал таймер T110? - ДА.
  * z200_0: Инициализация движения по траектории 'Маятник'.
  * z200_1: Добавить случайную составляющую к траектории 'Маятник'.
  * z200_2: Определить направление и скорость движения 'Маятник'.
} A3: Автомат A3 завершил свою работу в состоянии 0
Для объекта 'Супервизор':
{ A0: Автомат A0 запущен в состоянии 1 с событием e20
  T A0: Автомат A0 перешел из состояния 1 в состояние 2
  * z20: Вывести статистику раунда.
---- Статистика для gg.Tarsier ----
Выстрелов: 23, попаданий: 5
Вероятность: 0.217, базовая: 0.943
-----
Выстрелов: 23, попаданий: 5, промахов: 18
Меткость: 0.227
Попали в нас: 7
Столкновений со стенами: 0
} A0: Автомат A0 завершил свою работу в состоянии 2

```

1.8. Рефакторинг

После рассмотрения изложенного выше подхода Д.В. Кузнецовым было предложено выполнить рефакторинг исходного кода. При этом:

- выполнено явное отделение автоматов от управляемых ими объектов;
- произведена замена процедурной конструкции `switch` на объектно-ориентированный механизм, основанный на наследовании, который аналогичен шаблону `State` [18];
- отдельные элементы программы приведены в соответствие с рекомендациями по стилю написания

программ на языке Java, предложенными компанией Sun.

Полученный в результате рефакторинга проект представляет собой программное обеспечение, написанное в рамках объектно-ориентированного программирования с явным выделением состояний, в котором автоматы реализуются как классы. Этот подход будет рассмотрен в главе 2 настоящего отчета.

Программный проект и документация на него приведены на сайте <http://is.ifmo.ru> в разделе «Проекты» (Кузнецов Д.В., Шалыто А.А. Система управления танком для игры «Robocode». Вариант 2. СПб.: СПбГУ ИТМО, 2003).

1.9. Заключительные замечания к главе 1

Завершая главу, отметим, что, несмотря на большую важность объектно-ориентированного программирования и огромную его рекламу к этой разновидности, все программирование не сводится.

Во-первых, существуют другие парадигмы программирования на алгоритмических языках высокого уровня, а, во-вторых, широко используются вычислительные устройства, отличные от персональных компьютеров, такие как программируемые логические контроллеры, микроконтроллеры и микропроцессоры для встроенных систем, которые программируются либо на специализированных языках, либо в большинстве случаев на алгоритмических языках низкого уровня. При этом если еще недавно на долю процессоров для персональных компьютеров приходилось 6% от всех выпускаемых микропроцессоров, то в настоящее время их доля уже не превышает 2% [35], хотя объем программ, написанных для них, весьма значителен.

Многие из устройств указанных типов являются управляющими и могут программироваться с **явным выделением**

состояний [9], но не обязательно объектно, так как «объектность» обычно связана с недопустимой для таких устройств избыточностью.

В заключение раздела отметим, что понятие «состояние» является базовым практически во всех «развитых» науках, таких например, как механика, физика, металлургия, теория управления, теория автоматов. Более того, Нобелевскую премию по физике за 2001 год получили Э. Корнел, К. Виман и В. Кеттерле, которые дополнительно к четырем известным состояниям вещества, открыли (теоретически предсказанное Ш. Бозе и А. Эйнштейном еще в 1924 году) пятое состояние – сверхконденсированное, возникающее при сверхнизких температурах, когда атомы теряют свою самостоятельность, что приводит к резкому изменению всех свойств вещества.

Однако такая ситуация характерна не для всех наук. Так, например, в такой науке, как история, важнейшим является понятие «событие», которое в настоящее время является определяющим и в практическом программировании при использовании сред быстрой разработки программ, называемых также событийно-ориентированными [36].

В теоретическом программировании [25,37] понятие «состояние» применяется чаще, чем при практическом написании программ. В последнее время фирмой «Microsoft» предпринимаются усилия по использованию абстрактных машин состояний [38] в практическом программировании при построении формальных спецификаций. Однако «метод Гуревича», во-первых, требует от участников разработки знаний в области математической логики, а во-вторых, не охватывает все стадии создания программного обеспечения.

Подход, излагаемый в настоящей работе, является продолжением работ авторов, направленных на широкое внедрение таких понятий, как «состояние» и «автомат», в инженерное программирование (software engineering). Он

берет свое начало в теории автоматов и в проектировании систем управления, и не ставит своей целью учесть все особенности объектных языков программирования, которые описаны в столь толстых книгах, как [1] и [2]. Настоящая работа является первой редакцией предлагаемого подхода для рассматриваемого класса задач, и, возможно, в дальнейшем будет усовершенствована.

Из изложенного следует, что объектно-ориентированное программирование с явным выделением состояний базируется на двух парадигмах: объектной и автоматной. Это соответствует идее использования нескольких парадигм, развиваемой в настоящее время в программировании [39]. При этом если «объектный подход предоставляет программисту средства для решения задачи в ее пространстве [контексте]» [2], то автоматный подход — средства для описания поведения объектов в терминах пространства состояний. Применение автоматов проясняет поведение программы, также как применение объектов проясняет ее структуру.

После рассмотрения изложенного выше подхода был выполнен рефакторинг проекта (разд. 1.8) с целью перехода от реализации автоматов, как методов классов, к реализации автоматов, как классов.

2. Реализация автоматов, как классов

О книге Дональда Кнута «Искусство программирования» Билл Гейтс сказал: «Если вы считаете себя действительно хорошим программистом..., прочитайте «Искусство программирования»... Если Вы сможете прочесть весь этот труд, то вам определенно следует отправить мне резюме» [40]. Математики говорят, что «в «Корне» есть все!» [41]. Программисты могут то же самое сказать о Кнутае.

Д. Кнут, кроме математических основ программирования, пытается обучать также и искусству прикладного программирования. С этой целью он создает виртуальную машину MIX, для программирования которой разработал язык ассемблера. Одним из самых «больших» и подробно рассмотренных примеров, демонстрирующих «искусство программирования по Кнуту», является разработка программного обеспечения для системы управления лифтом (разд. 2.2.5 «Дважды связанные списки» в работе [40]).

Из этого примера следует, что искусством программирования Д. Кнут обладает, так как настоящее искусство не предполагает обоснования процесса создания произведения. Автор на основании технического задания, о качестве которого сказано выше, используя только словесное описание алгоритмов, на многих страницах книги приводит в окончательной форме программу на упомянутом языке низкого уровня, откомментированную не более внятно, чем было сформулировано задание.

В этом случае имеет место ситуация, не удовлетворявшая Э. Дейкстру, который говорил, «что программы часто приводятся в форме готовых изделий, почти без упоминания тех рассуждений, которые проводились в процессе разработки и служили обоснованием для окончательного вида завершенной программы» [42].

Переходя к науке программирования, которая, в отличие от искусства, должна объяснять, как создавалось произведение, отметим, что будем понимать ее существенно шире, чем Д. Грис [43], трактовавший ее только, как верификацию программ. В настоящее время наука программирования включает в себя также проектирование [44], реализацию [45], документирование [46] и отладку [47].

Программное обеспечение для системы управления лифтом можно разработать с помощью различных подходов, один из которых связан с использованием конечных автоматов [48–50].

Настоящая работа призвана продемонстрировать преимущества подхода, названного «автоматное программирование с явным выделением состояний» [48, 51–53], на примере задачи управления лифтом. Кроме того, приводится методика преобразования объектной программы, написанной в рамках предлагаемого подхода, в процедурную программу для выполнения ее на микроконтроллере.

Необходимо отметить, что в настоящей работе, для того, чтобы не загромождать исходный код не относящимися к делу функциями, инкапсуляция данных не используется. Основная причина отсутствия инкапсуляции связана с тем, что разработанная объектно-ориентированная программа впоследствии преобразуется в процедурную программу для микроконтроллера. При этом, если бы инкапсуляция применялась, то, в результате указанного преобразования, она перестала бы выполнять свою основную задачу – сокрытие данных.

2.1. Особенности объектно-ориентированного программирования с явным выделением состояний

Автоматное программирование может использоваться в одном из двух вариантов: как процедурное программирование с явным выделением состояний [51] или как объектно-ориентированное программирование с явным выделением состояний [52].

В настоящей работе, как и в работе [52], используется второй подход, основанный на двух парадигмах: объектной и автоматной. При этом отметим, что в указанной статье, как и в первой главе, автоматы реализуются, как методы

классов, в то время как в настоящей главе предлагается реализовывать их, как классы. Это позволяет в полной мере совместить гибкость объектно-ориентированного программирования с наглядностью и ясностью автоматного подхода.

Проектирование каждого автомата состоит в создании по словесному описанию (декларации о намерениях) схемы связей, описывающей его интерфейс, и графа переходов, определяющего его поведение. По этим двум документам формально и изоморфно может быть построен модуль программы, соответствующий автомату.

Используя объектную парадигму, автоматы предлагается разрабатывать, как наследники базового класса `Automat`. Этот класс реализует типовые функции автоматов (основные и вспомогательные). В наследниках определяются только функции, специфические для конкретных автоматов.

Перечислим основные функции автоматов, реализованные в базовом классе [53]:

- организация выполнения действий в вершинах графа переходов (для автоматов Мура), на его дугах и петлях (для автоматов Мили), а также в вершинах, на дугах и петлях (для автоматов Мура-Мили) [54];
- организация взаимодействия автоматов:
 - § вызов автоматов с определенными событиями;
 - § реализация вложенных автоматов;
 - § обмен номерами состояний.

Отметим, что если взаимодействие по вложенности возможно только «сверху вниз» в иерархии автоматов, то остальные два способа могут осуществляться в обе стороны, как «сверху вниз», так и «снизу вверх».

Из вспомогательных функций автоматов в классе `Automat` реализована поддержка протоколирования. При этом возможно:

- автоматическое протоколирование:
 - § при начале работы автомата в определенном состоянии с определенным событием;
 - § при переходах из состояния в состояние;
 - § при завершении работы автомата в определенном состоянии;
- добавление описаний входных и выходных воздействий автомата.

В классах наследниках переопределяется ряд функций базового класса, и добавляются входные воздействия (события и переменные), внутренние переменные, выходные воздействия, объекты управления, а также вложенные и вызываемые автоматы.

В настоящей работе, как отмечалось выше, предлагаемый подход иллюстрируется примером моделирования лифта. При этом с помощью среды `Microsoft Visual C++ 6` была разработана программа `Lift`. Эта программа размещена на сайте <http://is.ifmo.ru> в разделе «Проекты» (Наумов Л.А., Шалыто А.А. Автоматное решение задачи Д. Кнута о лифте. СПб.: СПбГУ ИТМО, 2003).

Как отмечалось выше, программа `Lift` является объектно-ориентированной. Такую программу удобно разрабатывать на персональном компьютере и легко переносить на PC-подобные контроллеры. Однако, кроме таких контроллеров, в системах управления используются также микроконтроллеры, для которых отсутствуют компиляторы с объектно-ориентированных языков. Поэтому для микроконтроллеров применяется процедурное программирование.

В настоящей главе предлагается методика преобразования ядра объектно-ориентированной программы с явным выделением состояний на языке C++ в процедурную программу с явным выделением состояний на языке C.

В данном случае, под ядром программы понимается ее фрагмент, в котором отсутствует интерфейсная часть и не реализованы функции входных и внутренних переменных, а также выходных воздействий. Методика иллюстрируется примером переноса ядра программы Lift на микроконтроллер Siemens SAB 80C515. При этом использовалась среда Keil μ Vision 2. Полученная в результате программа также приведена на сайте <http://is.ifmo.ru> в указанном выше проекте лифта.

2.2. Формулировка задачи о лифте

Попытаемся из потока слов, описаний «сопрограмм», протокола работы и исходного кода программы на языке машины MIX [40], извлечь формулировку задачи.

Дано пятиэтажное здание с одним лифтом. Этаж с номером ноль – подвальный, один – цокольный, два – первый, три – второй, а четыре – третий. Будем считать, что этажи пронумерованы числами от нуля до четырех.

На каждом этаже – две кнопки для вызова лифта на движение вверх и вниз. На нулевом этаже кнопка «Вниз» заблокирована, как и кнопка «Вверх» на четвертом этаже.

В кабине лифта имеется панель с пятью кнопками для перемещения на конкретный этаж. В ней также размещены два индикатора движения (вверх и вниз). Будем рассматривать их, как единое устройство, формирующее одно из трех значений: GoingUp (лифт движется вверх), GoingDown (лифт движется вниз) или Neutral (лифт находится в режиме ожидания).

Первоначально лифт находится на втором этаже в режиме ожидания (состояние Neutral). Ни одна кнопка вызова не нажата.

При моделировании необходимо ввести масштаб времени. Будем измерять его в десятых долях секунды. Зададим продолжительность характерных для системы действий. В табл. 1 приведены имена временных параметров, их значения и комментарии к ним.

Отметим, что время перемещения лифта из состояния покоя на один этаж вверх (вниз) с остановкой на этом этаже определяется соотношением $TStarting + TUp + TUpBraking$ ($TStarting + TDown + TDownBraking$). Прохождение этажа «транзитом» происходит быстрее, так как при этом лифт не должен замедляться.

Если с этажа, на котором сейчас находится лифт, поступил вызов или один из пассажиров лифта желает выйти на данном этаже, то производится открытие дверей. Это занимает $TDoors$ единиц времени. По истечении $TDoorsTimeout$ единиц времени после открытия дверей на этаже необходимо автоматически попытаться их закрыть. Если это не удалось, то далее лифт будет пробовать закрывать двери каждые $TWaitTimeout$ единиц времени.

Если лифт выполнил все вызовы и остался стоять на этаже, то, по истечении $TInactive$ единиц времени, он должен вернуться на второй этаж, так как считается, что там наиболее вероятно появление новых пассажиров.

Для реализации временных задержек в систему введены три таймера: таймер бездействия $C1$ (для отсчета $TInactive$ единиц времени), таймер $C2$ (для остальных временных операций, не связанных с работой дверей) и таймер $C3$ (для временных операций, связанных с работой дверей).

| Имя | Значение | Комментарий |
|---------------|----------|--|
| TInactive | 300 | Если лифт находится на каком-либо этаже без движения в течение этого времени, то он должен быть автоматически направлен на второй этаж. Для определения действий в этом случае необходимо выполнить четвертый шаг приводимого после табл. 2 словесного описания работы системы |
| TDoorsTimeout | 76 | Если после открытия дверей прошло TDoorsTimeout десятых долей секунды, то необходимо попытаться их закрыть |
| TDoors | 20 | Время открытия и закрытия дверей лифта |
| TWaitTimeout | 40 | Время, через которое система пытается закрыть двери лифта. Входящий/выходящий человек может помешать этому. Описываемая задержка требуется для того, чтобы после вхождения последнего человека в кабину, двери, через некоторое время, закрылись |
| TStarting | 15 | Время ускорения лифта при начале движения |
| TUp | 51 | Время равномерного подъема лифта на один этаж |
| TUpBraking | 14 | Время замедления лифта перед остановкой при подъеме |
| TDown | 61 | Время равномерного спуска лифта на один этаж |

| | | |
|--------------|-----|--|
| TDownBraking | 23 | Время замедления лифта перед остановкой при спуске |
| TAfterRest | 20 | Время перед стартом лифта после выхода из режима ожидания |
| THuman | 25 | Время, требуемое человеку, чтобы войти или выйти из кабины |
| TWaitLimit | 600 | Максимальное время, которое человек согласен ждать лифт |

Для моделирования описываемой системы используются переменные и структуры данных, перечисленные в табл. 2.

Таблица 2

| Имя | Комментарий |
|-------------|--|
| Floor | Номер этажа, на котором находится лифт. Эта переменная получает новое значение при начале движения с этажа |
| State | Состояние движения лифта (GoingUp, GoingDown или Neutral) |
| Queue[i] | Список людей, ожидающих лифт на i -м этаже |
| Elevator | Список пассажиров, находящихся в кабине лифта |
| CallCar[i] | Вектор, i -ая ячейка которого содержит единицу, если от какого-либо из пассажиров кабины поступал вызов для движения на i -й этаж, и ноль – в противном случае |
| CallUp[i] | Вектор, i -ая ячейка которого содержит единицу, если с i -го этажа поступал вызов на движение вверх |
| CallDown[i] | Вектор, i -ая ячейка которого содержит единицу, если с i -го этажа поступал вызов на движение вниз |

Опишем работу лифта по шагам.

0. **Ожидание вызова.** Floor = 2, State = Neutral. Если поступает вызов со второго этажа, то перейти к шагу 1. Если вызов – с другого этажа, то перейти к шагу 4.
1. **Открытие дверей.** Сбросить таймер C1. Открыть двери (TDoors единиц времени) и перейти к шагу 2. Все пассажиры считаются дисциплинированными, и поэтому они не будут входить до полного открытия дверей.
2. **Вход и выход пассажиров.**
 - a. Если пройдет TInactive единиц времени (сработает таймер C1), то перейти к шагу 4.
 - b. Сбросить таймер C2. Каждого пассажира кабины (элемент списка Elevator), который ехал до данного этажа, выпустить из лифта. Это займет THuman единиц времени на одного пассажира.
 - c. Пока на этаже есть люди, ждущие лифт для движения в том же направлении, в котором он двигался (их надо искать в списке Queue[Floor]), люди впускаются внутрь кабины по одному. Это займет THuman единиц времени на каждого пассажира. Как только человек входит в кабину лифта, он сразу же нажимает на кнопку целевого этажа (изменяет значение ячейки вектора CallCar[i]). Если вызов на этот этаж был произведен раньше, то фиксировать его не требуется.
 - d. Если по таймеру C2 прошло TDoorsTimeout единиц времени, то перейти к шагу 3.
3. **Закрытие дверей.**
 - a. Если пройдет TInactive единиц времени (сработает таймер C1), то перейти к шагу 4.
 - b. Попытаться закрыть дверь. Если не получилось, то повторять попытки каждые TWaitTimeout единиц времени.

с. Когда двери закроются – выполнить присваивания: $CallUp[Floor] = 0$ (если $State \neq GoingDown$), $CallDown[Floor] = 0$ (если $State \neq GoingUp$) и $CallCar[Floor] = 0$. Присвоения удаляют информацию об обработанных вызовах. Перейти к шагу 4.

4. **Принятие решения о дальнейшем движении.**

а. Если $State = GoingUp$ ($GoingDown$) и есть еще вызовы на движение вверх (вниз), то значение переменной $State$ не изменится. Это условие означает, что существует значение i большее (меньшее) значения переменной $Floor$, для которого значение из ячеек $CallCar[i]$, $CallUp[i]$ или $CallDown[i]$ отлично от нуля. Перейти к подпункту d.

б. Если вызовов на движение вверх (вниз) нет, но есть вызовы на движение вниз (вверх), то присвоить переменной $State$ значение $GoingDown$ ($GoingUp$). Перейти к подпункту d.

с. Если вызовов вообще нет и при этом значение $Floor$ меньше (больше) двух, то присвоить переменной $State$ значение $GoingUp$ ($GoingDown$). При $Floor = 2$ присвоить переменной $State$ значение $Neutral$. Перейти к подпункту d.

d. В результате, если $State = GoingUp$, то перейти к шагу 5, если $State = GoingDown$, то – к шагу 6, а если $State = Neutral$, то – к шагу 0.

5. **Подняться на этаж.**

а. Увеличить значение переменной $Floor$ на единицу и начать движение. Это займет $TStarting + TUp$ единиц времени.

б. Если есть вызов для движения на этаж $Floor$, то необходимо, чтобы лифт притормозил. Это займет $TUpBraking$ единиц времени. Перейти к шагу 1.

с. Если нет вызовов для движения на этаж `Floor`, то повторить шаг 5.

6. Спуститься на этаж.

а. Уменьшить значение переменной `Floor` на единицу и начать движение. Это займет `TStarting + TDown` единиц времени.

б. Если есть вызов для движения на этаж `Floor`, то необходимо, чтобы лифт притормозил. Это займет `TDownBraking` единиц времени. Перейти к шагу 1.

с. Если нет вызовов для движения на этаж `Floor`, то повторить шаг 6.

Человек, в рамках решаемой задачи, представляет собой сущность, характеризуемую тремя атрибутами:

- `CurFloor` – номер этажа, на котором он появляется;
- `TgtFloor` – номер этажа, на который он хочет попасть (`CurFloor ? TgtFloor`);
- `WaitFor` – максимальное время, которое человек согласен ждать лифт. Если по истечении этого времени он не попадет в кабину, то человек пойдет пешком (покинет этаж `CurFloor`). При этом его вызов остается в силе. Начальное значение атрибута `WaitFor` равно `TWaitLimit`.

Взаимодействие людей с лифтом осуществляется через списки `Queue[i]` и `Elevator`, а также вектора `CallUp[i]`, `CallDown[i]` и `CallCar[i]`.

2.3. Проектирование автоматов и классов

На основании словесного описания, приведенного в предыдущем разделе, можно построить схему алгоритма, которая будет весьма громоздкой. Поэтому в дальнейшем будем использовать автоматный подход.

Построим три автомата, каждый из которых определяет поведение соответствующей компоненты системы. Они обеспечивают:

- принятие решений (автомат A0);
- управление двигателем, перемещающим лифт между этажами (автомат A1);
- управление пятью двигателями, открывающими/закрывающими двери на этажах (автомат A2).

Схема связей автомата A0 приведена на рис. 8, а его граф переходов – на рис. 9. Схема связей автомата A1 – на рис. 10, а его граф переходов – на рис. 11, и, наконец, схема связей автомата A2 – на рис. 12, а его граф переходов – на рис. 13.

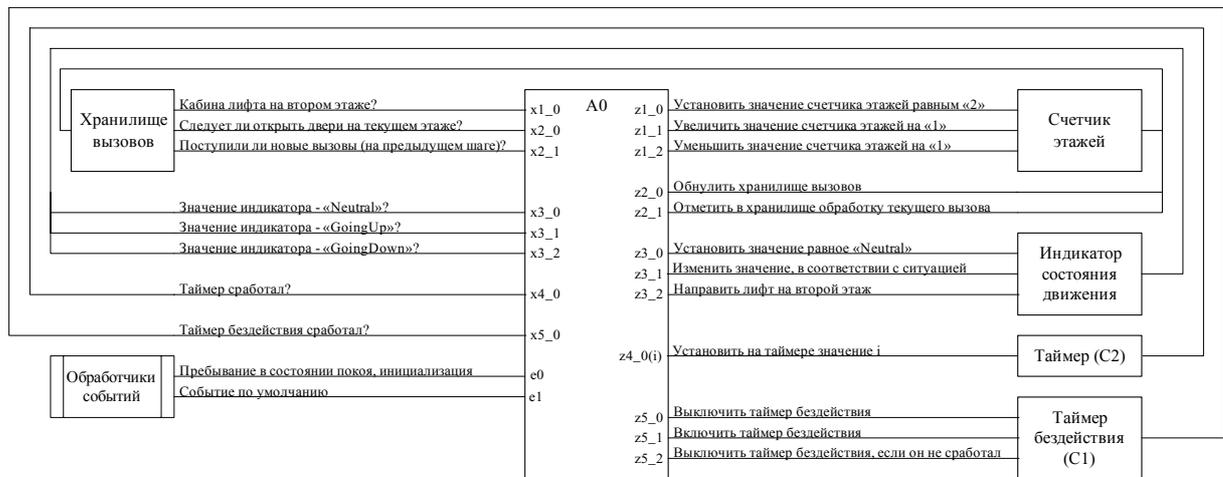


Рис. 8. Схема связей автомата A0. Принятие решений

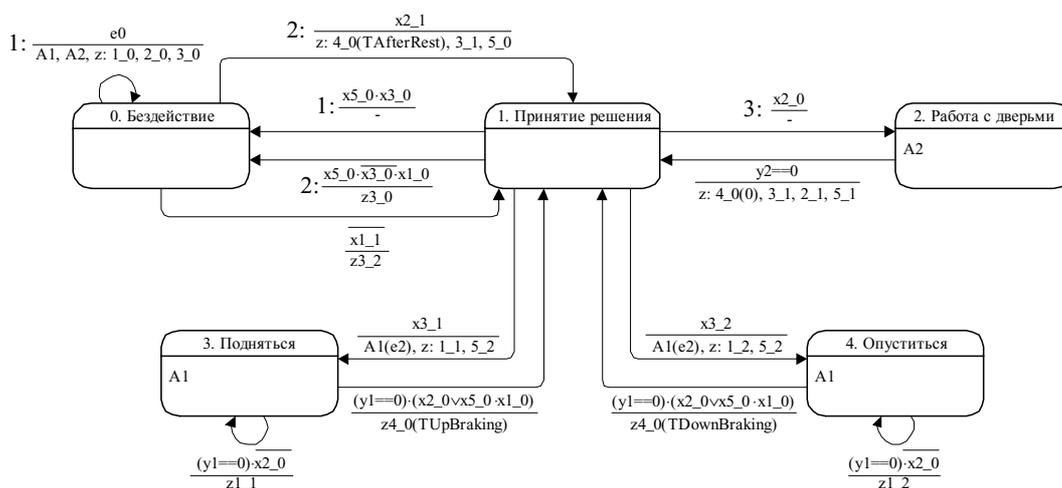


Рис. 9. Граф переходов автомата A0. Принятие решений

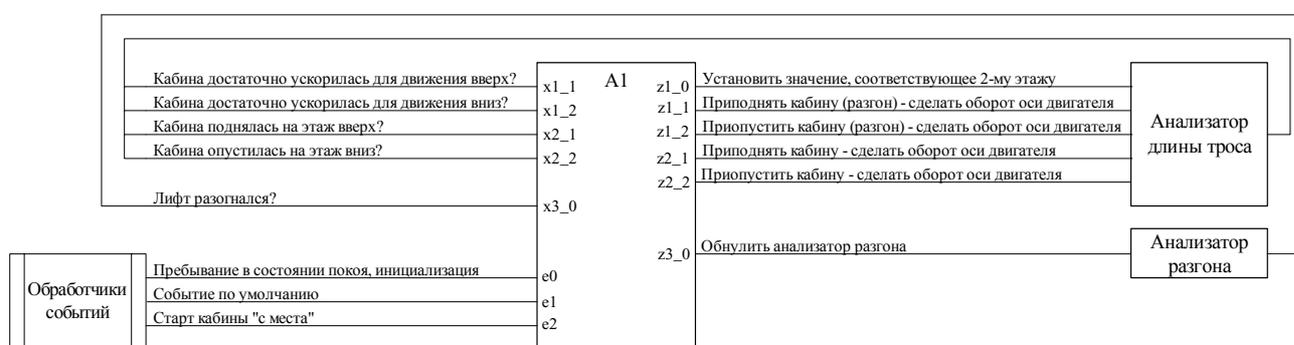


Рис. 10. Схема связей автомата A1.

Управление двигателем, перемещающим лифт между этажами

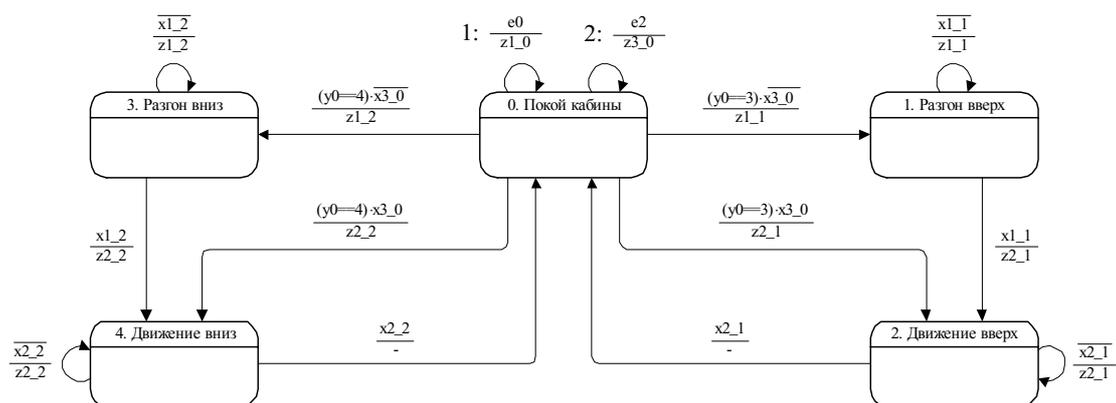


Рис. 11. Граф переходов автомата A1.

Управление двигателем, перемещающим лифт между этажами

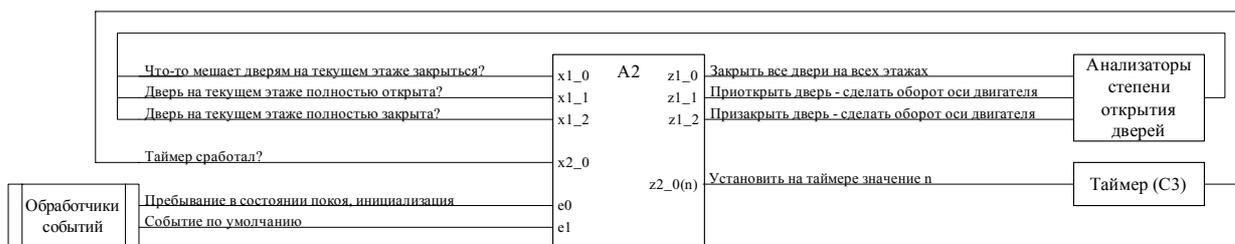


Рис. 12. Схема связей автомата А2.

Управление пятью двигателями, открывающими/закрывающими двери на этажах

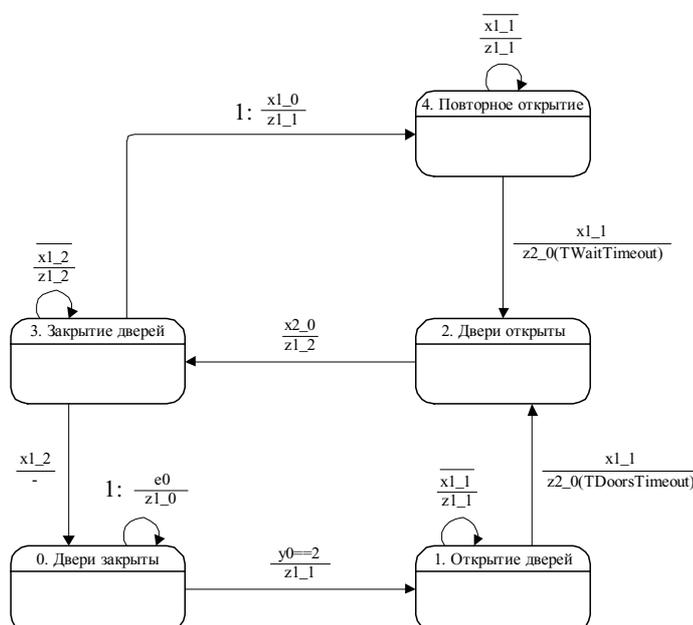


Рис. 13. Граф переходов автомата А2.

Управление пятью двигателями, открывающими/закрывающими двери на этажах

Все три автомата являются автоматами Мили, и поэтому каждый из них будет реализован с помощью одного оператора switch. Отметим, что для автомата А0 характерно, что в состоянии «Работа с дверьми» вложен автомат А2, а в состояниях «Подняться» и «Опуститься» – автомат А1. На переходах из состояния «Принятие решения» в состояния

«Подняться» и «Опуститься» осуществляется вызов автомата A1 с событием e2.

Класс A0 реализует автомат принятия решений и содержит пять объектов, управляемых им: хранилище вызовов (реализуемое векторами CallCar[i], CallUp[i] и CallDown[i]), счетчик этажей (Floor), индикатор состояния движения (State), таймер (C2) и таймер бездействия (C1). Кроме того, он содержит объекты классов A1 и A2, реализующих автоматы A1 и A2, соответственно.

Автоматы A1 и A2 не содержат вложенных автоматов и не вызывают другие автоматы с какими-либо событиями.

Класс A1 реализует автомат управления двигателем, перемещающим кабину лифта между этажами, и содержит этот двигатель.

Основной характеристикой положения лифта является длина части троса между кабиной и точкой перегиба троса. Будем измерять эту величину в «расстояниях между этажами». Таким образом, данная характеристика – число с плавающей точкой от нуля до четырех. Скорость перемещения лифта зависит от степени его разгона/торможения, которая определяется анализатором разгона. Анализаторы длины троса и разгона, в совокупности, моделируют двигатель, перемещающий кабину между этажами.

Класс A2 реализует автомат управления пятью двигателями, открывающими/закрывающими двери на этажах, а также содержит эти двигатели и таймер (C3).

Основной параметр, на который влияет двигатель, открывающий/закрывающий двери – степень ее открытия (вещественное число от нуля до единицы). Он равен отношению числа сделанных оборотов оси двигателя к общему числу оборотов, необходимых для того, чтобы дверь открылась полностью. Анализаторы степени открытия дверей моделируют соответствующие двигатели.

Кроме того, системе управления дверьми необходим таймер для определения времени закрытия дверей.

У каждого из рассмотренных автоматов имеется, так называемое, «событие по умолчанию», с которым он вызывается при отсутствии прочих событий.

В заключение раздела отметим, что включение объектов управления в классы автоматов не является обязательным.

2.4. Описание базового класса `Automat` и вспомогательных макроопределений

Рассмотрим базовый класс `Automat`, реализующий функциональность, описанную во введении:

```
class Automat
{
public:
    int y;           // Переменная, хранящая состояние автомата
    int y_old;      // Переменная, хранящая состояние, в котором автомат начал
                    // последнюю итерацию
    int BaseIndent; // Минимальный базовый сдвиг для всех записей в протоколе
    bool DoLog;     // Писать ли протокол? Вот в чем вопрос! J
    Automat* Nest; // Указатель на автомат, запустивший данный

    Automat(bool DoLog=false); // Конструктор
    void Main(int e); // Главная функция автомата

// Основные функции
protected:
    // Шаг автомата. Необходимо переопределять в наследниках
    virtual void Step(int e)=0;

    // Пост-шаг автомата, выполняемый только в случае перехода из состояния в
    // состояние (только для автоматов Мура и Мура-Мили). Необходимо
    // переопределять в наследниках
    virtual void PostStep(int e)=0;

public:
    // Выполняет шаг автомата A с событием e
    void DoNested(int e, Automat* A);

// Вспомогательные функции
public:
    // Записать в протокол строку s с отступом indent. Необходимо
    // переопределять в наследниках
    virtual void WriteLog(CString s, int indent=0)=0;

protected:
    // Записать в протокол информацию о том, что автомат запущен с событием e
    // (indent - величина отступа). Необходимо переопределять в наследниках
    virtual void WriteLogOnStart(int y,int e,int indent=0)=0;

    // Записать в протокол информацию о переходе автомата из состояния y_old в
    // состояние y (indent - величина отступа). Необходимо переопределять в
    // наследниках
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0)=0;
```

```

    // Записать в протокол информацию о том, что автомат завершил работу в
    // состоянии y (indent - величина отступа). Необходимо переопределять в
    // наследниках
    virtual void WriteLogOnFinish(int y,int indent=0)=0;

// Объекты управления. Добавить в наследниках
protected:

// Вызываемые автоматы. Добавить в наследниках
protected:

// События. Добавить в наследниках
protected:

// Входные переменные. Добавить в наследниках
protected:

// Внутренние переменные. Добавить в наследниках
private:

// Выходные воздействия. Добавить в наследниках
protected:
};

```

В конце рассмотренного кода приведен состав членов класса, которые необходимо добавлять в наследники класса `Automat` для реализации конкретных автоматов.

Кратко опишем функции-члены класса `Automat`. Одной из них является функция `Main` - главная функция автомата, обеспечивающая выполнение действий в вершинах графа переходов, на его дугах и петлях:

```

void Automat::Main(int e)           // Главная функция автомата
{
    y_old=y;                        // Запомнить состояние перед началом
                                    // итерации
    WriteLogOnStart(y,e);           // Протоколировать факт начала итерации
    Step(e);                        // Выполнить шаг
    WriteLogOnStateChanged(y,y_old); // Протоколировать факт перехода или
                                    // сохранения состояния
    if (y!=y_old) PostStep(e);     // Если переход был - выполнить пост-шаг
    WriteLogOnFinish(y);           // Протоколировать факт окончания итерации
}

```

Параметром этой функции является идентификатор события, который передается функциям `Step` и `PostStep`. При реализации автоматов Мили (Мура) единственный оператор `switch` необходимо разместить в переопределенной в наследнике функции `Step` (`PostStep`). Для автоматов Мура-Мили требуется переопределять обе эти функции.

Макроопределения `DECLARE_NO_STEP` и `DECLARE_NO_POSTSTEP` позволяют реализовать функции `Step` и `PostStep`, как пустые.

Протоколирование поведения автомата реализуется посредством переопределения функций `WriteLog`, `WriteLogOnStart`, `WriteLogOnStateChanged` и `WriteLogOnFinish`. Последние три функции, как правило, используют первую. Функцию `WriteLog` следует вызывать дополнительно, например, из функций входных и внутренних переменных, а также функций выходных воздействий.

Макроопределения `DECLARE_NO_LOG`, `DECLARE_NO_LOG_ON_START`, `DECLARE_NO_LOG_ON_STATE_CHANGED` и `DECLARE_NO_LOG_ON_FINISH` позволяют реализовать указанные выше функции, как пустые.

Все четыре функции протоколирования содержат необязательный параметр `indent` – отступ при размещении информации в протоколе. Сумма значений параметра `indent` и переменной `BaseIndent` определяют количество пробелов, добавляемых перед записью в протокол, что позволяет наглядно отразить в нем вложенность автоматов.

Для включения/выключения режима протоколирования используется булева переменная `DoLog`. Поэтому рекомендуется перечисленные выше функции протоколирования (или только функцию `WriteLog`) начинать со следующей строки:

```
if (!DoLog) return;
```

Для временного отключения протоколирования используются макроопределения `SWITCH_LOG_OFF` и `SWITCH_LOG_ON`. Необходимо, чтобы они применялись только парами, и при этом первое макроопределение должно всегда

предшествовать второму! Приведем пример их использования в программе Lift:

```
void A0::z5_2()
{
    WriteLog("z5_2: Выключить таймер бездействия, если он не сработал",3);
    SWITCH_LOG_OFF // Отключить протоколирование
    if (!x5_0()) z5_0();
    SWITCH_LOG_ON  // Включить протоколирование
}
```

В этом фрагменте программы временное отключение протоколирования используется для того, чтобы при вызове функции `z5_2()` в протоколе не отражались вызовы функций `x5_0()` и `z5_0()`.

Приведём конструктор класса `Automat`. Он предназначен для инициализации необходимых переменных:

```
Automat::Automat(bool DoLog)
{
    y=0;
    Nest=NULL;
    BaseIndent=0;
    this->DoLog=DoLog;
}
```

Используемая в тексте конструктора переменная `Nest` требуется для работы с вызываемыми автоматами. Вызов автомата выполняется с помощью функции `DoNested`, реализованной следующим образом:

```
void Automat::DoNested(int e, Automat* A)
{
    A->Nest=this;
    A->Main(e);
}
```

Так, например, вызов автомата `B` с событием `e` из автомата `A` должен быть осуществлен, как показано ниже:

```
DoNested(e, &B);
```

При этом автомат А, при необходимости, сможет определить состояние автомата В, обратившись к переменной В.у, а автомат В – узнать состояние автомата А, получив значение переменной Nest->у.

2.5. Разработка автоматов – потомков класса Automat

При разработке классов наследников необходимо:

1. Переопределить функции Step и/или PostStep.
2. Переопределить функции протоколирования WriteLog, WriteLogOnStart, WriteLogOnStateChanged и WriteLogOnFinish.
3. Объявить экземпляры структур данных, соответствующие объектам управления.
4. Объявить объекты, реализующие автоматы, которые вызывает данный автомат.
5. Объявить целочисленные статические константы, соответствующие обрабатываемым событиям.
6. Реализовать функции входных переменных, возвращающие значения для анализа.
7. Реализовать функции внутренних переменных. Эти функции могут возвращать значения произвольных типов.
8. Реализовать функции выходных воздействий, не возвращающие значений.

При решении задачи о лифте, классы, соответствующие автоматам А0, А1 и А2, разработаны по этому плану.

Большую часть исходного кода программы Lift занимает реализация интерфейса. Отметим, что вызовы главной функции автомата А0 из интерфейсной части программы происходят только в двух случаях: при повторной инициализации автомата и при выполнении шага. Они инициируются нажатием пользователем на кнопки «Reset», «Step», «Pass» или «Auto». Другие автоматы из

интерфейсной части не вызываются. Подробно интерфейс программы будет описан в следующем разделе.

2.6. Интерфейс программы

Опишем внешний вид окна программы (рис. 14).

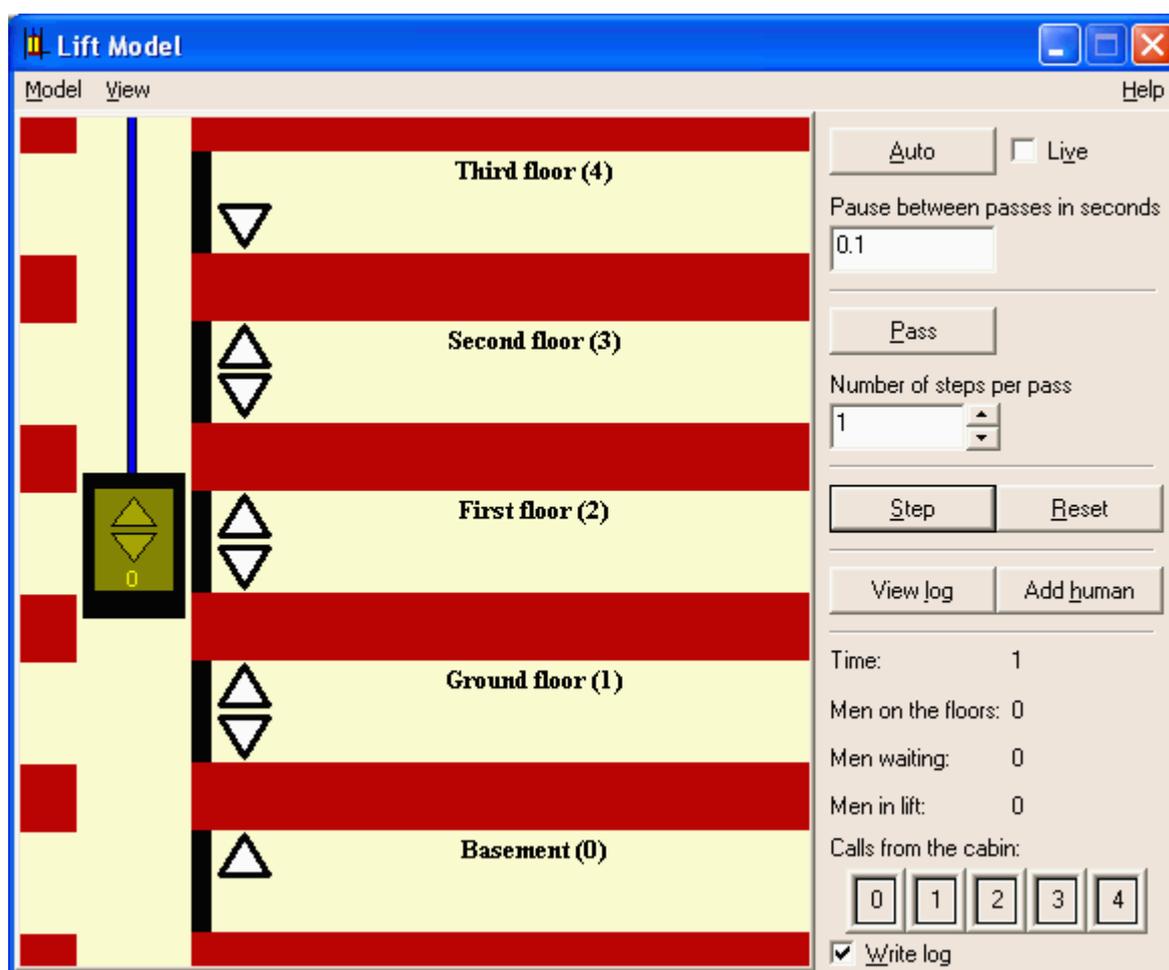


Рис. 14. Окно программы Lift

В его левой части находится область визуализации системы, предназначенная для отображения лифта, этажей, кнопок вызова на этажах и людей. Справа расположены средства управления моделью, а сверху – главное меню. Перечислим упомянутые средства управления:

- кнопка Auto предназначена для перехода в режим автоматического выполнения итераций и выхода из него. В этом режиме происходит постоянное выполнение пассивов (наборов итераций) одного за другим. Паузу между ними можно настроить;
- флажок Live предназначен для включения/выключения режима случайного автоматического добавления людей на этажи;
- поле ввода Pause between passes in seconds определяет паузу (в секундах) между пассивами при работе системы в автоматическом режиме;
- кнопка Pass позволяет выполнить набор итераций;
- поле ввода Number of steps per pass определяет количество итераций в пассиве;
- кнопка Step позволяет выполнить итерацию;
- кнопка Reset обеспечивает выполнение сброса системы (перевод каждого автомата и их объектов управления в начальные состояния). В результате, лифт пуст, находится на втором этаже, все двери закрыты, людей на этажах нет и вызовов нет;
- кнопка View log позволяет открыть/закрыть окно отображения протокола;
- кнопка Add human позволяет добавить человека на этаж. При этом появляется диалоговое окно для задания его параметров, описанных ниже;
- поле Time отображает номер текущей итерации;
- поле Men on the floors отображает число людей на этажах;
- поле Men waiting отображает число людей в очереди ожидания, размещение которых на этажах было запланировано на будущее;

- поле Men in lift отображает число людей в лифте. Это число также изображается на кабине лифта;
- панель Calls from the cabin с пятью кнопками представляет собой пульт управления в кабине лифта. При этом нажатые кнопки подсвечиваются;
- флажок Write log предназначен для включения/выключения режима протоколирования.

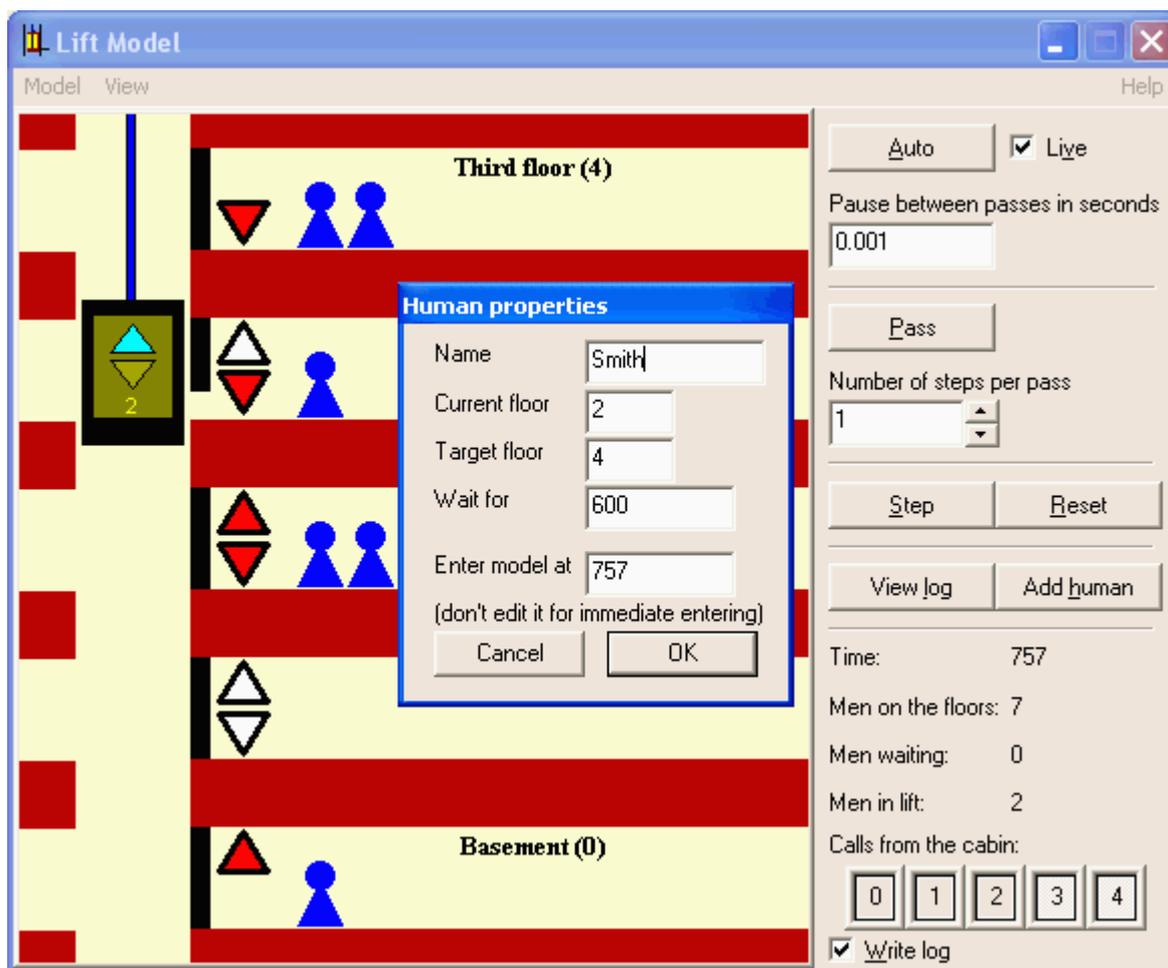


Рис. 15. Окно программы Lift после добавления людей

На рис. 15 изображено окно программы после появления людей на этажах. На нем также приведено окно настройки свойств добавляемого человека, открывающееся при нажатии

кнопки `Add human`, которое позволяет изменять следующие параметры «личности»:

- `Name` – имя человека. Имя вида «`Human#...`» генерируется автоматически, но, при необходимости, может быть изменено вручную. Этот параметр используется только при протоколировании;
- `Current floor` – этаж, на котором следует разместить человека;
- `Target floor` – этаж, на который должен попасть человек;
- `Wait for` – время ожидания лифта (в итерациях), по истечении которого человек покидает этаж, не дождавшись лифта;
- `Enter model at` – итерация, на которой человека следует разместить на этаже. Значение по умолчанию – текущая итерация. Если указанное значение не редактировать, то человек будет размещен незамедлительно.

Нажатие правой кнопки мыши на изображении человека приводит к отображению окна просмотра его параметров.

Когда человек появляется на этаже, он «нажимает» на кнопку вызова для движения вверх или вниз, а при входе в кабину он «нажимает» на кнопку на пульте управления.

Отметим, что пользователь программы может нажимать кнопки вызовов с этажей и кнопки пульта управления с помощью мыши.

Обратим внимание, что нажатие на кнопку отменить невозможно и вызов, рано или поздно, будет обработан. Так, например, если человек появился на этаже, вызвал лифт и ушел, не дождавшись его прибытия, то лифт все равно приедет на этаж, и двери откроются.

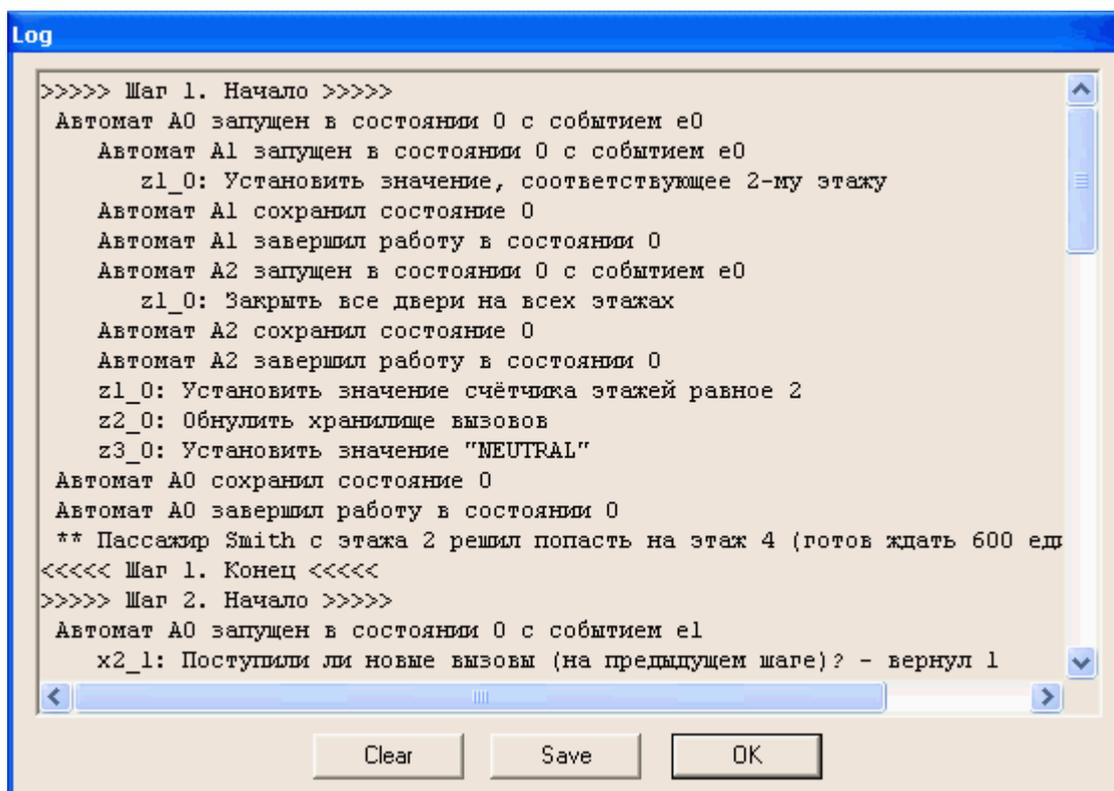


Рис. 16. Окно просмотра протокола

На кабине лифта, в виде двух стрелок, изображен индикатор направления движения лифта.

В случае, если на этаже пытаются разместиться более десяти человек, то изображаются только первые десять.

Как отмечалось выше, в системе имеются средства для ведения и просмотра протокола работы модели. Окно просмотра изображено на рис. 16.

В протоколе итерации отделены друг от друга специальными скобками. Вложенность автоматов отражается отступами. Информация о входных и внутренних переменных, а также выходных воздействиях добавляется с еще большими отступами.

Строка, начинающаяся с двух звездочек, содержит информацию, являющуюся внешней для системы управления лифтом, и может представлять собой одно из следующих сообщений:

- «Пассажир ... с этажа ... решил попасть на этаж ... (готов ждать ... единиц времени)»;
- «Пассажир ... вошел в лифт на этаже ...»;
- «Пассажир ... приехал на этаж ...»;
- «Пассажир ... решил пойти пешком».

Информация о нажатии кнопок вызова пользователем фиксируется в протоколе с помощью сообщений:

- «Поступил вызов на движение (вверх|вниз) с ... этажа»;
- «Из кабины лифта поступил вызов на движение на ... этаж».

Опишем назначение кнопок в окне протоколирования. Кнопка Clear предназначена для очистки протокола, кнопка Save позволяет записать протокол в текстовый файл, а кнопка ОК – закрыть окно.

Завершим описание интерфейса рассмотрением главного меню. В нем пункт Reset подменю Model повторяет функциональность кнопки Reset, а пункт Exit того же подменю служит для выхода из программы.

Подменю View содержит три пункта:

- Log window аналогичен кнопке View log, описанной выше;
- Timeouts tuning window открывает окно, позволяющее настраивать временные параметры, описанные в табл. 1 (кроме параметра TWaitLimit). По умолчанию эти параметры настроены, как в работе [40];
- Add human window аналогичен кнопке Add human, описанной выше.

Подменю Help содержит единственный пункт About, отображающий информацию о программе.

На этом завершается описание созданной модели лифта, функционирующей на персональном компьютере. Ее

работоспособность и соответствие техническому заданию (разд. 2) подтверждаются использованием.

2.7. Переход от объектного варианта программы к процедурному

При переходе от модели (разработанной выше программы) к реальному управлению лифтом на базе PC-подобного контроллера достаточно создать наследников классов, реализующих автоматы модели, переопределив лишь виртуальные функции входных и внутренних переменных, а также выходных воздействий. Таким образом, реализацию реальной системы на указанном вычислительном устройстве следует наследовать от модели.

В случае построения системы на микроконтроллерах, такой подход неприменим, так как они, обычно, программируются процедурно. Изложим методику, позволяющую перенести программу, написанную в рамках предлагаемого подхода на языке C++ на язык C, и проиллюстрируем ее примером переноса ядра программы Lift на микроконтроллер Siemens SAB 80C515. При этом в качестве среды разработки программного обеспечения для микроконтроллеров используется продукт Keil μ Vision 2.

Опишем эту методику.

1. Создать каталог, в котором будет размещен переносимый проект (в рассматриваемом примере он назван Hard). Скопировать в каталог файлы, реализующие автоматы (A0.h, A0.cpp, A1.h, A1.cpp, A2.h и A2.cpp). Изменить расширение файлов с «cpp» на «c». Создать файл «Common.h» и файл проекта μ Vision (Lift.uv2) для требуемого микроконтроллера, добавив в последний все упомянутые (семь) файлов.
2. Из каждого файла с исходным кодом, имеющим расширение «c», удалить директивы #include, кроме

тех, которые включают одноименные заголовочные файлы или заголовочные файлы стандартных библиотек. Во все файлы с исходным кодом следует добавить директиву «`#include "Common.h"`», а в файл `Common.h` скопировать необходимые общие определения макросов, структур данных из файла `stdafx.h` и т.п.

3. Преобразовать определения классов в заголовочных файлах. Для этого удаляются ключевые слова `class`, а методы преобразуются в функции с именами вида «<имя класса>_<имя функции>». Например, методы `Step` для трех рассматриваемых автоматов будут преобразованы в функции `A0_Step`, `A1_Step` и `A2_Step`. Кроме того, необходимо удалить из определений макросы «`DECLARE_NO_...`» и удалить или преобразовать функции протоколирования, перенаправив их вывод;
4. События, устройства и другие члены классов преобразуются по аналогии с методами, как указано в предыдущем пункте. В результате они преобразуются в константы и статические переменные с определенными префиксами в названии. При условии сохранения уникальности имен, префиксы можно опустить. Таким образом, для того, чтобы обратиться к некой сущности, бывшей до преобразования членом класса, следует записать имя класса, за ним – подчеркивание (вместо точки) и имя члена класса. Кроме того, для автомата A_i необходимо добавить в заголовочный файл переменную «`static int Ai_y`», в которой будет храниться номер состояния автомата. Объявления объектов, реализующих вложенные автоматы, необходимо поменять на объявления их главных функций, как внешних (`extern`).
5. В файлах с исходным кодом необходимо заменить подстроку «`::`» на символ «`_`». В результате функции-

члены класса получают требуемые имена. Внутри файла, реализующего автомат A_i , следует добавить префикс « $A_i_$ » для обращений к переменной состояния, событиям, функциям входных и внутренних переменных, а также функциям выходных воздействий. Это, почти всегда, можно сделать, выполнив в текстовом редакторе замену подстроки « y » на подстроку « A_i_y » (в режиме «только слово целиком»), а также произведя следующие замены: « e » на « A_i_e », « x » на « A_i_x », « t » на « A_i_t », а « z » на « A_i_z ».

6. Обращение к переменной состояния головного автомата, которое в объектном случае осуществляется посредством переменной `Nest`, следует преобразовать в обращение к внешней переменной, объявленной в заголовочном файле. При вызове вложенного автомата, обращение к функции-члену «`DoNested(e, &Ai)`» следует заменить на вызов функции «`Ai_Step(e)`». В общем случае, при необходимости обратиться к устройству, событию, состоянию или главной функции другого автомата, они должны быть объявлены с помощью директивы `extern`. Поэтому в рассматриваемом примере в файле `A0.h` должны присутствовать объявления следующих внешних переменных:

```
extern const int A1_e2;
extern int A1_y;
extern int A2_y;
```

7. Необходимо привести все конструкции в соответствие с особенностями микроконтроллера. Например, для рассматриваемого микроконтроллера

не предусмотрен тип данных `bool`. Поэтому в файле `Common.h` должны появиться следующие строки:

```
typedef int bool;  
static const int true=1;  
static const int false=0;
```

На этом завершается преобразование ядра объектно-ориентированной программы в процедурную. Одним из важнейших свойств изложенной методики является то, что все указанные выше замены и преобразования могут быть выполнены автоматически.

Для окончательного построения программы необходимо изменить или переписать функции входных и внутренних переменных, а также функции выходных воздействий, в которых осуществляются такие операции, как, например, взаимодействие с реальными устройствами, обращения к портам ввода-вывода и т.п.

2.8. Выводы по главе 2

Предложенный подход к построению системы управления лифтом обеспечивает создание хорошо документированного и легко модифицируемого проекта.

Методика преобразования объектно-ориентированной программы в процедурную позволяет, отладив первую из них, достаточно легко перенести ее в управляющее устройство, например, на базе микроконтроллера. На сайте <http://is.ifmo.ru> в разделе «Проекты» размещена программа `Lift`, а также ядро программного обеспечения системы управления лифтом для микроконтроллера `Siemens SAB 80C515`, полученное из программы `Lift` с помощью описанной выше методики переноса.

Кроме работы [40], использованной авторами в качестве прототипа, известны работы [55, 56], в которых также рассмотрена разработка программного обеспечения для задачи управления лифтом.

Как и в настоящей работе, в них используются объектно-ориентированное проектирование и программирование. Однако в работе [55] лифт проектируется для трехэтажного здания, а в [56] – для двухэтажного. В настоящей работе, как и в прототипе, моделируется лифт для пятиэтажного здания, что является более сложным.

Кроме того, в работах [55, 56] на этаже в каждый момент времени может находиться не более одного человека, что резко упрощает логику поведения системы. В настоящей работе число людей на этажах практически не ограничено.

К недостаткам работы [55] относится также и то, что состояния автомата «Управление лифтом» выделяются на основании нескольких несвязанных между собой характеристик (наличие пассажиров, степень открытия дверей и направление движения лифта). Это привело к построению весьма сложного автомата, содержащего одиннадцать состояний, который взаимодействует с рядом других автоматов. В настоящей работе выделяются только три автомата (по пять состояний), причем каждый из них отвечает за определенную составляющую системы.

Еще один недостаток работы [55] состоит в том, что при реализации автоматов объектный подход не используется в должной мере. При этом процедурная реализация каждого автомата «обертывается» в метод одного и того же класса.

Недостатком работы [56] является то, что в ней UML-проектирование и программирование на C++ не связаны формально.

Предлагаемый подход лишен указанных «минусов», за счет совместного использования преимуществ объектного и автоматного подходов.

В заключение работы, отметим, что в настоящее время все шире применяется технология объектно-ориентированного проектирования, названная Rational Unified Process [57]. Существуют и другие технологии, например, изложенные в работах [56, 58], а также в настоящей документации.

Авторы надеются, что, ознакомившись с предлагаемым подходом, читатели согласятся со словами Б. Гейтса, что «за последние двадцать лет мир изменился», по крайней мере, в области искусства программирования лифта.

Д. Кнут тоже не «стоит на месте». Он модернизировал модель MIX, разработав новую архитектуру MMIX [59]. Однако эти усовершенствования не коснулись области проектирования программ.

2.9. Листинги реализации автоматов

2.9.1. Файл Automat.h - Заголовочный файл базового класса

```

////////////////////////////////////
// Средства для программирования в рамках Switch-технологии
// Версия 1.0

////////////////////////////////////
// Вспомогательные макроопределения

// Определить в автомате пустые функции протоколирования
#define DECLARE_NO_LOG virtual void WriteLog(CString,int){} virtual void\
WriteLogOnStateChanged(int,int,int){}

// Определить пустую функцию протоколирования начала работы
#define DECLARE_NO_LOG_ON_START virtual void WriteLogOnStart(int,int,int){}

// Определить пустую функцию протоколирования при смене состояний
#define DECLARE_NO_LOG_ON_STATE_CHANGED virtual void\
WriteLogOnStateChanged(int,int,int){}

// Определить пустую функцию протоколирования завершения работы
#define DECLARE_NO_LOG_ON_FINISH virtual void WriteLogOnFinish(int,int){}

// Определить пустую функцию шага
#define DECLARE_NO_STEP virtual void Step(int){}

// Определить в автомате пустую функцию пост-шага
#define DECLARE_NO_POSTSTEP virtual void PostStep(int){}

```

```

// Отключить протоколирование в теле метода, если оно было включено.
// Это требуется, когда из метода вызываются другие методы, которые могут писать
// что-то в протокол. Будьте осторожны: используется локальная переменная "__b",
// чтобы можно было потом восстановить протоколирование с помощью
// макроопределения SWITCH_LOG_OFF
#define SWITCH_LOG_OFF bool __b=DoLog; DoLog=false;

// Восстановить протоколирование, отключенное SWITCH_LOG_OFF. Использовать
// последние два макроопределения необходимо в одном и том же методе: сначала
// SWITCH_LOG_OFF, а затем SWITCH_LOG_ON
#define SWITCH_LOG_ON DoLog=__b;

////////////////////////////////////
// Класс AutoService - класс, содержащий сервисную функцию

class AutoService
{
public:
    // Генерирует строку из n пробелов для организации протоколирования
    static CString IndentSpaces(int n);
};

////////////////////////////////////
// Класс Automat - базовый для автоматов

class Automat
{
public:
    int y;           // Переменная, хранящая состояние автомата
    int y_old;      // Переменная, хранящая состояние, в котором автомат начал
                    // последнюю итерацию
    int BaseIndent; // Минимальный базовый сдвиг для всех записей в протоколе
    bool DoLog;     // Писать ли протокол? Вот в чем вопрос! J
    Automat* Nest;  // Указатель на автомат, запустивший данный

    Automat(bool DoLog=false); // Конструктор
    void Main(int e); // Главная функция автомата

// Основные функции
protected:
    // Шаг автомата. Необходимо переопределять в наследниках
    virtual void Step(int e)=0;

    // Пост-шаг автомата, выполняемый только в случае перехода из состояния в
    // состояние (только для автоматов Мура и Мура-Мили). Необходимо
    // переопределять в наследниках
    virtual void PostStep(int e)=0;

public:
    // Выполняет шаг автомата A с событием e
    void DoNested(int e, Automat* A);

// Вспомогательные функции
public:
    // Записать в протокол строку s с отступом indent. Необходимо
    // переопределять в наследниках
    virtual void WriteLog(CString s, int indent=0)=0;

protected:
    // Записать в протокол информацию о том, что автомат запущен с событием e
    // (indent - величина отступа). Необходимо переопределять в наследниках
    virtual void WriteLogOnStart(int y,int e,int indent=0)=0;

    // Записать в протокол информацию о переходе автомата из состояния y_old в
    // состояние y (indent - величина отступа). Необходимо переопределять в
    // наследниках
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0)=0;

```

```

    // Записать в протокол информацию о том, что автомат завершил работу в
    // состоянии y (indent - величина отступа). Необходимо переопределять в
    // наследниках
    virtual void WriteLogOnFinish(int y,int indent=0)=0;

// Объекты управления. Добавить в наследниках
protected:

// Вызываемые автоматы. Добавить в наследниках
protected:

// События. Добавить в наследниках
protected:

// Входные переменные. Добавить в наследниках
protected:

// Внутренние переменные. Добавить в наследниках
private:

// Выходные воздействия. Добавить в наследниках
protected:
};

```

2.9.2. Файл Automat.cpp - Файл реализации базового класса

```

#include "stdafx.h"
#include "Automat.h"

////////////////////////////////////
// Реализация класса AutoService

// Генерирует строку из n пробелов для организации протоколирования
CString AutoService::IndentSpaces(int n)
{
    CString str="";
    for (int i=0;i<n;i++) str+=" ";
    return str;
}

////////////////////////////////////
// Реализация базового класса Automat

Automat::Automat(bool DoLog) // Конструктор класса
{
    y=0;
    Nest=NULL;
    BaseIndent=0;
    this->DoLog=DoLog;
}

void Automat::Main(int e) // Главная функция автомата
{
    y_old=y; // Запомнить состояние перед началом
             // итерации
    WriteLogOnStart(y,e); // Протоколировать факт начала итерации
    Step(e); // Выполнить шаг
    WriteLogOnStateChanged(y,y_old); // Протоколировать факт перехода или
                                     // сохранения состояния
    if (y!=y_old) PostStep(e); // Если переход был - выполнить пост-шаг
    WriteLogOnFinish(y); // Протоколировать факт окончания итерации
}

// Выполняет шаг автомата A, являющегося вложенным в данный
void Automat::DoNested(int e, Automat* A)
{
    A->Nest=this;
    A->Main(e);
}

```

```

}

// Присвоение значений идентификаторам событий
/*...*/

////////////////////////////////////
// Шаг автомата
////////////////////////////////////

////////////////////////////////////
// Входные переменные
////////////////////////////////////

////////////////////////////////////
// Выходные воздействия
////////////////////////////////////

////////////////////////////////////
// Внутренние переменные
////////////////////////////////////

2.9.3. Файл A0.h - Заголовочный файл класса автомата A0

////////////////////////////////////
// Автомат, реализующий функциональность кабины лифта
////////////////////////////////////

class A0 : public Automat
{
protected:
    virtual void Step(int e); // Шаг автомата
    DECLARE_NO_POSTSTEP;     // Пост-шаг не нужен

public:
    virtual void WriteLog(CString s,int indent=0);
protected:
    virtual void WriteLogOnStart(int y,int e,int indent=0);
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0);
    virtual void WriteLogOnFinish(int y,int indent=0);

// Вспомогательные определения
public:
    // Значения индикатора движения
    typedef enum StateValueType {NEUTRAL, GOINGUP, GOINGDOWN} StateValues;

    // Тип структуры, реализующей хранилище вызовов
    typedef struct CallsStoreType
    {
        int up[5],down[5],car[5];
        bool newcalls;
    } CallsStore;

// Устройства
public:
    unsigned floor;      // Счётчик этажей
    unsigned timer;      // Таймер
    int inactivitytimer; // Таймер бездействия
    CallsStore calls;    // Хранилище вызовов
    StateValues state;   // Индикатор движения

// Автоматы, к которым будет обращаться данный
public:
    A1 aA1; // Автомат A1
    A2 aA2; // Автомат A2

// События
public:
    const static int e0; // Пребывание в состоянии покоя, инициализация
    const static int e1; // Событие по умолчанию

// Входные переменные
protected:
    // Переменные, получаемые от счётчика этажей

```

```

int x1_0();          // Кабина лифта находится на втором (базовом) этаже?

// Переменные, получаемые от хранилища вызовов
int x2_0();          // Нужно ли открыть двери на текущем этаже?
int x2_1();          // Поступили ли новые вызовы (на предыдущем шаге)?

// Переменные, получаемые от индикатора движения
int x3_0();          // Значение индикатора - «Neutral»
int x3_1();          // Значение индикатора - «GoingUp»
int x3_2();          // Значение индикатора - «GoingDown»

// Переменные, получаемые от таймера
int x4_0();          // Таймер сработал?

// Переменные, получаемые от таймера бездействия
int x5_0();          // Таймер бездействия сработал?

// Выходные воздействия
protected:
// Воздействия на счётчик
void z1_0();         // Установить значение счётчика этажей равное двум
void z1_1();         // Увеличить значение счётчика этажей на один
void z1_2();         // Уменьшить значение счётчика этажей на один

// Воздействия на хранилище вызовов
void z2_0();         // Обнулить хранилище вызовов
void z2_1();         // Отметить в хранилище обработку текущего вызова

// Воздействия на индикатор движения
void z3_0();         // Установить значение «Neutral»
void z3_1();         // Изменить значение, в соответствии с текущей
                    // ситуацией
void z3_2();         // Направить лифт на второй (базовый) этаж

// Воздействия на таймер
void z4_0(unsigned n); // Установить значение таймера равное n

// Воздействия на таймер бездействия
void z5_0();         // Выключить таймер бездействия
void z5_1();         // Включить таймер бездействия
void z5_2();         // Выключить таймер бездействия, если он не сработал

// Внутренние переменные
private:
// Переменные, связанные с индикатором движения
StateValues i3_0(); // Вычислить состояние индикатора движения,
                    // адекватное текущей ситуации
};

```

2.9.4. Файл A0.cpp – Файл реализации класса автомата A0

```

#include "stdafx.h"
#include "../Lift.h"
#include "Automat.h"
#include "A1.h"
#include "A2.h"
#include "A0.h"

// Присвоение значений идентификаторам событий
const A0::e0=0;
const A0::e1=1;

////////////////////////////////////
// Шаг автомата A0

void A0::Step(int e)
{
    switch (y)

```

```

{
case 0: // Состояние бездействия
    if (e==e0) {
        DoNested(e, &aA1);
        DoNested(e, &aA2);
        z1_0();
        z2_0();
        z3_0();
    } else
    if (x2_1()) {
        z4_0(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TAfterRest));
        z3_1();
        z5_0();
        y=1;
    } else
    if (!x1_0()) {
        z3_2();
        y=1;
    }
break;
case 1: // Принятие решений
    if (x5_0()&&x3_0()) {
        y=0;
    } else
    if (x5_0()&&!x3_0()&&x1_0()) {
        z3_0();
        y=0;
    } else
    if (x2_0()) {
        y=2;
    } else
    if (x3_1()) {
        DoNested(aA1.e2, &aA1);
        z1_1();
        z5_2();
        y=3;
    } else
    if (x3_2()) {
        DoNested(aA1.e2, &aA1);
        z1_2();
        z5_2();
        y=4;
    }
break;
case 2: // Работа с дверьми
    DoNested(e, &aA2);
    if (aA2.y==0) {
        z4_0(0);
        z3_1();
        z2_1();
        z5_1();
        y=1;
    }
break;
case 3: // Подняться
    DoNested(e, &aA1);
    if (aA1.y==0&&(x2_0() || (x5_0()&&x1_0()))) {
        z4_0(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TUpBraking));
        y=1;
    } else
    if (aA1.y==0&&!x2_0()) {
        z1_1();
    }
break;
case 4: // Опуститься
    DoNested(e, &aA1);
    if (aA1.y==0&&(x2_0() || (x5_0()&&x1_0()))) {
        z4_0(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDownBraking));
        y=1;
    } else

```

```

        if (aA1.y==0&&!x2_0()) {
            z1_2();
        }
        break;
    }
}

// Записать в протокол строку s (indent - отступ)
void A0::WriteLog(CString s,int indent)
{
    if (!DoLog) return;
    ((CLiftApp*)AfxGetApp())-
    >m_LogDlg.m_sLog+=AutoService::IndentSpaces(indent+BaseIndent)+s+"\r\n";
}

// Записать в протокол информацию о том, что автомат запущен с событием e
// (indent - отступ)
void A0::WriteLogOnStart(int y,int e,int indent)
{
    if (!DoLog) return;
    CString str;
    CString s;
    if (e==e0) s="e0"; else
    if (e==e1) s="e1";
    str.Format("Автомат А0 запущен в состоянии %d с событием %s",y,s);
    WriteLog(str,indent);
}

// Записать в протокол информацию о том, что автомат перешёл из состояния y_old
// в y (indent - отступ)
void A0::WriteLogOnStateChanged(int y,int y_old,int indent)
{
    if (!DoLog) return;
    CString str;
    if (y==y_old) str.Format("Автомат А0 сохранил состояние %d",y); else
        str.Format("Автомат А0 перешёл из состояния %d в состояние %d",y_old,y);
    WriteLog(str,indent);
}

// Записать в протокол информацию о том, что автомат завершил работу в состоянии
// y (indent - отступ)
void A0::WriteLogOnFinish(int y,int indent)
{
    if (!DoLog) return;
    CString str;
    str.Format("Автомат А0 завершил работу в состоянии %d",y);
    WriteLog(str,indent);
}

////////////////////////////////////
// Входные переменные

// Переменные, получаемые от счётчика этажей

int A0::x1_0()
{
    int b=(floor==2);
    CString str;
    str.Format
        ("x1_0: Кабина лифта находится на втором (базовом) этаже? - вернул
%d",b);
    WriteLog(str,3);
    return b;
}

// Переменные, получаемые от хранилища вызовов

int A0::x2_0()
{
    StateValues st=i3_0();
}

```


// Воздействия на счётчик

```
void A0::z1_0()
{
    WriteLog("z1_0: Установить значение счётчика этажей равное 2",3);
    floor=2;
}
```

```
void A0::z1_1()
{
    WriteLog("z1_1: Увеличить значение счётчика этажей на 1",3);
    floor++;
}
```

```
void A0::z1_2()
{
    WriteLog("z1_2: Уменьшить значение счётчика этажей на 1",3);
    floor--;
}
```

// Воздействия на хранилище вызовов

```
void A0::z2_0()
{
    WriteLog("z2_0: Обнулить хранилище вызовов",3);
    for (int i=0; i<5; i++) {
        calls.car[i]=0;
        calls.up[i]=0;
        calls.down[i]=0;
    }
    calls.newcalls=false;
}
```

```
void A0::z2_1()
{
    WriteLog("z2_1: Отметить в хранилище обработку текущего вызова",3);
    calls.car[floor]=0;
    if (state!=GOINGDOWN) calls.up[floor]=0;
    if (state!=GOINGUP) calls.down[floor]=0;
}
```

// Воздействия на индикатор движения

```
void A0::z3_0()
{
    WriteLog("z3_0: Установить значение \"NEUTRAL\"",3);
    state=NEUTRAL;
}
```

```
void A0::z3_1()
{
    WriteLog("z3_1: Изменить значение, согласно текущей ситуации",3);
    state=i3_0();
}
```

```
void A0::z3_2()
{
    WriteLog("z3_2: Направить лифт на второй (базовый) этаж",3);
    state=((int)floor>2?GOINGDOWN:GOINGUP);
    if ((int)floor==2) state=NEUTRAL;
}
```

// Воздействия на таймер

```
void A0::z4_0(unsigned n)
{
    CString str; str.Format("z4_0(%d): Установить значение таймера равное %d",n,n); WriteLog(str,3);
    timer=n;
}
```

```

}

// Воздействия на таймер бездействия
void A0::z5_0()
{
    WriteLog("z5_0: Выключить таймер бездействия",3);
    inactivitytimer=-1;
}

void A0::z5_1()
{
    WriteLog("z5_1: Включить таймер бездействия",3);
    inactivitytimer=TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TInactive);
}

void A0::z5_2()
{
    WriteLog("z5_2: Выключить таймер бездействия, если он не сработал",3);
    SWITCH_LOG_OFF // Временное отключение протоколирования
    if (!x5_0()) z5_0();
    SWITCH_LOG_ON // Восстановление протоколирования
}

////////////////////////////////////
// Внутренние переменные

// Переменные, связанные с индикатором движения
A0::StateValues A0::i3_0()
{
    int i;
    switch (state)
    {
        case GOINGUP:
            if (calls.up[floor]!=0) return GOINGUP;
            for (i=floor+1; i<=5; i++)
            {
                if (i==5) break;
                if (calls.car[i]!=0 || calls.up[i]!=0 || calls.down[i]!=0)
                    break;
            }
            if (i==5)
            {
                bool r=false;
                for (i=0; i<=(int)floor-1; i++)
                    r|=((calls.car[i]!=0)|(calls.up[i]!=0)|(calls.down[i]!=0));
                if (r) state=GOINGDOWN; else if (!x5_0()) state=NEUTRAL;
            }
            break;

        case GOINGDOWN:
            if (calls.down[floor]!=0) return GOINGDOWN;
            for (i=floor-1; i>=-1; i--)
            {
                if (i==-1) break;
                if (calls.car[i]!=0 || calls.up[i]!=0 || calls.down[i]!=0)
                    break;
            }
            if (i==-1)
            {
                bool r=false;
                for (i=floor+1; i<=4; i++)
                    r|=((calls.car[i]!=0)|(calls.up[i]!=0)|(calls.down[i]!=0));
                if (r) state=GOINGUP; else if (!x5_0()) state=NEUTRAL;
            }
            break;

        case NEUTRAL:
            if (calls.up[floor]!=0||calls.down[floor]!=0) return NEUTRAL;
    }
}

```

```

        for (i=0; i<=5; i++)
        {
            if (i==5) break;
            if (calls.car[i]!=0 || calls.up[i]!=0 || calls.down[i]!=0)
                break;
        }
        if (i==5 || i==(int)floor) return NEUTRAL;
        state=((int)floor>i?GOINGDOWN:GOINGUP);
        break;
    }
    return state;
}

```

2.9.5. Файл A1.h - Заголовочный файл класса автомата A1

```

////////////////////////////////////
// Автомат, реализующий функциональность двигателя,
// перемещающего лифт между этажами

class A1 : public Automat
{
protected:
    virtual void Step(int e); // Шаг автомата
    DECLARE_NO_POSTSTEP;     // Пост-шаг

public:
    virtual void WriteLog(CString s,int indent=0);
protected:
    virtual void WriteLogOnStart(int y,int e,int indent=0);
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0);
    virtual void WriteLogOnFinish(int y,int indent=0);

// Устройства
public:
    double rope; // Анализатор длины троса
    bool accelerated; // Анализатор разгона

// Автоматы, к которым будет обращаться данный

// События
public:
    const static int e0; // Пребывание в состоянии покоя, инициализация
    const static int e1; // Событие по умолчанию
    const static int e2; // Старт кабины "с места"

// Входные переменные
protected:
    // Переменные, получаемые от анализатора длины троса
    int x1_1(); // Кабина достаточно разогнана для движения вверх?
    int x1_2(); // Кабина достаточно разогнана для движения вниз?
    int x2_1(); // Кабина поднялась на этаж вверх?
    int x2_2(); // Кабина опустилась на этаж вниз?

    // Переменные, получаемые от анализатора разгона
    int x3_0(); // Лифт разогнан?

// Выходные воздействия
protected:
    // Воздействия на анализатор длины троса
    void z1_0(); // Установить значение, соответствующее 2-му этажу
    void z1_1(); // Поднять кабину на единицу длины троса при разгоне
    void z1_2(); // Опустить кабину на единицу длины троса при разгоне
    void z2_1(); // Поднять кабину на единицу длины троса
    void z2_2(); // Опустить кабину на единицу длины троса

    // Воздействия на анализатор разгона
    void z3_0(); // Обнулить анализатор скорости
};

```

2.9.6. Файл A1.cpp - Файл реализации класса автомата A1

```

#include "stdafx.h"
#include "../Lift.h"
#include "Automat.h"
#include "A1.h"
#include "A2.h"
#include "A0.h"
#include "math.h"

// Присвоение значений идентификаторам событий
const A1::e0=0;
const A1::e1=1;
const A1::e2=2;

////////////////////////////////////
// Шаг автомата A1

void A1::Step(int e)
{
    switch (y)
    {
        case 0: // Покой кабины
            if (e==e0) {
                z1_0();
            } else
            if (e==e2) {
                z3_0();
            } else
            if (Nest->y==3&&!x3_0()) {
                z1_1();
                y=1;
            } else
            if (Nest->y==3&&x3_0()) {
                z2_1();
                y=2;
            } else
            if (Nest->y==4&&!x3_0()) {
                z1_2();
                y=3;
            } else
            if (Nest->y==4&&x3_0()) {
                z2_2();
                y=4;
            }
            break;
        case 1: // Разгон при движении вверх
            if (!x1_1()) {
                z1_1();
            } else
            if (x1_1()) {
                z2_1();
                y=2;
            }
            break;
        case 2: // Движение вверх
            if (!x2_1()) {
                z2_1();
            } else
            if (x2_1()) {
                y=0;
            }
            break;
        case 3: // Разгон при движении вниз
            if (!x1_2()) {
                z1_2();
            } else
            if (x1_2()) {
                z2_2();
            }
    }
}

```

```

        y=4;
    }
    break;
    case 4: // Движение вниз
        if (!x2_2()) {
            z2_2();
        } else
        if (x2_2()) {
            y=0;
        }
        break;
    }
}

// Записать в протокол строку s (indent - отступ)
void A1::WriteLog(CString s,int indent)
{
    if (!DoLog) return;
    ((CLiftApp*)AfxGetApp()->m_LogDlg.m_sLog+=
        AutoService::IndentSpaces(indent+BaseIndent)+s+"\r\n";
}

// Записать в протокол информацию о том, что автомат запущен с событием e
// (indent - отступ)
void A1::WriteLogOnStart(int y,int e,int indent)
{
    if (!DoLog) return;
    CString str;
    CString s;
    if (e==e0) s="e0"; else
    if (e==e1) s="e1"; else
    if (e==e2) s="e2";
    str.Format("Автомат A1 запущен в состоянии %d с событием %s",y,s);
    WriteLog(str,indent);
}

// Записать в протокол информацию о том, что автомат перешёл из состояния y_old
// в y (indent - отступ)
void A1::WriteLogOnStateChanged(int y,int y_old,int indent)
{
    if (!DoLog) return;
    CString str;
    if (y==y_old) str.Format("Автомат A1 сохранил состояние %d",y);
    else str.Format("Автомат A1 перешёл из состояния %d в состояние
%d",y_old,y);
    WriteLog(str,indent);
}

// Записать в протокол информацию о том, что автомат завершил работу в состоянии
// y (indent - отступ)
void A1::WriteLogOnFinish(int y,int indent)
{
    if (!DoLog) return;
    CString str;
    str.Format("Автомат A1 завершил работу в состоянии %d",y);
    WriteLog(str,indent);
}

////////////////////////////////////
// Входные переменные

// Переменные, получаемые от анализатора длины троса

int A1::x1_1()
{
    int b=(rope==5.0-((A0*)Nest)->floor-1.0/
        (TO(((CLiftApp*)AfxGetApp()->m_TODlg.m_TUp)+1)));
    CString str;
    str.Format
        ("x1_1: Кабина достаточно разогнана для движения вверх? - вернул %d",b);
}

```

```

    WriteLog(str,3);
    return b;
}

int A1::x1_2()
{
    int b=(rope==3.0-((A0*)Nest)->floor+1.0/
        (TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDown)+1));
    CString str;
    str.Format
        ("x1_2: Кабина достаточно разогнана для движения вниз? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

int A1::x2_1()
{
    int b=(rope==4.0-((A0*)Nest)->floor);
    CString str;
    str.Format("x2_1: Кабина поднялась на этаж вверх? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

int A1::x2_2()
{
    int b=(rope==4.0-((A0*)Nest)->floor);
    CString str;
    str.Format("x2_2: Кабина опустилась на этаж вниз? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

// Переменные, получаемые от анализатора разгона

int A1::x3_0()
{
    int b=(accelerated==true);
    CString str;
    str.Format("x3_0: Лифт разогнан? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

////////////////////////////////////
// Выходные воздействия

// Воздействия на анализатор длины троса

void A1::z1_0()
{
    WriteLog("z1_0: Установить значение, соответствующее 2-му этажу",3);
    rope=2.0;
}

void A1::z1_1()
{
    WriteLog("z1_1: Поднять кабину на единицу длины троса при разгоне",3);
    rope-=1.0/((TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TUp)+1)*
        TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TStarting));
    if (floor(rope+1)-rope>1.0/(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TUp)+1))
        rope=5.0-((A0*)Nest)->floor-1.0/
            (TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TUp)+1);
}

void A1::z1_2()
{
    WriteLog("z1_2: Опустить кабину на единицу длины троса при разгоне",3);
    rope+=1.0/((TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDown)+1)*
        TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TStarting));
}

```

```

        if (rope-floor(rope)>1.0/(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDown)+1))
            rope=3.0-((A0*)Nest)->floor+1.0/
                (TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDown)+1);
    }

void A1::z2_1()
{
    WriteLog("z2_1: Поднять кабину на единицу длины троса",3);
    accelerated=true;
    rope-=1.0/(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TUp)+1);
    if (4.0-((A0*)Nest)->floor>rope) rope=4.0-((A0*)Nest)->floor;
}

void A1::z2_2()
{
    WriteLog("z2_1: Опустить кабину на единицу длины троса",3);
    accelerated=true;
    rope+=1.0/(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDown)+1);
    if (4.0-((A0*)Nest)->floor<rope) rope=4.0-((A0*)Nest)->floor;
}

// Воздействия на анализатор разгона

void A1::z3_0()
{
    WriteLog("z3_0: Обнулить анализатор скорости",3);
    accelerated=false;
}

```

2.9.7. Файл A2.h – Заголовочный файл класса автомата A2

```

////////////////////////////////////
// Автомат, реализующий функциональность системы двигателей,
// обеспечивающих открывание дверей

class A2 : public Automat
{
protected:
    virtual void Step(int e); // Шаг автомата
    DECLARE_NO_POSTSTEP;     // Пост-шаг не нужен

public:
    virtual void WriteLog(CString s,int indent=0);
protected:
    virtual void WriteLogOnStart(int y,int e,int indent=0);
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0);
    virtual void WriteLogOnFinish(int y,int indent=0);

// Устройства
public:
    unsigned timer; // Таймер
    double mover[5]; // Система двигателей

// Автоматы, к которым будет обращаться данный
public:

// События
public:
    const static int e0; // Пребывание в состоянии покоя, инициализация
    const static int e1; // Событие по умолчанию

// Входные переменные
protected:
    // Переменные, получаемые от системы двигателей, открывающих двери
    int x1_0(); // Что-то мешает дверям на текущем этаже закрыться?
    int x1_1(); // Дверь на текущем этаже полностью открыта?
    int x1_2(); // Дверь на текущем этаже полностью закрыта?

    // Переменные, получаемые от таймера

```

```

    int x2_0(); // На таймере 0?

// Выходные воздействия
protected:
    // Воздействия на систему двигателей, открывающих двери
    void z1_0(); // Закрыть все двери на всех этажах
    void z1_1(); // Открыть дверь на текущем этаже на единицу
    void z1_2(); // Закрыть дверь на текущем этаже на единицу

    // Воздействия на таймер
    void z2_0(unsigned n); // Установить значение таймера равное n
};

```

2.9.8. Файл A2.cpp - Файл реализации класса автомата A2

```

#include "stdafx.h"
#include "../Lift.h"
#include "Automat.h"
#include "A2.h"
#include "A1.h"
#include "A0.h"

// Присвоение значений идентификаторам событий
const A2::e0=0;
const A2::e1=1;

////////////////////////////////////
// Шаг автомата A2

void A2::Step(int e)
{
    switch (y)
    {
        case 0: // Двери закрыты
            if (e==e0) {
                z1_0();
            } else
            if (Nest->y==2) {
                z1_1();
                y=1;
            }
            break;
        case 1: // Открытие дверей
            if (!x1_1()) {
                z1_1();
            } else
            if (x1_1()) {
                z2_0(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TDoorsTimeout));
                y=2;
            }
            break;
        case 2: // Двери открыты
            if (x2_0()) {
                z1_2();
                y=3;
            }
            break;
        case 3: // Закрытие дверей
            if (x1_0()) {
                z1_1();
                y=4;
            } else
            if (!x1_2()) {
                z1_2();
            } else
            if (x1_2()) {
                y=0;
            }
            break;
    }
}

```

```

        case 4: // Повторное открытие дверей
            if (!x1_1()) {
                z1_1();
            } else
            if (x1_1()) {
                z2_0(TO(((CLiftApp*)AfxGetApp())->m_TODlg.m_TWaitTimeout));
                y=2;
            }
            break;
        }
    }

// Записать в протокол строку s (indent - отступ)
void A2::WriteLog(CString s,int indent)
{
    if (!DoLog) return;
    ((CLiftApp*)AfxGetApp())->m_LogDlg.m_sLog+=
        AutoService::IndentSpaces(indent+BaseIndent)+s+"\r\n";
}

// Записать в протокол информацию о том, что автомат запущен с событием e
// (indent - отступ)
void A2::WriteLogOnStart(int y,int e,int indent)
{
    if (!DoLog) return;
    CString str;
    CString s;
    if (e==e0) s="e0"; else
    if (e==e1) s="e1";
    str.Format("Автомат A2 запущен в состоянии %d с событием %s",y,s);
    WriteLog(str,indent);
}

// Записать в протокол информацию о том, что автомат перешёл из состояния y_old
// в y (indent - отступ)
void A2::WriteLogOnStateChanged(int y,int y_old,int indent)
{
    if (!DoLog) return;
    CString str;
    if (y==y_old) str.Format("Автомат A2 сохранил состояние %d",y); else
        str.Format("Автомат A2 перешёл из состояния %d в состояние %d",y_old,y);
    WriteLog(str,indent);
}

// Записать в протокол информацию о том, что автомат завершил работу в состоянии
// y (indent - отступ)
void A2::WriteLogOnFinish(int y,int indent)
{
    if (!DoLog) return;
    CString str;
    str.Format("Автомат A2 завершил работу в состоянии %d",y);
    WriteLog(str,indent);
}

////////////////////////////////////
// Входные переменные

// Переменные, получаемые от системы двигателей, открывающих двери

int A2::x1_0()
{
    int b(((CLiftApp*)AfxGetApp())->m_MenMoving!=0);
    CString str;
    str.Format
        ("x1_0: Что-то мешает дверям на текущем этаже закрыться? - вернул
%d",b);
    WriteLog(str,3);
    return b;
}

```

```

int A2::x1_1()
{
    int b=(mover[((A0*)Nest)->floor]==1.0);
    CString str;
    str.Format("x1_1: Дверь на текущем этаже полностью открыта? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

int A2::x1_2()
{
    int b=(mover[((A0*)Nest)->floor]==0.0);
    CString str;
    str.Format("x1_2: Дверь на текущем этаже полностью закрыта? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

// Переменные, получаемые от таймера

int A2::x2_0()
{
    int b=(timer==0);
    CString str;
    str.Format("x2_0: На таймере 0? - вернул %d",b);
    WriteLog(str,3);
    return b;
}

////////////////////////////////////
// Выходные воздействия

// Воздействия на систему двигателей, открывающих двери

void A2::z1_0()
{
    WriteLog("z1_0: Закрыть все двери на всех этажах",3);
    for (int i=0; i<=4; i++) mover[i]=0.0;
}

void A2::z1_1()
{
    WriteLog("z1_1(): Открыть дверь на текущем этаже на единицу",3);
    mover[((A0*)Nest)->floor]+=1.0/TO(((CLiftApp*)AfxGetApp())-
>m_TODlg.m_TDoors);
    if (mover[((A0*)Nest)->floor]>1.0) mover[((A0*)Nest)->floor]=1.0;
}

void A2::z1_2()
{
    WriteLog("z1_2: Закрыть дверь на текущем этаже на единицу",3);
    mover[((A0*)Nest)->floor]-=1.0/TO(((CLiftApp*)AfxGetApp())-
>m_TODlg.m_TDoors);
    if (mover[((A0*)Nest)->floor]<0.0) mover[((A0*)Nest)->floor]=0.0;
}

// Воздействия на таймер

void A2::z2_0(unsigned n)
{
    CString str; str.Format("z2_0(%d): Установить значение таймера равное
%d",n,n);
    WriteLog(str,3);
    timer=n;
}

```

3. Другие подходы к объектно-ориентированному

программированию с явным выделением состояний

Кроме рассмотренных выше подходов к реализации объектно-ориентированных программ с явным выделением состояний создан также ряд других подходов для решения этой задачи, которые были разработаны и апробированы в ходе выполнения проектов, опубликованных на сайте <http://is.ifmo.ru> (раздел «Проекты»). Созданные подходы могут быть классифицированы следующим образом.

1. Автоматы, как методы классов (глава 1). Этот подход основан на процедурном стиле программирования [10] и может быть назван «обертыванием автоматов в классы».
2. Автоматы, как классы без использования класса базового автомата [60].
3. Автоматы, как классы с использованием класса базового автомата. Этот подход основан на совместном применении всех преимуществ, как объектного, так и автоматного стилей программирования. При этом автоматы разрабатываются, как наследники класса, реализующего базовую функциональность. Базовый класс и другие необходимые классы помещаются в библиотеку, предоставляемую разработчику.
 - 3.1. В главе 2 приводится пример простейшей библиотеки классов для разработки программного обеспечения в рамках объектно-ориентированного программирования с явным выделением состояний. В нее входят два класса: класс `Automat` для дальнейшего наследования от него создаваемых автоматов и вспомогательный класс `AutoService`. Первый из них обеспечивает, в частности, выполнение шага и пост-шага автомата, автоматическое протоколирование и т.п. Класс

AutoService используется при протоколировании.

Подход, похожий на изложенный в главе 2, описан в работе [61].

3.2. В работе [58] предложена библиотека STOOl (Switch-Technology Object Oriented Library), в которой не только автомат, но и его логические составные части имеют соответствующие базовые классы. Кроме того, библиотека предоставляет возможность разработки многопоточного программного обеспечения.

3.3. В работе [62] предложена еще одна библиотека для объектно-ориентированной реализации автоматов, названная Auto-Lib.

3.4. В работе [63] предложена библиотека, позволяющая «собирать» простые автоматы из наследников базовых классов «состояние автомата» и «переход между состояниями». Эта библиотека позволяет обеспечить изоморфизм между текстом программы и графом переходов при наличии в нем групповых переходов.

4. Использование паттернов проектирования [18].

Наряду с использованием библиотек при объектно-ориентированной реализации автоматов могут применяться и разрабатываться паттерны проектирования.

4.1. Описанный в работе [64] паттерн Automaton позволяет проектировать и реализовывать программное обеспечение, пользуясь классами, реализующими следующие понятия: «состояние», «условие перехода», «действие», «переход», «дуга перехода», «автомат». При этом класс, реализующий последнее понятие, является базовым

для разрабатываемых автоматов и содержит в себе их основную логику.

4.2. Использование паттерна State. Данный паттерн, описанный в работе [18], реализует абстракцию «состояние». Для реализации конкретного состояния необходимо разработать наследника базового класса State и переопределить в нем функцию переходов.

Похожий подход рассмотрен в работе [65]. В ней для каждого автомата создан базовый класс «Состояние», от которого наследуются конкретные классы, реализующие состояния данного автомата. Переходы между состояниями обеспечиваются базовыми классами состояний, а непосредственно осуществляются в классах наследниках.

5. Интерпретируемое описание автоматов.

5.1. В работах [66, 67] предложен подход, позволяющий автоматически преобразовывать графы переходов в текстовое описание в формате XML. На языке Java разработана среда исполнения полученного XML описания. При этом сначала указанное описание однократно и целиком преобразуется в соответствующее внутреннее объектное представление программы. В результате образуется система, состоящая из среды исполнения и объектного представления программы. При этом каждое входное и выходное воздействие реализуется вручную, в соответствии с его функциональностью. Упомянутая система при появлении события анализирует его и входные переменные и выполняет выходные воздействия, а также запускает вложенные автоматы.

5.2. В работе [68] предложено использовать XML для автоматного описания динамики изменения внешнего

вида виртуального устройства –
видеопроектировщика Crystal Player
(<http://www.crystalplayer.com>).

6. Механизм обмена сообщениями и автоматы.

6.1. В ходе выполнения работ [69-71] по реализации классического параллельного алгоритма синхронизации цепи стрелков выяснилось, что автоматы, построенные по предложенному в работе [10] шаблону, реализующему событийно-управляемые автоматы, который состоит из двух операторов `switch`, не позволяет реализовать взаимодействующие параллельные процессы. Для решения этой задачи в работе [69] было предложено использовать механизм обмена сообщениями. Для этого была разработана библиотека SWMEM (Switch Message Exchange Mechanism). При этом в шаблон для реализации автоматов были внесены следующие изменения: шаг работы автомата разделен на три этапа (выбор перехода, совершение действий на переходе и обновление переменной состояния); введены переменная для учета приоритетов условий на дугах графа переходов и переменная для хранения выбранного действия и последующего его выполнения.

6.2. В работе [72] механизм обмена сообщениями между параллельно «расположенными» автоматами реализуется за счет введения такой сущности, как «общая шина». Этот подход позволяет однотипно реализовать разнотипные по своей природе алгоритмы, которые могут быть иерархическими, вложенными или параллельными. Для реализации параллельно работающих автоматов было предложено

изменить шаблоны, рассмотренные в работах [10, 69], строя автомат с помощью двух функций: функции перехода-действия и функции обновления. Первая из них сначала осуществляет выходные воздействия, как в состоянии, так и на переходе, а потом определяет номер нового состояния и осуществляет выходные воздействия в нем. Вторая функция обеспечивает выполнение одинаковых действий: обновляет состояние автомата и массив поступающих ему сообщений. В общем цикле работы автоматов для их синхронизации сначала должны вызываться все функции перехода-действия, а затем – все функции обновления. После этого обновляется массив сообщений, посылаемых в «общую шину».

ЧАСТЬ II. ПРЕОБРАЗОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ В АВТОМАТНЫЕ

4. Преобразование итеративных алгоритмов в автоматные программы

Для систем логического управления и событийных систем, рассмотренных в предыдущих этапах работы, выделение состояний является естественным, так как они в значительной мере определяются физическими состояниями объектов управления.

Для вычислительных алгоритмов выделение состояний не столь естественно, однако и для этого класса задач автоматный подход может быть применен [73].

Такой подход позволяет унифицировать процесс построения структурированных программ с визуализацией их состояний, что имеет важное значение особенно для целей обучения [74]. Это не удастся обеспечить при традиционном процедурном подходе, так как в нем в качестве базовых используются понятия "условие" и "действие", а не "состояние".

Программы, построенные на основе явного выделения состояний, назовем *автоматными*.

Психологические трудности [73], возникающие при непосредственном построении алгоритмов для вычислительных задач с применением автоматов, связаны, в частности, с непривычной реализацией циклов. Эти трудности привели к тому, что в работе [75] автоматный подход был использован только для построения одного из алгоритмов поиска подстрок, в то время как для десятков других алгоритмов,

описанных в этой книге, автоматы не применялись ни при алгоритмизации, ни при реализации.

Разделим рассматриваемый класс алгоритмов на два подкласса: без использования событий (вычислительные алгоритмы) и с их использованием (событийно-вычислительные алгоритмы). Настоящая работа посвящена первому из указанных подклассов. В этом подклассе будем рассматривать схемы итеративных алгоритмов с одним входом и одним выходом, в которых через каждую вершину проходит путь от входа схемы к ее выходу (отсутствуют недостижимые вершины и бесконечные циклы).

Покажем, что схему произвольного итеративного алгоритма (программы), принадлежащую этому подклассу, можно реализовать одним оператором `do-while`, телом которого является один оператор `switch`. Последний изоморфно реализует граф переходов конечного автомата, формально построенный по исходной схеме. Отметим, что в отличие от событийных систем, в рассматриваемом подклассе алгоритмов при реализации каждого автомата используется один оператор `switch`.

Эта реализация проще получаемой при применении метода Ашкрофта-Манна [76, 77], который требует введения состояния для каждой вершины неструктурированной схемы алгоритма и приводит к построению его структурированной схемы с числом вершин, определяемым соотношением:

$$B = 4U + 3O + 2,$$

где U и O — количество условных и операторных вершин в неструктурированной схеме алгоритма, соответственно.

Подход к структурированию программ с использованием булевых признаков [77] в настоящей работе не рассматривается, так как он является эвристическим.

4.1. Изложение метода

Ниже излагается метод построения по схеме итеративного алгоритма (программы) графа переходов, по которому, в свою очередь, строится структурированная программа, являющаяся автоматной. Если задан текст программы, то для применения предлагаемого метода необходимо предварительно построить ее схему с "раскрытыми" циклами, которую назовем традиционной. Она может быть как структурированной, так и неструктурированной.

Метод состоит из этапов, перечисленных ниже.

1. Используя подход, предложенный в работе [78] для аппаратных реализаций схем алгоритмов, в заданную схему в зависимости от выбранного для ее замены типа автомата (Мура или Мили) вводятся пометки, соответствующие номерам его состояний. При этом для автоматов обоих типов начальной и конечной вершинам схемы присваивается номер 0, что обеспечивает построение "замкнутого" графа переходов. При построении автомата Мура k -й группе соединенных последовательно операторных вершин (она может состоять также и из одной вершины) присваивается номер k . При построении автомата Мили соответствующий номер присваивается точке, следующей за последней из последовательно соединенных операторных вершин группы, причем указанные точки для различных групп могут совпадать. Это приводит к тому, что автомат Мили имеет число состояний, не превышающее их число в эквивалентном автомате Мура. Номера, применяемые для построения автомата Мили будем указывать на схеме алгоритма в скобках, а номера для построения автомата Мура — без скобок. Используемая нумерация начальной и конечной вершин отличается от принятой в работах [76, 77], где

этим вершинам присваиваются различные номера, а получающийся при этом граф переходов разомкнут.

2. Обеспечение "замкнутости" графа переходов и выбранная стандартная программная реализация могут привести к необходимости введения дополнительного состояния (как для автоматов Мили, так и для автоматов Мура), если схема алгоритма содержит контур без операторных вершин и пометок других состояний.

3. В случае построения автомата Мили в схеме алгоритма за счет дублирования некоторых операторных вершин уменьшается число точек, следующих за ними, что обеспечивает сокращение числа состояний автомата этого типа.

4. В схеме программы выделяются пути между смежными точками (пометками), включая пути начинающиеся и оканчивающиеся в одной и той же точке. Для каждого пути выписываются условия перехода и выполняемые при этом действия. На основе выделенных состояний и переходов (включая петли) строится граф переходов автомата выбранного типа.

5. Построенный граф переходов изоморфно реализуется с помощью оператора `switch` языка Си или его аналога из других языков программирования.

6. Полученный фрагмент программы используется в качестве тела оператора `do-while`, условием выхода из которой является нахождение автомата в состоянии 0.

При необходимости после получения графа переходов по нему может быть построена структурированная схема программы, соответствующая оператору `do-while`, тело которого изоморфно графу переходов и содержит дешифратор состояний. Схемы с такой структурой назовем *автоматными*.

В отдельных случаях в построенной схеме может быть уменьшено число вершин за счет потери указанного изоморфизма (но не структурированности), по аналогии с тем, как это выполняется в [79] для схем, построенных методом Ашкрофта-Манна, с помощью перехода к рекурсивным структурированным схемам.

Завершив изложение метода, отметим, что подход, описанный в [78], состоит только из первого и четвертого этапов.

С помощью предлагаемого метода в настоящей работе решаются две задачи: преобразование традиционных программ (обычно структурированных) в автоматные (примеры 1-5) и преобразование неструктурированных схем алгоритмов в автоматные (примеры 6-12).

Перейдем к рассмотрению этих задач.

4.2. Примеры преобразования итеративных программ в автоматные

Пример 1. Задана программа поиска максимума в одномерном массиве (листинг 1), по которой требуется построить программу, соответствующую автомату Мура.

Листинг 1. Программа поиска максимума в одномерном массиве

```
void main ()
{
    enum          { n = 8 } ;
    int           a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
    int           max = a[0] ;
    int           i ;

    for( i = 1 ; i < n ; i++ )
        if( a[i] > max )
            max = a[i] ;
}
```

Схема этой программы, в которой без скобок указаны пометки, соответствующие состояниям автомата Мура, приведена на рис. 17.

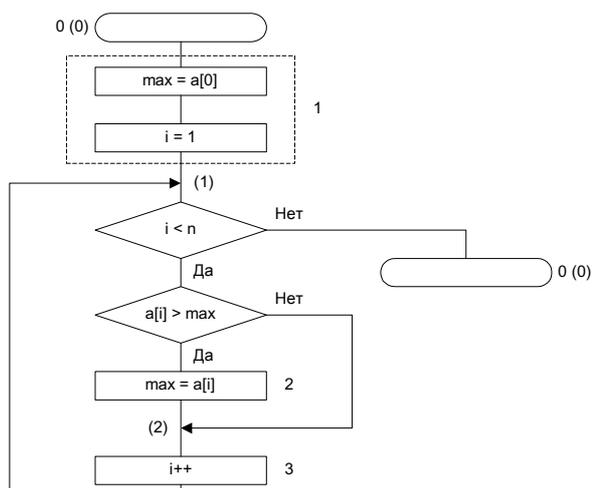


Рис. 17. Программа поиска максимума. Схема программы

Граф переходов автомата Мура с четырьмя вершинами, построенный по этой схеме, приведен на рис. 18. На этом графе пометки некоторых дуг упрощены (вычеркнуты) за счет введения приоритетов и пометки "иначе". Пометка петли исключена, так как ей при программной реализации соответствует оператор break.

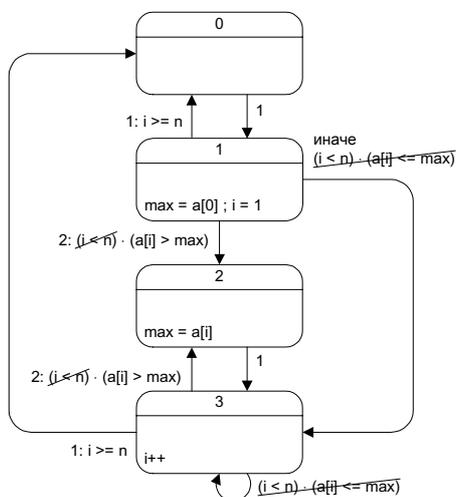


Рис. 18. Программа поиска максимума. Граф переходов автомата Мура

Используя этот граф переходов, построим автоматную программу требуемого типа (листинг 2).

Листинг 2. Программа поиска максимума в одномерном массиве, реализующая автомат Мура

```
void main()
{
    int      y = 0 ;
    enum    { n = 8 } ;
```

```

int      a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
int      max ;
int      i ;

do
  switch( y )
  {
    case 0:
                                     y = 1 ;
    break ;
    case 1:
      max = a[0] ; i = 1 ;
      if( i >= n )
        y = 0 ;
      else
        if( a[i] > max )
          y = 2 ;
        else
          y = 3 ;
    break ;
    case 2:
      max = a[i] ;
                                     y = 3 ;
    break ;
    case 3:
      i++ ;
      if( i >= n )
        y = 0 ;
      else
        if( a[i] > max )
          y = 2 ;
    break ;
  }
  while( y != 0 ) ;
}

```

Пример 2. По программе, приведенной в предыдущем примере, построить программу, соответствующую автомату Мили.

Используя пометки, указанные на рис. 17 в скобках, построим граф переходов автомата Мили с тремя состояниями (рис. 19).

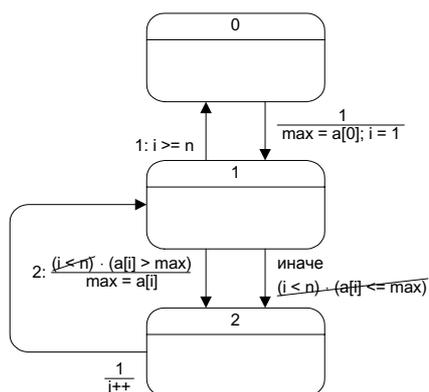


Рис. 19. Программа поиска максимума. Граф переходов автомата Мили

Используя этот граф переходов, построим автоматную программу требуемого типа (листинг 3).

Листинг 3 Программа поиска максимума в одномерном массиве, реализующая автомат Мили

```
void main()
```

```

{
  int      y = 0 ;
  enum    { n = 8 } ;
  int     a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int     max ;
  int     i ;

  do
    switch( y )
    {
      case 0:
        max = a[0] ; i = 1 ;    y = 1 ;
        break ;
      case 1:
        if( i >= n )          y = 0 ;
        else
          if( a[i] > max )
            { max = a[i] ;    y = 2 ; }
          else
            y = 2 ;
        break ;
      case 2:
        i++ ;                y = 1 ;
        break ;
    }
  while( y != 0 ) ;
}

```

Пример 3. Преобразуем схему программы на рис. 17 с целью получения автомата Мили с двумя состояниями и построим по этой схеме автоматную программу. Для этого продублируем операторную вершину с пометкой "i++" (рис. 20).

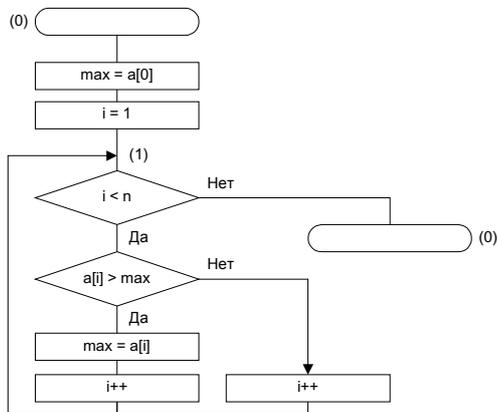


Рис. 20. Программа поиска максимума. Модифицированная схема программы

Схеме программы на рис. 20 соответствует граф переходов автомата Мили с двумя состояниями (рис. 21).

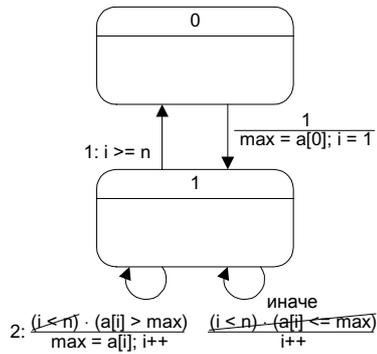


Рис. 21. Программа поиска максимума. Граф переходов автомата Мили

Приведем текст автоматной программы, реализующей этот граф переходов (листинг 4).

Листинг 4. Программа поиска максимума в одномерном массиве, реализующая автомат Мили

```

void main()
{
    int      y = 0 ;
    enum     { n = 8 } ;
    int      a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
    int      max ;
    int      i ;

    do
        switch( y )
        {
            case 0:
                max = a[0] ; i = 1 ;    y = 1 ;
                break ;
            case 1:
                if( i >= n )            y = 0 ;
                else
                    if( a[i] > max )
                        { max = a[i] ; i++ ; }
                    else
                        i++ ;
                break ;
        }
    while( y != 0 ) ;
}
  
```

Пример 4. Задана программа поиска максимума в двумерном массиве (листинг 5), по которой надо построить программу, соответствующую автомату Мили.

Листинг 5. Программа поиска максимума в двумерном массиве

```

void main()
{
    enum     { m = 2, n = 4 } ;
    int      a[m][n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
    int      max = a[0][0] ;
    int      i, j ;

    for( i = 0 ; i < m ; i++ )
        for( j = 0 ; j < n ; j++ )
            if( a[i][j] > max )
  
```

```

    max = a[i][j] ;
}

```

Если по схеме этой программы построить граф переходов автомата Мили, то он будет содержать четыре вершины. Дублирование операторной вершины с пометкой "j++" приводит к построению схемы программы, приведенной на рис. 22.

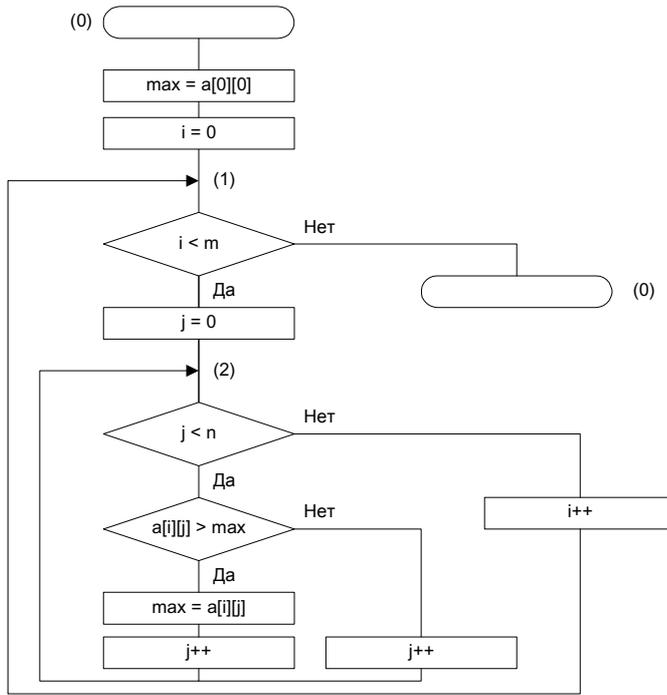


Рис. 22. Программа поиска максимума. Схема программы

Эта схема реализуется графом переходов автомата Мили с тремя состояниями (рис. 23).

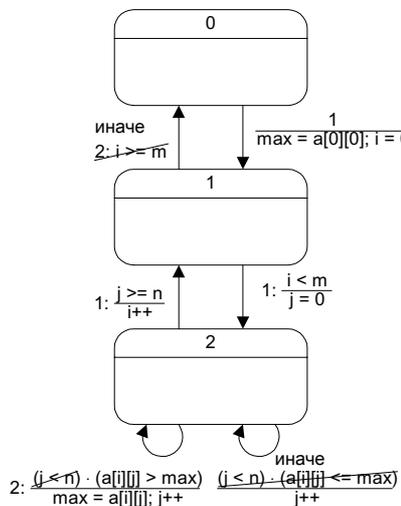


Рис. 23. Программа поиска максимума. Граф переходов автомата Мили

Приведем текст автоматной программы, построенной на основе этого графа переходов (листинг 6).

Листинг 6. Программа поиска максимума в двумерном массиве

```
void main()
{
  int      y = 0 ;
  enum     { m = 2, n = 4 } ;
  int      a[m][n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int      max = a[0][0] ;
  int      i, j ;

  do
  switch( y )
  {
    case 0:
      i = 0 ;          y = 1 ;
      break ;
    case 1:
      if( i < m ) { j = 0 ; y = 2 ; }
      else       y = 0 ;
      break ;
    case 2:
      if( j >= n ) { i++ ; y = 1 ; }
      else
        if( a[i][j] > max )
          { max = a[i][j] ; j++ ; }
        else
          j++ ;
      break ;
  }
  while( y != 0 ) ;
}
```

Пример 5. Автомат Мили, граф переходов которого приведен на рис. 23, можно декомпозировать на два автомата Мили, которые обозначим A0 и A1. Каждый из автоматов содержит по два состояния. При этом автомат A1 вложен в автомат A0 (рис. 24).

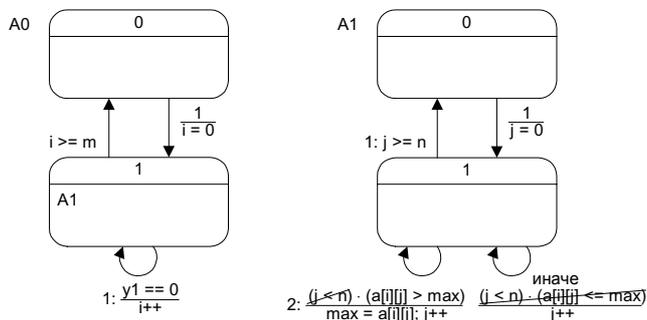


Рис. 24. Программа поиска максимума. Граф переходов автомата Мили

Эта система взаимосвязанных автоматов может быть реализована вложенными операторами switch. Однако "более

изоморфной" является программа, в которой каждый автомат реализуется отдельной функцией (листинг 7).

Листинг 7 Программа поиска максимума в двумерном массиве

```

int          y0 = 0 ;
int          y1 = 0 ;
enum        { m = 2, n = 4 } ;
int         a[m][n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
int         max ;
int         i, j ;

void main()
{
    max = a[0][0] ;
    do
        A0() ;
    while( y0 != 0 ) ;
}

void A0()
{
    switch( y0 )
    {
        case 0:
            i = 0 ;                y0 = 1 ;
            break ;
        case 1:
            A1() ;
            if( y1 == 0 ) i++
            else
                if( i >= m )        y0 = 0 ;
            break ;
    }
}

void A1()
{
    switch( y1 )
    {
        case 0:
            j = 0 ;                y1 = 1 ;
            break ;
        case 1:
            if( j >= n )            y1 = 0 ;
            else
                if( a[i][j] > max )
                    { max = a[i][j] ; j++ ; }
                else
                    j++ ;
            break ;
    }
}

```

Как отметил студент СПбГИТМО (ТУ) Макаров Н.С., алгоритм поиска максимума в массиве произвольной размерности может быть реализован автоматом Мили с двумя состояниями, если рассматривать массив любой размерности как одномерный.

Применяя предлагаемый подход для реализации алгоритма Дейкстры, предназначенного для поиска кратчайших

расстояний в положительном взвешенном графе от вершины с номером ноль до всех остальных вершин, удастся построенную традиционно структурированную программу, содержащую цикл `for`, в который вложены два таких же цикла, преобразовать в автоматную программу, состоящую из одной конструкции `do-while`, телом которой является конструкция `switch`, реализующая автомат Мили с четырьмя состояниями [80].

В заключение этого раздела работы отметим, что изложенный подход напоминает известный метод Ашкрофта-Манна [76], предназначенный для структурирования неструктурированных программ, который по этой причине (в отличие от предлагаемого) к структурированным программам не применяется.

4.3. Преобразование неструктурированных алгоритмов в автоматные программы

Предлагаемый подход при структурировании неструктурированных схем алгоритмов является более эффективным по сравнению с методом Ашкрофта-Манна ввиду явного использования автоматов разных типов и меньшего числа состояний в них. Продемонстрируем это на примерах.

Пример 6. Построить по неструктурированной схеме алгоритма (рис. 25), приведенной в [79], автоматную схему.

Используя изложенный метод, построим граф переходов автомата Мили (рис. 26).

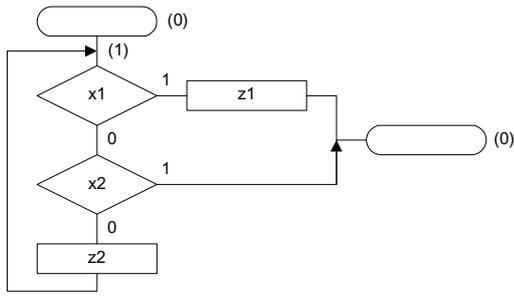


Рис. 25. Неструктурированная схема алгоритма

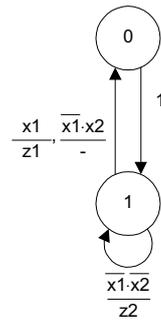


Рис. 26. Граф переходов автомата Мили

Автоматная схема алгоритма, являющаяся структурированной, которая изоморфно построена по этому графу, приведена на рис. 27.

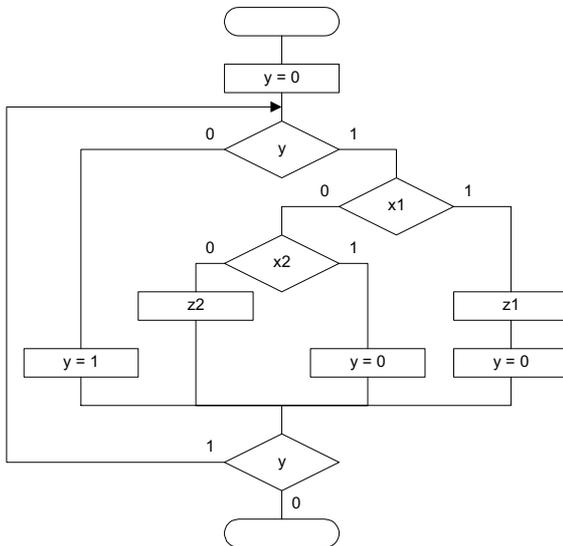


Рис. 27. Автоматная схема алгоритма

Эта схема содержит 10 условных и операторных вершин, в то время, как аналогичная схема, построенная методом Ашкрофта-Манни в [79], содержит 16 таких вершин ($U = O = 2$). Последняя схема может быть упрощена с помощью перехода к рекурсивной структурированной схеме. При этом она будет содержать 8 вершин [79].

Эта схема, не являющаяся автоматной, может быть также получена и в результате упрощения схемы, представленной на рис. 27.

Пример 7. Задана неструктурированная схема алгоритма (рис. 28). Построим графы переходов автоматов Мура (рис. 29) и Мили (рис. 30), которые изоморфно преобразуются в автоматные программы.

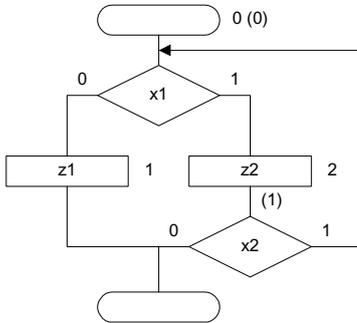


Рис. 28.
Неструктурированная
схема алгоритма

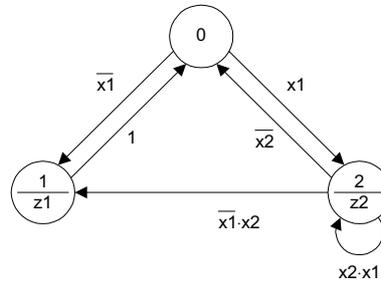


Рис. 29. Автомат Мура

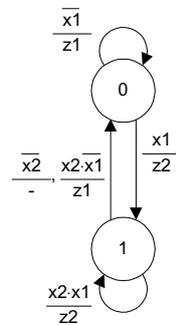


Рис. 30. Автомат
Мили

Пример 8. Задана неструктурированная схема алгоритма (рис. 31), в которой при $x_1 = x_2 = 1$ осуществляется задержка. Выполним на этом примере обоснование введения второго этапа в предлагаемый метод.

Применяя подход, изложенный в [80], построим графы переходов автоматов Мура (рис. 32) и Мили (рис. 33). Эти графы не могут быть использованы для построения автоматных программ. Так, например, автомат Мили имеет три петли, однократное исполнение двух из которых должно приводить к завершению алгоритма, а третья петля (с пометкой " x_1x_2 ") должна исполняться многократно. Однако, при принятой в настоящей работе реализации третья петля вместо многократного исполнения, будет также выполнена однократно с последующим завершением работы программы.

Полученный граф переходов автомата Мили обладает интересным свойством: он содержит только одну вершину, что характерно для автоматов без памяти. Однако, за счет умолчания выходных воздействий (прочерка) на одной из петель, этот граф описывает поведение автомата с памятью.

Кроме того отметим, что и на других петлях, указаны не все выходные переменные.

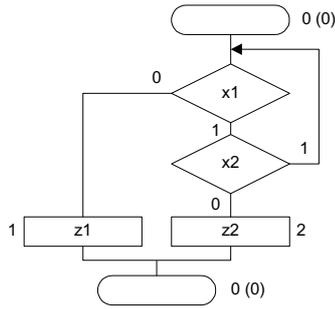


Рис. 31. Неструктурированная схема алгоритма

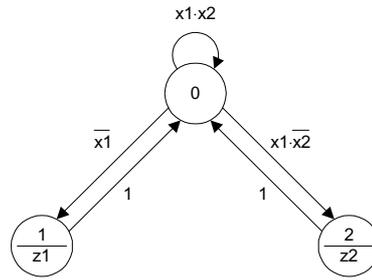


Рис. 32 Автомат Мура

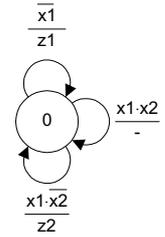


Рис. 33. Автомат Мили

Пример 9. Задана неструктурированная схема алгоритма (рис. 31). Так как она содержит контур без операторных вершин, замыкающийся на дуге, исходящей из начальной вершины, то в соответствии со вторым этапом предлагаемого метода введем пометку для дополнительного состояния автоматов Мура и Мили сразу после начальной вершины схемы (рис. 34). Отметим, что эта пометка достаточно естественна для построения автомата Мили, но нетрадиционна для автомата Мура.

Построенные графы переходов автоматов Мура и Мили приведены на рис. 35 и 20, соответственно. По этим графам, как изложено выше, могут быть построены автоматные схемы алгоритмов и автоматные программы.

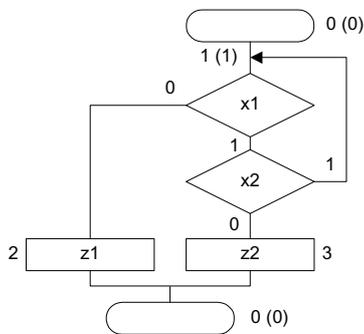


Рис. 34. Неструктурированная схема алгоритма

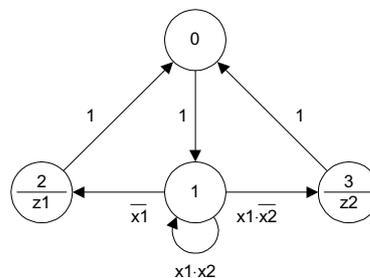


Рис. 35. Автомат Мура

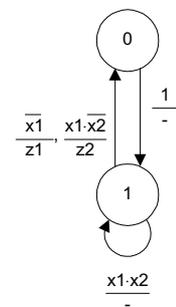


Рис. 36. Автомат Мили

Пример 10. Задана неструктурированная схема алгоритма (рис. 37), в которой при $x_2 = x_3 = 1$ запоминаются значения выходных воздействий, вычисленные в первом блоке. В отличие от предыдущего примера, наличие "пустого" контура в схеме не требует введения дополнительного состояния для автомата Мили, так как этот контур замыкается не после начальной вершины. Граф переходов автомата Мили, построенный для этой схемы, приведен на рис. 38. По этому графу, как изложено выше, могут быть построены автоматная схема алгоритма и автоматная программа.

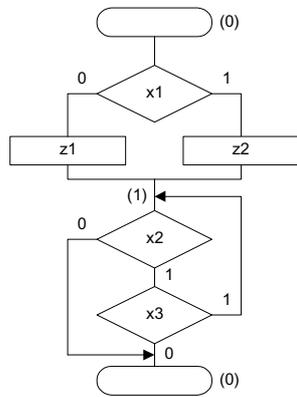


Рис. 37. Неструктурированная схема алгоритма

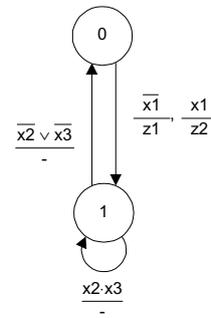


Рис. 38. Граф переходов автомата Мили

Пример 11. Задана неструктурированная и непланарная схема алгоритма (рис. 39). Граф переходов автомата Мили, построенный по этой схеме и содержащий всего лишь два состояния, приведен на рис. 40. По этому графу, как изложено выше, могут быть построены автоматная схема алгоритма и автоматная программа.

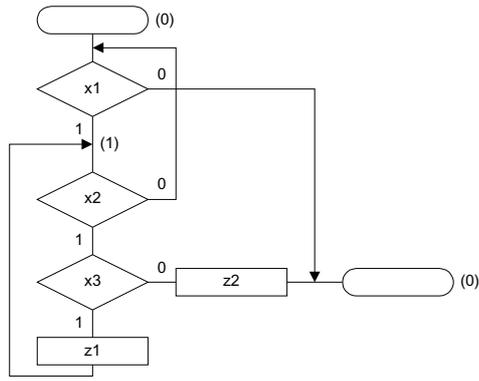


Рис. 39. Неструктурированная схема алгоритма

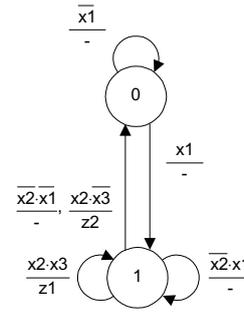


Рис. 40. Граф переходов автомата Мили

Рассмотренная схема содержит контур, замыкающийся после начальной вершины и не содержащий операторных вершин. Так как этот контур содержит пометку "(1)", соответствующую состоянию автомата Мили, то вводить дополнительное состояние не требуется.

Пример 12. Задана неструктурированная схема алгоритма, приведенная в [77] и представленная на рис. 41. Введя пометку для дополнительного состояния в соответствии со вторым этапом предлагаемого метода, можно построить граф переходов автомата Мили с четырьмя состояниями. Продублировав операторную вершину с пометкой "z3" в соответствии с третьим этапом метода (рис. 42), построим граф переходов автомата Мили с тремя состояниями (рис. 43).

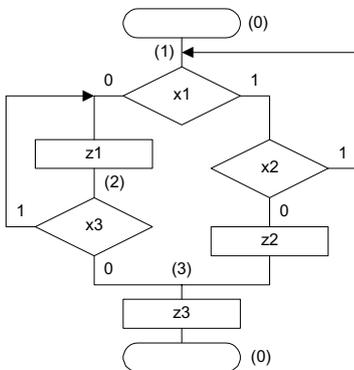


Рис. 41. Неструктурированная схема алгоритма

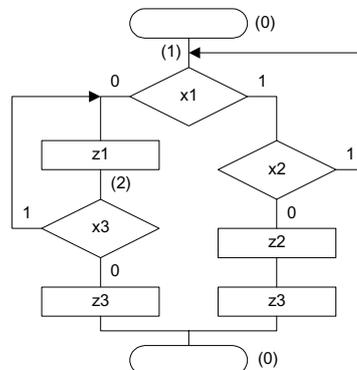


Рис. 42. Автомат Мура

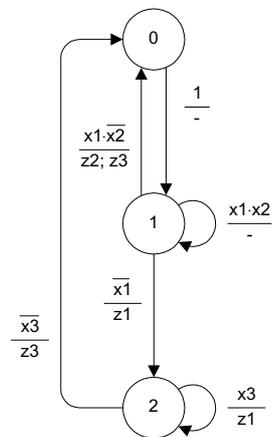


Рис. 43. Автомат Мили

Автоматная схема алгоритма, изоморфная этому графу, приведена на рис. 44.

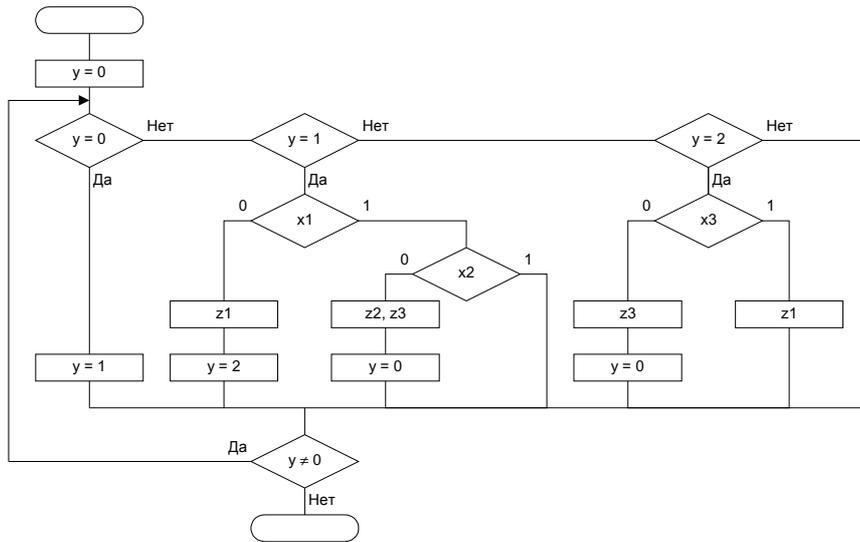


Рис. 44. Автоматная схема алгоритма

Построенная структурированная схема алгоритма содержит 16 условных и операторных вершин, в то время, как аналогичная схема, построенная методом Ашкрофта-Манна в [77] (с учетом исправления допущенной там ошибки) содержит 23 таких вершины ($U = O = 3$).

Приведенная на рис. 44 схема соответствует предлагаемой программной реализации с использованием операторов `do-while` и `switch`.

Все рассмотренные в примерах схемы алгоритмов и программ имели начальную и конечную вершины, что характерно для вычислительных алгоритмов. Предложенный метод может быть использован также и для преобразования алгоритмов, содержащих бесконечный внешний цикл. При этом граф переходов, построенный по такой схеме, будет обладать свойством недостижимости начальной вершины.

5. Преобразование программ с явной рекурсией в автоматные

В предыдущем разделе был предложен метод преобразования произвольных итеративных программ в автоматные программы. Этот метод позволяет реализовать произвольный итеративный алгоритм структурированной программой, содержащей один оператор `do-while`, телом которого является один оператор `switch`.

В работе [81] приведены примеры преобразований программ с концевой рекурсией 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. Как и ранее, будем называть программы построенные таким образом автоматными. Метод иллюстрируется примерами преобразований классических рекурсивных программ, которые приведены в следующих разделах в порядке их усложнения (факториал, числа Фибоначчи, задача о ранце, ханойские башни). При этом для задачи о ханойских башнях кроме предлагаемого метода преобразования рассматриваются и другие подходы к преобразованию рекурсивных алгоритмов в автоматные.

5.1. Изложение метода

Идея метода состоит в моделировании работы рекурсивной программы автоматной программой, явно (также, как и в работах [81]) 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 символные 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 предыдущем разделе 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 введенных 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.

В заключение изложения метода отметим, что для преобразования рекурсивных программ в автоматные может быть предложена аналогичная методика, базирующаяся на применении автоматов Мура вместо автоматов Мили. Однако эта методика и получаемые в результате графы переходов являются более громоздкими.

5.2. Вычисление факториала

Одной из простейших задач, использующих рекурсию, является вычисление факториала. Построим автоматную программу по рекурсивной программе, в которой рекурсивный вызов выделен отдельным оператором (листинг 8).

Листинг 8. Рекурсивная программа вычисления факториала

```
#include <stdio.h>

int    f = 0 ;

void fact( int n )
{
    if( n <= 1 )
        f = 1 ;
    else
    {
        fact( n-1 ) ;
        f = n * f ;
    }
}

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

Построим схему этой программы (рис. 45), а по ней — граф переходов автомата Мили (рис. 46).

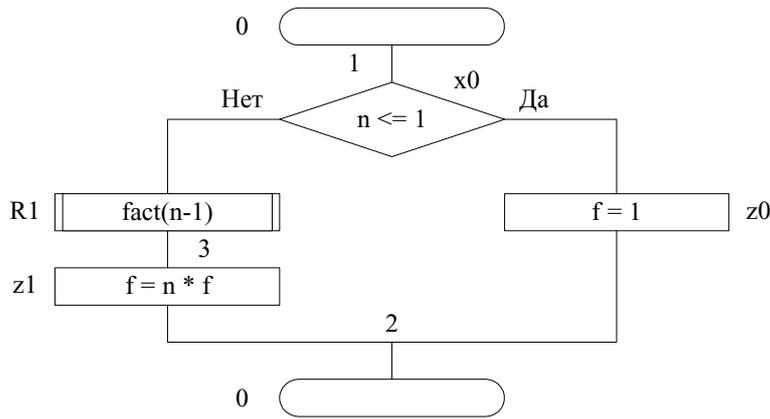


Рис. 45. Схема программы вычисления факториала

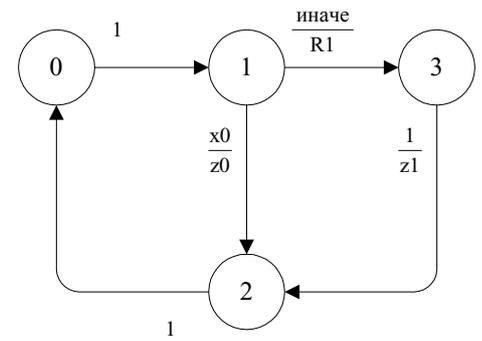


Рис. 46. Граф переходов до преобразования

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--)`, выполняемое над параметром рекурсивной функции при ее вызове. Это моделирует работу рекурсивной программы: при рекурсивном вызове значения локальных переменных запоминаются, а выполнение рекурсивной функции начинается сначала — с точки, соответствующей вершине с номером 1 графа переходов. Безусловный переход на дуге 2–0 заменяется на условие "стек пуст". Добавляется дуга 2–3, при переходе по которой выполняется действие `pop`. Эта дуга соответствует возврату из рекурсивной функции, при котором восстанавливаются значения локальных переменных и выполнение функции продолжается из точки, следующей за оператором рекурсивного вызова. Эта точка соответствует вершине с номером 3 графа переходов (рис. 47).

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. 48).

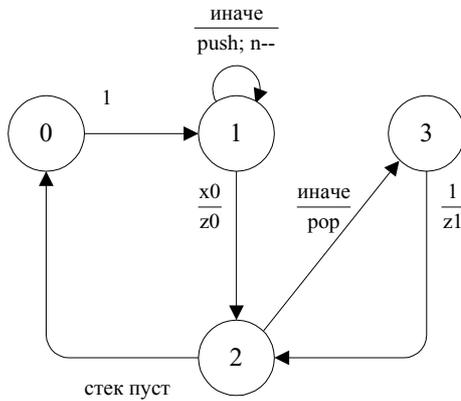


Рис. 47. Преобразованный граф переходов

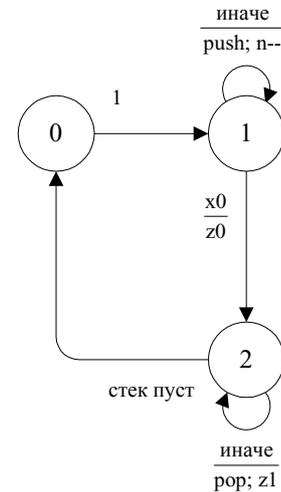


Рис. 48. Упрощенный граф переходов

Автоматная программа, построенная по упрощенному графу переходов, приведена в листинге 9.

Листинг 9. Автоматная программа вычисления факториала

```
#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\n", 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 ) ;
}

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

```

Протокол, отражающий работу со стеком, приведен в листинге 10.

Листинг 10. 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

```

5.3. Вычисление чисел Фибоначчи

Более сложной задачей является вычисление чисел Фибоначчи, программа решения которой содержит две рекурсии, выделенные в виде отдельных операторов (листинг 11).

Листинг 11. Рекурсивная программа вычисления чисел Фибоначчи

```

#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 ;
    }
}

void main()
{
    int input = 30 ;
    fibo( input ) ;
    printf( "\nФибоначчи(%d) = %d\n", input, res ) ;
}

```

Отметим, что приведенная программа весьма неэффективна ввиду повторного вычисления одних и тех же значений. Эта программа может быть усовершенствована с помощью динамического программирования [82]. Построим автоматную программу по приведенной выше рекурсивной программе.

Построим схему этой программы (рис. 49), а по ней — граф переходов автомата Мили (рис. 50).

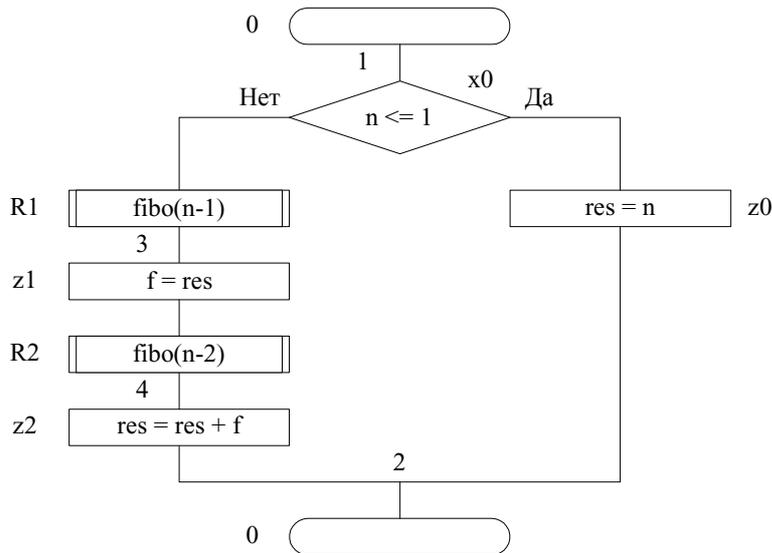


Рис. 49. Схема программы вычисления чисел Фибоначчи

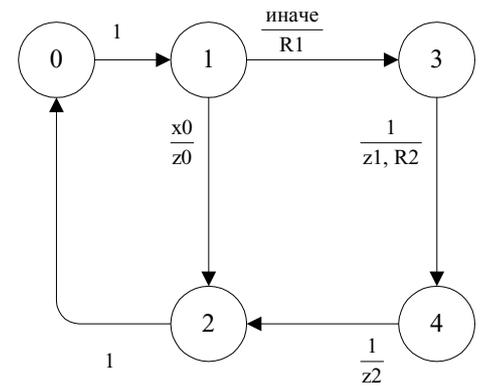


Рис. 50. Граф переходов до преобразования

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. 51).

Дуги 2–3 и 2–4 соответствуют возврату из рекурсивной функции. Так как исходная программа содержит две рекурсии

(две точки возврата из рекурсивных вызовов), необходимо запоминать в стеке не только локальные переменные n и f , но и значение переменной состояния автомата y , соответствующее точке, в которую необходимо осуществить возврат из рекурсивного вызова.

Упростим этот граф за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 52).

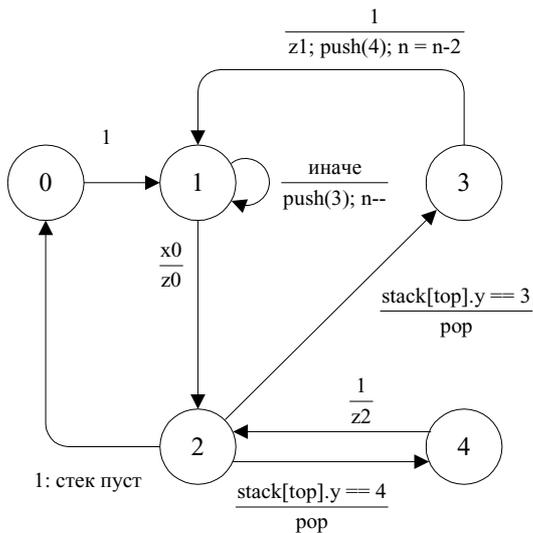


Рис. 51. Преобразованный граф переходов

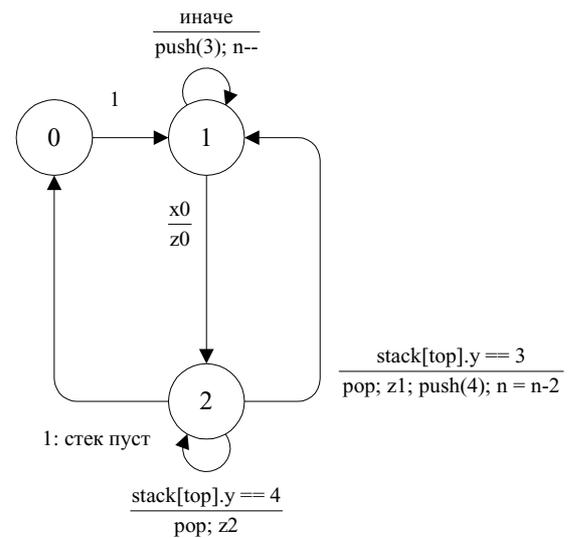


Рис. 52. Упрощенный граф переходов

Автоматная программа, построенная по упрощенному графу переходов, приведена в листинге 12.

Листинг 12. Автоматная программа вычисления чисел Фибоначчи

```
#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 ) ;

}

void main()
{
    int input = 4 ;
    fibo( input ) ;
}

```

```
printf( "\nФибоначчи(%d) = %d\n", input, res ) ;
}
```

Протокол, отражающий работу со стеком, приведен в листинге 13.

Листинг 13. Протокол, отражающий работу со стеком при вычислении чисел Фибоначчи

```
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

Фибоначчи(4) = 3
```

5.4. 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 неограничено. 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 [82]. 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 (листинг 14), эта задача рассматривается в настоящей работе для иллюстрации предлагаемого метода, так как соответствующая ей программа, построенная на основе программы, предложенной в работе [82], отличается весьма сложной логикой по сравнению с программами, приведенными выше. Также, как и в приведенных выше примерах, в рассматриваемой программе рекурсивный вызов выделен отдельным оператором.

Листинг 14. Рекурсивная программа решения задачи о ранце с использованием динамического программирования

```
#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 `itemKnown` применяется для определения содержимого ранца, полученного в результате решения задачи.

С целью обеспечения сокращения перебора на основе метода динамического программирования массив `maxKnown` используется для запоминания результатов решения подзадач. Для иллюстрации процесса перебора, функция `knap()` дополнительно содержит код, позволяющий построить дерево рекурсивных вызовов (рис. 53). В этом дереве прямоугольниками помечены вершины, соответствующие подзадачам, требующим решения, а кружками — подзадачи, результат решения которых уже известен и содержится в массиве `maxKnown`. Числа в вершинах означают объем ранца для соответствующей подзадачи.

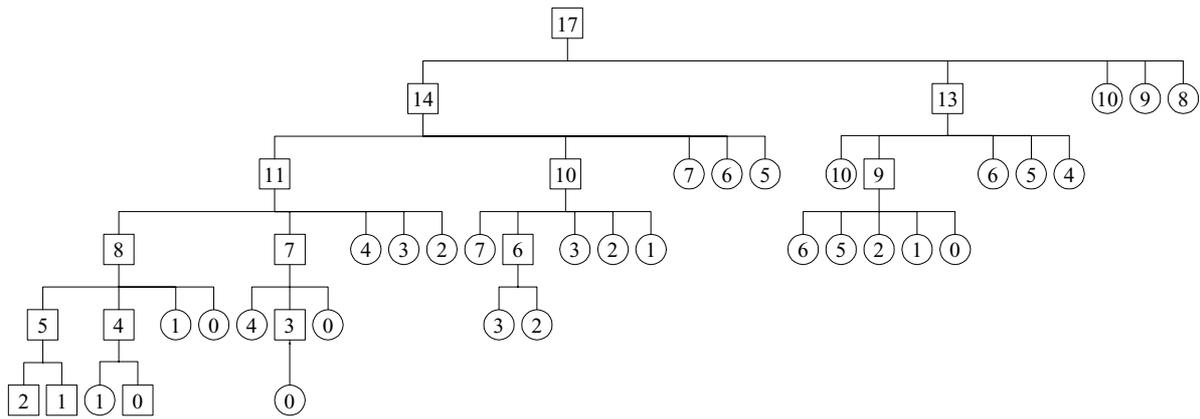


Рис. 53. Дерево рекурсивных вызовов при решении задачи о ранце

Содержимое массива `maxKnown`, хранящего результаты решения подзадач для ранцев разного объема, приведено в табл. 3. Значение "-1" означает, что решение подзадачи для указанного объема ранца не требовалось.

Таблица 3. Содержимое массива `maxKnown`

| cap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|----------------------------|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| <code>maxKnown[cap]</code> | 0 | 0 | 0 | 4 | 5 | 5 | 8 | 10 | 11 | 13 | 14 | 15 | -1 | 18 | 20 | -1 | -1 | 24 |

Проиллюстрируем особенности применения динамического программирования, частично рассмотрев процесс перебора. Перебор начинается с корня дерева, которое строится сверху вниз и слева направо. Количество подзадач, порождаемых на каждом уровне дерева (рис. 53), равно числу типов предметов, помещающихся в ранец. Например, второй уровень дерева получается в результате размещения в ранце предметов каждого типа по очереди. При размещении в ранце предметов объемом 3 и 4 выполняется дальнейшее решение подзадач для ранцев объемом 14 и 13, а для предметов объемом 7, 8 и 9 получаемые подзадачи для ранцев объемом 10, 9 и 8 оказываются уже решенными.

Отметим, что если динамическое программирование не использовать, дерево рекурсивных вызовов значительно увеличивается, так как все вершины, обозначенные кружками, заменяются поддеревьями, содержащими полный

процесс перебора, необходимый для решения соответствующих подзадач [82]. Построим схему рекурсивной программы (рис. 54), в которой не учитывается код, использованный для построения дерева рекурсивных вызовов.

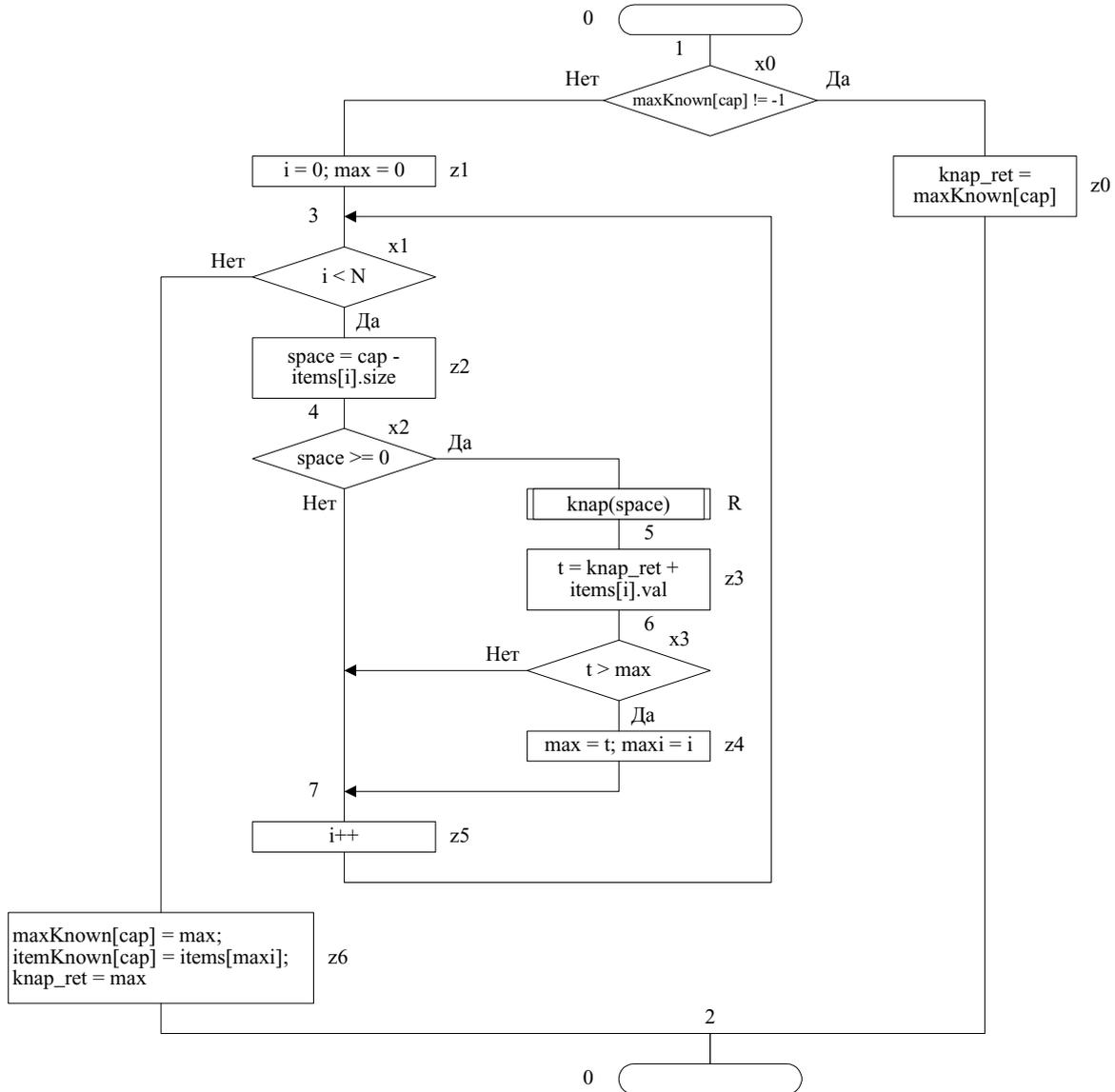


Рис. 54. Схема программы решения задачи о ранце

По схеме программы (рис. 54) построим граф переходов автомата Мили (рис. 55).

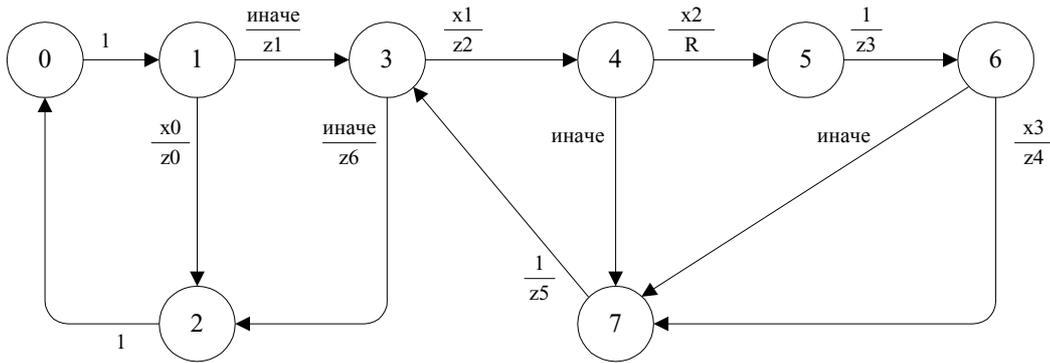


Рис. 55. Граф переходов до преобразования

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 ($\text{cap} = \text{space}$), 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. 56).

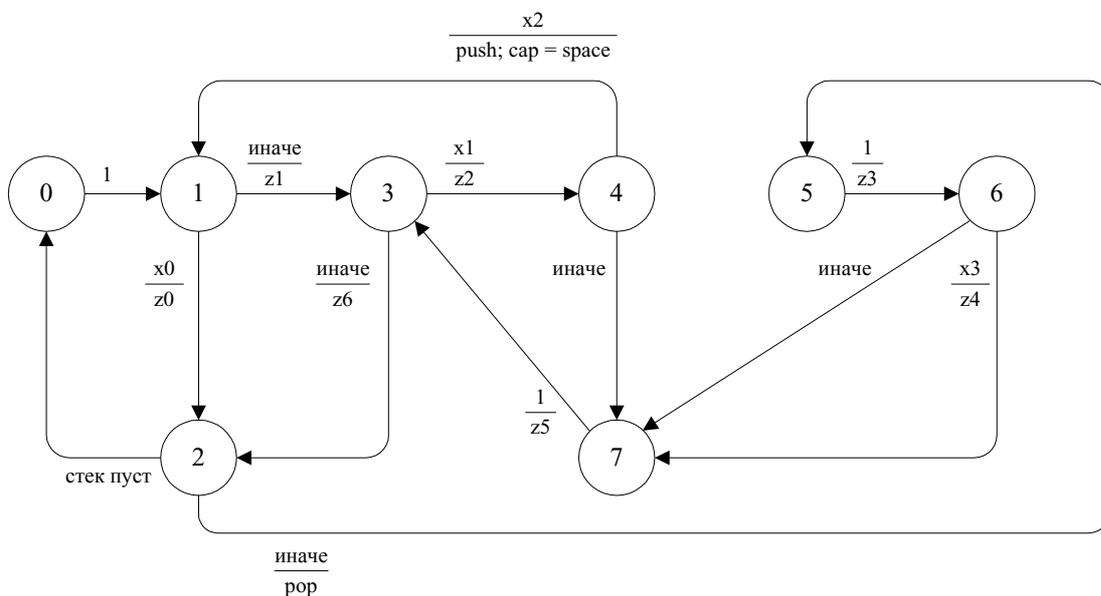


Рис. 56. Преобразованный граф переходов

Упростим этот граф за счет исключения неустойчивой вершины с номером 5 (рис. 57).

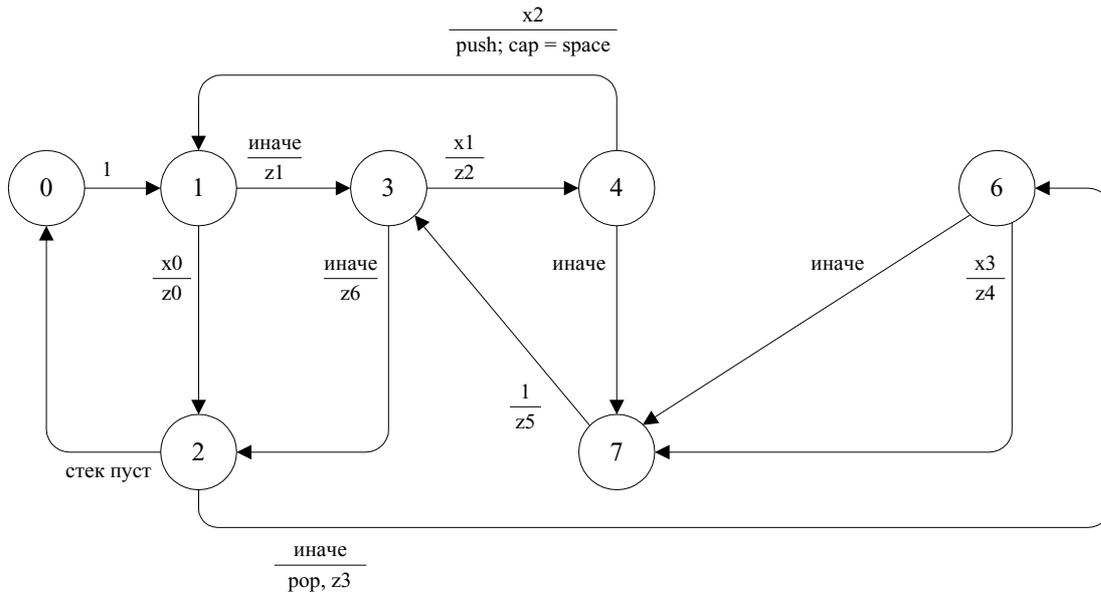


Рис. 57. Упрощенный граф переходов

Фрагмент автоматной программы, построенной по графу переходов автомата Мили (рис. 57), приведен в листинге 15. В этом фрагменте отсутствуют функции `show_knap()`, `init()` и `main()`, так как они не изменяются по сравнению с предыдущим листингом.

Листинг 15. Фрагмент автоматной программы решения задачи о ранце

```

#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 ) ;
}

```

Результат работы этой программы, идентичный результату работы рекурсивной программы, приведен в листинге 16.

Листинг 16. Результат работы автоматной программы решения задачи о ранце

```

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

5.5. Задача о ханойских башнях

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях [82], 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 перекладывании по одному диску со стержня на стержень

так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы, в конце концов, все диски оказались на другом стержне.

Рассмотрим три подхода, применение автоматов в которых позволяет либо формально переходить к итеративным алгоритмам решения этой задачи, либо строить такие алгоритмы непосредственно.

Первый из них заключается в использовании рассмотренного выше метода и обеспечивает построение автоматной программы на основе раскрытия рекурсии с применением стека. В дополнение к первому методу рассматривается подход, в котором при реализации автоматной программы используются классы. После этого изложены особенности применения к рекурсивным программам подхода, используемого в предыдущем разделе для преобразования итеративных программ в автоматные. Этот подход позволяет строить автоматнорекурсивные программы.

Второй метод также обеспечивает раскрытие рекурсии и состоит в построении автомата, осуществляющего обход дерева действий, выполняемых рекурсивной программой.

Третий метод обеспечивает решение задачи непосредственно в терминах дисков и стержней и не использует такие абстракции 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 2^N состояниями, которые возникают в процессе перекладывания дисков.

5.5.1. Классическое рекурсивное решение задачи

Известны рекурсивные алгоритмы для решения рассматриваемой задачи [82, 83]. Приведем один из таких

алгоритмов, построенный по методу "разделяй и властвуй".
Задача о перекладывании N дисков с i -ого на j -тый 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 [84], что

$$F_N = 2^N - 1.$$

Указанный процесс декомпозиции можно изобразить с помощью дерева декомпозиции (рис. 58).

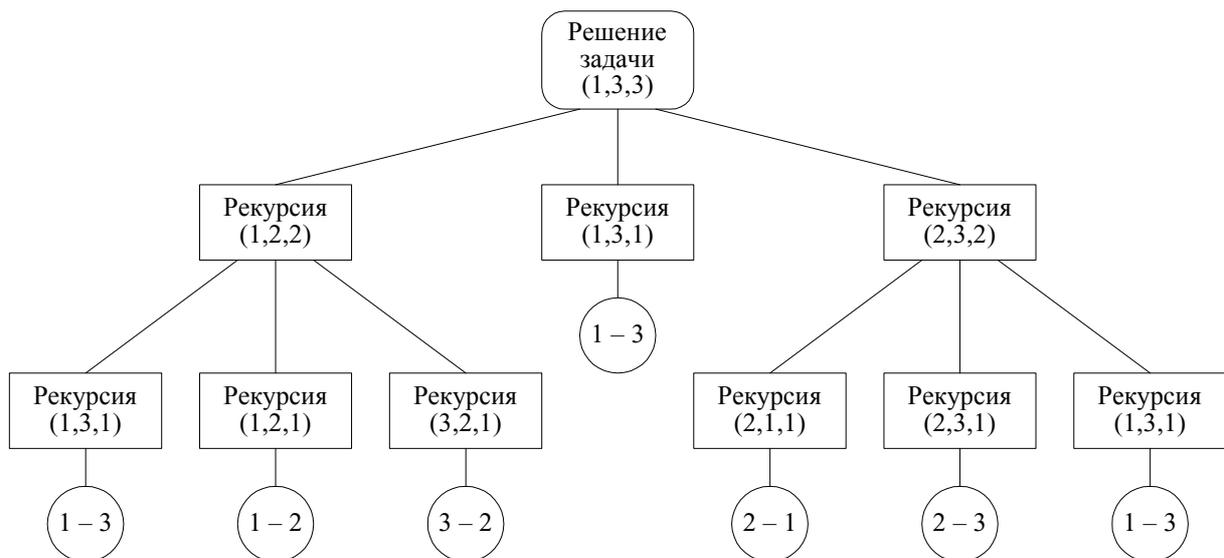


Рис. 58. Дерево декомпозиции для задачи о ханойских башнях при $N = 3$

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

Перекладывание 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 (листинг 17), реализующую рассматриваемый рекурсивный алгоритм. Ее поведение эквивалентно обходу дерева декомпозиции слева направо, причем в вершинах обозначенных кружками производятся перекладывания.

Листинг 17. Рекурсивная программа решения задачи о ханойских башнях

```
#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 ) ;
hanoi( 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. 58).

Листинг 18. Протокол работы рекурсивной программы

```

Ханой с 3 дисками:
hanoi(1,3,3)
  hanoi(1,2,2)
    hanoi(1,3,1)
      move(1,3)
      hanoi(1,2,1)
        move(1,2)
        hanoi(3,2,1)
          move(3,2)
    hanoi(1,3,1)
      move(1,3)
  hanoi(2,3,2)
    hanoi(2,1,1)
      move(2,1)
    hanoi(2,3,1)
      move(2,3)
    hanoi(1,3,1)
      move(1,3)

```

5.5.2. Моделирование рекурсии автоматной программой

Построим схему программы, приведенной в листинге 17 (рис. 59). Расставим пометки, соответствующие состояниям автомата Мили. При этом точка с номером 2 будет совпадать с точками, следующими за операторными вершинами R3 и z0.

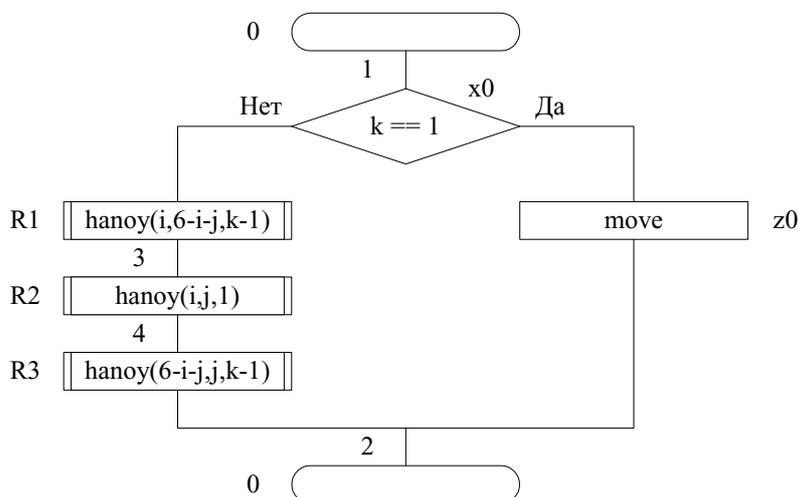


Рис. 59. Схема программы решения задачи о ханойских башнях

По этой схеме построим граф переходов автомата Мили (рис. 60).

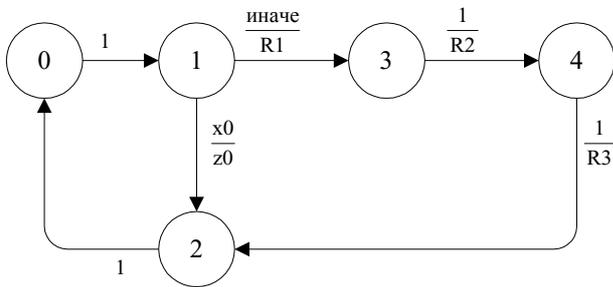


Рис. 60. Граф переходов до преобразования

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. 61 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.

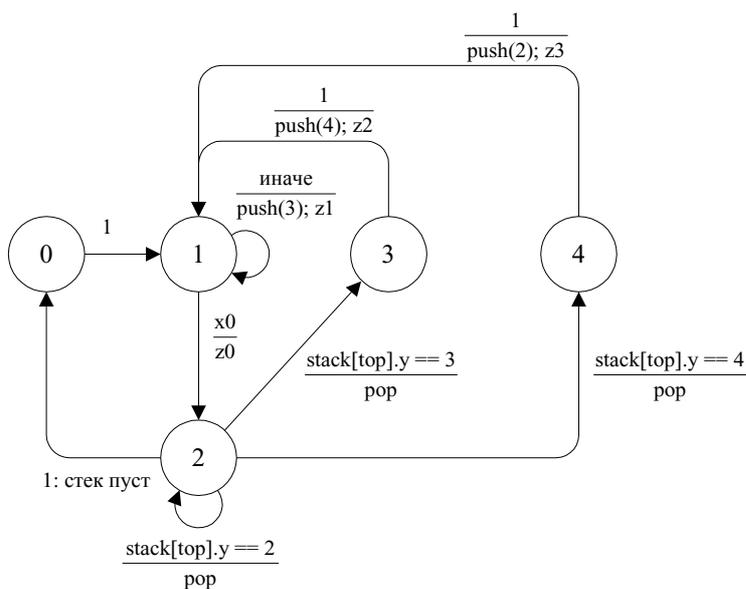


Рис. 61. Преобразованный граф переходов

Автоматная программа, построенная по графу переходов автомата Мили с пятью состояниями (рис. 61), приведена в листинге 19. В этой программе операторы, реализующие дуги, исходящие из вершины с номером 2 (за исключением дуги 2–0) могут быть 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.

Листинг 19. Автоматная программа решения задачи о ханойских башнях

```
#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 ) ;
}

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

```

Упростим граф переходов, приведенный на рис. 61, за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 62).

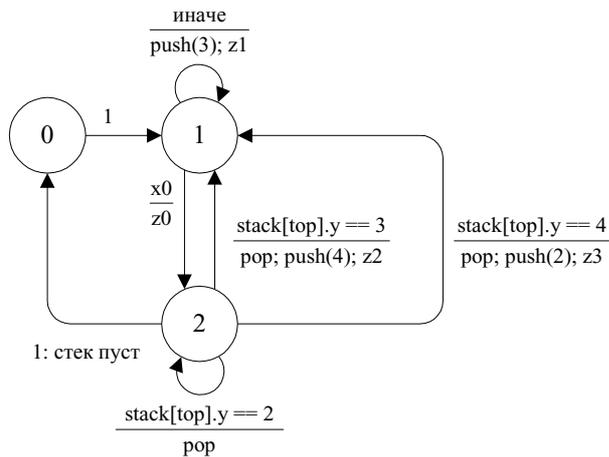


Рис. 62. Упрощенный граф переходов

При построении программы по графу переходов автомата Мили с тремя состояниями (рис. 62), функция `hanoi()` реализуется как показано в листинге 20. Остальная часть программы остается неизменной по сравнению с листингом 19. Отметим, что при такой реализации не удастся выполнить ее дальнейшее упрощение за счет замены трех условий переходов одной функцией `pop`.

Листинг 20. Автоматная функция решения задачи о ханойских башнях

```

void hanoi( 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 ) ;
}

```

Протокол, отражающий работу со стеком для программы, построенной по упрощенному графу переходов, приведен в листинге 21.

Листинг 21. Протокол, отражающий работу со стеком при решении задачи о ханойских башнях

```

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

5.5.3. Реализация рекурсивной программы автоматически-рекурсивной программой

Изложенный в предыдущем разделе метод построения автоматной программы по итеративной может быть применен и к рекурсивным программам. При этом в функциях, реализующих получаемые графы переходов, сохраняются рекурсивные вызовы. Получаемые в результате программы относятся к классу автоматически-рекурсивных.

Особенность применения указанного подхода для рекурсивных программ состоит в том, что для корректной работы автоматически-рекурсивных программ переменные состояния автоматов должны быть объявлены как локальные внутри соответствующих рекурсивных функций, в то время как для итеративных программ они могли быть и глобальными.

Автоматно-рекурсивная программа может быть построена непосредственно по графу переходов на рис. 60. Функция, реализующая этот граф переходов, приведена в листинге 22.

Листинг 22. Функция hanoi() для автоматически-рекурсивной программы

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

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

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

            case 2:
                y = 0 ;
                break ;

            case 3:
                hanoi( i, j, 1 ) ;
                y = 4 ;
```

```

break ;

case 4:
  hanoу( 6-i-j, j, k-1 ) ;      у = 2 ;
  break ;
}
while( у != 0 ) ;
}

```

Отметим, что в общем случае для применения этого подхода нет необходимости в выделении рекурсивных вызовов в отдельные операторы и введении дополнительных вершин в граф переходов, как это было сделано на рис. 59, 60.

5.5.4. Реализация автоматной программы с использованием классов

Рассмотренные в предыдущих примерах программы могут быть реализованы с применением классов. При этом область видимости используемых переменных ограничивается рамками класса, что позволяет эффективно реализовать входные и выходные воздействия в виде методов класса.

Покажем это на примере автомата (рис. 61), построенного в соответствии с рассмотренным подходом к раскрытию рекурсии. Схема связей для этого автомата приведена на рис. 63.

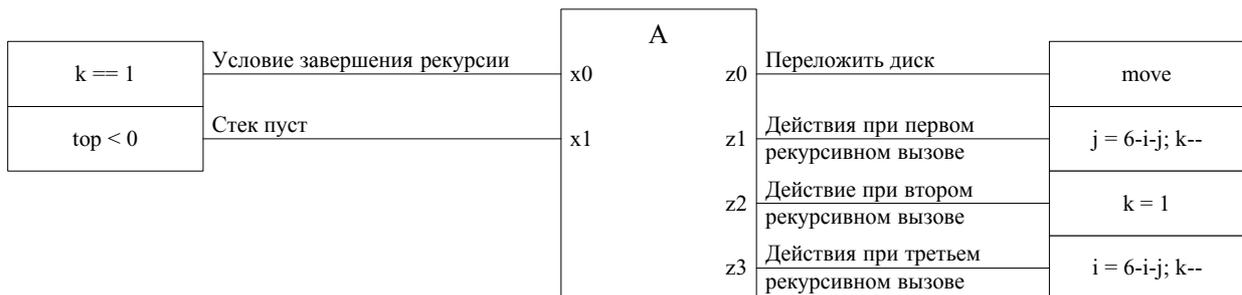


Рис. 63. Схема связей автомата

Граф переходов этого автомата приведен на рис. 64. Он отличается от графа, приведенного на рис. 61, использованием символа x1 на дуге 2-0, а также тем, что остальные дуги, исходящие из вершины с номером 2,

изображены пунктиром, так как они будут реализованы одной операцией pop.

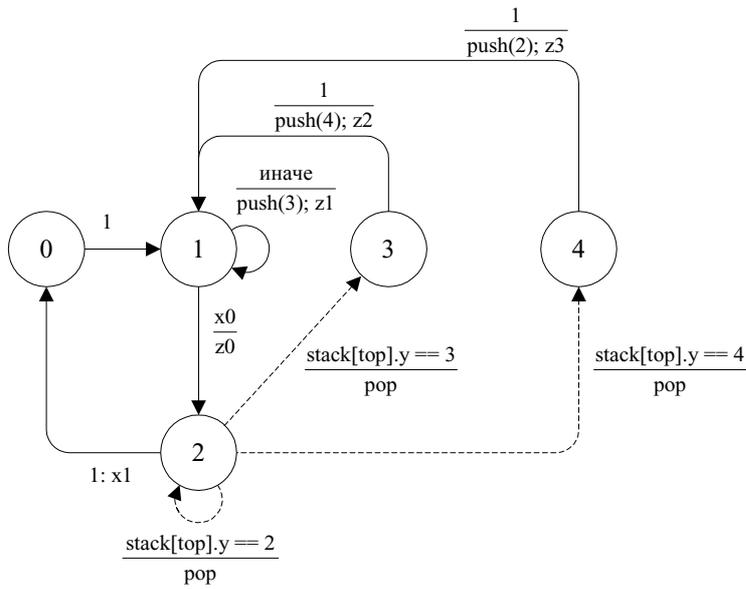


Рис. 64. Граф переходов автомата

Структурная схема класса, в котором автомат реализуется в методе hanoy(), приведена на рис. 65. К методам этого класса относятся также методы, реализующие входные переменные, выходные воздействия и операции со стеком.



Рис. 65. Схема класса

Текст программы, в которой функция, реализующая рассмотренный граф переходов, является методом класса, приведен в листинге 23.

Листинг 23. Программа решения задачи о ханойских башнях с применением классов

```
#include <stdio.h>
class CHanoy
```

```

{
public:
    CHanoy() : y(0), top(-1) {} ; // Конструктор.

    void calculate( int from, int to, int N ) ;

private:
    int y ;
    int i, j, k ;

    // Элемент стека.
    struct stackEntry
    {
        int y, i, j, k ;
        stackEntry() {} ;
        stackEntry( int y, int i, int j, int k )
            : y(y), i(i), j(j), k(k) {} ;
    } ;

    // Атрибуты и методы для работы со стеком.
    int top ;
    stackEntry stack[100] ;
    int stack_empty() { return top < 0 ; }
    void push( int y ) { stack[++top] = stackEntry ( y, i, j, k ) ; }
    void pop() ;

    void hanoy() ;

    int x0() { return k == 1 ; }
    int x1() { return stack_empty() ; }

    void z0() { printf( "%d -> %d\n", i, j ) ; }
    void z1() { j = 6-i-j ; k-- ; }
    void z2() { k = 1 ; }
    void z3() { i = 6-i-j ; k-- ; }
};

void CHanoy::calculate( int from, int to, int N )
{
    i = from ; j = to ; k = N ;
    printf( "\nХаной с %d дисками:\n", k ) ;
    hanoy() ;
}

void CHanoy::hanoy()
{
    do
        switch ( y )
        {
            case 0:
                y = 1 ;
                break ;

            case 1:
                if( x0() ) { z0() ; y = 2 ; }
                else
                    { push(3) ; z1() ; }
                break;

            case 2:
                if( x1() ) y = 0 ;
                else
                    { pop() ; }
                break;

            case 3:
                push(4) ; z2() ; y = 1 ;
                break;

            case 4:

```

```

        push(2) ; z3() ;    y = 1 ;
        break;
    }
    while( y != 0 ) ;
}

void CHanoy::pop()
{
    y = stack[top].y ;
    i = stack[top].i ;
    j = stack[top].j ;
    k = stack[top].k ;
    top-- ;
}

void main()
{
    CHanoy hanoy ;
    hanoy.calculate( 1, 3, 3 ) ;
}

```

Из рассмотрения приведенной программы следует, что "оборачивание" автомата классом обеспечивает сокрытие переменных внутри этого класса. Область видимости этих переменных ограничена рамками класса, и поэтому не требуется передавать их значения в виде параметров функций, реализующих входные и выходные воздействия. В качестве дополнительного достоинства приведенной программы следует отметить компактность реализации операции `push` по сравнению с другими приведенными выше программами.

5.5.5. Обход дерева действий

Дерево декомпозиции (рис. 58), 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. 66).

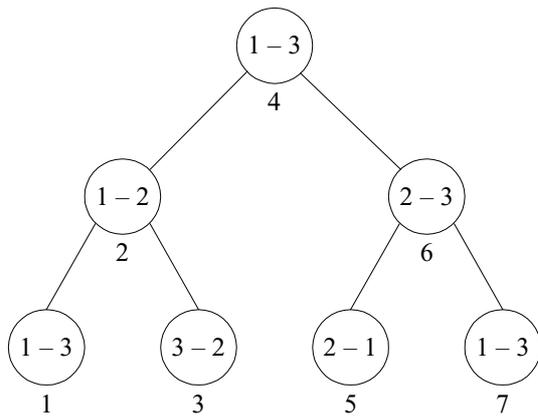


Рис. 66. Дерево действий при $N = 3$

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 переключений 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 дерево 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 поиска вершин работает сверху вниз.

Программа, реализующая этот алгоритм, приведена в листинге 24.

Листинг 24. 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 ) ;
}
```

Построим схему этой программы (рис. 67).

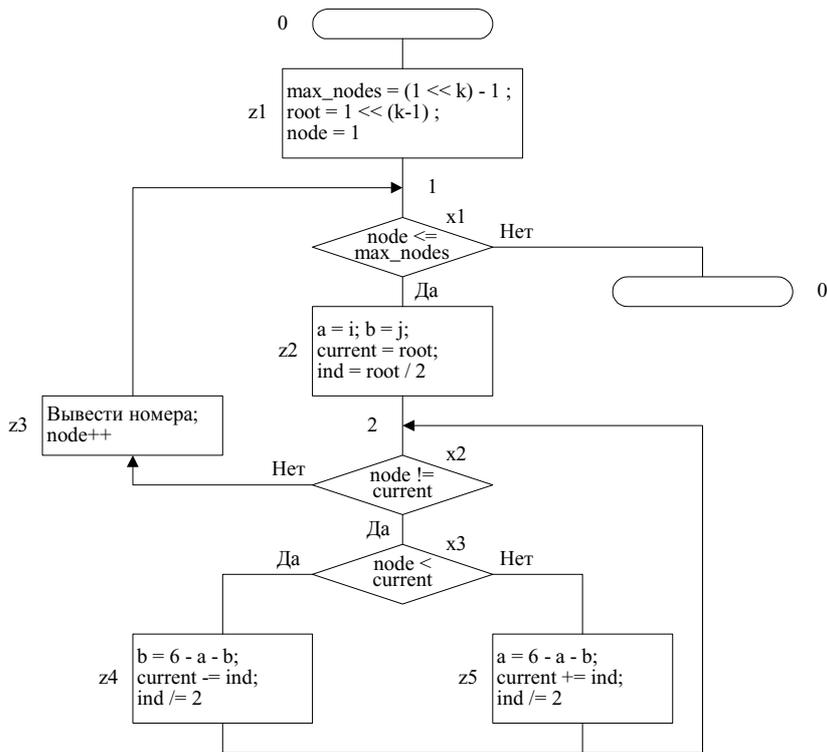


Рис. 67. Схема программы обхода дерева действий

В соответствии с методикой, изложенной в предыдущем разделе, построим по этой схеме граф переходов автомата Мили. При этом состояниям автомата Мили на схеме программы соответствуют точки, следующие за операторными вершинами. Нулевому состоянию соответствуют вершины схемы, обозначающие начало и конец программы. Построенный граф переходов приведен на рис. 68.

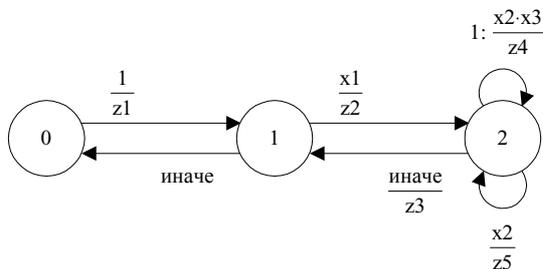


Рис. 68. Граф переходов автомата, реализующего обход дерева действий

Программа, реализующая этот автомат, приведена ниже.

Листинг 25. Автоматная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int j, int k )
```

```

{
  int y = 0 ;
  int max_nodes, root, node, a, b, current, ind ;

  do
    switch( y )
    {
      case 0:
        max_nodes = (1 << k) - 1 ;
        root = 1 << (k-1) ;
        node = 1 ;
        break ;
        y = 1 ;

      case 1:
        if( node <= max_nodes )
        {
          a = i ; b = j ;
          current = root ;
          ind = root / 2 ;
          break ;
          y = 2 ;

        }
        else
          break ;
          y = 0 ;

      case 2:
        if( node != current && node < current )
        {
          b = 6-a-b ;
          current -= ind ;
          ind /= 2 ;
        }
        else
          if( node != current )
          {
            a = 6-a-b ;
            current += ind ;
            ind /= 2 ;
          }
          else
          {
            printf( "Вершина %2d. %d -> %d\n", node, a, b ) ;
            node++ ;
            break ;
            y = 1 ;
          }
        }
    }
  while( y != 0 ) ;
}

void main()
{
  int input = 4 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoi( 1, 3, input ) ;
}

```

Авторы благодарны студенту СПбГИТМО (ТУ) Крылову Р.А., который в своей курсовой работе начал рассмотрение метода, излагаемого в настоящем разделе.

Отметим, что при использовании этого метода номер перекладываемого диска явно не указывается. Номера дисков могут быть определены с помощью подхода, изложенного в работе [85], который состоит в следующем: строится

таблица, строки которой содержат двоичное представление номера шага (табл. 4). При этом номер разряда двоичного числа, в котором размещена "младшая единица" (при условии, что счет разрядов начинается с единицы), является номером перекладываемого на данном шаге диска. Как отмечено в работе [86], последовательность номеров перекладываемых дисков является палиндромом.

Таблица 4. Определение номера перекладываемого диска (для $N = 3$)

| Номер шага | Номер разряда | | | Номер диска |
|------------|---------------|---|---|-------------|
| | 3 | 2 | 1 | |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 2 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 3 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | 2 |
| 7 | 1 | 1 | 1 | 1 |

Отметим, что аналитически номер перекладываемого диска может быть определен как единица плюс количество делений номера шага на два без остатка. При этом для нечетных номеров шагов количество делений на два без остатка равно нулю, и поэтому номер диска равен единице. Таким образом, на каждом нечетном шаге всегда перекладывается диск наименьшего диаметра.

Обобщая изложенное выше, построим дерево решения задачи (рис. 69), которое содержит исчерпывающую информацию для перекладывания дисков. В этом дереве для каждой вершины снизу указан номер шага, а внутри – номера диска и стержней, участвующих в перекладывании.

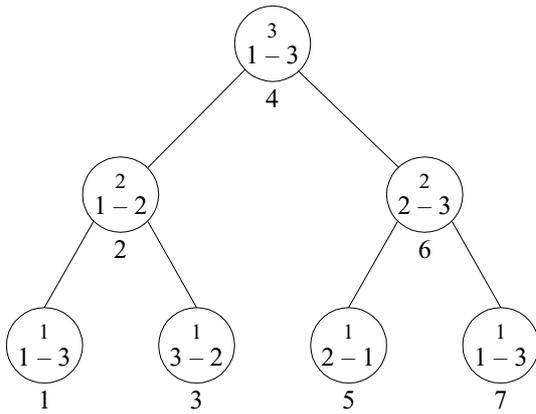


Рис. 69. Дерево решения задачи при $N = 3$

Из рассмотрения рис. 69 следует также, что номера перекладываемых дисков во всех вершинах одного уровня равны и совпадают с номером этого уровня.

5.5.6. Решение задачи в терминах "объекта управления"

Если алгоритм решения рассматриваемой задачи задавать непосредственно в терминах объекта управления (дисков и стержней), как это предлагается П. Бьюнманом и Л. Леви [87], то его удастся описать в виде графа переходов автомата с двумя состояниями (рис. 70).

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

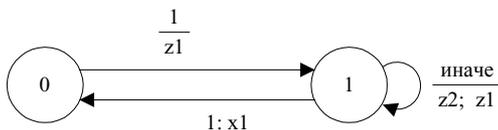


Рис. 70. Граф переходов при непосредственном перекладывании дисков

Текст программы, реализующей описываемый автомат, приведен в листинге 26. В этой программе функции $z1$, $z2$ и $x1$ могут быть реализованы по-разному. Например, определение номеров перекладываемого диска и участвующих

в перекладывании стержней может выполняться либо с помощью перебора (как это имеет место ниже), либо аналитически, например, как это предложено в работе [88].

Особенность рассматриваемой программы состоит в том, что несмотря на простой управляющий автомат, 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 стержня на третий, 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.

Листинг 26. Автоматная программа, реализующая непосредственное перекладывание дисков

```
#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 ;
}
```

6. Выводы по II части

Предложенная на предыдущем этапе НИР технология автоматного программирования расширена применительно еще к одному классу задач – вычислительным алгоритмам.

Сопоставление рассмотренных в настоящей работе традиционных схем алгоритмов и графов переходов показывает, что последние более компактны. Это связано с тем, что построение графа переходов сводится к исследованию всех путей между «соседними» пометками в традиционной схеме алгоритма и компактному их представлению в виде помеченных дуг и петель графа.

Новизна автоматного подхода по сравнению с традиционным обеспечивается за счет добавления к понятиям "входное и выходное воздействия" понятия "**состояние**", а по сравнению с методом Ашкрофта-Манни – за счет сокращения числа состояний.

Предложенное для вычислительных алгоритмов явное выделение состояний позволяет рассматривать такие алгоритмы не в терминах условных переходов (и, соответственно, флагов), а в терминах состояний.

Основное преимущество такой реализации алгоритмов состоит в резком упрощении их визуализации, что особенно важно для обучения, особенно дистанционного [79].

В четвертой главе предложен метод преобразования произвольной итеративной программы в автоматную. Это позволяет реализовать произвольный итеративный алгоритм программой, содержащей всего лишь один цикл `do-while`, телом которого является один оператор `switch`.

В пятой главе предложен метод преобразования программ с явной рекурсией в автоматные программы. Метод состоит из следующих этапов.

1. Программа с явной рекурсией преобразуется в итеративную программу, построенную с применением автомата Мили. При этом работа рекурсивной программы моделируется итеративной автоматной программой, в которой явно используется стек.

2. 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.

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

4. Для задачи о ханойских башнях кроме предлагаемого метода преобразования рассматриваются и другие подходы к преобразованию рекурсивных алгоритмов в автоматные: обход дерева действий, выполняемых рекурсивной программой, и решение задачи в терминах "объекта управления".

5. На примере задачи о ханойских башнях рассматривается подход, в котором при реализации автоматной программы применяются классы. Это позволяет ограничить область видимости используемых переменных пределами класса и эффективно реализовать сам автомат и его входные и выходные воздействия в виде методов.

6. Показано, что метод, предложенный для построения автоматных программ по итеративным, применим также и к рекурсивным программам.

7. В работе [90] отмечено, что итеративные алгоритмы по вычислительной мощности эквивалентны рекурсивным. Однако, в известной авторам литературе, формальный метод преобразования рекурсивных алгоритмов в итеративные отсутствует. Как отмечалось выше, в работе [81] приведены

примеры такого преобразования (для конечных рекурсий), выполненного эвристически. Материалы, изложенные в настоящей работе, устраняют указанный пробел.

Можно считать, что предложенные в работе методы преобразования итеративных и рекурсивных программ в автоматные обеспечивают преобразование процедурных знаний в декларативные.

7. Публикации по результатам этапа

По результатам выполненных в ходе этапа работ опубликованы следующие статьи:

1. Туккель Н.И., Шалыто А.А. Танки и автоматы // ВУТЕ/Россия. 2003. №2, с. 69-73.
2. Шалыто А.А. Новая инициатива в программировании. Движение за открытую проектную документацию // Мир ПК-ДИСК. 2003. №8; Мир ПК. 2003. №9, с. 52-56; PC Week. 2003. №40, с. 38-39, 42.
3. Шалыто А.А. Технология автоматного программирования // Мир ПК. 2003. №10, с. 74-78.
4. Туккель Н.И., Шалыто А.А. Автоматное и синхронное программирование // Искусственный интеллект. 2003. №4, с. 82-88.
5. Шалыто А.А. Технология автоматного программирования // Современные технологии. СПбГУИТМО, с. 18-26.
6. Шалыто А.А. Новая инициатива в программировании. Движение за открытую проектную документацию // Информатика. 2003. №44, с. 22-24, 31.
7. Анатолий Шалыто, Никита Туккель, Никита Шамгунов. Задача о ходе коня // Мир ПК. 2003. №1. с. 152-155.

8. Н.И. Туккель, А.А. Шалыто, Н.Н. Шамгунов. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. 2002. №5. с. 72-99.
9. А.А. Шалыто, Н.И. Туккель. Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. №5. с. 12-26.
10. Анатолий Шалыто, Никита Туккель, Никита Шамгунов. Ханойские башни и автоматы // Программист. 2002. №8. с. 82-90.
11. Анатолий Шалыто, Никита Туккель. От тьюрингова программирования к автоматному // Мир ПК. 2002. №2. с. 144-149.

По результатам выполненных в ходе этапа работ в материалах конференций опубликованы:

1. Гуров В.С., Нарвский А.С., Шалыто А.А. Автоматизация проектирования событийных объектно-ориентированных программ с явным выделением состояний // Труды X Всероссийской научно-методической конференции "Телематика-2003". 2003. Т.1, с. 282-283
2. Шопьрин Д.Г., Шалыто А.А. Применение класса "STATE" в объектно-ориентированном программировании с явным выделением состояний // Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. Т.1, с.284-285.
3. Корнеев Г.А., Шалыто А.А. Реализация конечных автоматов с использованием объектно-ориентированного программирования // Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. Т.2, с.377-378.

4. Корнеев Г.А., Казаков М.А., Шалыто А.А. Построение логики работы визуализаторов алгоритмов на основе автоматного подхода // Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. Т.2, с.378-379.
5. Мазин М.А., Парфёнов В.Г., Шалыто А.А. Автоматная реализация интерактивных сценариев образовательной анимации // Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. Т.2, с.379-380.
6. Шалыто А.А. Технология автоматного программирования // Труды Первой Всероссийской научной конференции "Методы и средства обработки информации". М.: МГУ. 2003, с.528-535.
7. Туккель Н.И., Шалыто А.А. Автоматное и синхронное программирование // Материалы международной научно-технической конференции "ИМС-2003" "Интеллектуальные и многопроцессорные системы - 2003". Таганрог-Донецк, ТРТУ. 2003.т.2, с. 15-17.
8. Штучкин А.А., Шалыто А.А. Совместное использование теории построения компиляторов и Switch-технологии // Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. Т.1, с.286-287.
9. Бабаев А.А., Чижова Г.А., Шалыто А.А. Метод создания скелетной анимации на основе автоматного программирования // Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. Т.2, с.375-376.

10. Shalyto A.A., Naumov L.A. Automata Programming as a Sort of Synchronous Programming // Proceedings of "East-West Design & Test Conference. (EWDTC-03). Yalta: Kharkov National University of Radio-electronics, 2003, p.140-143.
11. Shalyto A.A., Naumov L.A. Automata Theory for Multi-Agent Systems Implementation // Proceedings of International Conference Integration of Knowledge Intensive Multi-Systems: Modeling, Exploration and Engineering. KIMAS-03. IEEE Boston. 2003, p.65-70.

Список литературы

1. Страуструп Б. Язык программирования Си++. М.: Бином, СПб.: Невский диалект, 1999.
2. Эккель Б. Философия Java. СПб.: Питер, 2001.
3. Буч Г. Объектно-ориентированное проектирование с примерами применения. Киев: Диалектика, М.: ИВК, 1992.
4. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, СПб.: Невский диалект, 1998.
5. Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. Киев: Диалектика, 1993.
6. Терехов А.Н., Романовский К.Ю., Кознов Д.В. и др. REAL: Методология и CASE-средство разработки информационных систем и программного обеспечения систем реального времени //Программирование. 1999. № 5.
7. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. М.: ДМК, 2000.
8. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. СПб.: Питер, 2001.
9. Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний //Мир ПК. № 8,9.
10. Шалыто А.А., Туккель Н.И. SWITCH-технология – автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. № 5.
11. Шалыто А.А., Туккель Н.И. От тьюрингового программирования к автоматному программированию //Мир ПК. 2001. № 12.

12. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
13. Harel D., Politi M. Modeling Reactive Systems with Statecharts. NY: McGraw-Hill, 1998.
14. Martin R. Designing Object-Oriented C++ Applications Using the Booch Method. NJ: Prentice Hall, 1993.
15. Любченко В.С. О бильярде с Microsoft Visual C++ 5.0 //Мир ПК. 1998. № 1.
16. Шлепнев А. Системно-ориентированное программирование //Мир ПК. 2001. № 6.
17. Шалыто А.А. Программная реализация управляющих автоматов //Судостроит. пром-сть. Сер. Автоматика и телемеханика. 1991. Вып. 13.
18. Гамма Э., Хелм Р., Джонсон Р. и др. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.
19. Ваганов С.А., Туккель Н.И., Шалыто А.А. Повышение централизации управления при программировании "реактивных" систем //Труды Международной научно-методической конференции "Телематика' 2001". СПб.: СПбГИТМО (ТУ), 2001.
20. Шалыто А.А. Алгоритмизация и программирование для систем логического управления и "реактивных" систем (обзор) //Автоматика и телемеханика. 2001. № 1.
21. Фридман А.Л. Основы объектно-ориентированной разработки программных систем. М.: Финансы и статистика, 2000.

22. Cardelli L., Wegner P. On Understanding Types, Data Abstraction and Polymorphism //ACM Computing Surveys. 1985. Vol.17 (4). December.
23. Jacobson I. et al. Object-Oriented Software Engineering. NJ: Addison Wesley, 1992.
24. Бобровский С. Из пророков – в коммерсанты //PC WEEK/RE. 4.02.1997.
25. Лавров С.С. Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001.
26. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации алгоритмов. СПб.: Наука, 2000.
27. Романовский И.В. Дискретный анализ. СПб.: Невский диалект, 2000.
28. Фауллер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. М.: Мир, 1999.
29. <http://robocode.alphaworks.ibm.com>.
30. <http://www.softcraft.ru>.
31. <http://robocode.isbeautiful.org>.
32. UML Semantics. Version 1.0. Rational Software Corp., 1997.
33. Odell J. Advanced Object-Oriented Analysis & Design Using UML. NY: SIGS Books, 1998.
34. Booch G., Rumbaugh J., Jacobson I. The Unified Software Development Process. NJ: Addison-Wesley, 1999.
35. Петерсон Т. Там, где сходятся люди и машины //Computerworld Россия. 21.08.2001.

36. Бобровский С. Самоучитель программирования на языке C++ в системе Borland C++ Builder 5.0. М.: ДЕСС КОМ, 2001.
37. Соловьев И.П. Формальные спецификации вычислительных систем. Машины абстрактных состояний (Машины Гуревича). СПб.: СПбГУ, 1998.
38. Gyrevich Y. et al. Using Abstract State Machines at Microsoft: A Case Study /Proceeding of ASM'2000 in "Abstract State Machines: Theory and Applications". Lecture Notes in Computer Science. 2000. V.1912.
39. Budd T. Multiparadigm Programming in Leda. NJ: Addison-Wesley, 1995.
40. Кнут Д. Искусство программирования. Т. 1: Основные алгоритмы. М.: Вильямс, 2001.
41. Корн Г., Корн Т. Справочник по математике для научных работников и инженеров. М.: Наука, 1978.
42. Дейкстра Э. Дисциплина программирования. М.: Мир, 1979.
43. Грис Д. Наука программирования. М.: Мир, 1984.
44. Буч Г. Объектно-ориентированный анализ и проектирование. М.: Бином, СПб.: Невский диалект, 1998.
45. Страуструп Б. Язык программирования C++. М.: Бином, СПб.: Невский диалект, 2001.
46. Шалыто А.А. Новая инициатива в программировании. Движение за открытую проектную документацию // Мир ПК. 2003. № 9. <http://is.ifmo.ru>, раздел «Статьи».
47. Тэллес М., Хсих Ю. Наука отладки. М.: Кудиц-образ, 2003.
48. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

49. Любченко В.С. Конечно-автоматная технология программирования // Труды международной научно-методической конференции «Телематика'2001». СПб.: СПбГИТМО (ТУ), 2001.
50. Сацкий С. Дизайн шаблона конечного автомата на C++ // RSDN Magazine. 2003. № 1.
51. Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний // Мир ПК. 2001. № 8. № 9. <http://is.ifmo.ru>, раздел «Статьи».
52. Шалыто А.А., Туккель Н.И. Танки и автоматы // ВУТЕ/Россия. 2003. № 2. <http://is.ifmo.ru>, раздел «Статьи».
53. Шалыто А.А. Технология автоматного программирования // Мир ПК. 2003. № 10. <http://is.ifmo.ru>, раздел «Статьи».
54. Брауэр В. Введение в теорию конечных автоматов. М.: Радио и связь, 1987.
55. Наумов А.С., Шалыто А.А. Система управления лифтом. Проектная документация. <http://is.ifmo.ru>, раздел «Проекты».
56. Дейтел Х.М., Дейтел П.Дж. Как программировать на C++. Третье издание. М.: Бином, 2003.
57. Крачтен Ф. Введение в Rational Unified Process. М.: Вильямс, 2002.
58. Шопырин Д.Г., Шалыто А.А. Объектно-ориентированный подход к автоматному программированию. СПб.: СПбГУ ИТМО, 2003. <http://is.ifmo.ru>, раздел «Проекты».
59. Еремин Е.А. MMIX – учебный RISC-процессор нового тысячелетия от Дональда Кнута // Информатика. 2002. № 40.
60. Наумов А.С., Шалыто А.А. Система управления лифтом. СПб.: СПбГУ ИТМО, 2003. <http://is.ifmo.ru>, раздел «Проекты».

61. Корнеев Г.А., Шалыто А.А. Реализация конечных автоматов с использованием объектно-ориентированного программирования // Труды X Всероссийской научно-методической конференции "Телематика-2003". 2003. Т.2.
62. Фельдман П.И., Шалыто А.А. Совместное использование объектного и автоматного подходов в программировании. СПб.: СПбГУ ИТМО, 2004. <http://is.ifmo.ru>, раздел «Проекты».
63. Заякин Е.А., Шалыто А.А. Метод устранения повторных фрагментов кода при реализации конечных автоматов. СПб.: СПбГУ ИТМО, 2003. <http://is.ifmo.ru>, раздел «Проекты».
64. Астафуров А.А., Шалыто А.А. Разработка и применение паттерна «Automata». СПб.: СПбГУ ИТМО. 2003. <http://is.ifmo.ru>, раздел «Проекты».
65. Кузнецов Д.В., Шалыто А.А. Система управления танком для игры «Robocode». Вариант 2. СПб.: СПбГУ ИТМО, 2003.
66. Гуров В.С., Нарвский А.С., Шалыто А.А. Автоматизация проектирования событийных объектно-ориентированных программ с явным выделением состояний // Труды X Всероссийской научно-методической конференции "Телематика-2003". 2003. Т.1.
67. Гуров В.С., Шалыто А.А. XML и автоматы. СПб.: СПбГУ ИТМО. 2004. <http://is.ifmo.ru>, раздел «Проекты».
68. Бондаренко К.А., Шалыто А.А. Разработка XML - формата для описания внешнего вида видеопроигрывателя с использованием конечных автоматов. СПб.: СПбГУ ИТМО. 2003. <http://is.ifmo.ru>, раздел «Проекты».
69. Гуисов М.И., Кузнецов А.Б., Шалыто А.А. Интеграция механизма обмена сообщениями в Switch-технологиию.

- СПб.: СПбГУ ИТМО. 2003. <http://is.ifmo.ru>, раздел «Проекты».
70. Гуисов М.И., Шалыто А.А. Задача Д. Майхилла «Синхронизация цепи стрелков». Вариант 1. СПб.: СПбГУ ИТМО. 2003. <http://is.ifmo.ru>, раздел «Проекты».
71. Гуисов М.И., Кузнецов А.Б., Шалыто А.А. Задача Д. Майхилла «Синхронизация цепи стрелков». Вариант 2. СПб.: СПбГУ ИТМО. 2003. <http://is.ifmo.ru>, раздел «Проекты».
72. Альшевский Ю.А., Раер М.Г., Шалыто А.А. Система управления турникетом. СПб.: СПбГУ ИТМО. 2003. <http://is.ifmo.ru>, раздел «Проекты».
73. Кузнецов Б.П. Психология автоматного программирования // ВУТЕ/Россия. 2000. №11.
74. Казаков М.А., Столяр С.Е. Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования // Телематика 2000. Тез. докл. Международной научно-метод. конф. СПб.: СПбГИТМО (ТУ), 2000.
75. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ. М.: МЦНМО, 1999.
76. Aschcroft E., Manna Z. The translation of "goto" programm into "while" programm // Proceeding of 1971 IFIP Congress.
77. Йодан Э. Структурное проектирование и конструирование программ. М.: Мир, 1979.
78. Баранов С.И. Синтез микропрограммных автоматов (граф-схемы и автоматы). Л.: Энергия, 1979.
79. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. М.: Мир, 1982.
80. Казаков М.А., Шалыто А.А., Туккель Н.И. Использование автоматного подхода для реализации вычислительных алгоритмов // Труды международной

научно-методической конференции "Телематика'2001".
СПб.: СПбГИТМО, 2001.

81. Фон Нейман Дж. Общая и логическая теория автоматов
// В кн. Тьюринг А. Может ли машина мыслить? Саратов:
Колледж, 1999.
82. Стивенс Р. Delphi. Готовые алгоритмы. М.: ДМК,
2001.
83. Седжвик Р. Фундаментальные алгоритмы на С++. Киев:
ДиаСофт, 2001.
84. Бобак И. Алгоритмы: "возврат назад" и "разделяй и
властвуй" //Программист. 2002. №3.
85. Грэхем Р., Кнут Д., Поташник О. Конкретная
математика. М.: Мир, 1998.
86. Гарднер М. Математические головоломки и
развлечения. М.: Мир, 1971.
87. Романовский И.В., Столяр С.Е. Стек и его
использование. <http://ips.ifmo.ru>.
88. Анисимов А.В. Информатика. Творчество. Рекурсия.
Киев: Наукова думка, 1988.
89. Быстрицкий В.Д. Ханойские башни.
<http://alglib.chat.ru/paper/hanoy.html>
90. Брукшир Дж. Введение в компьютерные науки. М.:
Вильямс, 2001.