МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ МЕХАНИКИ И ОПТИКИ

Факультет **Информационных технологий и программирования**

Направление **Прикладная математика и информатика** Специализация: **2**

Академическая степень **магистр математики**

Кафедра **Компьютерных технологий** Группа **6839**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Метод построения логики визуализаторов алгоритмов

Автор магистерской диссертации Корнеев Г.А. (подпись)
(Фамилия, И., О.)

Научный руководитель Шалыто А.А. (подпись)
(Фамилия, И., О.)

Руководитель магистерской программы Парфенов В.Г. (подпись)
(Фамилия, И., О.)

К з а щ и т е д о п у с т и т ь
Зав. кафедрой Васильев В.Н. (подпись)
(Фамилия, И., О.)

“____”_____2004 г.

Санкт-Петербург, 2004 г.

Магистерская диссертация выполнена с оценкой _____

Дата защиты “____”_____20____г.

Секретарь ГАК _____

Листов хранения _____

Чертежей хранения _____

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

При изучении алгоритмов обработки информации, представляемой различными структурами данных [10, 11], важную роль играют визуализаторы алгоритмов, позволяющие в наглядной форме динамически отображать детали их работы. Это открывает возможность использования новой технологии при изучении дискретной математики и программирования [8, 55].

Визуализатор — это программа, в процессе работы которой на экране компьютера динамически демонстрируется применение алгоритма к выбранному набору данных. Визуализаторы позволяют изучать работу алгоритмов в пошаговом режиме, аналогичном режиму трассировки программ. Они при необходимости допускают трассировку укрупненными шагами, игнорируя рутинную часть вычислительного процесса, что существенно, например, для переборных алгоритмов.

Для некоторых алгоритмов динамический вариант демонстрации его работы является более естественным, чем набор статических иллюстраций. Для родственных алгоритмов (например, алгоритмов сортировки) визуализация позволяет наглядно продемонстрировать как общий подход, так и различие в механизмах их действия.

При изучении большинства алгоритмов наряду с режимом "шаг вперед" весьма полезен также и режим "шаг назад" [15, 29, 8], позволяющий более быстро и полно понять алгоритм. Например, в алгоритмах поиска с возвратом бывает необходимо сделать несколько шагов назад, для того чтобы понять, почему та или иная ветвь поиска отброшена.

Многолетний опыт построения и применения визуализаторов на кафедре “Компьютерные технологии” СПбГИТМО (ТУ) показал, что они могут быть использованы как основной инструмент преподавания указанных выше курсов [55], в частности, при дистанционном обучении (http://ips.ifmo.ru) [6].

Написание визуализатора “с нуля” является очень трудоемкой задачей. По этому, большинство визуализаторов разрабатываются в рамках системы

визуализации. Обычно система визуализации предоставляет разработчику графический инструментарий, позволяющий значительно быстрее построить визуализатор. Некоторые системы визуализации так же содержат средства помогающие выделить “интересные точки” визуализируемого алгоритма либо визуализировать изменения в структурах данных.

Несмотря на то, что визуализаторы разрабатывались еще с 1980-х годов, можно утверждать, что к настоящему времени основные достижения в проектировании визуализаторов относятся к сфере педагогики [8], а достижения в сфере технологии создания визуализаторов практически отсутствуют. В частности, отсутствует метод, позволяющий по алгоритму формально и единообразно создавать логику работы визуализатора. В работе [12] был предложен метод преобразования итеративных алгоритмов в автоматные. При этом было высказано предположение, что трудности с формальным построением визуализаторов связаны с тем, что в традиционном процедурном программировании понятие "состояние" явно не применяется, и поэтому "что" и "как" визуализировать каждый программист каждый раз решает заново и эвристически.

В автоматном программировании [13] состояния используются явно, и поэтому такое программирование было названо "программирование с явным выделением состояний" (http://is.ifmo.ru). В работе [55] было предложено использовать особенности автоматных программ для построения визуализаторов.

В настоящей работе предлагается метод построения логики работы визуализатора по заданному алгоритму и система визуализации построенная на основе этого метода. Метод позволяет представить логику работы визуализатора системой взаимосвязанных конечных автоматов. Система состоит из пар автоматов, каждая из которых состоит из “прямого” и “обратного” автоматов, которые обеспечивают пошаговое движение по алгоритму вперед и назад соответственно.

Некоторые шаги предлагаемого метода являются неформальными, например, реализация визуализируемого алгоритма на языке программирования.

Для описания логики работы визуализаторов выбраны автоматы Мура, в которых действия выполняются только в состояниях. Это позволяет отображать операции, выполняемые алгоритмом, наиболее естественным образом. При построении обратного автомата возможен переход к смешанному автомату, на ребрах которого выполняются только вспомогательные действия, но не шаги алгоритма.

Актуальность исследования. Визуализаторы алгоритмов

Глава 1 содержит описание текущего состояния в области визуализаторов и систем визуализации. В частности, рассмотрено применение визуализаторов в учебном процессе и возникающие при этом требования к визуализаторам.

В этой главе так же приводится анализ визуализаторов алгоритмов и систем визуализации с точки зрения выдвинутых требований и показывается, что они им не удовлетворяют. Таким образом, обосновывается необходимость разработки новой системы визуализации.

В Главе 2 приводится общее описание процесса построения визуализатора, рассматриваемое с различных точек зрения:

- разработка отдельных частей визуализатора;
- порядок разработки визуализатора;
- документирования процесса разработки.

В этой главе так же указываются места остальных глав работы.

В *Главе 3* рассматривается выделение модели данных из визуализируемой программы и упрощение ее структуры с целью дальнейшего преобразования в систему взаимодействующих автоматов.

Глава 4 содержит описание процесса преобразования программ в систему взаимодействующих автоматов. Рассматриваются как формальные и

неформальные методы преобразования, при этом для формального метода приведено доказательство корректности.

В *Главе 5* описывается XML-формат описания визуализаторов. Отдельно рассматриваются основное описание, описание алгоритма и описание конфигурации визуализатора.

В *Главе 6* описывается система визуализации *Vizi*, построенная на основе подхода разработанного в предыдущих главах. Здесь же приводится информация о практическом внедрении *Vizi*.

ГЛАВА 1. ВИЗУАЛИЗАТОРЫ АЛГОРИТМОВ

В разделе 1.1 рассмотрено применение визуализаторов в учебном процессе, в частности в разделе 1.1.2 сформулированы требования к визуализаторам применяемых при обучении. Далее в разделе 1.2 рассмотрены текущее состояние общедоступных визуализаторов на примере визуализаторов сортировок. В разделе 1.3 рассмотрены имеющиеся системы визуализаторов и указаны их недостатки.

1.1. Применение визуализаторов в учебном процессе

Визуализаторы алгоритмом широко применяются для обучения дискретной математике и информатике [6, 7, 8, 15, 55]. Исследования [19] показали, что применение визуализаторов помогает учащимся глубже и быстрее понять объясняемый материал. При этом так же выявлена большая заинтересованность учащихся в лекциях с применением визуализаторов, чем без таковых.

1.1.1. Варианты применения визуализатора

При обучении визуализаторы могут быть использованы несколькими различными способами. Опишем некоторые из них.

В начале визуализаторы алгоритмов использовались как сопроводительный материал на лекциях (например [14]). При этом, преподаватель заранее готовит визуализационный материал и показывает его на лекциях, поясняя работу алгоритма.

Визуализатор так же могут быть использованы для большего вовлечения учащихся в процесс обучения. Одним из таких способов является отображение некоторого состояния алгоритма и предложение учащимся определить, како действие выполнит алгоритм. После получения ответов, они проверяются путем совершения шага вперед. Визуализатор так же позволяет подробно разобрать почему было осуществлено выбрано именно это действие а не

другое. В таком режиме визуализаторы так же могут быть использованы для проверки знаний.

Другой способ использования визуализаторов — начальное знакомство с алгоритмом. Учащийся сначала работает с визуализатором, составляя для себя общую схему алгоритма. В последствии, преподаватель дает полное описание алгоритма, после чего учащийся может вернуться к работе с визуализатором.

Важной областью использования визуализаторов является дистанционное и самостоятельное обучение. В этом случае большое значение приобретает возможность задания пользователем входного набора данных. Таки образом, учащийся может не спрашивать преподавателя, что будет при обработке некоторого набора данных, а ввести его в визуализатор и посмотреть самостоятельно.

1.1.2. Требования к визуализаторам алгоритмов

Для использования в учебном процессе визуализаторы алгоритмов должны удовлетворять следующим требованиям:

1. *Управляемость* — у учащихся должны быть возможность задавать собственные наборы входных данных и рассматривать работу алгоритма на них.
2. *Интерактивность* — при работе с визуализаторам должна быть возможность совершать шаги как вперед, так и назад, так же должен быть предусмотрен режим автоматической визуализации.
3. *История* — визуализатор должен позволять сделать сколь угодно много шагов назад, отображая состояния алгоритма на соответствующий момент.
4. *Возможность отображения хода выполнения алгоритма*.
5. *Возможность комментирования выполнения программы*.
6. *Простота использования* — визуализатор должен быть понятен неподготовленному пользователю.

7. *Доступность* — учащиеся должна быть возможность доступа к визуализаторам не только на занятиях.
8. *Платформонезависимость* — визуализатор должен работать в независимости как от платформы (IBM-PC совместимые компьютеры, Apple Macintosh) так и операционной системы (Windows, Unix, Linux).
9. *Независимость от сети* — для работы визуализатора не должен требоваться доступ к сети.
10. *Удобство создания визуализаторов алгоритмов* — простота использования системы визуализаторов с точки зрения написания новых визуализаторов.

Управляемость, интерактивность и история позволяет более глубоко изучить работу алгоритма, чем визуализация на заранее выбранных наборах данных [19].

Простота использования важна при самостоятельном обучении, так как в этом случае у пользователи очень редко читают инструкции по применению.

Возможности отображения хода алгоритма и комментирования хода программы являются весьма важными. Первая — для пояснения простых алгоритмов и введения в информатику, а вторая — для понимания сложных алгоритмов.

Доступность, платформонезависимость и независимость от сети позволяет использовать визуализаторы как на лекциях и практических занятиях, так для самостоятельного обучения.

1.2. Обзор визуализаторов алгоритмов сортировок

Произведем обзор визуализаторов алгоритмов, на примере алгоритмов сортировки данных, так как это одна из основных тем дискретной математики.

1.2.1. Подходы к визуализации алгоритмов сортировок

Существует несколько подходов к визуализации алгоритмов сортировок:

1. “числовой”;

2. “палочный”,

3. “исторический”.

На рис. 1 изображены примеры изображений, генерируемых визуализаторами использующими эти подходы.

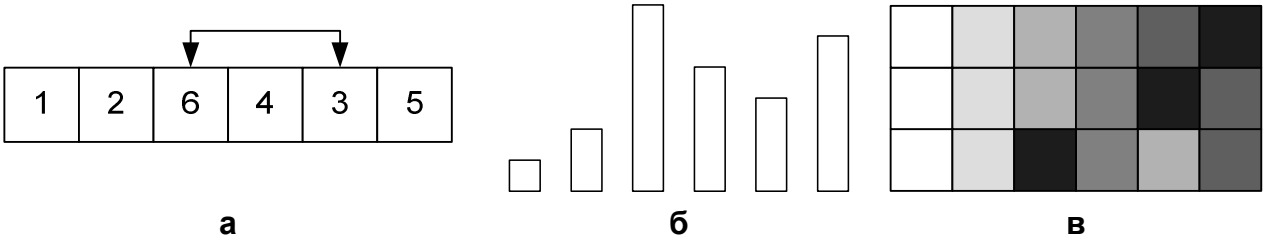


Рис. 1. Подходы к визуализации алгоритмов сортировки “числовой” (а), “палочный” (б) и “исторический” (в)

При “числовом” подходе сортируемые массив отображается набором содержащихся в нем чисел. При сравнении и обмене элементов они подсвечиваются. В некоторых случаях применяется анимация обмена элементов.

“Палочный” подход характеризуется представлением сортируемого массива в виде набора “палочек” различной длины. При этом обмен элементов визуализируется как физическое перемещение “палочек” [20].

При “историческом” подходе процесс сортировки отображается в виде одного изображения. При этом, обычно, значения элементов отображают цветом [15].

Обычно визуализируется следующим “стандартный” набор алгоритмов сортировки: обменная, пузырьковая, простого выбора, простой вставки, пирамидальная, слиянием и быстрая сортировка.

К сожалению, в открытом доступе не удалось найти российские визуализаторы сортировок, не выполненные на основе системы визуализации *Vizi* (*BaseApplet*).

1.2.2. Обзор визуализаторов алгоритмов сортировок

Brown University (BALSA)

BALSA является одним из первых опытов создания визуализаторов алгоритмов. Принципы ее построения *BALSA* описаны в [15] (так же см. раздел 1.2). При визуализации можно открыть несколько “видов” состояния структур данных. На пример, “палочное” или “числовое” представление. Так же имеются “исторические” виды, представляющие процесс сортировки в целом.

Так как визуализируются наборы данных, отображение алгоритма и комментариев отсутствуют (процесс комментируется преподавателем). Визуализация осуществляется посредством заранее написанных программ на языке сценариев. Возможность ручного ввода данных не предусмотрена.

Существуют возможности комментировать исполнение визуализируемой программы и отображать ход ее выполнения, но они практически не используется.

Визуализаторы выполнены в виде модулей системы визуализации Balsa и не являются платформонезависимымис, что иак же ограничивает их доступность.

State University of New York College at Brockport

Коллекция визуализаторов сортировок расположена по адресу [42]. Представлен “стандартный” набор визуализаторов сортировок, созданных на общем ядре, при этом для алгоритма пирамидальной сортировки использовано специальное решение.

При визуализации используется “палочный” подход. Визуализация производится на предопределенном наборе данных, без возможности изменения. Существует возможность осуществлять шаги назад по алгоритму, но она не всегда работает.

Алгоритмы представлены только на сопроводительных страницах. Присутствуют комментарии к текущим действиям.

Визуализаторы выполнены в виде *Java*-апплетов, что обуславливает их досутепность и платформонезависимость. Для визуализации свызи с сервером не требуется, таким образом визуализаторы так же являются и независимыми о сети.

Princeton University

Визуализаторы исполнены в виде единого *Java*-апплета, доступного по адресу [43]. В дополнение к “стандартному” набору визуализаторов сортировок визуализируются алгоритмы построения выпуклых оболочек.

Для визуализации применяется как “числовой” так и “палочный” подход с возможностью выбора. Присутствует только возможность случайного задания набора данных для работы.

При визуализации алгоритмы не отображаются, так как использована визуализация данных (см. раздел 1.3.1). По той же причине, осуществляемые действия не комментируются. Третьим недостатком является невозможность осуществлять шаги назад по алгоритму.

Hope College

Визуализатор сортировок в единой оболочке доступен по адресу [44]. Для визуализации используется “палочный” подход. Ввод исходных данных не возможен, так же отсутсвует возможноть передвижения по алгоритму назад.

При визуализации отображается код программы и подсвечивается текущий оператор, но производимые действия не комментируются.

Визуализатор выполнен в виде *Java*-апплета, и является платформонезависимым.

Jeliot

Рассмотрим визуализаторы пузырьковой и быстрой сортировок, являющихся примерами к системе визуализации *Jeliot*.

При выполнении алгоритма используется низкоуровневая визуализация, отображающая не только выполнения опреаторов, но и вычисление выражений. При этом подсвечиватеся текущий оператор.

Ввод и исходных данных осуществляется через специальные классы ввода-вывода. Передвижение по программе в обратную сторону невозможно.

Для просмотра визуализаторов построенных на базе *Jeliot* требуется присутсвие самой системы визуализации, выполненной в виде *Java*-приложения, поставляемого для различных платформ. Таким образом система не являетяс в полной мере платформонезацисимой.

1.2.3. Выводы

Рассмотрим описанные визуализаторы с точки зрения требований изложенных в разделе 1.1.2. Удобство создания визуализаторов не оценивается, как неприменимое.

Столбцы табл. 1 соответствуют свойствам, указанным в разделе 1.1.2. При этом используются следующие обозначения:

- “+” — свойство выполняется,
- “–” — свойство не выполняется,
- “±” — свойство выполняется частично.

Табл. 1. Сравнительные характеристики визуализаторов алгоритмов сортировок

Визуализатор	1	2	3	4	5	6	7	8	9
Brown University	–	+	+	±	±	+	–	–	–
SUNY Brockport	–	±	±	–	±	–	+	+	+
Princeton University	±	–	–	–	±	±	+	+	+
Hope College	–	–	–	+	–	+	+	+	+
Jeliot	+	–	–	+	–	+	+	±	+

Таким образом, не один из рассмотренных визуализаторов не удовлетворяет все выдвинутым свойствам. Из этого следует, что необходимо

разработать новую систему визуализации, позволяющую строить визуализаторы для которых все эти свойства выполняются.

1.3. Системы визуализации

Система визуализации представляет собой программу или набор библиотек, позволяющие создавать и исполнять визуализаторы.

1.3.1. Классификация систем визуализации

С точки зрения визуализации алгоритмов системы визуализации можно разделить на три класса:

- *общие системы визуализации*,
- *системы визуализации алгоритмов*,
- *прочие системы визуализации*.

Общие системы визуализации могут применятся для визуализации любых объектов и данных.

Системы визуализации алгоритмов предназначены для визуализации программ и алгоритмов. Для этого в них предусмотрены специальные средства. В свою очередь, системы визуализации, алгоритмов можно классифицировать по типу получаемых с их помощью визуализаторов:

- *визуализаторы программ*,
- *визуализаторы данных*,
- *смешанные визуализаторы*.

Визуализаторы программ отображают ход выполнения визуализируемого алгоритма и действия, выполняемые при этом. Обычно они построены на концепции интересных состояний. При подготовке визуализатора в интересные точки программы вставляются вызовы процедур визуализации (в том числе, установка комментариев, если это предусмотрено). Таким образом, это метод визуализации является императивным.

Визуализаторы программ отображают изменения в структурах данных, происходящие при выполнении визуализируемого алгоритма. При таком

подходе визуализация действий, не связанных с изменением данных (например, сравнения в алгоритме сортировок, [20]) напрямую не поддерживаются и осуществляются искусственными методами. Вторым недостатком данного подхода является невозможность отображения комментариев к осуществляемым действиям.

Смешанные визуализаторы алгоритмов объединяют положительные стороны визуализаторов программ и данных. При этом одновременно может отображаться как выполнение программы, так и изменения соответствующих структур данных.

Хороший сравнительный обзор визуализаторов программ и данных дан в работе [20].

Прочие системы визуализации предназначены для визуализации в своих областях и не применимы для визуализации алгоритмов.

1.3.2. История систем визуализации алгоритмов

История визуализаторов алгоритмов насчитывает более двадцати лет. Первый визуализатор был создан в *Bell Telephone Laboratories* в 1996 для пояснения работы связных списков. Широкое распространение визуализаторы стали получать в начале 80-х.

В 1981 году в Университете Торонто был создан фильм о сортировках [14], что дало большой толчок разработке визуализаторов. Началось использование визуализаторов алгоритмов в процессе обучения. В конце 80-х, начале 90-х были созданы первые системы визуализации: BALSA (Brown ALgorithm Simulator and Animator) [15] и TANGO (Transition-based Animation GeneratiOn) [16]. Эти системы оказали большое влияние на последующие разработки в данной области.

BALSA разработана М. Брауном и Р. Седжвиком в Университете им. Брауна (США) и стала первой широко распространенной системой визуализации. *BALSA* может одновременно отображать как несколько видов одной программы, так и виды разных программ. В последствии Браун и

Седжвик приняли участие в разработке других систем визуализации, в которых были исправлены некоторые недостатки *BALSA*, в частности уменьшены требования к вычислительным ресурсам.

TANGO разработана Дж. Стаско и базируется на представлении данных, которыми оперирует визуализируемый алгоритм, что являлось новым подходом к визуализации алгоритмов.

Впоследствии, идеи заложенные в *TANGO* были развиты в *XTango* (версия *TANGO* для *XWindows*) в которой была реализована плавная анимация. Другим последователем *TANGO* явилась система визуализации *Polka*, оптимизированная для визуализации параллельных программ. В последствии в *Polka* была добавлена возможность трехмерной визуализации.

Другими распространенными системами визуализации являются *Zeus*, *Leonardo*, *CATAI* и *Mocha*. *Zeus* была разработана М. Брауном и являлась последователем *BALSA*. *Zeus* поддерживает возможность отображения нескольких синхронизированных видов программы и может визуализировать параллельные программы. *Leonardo* является интегрированной средой для разработки визуализаторов на языке *C*. *CATAI* так же предназначена для визуализации программ на языке *C*. В ней объединены возможности разработки и визуализации программ. *Mocha* так же был разработана в Университете им. Брауна. В ней использован клиент-серверный подход. Клиенты были реализованы как на *Java* так и под *XWindow*. При этом, на клиентской машине исполняются только команды визуализации, а визуализируемая программа запускается на сервере.

Ранние системы визуализации алгоритмов являлись платформо-зависимыми. В последствии с распространением языка *Java* был осуществлен переход на него, таким образом системы визуализации стали платформо-независимыми. *Java* так же позволила легко выкладывать визуализаторы в *Internet*, что способствовало их широкому распространению. Таким образом стало возможным использовать визуализаторы не только в

учебных заведениях, но и для дистанционного (в том числе, самостоятельного) образования.

На данный момент разработано большое количество систем визуализации. Большинство из них было разработано для целей обучения, хотя некоторые из могут быть использованы для исследования и создания новых алгоритмов. Некоторые системы достаточно просты, например *ANIMAL* и *Agat*, другие же весьма сложны, как, например *Leonardo*.

1.3.3. Обзор общих систем визуализации

Animal

Animal является относительно новой системой визуализации. Работа над первой версией была начата в 1998 году. Сейчас имеется вторая версия, доступная по адресу [45] и описанная в работе [22]. Опыт использования *Animal* в *University of Siegen* (Германия) изложен в [23].

Система создавалась для удовлетворения следующим требованиям [22]:

1. платформонезависимость,
2. доступность,
3. простота использования,
4. поддержка визуализации программ.

Визуализация в *Animal* создается как последовательность кадров, отображающие действия алгоритма. Кадры создаются задаются в специальном, при этом есть возможность описания движения элементов кадра.

Так как визуализация описывается покадрово, ручной ввод данных в созданные визуализаторы невозможен.

Существенным недостатком этой системы визуализации является ручное здание кадров визуализации, что существенно усложняет создание визуализаторов сложных алгоритмов и не обеспечивает должной гибкости.

JAWAA

JAWAA является системой создания визуализаторов с последующим их размещением в *Internet*. Подробное описание *JAWAA* дано в работе [24]. Использование JAWAA в *Duke University* (США) описано в работах [25] и [26].

Визуализаторы в *JAWAA* строятся из примитивов таких как многоугольники, круги и текст. Визуализация задается программой на специальном языке сценариев. В языке предусмотрены команды перемещения как отдельных объектов, так и их групп. Предусмотрена возможность генерации сценария в результате вызовов библиотечных методов из программы на *C/C++*. Таким образом *JAWAA* может быть использована как визуализатор программ (см. раздел [1.3.1]).

Создание визуализаторов алгоритмов на основе *JAWAA* осложнена отсутствием соответствующих библиотек. Вторым недостатком является невозможность создания интерактивных визуализаторов.

1.3.4. Обзор систем визуализации алгоритмов

Обзор систем визуализаторов алгоритмов так же имеется в [21]

BALSA

BALSA является первой системой визуализации получившей широкую известность. Она была разработана в *Brown University* (США) и основана на визуализации программ. *BALSA* и опыт его использования подробно описаны в в [15, 29]. При просмотре визуализатора можно открыть несколько видом программы — различных изображений структур данных и текущей выполняемой операции. В том числе доступны “исторические” виды (см. раздел 1.2).

Визуализаторы описываются на специально разработанном языке описания сценариев, для чего требуется достаточно высокая квалификация, что признавалось и самими авторами [15]. В последствии были разработаны системы *BALSA-II* и *Zeus* построенные на той же идеологии.

Недостатком *BALSA* и ее последователей является сложность описания визуализаторов и ручное обращение операций со структурами данных. Другими недостатками является слабая переносимость и невозможность задания входных данных пользователем.

Tango и XTango

Tango была первой системой визуализации данных. Она разработана разрабатывалась в *Brawn Univesity* (США). В последствии была разработана система *XTango* — версия *Tango* работающая под *XWindow*. *Tango* и *XTango* подробно описаны в работах [16, 17, 18].

Визуализация состоит в отображении структур данных которыми оперирует визуализируемый алгоритм.

Как и в любой системе визуализации данных в ней сложно визуализировать операции не связанные с изменением данных, в том числе отслеживать выполнение алгоритма.

Jeliot

Jeliot — система визуализации алгоритмов, разрабатывается в *Weizmann Institute of Science* (Финляндия) она является *Java*-версией системы визуализации *Eliot*, разработанной там же. Система доступна по адресу [46]. Описание общей *Jeliot* приведено в работах [30, 31], а описание Eliot в работе [32].

Визуализируемые программы пишутся на языке *Java*. При визуализации java-код преобразуется во внутреннее представление. Визуализация очень низкоуровневая, в частности показывается вычисление всех выражений, без возможности пропуска. Осуществляется подсветка текущего оператора и отображается стек вызовов. Так же поддерживается анимационный режим.

Возможно создание интерактивных визуализаторов, посредством использования нестандартных классов ввода-вывода.

Основным недостатком *Jeliot* является неконтролируемость визуализации, т.е. невозможность влияния на изображения, показываемые

пользователю. Это, в совокупности с низкоуровневой визуализацией делает *Jeliot* малопригодным для визуализации сложных алгоритмов.

Polka и Samba

Polka — система визуализации разработанная в *Georgia Institute of Technology* (США). *Samba* является графическим интерфейсом к *Polka*, и образует с ней единую (более развитую) систему визуализации. По-этому они рассматриваются совместно. *Polka* и *Samba* доступны по адресам [47] и [48] соответственно.

Polka является развитием *XTango*, и является системой визуализации данных, специализированной на визуализации параллельных программ. С этой целю, пользователь может одновременно открыть несколько окон с графической информацией. Описание визуализатора осуществляется на специальном языке описания сценариев, программы на котором можно создавать с помощью *Tanga*. Визуализируемые алгоритмы пишутся на языке *C* и компилируются вместе с библиотекой *Polka*.

Возможна организация взаимодействия визуализатора с пользователем, но только через консоль, что является существенным недостатком. Другим недостатком *Polka* является платформозависимость созданных на ее базе визуализаторов. Так же не предусмотрена возможность автоматического вывода комментирующего текста.

Leonardo

Система визуализации данных *Leonardo* разработана в *Universita di Roma "La Sapienza"* (Италия). Версия для MacOS доступна по адресу [49]. Принципы ее работы и построения описаны в [27] и [28].

Для описания визуализации данных был разработан декларативный язык *ALPHA*, комментарии на котором вставляются в текст визуализируемой программы на языке *C*. Программы транслируются в команды для виртуального процессора, который может выполнять только обратимые

вычисления, таким образом все визуализаторы основанные на *Leonardo* являются обратимыми.

Как и в любой системе визуализации данных в Leonardo отсутствует непосредственная возможность вывода программы, комментариев и визуализация действий не изменяющих данные.

Agat

Agat является относительно простой смешанной системой визуализации программ, нацеленной на использование при разработке алгоритмов. Система доступна по адресу [50] и описана в [51].

При выполнении программа с помощью библиотечных функций выводит один или несколько потоков данных. Обычно соответствующие операторы располагают в “интересных” точках программы.

На встроенном языке сценарием можно описывать как взаимодействие потоков так и создавать новые потоки из уже имеющихся. Полученные потоки данных визуализируются по отдельности, в различных окнах программы. Способ визуализации задается формулой, отображающей данные из потоков в точку на экране (в окне). Такой подход позволяет глубоко анализировать алгоритмы и разрабатывать на основе полученных данных новые алгоритмы.

К сожалению, *Agat* практически не применим в целях обучения, так как не позволяет отслеживать выполнение алгоритма и создавать сложные графические образы.

1.3.5. Выводы

Обобщим изложенное в разделах 1.3.3 и 1.3.4 сравнив их с требованиями изложенными в разделе 1.1.2. Столбцы табл. 2 соответствуют свойствам, как указано в разделе 1.1.2. При этом “+” соответствует присутствию свойства, “–” — его отсутствию, а “±” — возможность удовлетворения свойства путем сложной доработки визуализатора.

Табл. 2. Сравнительные характеристики систем визуализации

Система визуализации	1	2	3	4	5	6	7	8	9	10
Animal	–	+	+	+	±	+	+	+	+	–
JAWAA	–	±	–	±	±	+	+	+	–	–
BALSA	–	±	±	+	+	–	–	–	–	–
Tango	–	+	+	–	–	–	–	–	–	+
XTango	–	+	+	–	–	–	–	–	–	+
Jeliot	+	–	–	–	–	–	+	+	–	–
Polka	+	–	–	±	±	–	+	–	+	–
Leonardo	+	+	+	–	±	+	–	–	+	+
Agat	+	–	–	–	–	+	+	+	+	+

Таким образом, ни одна из рассмотренных программ не удовлетворяет всем выдвинутым требованиям. Наиболее полно им удовлетворяют система *Animal*, но визуализаторы созданные с ее использованием не обладают управляемостью, что существенно снижает их педагогическую ценность. Второй недостаток этой системы так же весьма существенен, так как создание кадров визуализаторов сложных алгоритмов вручную практически невозможно.

1.4. Выводы

Ни одна из рассмотренных систем визуализаторов не позволяет создавать визуализаторы полностью удовлетворяющими требованиям изложенным в разделе 1.1.2. Таким образом, встает вопрос об реализации системы визуализации, удовлетворяющей им.Такой системой и должна стать *Vizi*, описанию устройства которой и посвящена оставшаяся часть работы.

Как было показано выше, визуализаторы данных обладают меньшими выразительными способностями по сравнению с визуализатором программ. По этому, система визуализации *Vizi* разрабатывалась как система визуализации программ. В последующих главах построение визуализаторов будет рассматриваться с точки зрения визуализаторов программ.

ГЛАВА 2. ПРОЦЕСС ПОСТРОЕНИЯ ВИЗУАЛИЗАТОРОВ

Процесс построения визуализатора алгоритма достаточно сложен, по этому мы разобьем его на отдельные составляющие. И рассмотрим его с нескольких точек зрения.

В разделе 2.1 рассматриваются основные части визуализатора, и их разработка с использованием системы визуализации *Vizi*, что существенно упрощает разработку. В разделе 2.2 описываются общий порядок разработки визуализатора? порядок разработки с использованием *Vizi* и их различие. Затем рассматривается документация создаваемая при выполнении проекта визуализатора (раздел 2.3). В заключении главы приведены список используемых терминов (раздел 2.4) и полученные выводы

2.1. Основные части визуализатора

Визуализатор является сложным программным продуктом, по этому выделим из него основные части, которые можно рассматривать самостоятельно. Тогда процесс построения визуализатора сведется к построению этих частей и их интеграции.

В разделе 2.1.1 рассматриваются основные части, которые можно выделить из визуализатора. Затем в разделе 2.1.2 рассматривается разработка частей визуализатора и помощь, которую может оказать при этом система визуализации *Vizi*. В заключении раздела следуют выводы о применимость *Vizi* при создании визуализаторов алгоритмов (раздел 2.1.3).

2.1.1. Выделение основных частей визуализатора

Предлагаем выделить следующие основные части визуализатора:

- *логика визуализатора*;
- *визуальное представление*;
- *набор комментариев*;
- *элементы управления*;
- *интерфейс визуализатора*;

- *проектная документация*.

Логика визуализатора — часть визуализатора, осуществляющая передвижение по алгоритму и предоставляющая данные другим частям визуализатора для отображения их пользователю. Логика визуализатора так же определяет интересные состояния, в которых информация отображается пользователю.

На пример, в алгоритме поиска максимального потока в сети вряд ли стоит визуализировать поиск кратчайшего пути между вершинами. В таком случае, действие таких “подалгоритмов” не содержит интересных состояний и отображается как одна операция.

В соответствии с требованиями, выдвинутыми в разделе 1.1.2 логика визуализатора должна обеспечивать возможность движения по алгоритму как вперед так и назад на неограниченное количество шагов. При этом, состояние визуализатора на любом шаге должно оставаться неизменным, вне зависимости от последовательностью действий совершенных пользователем.

Визуальное представление — часть визуализатора, определяющая что и как будет отображаться пользователю различных интересных состояниях. Обычно визуальное представление задается набором изображений, (схем изображений), которые отображаются пользователю в процессе визуализации.

Визуальное представление служит для облегчения понимания визуализируемого алгоритма. На пример, при построении визуализатора алгоритма на графах граф может отображаться различными способами: графически (кружками и стрелками), матрицей смежности, списками ребер и т.д. При построении визуального представления мы должны выбрать, как будет отображаться граф в нашем случае. На пример при визуализации алгоритма Флойда естественным является представление графа матрицей весов, а для алгоритма Краскала — списками ребер.

Набор комментариев — часть визуализатора, определяющие какие комментарии будут отображаться пользователю в каждом интересном

состоянии. Комментарий, отображаемый пользователю может включать в себя данные, предоставленные логикой визуализатора.

Комментарии поясняют текущее действие алгоритма и помогают пользователю глубже и быстрее понять смысл производимых действий, а следовательно и сам алгоритм. Хороший набор комментариев позволяет понять алгоритм без визуального представление.

Элементы управления — часть визуализатора, через которую пользователь управляет визуализатором. При этом, определяются набор действия которые пользователь может осуществлять с визуализатором и связь этих действий с логикой визуализатора. Примерами таких действий могут служить передвижение по алгоритму вперед и назад (при этом могут быть использованы маленькие и большие шаги), изменение количества элементов (вершин в графе, размерность массива), генерация случайных исходных данных и другие возможности.

Интерфейс визуализатора определяет, каким остальные части визуализатора отображаются на экране и взаимодействие пользователя с элементами управления. В нашем случае интерфейс визуализатора должен определять отображение визуального представления и комментариев пользователю.

Проектная документация отображает все стадии разработки визуализатора и описывает получившийся продукт.

2.1.2. Разработка основных частей

Рассмотрим разработку основных частей визуализатора по отдельности и покажем, как система визуализации *Vizi* может быть использована для упрощения разработки.

Заметим, что несмотря на то что строится визуализатор алгоритма мы часто говорим о визуализируемой программе, а не о визуализируемом алгоритме. Эти понятия существенно различаются.

- *Визуализируемый алгоритм* — алгоритм, который поясняет визуализатор.
- *Визуализируемая программа* — конкретная реализация алгоритма, на основе которой строится визуализатор.

Логика визуализатора

Логика визуализатора разрабатывается для каждого визуализатора в отдельности, при этом наличие логики визуализатора схожего алгоритма не облегчает этот процесс.

В соответствии с назначением логика визуализатора может быть представлена как совокупность двух частей:

- *программа визуализации*,
- *модель данных.*

Программа визуализации служит для передвижения по алгоритму. В соответствии с выдвинутыми требованиями она должна быть обратимой, т.е. допускать движение как вперед так и назад. Программа визуализации так же определяет интересные состояния и должны сигнализировать другим частям об их наступлении.

Модель данных является структура данных, в которой хранятся значения всех переменных программы. Она служит для сохранения состояния алгоритма а так же передачи информации от реализации к другим частями визуализатора.

Таким образом текущее состояние модели данных и точка выполнения программы визуализации (отсчитываемая по динамическим и текстуальным индексам [9]) полностью задает состояние визуализатора.

Система визуализации *Vizi* (см. главу 6) облегчает создание обоих частей логики визуализатора. Для этого визуализируемая программа записывается в виде XML-описания (см. раздел 5.3), которое полностью повторяет структуру исходной программы.

Программа визуализации строится на основе системы взаимосвязанных конечных автоматов, которая генерируется по XML-описанию

визуализируемой программы. При этом для каждой процедуры строятся прямой и обратный автоматы, позволяющие осуществлять шаги вперед и назад (см. главу 4).

Модель данных так же автоматически генерируется по XML-описанию визуализируемой программы и является структурой (классом), в которую вынесены переменный используемые визуализируемой программой.

Визуальное представление

Визуальное представление шагов алгоритма должно быть выбрано так, чтобы наиболее наглядным образом отобразить состояние визуализируемого алгоритма.

В большинстве случаев не имеет смысла разрабатывать визуальное представление с нуля. Для этого воспользоваться уже созданными элементами, входящими в систему визуализации *Vizi* (см. раздел 6.3.2). На пример, во многих случаях требуется отображать одинаковые структуры данных, такие как таблицы, массивы и списки. В этих случаях следует воспользоваться уже разработанными элементами для их отображения.

Действия осуществляемые для создания соответствующего изображения в каждом интересном состоянии могут быть указаны непосредственно в XML-описании визуализируемой программы (см. раздел 5.3.3), что облегчает программирование.

Визуальное представлении разрабатывается после того, как будут определены интересные состояния алгоритма. При этом разработанное визуальное представление может не быть самодостаточным, а требовать дополнительных комментариев, для чего и введена соответствующая возможность.

Набор комментариев

В отличие, от визуального представления и элементов управления комментарии необходимо разрабатывать “с нуля”, так как для каждого алгоритма они уникальны.

По комментариям в совокупности с визуальным представлением должно быть понятно, как работает алгоритм, и зачем выполняются те или иные действия. Оперирования с конкретными числами (значениями) во многом помогает разобраться в алгоритме. По тому, система визуализации *Vizi* предоставляет средства для включения в комментарии конкретных значений переменных во время исполнения (см. раздел 5.3.3).

Комментарии разрабатываются, когда уже выбраны интересные состояния алгоритма, и могут ссылаться на визуальное представление.

Элементы управления

В большинстве визуализаторов используются сходные элементы управления. В частности все визуализаторы должны содержать кнопки переходов в перед и назад по алгоритму.

В система визуализации *Vizi* (см. раздел 6.3.3) представлены часто используемые элементы управления, интегрированные с автоматным представлением программы визуализатора.

Интерфейс визуализатора

При выполнении нескольких визуализаторов следует придерживаться единого интерфейса. Таким образом, пользователям освоившим один визуализатор не составит труда разобраться в остальных.

В системе визуализации *Vizi* реализован единый интерфейс визуализаторов, состоящий из трех областей, расположенных одна над другой (см. раздел 6.3.1):

- область визуального представления;
- область комментариев;
- область элементов управления.

Такой интерфейс делает простой и единообразной работу с визуализаторам и позволяет получать скриншоты визуального представления с комментариями, например для публикации.

Проектная документация

Проектная документация является неотъемлемой частью визуализатора. Она позволяет легко разобраться во внутренней структуре визуализатора и помогает искать и устранять ошибки. Более подробно проектная документация рассмотрена в разделе 2.3.

2.1.3. Выводы

Как показано выше система визуализации *Vizi* позволяет упростить программирование всех основных частей визуализатора, кроме создания проектной документации, что существенно убыстряет создание визуализаторов.

2.2. Порядок разработки визуализатора

В настоящем разделе рассматривается порядок разработки визуализаторов как с использованием системы визуализации *Vizi* (раздел 2.2.2), так и без нее (раздел 2.2.1) и показывается, что первый существенно менее трудоемок.

2.2.1. Общий порядок разработки визуализатора

Рассмотрим порядок разработки визуализаторов, состоящий их десяти этапов. Некоторые этапы дополнительно разбиты на отдельные шаги.

1. Анализ литературы.
2. Создание визуализируемой программы.
    - Реализация алгоритма.
    - Отладка реализации.
3. Разработка концепции визуализатора.
    - Выделение интересных состояний.
    - Разработка концепции визуального представления.
    - Разработка набора комментариев.
    - Разработка элементов управления.
4. Построение логики визуализатора.
    - Выделение модели данных.

- Построение и отладка программы визуализатора.

5. Реализация визуального представления.
6. Реализация элементов управления.
7. Интеграция логики визуализатора, визуального представления, набора комментариев и элементов управления.
8. Создание интерфейса визуализатора.
9. Общая интеграция и отладка визуализатора.
10. Оформление проектной документации.

На первом этапе производится анализ литературы, при этом рассматриваются существующие модификации алгоритма и одна из них выбирается для визуализации.

На следующем этапе реализуется выбранная модификация алгоритма. Одновременно полученная реализация отлаживается. При этом выделяются шаги алгоритма, требующие особого внимания. Полученная информация используется на следующем этапе.

На третьем этапе разрабатывается общая концепция визуализатора. Для этого с начала выделяются те шаги алгоритма, которые представляют наибольший интерес, и соответствующие интересные состояния. Затем разрабатывается общая концепция визуального представления и ее конкретизация для каждого выделенного шага. Одновременно с разработкой визуального представления для каждого шага пишутся комментарии, поясняющие действия, выполняемые алгоритмом. После этого разрабатываются элементы управления визуализатором, в частности определяется, какие параметры и в каких пределах сможет регулировать пользователь.

На четвертом этапе производится построение логики визуализатора, в частности выделение модели данных и создание программы визуализатора. Построение программы визуализации может оказаться не только очень трудоемким, но и сложным с идейной точки зрения, так как требуется создание обратимой программы.

На пятом и шестом этапе осуществляется реализация концепций визуального представления и элементов управления соответственно.

После чего на седьмом этапе производится интеграция результатов этапов 4-7, при этом могу возникнуть сложности, для решения которых придется вернуться на один из предыдущих этапов.

Восьмой этап может быть выполнен один раз при разработке набора визуализаторов, что может существенно уменьшить затраты.

На девятом этапе производится окончательная интеграция визуализатора и его отладка. Выделенные ошибки для своего исправления могут потребовать возврата к предыдущим этапам.

На заключительном этапе оформляется проектная документация. Заметим, что проектная документация должна вестись все время пока выполняется проект. А на последнем этапе производится только ее оформление и обобщение полученных результатов.

2.2.2. Порядок разработки визуализатора с использованием Vizi

Заметим, что общий порядок разработки визуализатора достаточно сложен, а этапы 4-9 весьма трудоемки. При этом четвертому этапу присуща не только техническая, но и идеологическая сложность.

Система визуализации *Vizi* позволяет упростить и убыстрить этот процесс. Опишем соответствующий порядок разработки визуализатора.

1. Анализ литературы.
2. Создание визуализируемой программы.
    - Реализация алгоритма.
    - Отладка реализации.
3. Разработка концепции визуализатора
    - Выделение интересных состояний.
    - Разработка концепции визуального представления.
    - Разработка набора комментариев.
    - Разработка элементов управления.

4. Построение XML-описания визуализируемой программы.
    - Выделение модели данных.
    - Упрощение конструкции программы.
    - Запись XML-описания.
    - Отладка XML-описания.
5. Добавление комментариев к XML-описанию.
6. Реализация визуального представления
7. Реализация элементов управления.
8. Интеграция визуализатора.
    - Генерация кода по XML-описанию.
    - Совместная отладка логики и визуального представления.
9. Оформление проектной документации.

Первые три этапа повторяют соответствующие этапы общего порядка разработки визуализатора. Заметим, что реализации алгоритма на втором этапе следует избегать использования сложных структурных конструкций, что в последствие облегчит упрощение программы для записи XML-описания.

На четвертом этапе осуществляется построение XML-описания визуализируемой программы.

Для этого сначала выделяется модель данных. При использовании системы визуализации *Vizi* этот шаг существенно упрощается, так как возможно использование локальных переменных и параметров процедур предусмотрена в XML-описании. Таким образом, этот шаг сводится к простому преобразованию заголовков процедур в XML-формат (см. раздел 5.3.2). Последующее преобразование модели данных описано в разделах 3.2 и 3.3.

После выделения модели данных производится упрощение структуры. При этом программа должна использовать только следующие операторы, последние пять из которых являются управляющими:

- оператор присваивания;
- последовательность операторов (составной оператор);

- укороченный оператор ветвления (`if-then`);
- полный оператор ветвления (`if-then-else`);
- цикл с предусловием (`while`);
- вызов процедуры.

После упрощения получается программа, более удобная для преобразования в систему автоматов, так как она не содержит громоздких синтаксических конструкций. Данное преобразование основано на теореме структурирования [5], в соответствии с которой любую программу можно преобразовать так, что она будет содержать только три типа управляющих конструкций: последовательность операторов, один из операторов ветвления и один из операторов цикла. Набор используемых управляющих конструкций может быть, как сокращен до последовательности операторов и оператора цикла с предусловием, так и расширен, например, как в настоящей работе, за счет введения процедур (также с одним входом и одним выходом) и применения двух типов оператора ветвления: полного и укороченного. Цикл с предусловием (`while`) выбран потому, что он является наиболее универсальным. Упрощение структуры программы рассматривается в разделе 3.4.

Упрощенная программа может быть непосредственно записана в XML-формате, так как в нем предусмотрены элементы для кодирования всех видов операторов. Программа, записанная в XML-формате может быть автоматически преобразована системой визуализации *Vizi* в систему взаимосвязанных автоматов, как указано (в главе 4). XML-формат подробно описан (в главе 5).

После записи программы в XML-формате она отлаживается с использованием средств, предоставляемых системой визуализации *Vizi* (см. раздел 6.4.2).

На пятом этапе к XML-описанию визуализатора добавляются комментарии, разработанные на третьем этапе. Для этого в каждом у каждого

элемента, соответствующего оператору исходной программы заполняются поля комментариев и их параметров (см. раздел 5.3.3).

На шестом этапе реализуется концепция визуального представлении шагов визуализатора. Этот этап может быть упрощен за счет использование визуальных элементов входящих в *Vizi*. Для связи визуального представления и XML-описания используются элементы `draw` (см. раздел 5.3.3).

Реализация элементов управления на седьмом этапе существенно упрощается за счет использование элементов управления входящих в систему визуализации *Vizi*.

На восьмом этапе осуществляется интеграция визуализатора с использованием средств предоставляемых *Vizi*. При этом по XML-описанию автоматически генерируется программа визуализации, модель данных, система отображение комментариев, а так же осуществляется интеграция с реализацией визуального представления и стандартными элементов управления. После отладки получается готовый визуализатор.

Последний этап по своей сути эквивалентен десятому этапу общего порядка разработки визуализаторов.

Сравнение общего порядка разработки визуализатора и порядка разработки с использованием системы визуализации *Vizi* показывает, что последний требует меньших затрат как технических так и интеллектуальных усилий. Что особенно сильно проявляется при построении логики визуализатора.

2.3. Проектная документация

Проектная документация является неотъемлемой частью проекта визуализатора. Она существенно облегчает как последующее сопровождение визуализатора, так и разработку похожих визуализаторов за счет заимствования проектных решений.

2.3.1. Структура проектной документации

Предлагаемая проектная документация должна содержать следующие части:

- аннотация;
- пояснительная записка;
- описание интерфейса визуализатора.

В *аннотации* содержится краткое описание выполненного проекта (10-20 строк) в частности:

- название визуализируемого алгоритма;
- источник, по которому реализовывался алгоритм;
- информация об исполнителе проекта;
- информация о научном руководителе (если есть);
- общие сведения о визуализаторе.

Пояснительная записка — наиболее объемный документ. В нем отражаются все этапы выполнения визуализатора и достигнутый результат. Подробно структура пояснительной записки рассмотрена в разделе 2.3.1.

Описание интерфейса визуализатора содержит краткое описание различных частей визуализатора и его элементов управления. С помощью этого документа неподготовленный пользователь может научиться выполнять не только элементарные действия (такие как просмотр алгоритма), но и более сложные (на пример, сохранения и загрузка состояния визуализатора).

2.3.2. Структура пояснительной записки

Пояснительная записка к визуализатору должна иметь следующую структуру:

- *Введение*
- *Раздел 1*. Анализ литературы.
- *Раздел 2*. Описание алгоритма.
- *Раздел 3*. Реализация визуализируемого алгоритма.
- *Раздел 4*. Концепция визуализатора.

- *Раздел 5*. Преобразование визуализируемой программы в XML-описание.
- *Раздел 6.* Реализация концепции визуализатора.
- *Раздел 7.* Описание конфигурации визуализатора.
- *Раздел 8.* Интеграция визуализатора и отладка
- *Заключение*
- *Список источников*
- *Приложения*

Во *введении* содержится краткое описание выполненного проекта. В частности, описание сути алгоритма, области его применения и примеры использования.

В *разделе 1* рассматриваются источники, из которых взят алгоритм. Если существует несколько модификация алгоритма, то требуется указать ссылки на них. Так же в этом разделе даются ссылки на другую используемую литературу.

Раздел 2 должен содержать подробное описание выбранной модификации (если таковые есть) визуализируемого алгоритма. Если визуализируемый алгоритм использует другие алгоритмы как составные части, то должны быть указаны ссылки на модификации используемых “подалгоритмов”.

В *разделе 3* описывается реализация алгоритма, которая будет визуализирована. В частности, приводятся комментарии к реализации алгоритма, и обоснование неочевидных решений. Данный раздел ссылается на *приложение 1*.

В *разделе 4* описывается концепция визуализатора, в том числе:

- выделение интересных состояний;
- разработка визуального представления и комментариев для каждого интересного состояния;
- элементы управление визуализатора.

В *разделе 5* описывается преобразование визуализируемой программы в XML-описание, описание возникших при этом проблем и их решение Раздел должен ссылаться на реализацию алгоритма после выделения модели данных (*приложение 2-4*).

В Разделе 6 описывается реализация концепции визуализатора в соответствии с идеями разработанными в четвертом разделе. интерфейс визуализатора, в частности назначение элементов управления.

Раздел 7 содержит описание конфигурации визуализатора. В частности, описание всех параметров визуализатора, их назначения и значений по умолчанию. Ссылается на XML-описание визуализатора (*приложение 4*).

Раздел 8 описывает процесс интеграции визуализатора, позникшие при этом проблемы и их решение. Так же в этом разделе приводятся скриншоты визуализатора и описываются детали реализации визуального представления. Данный раздел ссылается на *приложение 6*.

В *Заключении* отражается точка зрения автора на выполненный проект.

Список источников должен содержать информацию об использованных источниках, на которые ссылаются другие разделы. Источники должны быть разделены на печатные издания и Интернет-ресурсы.

Приложения содержат код, получаемый на различных этапах построения визуализатора:

- *Приложение 1.* Исходный текст реализации алгоритма на языке *Java*.
- *Приложение 2.* Исходный текст реализации алгоритма после ее упрощения.
- *Приложение 3*. XML-описание визуализатора, включая конфигурацию.
- *Приложение 5.* Сгенерированные исходные коды автомата.
- *Приложение 6.* Исходные коды интерфейса визуализатора.

Заметим, что структур пояснительной записки соответствует порядку построения визуализатора на основе системы визуализации *Vizi*, описанной в разделе 2.2.2.

2.4. Используемые термины

Визуализаторы и системы визуализации

Визуализатор алгоритма (визуализатор) — программа, отображающая на экране ход и/или результаты выполнения алгоритма (программы).

Система визуализации — программный комплекс позволяющий создавать и исполнять визуализаторы.

Система визуализации алгоритмов — *система визуализации*, предназначенная для создания *визуализаторов алгоритмов*.

Система визуализации данных — система визуализации алгоритмов, предназначенная для создания *визуализаторов данных*.

Визуализатор данных — визуализатор, отображающий изменения в структурах данных, происходящих при выполнении алгоритма.

Система визуализации программ — система визуализации алгоритмов, предназначенная для создания *визуализаторов программ*.

Визуализатор программы — визуализатор, отображающий действия осуществляемые при выполнении алгоритма.

Интересное состояние — состояние алгоритма, отображаемое пользователю.

Части визуализатора

Логика визуализатора — часть визуализатора, осуществляющая передвижение по алгоритму и предоставляющая данные другим частям визуализатора для отображения их пользователю.

Визуальное представление — часть визуализатора, определяющая что и как будет отображаться пользователю на различных стадиях визуализации.

Набор комментариев — часть визуализатора, определяющие какие комментарии будут отображаться пользователю в каждом интересном состоянии.

Элементы управления — часть визуализатора, через которую пользователь управляет визуализатором.

Интерфейс визуализатора — часть визуализатора, определяющая, каким образом остальные части визуализатора отображаются на экране и взаимодействие пользователя с *элементами управления*.

Проектная документация — часть проекта визуализатора, содержащая информацию о всех стадиях разработки визуализатора и описывающая получившийся продукт.

Программа визуализации — часть визуализатора, служащая для передвижения по алгоритму и сигнализирующая о наступлении интересных состояний.

Модель данных — структура данных, в которой хранятся значения всех переменных *программы визуализации*.

Программы и автоматы

Итеративная программа — программа не использующая рекурсию.

Рекурсивная программа — программа использующая рекурсию.

Прямая рекурсия — рекурсивная процедура непосредственно содержит вызов самой себя.

Косвенная рекурсия — рекурсивная процедура может осуществлять вызов самой себя посредством других процедур.

Выделение модели данных из программы — приведение программы к виду, использующему модель данных. Состоит из двух этапов: *построение модели данных по программе* и последующей модификации программы.

Построение модели данных по программе — создание определения модели данных (переменные и их типы) по программе.

Пара автоматов — прямой и *обратный* автоматы построенные по одной процедуре и имеющие общие состояния.

Прямой автомат — автомат, осуществляющий передвижение по алгоритму (программе) “вперед”.

Обратный автомат — автомат, осуществляющий передвижение по алгоритму (программе) “назад”.

Прочие термины

Визуализируемая программа — конкретная реализация алгоритма, на основе которой строится визуализатор.

Визуализируемый алгоритм — алгоритм, который поясняет визуализатор.

XML-описание визуализатора — запись информации о визуализаторе в *XML-формате*, для ее последующей автоматической обработки.

XML-описание визуализируемой программы — запись визуализируемой программы в *XML-формате*.

XML-формат — формат записи описания визуализатора, подробно описывается в (главе 5).

2.5. Выводы

Использование системы визуализации *Vizi* может существенно упростить реализацию как отдельных частей визуализатора, так и проекта в целом. Это достигается за счет интеграции частей визуализаторов через XML-описание, с последующей его автоматической обработкой.

Так же изложена предлагаемая структура проектной документации, согласованная с порядком разработки визуализатора и позволяющая раскрыть все стороны проекта.

ГЛАВА 3. ПОСТРОЕНИЕ МОДЕЛИ ДАННЫХ ПО ПРОГРАММЕ. УПРОЩЕНИЕ СТРУКТУРЫ ПРОГРАММЫ

В данном разделе описывается выполнения второго и третьего шагов предложенного метода. В разделе 3.1 рассматриваются требования, предъявляемые к преобразуемой программе.

Далее в разделах 3.2 и 3.3 рассматривается выделение модели данных из итеративных (не содержащих рекурсию) и рекурсивных программ соответственно. Выделение модели данных осуществляется в два этапа:

- *Построение модели данных*. По программе строится модель данных, в которую выносятся все используемые в программе переменные.
- *Модификация программы*. Программа модифицируется так, что бы она использовала только переменные модели данных.

Так же в каждом разделе приведен пример построения модели данных и модификации программы.

В разделе 3.4 рассматривается упрощение структуры программы (третий шаг предложенного метода). В результате упрощения, программа приводится к виду, удобному для преобразования в систему взаимодействующих автоматов.

3.1. Требования к исходной программе

В последующих разделах рассматривается программа (реализация алгоритма), построенная на первом этапе. При этом она должна удовлетворять следующим требованиям.

- Параметры и возвращаемые значения процедур передаются по значению (это не исключает передачу ссылок или указателей).
- В идентификаторах не должен использоваться символ подчеркивания (“_”).

- Используемые выражения не должны иметь побочных эффектов, то есть значения переменных изменяются только оператором присваивания.

Первое ограничение не существенно, так как все современные языки позволяют возвращать из процедур объекты и/или структуры. В языках *Java*, *C* и *C++* все значения передаются по значению. В языке *Паскаль* возможен обход этого правила, с использованием ключевого слова `var`, но такие программы легко преобразуются в виду, не использующему данную особенность, путем использования указателей.

Второе ограничение наложено для того, что бы символ подчеркивания мог быть использован для образования составных идентификаторов.

Третье ограничение исключает использование цепных операторов присваивания, а так же операции инкремента (++) и декремента (--)в языках *Java*, *C* и *C++*. Цепные операторы присваивания легко преобразуются в последовательность присваиваний. В свою очередь, выражения использующие операции инкремента и декремента преобразуются в набор выражений.

3.2. Построение модели данных по итеративной программе

В соответствии с синтаксисом большинства процедурных языков программирования все переменные, используемые в реализации алгоритма можно разделить на два класса:

- *глобальные* — определенные на уровне программы и доступные во всех процедурах;
- *локальные* — определенные в рамках процедуры и доступные только в ней.

Отметим, что если переменным модели давать те же имена, что и исходным переменным, то возможно дублирование имен переменных модели, что не допустимо. Для решения этой проблемы предлагается использовать правила изложенные ниже. При этом правила выделения глобальных и локальных переменных различны и будут рассматриваться отдельно.

3.2.1. Построение модели данных

В случае итеративных программ построение модели разбивается на четыре шага.

Первоначально модель данных пуста.

На первом шаге в модель выносятся глобальные переменные. Так как имена всех глобальных переменных различны, то их можно непосредственно вынести в модель. Этот шаг можно формализовать следующим образом.

1. Для каждой глобальной переменной в модель добавляется переменная с именем, совпадающим с именем соответствующей глобальной переменной. Тип добавленной переменной соответствует типу глобальной переменной.

На втором шаге осуществляется выделение локальных переменных в модель. Работа с локальными переменными усложняется, так как в разных процедурах могут быть определены переменные с одинаковыми именами. Поэтому при построении модели используется следующий прием: если переменная `a` определена в процедуре `calcSum`, то соответствующая ей переменная модели будет иметь имя `calcSum_a`. В общем случае этот шаг может быть описан следующим образом.

2. Для каждой локальной переменной в модель добавляется переменная с именем вида:

   <имя процедуры>_<имя переменной>

   При этом используется второе требование к исходной программе (раздел 3.1). Тип добавленной переменной соответствует типу локальной переменной.

Переходя к третьему шагу, отметим, что в случае итеративных программ для аргументов процедур можно использовать ту же схему именования, как и для локальных переменных, что может быть сформулировано следующим образом:

3. Для каждого формального аргумента процедуры в модель добавляется переменная с именем вида:

<имя процедуры>_<имя формального аргумента>

Тип переменной модели соответствует типу формального аргумента.

В заключение, на четвертом шаге для процедур, возвращающих значения, необходимо добавить в модель переменные для хранения возвращаемых значений.

4. Для каждой процедуры, возвращающей значения, в модель добавляется переменная с именем вида:

<имя процедуры>_

и типом, соответствующим типу возвращаемого значения.

Предложенная схема образования идентификаторов позволяет не только избежать дублирования имен переменных модели, что требуется синтаксисом языка, но и однозначно восстанавливать по имени переменной из модели, какой переменной из какой процедуры она порождена. Правила такого восстановления изложены в разделе 3.3.4.

Заметим, что на шагах два и три используется правило большинства языков, не позволяющее в одной процедуре иметь локальные переменные и формальные параметры с одинаковыми именами. В языках, разрешающих такие именования, можно имена переменных модели, соответствующих формальным аргументам процедуры, начинать с символа подчеркивания.

3.2.2. Модификация программы

Модифицируем программу, так что бы она использовала только переменные модели данных, построенной, как указанно в предыдущем разделе.

В дальнейшем, предполагается, что экземпляр модели доступен всем процедурам, как переменная с именем `data`.

Так как программа итеративная, то для каждой локальной переменной, формального параметра процедуры и возвращаемого значения можно выделить статический участок памяти. Фактически это и производится при построении модели данных (шаги 1–4 предыдущего раздела).

Для модификации программы совершим следующие действия.

1. Удалим объявления глобальных переменных.
2. Добавим объявление глобальной переменной `data`.
3. Так как имена глобальных переменных при вынесении в модель не изменились, то для использования глобальных переменных требуется добавить к ним префикс “`data.`” .
4. Если некоторые локальные переменные инициализируются одновременно с их объявлением, то разобьем каждое объявление на две части: объявление и инициализация.
5. Удалим объявления локальных переменных.
6. Ко всем обращениям к локальным переменным добавим префикс вида

   `data.`<*имя процедуры*>`_`

В результате выполнения этих шагов все обращения к локальным переменным будут заменены обращениями к переменным модели.

Последующие шаги преобразуют вызовы процедур.

7. Операторы, вызывающие одну и туже процедуру более одного раза, разбиваются так, что бы ни одна процедура не вызывалась дважды в одном операторе.
8. Вызов процедуры в выражении разбивается на две части:
   - оператор вызова процедуры (с сохранением возвращаемого значения в соответствующей переменной модели);
   - исходное выражение, с заменой вызова процедуры на обращение к переменной вида `data.`<*имя процедуры*>`_`.
9. Вызовы процедур преобразуются следующим образом. Пусть вызывается процедура с *N* формальными аргументами. Обозначим имена этих аргументов следующим образом:

   аргумент1, *аргумент2*, ... *аргументN*.

Реальные аргументы обозначим как

выражение1, *выражение2*, ..., *выражениеN*

соответственно. Тогда оператор вызова процедуры заменяется на *N* операторов, вида:

`data.`*<имя процедуры>*`_`*<аргументM>* `=` *<выражениеM>*`;`

где *M* пробегает значения от 1 до *N* и вызов процедуры без аргументов.

10. Обращения к формальным аргументам процедуры заменим обращением к соответствующим переменным модели.
11. Заменим каждый оператор возврата значения

    `return` *выражение*`;`

    на присваивание значения выражения переменной

    `data.`*<имя процедуры>*`_`

    и простой оператор возврата (без возвращаемого значения).
12. Заголовки всех процедур изменяются так, чтобы процедуры не имели параметров и возвращаемых значений.

Отметим, что программа является работоспособной, не только после выполнения всех шагов, но и после выполнения первых двух, трех, четырех, шести или семи шагов, что может быть использовано для проверки правильности производимых преобразований.

3.2.3. Пример построения модели данных и модификации программы

Рассмотрим изложенный метод на примере программы, вычисляющей максимальный из локальных минимумов в массиве натуральных целых чисел (для примеров здесь и в дальнейшем используется язык *Java*).

Исходная программа имеет следующий вид:

```
int[] a; // Массив, в котором осуществляется поиск

/** Подсчитывает указанный максимум */
void calc() {
  int m = 0;
  for (int i = 1; i < a.length - 1; i++) {
    if (isMin(a[i-1], a[i], a[i+1]) && a[i] > m) m =
        a[i];
  }
```

```
}

/** Возвращает, является ли b минимумом аргументов */
boolean isMin(int a, int b, int c) {
  int m = a > b ? a : b;
  return b == (m > c ? m : c);
}
```

Построим модель данных, как указано в разделе 3.2.1:

```
/** Модель данных */
Class Model {
  int[] a; // Глобальная переменная
  int calc_m, calc_i, isMin_m; // Локальные переменные
  int isMin_a, isMin_b, isMin_c; // Аргументы isMin
  boolean isMin_; // Возвращаемое значение isMin
}
```

В примерах, для экономии места будем использовать @-нотацию: запись вида `@a` будет означать обращение к переменный `a` модели данных и эквивалентна записи

```
data.a
```

А запись вида `#a` будет означать ссылку на локальную переменную, вынесенную в модель данных. На пример, если в процедуре `calcSum` была определена переменная `a`, то в ней запись `#a` будет эквивалентна записи

```
data.calcSum_a
```

Перейдем к модификации программы. После выполнения первых трех шагов из раздела 3.2.2 процедура `calc` приобретает следующий вид:

```
Model d; // Объявление переменной модели

void calc() {
  int m = 0;
  for (int i = 1; i < a.length - 1; i++) {
    if (isMin(@a[i-1], @a[i], @a[i+1]) && @a[i] > m)
        m = @a[i];
  }
}
```

А процедура `isMin` осталась без изменений.

Выполнение шагов 4–6 изменяет обе процедуры:

```
void calc() {
  #m = 0;
  for (#i = 1; #i < @a.length-1; #i++) {
    if (isMin(@a[#i-1], @a[#i], @a[#i+1]) && @a[#i] > #m)
        #m = @a[#i];
  }
}
```

```
boolean isMin(int a, int b, int c) {
  #m = a > b ? a : b;
  return b == (#m > c ? #m : c);
}
```

Так как в рассматриваемой программе осуществляется единственный вызов процедуры, то на седьмом шаге программа не модифицируется.

После выполнения шагов восемь и девять имеем:

```
void calc() {
  #m = 0;
  for (#i = 1; #i < @a.length-1; #i++) {
    @isMin_a = @a[#i - 1];
    @isMin_b = @a[#i];
    @isMin_c = @a[#i + 1];
    isMin();
    if (@isMin_ && @a[#i] > #m)
        #m = @a[#i];
  }
}

boolean isMin(int a, int b, int c) {
  #m = a > b ? a : b;
  return b == (#m > c ? #m : c);
}
```

Модификация завершается применением шагов 10–12 к процедуре `isMin`. Итоговая программа выглядит следующим образом:

```
class Model {
  int[] a; // Глобальная переменная
  int calc_m, calc_i, isMin_m; // Локальные переменные
  int isMin_a, isMin_b, isMin_c; // Аргументы isMin
  boolean isMin_; // Возвращаемое значение isMin
}

Model d; // Объявление переменной модели

void calc() {
  #m = 0;
  for (#i = 1; #i < @a.length-1; #i++) {
    @isMin_a = @a[#i - 1];
    @isMin_b = @a[#i];
    @isMin_c = @a[#i + 1];
    isMin();
    if (@isMin_ && @a[#i] > #m)
        #m = @a[#i];
  }
}

void isMin() {
  #m = #a > #b ? #a : #b;
  @isMin_ = #b == (#m > #c ? #m : #c);
```

```
}
```

Заметим, что при выделении модели, объем исходного кода несколько увеличился, но при автоматизированной обработке это несущественно.

3.3. Построение модели данных по рекурсивной программе

В отличие от итеративных программ, при рекурсивном вызове процедур, для каждой локальной переменной выделяется новая область памяти, в то время, как в модели ей соответствует только одна переменная. Таким образом, возникает проблема, связанная с тем, что при рекурсивном вызове могут изменяться локальные переменные другого экземпляра рекурсивной процедуры. Для учета этой особенности нужно несколько изменить правила изложенные в разделе 3.2.1.

3.3.1. Построение модели данных

Если в программе используется только косвенная рекурсия, то можно непосредственно применять правила именования, изложенные в разделе 3.2.1.

При использовании непосредственной рекурсии требуется ввести дополнительные переменные модели:

5. для каждого формального аргумента процедуры, в модель дополнительно к переменным, добавленным на шаге 3, добавляется переменная с именем вида

   <имя процедуры>_<имя формального аргумента>_

   Тип добавленной переменной соответствует типу формального аргумента.

Добавленные таким образом переменные будут использоваться как временные хранилища значений реальных аргументов вызываемой процедуры.

3.3.2. Модификация программы

Для хранения локальных переменных и реальных аргументов процедур требуется не только модель, но и дополнительная память. Так как целью второго и третьего шагов описываемого метода является подготовка

программы к преобразованию в автомат, чему посвящена глава 4, то будет использован прием со стеком описанный в разделе 4.3.

Для этого введем оператор *квази-присваивания* (в программах он будет обозначаться “`@=`”) который не только изменяет значение своего левого аргумента, но и сохраняет замещенное значение в стеке. Так же нам понадобится оператор извлечения значений, сохраненных оператором *квази-присваивания*, обозначаемый “`?=`”. Эквиваленты этих операторов легко реализуются на любом языке программирования.

При модификации программ, использующих рекурсию, шаги 1-8 и 10-12 (раздел 3.2.2) остаются без изменений, а шаг 9 заменяется в зависимости от типа рекурсивного вызова.

Если вызов не является непосредственной рекурсией, то девятый шаг выглядит следующим образом (для сокращения записи в место <*имя процедуры*>_<*аргументM*> будем писать *имяM*):

- Вызовы процедур преобразуются следующим образом. Пусть вызывается процедура с *N* формальными аргументами. Обозначим имена этих аргументов следующим образом:

  аргумент1, *аргумент2*, ... *аргументN*.

Реальные аргументы обозначим как

выражение1, *выражение2*, ..., *выражениеN*

соответственно. Тогда оператор вызова процедуры заменяется на:

- *N* операторов квази-присваивания:

  `data.`<*имя процедуры*>_<*аргументM*> `@=` <*выражениеM*>`;`

  где *M* пробегает значения от 1 до *N*;

- вызов процедуры без аргументов.
- *N* операторов восстановления значений сохраненных квази-присваиваниями:

  `data.`<*имяM*> `?=;`

  где M пробегает значения от 1 до *N* в обратном порядке.

Если же вызов является непосредственный рекурсией, то шаг 9 еще более усложняется:

1. Вызовы процедур преобразуются следующим образом. Пусть вызывается процедура с *N* формальными аргументами. Обозначим имена этих аргументов следующим образом:

   аргумент1, *аргумент2*, ... *аргументN*.

Реальные аргументы обозначим как

выражение1, *выражение2*, ..., *выражениеN*

соответственно. Тогда оператор вызова процедуры заменяется на:

- *N* операторов присваивания вида

`data.`*<имяM>*`_ = `*<выражениеM>*`;`

где *M* пробегает значения от 1 до *N*.

- *N* операторов квази-присваивания, копирующие значения реальных аргументов:

`data.`*<имяM>* `@= data.`*<имяM>*`_;`

где *M* пробегает значения от 1 до *N*.

- Вызов процедуры без аргументов.
- *N* операторов восстановления значений сохраненных квази-присваиваниями:

`data.`*<имяM>* `?=;`

где *M* пробегает значения от 1 до *N* в обратном порядке.

Указанные модификации позволяют формально выделять модель и преобразовывать программы использующие рекурсию.

3.3.3. Пример выделения модели и модификации программы

Рассмотрим изложенные модификации на примере программы, вычисляющей факториалы первых *n* натуральных чисел:

```
/** Количество чисел, для которых требуется вычислить
    факториал. */
int n;

/** Основная процедура. */
```

```
void calc() {
  for (int i = 1; i <= n; i++) {
    // Вывод значения факториала
    System.out.println(fact(i));
  }
}

/** Вычисляет факториал числа a. */
int fact(int a) {
  if (a == 0) {
    return 1;
  } else {
    return a * fact(a - 1);
  }
}
```

После выполнения шагов 1-8 имеем (как и в разделе 3.2.3 используется @-нотация):

```
/** Модель данных. */
class Model {
  int n; // Глобальная переменная
  int calc_i; // Локальная переменная процедуры calc
  int fact_; // Возвращаемое значение процедуры calc
  // Переменная для формального параметра a процедуры
       fact
  int fact_a, fact_a_;
}
Model d;

void calc() {
  for (#i = 1; #i <= @n; #i++) {
    fact(#i);
    System.out.println(@fact_);
  }
}

int fact(int a) {
  if (#a == 0) {
    @fact_ = 1;
  } else {
    fact(#a - 1);
    @fact_ = #a * @fact_;
  }
}
```

После выполнения остальных шагов процедуры изменяется описание и вызовы процедуры `fact`:

```
void calc() {
  for (#i = 1; #i <= @n; #i++) {
    @fact_a @= #i;
    fact();
    @fact_a ?=;
```

```
    System.out.println(@fact_);
  }
}

void fact() {
  if (#a == 0) {
    @fact_ = 1;
  } else {
    #a_ = #a - 1;
    #a @= #a_;
    fact();
    #a ?=;
    @fact_ = #a * @fact_;
  }
}
```

Заметим, что вызов процедуры `fact` из `calc` не является прямой рекурсией, а вызова `fact` из себя самой ей является. По этому, код для первого из этих вызовов не использует переменную модели `#a_`.

3.3.4. Обращение правил именования

При применении указанных правил именования переменных по имени переменной модели можно легко узнать исходное имя переменной, и в какой процедуре она была объявлена. Соответствующие правила приведены в табл. 3.

Табл. 3. Обращение правил именования

Имя переменной модели	Исходная переменная
<*имя*>	Глобальная переменная с именем *имя*
<*имя1*>_<*имя2*>	Локальная переменная или формальный аргумент с именем *имя2*, объявленная в процедуре *имя1*
<*имя1*>_<*имя2*>_	Формальный аргумент с именем *имя1*, объявленный в процедуре *имя2*
<*имя*>_	Возвращаемое значение процедуры *имя*

3.4. Упрощение структуры программы

В данном разделе формализуется третий шаг метода построения логики визуализатора алгоритмов. Описываемый метод может быть применен как к программе полученной после выделения модели данных, так и к программа, из которых модель данных не выделялось. Целью третьего шага является максимальное сокращение количества типов используемых конструкций в программе.

В языках программирования *Java*, *C* и *C++* определены следующие виды операторов:

1. выражение (`expression`);
2. составной (`{ ... }`)
3. укороченный оператор ветвления (`if-then`);
4. полный оператор ветвления (`if-then-else`);
5. цикл с предусловием (`while`);
6. пустой (`;`);
7. оператор выбора (`switch`);
8. цикл с постусловием (`do`);
9. цикл повторения (`for`);
10. продолжения цикла (`continue`);
11. выхода из цикла (`break`);
12. выхода из процедуры (`return`);

Требуется преобразовать программу так, что бы она использовала только операторы первых пяти видов.

Проще всего исключить из программы пустые операторы. Для этого необходимо просто удалить их.

Исключение других операторов более сложно и рассматривается в разделах 3.4.1-3.4.6. В заключительном разделе 3.4.7 описывается порядок, в котором необходимо исключать операторы из программы.

3.4.1. Оператор цикла с постусловием

Оператор цикла с постусловием описывается следующей грамматикой (здесь и далее и грамматики записываются в расширенной форме Бэкуса-Наура):

ЦиклСПостусловием ::= `do` *Оператор* `while ( Выражение ) ;`

Его можно преобразовать в цикл с предусловием следующим образом. Введем временную (дополнительную) переменную типа `boolean`,

инициализировав ее значением `true`. После чего цикл можно записать следующим образом:

```
boolean ВременнаяПеременная = true ;
while ( ВременнаяПеременная || Выражение ) {
    ВременнаяПеременная = false ;
    Оператор
}
```

При этом для различных циклов должны использоваться различные временные переменные. Таким образом, имя временной переменной необходимо генерировать каждый раз заново.

3.4.2. Оператор цикла повторения

Оператор цикла с постусловием описывается следующей грамматикой:

ЦиклПовторения ::= `for` (*Инициализация*? `;` *Выражение*? `;` *Обновление*?) *Оператор*
Инициализация ::= *СписокВыражений*
ОпределениеПеременной
Обновление ::= *СписокВыражений*
СписокВыражений ::= *Выражение*
СписокВыражений `,` *Выражение*

Его так же можно преобразовать в цикл с предусловием. Это делается следующим образом:

```
Инициализация
while ( Выражение ) {
    Оператор
    Обновление
}
```

При этом в списках выражений запятые, разделяющие элементы списка заменяются точками с запятой.

3.4.3. Оператор продолжения цикла

Рассмотрим цикл, для которого действует оператор продолжения цикла. Для исключения оператора продолжения цикла потребуется временная

переменная (флаг продолжения) типа `boolean` (для каждого цикла своя), объявленная до заголовка цикла и инициализированная константой `false`. Будем преобразовывать оператор снизу вверх (от самого вложенного блока, к самому внешнему).

В общем виде он может быть записан следующим образом:

ЗаголовокЦикла { … { *Пролог БСОПЦ Эпилог* } … }

где *БСОПЦ* означает *блок, содержащий оператор продолжения цикла*, а вложенные фигурные скобки обозначают вложенные составные операторы.

Оператор продолжения цикла заменим на присваивание флагу продолжения выхода значения `true`.

Рассмотрим составной оператор, содержащий *БСОПЦ* (при этом отдельный оператор можно рассматривать как блок, содержащий один оператор). Тогда указанный фрагмент преобразуется следующим образом:

`boolean` *ВременнаяПеременная* `= false ;`

ЗаголовокЦикла { … {

Пролог

БСОПЦ

`if ( !` *ВременнаяПеременная* `) {` *Эпилог* `}`

} … }

Если *Эпилог* не содержит операторов, то соответствующий ему условный оператор можно не вводить.

Преобразование продолжается, пока не будет достигнут оператор цикла, для которого осуществляется продолжение.

Рассмотрим пример (часть процедуры рекурсивного поиска):

```
while (true) {
  k++;
  int j = f(k);
  if (j < 0) break;
  if (c[j]) continue;
  b[i] = a[j];
  c[j] = true;
  f2(i + 1);
  c[j] = false;
}
```

В соответствии с изложенными правилами введем временную переменную `tempVar`, тогда строка

```
if (c[j]) continue;
```

заменяется на

```
if (c[j]) continueFlag tempVar = true;
```

В пролог входят строки 1-4, а в эпилог — строки 6-9. Полностью преобразованный фрагмент выглядит следующим образом:

```
boolean continueFlag; // Значение false по умолчанию
while (true) {
  k++;
  int j = f(k);
  if (j < 0) break;
  if (c[j]) continueFlag = true;
  if (!continueFlag) {
    b[i] = a[j];
    c[j] = true;
    f2(i + 1);
    c[j] = false;
  }
}
```

Заметим, что если в теле одного цикла есть несколько операторов продолжения цикла, то для них используется один флаг выхода.

3.4.4. Оператор выхода из цикла

Оператор выхода из цикла преобразуется аналогично оператору продолжения цикла, только в место флага продолжения заводится флаг выхода, после чего к заголовку цикла добавляется условие вида

`!` *ФлагВыхода*

ответственное за выход из цикла, после выполнения оператора выхода из цикла.

Например, фрагмент из предыдущего раздела преобразуется к следующему виду:

```
boolean breakFlag = false;
while (true && !tempVar) {
  k++;
  int j = f(k);
  if (j < 0) breakFlag = true;
  if (!tempVar) {
    if (c[j]) continue;
    b[i] = a[j];
```

```
      c[j] = true;
      f2(i + 1);
      c[j] = false;
    }
  }
```

Где `breakFlag` — флаг выхода.

3.4.5. Оператор возврата из процедуры

Если процедура не имеет возвращаемого значение, то оператор выхода из процедуры преобразуется аналогично оператору выхода из цикла, при этом флаг выхода заводится в самом внешнем блоке процедуры.

Если процедура имеет возвращаемое значение, то требуется добавить еще одну переменную, имеющую тот же тип, что и возвращаемое значение (результат).

Оператор выхода из процедуры вида

`return` *выражение* `;`

преобразуется в два оператора, объединенных в блок:

```
{
    результат = выражение ;
    ФлагВыхода = true ;
}
```

Приведем пример (объединение двух множеств в системе не пересекающихся множеств):

```
Node join(Node n1, n2) {
  n1 = get(n1);
  n2 = get(n2);
  if (n1 = n2) return n1;
  if (n1.rank == n2.rank) {
    n1.rank++;
  }
  if (n1.rank > n2.rank) {
    return n2.parent = n1;
  } else {
    return n1.parent = n2;
  }
}
```

В этой процедуре используется три оператора возврата (в стоках четыре, девять и одиннадцать). После преобразования процедура выглядит следующим образом:

```
Node join(Node n1, n2) {
  Node result;
  boolean returnFlag;
  n1 = get(n1);
  n2 = get(n2);
  if (n1 = n2) return {result = n1; returnFlag = true;}
  if (!returnFlag) {
    if (n1.rank == n2.rank) {
      n1.rank++;
    }
    if (n1.rank > n2.rank) {
      { result = n2.parent = n1; returnFlag = true;}
    } else {
      { result = n1.parent = n2; returnFlag = true;}
    }
  }
}
```

Заметим, что при преобразовании был добавлен только один оператор ветвления (в седьмой строке), так как эпилоги для операторов возврата в девятой и одиннадцатой строках пусты.

3.4.6. Оператор выбора

В соответствии со спецификацией языков *Java* [37], *C* и *C++* оператор выбора может иметь очень сложную семантику. Мы рассмотрим несколько упрощенный оператор выбора, в котором не используется эффект “проваливания” через метки (когда выполняются блоки кода не оканчивающиеся инструкцией `break`). При этом, разрешим помечать блок кода несколькими метками. Таким образом, отсекаются некоторые операторы выбора, например, такой как:

```
switch (n%4) {
  case 0: i++;
  case 1: i++;
  case 2: i++;
}
```

При использовании указанных ограничений, оператор выбора описывается следующей грамматикой:

SwitchStatement ::= `switch` (*Выражение*) *ТелоОператора*

ТелоОператора ::= { *Блок* Метка** }

Блок ::= *Метка+ Оператор* `break ;`

Метка ::= `case` *КонстантноеВыражение* `:`

`default:`

Заметим, что в соответствии с семантикой оператора метки (в том числе `default`) не повторяются.

Что бы не вычислять выражение несколько раз, требуется завести дополнительную переменную (*ЗначениеВыражения*), тип которой совпадает с типом *Выражения*.

Пусть до преобразования оператор выбора имеет следующую структуру:

```
switch ( Выражение ) {
    Метки1 Блок1 break ;
    Метки2 Блок2 break ;
    ...
    МеткиN БлокN break ; // Среди меток есть dafault
}
```

Тогда преобразованный оператор имеет структуру:

```
Тип ЗначениеВыражения = Выражение ;
if (ЗначениеВыражения == КонстантноеВыражение1 ) {
    Блок1
} else if (ЗначениеВыражения == КонстантноеВыражение2 )
{
    Блок2
…
} else {
    БлокN
}
```

Где *КонстантноеВыражения* — выражения из соответствующих меток.

На пример, оператор выбора

```
boolean value;
switch (key) {
```

```
  case 'Y': case 'y':
    value = true;
  case 'N': case 'n':
    value = false;
  default:
    // Действия при неправильном вводе
}
```

Преобразуется в:

```
boolean value;
char temp = key;
if (temp == 'Y' || temp == 'y') {
  value = true;
} else if (temp == 'N' || temp == 'n') {
  value = false;
} else {
  // Действия при неправильном вводе
}
```

3.4.7. Порядок преобразования операторов

Не все из приведенных преобразований операторов независимы. Некоторые преобразования необходимо выполнять в строгой последовательности. Такие зависимости изображены на рис. 2.

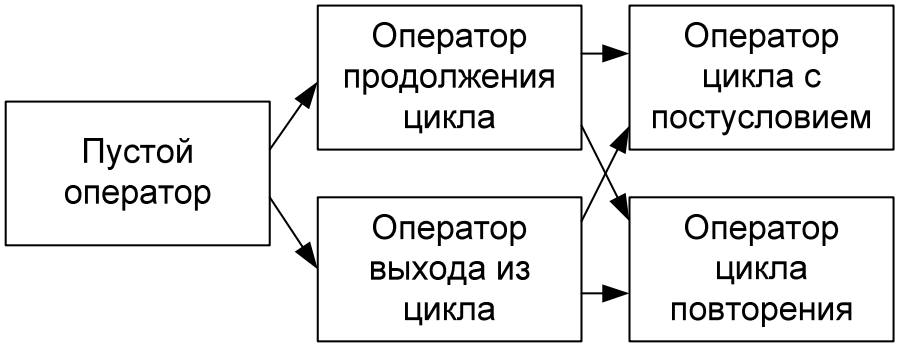


Рис. 2. Зависимости преобразований

3.5. Выводы

В результате применения преобразований изложенных в данной главе визуализируемая программа будет преобразована к виду, удобному для преобразования в систему взаимодействующих конечных автоматов. При этом, структура программы существенно упрощается, как с точки зрения использования переменных (так как все они вынесены в модель данных), так и с точки зрения используемых операторов.

Заметим, что при таком преобразовании длина программы обычно увеличивается, но это не является критичным при автоматизированной обработке.

После преобразования программа характеризуется следующими свойствами:

1. программа использует только переменные модели данных;
2. каждый оператор (последовательность операторов) имеет один вход и один выход.
3. используются только следующие операторы:
   - оператор присваивания;
   - последовательность операторов (составной оператор);
   - оператор ветвления (`if`);
   - цикл с предусловием (`while`);
   - вызов процедуры.

В последующей части данной работы будут рассматриваться только такие программы, что позволяет описывать преобразования для сравнительно небольшого количества конструкций.

Изложенный метод позволяет выделять модель данных и упрощать структуру любой процедурной программы, что может быть полезно не только для построения логики визуализатора, но и в других задачах, в которых требуется автоматизированное преобразование программ.

ГЛАВА 4. ПРЕОБРАЗОВАНИЕ ПРОГРАММЫ В СИСТЕМУ ВЗАИМОСВЯЗАННЫХ АВТОМАТОВ

4.1. Преобразование процедуры в автомат

В результате выполнения третьего шага описываемого метода процедура содержит только следующие операторы: оператор присваивания, последовательность операторов, полный и укороченный операторы ветвления, цикл с предусловием, вызов процедуры.

Построение автомата по процедуре осуществляется последовательно. При этом для каждого оператора строится фрагмент автомата с одним входом и одним выходом, которые связывают его с другими фрагментами. На приведенных ниже рисунках входы и выходы фрагмента автомата будем представлять в виде входящей и исходящей стрелок, у которых начало (для входа) и конец (для выхода) не связаны ни с одним состоянием. Из фрагментов автомата, соответствующих отдельным операторам, строятся блоки, соответствующие последовательности операторов.

4.1.1. Оператор присваивания

Оператор присваивания преобразуем в состояние автомата, в котором в качестве действий, выполняются присваивания. Например, оператор

```
d.m = max(d.m, d.a[d.i]);
```

преобразуется в состояние, как изображено на рис. 3.

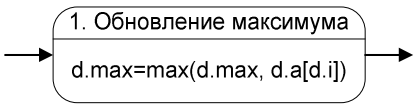


Рис. 3. Оператор присваивания

Здесь и далее над чертой, разделяющей на две части изображения состояния (вершины графа переходов), записаны номер состояния и его название, а под ней — действия, выполняемыеы в состоянии.

4.1.2. Последовательность операторов

Преобразование последовательности из двух операторов осуществляется “связыванием” стрелки, выходящей из фрагмента автомата, соответствующего первому оператору, и стрелки, входящей во фрагмент автомата соответствующего второму оператору. Преобразование последовательности из большего числа операторов осуществляется аналогично.

Например, последовательность операторов

```
m = max(m, a[i]);
m = min(m, b[i]);
```

преобразовывается во фрагмент автомата, изображенный на рис. 4.

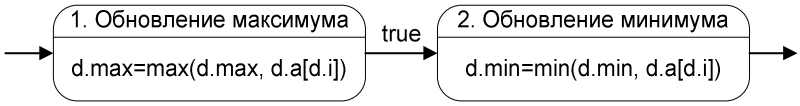


Рис. 4. Последовательность состояний

Обратим внимание, что дуга между вершинами помечена условием перехода `true`, что обозначает безусловный переход. В дальнейшем для экономии места и улучшения читаемости, везде где это целесообразно, условие `true` будем опускать.

Так как во фрагмент автомата, представляющей оператор или последовательность операторов, входит и выходит только одна стрелка, то такой фрагмент можно обозначить, как показано на рис. 5.

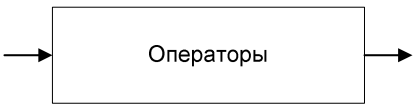


Рис. 5. Блок операторов

4.1.3. Оператор вызова процедуры

Оператор вызова процедуры преобразуем в состояние, в котором осуществляется вызов автомата, соответствующего ей. Например, вызов процедуры

```
findMin()
```

преобразуем, как показано на рис. 6.

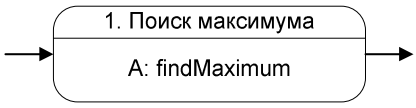


Рис. 6. Вызов процедуры

Здесь префикс "`A:`" обозначает вызов автомата. Более подробно вызовы процедур и автоматов будут рассмотрены в разделе 4.5.

4.1.4. Укороченный оператор ветвления

Укороченный оператор ветвления

```
if (expr) {
  Операторы
}
```

будем преобразовывать во фрагмент автомата, изображенный на рис. 7.

Рис. 7. Укороченный оператор ветвления

Переход происходит по той ссылке, условие для которой истинно. Здесь и далее `expr` обозначает условие, а `¬expr` — его отрицание.

4.1.5. Полный оператор ветвления

Полный оператор ветвления

```
if (expr) {
  Операторы 1
} else {
  Операторы 2
}
```

преобразуем, как показано на рис. 8.

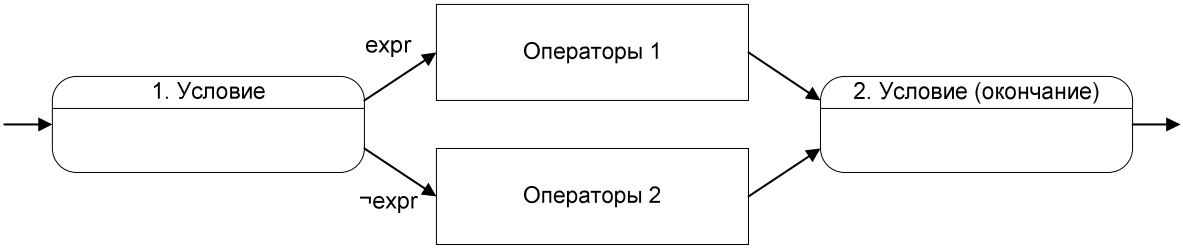


Рис. 8. Полный оператор ветвления

Обоснование введения состояния, помеченного как “Условие (окончание)”, будет приведено в разделе 4.3.2.

4.1.6. Цикл с предусловием

Цикл с предусловием

```
while (expr) {
  Операторы
}
```

преобразуем, как показано на рис. 9.

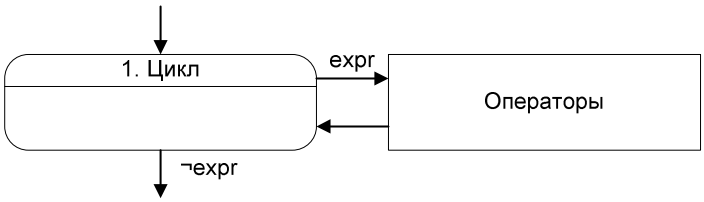


Рис. 9. Цикл с предусловием

4.1.7. Завершение построения автомата

Для завершения построения автомата к фрагменту, соответствующему телу процедуры, добавим начальное и конечное состояния. Начальное состояние связывается со свободной стрелкой, входящей во фрагмент автомата, а конечное — со свободной стрелкой, выходящей из него. Таким образом, получаем прямой автомат, соответствующий процедуре.

Для пошагового выполнения процедуры можно использовать управляемый тактовый генератор. При этом автомат совершает переход только при импульсе тактового генератора. В частности, можно применить тактовый генератор, производящий импульс только при нажатии пользователем кнопки “Переход к следующему шагу”.

4.2. Пример преобразования процедуры в автомат

Несмотря на то, что визуализируемые алгоритмы являются процедурными, в целом визуализатор удобно реализовать как систему взаимосвязанных классов. При этом логика работы визуализатора реализуется по принципу: один автомат — один класс.

В этом и последующих примерах будем использовать язык *Java*.

Проиллюстрируем применение метода, изложенного в предыдущем разделе, на примере построения прямого автомата для процедуры поиска максимума в массиве натуральных чисел.

На первом шаге метода напишем следующую программу:

```
int max = 0;
for (int i = 0; i < a.length; i++) {
  if (max < a[i]) {
    max = a[i];
  }
}
```

В соответствии со вторым шагом предложенного метода, вынесем используемые переменные в модель данных — класс `Data`:

```
public final static class Data {
  public int max;
  public int i;
  public int a[];
}
```

Считая, что экземпляр класса `Data` доступен как переменная `d`, получим:

```
d.max = 0;
for (d.i = 0; d.i < d.a.length; d.i++) {
  if (d.max < d.a[d.i]) {
    d.max = d.a[d.i];
  }
}
```

В данной процедуре использован цикл `for`, который в соответствии с третьим шагом предлагаемого метода необходимого преобразовать в цикл с предусловием `while`:

```
d.max = 0;
d.i = 0;
while (d.i < d.a.length) {
  if (d.max < d.a[d.i]) {
    d.max = d.a[d.i];
  }
  d.i++;
}
```

В этой процедуре части выделенные курсивом соответствуют частям заголовка исходного цикла.

Преобразуем полученную процедуру в автомат. Два первых оператора преобразуются во фрагмент автомата, изображенный на рис. 10.

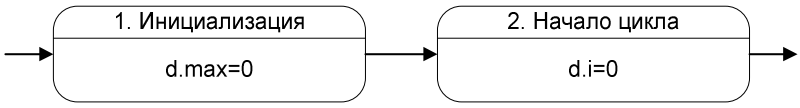


Рис. 10. Начальная инициализация

Прежде чем преобразовать цикл `while` во фрагмент автомата, необходимо построить фрагмент, соответствующий телу цикла. Для этого преобразуем оператор ветвления `if` (в данном случае — укороченный). Ему соответствует фрагмент автомата, изображенный на рис. 11.

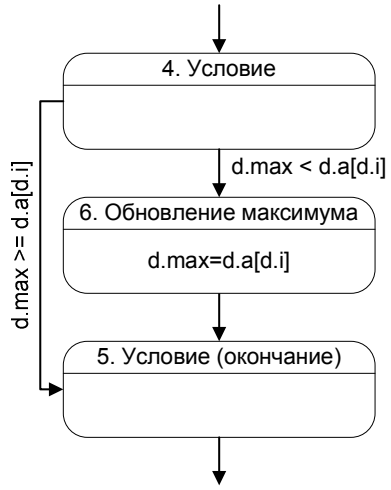


Рис. 11. Оператор ветвления

Присоединив к полученному фрагменту (состояния 4 – 6 на рис. 12) состояние 7, соответствующее оператору `d.i++`, получим преобразованное тело цикла (состояния 4 – 7). Заканчивая преобразование цикла `while`, добавим состояние 3.

Присоединяя к фрагменту, соответствующему циклу `while` (рис. 9), состояния 1 и 2, а также, вводя начальное и конечное состояния (0 и 8), получим автомат, изображенный на рис. 12.

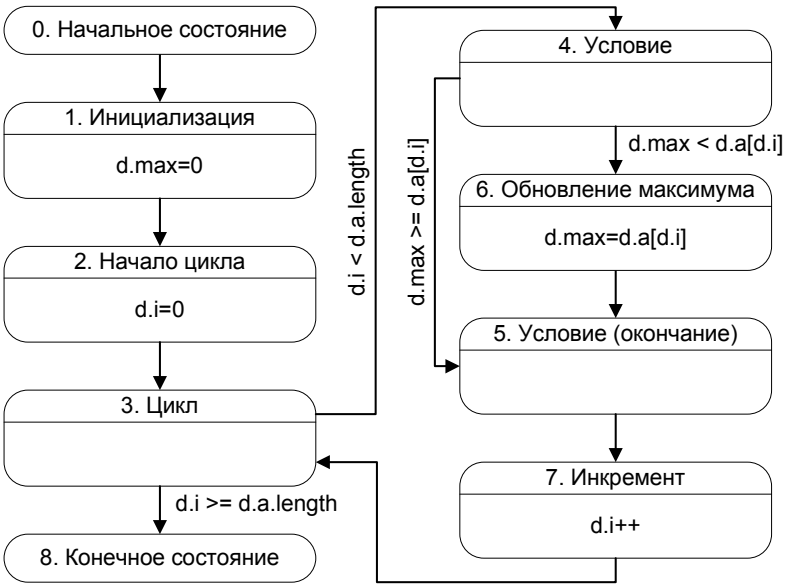


Рис. 12. Автомат поиска максимума

Полученный автомат Мура эквивалентен исходной процедуре по построению.

4.3. Построение обратного автомата

В предыдущих разделах было показано, как по процедуре получить эквивалентный автомат. Основная функция прямого автомата — перемещение по алгоритму вперед.

Существует несколько методов обеспечения возможности передвижения по алгоритму в обратную сторону.

1. *Метод пошагового сохранения*. Для каждого шага сохраняется значение всех переменных модели данных и состояние прямого автомата. При совершении шага назад значения, сохраненные на предыдущих шагах, восстанавливаются. Данный метод требует большей памяти по сравнению с методами 2 и 3.
2. *Метод пересчета*. При движении вперед запоминается количество шагов, сделанное автоматом. Для совершения шага назад прямой автомат перезапускается сначала и останавливается, сделав на один шаг меньше. Этот метод требует существенных затрат времени на повторные вычисления.

3. *Метод обращения*. Для передвижения в обратную сторону строится “обратный” автомат. Данный метод не требует ни большой памяти, ни дополнительных затрат времени на повторные вычисления, и будет рассмотрен ниже.

В предлагаемом подходе используется *метод обращения*, как наиболее эффективный. Заметим, что хотя этот метод внешне напоминает обращение программ, описанное в [1], однако эти методы различаются по сути.

Обратный автомат имеет одноименные состояния с прямым автоматом. Все переходы прямого автомата сохраняются, но направляются в противоположную сторону. Пометки переходов модифицируются, как будет изложено ниже. Построенную таким образом пару автоматов можно рассматривать, как обобщенный автомат с двумя функциями переходов и двумя наборами действий. При этом одна функция переходов и набор действий соответствуют движению по алгоритму вперед, а вторая функция и второй набор действий — движению назад.

В дальнейшем движение по алгоритму вперед будем называть *прямым проходом*, а движение назад — *обратным проходом*.

При *прямом проходе* остановка автомата для визуализации осуществляется перед каждым переходом (сразу после выполнения действия в состоянии). Соответственно, при *обратном проходе* остановку необходимо осуществлять непосредственно перед каждым состоянием. При использовании этого соглашения текущее состояние прямого и обратного автоматов можно описывать одним номером состояния.

Перейдем к изложению метода построения обратного автомата.

Последовательность операторов обращается путем выполнения обращений операторов в противоположном порядке. Таким образом, построение обратного автомата сводится к обращению отдельных операторов.

Обращение оператора можно произвести двумя способами.

4. *Непосредственный способ.* Состояние автомата и значения переменных модели данных вычисляются непосредственно по их значениям на последующем шаге.
5. *Косвенный способ.* Состояние автомата и значения переменных модели данных восстанавливаются по информации, запомненной при прямом проходе, как описано ниже.

Обращение *непосредственным способом* требует неформального подхода, но позволяет экономить память.

Обращение *косвенным способом* может быть произведено формально. При этом необходимо производить модификацию прямого автомата, с тем, чтобы он сохранял информацию, необходимую при *обратном проходе*.

В дальнейшем, вместо выражений *обращение непосредственным способом* и *обращение косвенным способом*, будем применять выражения *непосредственное обращение* и *косвенное обращение* соответственно.

4.3.1. Обращение оператора присваивания

В различных ситуациях выгодно использовать тот или иной метод. Проиллюстрируем это на примере обращения оператора присваивания.

Косвенное обращение оператора присваивания осуществляется помещением замещаемого значения переменной в стек при *прямом проходе* и извлечением его из стека при *обратном проходе*. Стек подходит для сохранения замещаемых значений, так как они используются в обратном порядке по сравнению с порядком запоминания.

В дальнейшем этот стек будем называть *общим стеком*, так как он будет использован при обращении и других операторов. На рисунках *общий стек* будем обозначать как `stack`.

Для стеков будем использовать следующие операции:

6. `push(expr)` — поместить значение выражения `expr` на вершину стека;
7. `peek()` — прочитать значение на вершине стека;
8. `pop()` — прочитать значение на вершине стека и удалить его.

Таким образом, при *прямом проходе* прямой автомат помещает замещаемое значение в *общий стек*, а при *обратном проходе* сохраненное значение извлекается из *общего стека*.

Приведем пример косвенного обращения оператора присваивания для оператора, осуществляющего обновление максимума (из процедуры поиска максимума в разделе 4.2):

```
d.max = d.a[d.i];
```

Этот оператор обращается, как показано на рис. 13.

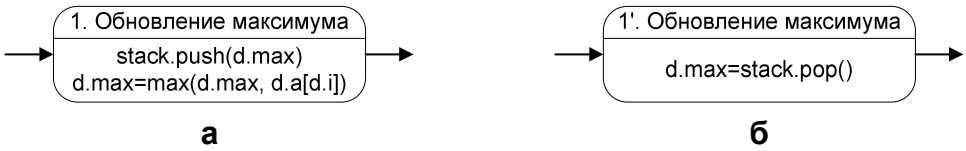


Рис. 13. Оператор присваивания (а) и его косвенное обращение (б)

Здесь и далее будем помечать состояния обращенного автомата добавлением штриха к его номеру.

Отметим, что обращение данного оператора *непосредственным способом* затруднено тем, что для вычисления предыдущего значения максимума необходимо знать все предыдущие значения в массиве, а если бы процедура обнуляла просмотренный элемент массива, то это было бы вообще невозможно.

В тоже время *непосредственное обращение* оператора

```
d.i++;
```

осуществляющего переход к следующему элементу (из процедуры поиска максимума в разделе 4.2), осуществляется оператором

```
d.i--;
```

Таким образом, при прямом проходе единица прибавляется, а при обратном проходе — вычитается. Обращение данного оператора выглядит, как показано на рис. 14.

Рис. 14. Оператор присваивания (а) и его непосредственное обращение (б)

4.3.2. Обращения операторов ветвления

При обращении оператора ветвления возникает следующая проблема: решение о выборе ветви для обратного прохода невозможно принять в состоянии, в котором принимается решение при прямом проходе, так как в этот момент обращение ветви уже завершено.

Данная проблема решается введением состояния “Условие (окончание)” во фрагменты автоматов, соответствующих оператору ветвления. Здесь и далее будем называть состояние, помеченное как “Условие”, *открывающим* состоянием оператора ветвления, а состояние, помеченное как “Условие (окончание)” — *закрывающим*. Таким образом, решение о выборе ветви при обратном проходе принимается в *закрывающем состоянии*. Так же, как и при обращении оператора присваивания, существует два способа произвести этот выбор: *непосредственный* и *косвенный*.

При *непосредственном способе* после обращении операторов ветвления получаются фрагменты автоматов, приведенные на рис. 15 и рис. 16.

Рис. 15. Непосредственное обращение укороченного оператора ветвления

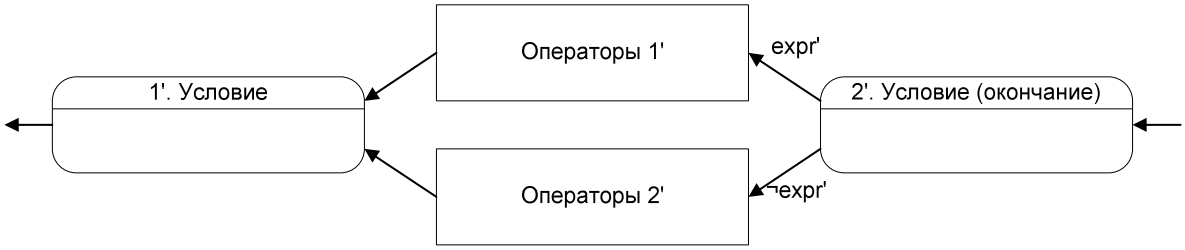


Рис. 16. Непосредственное обращение полного оператора ветвления

Здесь и далее под `expr'` подразумевается условие, позволяющее выбрать ветвь для обратного прохода.

При *косвенном способе* необходимо запоминать ветвь, выбранную при прямом проходе. Существует два варианта сохранения данной информации. В *первом варианте* обратный автомат является смешанным [2] (действия могут

выполняться не только в состояниях, но и на переходах). При этом прямой автомат также преобразуется в смешанный. *Второй вариант* требует применения двух стеков, но получаемые автоматы являются автоматами Мура.

Рассмотрим *первый вариант*. Будем запоминать информацию о выбранной ветви при переходе в состояние, соответствующее окончанию оператора ветвления. Таким образом, данную информацию можно будет использовать для принятия решения при обратном проходе.

Фрагменты прямого и обратного автоматов для *первого варианта косвенного обращения* оператора ветвления приведены на рис. 17 и рис. 18.

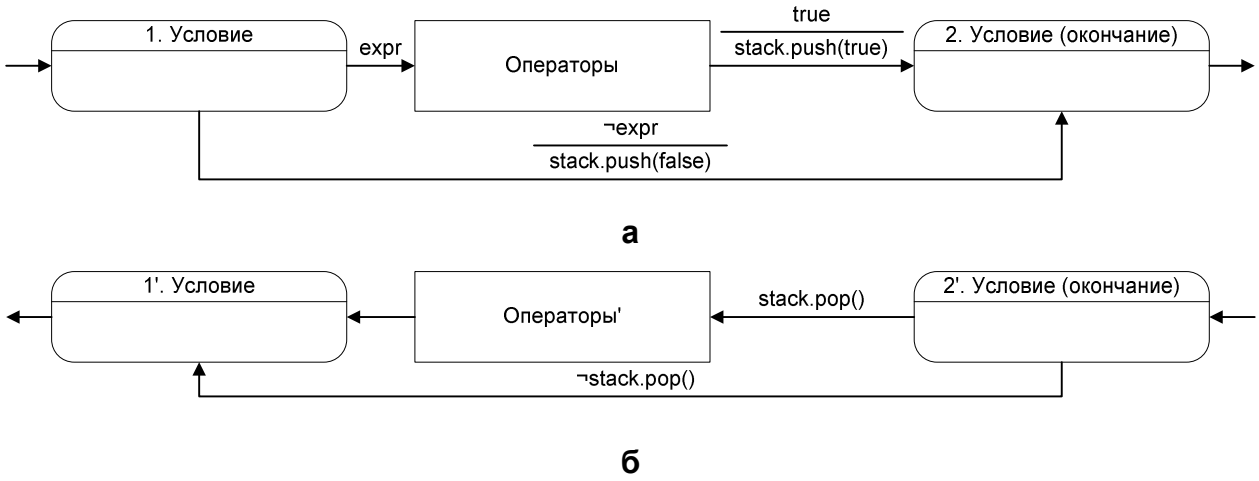


Рис. 17. Укороченный оператор ветвления (а) и первый вариант его косвенного обращения (б)

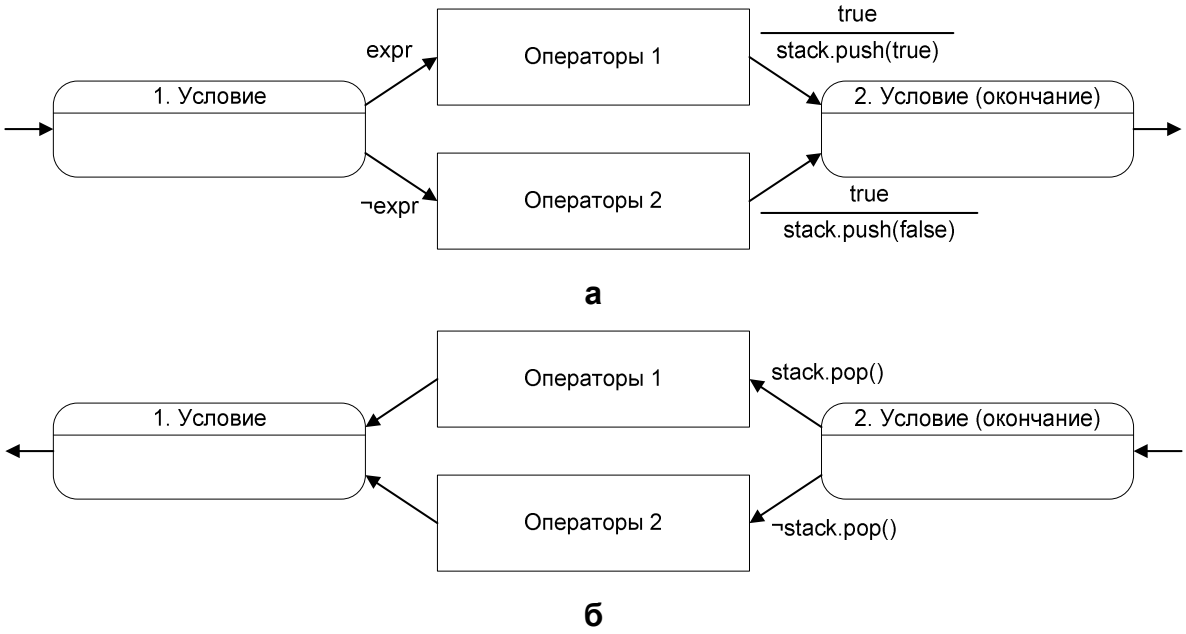


Рис. 18. Полный оператор ветвления (а) и первый вариант его косвенного обращения (б)

Здесь и далее на переходах под чертой указываются выполняемые действия.

Функция `stack.pop()` при работе удаляет значение из вершины стека. Если выполнять действие при проверке условия нежелательно, то применение функций `stack.pop()` и `¬stack.pop()` следует заменить на $\frac{\text{stack.peek()}}{\text{stack.pop()}}$ и $\frac{\neg\text{stack.peek()}}{\text{stack.pop()}}$ соответственно.

Таким образом, в *первом варианте косвенного обращения* операторов ветвления строятся фрагменты смешанного автомата.

Перейдем ко *второму варианту*. Информацию о ветви, выбранной при прямом проходе, нельзя запоминать в стеке непосредственно при ее выборе, так как тогда сохранять и использовать эту информацию необходимо в различных состояниях автомата (в *открывающем* и *закрывающем* соответственно), что недопустимо. Данную проблему можно обойти, используя *стек открытых ветвлений* — ветвлений, для которых пройдено *открывающее* и еще не пройдено *закрывающее* состояние. На рисунках *стек открытых ветвлений* будем обозначать как `open`.

При *прямом проходе* в *открывающем* состоянии оператора ветвления в *стек открытых ветвлений* помещается флаг, соответствующий выбранной ветви (`true`, если условие выполнено, и `false`, если не выполнено). В *закрывающем* состоянии оператора ветвления данное значение переносится из *стека открытых ветвлений* в *общий стек*. При *обратном проходе* в *закрывающем* состоянии оператора ветвления значение из вершины *общего стека* переносится в *стек открытых ветвлений*. Если перенесенное значение равно `true`, то для обратного прохода выбирается положительная ветвь оператора ветвления, а если `false` — отрицательная.

Фрагменты прямого и обратного автоматов для *второго варианта косвенного обращения* оператора ветвления приведены на рис. 19 и рис. 20.

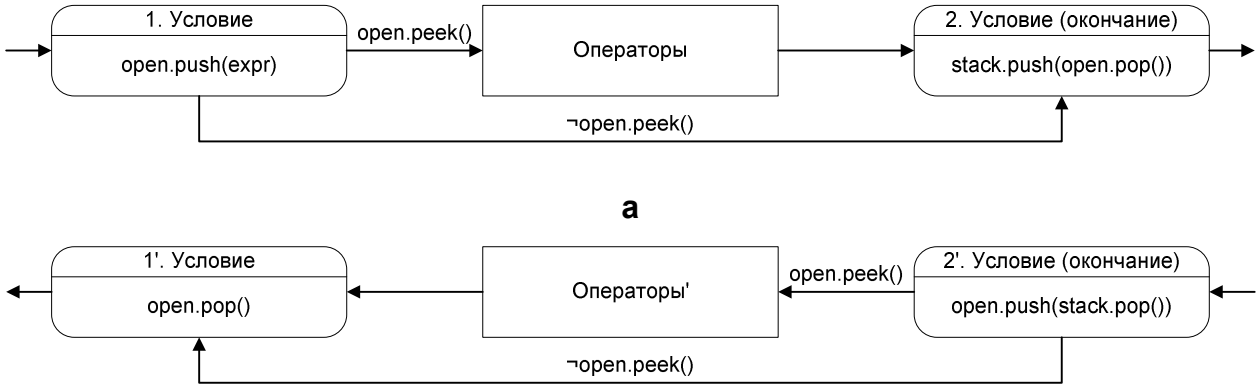


Рис. 19. Укороченный оператор ветвления (а) и второй вариант его косвенного обращения (б)

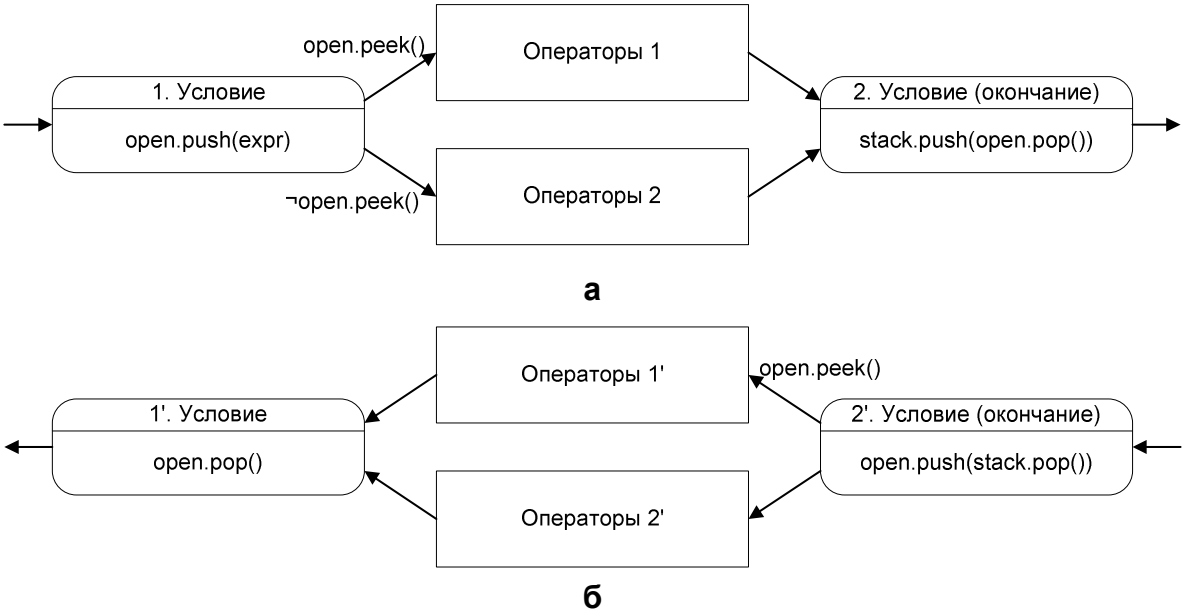


Рис. 20. Полный оператор ветвления (а) и второй вариант его косвенного обращения (б)

Таким образом, во *втором варианте косвенного обращения* операторов ветвления строятся фрагменты автоматов Мура.

4.3.3. Обращение оператора цикла с предусловием

Так же, как и для других операторов, обращение цикла с предусловием можно производить *непосредственным* и *косвенным способами*.

Обращение цикла с предусловием *непосредственным способом* приведено на рис. 21.

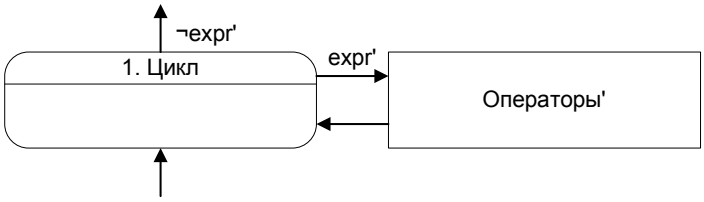


Рис. 21. Непосредственное обращение цикла с предусловием

Косвенное обращение цикла с предусловием также можно осуществить в двух вариантах: с образованием фрагментов смешанных автоматов и с образованием фрагментов автоматов Мура, но требующим двух стеков.

Рассмотрим *первый вариант косвенного обращения* цикла с предусловием. При *прямом проходе* будем сохранять в стеке информацию о

том, был ли переход в состояние цикла совершен извне (`false`) или из тела цикла (`true`). Соответственно, при *обратном проходе* данную информацию можно будет использовать для определения было ли тело цикла выполнено при *прямом проходе*. При этом определяется надо ли выполнить “обращенное” тело цикла еще раз или следует выйти из цикла.

Фрагменты прямого и обратного автоматов для *первого варианта косвенного обращения* цикла с предусловием приведены рис. 22.

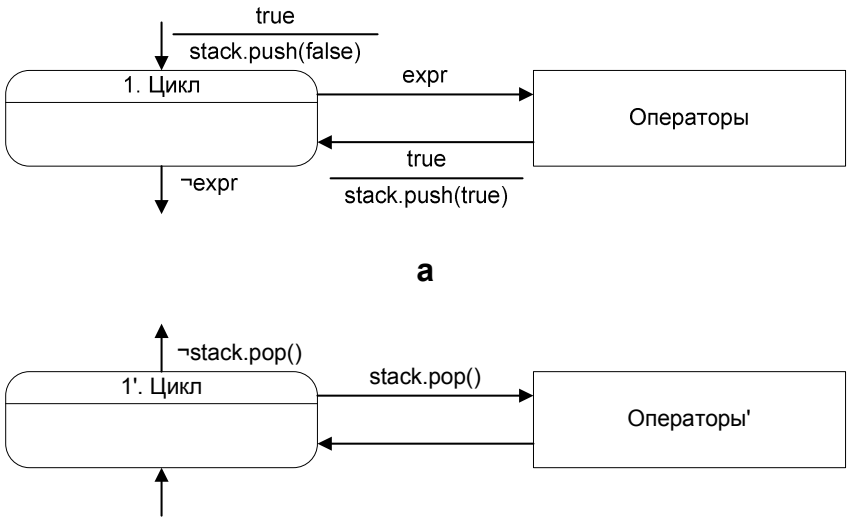


Рис. 22. Цикл с предусловием (а) и первый вариант его косвенного обращения (б)

Второй вариант косвенного обращения оператора цикла с предусловием требует введения двух дополнительных состояний: “Цикл (начало)” и “Цикл (окончание)”. В дальнейшем эти состояния будем назвать *открывающее* и *закрывающее* соответственно, а состояние “Цикл” будем называть *основным*.

Рассмотрим *прямой проход*. В начальном состоянии в *стек открытых ветвлений* помещается значение `false`. В *основном* состоянии значение из вершины *стека открытых ветвлений* переносится в *общий стек*, а в *стек открытых ветвлений* помещается значение `true`. В *заключительном* состоянии значение из вершины *стека открытых ветвлений* удаляется.

Перейдем к рассмотрению *обратного прохода*. В *конечном* состоянии в *стек открытых ветвлений* заносится значение `true`. В *основном* состоянии

значение из вершины *стека открытых ветвлений* удаляется. После этого значение из вершины *общего стека* переносится в *стек открытых ветвлений*. Затем для определения необходимости выполнения “обращения” тела цикла анализируется значение в вершине *стека открытых ветвлений*. Если это значение `true`, то осуществляется переход к “обращению” тела цикла, в противном случае осуществляется переход в *начальное* состояние. В *начальном* состоянии значение `false` удаляется из вершины *стека открытых ветвлений*.

Фрагменты прямого и обратного автоматов для *второго варианта косвенного обращения* цикла с предусловием приведены на рис. 23.

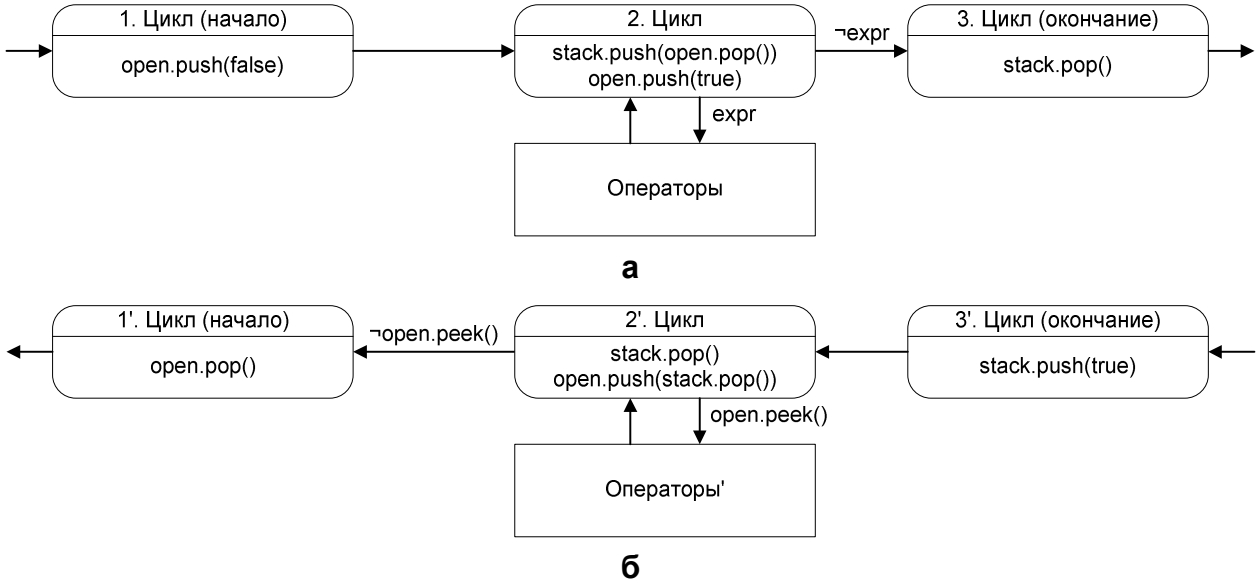


Рис. 23. Цикл с предусловием (а) и второй вариант его косвенного обращения (б)

4.3.4. Классификация вариантов построения обратного автомата

В подразделах 4.3.1 – 4.3.3 рассмотрены различные варианты построения обратного автомата. Обобщим сведения о предложенных вариантах в табл. 4.

Табл. 4. Варианты построения обратного автомата

Особенности обращения	Непосредственное обращение	Косвенное обращение		
		Оператор присваивания	Первый вариант	Второй вариант
1	2	3	4	5
Операторы	Все	Присваивание	Ветвления и цикл	
Формализм	Неформальный	Формальный	Формальный	Формальный
Тип автоматов	Мура	Мура	Смешанный	Мура
Количество стеков	0	1	1	2
Модификация прямого автомата	Не требуется	Требуется		

Построить обратный автомат по процедуре можно, как формальным, так и неформальным способом. Формальное построение может быть осуществлено в двух вариантах. В первом случае для обращения операторов используются столбцы 3 и 4 таблицы, а во втором — столбцы 3 и 5. Соответственно, в первом случае получается пара смешанных автоматов, использующих *общий стек*, а во втором варианте — пара автоматов Мура, использующих *общий стек* и *стек открытых ветвлений*.

4.4. Пример построения обратного автомата

В разделе 4.2 был построен прямой автомат для процедуры поиска максимума. Модифицируем его и построим обратный автомат, как изложено в разделе 4.3.

Оператор, осуществляющий обновление максимума, обратим *косвенным способом* (столбец 3 таблицы), так как его обращение *непосредственным способом* требует просмотра всех предыдущих значений массива.

По тем же причинам оператор ветвления обратим *косвенным способом*. Для обращения выберем вариант с образованием смешанного автомата (столбец 4 таблицы). Обращенный оператор ветвления приведен на рис. 24.

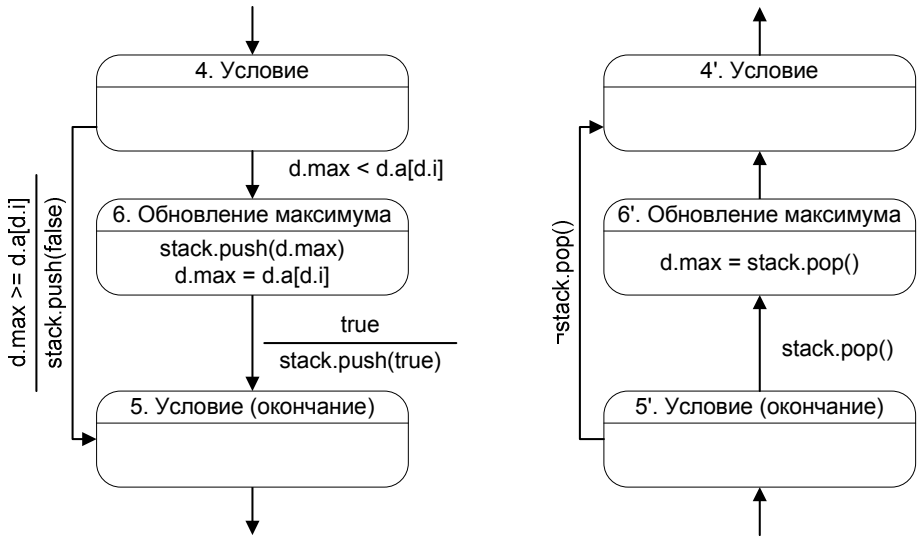


Рис. 24. Оператор ветвления (а) и его обращение (б)

Оператор, осуществляющий переход к следующему элементу, обратим *непосредственным способом* (столбец 2 таблицы). Также *непосредственным способом* обратим цикл (обращение условия продолжения цикла `d.i > 0`). Обращенный цикл соответствует состояниям 3 – 7 на рис. 25 и рис. 26.

Заметим, что при *обратном проходе* после прохождения цикла значения переменных `d.max` и `d.i` равны нулю. Воспользовавшись этим, обратим операторы, осуществляющие инициализацию, *непосредственным способом*. При этом в соответствующих состояниях обратного автомата действия выполняться не будут.

Завершим построение обратного автомата добавлением начального и конечного состояний. Полученная пара автоматов изображена на рис. 25 и рис. 26.

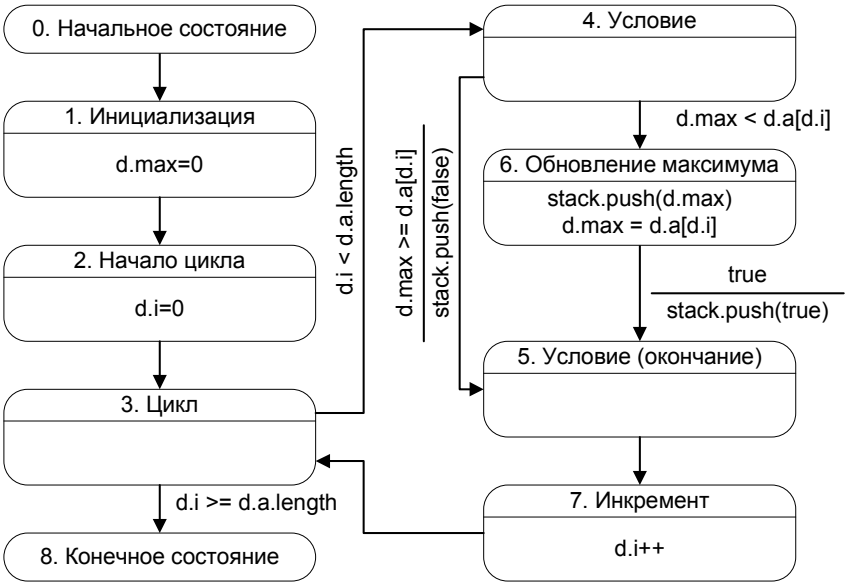


Рис. 25. Прямой автомат поиска максимума

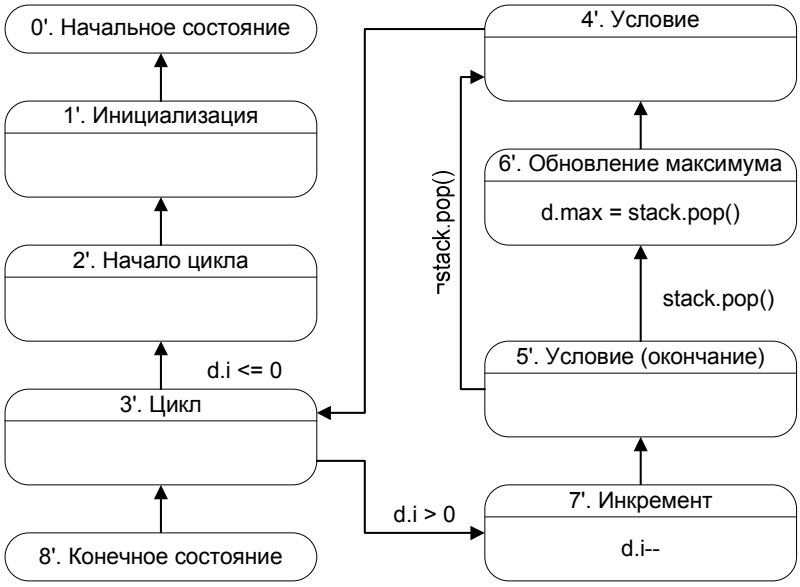


Рис. 26. Обратный автомат поиска максимума

Покажем, как произвести построение обратного автомата формально. Для этого будем обращать все операторы косвенным способом (столбцы 3 – 5 таблицы).

Пара автоматов, построенная при использовании столбцов 3 и 4 таблицы, приведена на рис. 27 и рис. 28.

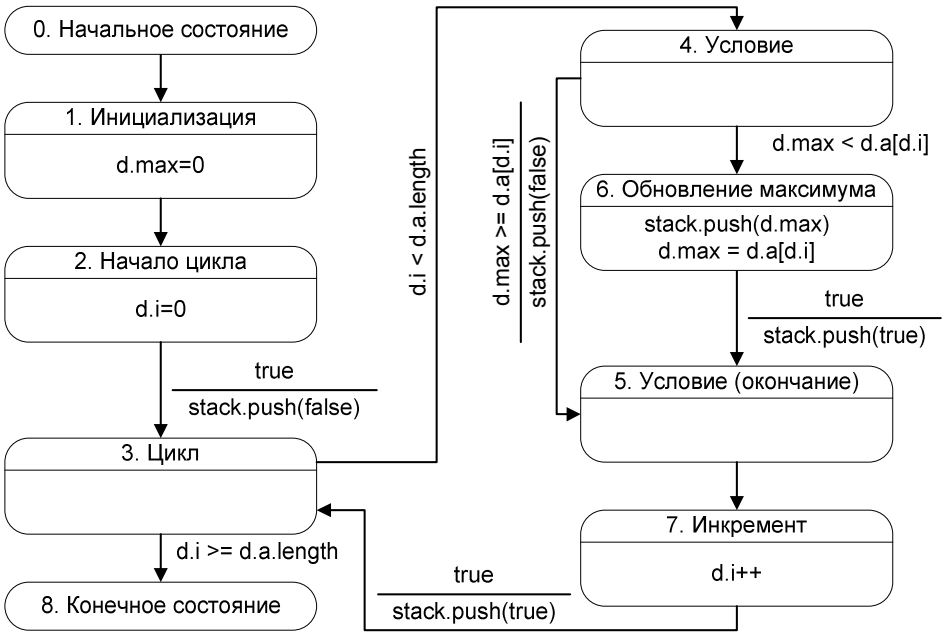


Рис. 27. Прямой автомат поиска максимума (второй вариант)

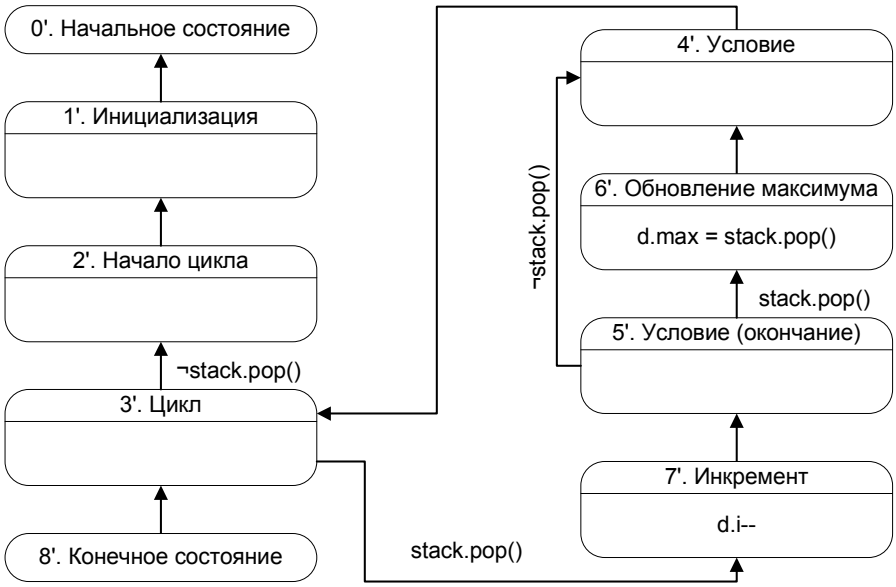


Рис. 28. Обратный автомат поиска максимума (второй вариант)

В заключение раздела отметим, что в рассмотренном примере при разных видах обращения получаются различные пары из автоматов, что является спецификой предлагаемого метода.

4.5. Процедуры и вызовы автоматов

Как было отмечено выше, если реализация визуализируемого алгоритма представляет собой набор процедур, то каждая из них преобразуется отдельно.

При этом полученные автоматы взаимодействуют между собой посредством *вызовов* друг друга.

Так как автоматы для разных процедур строились независимо, то для осуществления взаимодействия их необходимо модифицировать. При этом возможны два случая:

9. для итеративных (не рекурсивных) программ;
10. для рекурсивных программ.

Первый случай более прост, но имеет существенное ограничение на класс преобразуемых программ. Второй случай сложнее, но позволяет преобразовать как итеративные, так и рекурсивные программы.

4.5.1. Итеративные программы

Преобразование итеративной программы

Пронумеруем процедуры некоторым способом. Обозначим прямой и обратный автоматы для процедуры номер *i* через *Ai* и *Ai'* соответственно.

Введем несколько условий для связи между автоматами:

13. `state(Ai)==j` — автомат Ai находится в состоянии с номером j;
14. `isAtStart(Ai)` — автомат Ai находится в начальном состоянии;
15. `isAtEnd(Ai)` — автомат Ai находится в конечном состоянии.

Так как автоматы *Ai* и *Ai'* всегда находятся в одном и том же состоянии (разделе 4.3), то аналогичные условия для обратного автомата не вводятся.

Как было сказано выше, каждая пара автоматов соответствует одной процедуре, а каждому вызову этой процедуры (кроме главной) соответствует некоторое состояние. Составим *список вызовов* автомата *Ai*. Он состоит из пар (*Aj*, *k*), где *Aj* — автомат, который осуществляет вызов автомата *Ai*, а *k* — номер состояния, в котором осуществляется вызов. Заметим, что если автомат *Aj* содержит несколько вызовов автомата *Ai*, то ему соответствует несколько пар указанного вида. Для каждой пары из этого списка построим условие `state(Aj)==k`. Таким образом, всему *списку вызовов* будет соответствовать

дизъюнкция условий, построенных для каждой пары. Обозначим полученное условие как *R(Ai)*.

Модифицируем автоматы, соответствующие всем процедурам, кроме главной, следующим образом. Для прямого автомата *Ai*:

11. к переходу из *начального состояния* добавим условие *R(Ai)*;
12. введем безусловный переход из *конечного состояния* в *начальное*.

Для обратного автомата *Ai'*:

13. к переходу из *конечного состояния* добавим условие *R(Ai)*;
14. введем безусловный переход из *начального состояния* в *конечное*.

После этого преобразуем состояния, в которых осуществляется вызов автоматов. Пусть в состоянии *j* автомата *Ai* осуществляется вызов автомата *Ak* (рис. 29).

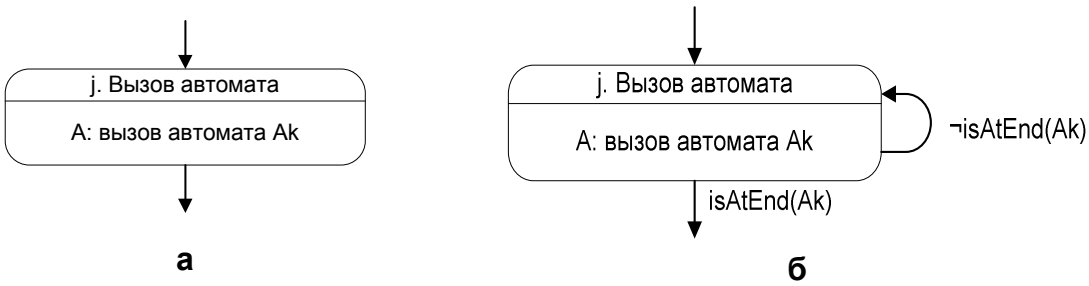


Рис. 29. Состояние прямого (а) и обратного (б) автоматов, из которых вызывается другой автомат

Преобразуем их, как показано на рис. 30.

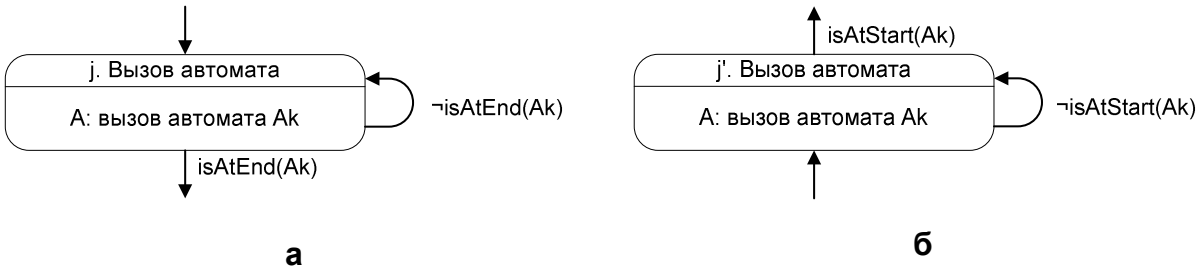


Рис. 30. Преобразованные состояния прямого (а) и обратного (б) автоматов, из которых вызывается другой автомат

Преобразовав все состояния, в которых осуществляются вызовы процедур изложенным выше образом, получим итоговый набор автоматов (по два автомата на процедуру).

Рассмотрим работу построенных автоматов. Они должны осуществлять переходы одновременно. Таким образом, сначала вычисляются все условия на переходах, а затем каждый автомат осуществляет соответствующий переход.

Пример преобразования итеративной программы

Проиллюстрируем рассмотренные преобразования на следующем (весьма надуманном) примере (рис. 31).

```
void main() {
  inc();
  dec();
}
```

а

```
void inc() {
  d.i += 2;
  dec();
}
```

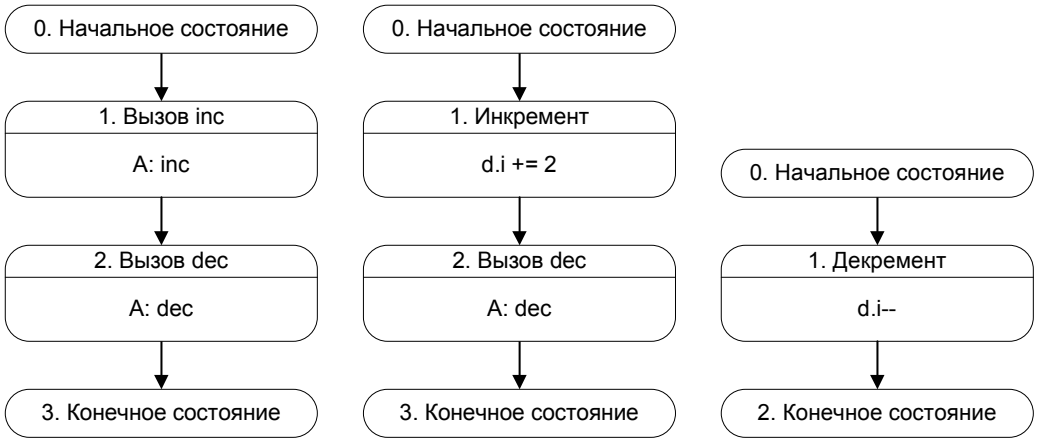
б

```
void dec() {
  d.i--;
}
```

в

Рис. 31. Преобразуемая программа. Главная процедура (а), процедура inc (б), процедура dec (в)

Им соответствуют прямые автоматы, изображенные на рис. 32.



а **б** **в**

Рис. 32. Прямые автоматы для процедур main (а), inc (б) и dec (в)

Соответствующие обратные автоматы изображены на рис. 33.

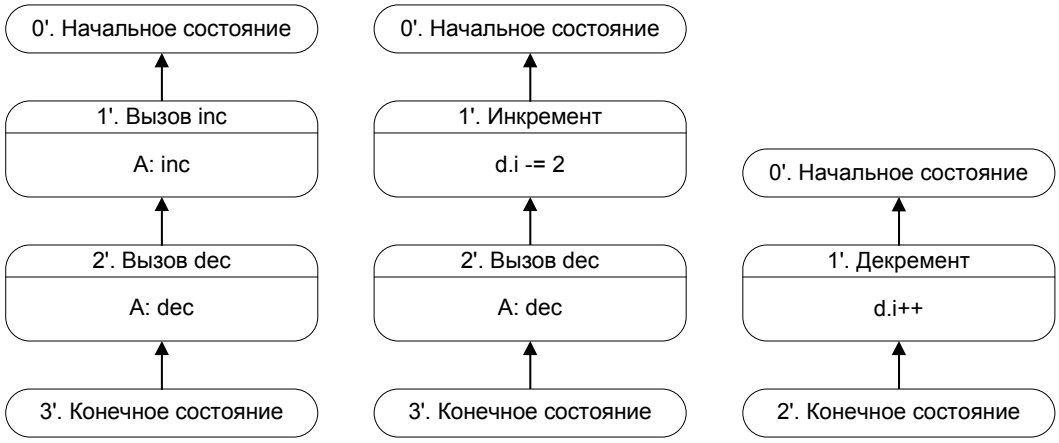


Рис. 33. Обратные автоматы для процедур main (а), inc (б) и dec (в)

Обозначим автоматы для процедур `main`, `inc` и `dec` через *A1*, *A2* и *A3* соответственно.

Рассмотрим преобразование автомата *A3*. *Список вызовов* для автомата *A3* состоит из двух пар: (*A1*, *2*) и (*A2*, *2*). При этому условие *R(A3)* записывается следующим образом:

```
state(A1) == 2 | state (A2) == 2,
```

где "|" — дизъюнкция. Преобразованные автоматы *A3* и *A3'* приведены на рис. 34.

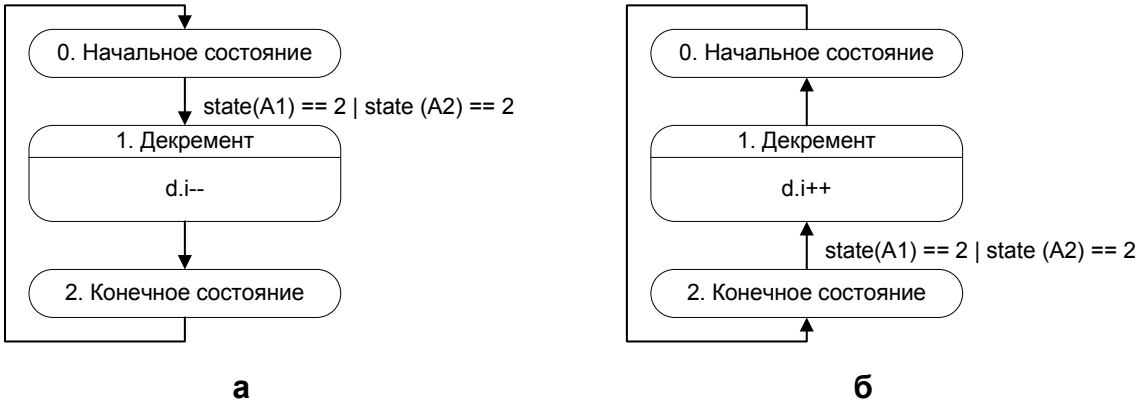
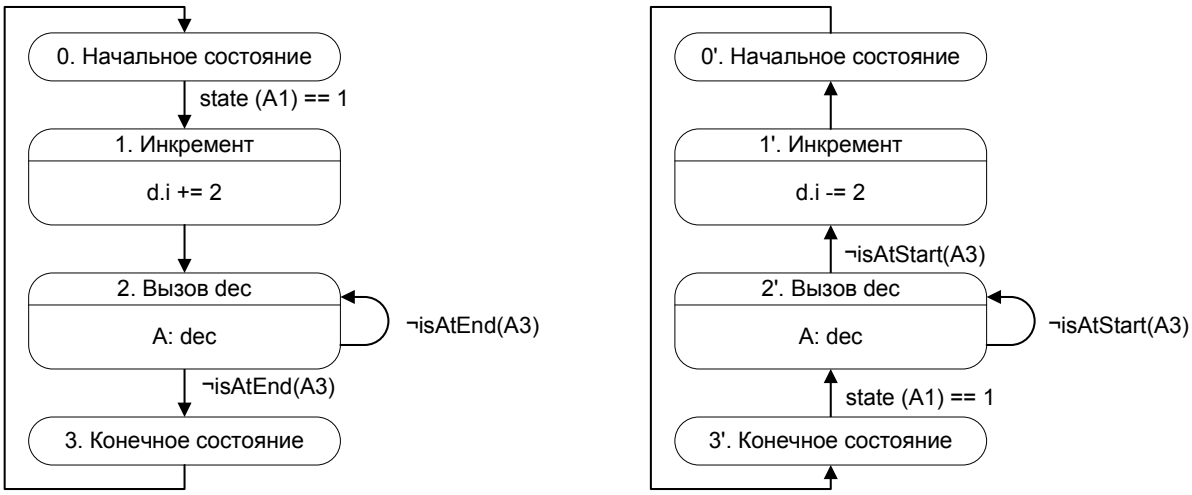


Рис. 34. Преобразованные автоматы A3 (а) и A3' (б)

Для автомата *A2* условие *R(A2)* имеет вид:

```
state(A1) == 1
```

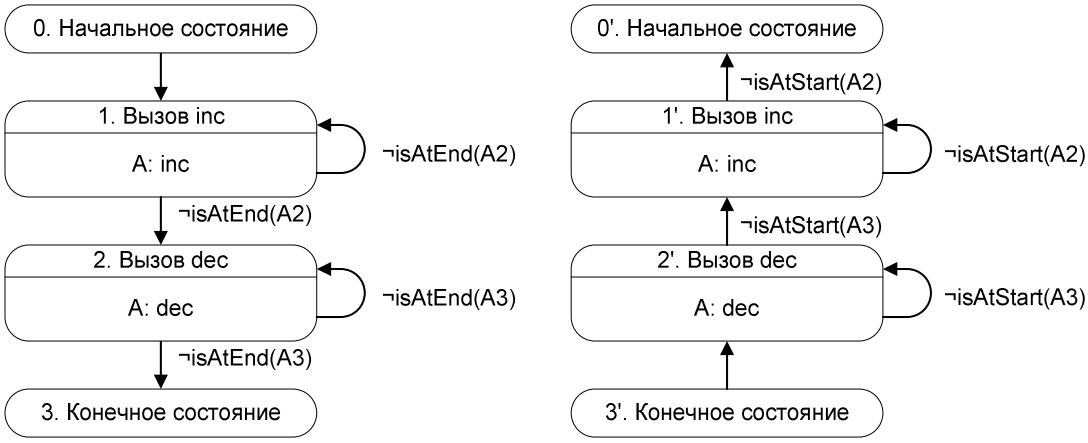
При этом в автоматах *A2* и *A2'* необходимо преобразовать состояния 2 и 2', как показано выше. Таким образом, получаем автоматы, изображенные на рис. 35.



а **б**

Рис. 35. Преобразованные автоматы A2 (а) и A2’ (б)

Аналогично преобразуя автоматы *A1* и *A1’*, получим рис. 36.



а **б**

Рис. 36. Преобразованные автоматы A1 (а) и A1’ (б)

Система взаимодействующих конечных автоматов *A1* (*A1’*), *A2* (*A2’*) и *A3* (*A3’*) реализует исходную программу.

Некоторые замечания о корректности преобразования программ

Обозначим совместное состояние автоматов *A1*, *A2*, *A3* тройкой вида: *(состояние автомата A1, состояние автомата A2, состояние автомата A3)*. Тогда, при прямом проходе имеем следующую цепочку состояний:

```
(0, 0, 0) → (1, 0, 0) → (1, 1, 0) → (1, 2, 0) →
(1, 2, 1) → (1, 2, 2) → (1, 3, 0) → (2, 0, 0) →
(2, 0, 1) → (2, 0, 2) → (3, 0, 0),
```

а при обратном проходе:

(3, 0, 0) → (2, 3, 2) → (2, 3, 1) → (2, 3, 0) →
(1, 3, 2) → (1, 2, 2) → (1, 2, 1) → (1, 2, 0) →
(1, 1, 3) → (1, 0, 3) → (0, 3, 2).

Заметим, что при *обратном проходе* цепочка состояний не совпадает с перевернутой цепочкой состояний, пройденных при *прямом проходе*.

Пусть в некоторый момент времени имеем цепочку вызовов. Все процедуры, участвующие в этой цепочке, назовем э*мулируемыми* в рассматриваемый момент времени, а остальные процедуры — *не эмулируемыми*.

При *прямом проходе* автоматы, соответствующие *не эмулируемым* в данный момент процедурам, находятся в *начальном состоянии,* а при *обратном проходе* эти автоматы находятся в *конечном состоянии*. Заметим, что при выполнении шага вперед все “незадействованные” автоматы переходят в *начальное состояние*, а при выполнении шага назад они переходят в *конечное состояние*. Таким образом, как при прямом, так и при обратном проходе, после перехода в состояние, вызывающее один из “незадействованных” автоматов, тот находится в состоянии, при котором будет осуществлена соответствующая эмуляция (либо прямого прохода, либо обратного). При рассмотрении “задействованных” автоматов такой проблемы не возникает, так как они и при прямом и при обратном проходе находятся в одном и том же состоянии.

Отдельного рассмотрения заслуживает случай, когда два вызова одной и той же процедуры идут подряд. При *прямом проходе* вызываемый автомат переходит из *конечного состояния* в *начальное* одновременно с переходом вызывающего автомата из одного состояния, осуществляющего вызов, в другое. Соответственно, когда вызывающий автомат находится в состоянии, соответствующем второму вызову, вызываемый автомат уже находится в начальном состоянии, и поэтому происходит повторная эмуляция процедуры. При *обратном проходе* одновременно с переходом вызывающего автомата выполняется переход вызываемого автомата в *конечное состояние*. Таким образом, и в этом случае преобразования выполненные на основе предлагаемого подхода корректны.

4.5.2. Рекурсивные программы

Преобразование рекурсивной программы

Вопрос о преобразовании рекурсивных программ в автоматные рассматривался в статье [4]. В данной работе предложен альтернативный метод, основанный на понятии *экземпляр автомата.*

В рекурсивной программе одна и та же процедура может встречаться в цепочке вызовов несколько раз. Назовем каждое вхождение процедуры в цепочку вызовов *экземпляром процедуры*. Проиллюстрируем это на примере функции, осуществляющей рекурсивное вычисление факториала:

```
int factorial(int a) {
  int r = 1;
  if (a > 1) {
    r = a * factorial(a - 1);
  }
  return r;
}
```

Данная функция вызывает себя в строке 4. При этом в нескольких *экземплярах процедуры* (тех, которые вызвали новый экземпляр этой процедуры) текущей будет строка 4, а в еще одном экземпляре (текущем) — некоторая другая строка.

Для отражения этого факта введем понятие *экземпляр автомата*. Как указывалось выше, пара из прямого и обратного автоматов может быть рассмотрена, как один автомат с двумя функциями переходов. *Экземпляр автомата* — объект, хранящий состояние такого автомата.

Определим для *экземпляра автомата* следующие функции:

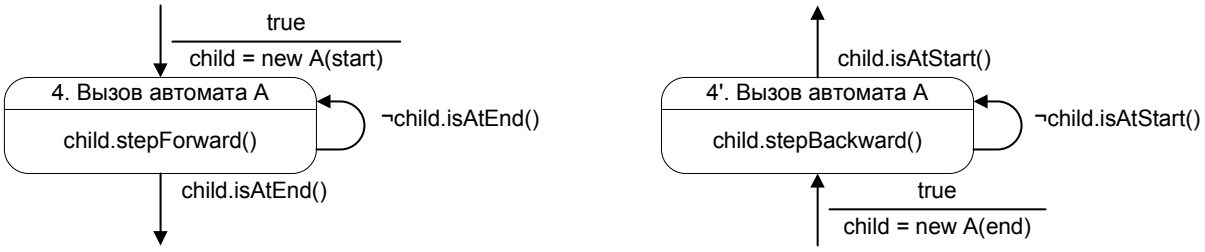
- `stepForward()` — сделать шаг вперед (изменить состояние и выполнить действия в соответствии с функцией переходов *прямого автомата*);
- `stepBackward()` — сделать шаг назад (изменить состояние и выполнить действия в соответствии с функцией переходов *обратного автомата*);

16. `isAtStart()` — проверка, находится ли *экземпляр автомата* в *начальном состоянии*;
17. `isAtEnd()` — проверка, находится ли *экземпляр автомата* в *конечном состоянии*;
18. `new автомат(start)` — создать новый *экземпляр автомата*, находящийся в *начальном состоянии*;
19. `new автомат(end)` — создать новый *экземпляр автомата*, находящийся в *конечном состоянии*.

Для каждого автомата можно создать несколько *экземпляров*, каждый из которых находится в своем состоянии.

При *прямом проходе* для каждого вызова процедуры создается новый *экземпляр автомата*, находящийся в *начальном состоянии*. Вызывающий автомат ожидает, пока созданный *экземпляр автомата* не закончит работу (перейдет в *конечное состояние*). Соответственно, при *обратном проходе* для каждого вызова процедуры создается новый *экземпляр автомата*, находящийся в *конечном состоянии*. Вызывающий автомат ожидает, пока созданный *экземпляр автомата* не закончит работу (перейдет в *начальное состояние*).

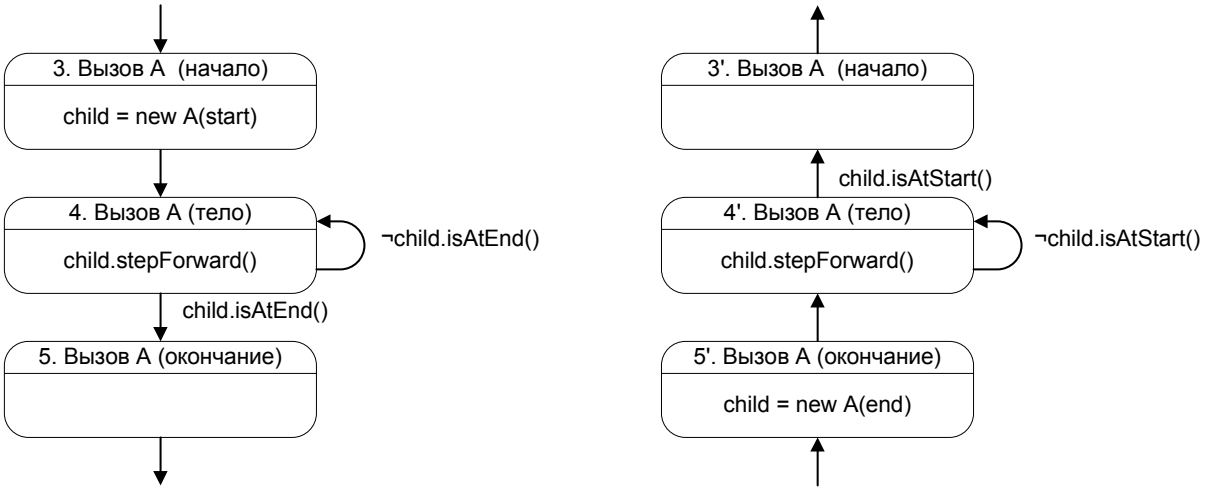
Фрагменты *прямого* и *обратного автоматов*, осуществляющих вызов автомата A, приведены на рис. 37.



а **б**

Рис. 37. Фрагменты прямого (а) и обратного (б) автоматов, вызывающие автомат A

Этими фрагментами можно заменить состояние, вызывающее автомат A в исходном автомате. Для получения автомата Мура следует использовать фрагменты, изображенные на рис. 38.



а **б**

Рис. 38. Фрагменты прямого (а) и обратного (б) автоматов Мура, вызывающие автомат A

Таким образом, в этом случае к *прямому* и о*братному автоматам* необходимо добавить по два новых состояния.

Пример преобразования рекурсивной программы

Проиллюстрируем предложенный метод на примере функции, вычисляющей факториал (исходная программа приведена в разделе 4.5.2). Данная функция использует явную рекурсию. Преобразование программ с косвенной рекурсией производится аналогично.

В соответствии с предложенным методом (см. раздел 2.2.2) преобразуем программу к виду, использующему модель данных. Передачу параметра будем производить через переменную `d.a`, а возврат результата — через переменную `d.r`:

```
void factorial() {
  d.r = 1;
  if (d.a > 1) {
    d.a--;
    factorial();
    d.r = d.r * (d.a + 1);
  }
```

```
}
```

Прямой и обратные автоматы, построенные по данной программе, изображены на рис. 39.

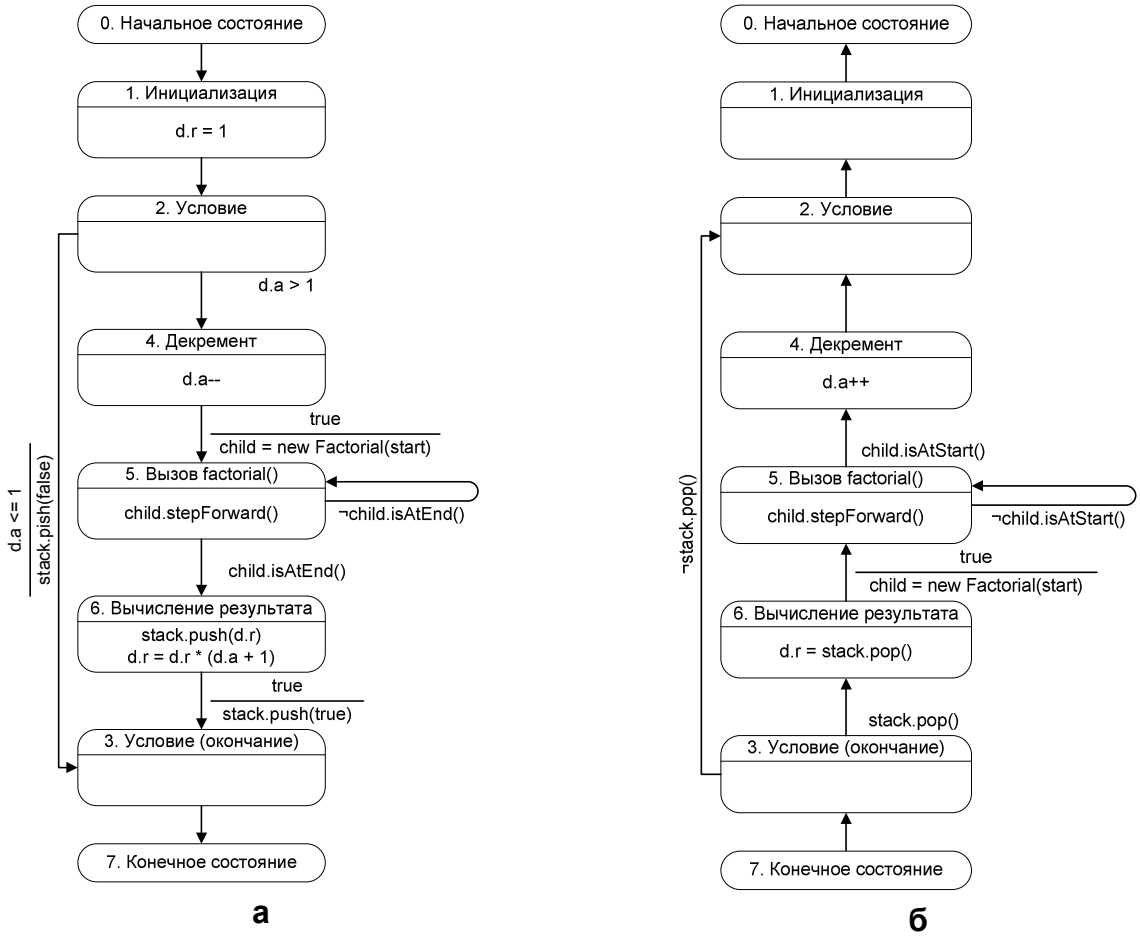


Рис. 39. Прямой (а) и обратный (б) автоматы для процедуры вычисления факториала

4.6. Формализация преобразования программы

Опишем формально процедуру построения системы взаимодействующих автоматов по программе, основанную на рассуждениях приведенных в разделах 4.1, 4.3 и 4.5.

Построенный автомат может взаимодействовать со стеком посредством операций, описанных в разделе 4.3.1.

Введем понятие “фрагмент автомата”, являющегося набором состояний автомата и переходов. При этом у некоторых переходы может быть не определено начальное или конечное состояние. Такие переходы называются входами и выходами фрагмента соответственно. Заметим, что один переход

может быть входом и выходом одновременно. В дальнейшем будем рассматривать фрагменты автоматов ровно с одним входом и выходом.

Рассмотрим язык, порождаемый следующей грамматикой:

1	*Программа*	::=	*Процедура Программа*
2		|	*Процедура*
3	*Процедура*	::=	*Операторы*
4	*Операторы*	::=	*Операторы Оператор*
5		|	
6	*Оператор*	::=	*ОператорПрисваивания*
7		|	*ОператорВетвления*
8		|	*ОператорЦикла*
9		|	*ВызовПроцедуры*
10	*ОператорПрисваивания*	::=	*Переменная = Выражение*
11	*ОператорВетвления*	::=	*Выражение* $Операторы_1$ $Операторы_2$
12	*ОператорЦикла*	::=	*Выражение Операторы*
13	*ВызовПроцедуры*	::=	*Процедура*

В рамках этого языка можно построить все управляющие конструкции, рассмотренные выше. При этом будем предполагать, что при вычислении выражений побочные действия не производятся. Таким образом, значения переменных меняются только при выполнении оператора присваивания.

Рассмотрим дерево вывода преобразуемой программы. Для каждой его вершины построим фрагменты прямого и обратного автоматов, соответствующих поддереву с корнем в данной вершине. Вершины будем преобразовывать в указанные фрагменты, в порядке выхода из них при обходе в глубину. Следовательно, при преобразовании вершины все ее потомки уже будут преобразованы.

Для каждого автомата требуется доказать следующие свойства:

1. *Адекватность* — система автоматов выполняет те же действия, что исходная программа.

2. *Обратимость* — при выполнении действий обратного автомата будут восстановлены исходные значения всех переменных, при условии, что в точке входа они были такими же, как в точке выхода при прямом проходе.
3. *Полнота* — для каждого не конечного состояния при любых значениях переменных условие на одном из переходов должно быть истинно, то есть дизъюнкция всех условий на переходах из состояния является тавтологией.
4. *Непротиворечивость* — условия на переходах из одного состояния не могут быть истинными одновременно.
5. *Отсутствие недостижимых состояний* — любое состояние может быть достигнуто по переходам из начального состояния.

При проверке на отсутствие недостижимых состояний, мы не будем требовать для каждого состояния наличие такой комбинации переменных, чтобы можно было достигнуть этого состояния, так как в общем случае эта задача не разрешима. Будем проверять только принципиальную возможность достижения этой вершины из начальной. При этом условия на переходах фактически игнорируются.

Указанные свойства будут доказываться посредством структурной индукции. Будем предполагать, что эти свойства выполняются для фрагментов полученных, преобразованием потомков вершины дерева и доказывать эти свойства для фрагмента, полученного преобразованием текущей вершины. Приведенные доказательства не являются математически строгими, но, на наш взгляд, служат достаточно хорошим обоснованием принятых решений.

Для фрагментов автоматов будем доказывать от входа фрагмента (напомним, что мы рассматриваются фрагменты с ровно одним входом и выходом). При этом так же будем доказывать достижимость выхода из фрагмента.

Под обратимостью фрагмента будем понимать, что при его обратном проходе восстанавливаются значения переменных на момент входа во

фрагмент, при условии, что в точке входа при обратном проходе они были такими же, как в точке выхода при прямом проходе.

Для описания состояний автоматов будем использовать следующую нотацию:

Состояние ::= *ПереходыВ* < *Действия* > *ПереходыИз*
Действия ::= *Действие; Действия*
| *Действие*
ПереходыВ ::= *ПереходВ*
| *ПереходВ*; *ПереходыВ*
ПереходВ ::= *ИмяФрагмента* {, *Действие*}
ПереходыИз ::= *ПереходИз*
| *ПереходИз*; *ПереходыИз*
ПереходИз ::= *Условие*, *ИмяФрагмента* {, *Действие*}
Действие ::= *Переменная* = *Выражение*
| `push(`*Выражение*`)`
| *Переменная* = `pop()`

Заметим, что переход во фрагмент автомата не порождает новый переход, а лишь “замыкает” вход соответствующего фрагмента на описываемое состояние. То же верно и для перехода из фрагмента.

Если для перехода в состояние указан фрагмент *Вход*, то он становится входом фрагмента. Аналогично, если для перехода из состояния указан фрагмент *Выход*, то переход становится выходом фрагмента.

Действие на переходе можно добавить как при определении входящего, так и при определении исходящего перехода. Таким образом, на переходе может осуществляться два действия. При этом сначала будет выполняться действие, определенное состоянием из которого осуществляется переход, а затем действие, определенное состоянием в которое происходит переход.

Доказательство свойства, обратимости будем осуществлять на границах переходов, то есть когда действия, указанные на выходах уже выполнены, а на входах еще нет.

Начнем доказательство с отдельных операторов (продукции 10-13), затем перейдем к последовательностям операторов (продукции 4-9), а от них — к процедурам (продукции 1-3).

4.6.1. Преобразование оператора присваивания

Оператор присваивания преобразуется, как указано в разделах 4.1.1 и 4.3.1, при этом фрагменты прямого и обратного автоматов будут состоять из одного состояния каждый.

Преобразуемая продукция:

10 *ОператорПрисваивания* ::= *Переменная* = *Выражение*

Состояние прямого автомата в используемой нотации имеет вид:

Вход < `push(`*Переменная*`);` *Переменная* = *Выражение* > `true,` *Выход*

При этом состояние обратного автомата имеет вид:

Вход < *Переменная* = `pop()>` *Выход*

Докажем выполнение свойств для оператора присваивания. Здесь и далее начало и конец доказательства помечаются значками ► и ◄ соответственно.

Адекватность. ► При выполнении действий в состоянии значение *переменной* сохраняется в стеке и ей присваивается значение выражения. Таким образом, на выходе из фрагмента значение переменной равно выражению. ◄

Обратимость. ► На выходе из фрагмента значение *переменной* равно значению в вершине стека при входе во фрагмент. Так как по индуктивному предположению все фрагменты обратимы, то на вершине стека содержалось значение, сохраненной при прямом проходе. Таким образом, на выходе переменная имеет значение, которое она имела при входе во фрагмент прямого автомата при прямом проходе, а стек возвращен к исходному состоянию.◄

Полнота и непротиворечивость. ► Из каждого состояния осуществляется безусловный переход, следовательно, свойства выполняются. ◄

Отсутствие недостижимых состояний. ► Вход ведет непосредственно в добавленное состояние. Выход является безусловным переходом из достижимого состояния. ◄

4.6.2. Преобразование оператора ветвления

Оператор присваивания преобразуется, как указано в разделах 4.1.4, 4.1.5 и 4.3.2, при этом дополнительный стек не используется.

Преобразуемая продукция:

11 *ОператорВетвления* ::= *Выражение Операторы*$_1$ *Операторы*$_2$

Для прямого автомата добавляются два состояния:

1: *Вход* < > *Условие*, *Да*; ¬*Условие*, *Нет*

2: *Да*, `push(true)`; *Нет*, `push(false)` < > *Выход*

Для обратного автомата так же добавляются два состояния:

1': *Да*; *Нет* < > *Выход*

2': *Вход* < > `peek()`, *Да'*, `pop()`; ¬`peek()`, *Нет'*, `pop()`

Построенные фрагменты ссылаются на фрагменты для операторов, выполняемых при истинности и ложности условия: *Да* (построен для ребенка *Операторы*$_1$) и *Нет* (построен для ребенка *Операторы*$_2$) соответственно. Фрагменты *Да'* и *Нет'* — соответствующие фрагменты обратных автоматов. В каждом фрагменте к состояниям, определенным во фрагментах добавляются новые состояния.

Номера, указанные для состояний используются исключительно в доказательствах свойств и не переносятся в построенный фрагмент.

Докажем выполнение рассматриваемых свойств.

Адекватность. ► Если *Условие* истинно, то будет произведен переход во фрагмент *Да* и на вершине стека будет значение `true`. В противном случае, будут произведен переход во фрагмент *Нет*, а в вершине стека будет значение `false`. По индуктивному предположению, действия выполняемые фрагментами *Да* и *Нет* соответствуют *Операторы*$_1$ и *Операторы*$_2$. ◄

Обратимость. ► При обратном проходе вход осуществляется в состояние 2’. Если на вершине стека значение true, то оно снимается со стека и осуществляется переход во фрагмент *Да’*, в противном случае после снятия значения со стека происходит переход во фрагмент *Нет’*. По индуктивному предположению, на входе в вершине стека содержится значение, помещенное туда при прямом проходе, таким образом, фрагмент обратного автомата выбирается верно, после этого значение удаляется из стека. По тому же индуктивному предположению, после выхода из фрагментов *Да’* или *Нет’*, значения всех переменных будут восстановлены. ◄

Полнота и Непротиворечивость. ► Переходы из состояний 2 и 1’ являются безусловными. Переходы из состояний 1 (2’) помечены взаимно обратными условиями. ◄

Отсутствие недостижимых состояний. ► Вход ведет непосредственно в состояние 1 (2’), из которого выходят переходы во фрагменты *Да* и *Нет* (*Да’* и *Нет’*), от входов которых по индуктивному предположению достижимы все определенные в них состояния, а так же выходы. Следовательно состояние 2 (1’) так же достижимо. По этому, так же достижим и выход. ◄

4.6.3. Преобразование оператора цикла

Оператор цикла преобразуется, как указано в разделах 4.1.6 и 4.3.3, при этом дополнительный стек так же не используется.

Преобразуемая продукция:

12 *ОператорЦикла* ::= *Выражение Операторы*

Состояние, добавляемое к прямому автомату:

Вход, `push(false)`; *Операторы*, `push(true)` <

> *Условие*, *Операторы*; ¬*Условие*, *Выход*

Состояние, добавляемое к обратному автомату:

Вход; *Операторы’* <

> `peek()`, *Операторы’*, `pop()`; ¬`peek()`, *Выход*, `pop()`

Здесь *Операторы* и *Операторы'* — фрагменты прямого и обратного автоматов для тела цикла.

Докажем выполнение рассматриваемых свойств.

Адекватность. ► При входе во фрагмент в вершину стека помещается `false`. После этого пока *Условие* истинно осуществляет переход во фрагмент *Операторы*, на выходе из которого, каждый раз, в вершину стека помещается `true`. Таким образом, по индуктивному предположению тело цикла выполняется, пока истинно условие. При этом стек содержит столько элементов `true`, сколько было итераций цикла и один элемент со значением `false`. ◄

Обратимость. ► При обратном проходе, пока на вершине стека лежит `true`, оно снимается и осуществляется переход во фрагмент *Операторы'*. При этом по индуктивному предположению, при каждом попадании в добавленное состояние, вершина стека содержит значение `true` или `false`, помещенное туда при прямом проходе. Следовательно, во фрагмент *Операторы'* будет осуществлено столько переходов, сколько их было осуществлено во фрагмент *Операторы'* при прямом проходе. При этом так как на каждом переходе из добавленного состояния значение удаляется из вершины стека, то стек правильно восстанавливается как при переходе во фрагмент *Операторы'*, так и при выходе. ◄

Полнота и Непротиворечивость. ► Переходы из добавленных состояний помечены взаимно обратными условиями. ◄

Отсутствие недостижимых состояний. ► Добавленные состояния достижимы непосредственно от входа. При этом, один из переходов ведет во фрагмент *Операторы* (*Операторы'*), следовательно, по индуктивному предположению все его состояния достижимы. Выход достижим непосредственно из добавленного состояния. ◄

4.6.4. Преобразование оператора вызова процедуры

Оператор вызова процедуры преобразуется, как указано в разделах 4.1.3 и 4.5.2, при этом возможная не рекурсивность некоторых процедур не используется.

Преобразуемая продукция:

13 *ВызовПроцедуры* ::= *Процедура*

Состояние, добавляемое к прямому автомату:

Вход, `child` = *Процедура*`(start)`; *Состояние*

`<child.stepForward()>`

`¬child.isAtEnd()`, *Состояние*; `child.isAtEnd()`, *Выход*

Состояние, добавляемое к обратному автомату:

Вход, `child` = *Процедура*`(end)`; *Состояние*

`<child.stepBackward()>`

`¬child.isAtStart()`, *Состояние*; `child.isAtStart()`, *Выход*

Здесь *Состояние* — добавляемое состояние; *Процедура*`(start)` — создание экземпляра прямого автомата для процедуры в начальном состоянии; *Процедура*`(end)` — создание экземпляра обратного автомата для процедуры в конечном состоянии; `child.stepForward()` — шаг экземпляра автомата вперед, а `child.stepForward()` — шаг экземпляра автомата назад.

Докажем выполнение рассматриваемых свойств при дополнительном предположении, что создаваемый экземпляр автомата адекватен и обратим. Это предположение будет доказано в разделе 4.6.6.

Адекватность. ► После входа во фрагмент, пока созданный экземпляр автомата не придет в конечное состояние производятся шаги созданного экземпляра автомата вперед. По дополнительному предположению, это осуществляется корректно. ◄

Обратимость. ► При обратном проходе, осуществляются обратные шаги экземпляра автомата, пока тот не придет в исходное состояние. По дополнительному предположению после каждого такого шага значения

переменных восстанавливают свои значения, а следовательно при выходе они примут исходные значения. ◄

Полнота и Непротиворечивость. ► Переходы из добавленных состояний помечены взаимно обратными условиями. ◄

Отсутствие недостижимых состояний. ► Вход ведет непосредственно в добавленное состояние. Выход является безусловным переходом из достижимого состояния. ◄

4.6.5. Преобразование последовательностей операторов

Продукции 6-9 введены для удобства записи, и их преобразование совпадает с результатом преобразования левой части соответствующей продукции.

Продукция

5 *Операторы* ::=

фактически определяет пустой оператор, в результатом преобразование которого будет отдельный переход (фрагмент без состояний), являющийся одновременно входом и выходом. Для такого фрагмента, все свойства, очевидно, выполняются:

Адекватность и Обратимость. ► Действия не осуществляются. ◄

Полнота и Непротиворечивость. ► Фрагмент не содержит состояний. ◄

Отсутствие недостижимых состояний. ► Вход одновременно является выходом. ◄

В свою очередь, продукция

4 *Операторы* ::= *Операторы Оператор*

определяет последовательность операторов. Для преобразования вершины дерева, соответствующей такой продукции необходимо объединить фрагменты, соответствующие *Операторам* (*Ф1* и *Ф1'*) и *Оператору* (*Ф2* и *Ф2'*). При этом добавлять новые состояния не требуется, а следует "замкнуть" выход *Ф1* и вход *Ф2*. Докажем, что при этом все рассматриваемые свойства выполняются.

Адекватность. ► После выхода из фрагмента *Ф1* осуществляется вход во фрагмент *Ф2*. Таким образом, операторы, преобразованием которых были получены фрагменты *Ф1* и *Ф2* будут выполняться последовательно и по в соответствии с индуктивным предположением будет получен корректный результат ◄

Обратимость. ► По индуктивному предположению, после выхода из фрагмента *Ф2'* значения переменных будут восстановлены в промежуточные значения, следовательно и после выхода из фрагмента *Ф1'* они будут восстановлены в исходные значения. ◄

Полнота и Непротиворечивость. ► Состояния не добавляются ◄

Отсутствие недостижимых состояний. ► По индуктивному предположению выходы фрагментов *Ф1* и *Ф2'* достижимы, следовательно достижимы входы фрагментов *Ф2* и *Ф1'*. Соответственно, достижимы все состояния построенного фрагмента, а так же его выход. ◄

4.6.6. Преобразование процедуры

Рассмотрим преобразования вершин дерева, соответствующих результатам продукций

1 *Программа* ::= *Процедура Программа*
2 | *Процедура*
3 *Процедура* ::= *Операторы*

Продукции 1 и 2 определяют программу как последовательность процедур. Таким образом, результат преобразования программы — множество пар автоматов, полученных при преобразовании процедур.

Третья продукция определяет процедуру как последовательность операторов. Как указано в предыдущем разделе, последовательность операторов может быть преобразована во фрагмент автомата. Для построения автомата по процедуре к соответствующим фрагментам требуется добавить по два состояния: начальное и конечное.

Для прямого автомата:

начальное состояние: < > `true`, *Операторы*

конечное состояние: *Операторы* < >

Для обратного автомата:

начальное состояние: < > `true`, *Операторы'*

конечное состояние: *Операторы'* < >

Здесь *Операторы* и *Операторы'* — фрагменты прямого и обратного автоматов, соответствующие телу процедуры.

Докажем, что рассматриваемые свойства выполняются для построенного автомата.

Адекватность. ► По индуктивному предположению, фрагмент *Операторы*, выполняет действия, описанные в теле процедуры. ◄

Обратимость. ► По индуктивному предположению, фрагмент *Операторы'*, правильно обращает действия, описанные в теле процедуры. ◄

Полнота и Непротиворечивость. ► В построенных автоматах из начальных состояний выходят безусловные переходы, а из конечного состояния переходы не выходят. Но для полноты, переходы из конечного состояния не требуются. ◄

Отсутствие недостижимых состояний. ► Из начального состояния непосредственно достижим вход фрагмента *Операторы* (*Операторы'*), следовательно, по индуктивному предположению, достижим и выход этого фрагмента, который ведет в конечное состояние. Таким образом, все состояния построенных автоматов достижимы. ◄

4.6.7. Завершение доказательства

Так как для каждой из продукций выполняется индуктивное предположение, при условии, что оно выполняется для всех детей вершины, порожденной продукцией, то осталось доказать базу индукции. В нашем дереве листами могут быть только вершины, соответствующие операторам присваивания, пустому оператору и оператору вызова процедуры. Для первых

двух из них, было приведено непосредственное доказательство выполнения рассматриваемых свойств.

Доказательства полноты, непротиворечивости и отсутствия недостижимых состояний для оператора вызова процедуры были приведены в явном виде, следовательно эти свойства выполняются для всех автоматов.

Доказательства *адекватности* и *обратимости* основывались на предположении истинности этих фактов для процедуры целиком. Таким образом, для них доказательства следует дополнить.

► Рассмотрим дерево вызовов при трассировке программы посредством построенных автоматов. По построению, листы этого дерева не содержат вызовов других процедур. Заметим, что если экземпляр автомата не вызывал другие автоматы, то для него *адекватность* и о*братимость* уже доказана по индукции (так как для этого не требуется доказывать базу для вызовов). Таким образом, у дерева можно “оборвать листья”, при этом все вызовы, соответствующие “оборванным” листьям являются *адекватными* и *обратимыми* по доказанному выше.

Заметим, что, если дерево вызовов конечно, то последовательно “обрывая листья” можно оставить только корень, для которого так же подходит приведенное доказательство. Следовательно, в любой момент все выполненные действия являются *адекватными* и *обратимыми*. Таким образом, построенная система автоматов так же является *адекватной* и *обратимой*. ◄

4.7. Выводы

В данном разделе были приведены общие соображения о построении системы взаимодействующих автоматов по программе, а так же приведены формальные и неформальные методы построения таких систем по тексту программы.

Для формального метода было приведено доказательство *адекватности*, *обратимости*, *полноты*, *непротиворечивости* и достижимости всех состояний для построенной системы автоматов. По построению, такая система содержит

по одному прямому и обратному автомату для каждой процедуре, при этом количество состояний автомата линейно относительно количества операторов в процедуре.

ГЛАВА 5. XML-ФОРМАТ ОПИСАНИЯ ВИЗУАЛИЗАТОРОВ

5.1. Общая структура

XML-описание визуализатора состоит из трех основных частей:

1. общее описание визуализатора;
2. описание логики визуализатора;
3. описание конфигурации визуализатора.

Общее описание визуализатора содержит основную иуформацию о визуализаторе:

1. название визуализируемого алгоритма;
2. информацию об авторе визуализатора и его руководителе;
3. название пакета, в котором находится визуализатор и его главного класса;
4. предпочтительные размеры визуализатора.

Описание логики визуализатора содержит описания процедур (автоматов) реализующих визуализируемый алгоритм. Так же в описании логики визуализатора может присутствовать информация, облегчающая построение обратного автомата.

Описание конфигурации визуализатора содержит информацию о том, как визуализатор должен отображаться на экране, в том числе:

1. размеры частей визуализатора,
2. конфигурация нестандартных элементов управления;
3. сообщения, выдаваемые пользователю (в том числе, сообщения об ошибках);
4. таблицы стилей для элементов оформления;
5. допустимые границы изменения параметров пользователем.

В следующем разделе подробно описываются все теги, которые могут быть использованы в описании визуализатора.

5.2. Описания тегов

В этом разделе описываются теги, которые могут быть использованы в XML-описании визуализатора. Для каждого тега приведено его краткое описание, описание атрибутов и вложенных тегов. Для формализации использования тегов для каждого из них приведены фрагменты определения типа документа [33] (Document Type Definition, DTD) и XML-схема [34, 35, 36].

Описание тегов и атрибутов содержит столбец, помеченный диезом (#), обозначающий какое количество раз может встречаться этот тег. Используются следующие обозначения:

- `*` — произвольное количество раз;
- `+` — не менее одного раза;
- `1` — ровно один раз.
- `?` — не более одного раза.

Описание атрибутов так же включает столбец, помеченный буквой “И”. В этом столбце стоит “+”, если значение атрибута должно задаваться на нескольких языках. Имена атрибутов для различных языков образуются из имени атрибута с добавленным к нему дефисом и двухбуквенным кодом языка. На пример, вместо атрибута `message` нужно будет задать атрибуты `message-ru` и `message-en` для русского и английского языка соответственно.

Для каждого тега приведены его атрибуты и описание содержимого. При описании вложенных тегов используется сокращение `steps` обозначающее одно из `step`, `call-auto`, `if` или `while`.

5.2.1. Общее описание визуализатора (тег visualizer)

Атрибуты тега `visualizer` содержат общее описание визуализатора, а его потомки содержат описание логики (тег `algorithm`) и конфигурации (тег `configuration`) визуализатора.

DTD

```
<!ELEMENT visualizer
  ((algorithm, configuration) | (configuration, algorithm))
```

```
>

<!ATTLIST visualizer
  id                  ID          #REQUIRED
  package             CDATA       #REQUIRED
  main-class          CDATA       #REQUIRED

  preferred-width     CDATA       #REQUIRED
  preferred-height    CDATA       #REQUIRED

  name-ru             CDATA       #REQUIRED
  name-en             CDATA       #REQUIRED

  copyright-ru        CDATA       #REQUIRED
  copyright-en        CDATA       #REQUIRED

  author-en           CDATA       #REQUIRED
  author-ru           CDATA       #REQUIRED
  author-email        CDATA       #REQUIRED

  supervisor-en       CDATA       #REQUIRED
  supervisor-ru       CDATA       #REQUIRED
  supervisor-email    CDATA       #REQUIRED

  xmlns:xsi           CDATA       #FIXED
      "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation   CDATA #FIXED
      "http://ips.ifmo.ru/vizi/schema/visualizer.xsd"
>
```

XML-схема

```
<xsd:element name = "visualizer" type = "visualizer"/>
<xsd:complexType name = "visualizer">
  <xsd:sequence>
    <xsd:element
      name = "algorithm"
      type = "algorithm"
    />
    <xsd:element
      name = "configuration"
      type = "configuration"
    />
  </xsd:sequence>
  <xsd:attribute name = "id"
        type = "id"                          use = "required"/>
  <xsd:attribute name = "package"
        type = "xsd:string"                  use = "required"/>
  <xsd:attribute name = "main-class"
        type = "xsd:string"                  use = "required"/>
  <xsd:attribute name = "preferred-width"
        type = "xsd:positiveInteger"         use = "required"/>
  <xsd:attribute name = "preferred-height"
        type = "xsd:positiveInteger"         use = "required"/>
```

```
    <xsd:attribute name = "name-ru"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "name-en"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "copyright-ru"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "copyright-en"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "author-en"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "author-ru"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "author-email"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "supervisor-en"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "supervisor-ru"
          type = "xsd:string"                use = "required"/>
    <xsd:attribute name = "supervisor-email"
          type = "xsd:string"                use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
id	1		Идентификатор визуализатора
package	1		Пакет, в котором находится визуализатор
main-class	1		Имя основного класса визуализатора
preferred-width	1		Предпочтительная ширина визуализатора в пикселях
preferred-height	1		Предпочтительная высота визуализатора в пикселях
name	1	+	Название визуализатора
author	1	+	Информация об авторе визуализатора
author-email	1		E-mail автора визуализатора
supervisor	1	+	Информация о руководителе проекта
supervisor-email	1		E-mail руководителя проекта
copyright	1	+	Информация о подразделении, создавшем визуализатор

Содержимое

Вложенные элементы

Тег	#	Описание
algorithm	1	Описание логики визуализатора
configuration	1	Описание модели данных

Пример

```
<visualizer
  id          = "FindMaximum"
  package     = "ru.ifmo.vizi.find_max"
```

```
  main-class = "FindMaximumVisualizer"

  preferred-width  = "400"
  preferred-height = "250"

  name-ru      = "Поиск максимума в массиве "
  name-en      = "Search for maximimum element in the array"

  author-ru    = "Георгий Корнеев"
  author-en    = "Georgiy Korneev"
  author-email = "kgeorgiy@rain.ifmo.ru"

  supervisor-ru    = "Георгий Корнеев"
  supervisor-en    = "Georgiy Korneev"
  supervisor-email = "kgeorgiy@rain.ifmo.ru"

  copyright-ru = "Copyright \u00A9 Кафедра КТ, СПбГУ ИТМО,
       2003-2004"
  copyright-en = "Copyright \u00A9 Computer Technologies
       Department, SPb IFMO, 2003-2004"
> ... </visualizer>
```

5.3. Описание визуализируемой программы

5.3.1. Основные элементы

Описание логики визуализатора (тег algorithm)

Тег `algorithm` является контейнером для описаний автоматов и модели данных, используемой визуализатором.

DTD

```
<!ELEMENT algorithm
  (import*, variable*, toString, auto+, method*)>
<!ATTLIST algorithm>
```

XML-схема

```
<xsd:complexType name="algorithm">
  <xsd:sequence>
    <xsd:element
      name        = "import"
      type        = "xsd:string"
      minOccurs   = "0"
      maxOccurs   = "unbounded"
    />
    <xsd:element
      name        = "variable"
      type        = "global-variable"
      minOccurs   = "0"
      maxOccurs   = "unbounded"
```

```
    />
    <xsd:element
      name        = "toString"
      type        = "xsd:string"
    />
    <xsd:element
      name        = "auto"
      type        = "auto"
      maxOccurs   = "unbounded"
    />
    <xsd:element
      name        = "method"
      type        = "method "
      minOccurs   = "0"
      maxOccurs   = "unbounded"
    />
  </xsd:sequence>
</xsd:complexType>
```

Атрибуты

Нет

Содержимое

Вложенные элементы

Тег	#	Описание
`import`	*	Описание импортируемых пакетов и классов
`variable`	*	Описание глобальных переменных
`toString`	1	Описание процедуры преобразования текущего состояния в строку
`method`	*	Описание глобальных методов
`auto`	+	Описание автомата

Пример

```
<algorithm> ... </algorithm>
```

Описание импортируемых пакетов и классов (тег import)

Тег `import` служит для описания пакетов и классов, импортируемых реализацией алгоритма. Тело тега должно содержать описание импортируемого пакета или класса в виде, определенном в разделе 7.5 [37].

Автоматически импортируются пакет `ru.ifmo.vizi.base.auto` и класс `java.util.Locale`.

DTD

```
<!ELEMENT import (#PCDATA)*>
<!ATTLIST import>
```

XML-схема

```
<xsd:element
  name         = "import"
  type         = "xsd:string"
  minOccurs    = "0"
  maxOccurs    = "unbounded"
/>
```

Атрибуты

Нет

Содержимое

Текст — описание импортируемого пакета или класса.

Примеры

```
<import>java.io.*</import>
<import>java.util.List</import>
```

Описание глобальных переменных (тег variable)

Тег `variable` непосредственно внутри тега `algorithm` служит для описания глобальных переменных, доступных во всех автоматах. Имена переменных должны быть корректными java-идентификаторами и не содержать символов подчеркивания.

DTD

```
<!ELEMENT variable EMPTY>
<!ATTLIST variable
  description         CDATA       #REQUIRED
  name                CDATA       #IMPLIED
  type                CDATA       #IMPLIED
  value               CDATA       #IMPLIED
>
```

XML-схема

```
<xsd:complexType name="global-variable">
  <xsd:attribute
    name = "name"
    type = "id"
    use  = "required"
  />
  <xsd:attribute
    name = "type"
    type = "type"
    use  = "required"
  />
  <xsd:attribute
    name = "value"
```

```
    type = "value"
    use  = "required"
  />
  <xsd:attribute
    name = "description"
    type = "description"
    use  = "required"
  />
</xsd:complexType>
```

Атрибуты

Атрибут	#	Описание
name	1	Имя переменной
type	1	Тип переменной
value	1	Исходное значение переменной
description	1	Словесное описание переменной

Содержимое

Нет

Примеры

```
<variable
  description = "Массив для поиска"
  name        = "a"
  type        = "int[]"
  value       = "new int[]{1, 2, 3, 1, 3, 5, 6}"
/>
```

Описание глобальных методов (тег method)

Тег method служит для описания глобальных методов.

DTD

```
<!ELEMENT method (#PCDATA)*>
<!ATTLIST method
  header              CDATA       #REQUIRED
  comment             CDATA       #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="method">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute
        name = "header"
        type = "xsd:string"
        use  = "required"
      />
      <xsd:attribute
        name = "comment"
```

```
        type = "description"
        use  = "required"
      />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Атрибуты

Нет

Содержимое

Текст — текст метода на *Java*.

Пример

```
<method
  header  = "int strToInt(String s)"
  comment = "Преобразования строки в целое"
>
  try {
    return Integer.parseInt(s);
  } catch (Exception e) {
    return 0;
  }
</method>
```

Описание метода преобразования текущего состояния в строку (тег toString)

Тег `toString` служит для описания процедуры для преобразования текущего состояния модели данных в строку. Метод преобразование модели данных в строку используется для проверки корректности сгенерированного обратного автомата. Для дополнительной информации смотри раздел 6.4.2.

Атрибуты

Нет

Содержимое

Тест — текст процедуры

Пример

```
<toString>
  StringBuffer s = new StringBuffer();
  s.append("max = ").append(@Main@max).append("\n");
  s.append("i = ").append(@Main@i).append("\n");
  return s.toString();
</toString>
```

5.3.2. Описание процедур

Описание процедур (тег auto)

Тег `auto` служит для описания процедур (автоматов), реализующих визуализируемый алгоритм. Каждая процедура имеет уникальный идентификатор, который должен быть корректным Java-идентификатором класса и не содержать символов подчеркивания.

DTD

```
<!ELEMENT auto (variable*, start?, (%steps;)+, finish?)>
<!ATTLIST auto
  id                  ID          #REQUIRED
  description         CDATA       #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="auto">
  <xsd:sequence>
    <xsd:element
      name        = "variable"
      type        = "local-variable"
      minOccurs   = "0"
      maxOccurs   = "unbounded"
    />
    <xsd:element
      name        = "start"
      type        = "start-finish"
      minOccurs   = "0"
    />
    <xsd:group
      ref         = "automata-steps"
      maxOccurs   = "unbounded"
    />
    <xsd:element
      name        = "finish"
      type        = "start-finish"
      minOccurs   = "0"
    />
  </xsd:sequence>
  <xsd:attribute
    name    = "id"
    type    = "id"
    use     = "required"
  />
  <xsd:attribute
    name    = "description"
    type    = "description"
    use     = "required"
```

```
  />
</xsd:complexType>
```

Атрибуты

Атрибут	#	Описание
id	1	Идентификатор процедуры
description	1	Словесное описание процедуры

Содержимое

Вложенные элементы

Тег	#	Описание
variable	*	Описание локальных переменных
start	?	Описание начального состояния автомата
steps	+	Описание шагов автомата
finish	?	Описание конечного состояния автомата

Примеры

```
<auto id="Main" description="Поиск максимума в массиве">
  ...
</auto>
```

Описание локальных переменных (тег variable)

Тег `variable` внутри тега `auto` служит для описания локальных переменных, доступных только внутри этого автомата. Если определены локальная и глобальная переменные с совпадающими именами, то в автомате будет доступна только локальная переменная.

Имена переменных должны быть корректными java-идентификаторами и не содержать символов подчеркивания.

DTD

```
<!ELEMENT variable EMPTY>
<!ATTLIST variable
  description         CDATA       #REQUIRED
  name                CDATA       #IMPLIED
  type                CDATA       #IMPLIED
>
```

XML-схема

```
<xsd:complexType name="local-variable">
  <xsd:attribute
    name = "name"
    type = "id"
    use  = "required"
  />
```

```
  <xsd:attribute
    name = "type"
    type = "type"
    use  = "required"
  />
  <xsd:attribute
    name = "description"
    type = "description"
    use  = "required"
  />
</xsd:complexType>
```

Атрибуты

Атрибут	#	Описание
name	1	Имя переменной
type	1	Тип переменной
description	1	Словесное описание переменной

Содержимое

Нет

Примеры

```
<variable
  description = "Переменная цикла"
  name        = "i"
  type        = "int"
/>
```

Описание начального и конечного состояния автомата (теги start и finish)

Тег `start` (`finish`) служит для задания метода отображения и комментария к начальному (конченому) состоянию автомата.

DTD

```
<!ELEMENT start (draw?)>
<!ATTLIST start
  comment-ru          CDATA       #IMPLIED
  comment-en          CDATA       #IMPLIED
  comment-args        CDATA       #IMPLIED
>
<!ELEMENT finish (draw?)>
<!ATTLIST finish
  comment-ru          CDATA       #IMPLIED
  comment-en          CDATA       #IMPLIED
  comment-args        CDATA       #IMPLIED
>
```

XML-схема

```
<xsd:complexType name="start-finish">
  <xsd:sequence>
```

```
    <xsd:element
      name        = "draw"
      type        = "draw"
      minOccurs   = "0"
    />
  </xsd:sequence>
  <xsd:attributeGroup ref="comment"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
comment	1	+	Комментарий к начальному состоянию
comment-args	?		Аргументы комментария

Содержимое

Вложенные элементы

Тег	#	Описание
draw	?	Метод отображения начального состояния.

Примеры

```
<start
  comment-ru="На экране изображен массив, в котором будет
        осуществляться поиск максимума"
  comment-en="There is an array on the display"
>
  ...
</start>

<finish
  comment-ru="Максимум найден ({0})"
  comment-en="Maximum found ({0})"
  comment-args="new Integer(@max)"
>
  <draw>
    @visualizer.updateArray(0, 0);
  </draw>
</finish>
```

5.3.3. Описание шагов алгоритма

Каждый шаг алгоритма, кроме вызова вложенного автомата (процедуры) имеет идентификатор, описание и уровень шага. Идентификатор шага должен быть корректным java-идентификатором, и быть уникальным в рамках процедуры.

Уровень шага задает, при шагах какого размера этот шаг отображается визуализатором. Значение уровня по умолчанию — 0, соответствует малым шагам.

Простой шаг (тег step)

Простой шаг может содержать одно или несколько присваиваний переменным модели.

DTD

```
<!ELEMENT step (draw?, (action | (direct, reverse?)))>
<!ATTLIST step
  id                  NMTOKEN     #IMPLIED
  description         CDATA       #REQUIRED
  level               CDATA       "0"
  comment-ru          CDATA       #IMPLIED
  comment-en          CDATA       #IMPLIED
  comment-args        CDATA       #IMPLIED
>
```

XML-схема

```
<xsd:complexType name="step">
  <xsd:sequence>
    <xsd:element
      name        = "draw"
      type        = "draw"
      minOccurs   = "0"
    />
    <xsd:choice>
      <xsd:element
        name    = "action"
        type    = "action"
      />
      <xsd:group ref="direct-reverse"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name = "id"          type = "id"/>
  <xsd:attribute name = "description" type = "description"/>
  <xsd:attribute name = "level" type = "level" default =
        "0"/>
  <xsd:attributeGroup ref="comment"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
id	1		Идентификатор шага
description	1		Описание шага
level	?		Уровень шага
comment	1	+	Комментарий к шагу
comment-args	?		Аргументы комментария

Содержимое

Вложенные элементы

Тег	#	Описание
draw	?	Метод отображения состояния
action	?	Автоматически обращаемое действие
direct	?	Действие, обращаемое в ручную
reverse	?	Ручное обращение действия

Элементы `direct` и `action` не могут использоваться одновременно. Элемент `direct` может сопровождаться элементом `reverse`, для указания ручного обращения действия.

Примеры

```
<step
  id="newMax"
  description="Обновление максимума"
  comment-ru="Обновляем текущий максимум"
  comment-en="Updating current maximum"
>
  <draw>
    @visualizer.updateArray(@i, 2);
  </draw>
  <direct>
    stack.pushInteger(@max);
    @max = @a[@i];
  </direct>
  <reverse>
    @max = stack.popInteger();
  </reverse>
</step>

<step
  id="newMax"
  description="Обновление максимума"
  comment-ru="Обновляем текущий максимум"
  comment-en="Updating current maximum"
>
  <draw>
    @visualizer.updateArray(@i, 2);
  </draw>
  <action>
```

```
    @max @= @a[@i];
  </action>
</step>
```

Описание действий (теги action, direct и reverse)

Теги `action`, `direct` и `reverse` служат для описания действий, выполняемых в простом шаге. В теге `action` описываются действия, которые будут обращены в автоматическом режиме (см. раздел 6.4.1). В тегах `direct` и `reverse` описываются действия, выполняемые на прямом и обратном проходе соответственно.

DTD

```
<!ELEMENT direct (#PCDATA)*>
<!ATTLIST direct>

<!ELEMENT reverse (#PCDATA)*>
<!ATTLIST reverse>

<!ELEMENT action (#PCDATA)*>
<!ATTLIST action>
```

XML-схема

```
<xsd:simpleType name="action">
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="direct">
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="reverse">
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>
```

Атрибуты

Нет

Содержимое

Текст — описание действий на языке *Java* (см. раздел 6.4.1).

Примеры

```
<action>
  @i @= @i + 1;
</action>

<direct>
```

```
  @i++;
</direct>

<reverse>
  @i--;
</reverse>
```

Вызов вложенного автомата (тег call-auto)

Тег `call-auto` служит для вызова вложенного автомата (процедуры).

DTD

```
<!ELEMENT call-auto EMPTY>
<!ATTLIST call-auto
  id                  IDREF       #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="call-auto">
  <xsd:attribute
    name    = "id"
    type    = "id"
    use     = "required"
  />
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
id	1		Идентификатор вызываемого автомата

Содержимое

Нет

Примеры

```
<call-auto id="factorial"/>
```

Оператор выбора (теги if, then, else)

Тег `if` служит для описания оператора ветвления. Он должен содержать элемент `then`, в котором описываются действия, выполняемые при истинности условия. Так же он может содержать элемент `else`, в котором описываются действия, выполняемые при ложности условия.

Проверяемое условие задается атрибутом `test`. Условие, проверяемое при обратном проходе (“обратное условие”) задается не обязательным атрибутом `rtest`.

DTD

```
<!ELEMENT if (draw?, then, else?)>
<!ATTLIST if
  id                  NMTOKEN     #IMPLIED
  description         CDATA       #REQUIRED
  test                CDATA       #REQUIRED
  rtest               CDATA       #IMPLIED
  level               CDATA       "0"
  true-comment-ru     CDATA       #IMPLIED
  true-comment-en     CDATA       #IMPLIED
  false-comment-ru    CDATA       #IMPLIED
  false-comment-en    CDATA       #IMPLIED
  comment-args        CDATA       #IMPLIED
>

<!ELEMENT then (%steps;)*>
<!ATTLIST then>

<!ELEMENT else (%steps;)*>
<!ATTLIST else>
```

XML-схема

```
<xsd:complexType name="if">
  <xsd:sequence>
    <xsd:element
      name        = "draw"
      type        = "draw"
      minOccurs   = "0"
    />
    <xsd:element
      name        = "then"
      type        = "then"
    />
    <xsd:element
      name        = "else"
      type        = "else"
      minOccurs   = "0"
    />
  </xsd:sequence>
  <xsd:attribute name = "id"          type = "id"/>
  <xsd:attribute name = "description" type = "description"/>
  <xsd:attribute name = "rtest"       type = "test"/>
  <xsd:attribute name = "test" type = "test" use =
        "required"/>
  <xsd:attribute name = "level" type = "level" default =
        "0"/>
  <xsd:attributeGroup ref="choice-comment"/>
</xsd:complexType>

<xsd:complexType name="then">
  <xsd:group
    ref         = "automata-steps"
```

```
    minOccurs   = "0"
    maxOccurs   = "unbounded"
  />
</xsd:complexType>

<xsd:complexType name="else">
  <xsd:group
    ref         = "automata-steps"
    minOccurs   = "0"
    maxOccurs   = "unbounded"
  />
</xsd:complexType>
```

Атрибуты

Для тега `if`:

Атрибут	#	И	Описание
`id`	1		Идентификатор шага
`description`	1		Описание шага
`test`	1		Условие
`rtest`	1		Обращенное условие
`level`	?		Уровень шага
`true-comment`	1	+	Комментарий, отображаемый, когда условие выполняется
`false-comment`	1	+	Комментарий, отображаемый, когда условие не выполняется
`comment-args`	?		Аргументы комментариев

У тегов `then` и `else` атрибуты отсутствуют.

Содержимое

Для тега `if`:

Тег	#	Описание
`draw`	?	Метод отображения состояния
`then`	1	Действия, выполняемые при истинности условия
`else`	?	Действия, выполняемые при ложности условия

Теги тегов `then` и `else` могут содержать описания произвольных шагов (`step`, `call-auto`, `if` и `while`).

Примеры

```
<if
  id="Cond"
  description="Условие"
  test="@max &lt; @a[@i]"
  true-comment-ru="{0} больше текущего максимума ({1})"
  true-comment-en="{0} greater than current maximum ({1})"
```

```
  false-comment-ru="{0} не больше текущего максимума ({1})"
  false-comment-en="{0} not greater than current maximum
        ({1})"
  comment-args="new Integer(@a[@i]), new Integer(@max)"
>
  <draw>visualizer.updateArray(@i, 1); draw>
  <then> ... </then>
  <else> ... </else>
</if>
```

Цикл с предусловием (тег while)

Тег `while` служит для описания циклов с предусловием. Проверяемое условие задается атрибутом `test`. Условие, проверяемое при обратном проходе (“обратное условие”) задается не обязательным атрибутом `rtest`.

DTD

```
<!ELEMENT while (draw?, (%steps;)+)>
<!ATTLIST while
  id                  NMTOKEN     #IMPLIED
  description         CDATA       #REQUIRED
  test                CDATA       #REQUIRED
  rtest               CDATA       #IMPLIED
  level               CDATA       "0"
  true-comment-ru     CDATA       #IMPLIED
  true-comment-en     CDATA       #IMPLIED
  false-comment-ru    CDATA       #IMPLIED
  false-comment-en    CDATA       #IMPLIED
  comment-args        CDATA       #IMPLIED
>
```

XML-схема

```
<xsd:complexType name="while">
  <xsd:group
    ref         = "automata-steps"
    maxOccurs   = "unbounded"
  />
  <xsd:attribute name = "id"            type = "id"/>
  <xsd:attribute name = "description" type = "description"/>
  <xsd:attribute name = "test" type="test" use="required"/>
  <xsd:attribute name = "rtest"       type = "test"/>
  <xsd:attribute name = "level" type = "level" default =
        "0"/>
  <xsd:attributeGroup ref="choice-comment"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
id	1		Идентификатор шага
description	1		Описание шага
test	1		Условие
rtest	1		Обращенное условие
level	?		Уровень шага
true-comment	1	+	Комментарий, отображаемый, когда условие выполняется
false-comment	1	+	Комментарий, отображаемый, когда условие не выполняется
comment-args	?		Аргументы комментариев

Содержимое

Тег	#	Описание
draw	?	Метод отображения состояния
steps	*	Действия, выполняемые при истинности условия

Пример

```
<while
  id="Loop"
  description="Цикл"
  test="@i &lt; @a.length"
  level="-1"
> ... </while>
```

Описание визуального представления (тег draw)

Тег `draw` служит для описания визуального представления шага (если оно требуется).

DTD

```
<!ELEMENT draw (#PCDATA)*>
<!ATTLIST draw>
```

XML-схема

```
<xsd:simpleType name="draw">
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>
```

Атрибуты

нет

Содержимое

Текст — описание действий по созданию визуального представления на языке *Java* (см. раздел 6.4.1).

Пример

```
<draw>
  @visualizer.updateArray(0, 0);
</draw>
```

5.4. Описание конфигурации визуализатора

Корневым элементом конфигурации является тег `configuration`. Конфигурация может содержать следующие элементы:

- Описание элементов интерфейса
  - Описание панели (тег `panel`)
  - Описание кнопки (тег `button`)
  - Описание новой панели выбора (тег `adjustablePanel`)
- Описания таблиц стилей
  - Описание стиля (тег `style`)
  - Описание таблицы стилей (тег `style-set`)
  - Описание шрифта (тег `font`)
  - Описание цвета (тег `color`)
- Описания групп, свойств и сообщений
  - Описание группы (тег `group`)
  - Описание свойства (тег `property`)
  - Описание сообщений (тег `message`)

Описание конфигурации визуализатора (тег configuration)

Тег `configuration` является контейнером для элементов, описывающих конфигурацию визуализатора.

DTD

```
<!ELEMENT configuration (%elements;)*>
<!ATTLIST configuration>
```

XML-схема

```
<xsd:complexType name="configuration">
  <xsd:group
    ref         = "elements"
    minOccurs   = "0"
    maxOccurs   = "unbounded"
```

```
  />
</xsd:complexType>

<xsd:group name="elements">
  <xsd:choice>
    <xsd:element name = "font"       type = "font"      />
    <xsd:element name = "color"      type = "color"     />
    <xsd:element name = "property"   type = "property"  />
    <xsd:element name = "message"    type = "message"   />
    <xsd:element name = "panel"      type = "panel"     />
    <xsd:element name = "style"      type = "style"     />
    <xsd:element name = "button"     type = "button"    />
    <xsd:element name = "styleset"   type = "styleset"  />
    <xsd:element name = "group"      type = "group"     />
    <xsd:element name = "choice"     type = "choice"    />
    <xsd:element name = "adjustablePanel" type = "
        adjustablePanel"/>
  </xsd:choice>
</xsd:group>
```

Атрибуты

Нет

Содержимое

Вложенные элементы: любые конфигурационные элементы

Пример

```
<configuration> ... </configuration>
```

5.4.1. Описание элементов управления

Описание панели (тег panel)

Тег `panel` предназначен для описания вида панелей (наследников класса `java.awt.Panel`).

DTD

```
<!ELEMENT panel (font)>
<!ATTLIST panel
  description         CDATA                    #IMPLIED
  param               NMTOKEN                  #REQUIRED
  foreground          CDATA                    #REQUIRED
  background          CDATA                    #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="panel">
  <xsd:sequence>
    <xsd:element name = "font" type = "font"/>
```

```
    </xsd:sequence>
    <xsd:attribute name = "description"     type =
          "xsd:string"/>
    <xsd:attribute name = "param"           type = "xsd:string"
        use = "required"/>
    <xsd:attribute name = "foreground"      type = "color-type"
        use = "required"/>
    <xsd:attribute name = "background"      type = "color-type"
        use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	Описание
description	?	Описание конфигурируемой панели
param	1	Имя параметра для записи конфигурации
foreground	1	Основной цвет (цвет надписей)
background	1	Цвет фона

Содержимое

Тег	#	Описание
font	1	Шрифт надписей

Пример

```
<panel
  description = "Конфигурация клиенсткой области"
  param       = "client"
  foreground  = "000000"
  background  = "ffffff"
>
  <font face="Serif" size="14" style="plain"/>
</panel>
```

Описание кнопки (тег button)

Тег `button` предназначен для описания кнопок (наследников класса `java.awt.Button`).

DTD

```
<!ELEMENT button EMPTY>
<!ATTLIST button
  description         CDATA                   #IMPLIED
  param               NMTOKEN                 #REQUIRED
  caption-ru          CDATA                   #REQUIRED
  caption-en          CDATA                   #REQUIRED
  hint-ru             CDATA                   #REQUIRED
  hint-en             CDATA                   #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="button">
  <xsd:attribute name = "description"     type =
        "xsd:string"/>
  <xsd:attribute name = "param"           type = "xsd:string"
      use = "required"/>
  <xsd:attribute name = "caption-ru"      type = "xsd:string"
      use = "required"/>
  <xsd:attribute name = "caption-en"      type = "xsd:string"
      use = "required"/>
  <xsd:attribute name = "hint-ru"         type = "xsd:string"
      use = "required"/>
  <xsd:attribute name = "hint-en"         type = "xsd:string"
      use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание конфигурируемой кнопки
param	1		Имя параметра для записи конфигурации
caption	1	+	Надпись на кнопке
hint	1	+	Подсказка для кнопки

Содержимое

Нет

Пример

```
<button
  description = "Next-button"
  param       = "next"
  caption-ru  = ">>"
  caption-en  = ">>"
  hint-ru     = "Шаг вперед"
  hint-en     = "Step forward"
/>
```

Описание новой панели выбора (тег adjustablePanel)

Тег `adjustablePanel` описывает конфигурацию панели выбора. Для дополнительной информации смотри раздел 6.3.3.

DTD

```
<!ELEMENT adjustablePanel (button, button)>
<!ATTLIST adjustablePanel
  description         CDATA                  #IMPLIED
  param               NMTOKEN                #REQUIRED
  caption-ru          CDATA                  #REQUIRED
  caption-en          CDATA                  #REQUIRED
  hint-ru             CDATA                  #REQUIRED
  hint-en             CDATA                  #REQUIRED
```

```
  value                CDATA                    #REQUIRED
  maximum              CDATA                    #REQUIRED
  minimum              CDATA                    #REQUIRED
  unitIncrement        CDATA                    #REQUIRED
  blockIncrement       CDATA                    #REQUIRED
  blockInterval        CDATA                    #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="adjustablePanel">
  <xsd:sequence>
    <xsd:element
      name        = "button"
      type        = "button"
      minOccurs   = "2"
      maxOccurs   = "2"
    />
  </xsd:sequence>
  <xsd:attribute name = "description"     type =
        "xsd:string"/>
  <xsd:attribute name = "param"           type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "caption-ru"      type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "caption-en"      type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "hint-ru"         type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "hint-en"         type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "value"           type = "xsd:double"
        use = "required"/>
  <xsd:attribute name = "maximum"         type = "xsd:double"
        use = "required"/>
  <xsd:attribute name = "minimum"         type = "xsd:double"
        use = "required"/>
  <xsd:attribute name = "unitIncrement"   type = "xsd:double"
        use = "required"/>
  <xsd:attribute name = "blockIncrement"  type = "xsd:double"
        use = "required"/>
  <xsd:attribute name = "blockInterval"   type = "xsd:double"
        use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание конфигурируемой панели
param	1		Имя параметра для записи конфигурации
caption	1	+	Шаблон надписи на панели
hint	1	+	Подсказка для панели
value	1		Начальное значение
minimum	1		Минимальное значение
maximum	1		Максимальное значение
unitIncrement	1		Маленький шаг
blockIncrement	1		Большой шаг
blockInterval	1		Когда делать большой шаг

Большой шаг используется когда между двумя нажатиями на кнопку увеличения (уменьшения) значения проходим меньше `blockInterval` миллисекунд, в противном случае используется маленький шаг.

Содержимое

Тег	#	Описание
button	2	Конфигурация кнопок увеличения/уменьшения значений.

Пример

```
<adjustablePanel
  description     = "Delay tuning panel"
  param           = "delay"
  caption-ru      = "Задержка: {0,number,####}"
  caption-en      = "Delay: {0,number,####}"
  hint-ru  = "Задержка между шагами\nв автоматическом режиме"
  hint-en         = "Delay between steps\nin automated mode"
  value           = "1000"
  minimum         = "100"
  maximum         = "5000"
  unitIncrement   = "100"
  blockIncrement  = "300"
  blockInterval   = "500"
>
  <button
    description = "Button for decreasing delay"
    param       = "decrementButton"
    caption-ru  = "&lt;&lt;" hint-ru     = "Уменьшить
        задержку"
    caption-en  = "&lt;&lt;" hint-en     = "Decrease delay"
  />
  <button
    description = "Button for increasing delay"
    param       = "incrementButton"
    caption-ru  = "&gt;&gt;" hint-ru     = "Увеличить
        задержку"
```

```
    caption-en  = "&gt;&gt;" hint-en     = "Increase delay"
  />
</adjustablePanel>
```

5.4.2. Таблицы стилей

Описание стиля (тег style)

DTD

```
<!ELEMENT style (font?)>
<!ATTLIST style
  description         CDATA                    #IMPLIED
  param               NMTOKEN                  #IMPLIED
  text-color          CDATA                    #IMPLIED
  text-align          CDATA                    #IMPLIED
  message-align       CDATA                    #IMPLIED
  border-color        CDATA                    #IMPLIED
  border-status       %boolean;                #IMPLIED
  fill-color          CDATA                    #IMPLIED
  fill-status         %boolean;                #IMPLIED
  aspect              CDATA                    #IMPLIED
  aspect-status       %boolean;                #IMPLIED
  padding             CDATA                    #IMPLIED
>
```

XML-схема

```
<xsd:complexType name="style">
  <xsd:sequence>
    <xsd:element name = "font" type = "font"
        minOccurs = "0"/>
  </xsd:sequence>
  <xsd:attribute name="description"     type="xsd:string"/>
  <xsd:attribute name="param"           type="xsd:string"/>
  <xsd:attribute name="text-color"      type="color-type"/>
  <xsd:attribute name="text-align"      type="align"/>
  <xsd:attribute name="message-align"   type="align"/>
  <xsd:attribute name="border-color"    type="color-type"/>
  <xsd:attribute name="border-status"   type="xsd:boolean"/>
  <xsd:attribute name="fill-color"      type="color-type"/>
  <xsd:attribute name = "fill-status"   type="xsd:boolean"/>
  <xsd:attribute name = "aspect"
        type = "positiveRealNumber"/>
  <xsd:attribute name = "aspect-status"
        type = "xsd:boolean"/>
  <xsd:attribute name = "padding"
        type = "positiveRealNumber"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание стиля
param	1		Имя параметра для записи конфигурации
text-color	?	+	Цвет текста
text-align	?	+	Выравнивание текста в сообщении
message-align	?		Выравнивание сообщения относительно фигуры
border-color	?		Цвет рамки
border-status	?		Отображать ли рамку
fill-color	?		Цвет фона
fill-status	?		Отображать ли фон
aspect	?		Отношение длины к ширине
aspect-status	?		Использовать ли отношение длины к ширине
padding	?		Отступ от границы до теста (в пикселях)

Для подробностей см. раздел 6.3.2.

Содержимое

Тег	#	Описание
font	1	Шрифт надписей

Пример

```
<style
  description     = "Обычная ячейка"
  text-color      = "000000"
  text-align      = "0.5"
  border-color    = "000000"
  border-status   = "true"
  fill-color      = "8080ff"
  fill-status     = "true"
  aspect-status   = "false"
  padding         = "0.2"
>
  <font face="Serif" size="12" style="plain"/>
</style>
```

Описание таблицы стилей (style-set)

Таблица стилей задает набор стилей (тег `style`), используемых для отображения одного элемента в зависимости от состояния. Если в стиле не определены некоторые свойства, то их значения берутся из описания первого стиля в наборе. Стили нумеруются, начиная с 0.

DTD

```
<!ELEMENT styleset (style*)>
<!ATTLIST styleset
  description         CDATA                    #IMPLIED
```

```
  param                NMTOKEN                    #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="styleset">
  <xsd:sequence>
    <xsd:element
      name        = "style"
      type        = "styleInStyleset"
      minOccurs   = "0"
      maxOccurs   = "unbounded"
    />
  </xsd:sequence>
  <xsd:attribute name = "description"     type =
       "xsd:string"/>
  <xsd:attribute name = "param"           type = "xsd:string"
      use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание таблицы стилей
param	1		Имя параметра для записи конфигурации

Содержимое

Тег	#	Описание
style	1	Стили в таблице

Пример

```
<styleset
  description = "Таблица стилей для ячеек массива"
  param       = "array"
> ... </styleset>
```

Описание шрифта (тег font)

Тег `font` служит для описания типа и размера шрифта, обычно он вложен в описание других элементов.

DTD

```
<!ELEMENT font EMPTY>
<!ATTLIST font
  description         CDATA                  #IMPLIED
  param               NMTOKEN                "font"
  face                %fonts;                #IMPLIED
  size                CDATA                  #IMPLIED
  style               %font-style;           #IMPLIED
>
```

XLM-схема

```
<xsd:complexType name="font">
  <xsd:attribute name = "description" type = "xsd:string"/>
  <xsd:attribute name = "param"       type = "xsd:string"
        default="font"/>
  <xsd:attribute name = "face"        type = "font-name"/>
  <xsd:attribute name = "size"        type =
        "xsd:positiveInteger"/>
  <xsd:attribute name = "style"       type = "font-style"/>
</xsd:complexType>

<xsd:simpleType name="font-name">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Serif"/>
    <xsd:enumeration value="SansSerif"/>
    <xsd:enumeration value="Symbol"/>
    <xsd:enumeration value="Monospaced"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="font-style">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="plain"/>
    <xsd:enumeration value="bold"/>
    <xsd:enumeration value="italic"/>
    <xsd:enumeration value="bolditalic"/>
  </xsd:restriction>
</xsd:simpleType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание шрифта
param	1		Имя параметра для записи конфигурации
face	1		Начертание шрифта
size	1		Размер шрифта в пунктах
style	1		Стиль шрифта

Определены следующие начертания шрифтов:

6. `Serif` — с засечками (Times New Roman, Roman).
7. `SansSerif` — без засечек (Arial, Helvetica).
8. `Symbol` — для специальных символов (Symbol).
9. `Monospaced` — моноширинной (Courier New, Courier New).

Для стиля шрифта определены следующий значения:

10. `Plain` — обычное начертание.
11. `Bold` — полужирное начертание.
12. `Italic` — курсивное начертание.

13. `BoldItalic` — полужирный курсив.

Содержимое

Нет

Пример

```
<font face="Serif" size="12" style="plain"/>
```

Описание цвета (тег color)

Тег `color` служит для описания цветов элементов, обычно он вложен в описание других элементов. Цвет представляется в виде 6-значного шестнадцатеричного числа: `RRGGBB`.

DTD

```
<!ELEMENT color EMPTY>
<!ATTLIST color
  description         CDATA                    #IMPLIED
  param               NMTOKEN                  #REQUIRED
  value               CDATA                    #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="color">
  <xsd:attribute name = "description" type = "xsd:string"/>
  <xsd:attribute name = "param"       type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "value"       type = "color-type"
        use = "required"/>
</xsd:complexType>

<xsd:simpleType name="color-type">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9a-fA-F]{6}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Атрибуты

Атрибут	#	И	Описание
`description`	?		Описание цвета
`param`	1		Имя параметра для записи конфигурации
`value`	1		Цвет

Содержимое

Нет

Пример

```
<color param="foreground" value="000000"/>
```

5.4.3. Группы, свойства и сообщения

Группы служат для объединения свойств и сообщений, имеющих общий смысл, на пример сообщения, отображаемые одним элементом управления. Так же группы могут использоваться для конфигурирования нестандартных компонент.

Описание группы (тег group)

DTD

```
<!ELEMENT group (%elements;)*>
<!ATTLIST group
  description         CDATA                    #IMPLIED
  param               NMTOKEN                  #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="group">
  <xsd:group
    ref         = "elements"
    minOccurs   = "0"
    maxOccurs   = "unbounded"
  />
  <xsd:attribute name = "description" type = "xsd:string"/>
  <xsd:attribute name = "param"       type = "xsd:string"
        use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание группы
param	1		Имя параметра для записи элементов группы

Содержимое

Вложенные элементы: любые конфигурационные элементы

Пример

```
<group
  description = "Конфигурация подсказок"
  param       = "hint"
> ... </group>
```

Описание свойства (тег property)

Свойства описывают нестандартные конфигурационные компоненты, не зависящие от языка

DTD

```
<!ELEMENT property EMPTY>
<!ATTLIST property
  description         CDATA                    #IMPLIED
  param               NMTOKEN                  #REQUIRED
  value               CDATA                    #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="property">
  <xsd:attribute name = "description" type = "xsd:string"/>
  <xsd:attribute name = "param"       type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "value"       type = "xsd:string"
        use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание свойства
param	1		Имя параметра для записи свойства
value	1		Значение свойства

Содержимое

Нет

Пример

```
<property
  description = "Вертикальное расстояние от текста до
        границы"
  param       = "vgap"
  value       = "-1"
/>
```

Описание сообщений (тег message)

Тег `message` служит для описания сообщений, выводимых пользователю. Обычно сообщения объединяются в группы.

DTD

```
<!ELEMENT message EMPTY>
<!ATTLIST message
  description         CDATA                    #IMPLIED
```

```
  param                NMTOKEN                  #REQUIRED
  message-ru           CDATA                    #REQUIRED
  message-en           CDATA                    #REQUIRED
>
```

XML-схема

```
<xsd:complexType name="message">
  <xsd:attribute name = "description" type = "xsd:string"/>
  <xsd:attribute name = "param"       type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "message-ru"  type = "xsd:string"
        use = "required"/>
  <xsd:attribute name = "message-en"  type = "xsd:string"
        use = "required"/>
</xsd:complexType>
```

Атрибуты

Атрибут	#	И	Описание
description	?		Описание свойства
param	1		Имя параметра для записи свойства
message	1	+	Шаблон сообщения

Содержимое

Нет

Пример

```
<message
  description = "Заголовок окна"
  param       = "title"
  message-ru  = "О визуализаторе"
  message-en  = "About this visualizer"
/>
```

5.5. Выводы

Приведенный XML-формат описания визуализаторов позволяет описывать как логики визуализатора (задаваемую визуализируемой программы), так и его конфигурацию.

При этом в XML-описании визуализируемой программы дополнительно указываются комментарии, к каждому интересному состоянию, а так же как визуализировать каждое интересное состояние. Таким образом первый этап интеграции логики визуализатора, визуального представления и набора комментариев осуществляется уже при записи XML-описания. Это уменьшает

количество ошибок по сравнению с раздельным кодирование логики и остальных частей визуализатора.

ГЛАВА 6. ОПИСАНИЕ СИСТЕМЫ ВИЗУАЛИЗАЦИИ VIZI

В данной главе описывается система визуализации *Vizi* и ее применение при программировании визуализаторов.

Глава начинается с рассмотрения структуры системы визуализации *Vizi* и описания назначения ее основных частей (раздел 6.1). Из которых две используются для построения визуализатора, а третья — обеспечивает его работу при запуске пользователем.

Далее рассматривается структура проекта визуализатора, выполняемого при помощи системы визуализации *Vizi* (раздел 6.2). Описываемая структура непосредственно поддерживается *Vizi*, что позволяет легко переключаться между разработками различных визулизаторов.

В разделе 6.3 описываются средства, предоставляемые визуализаторам java-библиотекой входящей в *Vizi*. В частности, показывается как с ее помощью можно облегчить создание визуального представления и элементов управления визуализатора.

Генерация исходного кода логики визуализатора и ее отладка рассматриваются в разделе 6.4. Сначала описывается @-нотация используемая для обращения к переменным модели, а затем — средства отладки как кода визуализируемой программы, так и построенной системы взаимодействующих автоматов.

В заключении главы рассматривается использование системы визуализации *Vizi* на кафедре Компьютерных технологий СПбГУ ИТМО, в частности приводится список визуализаторов, выполненных на ее основе за последний год (раздел 6.5).

6.1. Структура системы визуализации Vizi

Система визуализации *Vizi* состоит из двух основных частей:

1. сценарная часть;
2. *build*-скрипт;
3. *Java*-библиотека.

Сценарная часть осуществляет построение системы взаимодействующих автоматов по XML-описанию визуализируемой программы. *Build*-скрипт осуществляет построение визуализатора в целом. *Java*-библиотека используется при исполнении визуализатора.

6.1.1. Сценарная часть

Сценарная часть отвечает за построение логики визуализатора, интеграцию визуального представления и набора комментариев, а так же компиляцию проекта.

Построение логики визуализатора состоит в преобразовании XML-описания визуализатора в систему взаимодействующих автоматов и построение по ней исходного кода. Для этого в начале программа рассматривается как отдельный набор процедур, по каждой из которых строится прямой и обратные автоматы. После этого автоматы для отдельных процедур связываются друг с другом, с получением системы взаимодействующих автоматов.

Более строго. Для программы в целом осуществляется:

1. Разбиение на отдельные процедуры.
2. Преобразование каждой процедуре в пару автоматов.
3. Объединение автоматов в систему взаимодействующих автоматов.
4. Запись кода системы автоматов.

Для каждой процедуры на втором шаге производятся следующие действия:

1. Разбиение на отдельные операторы.
2. Кодирование состояний.
3. Преобразование операторов с получением фрагментов автомата
4. Объединение фрагментов автомата в автомат.

Заметим, что в полученную систему автоматов входят средства для получения комментариев и интеграции с визуальным представлением.

Полученная таким образом *java*-реализация системы взаимодействующих автоматов, соответствующая визуализируемой программы компилируется *build*-скриптом и окончательно интегрируется с другими частями визуализатора.

6.1.2. Build-скрипт

Build-скрипт написан на языке Apache Ant [39], построенном на основе XML. Запуск *build*-скрипта осуществляется командным файлом `ant.bat` (под Microsoft Windows) или `ant.sh` (под Unix/Linux), расположенном в корневом каталоге поставки системы визуализации *Vizi*. При запуске *build*-скрипта может быть указана одна из целей, описанных ниже. Если цель не указана, используется цель `all`.

Описание файла build.properties

Build-скрипт использует информацию записанную в файле `build.properties`, в котором указываются следующие параметры:

- `project` — путь к файлу описания проекта;
- `build` — каталог для временных файлов;
- `target` — каталог для файлов визуализатора;
- `docs` — каталог для описания программного интерфейса системы визуализации *Vizi* (см. описание цели `api-docs`);
- `description` — каталог описания параметров конфигурации визуализатора (см. описание цели `description`).

Файл `build.properties` имеет формат *property*-файла (см. [40]). Относительные пути, указанные в этом файле отсчитываются от корневого каталога поставки системы визуализации *Vizi*.

Путь к файлу с описание проекта определяет текущий проект. При изменении этого пути при вызове любой цели осуществляется очистка временных файлов. Таким образом, при переходе к разработке другого визуализатора действия по очистки можно не выполнять.

В каталоге для временных файлов помещаются промежуточные файлы, создаваемые при построении визуализатора. Структура этого каталога подробно рассматривается в разделе 6.2.2. По умолчанию, каталог временных файлов называется `build`.

В каталоге файлов визуализатора помещаются результаты выполнения *build*-скрипта. Структура этого каталога описана в разделе 6.2.3. По умолчанию, этот каталог называется `deploy`.

Цели определенные build-скриптом

В *build*-скрипте определены следующие цели:

- `all` — осуществляет полное построение визуализатора. В частности, создает *jar*-, *cmd*- и *html*-файлы для визуализатора (см. описания соответствующих целей). Данная цель вызывается по умолчанию.
- `jars` — генерирует исходные коды автоматов по XML-описанию; компилирует визуализатор; выделяет комментарии из XML-описания и упаковывает результаты свое работы в *jar*-файл визуализатора.
- `cmds` — создает командные файлы для запуска визуализатора.
- `htmls` — создает HTML-файлы для запуска визуализатора. В создаваемые файлы помещается краткая информация об авторе визуализатора.
- `description` — создает описание параметров конфигурации визуализатора и записывает его в соответствующую директорию.
- `debug` — создает файл с визуализируемой программой для отладки (см. раздел 6.4.2).
- `check` — создает командный файл для автоматической проверки автомата (см. раздел 6.4.2).

- `api-docs` — создает описание программного интерфейса системы визуализации *Vizi*, компилируя *JavaDoc*-комментарии исходных текстов.
- `vizi` — перестраивает систему визуализации *Vizi*. Используется для модернизации системы и ее отладки.
- `clean` — удаляет файлы, созданные предыдущими запусками *build*-скрипта (в том числе, файлы из каталога визуализатора).
- `help` — выводит краткое сообщение о назначении целей *build*-файла.

Build-файл является гибким и мощным средством управления проектом визуализатора.

6.1.3. Java-библиотека

Java-библиотека используется при работе визуализатора. Она решает следующие задачи:

1. Запуск визуализатора в виде приложения либо апплета.
2. Конструирование интерфейса визуализатора.
3. Создание основных элементов управления.
4. Управление системой взаимодействующих автоматов.
5. Отображение комментариев.
6. Связь логики визуализатора и визуального представления.
7. Предоставление визуализатору расширенных элементов управления.
8. Предоставление визуализатору расширенных средств для визуального представления.

В библиотеку входят следующие пакеты:

ru.ifmo.vizi.base	Основной пакет
ru.ifmo.vizi.base.auto	Пакет поддержки систем взаимосвязанных автоматов
ru.ifmo.vizi.base.timer	Пакет работы со временем

ru.ifmo.vizi.base.ui	Пакет расширенных элементов управления
ru.ifmo.vizi.base.widgets	Пакет расширенных средств визуального представления

Основной пакет содержит классы отвечающие за загрузку и конфигурацию визуализатора. В нем же находятся вспомогательные классы.

Пакет поддержки систем взаимосвязанных автоматов содержит интерфейсы и базовые классы для реализации конечных автоматов. В этом пакете так же находится валидатор автоматов (см. раздел 6.4.2).

Пакет работы со временем расширяет стандартные методы работы со временем и позволяет генерировать события (в том числе и повторяющиеся) через заданные интервалы времени. В частности, пакет содержит заменителя класса `java.swing.Timer`, не входящего в Java 1.1.8.

Пакет расширенных элементов управления содержит элементы управления облегчающие построения визуализатора. Классы, содержащиеся в этом пакете рассмотрены в разделе 6.3.3.

Пакет расширенных средств визуального представления содержит классы дополняющие таковы средства языка *Java* и исправляющие ошибки допущенные в них (в частности, некорректное отображение окружностей). Классы, содержащиеся в этом пакете рассмотрены в разделе 6.3.2.

Таким образом библиотека является “скелетом” визуализатора, обеспечивающим его работу.

6.2. Структура проекта визуализатора

Проект визуализатора состоит из файла описания проекта, XML-описания визуализатора и исходных кодов визуализатора. Место нахождения двух последних указывается в файле описания проекта (см. раздел 6.2.1).

При построении проекта визуализатора с использованием *build*-скрипта так же создаются каталог временных файлов и каталог визуализатора, описанные в разделах 6.2.2 и 6.2.3 соответственно.

6.2.1. Файл описания проекта

В файле описания проекта указывается общая информация о визуализаторе и пути к его исходным файлам. Путь к файлу описания указывается в файле `build.properties` (см. раздел 6.1.2).

Файл описания проекта имеет формат *property*-файла (см. [40]). Относительные пути, указанные в этом файле отсчитываются от каталога, указанного в параметре `dir`.

Параметры файла описания проекта:

- `vizi.version` — версия используемой системы визуализации *Vizi*. При не совпадении версии системы и указанной либо используется режим обратной совместимости либо выдается сообщение об ошибке.
- `dir` — директория проекта.
- `file` — файл, содержащий XML-описание визуализатора.
- `src` — каталог, содержащий исходные коды визуализатора.

При построении проекта в каталог указанный в параметре `src` так же помещаются сгенерированные исходные коды автоматов.

6.2.2. Структура каталога временных файлов

В каталоге временных файлов сохраняются промежуточные файлы, используемые при построении визуализатора. Этот каталог имеет достаточно сложную структуру и содержит три основных подкаталога: `jars`, `properties` и `temp`.

В подкаталог `jars` помещаются преобразованные *property*-файлы, которые затем упаковываются в *jar*-файл визуализатора.

Подкаталог `properties` содержит *property*-файлы с конфигурацией визуализатора (файлы `Configuration.properties` и `Localization_??.properties`), а так же файлы комментариев к состояниям визуализатора (`Comments_??.properties`).

В подкаталоге temp содержаться временные файлы генерируемые *XSLT*-скриптами при построении исходного кода автоматов по XML-описанию и выделении автоматов.

6.2.3. Структура каталога визуализатора

Каталог визуализатора служит для хранения файлов построенного визуализатора. Он является отчуждаемой частью проекта визуализатора и может быть перенесен с машины на машину или опубликован в сети *Internet*, отдельно от остальных частей проекта.

После построения визуализатора каталог визуализатора содержит следующие файлы:

- *ИмяВизуализатора*`.jar` — jar-файл визуализатора;
- *ИмяВизуализатора*`_??.cmd` — командные файлы для запуска визуализатора;
- *ИмяВизуализатора*`_??.html` — *HTML*-файлы для запуска визуализатора;
- `vizi-`*Версия*`.jar` — jar-файл системы визуализации *Vizi*.

Первые три типа файлов создаются целями `jars`, `cmds` и `htmls` соответственно (см. раздел 6.1.2). *Jar*-файл системы визуализации *Vizi* копируется из каталога `meta`.

6.3. Средства предоставляемые системой визуализации Vizi

Система визуализации *Vizi* предоставляет средства для облегчения создания визуального представления.

Единый интерфейс визуализатора (раздел 6.3.1) облегчает использование визуализатора пользователем, за счет того что визуализаторы одинаково выглядят, работают и имеют схожие элементы управления.

Средства визуального представления (раздел 6.3.2) облегчают создание схем, поясняющих работу алгоритма.

Элементы управления (раздел 6.3.3) входящие в состав систему визуализации *Vizi* облегчают как создание визуализатора (так как позволяют не задумываться о деталях реализации всплывающих подсказок и взаимодействии с автоматами) так и использование визуализатора.

Встроенная поддержка комментариев (раздел 6.3.4) позволяет определять комментарии вместе с комментируемым кодом в XML-описании и отображать их в соответствующей области единого интерфейса.

6.3.1. Единый интерфейс визуализатора

Система визуализации *Vizi* разработана для облегчения создания визуализаторов имеющих единый (общий) интерфейс. Единый интерфейс построен на концепции Модель-Вид-Контроллер (Model-View-Controller). При этом моделью является система взаимодействующих автоматов, построенная по визуализируемой программе и выделенная из нее модель данных. Видом является совокупность визуальное представление визуализатора, комментариев и элементов управления. А контроллером является java-библиотека, входящая в систему визуализации *Vizi* (см. раздел 6.1.3).

Описание окна визуализатора

При этом окно визуализатора разделено на три области:

- область визуального представления;
- область комментариев;
- область элементов управления.

На рис. 40 изображено разделение на области для визуализатора алгоритма поиска максимума в массиве.

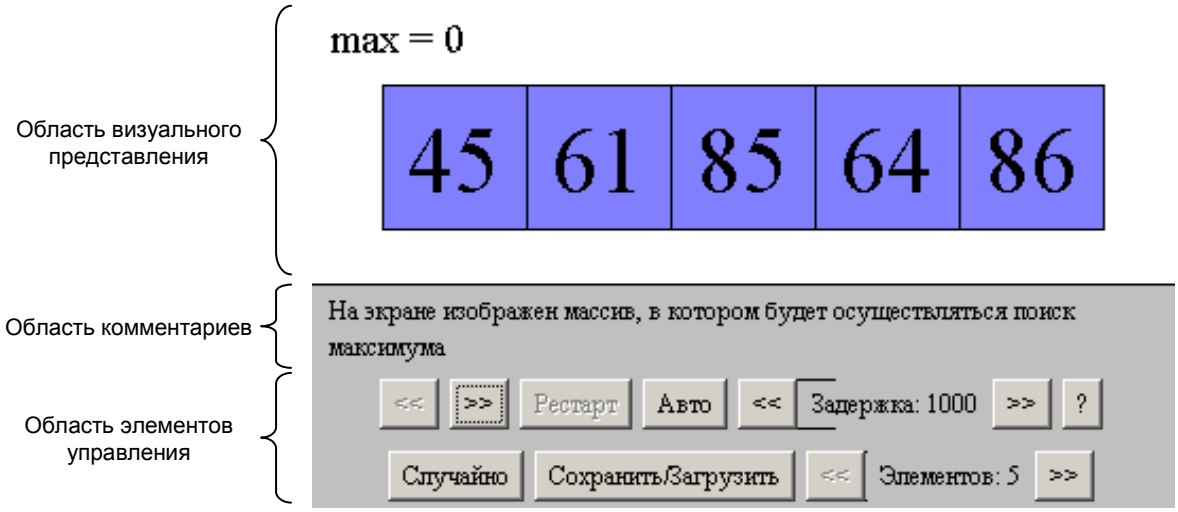


Рис. 40. Единый интерфейс визуализатора

Область визуального представления служит для отображения рисунков и схем облегчающих понимание алгоритмов. В этой области могут быть использованы средства визуального представления описанные в разделе 6.3.2.

В *области комментариев* отображается комментарий к текущему шагу алгоритма. Для отображения комментариев требуется задать их в XML-описании визуализатора, после чего они будут подставляться автоматически.

Область элементов управления состоит из нескольких частей. При этом, одной из частей является панель управления визуализатором (первая строка на рис. 40), которая отвечает за основные операции с визуализатором.

Элементы управления

В единый интерфейс так же входят основные элементы управления визуализатором:

<<< и >>>	Большой шаг назад и вперед (могут отсутствовать)
<< и >>	Шаг назад и вперед
Рестарт	Начало визуализации с начала
Авто и Стоп	Вход в автоматический режим и выход из него
<< Задержка: 1000 >>	Регулирование задержки между шагами в

автоматическом режиме. Задержка измеряется в миллисекундах

 Вывод информации о визуализаторе

Некоторым элементам управления присвоены горячие клавиши:

- Для кнопок шагов вперед и назад — клавиши стрелок вправо и влево соответственно.
- Для кнопок больших шагов вперед и назад — *Shift* плюс клавиши стрелки вправо и влево.
- Для кнопки вывода информации о визуализаторе — клавиша *F1*.

Требования к интерфейсу визуализатора

Для визуализаторов использующих единый интерфейс так же должны выполняться следующие требования:

1. Интерфейс должен быть понятен неподготовленному пользователю.
2. Все элементы управления должны иметь всплывающие подсказки, облегчающие использование визуализатора.
3. Нажатие на активную (enabled) кнопку, выбор из активного списка и аналогичные действия пользователя должны сопровождаться видимыми изменениями состояния визуализатора.
4. В интерфейсе не должно быть зависимостей вида “Пользователь не может изменить количество вершин, пока не нажмет кнопку *Restart*”.

6.3.2. Средства визуального представления

В систему визуализации *Vizi* входят средства визуального представления, облегчающие создание интерфейса визуализатора. Соответствующие классы определены в пакете `ru.ifmo.vizi.base.widgets`.

Визуальное представление основано на понятии *фигуры*. На данный момент в систему визуализации *Vizi* входят прямоугольники и эллипсы. Для каждой фигуры задается набор стилей, определяющих как она отображается. Выбор текущего стиля из таблицы осуществляется методом `setStyle`.

Каждая фигура может содержать некоторое сообщение (в том числе, многострочное). Сообщения устанавливаются методом `setMessage`.

Фигуры могут подбирать или размер шрифта по размеру фигуры (метод `adjustFontSize`) или размер фигуры по шрифту (метод `adjustSize`). При этом размеры подбираются так, чтобы сообщение полностью заполняло фигуру.

Стили

Система стилей является двухуровневой. На первом уровне находятся статические стили определенные в конфигурации (класс `ShapeStyle`). На втором уровне находятся стили, создаваемые динамически (класс `ShapeLook`).

Динамические стили создаются модификацией статических и могут переопределять их свойства. Это позволяет создавать несколько похожих стилей. Например стиль выделенного элемента может отличаться тем, что в нем используется полужирный шрифт.

Для стилей определены следующие параметры (так же см. раздел 5.4.2):

Атрибут	Описание
`textColor`	Цвет текста
`textAlign`	Выравнивание текста в сообщении
`messageAlign`	Выравнивание сообщения относительно фигуры
`borderColor`	Цвет рамки
`borderStatus`	Отображать ли рамку
`fillColor`	Цвет фона
`fillStatus`	Отображать ли фон
`aspect`	Отношение длины к ширине
`aspectStatus`	Использовать ли отношение длины к ширине
`padding`	Отступ от границы до теста (в пикселях)

Выравнивание текста сообщения и сообщения задаются числом в диапазоне 0..1. При этом, если выравнивание равно *a*, то свободное место будет распределено на лево и на право в пропорции *a:(1-a)*. Таким образом:

- 0 — выравнивание по левому краю;
- 0.5 — выравниванию по центру;
- 1 — выравнивание по правому краю.

При этом, если параметр стиля определен в динамическом стиле, то используется установленное значение, в противном случае используется значение из статического стиля.

При загрузке статического стиля из конфигурации все его параметры должны быть определены, в противном случае выдается ошибка. Так же, возможна загрузка стилей с использованием стиля-шаблона. При этом, параметры неопределенные в конфигурации берутся из шаблона.

Стандартные фигуры

Прямоугольник

Прямоугольник определен в классе `Rect` и как следует из его названия служит для отображения прямоугольников.

У прямоугольников могут быть скруглены углы. Радиус скругления задается методом `setRadius`. Для отображения скруглений используется реализация алгоритма Брезенхема (см. ниже).

Эллипс

Эллипс определен в классе `Ellipse`. К сожалению, *Java* версий *1.1.x* содержит ошибку, в следствие чего эллипс не может быть отображен стандартными методами. По этому, для отображения эллипсов используется реализация алгоритма Брезенхема, определенная в классе `Bresenham`.

6.3.3. Элементы управления

Стандартные элементы управления позволяют концентрироваться не вопросе “Как же сделать кнопку с подсказкой?”, а на вопросах непосредственно связанных с удобством интерфейса для пользователя.

Элементы управления, входящие в состав *Vzi* находятся в пакете `ru.ifmo.vizi.base.ui`. В документации к этому пакету (построенной с

использование *JavaDoc* или цели `api-docs` *build*-скрипта) для каждого класса указываются возможные его настройки и элементы конфигурации, используемые им.

Простые элементы управления

Простые элементы управления либо не имеют конфигурации либо их конфигурация состоит из одного элемента.

Отображатель подсказок (Hinter)

Отображатель подсказок служит для отображения всплывающих подсказок (*hints*). Он является *AWT*-контейнером (`java.awt.Container`). Для отображения подсказки элемент управления должен быть помещен (непосредственно или через другие компоненты) в отображатель подсказок. После этого подсказку можно установить вызовом статического метода `applyHint`.

Всплывающие подсказки отображаются если пользователь наводит курсор мыши на элемент управления и не двигает его некоторое время. При передвижении мыши подсказка убирается с экрана.

Конфигурация отображателя подсказок задается группой (см. 5.4.3) в которой должны быть определены следующие элементы, определяющие свойства всплывающих подсказок:

Параметр	Тип	Описание
`font`	`font`	Шрифт всплывающих подсказок
`foreground`	`color`	Цвет текста
`background`	`color`	Цвет подсказки
`vgap`	`property`	Вертикальное расстояние от текста до рамки
`hgap`	`property`	Горизонтальное расстояние от текста до рамки
`delay`	`property`	Задержка перед появление подсказки

Кнопка с подсказкой (HintedButton)

Кнопки с подсказкой являются одним из основных элементов управления. Они описываются элементом `button` (см. раздел 5.4.1). Из конфигурации загружается как надпись на кнопке так и всплывающая подсказка.

Ранее кнопки с подсказкой предоставляли непосредственную возможность определять действие осуществляемое при нажатии путем переопределения метода `click`. В настоящее время рекомендуется осуществлять такое определение посредством слушателей (`listeners`).

Кнопка с состояниями (MultiButton)

Кнопка с состояниями является расширением кнопки с подсказкой. В зависимости от состояния кнопки она может иметь различные названия и всплывающие подсказки. Например кнопка *Авто/Стоп* из единого интерфейса визуализатора является кнопкой с состояниями.

При создании кнопки с состояниями ей передается название узлов конфигурации описывающих состояния кнопки.

Состояние кнопки может быть получено вызовом метода `getState` и установлено методом `setState`. При нажатии на кнопку вызывается метод `click` с номером текущего состояния. Этот метод возвращает номер нового состояния кнопки, для чего он должен быть переопределен в потомке.

Составные элементы управления

Конфигурации составных элементов управления состоит не только из узла конфигурации самого элемента, но и из узлов конфигурации его подэлементов.

Панель настройки (AdjustablePanel)

Панель настройки служит для управления целочисленным параметром визуализатора (например, количеством элементов в массиве). Она состоит из кнопок уменьшения и увеличения параметра, между которыми расположена подсказка, отображающая текущее значение регулируемого параметра.

Значение параметра может изменять большими (`blockIncrement`) и маленькими (`unitIncrement`) шагами, что задается в конфигурации панели. Так же в конфигурации задаются (см. раздел 5.4.1):

- исходное значение регулируемого параметра;

- максимальное и минимальное значение регулируемого параметра;
- шаблон отображения текущего значения;
- подсказки (как для панели в целом, так и для ее кнопок);
- надписи на кнопках панели.

Для последних трех параметров возможно задание значений на нескольких языках.

Панель управления визуализатором (AutoControlsPane)

Панель управления визуализатором рекомендуется для использования как единый элемент управления, позволяющий пользователю:

- совершать большие и маленькие шаги;
- запускать и останавливать автоматический режим;
- регулировать задержку между шагами в автоматическом режиме;
- перезапускать процесс визуализации;
- просматривать информацию о визуализаторе (используется диалог “О программе”).

Обычно панель управления визуализатором используется как первая строка панели управления в целом. При этом в остальных строках используется элементы управления, специфичные для конкретного визуализатора.

Диалоги

Модальный диалог (ModalDialog)

Класс `ModalDialog` служит для создания модальных диалогов, отображаемых визуализатором. Отличие `ModalDialog` от класса `Dialog`, определенного в пакете `java.awt` (наследником которого и является `ModalDialog`) состоит в следующем.

- Метод `center` — центрующий диалог относительно визуализатора, что удобно для пользователя;
- Закрытие диалога стандартными средствами ОС (например, кнопкой закрытия окна в *Microsoft Windows*).

На основе этого класса созданы остальные диалоги *Vizi*.

Диалог “О программе” (AboutDialog)

Класс `AboutDialog` служит для отображения информации об авторе визуализатора, помещенной в общее описание визуализатора (см. раздел 5.2.1). Соответствующую информацию он берет из конфигурации, куда она сохраняется на этапе построения визуализатора.

Диалог загрузки/сохранения (SaveLoadDialog)

Класс `SaveLoadDialog` рекомендуется для создания диалогов, позволяющих пользователю сохранять и загружать состояние визуализатора, что может быть полезно, например, при демонстрации его на лекции.

Если визуализатор запущен из командной строки, то пользователь может сохранять и загружать состояние визуализатора из файлов. При запуске визуализатора как аплета, эта опция доступна, только если она разрешена установками безопасности.

При конфигурации доступны следующие параметры:

- `commentPane-lines` — высота области комментария в строках;
- `columns` — исходное количество столбцов в области ввода;
- `rows` — исходное количество строк в области ввода;

6.3.4. Комментарии

Комментарии, отображаемые визуализатором задаются в XML-описании (см. разделы 5.3.2 и 5.3.3). При этом для каждого комментария задаются его параметры. При выполнение визуализатора параметры подставляются в комментарии как описано в [41].

Комментарии отображаются в области комментариев синхронно с изменение состояния визуализатора.

Возможно программное изменение текущего комментария при помощи вызова метода `setComment` класса `Base`. Эта возможность часто используется в режиме редактирования для отображения подсказки о возможных действиях.

6.3.5. Вспомогательные средства

Основные классы

Все визуализаторы, построенные на основе системы визуализации *Vizi* должны являться потомками класса `Base` (пакет `ru.ifmo.vizi.base`). Этот класс обеспечивает унификацию программного интерфейса при запуске визуализатора как аплета и самостоятельного приложения.

Потомки класса `Base` должны определить метод `createControlsPane`, который возвращает компоненту (`java.awt.Component`), отображаемую в области элементов управления. Обычно это компонента реализуется в виде панели (`java.awt.Panel`), содержащей несколько подпанелей, верхняя из которых является панелью управления визуализатора.

Так же класс `Base` инкапсулирует логику визуализатора, построенную на основе автоматов и определяет основные методы передвижения:

- `doNextStep` — совершает шаг вперед;
- `doNextBigStep` — совершает большой шаг вперед;
- `doPrevStep` — совершает шаг назад;
- `doPrevBigStep` — совершает большой шаг назад;
- `doRestart` — переходит в начальное состояния.

Классы `AppletView` и `FrameView` служат для запуска визуализаторов в виде аплета и самостоятельного приложения соответственно.

Поддержка ввода-вывода

Ввод-вывод поддерживается связкой `SaveLoadDialog` (см. раздел 6.3.3) и `SmartTokenizer`.

SaveLoadDialog

`SaveLoadDialog` позволяет унифицировать окна сохранения (восстановления) состояния визуализатора. При этом он определяет возможность работы с файлами и буфером обмена и отражает соответствующие элементы интерфейса.

Как и основной визуализатор окно `SaveLaodDialog` состоит из трех областей:

- область редактирования — позволяет пользователю изменять пример, на котором работает визуализатор;
- область комментария — отображает подсказку или информацию об ошибке;
- область элементов управления — отображает кнопки позволяющие возвращаться к исходному значению, производить операции с буфером обмена а так же сохранять и загружать состояние визуализатора из файлов.

Для использования `SaveLoadDialog` требуется создать класс унаследованный от него и определяющий метод `load`. Метод `load` может возвращать логическое значение или бросать исключение. Если метод вернул "`true`", то `SaveLoadDialog` закрывается. Если метод бросил исключение, то сообщения этого исключения отображается в строке комментария, что позволяет генерировать сообщение в месте обнаружения ошибки

Для `SaveLoadDialog` определены следующие свойства:

- `comment` — комментарий отображаемый в строке комментария;
- `content` — текущее содержимое;
- `initialContent` — исходной содержимое (используется при нажатии на кнопку "вернуть";

SmartTokenizer

`SmartTokenizer` позволяет в удобном виде разбирать данные полученные от `SaveLoadDialog`. Он позволяет легко читать числа (целые и вещественные) и строчки. При этом, выводятся локализованные сообщения об ошибках. Так же возможна проверка на окончание ввода. При разборе игнорируются комментарии (как `/* */` так и `//`). При заключении строки в кавычки она считывается как одно слово.

SmartTokenizer удобно использовать непосредственно в методе load SaveLoadDialog, так как при этом практически не придется делать обработку ошибок.

SmartTokenizer является потомком класса java.util.Tokenizer и предоставляет все методы определенные им. В дополнению к этому определены следующие методы:

- nextDouble — чтение вещественного числа;
- nextInt — чтение целого числа.
- nextWord — чтение следующего слова;
- nextBoolean — чтение булевского значения.

Словом считается последовательность символов, ограниченная пробелами и/или знаками препинания. Булевским значением считаются слова “true”, “yes”, “1” (истинное значение) и “false”, “no”, “0” (ложное значение).

Для целых и вещественных чисел так же определены методы чтения числа из диапазона, при этом, если число выходит за указанный диапазон, то создается исключительная ситуация:

- nextDouble(double min, double max) — чтение вещественного числа в диапазоне [min…max].
- nextInt(int min, int max) — чтение целого числа в диапазоне [min…max].

Так же определены методы ожидающие заданное слово либо конец ввода, при этом, если обнаружено неожидаемое значение, то бросается исключение:

- expect(String word) — ожидает появление заданного слова;
- expectEOF — ожидает появление конца файла.

Конфигурация

Конфигурация хранит все параметры визуализатора. Узел конфигурации представляется классом Confiuration (пакет ru.ifmo.vizi.base). Конфигурация визуализатора загружается из XML-описания (см. раздел 5.4).

Все конфигурируемые классы системы визуализации *Vizi* имеют конструкторы, получающие узел конфигурации и имя используемого параметра. Конфигурационный узел позволяет получать параметры стандартных *Java*-классов, так как их конструкторы не могут быть переопределены. Для это используются методы:

- `getColor` — загружает цвет (класс `java.awt.Color`) в формате `RRGGBB`, где `R`, `G`, `B` — шестнадцатиричные цифры;
- `getInteger` — загружает целое число;
- `getDouble` — загружает вещественное число;
- `getBoolean` — загружает булевское значение;
- `getFont` — загружает шрифт (класс `java.awt.Font`, см. раздел 5.4.2);
- `getParameter` — загружает строковое значение
- `hasParameter` — определяет, есть ли такой параметр.

Для каждого из этих методов задается имя параметра, из которого загружается значение. Для каждого указанного метода кроме `hasParameter` есть метод со значение по умолчанию, которое используется если указанный параметр не задан.

6.4. Генерация кода и отладка логики визуализаторов

6.4.1. Генерация кода

Реализация программы визуализации автоматически генерируется по XML-описанию визуализируемого алгоритма. По нему так же генерируется код отображающий комментарии и связка программы визуализации и реализации визуального представления.

Описание визуализируемой программы

Описание визуализируемой программы задается тегом `algorithm`. Внутри этого тега описываются:

- ссылки на включаемые классы и пакеты (тег `import`);

- описание глобальных переменных (тег `variable`);
- описание процедуры преобразования текущего состояния программы в строку (тег `toString`);
- описание отдельных процедур (тег `auto`).

Описания должны идти в указанном порядке.

Процедура описания преобразования текущего состояния в строку служит для отладочных целей (см. раздел 6.4.2).

Описание отдельной процедуры может включать описание переменных, тогда они будут локальными для этой процедуры (см. следующий подраздел).

Основной процедурой считается процедура с именем `Main` для нее можно описать комментарий и визуального представления в начальном и конечном состояниях (теги `start` и `finish` соответственно).

Использование переменных и @-нотация

При записи XML-описания визуализируемой программы используется разделение переменных на глобальных переменные и локальные переменные процедур. Глобальные переменные доступны во всех автоматах, а локальные — только в той процедуре, в которой они объявлены.

Глобальные переменные описываются непосредственно в элементе `algorithm`, а локальные переменные описываются в элементах процедур (`auto`), в которых они определены. И в том и в другом случае для описания переменных используется тег `variable`. Для объявления переменной необходимо указать ее имя, тип и описание, а для глобальной переменной и ее начальное значение (см. разделы 5.3.1 и 5.3.2).

Для обращения к переменным используется @-нотация. При этом имя переменной, хранящей модель данных не используется в явном виде, а подставляется автоматически. Обращение к переменной *<имя переменной>* имеет вид

`@`*<имя переменной>*

На пример:

```
@max = @a[@i]
```

осуществляет присваивание переменной `max` значения `i`-го элемента массива `a`. Заметим, что при наличии в области видимости глобальной и локальной переменной с одним и тем же именем используется локальная переменная.

Для создания строкового представления автомата в процедуре `toString` введен синтаксис позволяющий обращения к локальным переменных других процедур. Обращение к переменной *<имя переменной>* объявленной в процедуре *<имя процедуры>* записывается следующим образом:

`@`*<имя процедуры>*`@`*<имя переменной>*

На пример:

```
buffer.append(@Main@i);
```

осуществляет добавление к буферу значения локальной переменной `i` автомата `Main`.

Для доступа к переменным модели из кода визуализатора используется синтаксис

<переменная модели>`.`*<имя переменной>*

для глобальных переменных и

<переменная модели>`.`*<имя процедуры>*`_`*<имя переменной>*

для локальных переменных. На пример:

```
data.max = 0;
System.out.println(data.Main_i);
```

6.4.2. Отладка XML-описания визуализируемой программы

Отладка XML-описания состоит из двух этапов:

1. отладка визуализируемой программы;
2. отладка программы визуализатора.

На первом этапе осуществляется поиск ошибок переноса визуализируемой программы в XML-описание, а на втором этапе — отладка автоматически построенной программы визуализации.

Отладка визуализируемой программы

Для упрощения отладки визуализируемой программы предназначена цель `debug`, определенная в *build*-сценарии. Она строит реализацию визуализируемой программы, без использования конечных автоматов. При этом элементы `if` и `while` преобразуются в соответствующий операторы, а `step` и `call-auto` в части кода и вызовы процедур соответственно.

В полученный исходный код переносятся комментарии из XML-описания. В нем так же расставляются отступы, что может существенно упростить отладку. Сгенерированный код помещается в файл с именем указанным в XML-описании с добавленным суффиксом `Debug`. Полученный исходный код может отлаживаться с использованием любого программного инструмента.

Отладка программы визуализатора

Отладка программы визуализации производится при помощи валидатора автоматов. Для его использования служит цель `check`, определенная в *build*-сценарии. Она создает командный файл служащий для запуска валидатора и помещает его в каталок файлов визулизатора.

Валидатор осуществляет последовательную прогонку визуализатора на все большее количество шагов и осуществляет проверку правильности обращения на обратном проходе. При этом сравниваются значения выданные процедурой преобразования состояния автомата в строку (`toString`). При неравенстве строк на соответствующих шагах выдается сообщение об ошибке.

При запуске валидатор выводит номер текущего шага и комментарий к нему в стандартный поток вывода. При обнаружении ошибки в стандартный поток вывода выдается отладочная информация:

- номер шага, на котором произошла ошибка;
- результат функции `toString` при обратном проходе;
- результат функции `toString` при прямом проходе;

- результат функции `toString` для шага, после возвращения от которого произошла ошибка;
- результат функции `toString` для шага, предшествовавшего ошибке (при обратном проходе, т.е. шаг с номером на единицу больше).

При этом валидатор завершается кодом возврата `1`, в отличие от нормального завершения, когда ко возврата равен `0`.

Заметим, что для вывода отладочной информации используется расширенная функция `toString`, которая дополнительно выдает стек вызовов автоматов для шага и стека в котором сохраняются значения для обращения. При этом в функции `toString`, определенной в XML-описании для облегчения отладки имеет смысл выводить значения как можно большего количества переменных.

При запуске валидатора ему можно передать до двух параметров. Первый из них означает шаги с каким уровнем проверять (по умолчанию `-1`), а второй — с какого шага начинать проверку (по умолчанию `0`). Таким образом проверка может осуществляется следующим образом: сначала быстрая проверка на больших шагах, затем, при обнаружении ошибка локализуется с использованием маленьких шагов.

6.5. Практическое использование результатов работы

Система визуализации *Vizi* (как и ее предшественник *BaseApplet*) была использована в учебном процессе на кафедре Компьютерных технологий СПбГУ ИТМО. Студенты первого курса выполняли визуализаторы алгоритмов, которые в последствии будут использованы для преподавания курса дискретной математики и выложены на сайте Интернет-школы программирования.

6.5.1. Визуализаторы, выполненные на основе системы визуализации Vizi

На основе системы визуализации *Vizi* с использованием автоматов для представления логики визуализаторов было выполнено более двадцати визуализаторов. В том числе такие простые как алгоритм работы с деревом отрезков (один автомат и 15 состояний), так и такие сложные как алгоритмы работы с 2-3 деревьями (7 автоматов и 195 состояний) а так же алгоритм Малхотры-Кумара-Махешвари (8 автоматов и 89 состояний).

Список визуализаторов, выполнены на основе системы визуализации *Vizi* в 2003-2004 учебном году, представлен в табл. 5.

Табл. 5. Визуализаторы выполненные при помощи Vizi

Алгоритм	Автор	A	C
Построение кратчайшего дерева в ориентированном графе	Пименов С.	2	26
Битонический алгоритм для задача коммивояжера	Красильников Н.	2	25
2-3 Деревья	Красильников Н.	14	195
Циклы и разрезы в графах	Ахметов И.	16	97
Алгоритм Укконена	Ахметов И.	2	56
Алгоритм Прима	Ярцев Б.	2	21
Генерация всех простых строк и построение цикла де Брюина	Лоторейчик В.	2	21
Работа со стеком	Поликарпова Н.	14	54
Венгерский алгоритм	Кудинов М.	8	46
Нахождение максимального потока в сети методом Малхотры-Кумара-Махешвари	Бедный Ю.	18	89
Нахождение максимального потока в сети методом Диница	Бедный Ю.	8	68
Алгоритм Штрассена	Котов А.	2	16
Алгоритм Флойда	Колыхматов И.	2	47
Сортировка слиянием	Параценко Д.	6	31
Быстрая сортировка	Кочелаев. Д.	4	40
Дерево отрезков	Вокин А.	2	15
Сортировка кучей	Пименов И.	8	38
Алгоритм Краскала	Данилов В.	6	32

Здесь в столбце A обозначено количество автоматов, а в столбце C — суммарное количество состояний в них.

6.5.2. Требования к визуализаторам выполненным на основе системы визуализации Vizi

Для осуществления проверки визуализаторов были разработаны требования, позволяющие объективно оценить визуализатор с точки зрения использования технических средств.

Требования к языку и средам исполнения

Языки программирования

Визуализатор должен быть написан на языке Java.

При использовании следующих компиляторов не должно выдаваться сообщений об ошибках (*error*) и предупреждений (*warning*), в том числе предупреждений об использовании *deprecated API*:

1. Sun JDK 1.1.4
2. Sun JDK 1.1.8
3. Sun JDK 1.3
4. Sun JDK 1.4

Среды исполнения

Визуализатор должен работать в следующих средах:

1. AppletViever vv. 1.1.4, 1.1.8, 1.3, 1.4
2. Internet Explorer v. 5.0+
3. Netscape Navigator v. 4.0+
4. Opera v. 5.0+
5. Mozilla v. 1.0+

Требования к интерфейсу

Поддержка многоязычности

Визуализатор должен полностью поддерживать многоязычность (internationalization) на уровне XML-описания. При этом не допускается использование явной или неявной конкатенации строк для получения сообщений выводимых пользователю.

Совместимость

Визуализатор должен быть полностью совместим с последней версией системы визуализации *Vizi*, в том числе по интерфейсу пользователя, параметрам, значениям параметров по умолчанию.

Интерфейс визуализатора

К интерфейсу визуализатора предъявляются следующие требования.

1. Все действия визуализатора должны быть прокомментированы.
2. Интерфейс должен быть понятен неподготовленному пользователю.
3. Для каждого элемента управления должна отображаться соответствующая подсказка (hint).
4. Нажатие на активную (enabled) кнопку, выбор из активного списка и аналогичные действия пользователя должны сопровождаться видимыми изменениями состояния визуализатора.
5. В интерфейсе не должно быть зависимостей вида “Пользователь не может изменить количество вершин, пока не нажмет кнопку *Restart*”.
6. Визуализатор, должен подстраивать размер своих элементов (в том числе, размер шрифтов) под размер области, отведенной для рисования. При изменении размеров визуализатора должно происходить динамическое изменение размеров отображаемых элементов (в том числе, шрифтов).
7. Визуализатор должен “прилично” выглядеть при ширине 400+ и высоте 300+ пикселей. Для визуализаторов работы с деревьями и графами — соответственно 600+ и 400+ пикселей. При этом шрифт должен быть удобочитаем, в частности, не слишком мелким.

Требования к визуализаторам алгоритмов на графах

К визуализаторам алгоритмов на графах дополнительно предъявляются следующие требования.

1. Ребра графа, инцидентные одной вершине, не должны пересекаться.
2. Рекомендуется ввести два режима: “Визуализация” и “Редактирование ”, переключение между которыми осуществляется при нажатии одной из кнопок управления.
3. При визуализации должен отображаться не только граф (при необходимости, с весами), но и его матрица смежности (с возможностью отключения).
4. Если на некотором шаге алгоритм оперирует с вершиной (ребром, дугой), то она должна быть выделена не только на графе, но и в матрице смежности. При этом цвета выделения на рисунке графа и в матрице должны совпадать.
5. Столбцы и строки матрицы смежности должны быть помечены названиями (номерами) вершин.
6. Если в графе не допускаются петли, то в режиме редактирования на диагонали матрицы смежности должны отображаться крестики.

Прочие требования

Параметры визуализатора

1. Все строки, выводимые визуализатором, должны быть заданы в XML-описании визуализатора.
2. Должна быть предусмотрена возможность задания исходного примера, на котором работает визуализатор, в интуитивно понятной форме.
3. Наличие кнопки "Save/Load" должно конфигурироваться.
4. Цвета всех графических элементов визуализатора должны конфигурироваться через XML-описание визуализатора.

Прочие требования

1. По визуализатору (без дополнительной информации) подготовленному пользователю должно быть понятно, как работает алгоритм (при этом разрешаются ссылки на другие алгоритмы, как шаги визуализатора).
2. При иерархической структуре алгоритма должны быть предусмотрены шаги различных размеров.
3. Визуализатор должен обеспечивать сохранение своего состояния в и восстановление его из human-operable формата.

6.6. Выводы

Система визуализации *Vizi* существенно упрощает процесс создания визуализатора. Это достигается за счет сокращения как стадии проектирования визуализатора, так и стадии реализации.

На стадии проектирования система визуализации *Vizi* позволяет сконцентрироваться на разработке визуального представления и набора комментариев. При этом программисту предоставляются удобные средства для связывания этих частей с логикой визуализатора.

На стадии реализации программист получает автоматически сгенерированный код программы визуализатора. В этот код уже интегрирована возможность прозрачного взаимодействия с визуальным представлением и набором комментариев. При этом, элементы управления и визуального представления предоставляемые *Vizi* позволяют так же сократить затраты на реализацию соответствующих частей визуализатора.

Система визуализации *Vizi* была успешно внедрена на кафедре компьютерных технологий СПбГУ ИТМО и позволила реализовать визуализаторы многих сложных алгоритмов.

ЗАКЛЮЧЕНИЕ

В данной работе предложен метод построения логики визуализаторов на основе конечных автоматов. Анализ зарубежных и отечественных результатов в области построения визуализаторов позволяет утверждать, что разработанный метод является существенно новым и отличается формализованностью процесса построения логики визуализатора. Это позволяет строить на базе этого метода системы автоматизированного проектирования визуализаторов, на новом уровне.

В рамках метода был предложен способ преобразования программ (в том числе и рекурсивных) в систему взаимодействующих конечных автоматов. При этом полученная система автоматов позволяет эмулировать выполнения программы к вперед, так и назад. Ранее системы автоматов позволяли движение по программе только в одном направлении.

Для описания логики визуализаторов был разработан XML-формат описания визуализатора. В рамках этого формата удобно описывается как логика визуализатора (при этом XML-описание повторяет структуру исходной программы), так и его конфигурация. Второй отличительной особенностью предложенного формата является одновременно описание логики визуализатора, набора комментариев и визуального представления. Таким образом, производится ранняя интеграция этих частей визуализатора, что существенно упрощает дальнейшую разработку визуализатора и его отладку.

На основе разработанного метода была построена система визуализации *Vizi*, позволившая существенно упростить создание визуализаторов сложных алгоритмов и сократить сроки их разработки. Система визуализации *Vizi* напрямую поддерживает разработанный XML-формат описания визуализатора и позволяет как автоматически генерировать по нему исходные коды программы визуализатора, так и отлаживать ее. При этом существует возможность не только отладки программы визуализатора в целом, но и

отдельная отладка визуализируемой программы. Указанные возможности реализуются посредством инструментов входящих в систему визуализации *Vizi*.

Система визуализации была внедрена в 2003-2004 учебном году на кафедре компьютерных технологий СПбГУ ИТМО. При этом с ее использование студентами первых курсов были выполнены визуализаторы таких сложных алгоритмов как поиск максимального потока в сети методами Диница и Малхотры-Кумара-Махешвари. Создание визуализаторов таких алгоритмов ранее было сопряжено с большими трудностями, снятыми системой визуализации *Vizi*.

Таким образом, результаты достигнутые в данной работе позволяют существенно изменить процесс построения визуализатора. При этом в следствие формализации процесс для построения реализации визуализатора могут быть использованы менее квалифицированные кадры.

ИСТОЧНИКИ

Печатные издания на русском языке

1. *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 (ТУ), 2001.
2. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
3. *Грис Д.* Наука программирования. М.: Мир, 1984.
4. *Туккель Н.И.*, *Шалыто А.А.*, *Шамгунов Н.Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2002. № 5.
5. *Лингер Р.*, *Миллс Х.*, *Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982.
6. *Казаков М. А., Мельничук О. П., Парфенов В.Г.* Интернет-школа программирования в СПбГИТМО. Реализация и внедрение //Труды Всероссийской научно-методической конференции "Телематика-2002". СПб.: СПбГИТМО (ТУ), 2002.
7. *Столяр С. Е., Осипова Т. Г., Крылов И. П., Столяр С. С.* Об инструментальных средствах для курса информатики // II Всероссийская конференция “Компьютеры в образовании”. СПб.: 1994.
8. *Казаков М. А., Столяр С. Е.* Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования // Международная научно-методическая конференция “Телематика-2000” . СПб.: 2000. – С.189-191.
9. Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975.
10. *Кнут Д.* Искусство программирования. Том 1. Основные алгоритмы. М.: Вильямс, 2000.

11. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦНМО, 1999.
12. *Шалыто А.А.*, *Туккель Н.И.* Преобразование итеративных алгоритмов автоматные //Программирование. 2002. № 5.
13. *Шалыто А.А.*, *Туккель Н.И.* Программирование с явным выделением состояний //Мир ПК. 2001. № 8, № 9.

Печатные издания на английском языке

14. *Baecker R.* Sorting out Sorting // SIGGRAPH 1981, Dallas, Texas.
15. *Brown M., Sedgewick R.* A system for Algorithm Animation // Computer Graphics, Proceedings of the 11th annual conference on Computer graphics and interactive techniques, July 1984, pp. 177-186.
16. *Stasko J.* Tango: A Framework and System for Algorithm Animation // IEEE Computer 23, 1990, pp.27-39.
17. *Stasko J.* Animating Algorithms with XTANGO // SIGACT News, volume 23, issue 2, Spring 1992, pp. 67-71.
18. Stasko J. Tango: A Framework and System for Algorithm Animation // IEEE Computer, September 1990, pp.27-39.
19. *Lawrence A., Badre A., Stassko J.* Empirically evaluating the use of animations to teach algorithms // Technical report, Graphics Visualization and Usability Center, Georgia Institute of Technology, 1993.
20. *Demetrescu C., Finocchi I., Stasko J.* Specifying Algorithm Visualizations: Interesting Events or State Mapping? // Proceedings of the International Dagstuhl Seminar on Software Visualization, Schloss Dagstuhl, May 2001, appears in Software Visualization State-of-the-Art Survey, LNCS 2269, Stephan Diehl (ed.), Springer Verlag, 2002, pp. 16-30.
21. *Wilson J., Aiken R. Katz I.* Review of animation systems for algorithm understanding // SIGCSE'96, Proceedings of the 1st conference on integrating technology into computer science education, Volume 28 Issue SI.

22. *Rößling G., Freisleben B.* ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. // Journal of Visual Languages and Computing, Volume 13 issue 3. pp. 341-354, Elsevier, Amsterdam, The Netherlands, 2002.
23. *Naps T., Rößling G., Almstrum V., Dann W., Fleischer F., Hundhausen C., Korhonen A., Malmi L., McNally M., Rodger S.* Exploring the Role of Visualization and Engagement in Computer Science Education. // inroads — Paving the Way Towards Excellence in Computing Education. pp. 131-152, ACM Press, New York, 2003.
24. *Pierson W., Rodger S. H.* Web-based Animation of Data Structures Using JAWAA // Twenty-ninth SIGCSE Technical Symposium on Computer Science Education, p. 267-271, 1998.
25. *Rodger S. H.* Using Hands-on Visualizations to Teach Computer Science from Beginning Courses to Advanced Courses, Second Program Visualization Workshop, Hornstrup Centert, Denmark, June 2002.
26. *Rodger S. H.* Introducing Computer Science Through Animation and Virtual Worlds, Thirty-third SIGCSE Technical Symposium on Computer Science Education, p. 186-190, 2002.
27. *Crescenzi P., Demetrescu C., Finocchi I., Petreschi R.* Reversible execution and visualization of programs with Leonardo // Journal of Visual Languages and Computing (JVLC), volume 11 issue 2, pp. 125-150, Academic Press, April 2000.
28. *Demetrescu C., Finocchi I.* Smooth animation of algorithms in a declarative framework // Journal of Visual Languages and Computing (JVLC), Volume Issue 3, Special Issue devoted to selected papers from the 15th IEEE Symposium on Visual Languages, Academic Press, 2001.
29. *Brown M.* Algorithm Animation. MIT Press, Cambridge, Massachussets, 1988.

30. *Moreno A., Myller N., Sutinen E., Ben-Ari M.* Visualizing Programs with Jeliot 3 // Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004, Gallipoli (Lecce), Italy, 25-28 May, 2004
31. *Moreno A., Myller N.* Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family // Proceedings of International Conference on Networked e-learning for European Universities.
32. *Lahtinen S., Sutinen E., Tarhio J.* Automated Animation of Algorithms with Eliot // Journal of Visual Languages and Computing 9 (3), p. 337–349.

Ресурсы сети Internet

33. Extensible Markup Language (XML) 1.0 (Second Edition) // http://www.w3.org/TR/2000/REC-xml-20001006/.
34. XML Schema Part 0: Primer // http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.
35. XML Schema Part 1: Structures // http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.
36. XML Schema Part 2: Data Types // http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.
37. *B. Joy, G. Steele, J. Gosling, G. Bracha* Java Language Specification (Second Edition). Addison-Wesley. 2000. // http://java.sun.com/docs/books/jls/.
38. Code conventions for Java programming language. // http://java.sun.com/docs/codeconv/.
39. Apache Ant Home Page // http://ant.apache.org/.
40. Properties class description (Java 2 Platform SE 1.4.2) // http://java.sun.com/j2se/api/java/util/Properties.html.
41. MessageFormat class description (Java 2 Platform SE 1.4.2) // http://java.sun.com/j2se/api/java/text/MessageFormat.html.
42. State University of New York College at Brockport sorting animators collection // http://www.cs.brockport.edu/cs/javasort.html.

43. Princeton University animators collection // http://www.cs.princeton.edu/~ah/alg_anim/version1/Animator.html.
44. Hope College animator page // http://www.cs.hope.edu/~alganim/animator/Animator.html.
45. Animal Home Page // http://www.animal.ahrgr.de.
46. Jeliot Home Page // http://cs.joensuu.fi/jeliot/.
47. Polka Animation System // http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html.
48. Samba algorithm animation system // http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html.
49. Leonardo home page // http://www.dis.uniroma1.it/~demetres/Leonardo/Leonardo.html.
50. *Arsac O., Lavirotte S.* The Agat Manual // http://www-sop.inria.fr/cafe/Olivier.Arsac/agat/Doc/refman.ps.gz.
51. Agat: Another Graphical Animation Tool http://www-sop.inria.fr/cafe/Olivier.Arsac/agat/agat.html.

Публикации

52. *Казаков М. А., Корнеев Г. А., Шалыто А. А.* Метод построения логики работы визуализатора алгоритмов на основе конечных автоматов //Телекоммуникации и информатизация образования. 2003. №6, с. 27-58.
53. *Корнеев Г. А., Шалыто А. А.* Реализация конечных автоматов с использованием объектно-ориентированного программирования. //Труды международной научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ), 2003.
54. *Корнеев Г. А., Казаков М.А., Шалыто А. А.* Построение логики работы визуализаторов авлгоритмов на основе автоматного подхода. //Труды международной научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ), 2003.

55. *Корнеев Г.А., Парфенов В.Г., Столяр С.Е., Васильев В.Н.* Визуализаторы алгоритмов как основной инструмент технологии преподавания дискретной математики и программирования //Труды международной научно-методической конференции "Телематика-2001". СПб.: СПбГИТМО (ТУ), 2001.

ПРИЛОЖЕНИЯ

Приложение 1. Пример XML-описания визуализатора

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!DOCTYPE visualizer PUBLIC
  "-//IFMO Vizi//Visualizer description"
  "http://ips.ifmo.ru/vizi/dtd/visualizer.dtd"
>

<visualizer
  id="FindMaximum"
  package="ru.ifmo.vizi.find_max"
  main-class="FindMaximumVisualizer"

  preferred-width="400"
  preferred-height="250"

  name-ru="Поиск максимума в массиве\nнатуральных чисел (пример)"
  name-en="Search for maximimum element in the array\nof natural
           numbers (example)"

  author-ru="Георгий Корнеев"
  author-en="Georgiy Korneev"
  author-email="kgeorgiy@rain.ifmo.ru"

  supervisor-ru="Георгий Корнеев"
  supervisor-en="Georgiy Korneev"
  supervisor-email="kgeorgiy@rain.ifmo.ru"

  copyright-ru="Copyright \u00A9 Кафедра КТ, СПб ГИТМО (ТУ), 2003"
  copyright-en="Copyright \u00A9 Computer Technologies Department,
           SPb IFMO, 2003"
>
  <algorithm>
    <variable
      description = "Массив для поиска"
      name    = "a"
      type    = "int[]"
      value    = "new int[]{1, 2, 3, 1, 3, 5, 6}"
    />
    <variable
      description = "Экземпляр апплета"
      name    = "visualizer"
      type    = "FindMaximumVisualizer"
      value    = "null"
    />
    <variable
      description = "Текущий максимум"
      name    = "max"
```

```
  type    = "int"
  value    = "0"
/>
<data>
  <toString>
    StringBuffer s = new StringBuffer();
    s.append("max = ").append(@max).append("\n");
    s.append("i = ").append(@Main@i).append("\n");
    return s.toString();
  </toString>
</data>

<auto id="Main" description="Ищет максимум в массиве">
  <variable
    description = "Переменная цикла"
    name    = "i"
    type    = "int"
  />
  <start
    comment-ru="На экране изображен массив, в котором будет
        осуществляться поиск максимума"
    comment-en="There is an array on the display"
  >
    <draw>
      @visualizer.updateArray(0, 0);
    </draw>
  </start>
  <step
    id="Initialization"
    description="Инициализация"
    comment-ru="Инициализируем максимум нулем (так как в
        массиве только натуральные числа)."
    comment-en="Intitialize maximum by zero (because array
        contains only positive numbers)."
  >
    <draw>
      @visualizer.updateArray(0, 0);
    </draw>
    <action>
      @max @= 0;
    </action>
  </step>
  <step
    id="LoopInit"
    description="Начало цикла"
    level="-1"
  >
    <action>
      @i @= 0;
    </action>
  </step>
  <while
    id="Loop"
```

```
  description="Цикл"
  test="@i &lt; @a.length"
  level="-1"
>
  <if
    id="Cond"
    description="Условие"
    test="@max &lt; @a[@i]"
    true-comment-ru="{0} больше текущего максимума ({1})"
    true-comment-en="{0} greater than current maximum ({1})"
    false-comment-ru="{0} не больше текущего максимума ({1})"
    false-comment-en="{0} not greater than current maximum
      ({1})"
    comment-args="new Integer(@a[@i]), new Integer(@max)"
  >
    <draw>
      @visualizer.updateArray(@i, 1);
    </draw>
    <then>
      <step
        id="newMax"
        description="Обновление максимума"
        comment-ru="Обновляем текущий максимум"
        comment-en="Updating current maximum"
      >
        <draw>
          @visualizer.updateArray(@i, 2);
        </draw>
        <action>
          @max @= @a[@i];
        </action>
      </step>
    </then>
  </if>
  <step
    id="inc"
    description="Инкремент"
    level="-1"
  >
    <action>
      @i @= @i + 1;
    </action>
  </step>
</while>
<finish
  comment-ru="Максимум найден ({0})"
  comment-en="Maximum found ({0})"
  comment-args="new Integer(@max)"
>
  <draw>
    @visualizer.updateArray(0, 0);
  </draw>
</finish>
```

```
  </auto>
</algorithm>
<configuration>
  <property
    description = "Comment pane height"
    param    = "comment-height"
    value    = "40"
  />
  <adjustablePanel
    description = "Number of elements in the array"
    param      = "elements"
    caption-ru    = "Элементов: {0,number,####}"
    caption-en    = "Elements: {0,number,####}"
    hint-ru    = "Количество элементов в массиве"
    hint-en    = "Number of elements in the array"
    value      = "5"
    minimum    = "5"
    maximum    = "20"
    unitIncrement   = "1"
    blockIncrement  = "2"
    blockInterval   = "500"
  >
    <button
      param    = "incrementButton"
      caption-ru  = "&gt;&gt;"
      caption-en  = "&gt;&gt;"
      hint-ru  = "Увеличить количество элементов"
      hint-en  = "Increase number of elements"
    />
    <button
      param    = "decrementButton"
      caption-ru  = "&lt;&lt;"
      caption-en  = "&lt;&lt;"
      hint-ru  = "Уменьшить количество элементов"
      hint-en  = "Decrease number of elements"
    />
  </adjustablePanel>
  <button
    description = "Fills the array with random values"
    param    = "button-random"
    caption-ru  = "Случайно"
    caption-en  = "Random"
    hint-ru  = "Заполнить массив случайными значениями"
    hint-en  = "Fill the array with random values"
  />
  <message
    description = 'Message for the "Max" label'
    param    = "max-message"
    message-ru  = "max = {0}"
    message-en  = "max = {0}"
  />
  <message
```

```
  description = 'Comment for "ArrayLength" parameter in the
        output file'
  param    = "ArrayLengthComment"
  message-ru  = "Длина массива ({0} ... {1})"
  message-en  = "Array length ({0} ... {1})"
/>
<message
  description = 'Comment for "Elements" parameter in the output
        file'
  param    = "ElementsComment"
  message-ru  = "Элементы массива ({0} ... {1})"
  message-en  = "Array elements ({0} ... {1})"
/>
<message
  description = 'Comment for "Step" parameter in the output
        file'
  param    = "StepComment"
  message-ru  = "Номер шага"
  message-en  = "Current step"
/>
<styleset
  description = "Array style set"
  param    = "array"
>
  <style
    description  = "Ordinary cell"
    text-color    = "000000"
    text-align    = "0.5"
    border-color  = "000000"
    border-status   = "true"
    fill-color    = "8080ff"
    fill-status  = "true"
    aspect-status   = "false"
    padding    = "0.2"
  >
    <font
      face    = "Serif"
      size    = "12"
      style    = "plain"
    />
  </style>
  <style
    description  = "Selected cell"
    fill-color    = "80ff80"
  />
  <style
    description  = "Local-maximum cell"
    fill-color    = "ff8080"
  />
</styleset>
<style
  description  = 'Style of the "Max" label'
  param      = "max-style"
```

```
      border-status   = "false"
      fill-status  = "false"
    >
      <font size="20"/>
    </style>
    <property
      description = "Maximum possible value of the element"
      param    = "max-value"
      value    = "99"
    />
    <property
      description = "Widest possible value of the element"
      param    = "max-value-string"
      value    = "88"
    />
    <group
      description = "Save/Load dialog configuration"
      param    = "SaveLoadDialog"
    >
      <property
        description = "Height of the comment pane"
        param    = "CommentPane-lines"
        value    = "2"
      />
      <property
        description = "Width of the text area"
        param    = "columns"
        value    = "40"
      />
      <property
        description = "Height of the text area"
        param    = "rows"
        value    = "7"
      />
    </group>
  </configuration>
</visualizer>
```

Приложение 2. История изменений системы визуализации Vizi

Обозначения

- [+] Новое в программе
- [-] Исправленная ошибка
- [!] Важное изменение

История изменений системы визуализации Vizi

Vizi 0.4b4 (12.05.04)

[-] Исправлена ошибка с работой с очередью собщений в `Timer`.

[-] Исправлена ошибка с перерисовкой в *Internet Explorer*.

[-] При использовании `rtest` @-нотация не обрабатывалась.

[-] При загрузке данных в примере `FindMaximum` не всегда осуществлялась корректная обработка введенных значений.

Vizi 0.4b3 (06.05.04)

[-] Исправлена ошибка с вызовом вложенных автоматов.

[+] В схему добавлен тег `method`, служаший для определения глобальных методов.

[!] Из описания визуализтора (тег `visualizer`), удален аттрибут `type`.

Vizi 0.4b2 (23.03.04)

[+] Добавлен класс `AdjustablePanel`, который имеет большие и маленькие шаги.

[!] `SpinPanel` использовать больше не рекомендуется. Используйте `AdjustablePanel`.

[!] Следующие классы, использовавшиеся панелью управления удалены: `AboutButton`, `AutoButton`, `DelayPanel`, `NextBigStepButton`, `NextStepButton`, `PrevBigStepButton`, `PrevStepButton`, `RestartButton`.

[+] В readme добавлен раздел *JavaBeans*.

[+] В readme добавлен раздел *Благодарности*.

[!] Переделан `Timer`. Теперь он правильно работает с потоком AWT (через события). Метод `tick()` использовать не рекомендуется.

[!] В классе `HintedButton` метод `click()` больше не является абстрактным (как и сам класс).

Vizi 0.4b1 (18.03.04)

[!] Переработана структура классов автомата, с целью уменьшения размера class-файлов.

[!] Введено разделение переменных на локальные и глобальные (см.

раздел *Использование переменных* в readme).

[!] Обновлен *FindMaximum* (теперь он использует глобальные и локальные переменные).

[!] `WhatsNew` частично переведен в HTML.

[-] Исправлена ошибка в обращении цикла `while`, когда он был первым ребенком контейнера.

[-] При генерации описаний ошибки больше не выдаются.

[-] Исправлена ошибка с отображением подсказок.

[-] Ошибки при генерации описания конфигурации (цель `description`) больше не появляются.

[+] В readme добавлен раздел *Использование переменных*

[!] В readme обновлены разделы *Автоматическое обращение шагов типа step* и *Важные замечания*

Vizi 0.3sp2 (17.03.04)

[+] Введен параметр фигуры `message-align`, который отвечает за выравнивание надписи относительно фигуры (`text-align` отвечает за выравнивание строк в многострочных надписях)

[-] Метод для установки отступов переименован из `setTextAlign` в `setPadding`.

[!] Обновлен проверщик автоматов (`ru.ifmo.vizi.base.auto.Check`)

[+] В readme добавлены разделы *Автоматическая проверка автоматов* и *Важные замечания*

[!] Readme переведен в HTML.

Vizi 0.3sp1

[-] Исправлены англоязычные сообщения для `SmartTokenizer`.

[-] Исправлена загрузка в примере *FindMaximum*.

Vizi 0.3 (29.12.03)

[-] Исправлено рисование эллипсов и скругленных углов.

Vizi 0.3b3

[!] Практически полность переписан `SaveLoadDialog`, изменена концепция его работы (см. *Использование SaveLoadDialog* в readme).

[+] Добавлен `SmartTokenizer` (см. *Использование SmartTokenizer* в readme).

[!] `FindMaximumVisualizer` теперь умеет сохранять/восстанавливать состояния (правильно использует `SaveLoadDialog` и `SmartTokenizer`, см. исходники).

[+] Добавлен класс `ModalDialog` позволяющий легко создавать модальные диалоги и центрировать окна относительно компонентов.

[!] `AboutDialog` и `SaveLoadDialog` теперь используют `ModalDialog`.

[!] `CommentPane` перенесена в пакет "`ui`".

[+] Добавлен метод форматирования (message) с массивом параметров в `I18n`.

[+] Формат кнопки *Save/Load* добавлен в стандартную конфигурации (`button-SaveLoad`) так же добавле флаг, указывающий необходимость отображать эту кнопку (`button-ShowSaveLoad`). Пример использования см. в *FindMaximumVisualizer*.

[!] Для билда теперь требуется *Xerces Java 2 2.5.0+* и новый `SchemaValidator` (незабудте скачать).

[-] Исправлена двойная буфферизация.

Vizi 0.3b2

[!] Изменена структура каталогов проекта (смотри readme).

[!] Теперь каждый проект -- отдельный каталог (не требуется таскать весь *Vizi*).

[!] Для каждого проекта требуется файл с его свойствами (`project.properties`).

[+] При изменении текущего проекта автоматически очищаются все временные каталоги.

[+] При изменении любого .xml файла в каталоге с описанием визуализатора вызывает перекомпиляцию описания визуализатора (теперь не требуется каждый раз говорить `ant clean`).

[!] Цель `docs` переименована в `api-docs`

[+] Добавлена цель `vizi` строящая `vizi.jar`.

[!] В файле проекта теперь указывает версия *Vizi* используемая этим проектом (незабывайте ее изменять при переходе на новые версии).

Vizi 0.3b1

[+] Добавлен `SaveLoadDialog`. (см. раздел *Использование SaveLoadDialog* в readme).

[+] Добавлена schema для XML-описания визуализатора.

Vizi 0.2 (25.08.03)

[-] Класс `Ellipse` теперь польностью определенный (не `abstract`).

[-] Возвращен параметр стиля фигуры `padding` (измеряется в долях ширины и высоты шрифта).

[+] Поддержка параметра фигуры `aspect` (отношение ширины к высоте).

[+] Методы для загрузки наборов стилей (`loadStyleSet(...)`) добавлены в класс `ShapeStyle`.

[!] Набор стилей по умолчанию (`styleSet[]`) больше не поддерживается классом `Base`.

[!] Конфигурирование визуализатора через `.properties` файлы отменено.

Vizi 0.1 (14.08.03)

[+] Добавлена фигура `Ellipse`.

[!] Теперь в описание шрифта не вносится слово `font`. То есть следует писать "`controls-font`" вместо "`cotrols`".

[!] В классе `Configuration` метод `getString` переименован в `getParameter`.

[+] В классе `Configuration` для `getColor`, `getInteger`, `getdouble`, `getBoolean`, `getParameter`, `getFont` сделаны методы как со значениями по умолчанию так и без них. Методы со значениями по умолчанию не следует использовать для загрузки начальных параметров.

[!] Изменен формат представления визуализатора (алгоритма). Теперь `<algorithm>` вложен в `<visualizer>`. Так же в `<visualizer>` вложен тэг `<configuration>`. Соответствующая DTD: `<!DOCTYPE visualizer PUBLIC "-//IFMO Vizi//Visualizer description" "http://ips.ifmo.ru/vizi/visualizer.dtd" >`

[!] Теперь вся конфигурация должна записываться в описаниии визуализатора.

История изменений BaseApplet

BaseApplet 1.5b2 (05.08.03)

[+] Добавлена возможность автаматического обращения шагов step.

[+] Добавлен пример `AutoReverseFindMaximum.xml`, полностью построенный на автоматическом обращении.

BaseApplet 1.5b1 (04.08.03)

[-] Исправлен подсчет шагов автомата. Теперь считается количество шагов самого вложенного автомата.

[+] В XML описание алгоритма добавлена информация об авторе.

[+] Добавлен пример `AutoReverseFindMaximum.xml`, полностью построенный на автоматическом обращении.

[!] В лог выдается информация только о пропущенных и повторяющихся параметрах.

[!] Переработана структура каталогов проекта.

[!] Переработан build скрипт.

[+] .property файлы конвертируются автоматически. Больше не нужны файлы `*.ru.properties`.

[+] Автоматическая проверка используемых классов и методов на совместимость с jdk 1.1.8.

[-] Теперь build скрипт всегда создает .class файлы совместимые с jdk 1.1 (в том числе под jdk 1.4+).

[-] Теперь правильно рисуются эллипсы (`drawOval` и `fillOval`).

[+] Краткое описание структуры проекта

[!] Убраны переменные `drawColor*` и `fillColor*`

[!] Классы в место BaseApplet дожны расширять Base.

[+] Визуализаторы могоут запускаться как отдельные приложения.

[!] Теперь автомат создается с параметром Locale.

[!] Теперь у конструктора визуализатора есть параметры.

[+] Автоматически создаеются `About_??.properties`, незабудьте включить их в `Configuration.properties`.

[!] В место метода `init` теперь используется конструктор, в конце которого следует вызвать `createInterface`.

[!] Считается что `<класс визуализатора>=<класс алгоритма>Visualizer` (а не `Applet`).

[+] Добавлена возможность ассоциировать действия с клавишами (через `ActionManager`).

[+] Добавлена возможность вызывать методы по именам (через `ActionManager`).

[+] `clientPaneReshaped` переименован в `layoutClientPane`, для явного вызова этого метода используйте `clientPane.doLayout()`.

BaseApplet 1.4_05 (29.07.03)

[+] К стандартным кнопкам добавлена "About".

[+] Добавлен `AboutDialog` и соответствующие параметры конфигурации.

[+] В описание автоматов добавлены элементы `start` и `finish`, с возможностью добавлять комментарии к начальному и конечному состоянииию, а также отрисовывать их специальным образом.

[-] Исправлен `auto.dtd`.

[+] Скрипты автоматической генерации компилируют апплет, создают `jar` и `html` в каталоге `deploy`.

BaseApplet 1.4_04 (10.04.03)

[+] В параметры `Shape` добавлен `padding` — отступ по сторонам текста в долях от высоты и ширины символа.

BaseApplet 1.4_03 (10.04.03)

[!] Хинты отображаются по-новому (тормоза должны пропасть).

[-] Исправлено получение коментария от вложенного автомата.

[-] Исправлена отрисовка для вложенного автомата.

[+] В `Shape` добавлен метод `getMessage()`.

BaseApplet 1.4_02 (09.04.03)

[-] Исправлено имя конструктора в `generate.xsl`.

[!] В `auto.dtd` для тега `step` имя ат`трибута `args` исправленл на `comment-args`.

[-] Исправлены рускоязычные сообщения для `delay-panel`.

[+] В класс `Message` добавлен метод `getMessage()`.

BaseApplet 1.4_01 (25.03.03)

[!] Исходная версия ветви `BaseApplet 1.4.x`