Санкт-Петербургский государственный университет информационных технологий, механики и оптики

На правах рукописи

Корнеев Георгий Александрович

Автоматизация построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода

Специальность 05.13.12 — Системы автоматизированного проектирования (приборостроение)

диссертация на соискание ученой степени кандидата технических наук

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

Санкт-Петербург

2006

ОГЛАВЛЕНИЕ

СПИСОК ТЕРМИНОВ

Визуализаторы и системы визуализации

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

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

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

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*.

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

Интересное состояние — управляющее состояние программы, отображаемое пользователю.

Состояние алгоритма — вычислительное состояние алгоритма (включает в себя значения всех переменных).

Части визуализатора

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

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

Логика визуализатора — часть визуализатора, обеспечивающая трассировку алгоритма и предоставляющая данные другим частям визуализатора для отображения их пользователю.

Модель данных — часть визуализатора, хранящая значения переменных, используемых в программе, и предоставляющая к ним доступ другим частям визуализатора.

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

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

Элементы управления — часть визуализатора, при помощи которой пользователь управляет визуализатором.

Программы

Итеративная программа — программа без рекурсии.

Рекурсивная программа — программа, использующая рекурсию.

Приведенная форма (программы) — программа, записанная в форме, содержащей только операторы присваивания, блочные, цикла с предусловием, ветвления и вызова процедур.

Приведение программы — преобразование программы в *приведенную форму*.

Явная рекурсия — рекурсивная процедура непосредственно содержит вызов самой себя.

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

Построение модели данных — создание модели данных по программе и модификация программы к виду, использующему модель данных.

Автоматы

Управляющее состояние — состояние, управляющего автомата (непосредственно влияет на выполняемые переходы).

Вычислительное состояние — состояние управляемого объекта (влияет на исполняемые переходы только косвенно).

Прямой автомат — автомат, осуществляющий трассировку программы в прямом направлении.

Обратный автомат — автомат, осуществляющий трассировку программы в обратном направлении.

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

Фрагмент автомата — набор состояний и переходов, при этом начальные и/или конечные состояния переходов могут быть не определены.

*Вход (*фрагмента автомата*)* — переход, у которого не определено начальное состояние.

*Выход (*фрагмента автомата*)* — переход, у которого не определено конечное состояние.

Замыкание — объединение входа фрагмента автомата с выходом другого фрагмента автомата в один переход.

Прочие термины

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

Визуализируемый алгоритм — алгоритм, который поясняет визуализатор.

Описание визуализатора — запись информации о визуализаторе для последующей автоматизированной обработки.

Описание визуализируемой программы — запись визуализируемой программы для последующей автоматизированной обработки.

ВВЕДЕНИЕ

Актуальность проблемы. В настоящее время приборы и устройства становятся все сложнее. Для автоматизации их проектирования требуется знание алгоритмов на графах [17, 22], операций над матрицами [20], вычислительной геометрии [27], линейного и динамического программирования [5, 18] и т.д. Часто эти алгоритмы являются весьма сложными и, поэтому, трудны для изучения. Это требует новых подходов к разработке научных основ обучения автоматизированному проектированию, и, в частности, обучению указанным алгоритмам [11, 97]. Таким новым подходом является применение визуализаторов алгоритмов [34]. Так как указанные алгоритмы часто сложны, то создание их визуализаторов требует разработки методов построения визуализаторов с целью формализации процесса их проектирования и реализации.

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

При изучении большинства алгоритмов наряду с режимом «шаг вперед» весьма полезен также и режим «шаг назад», 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» СПбГУ ИТМО [97] показал, что они могут быть использованы как основной инструмент преподавания указанных выше курсов, и, в частности, при дистанционном обучении [10].

Написание визуализатора «с нуля» является трудоемкой задачей. Для ее решения используются *системы визуализации*, предоставляющие как средства создания визуализаторов, так и поддержку времени выполнения.

Обычно системы визуализации предоставляют графический инструментарий, позволяющий разработчику строить визуализаторы. Некоторые системы визуализации дополнительно содержат средства, которые помогают выделять *интересные состояния* визуализируемого алгоритма либо визуализировать изменения в структурах данных.

Несмотря на то, что визуализаторы разрабатываются с 80-х годов XX века, можно утверждать, что к настоящему времени основные достижения в проектировании визуализаторов относятся к сфере применения их в учебном процессе [11], а успехи в сфере технологии создания визуализаторов практически отсутствуют. В частности, отсутствует метод, позволяющий по алгоритму формально и единообразно создавать логику работы визуализатора.

Поэтому весьма актуальными являются исследования, направленные на решение проблем построения визуализаторов сложных алгоритмов, особенно, встречающихся в системах автоматизации проектирования (САПР) приборов.

Целью диссертационной работы является разработка и реализация методов построения визуализаторов алгоритмов дискретной математики, которые позволят формализовать процесс построения визуализаторов и обеспечить корректность их построения.

Основные задачи диссертационной работы состоят в следующем.

1. Развитие методов преобразования императивных программ в автоматные.

2. Разработка языка описания визуализаторов алгоритмов дискретной математики.
3. Разработка технологии построения визуализаторов алгоритмов.
4. Внедрение результатов работы в практику программирования визуализаторов и учебный процесс в СПбГУ ИТМО.

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

1. Предложен метод, позволяющий формально выполнять преобразование императивных (в том числе рекурсивных) программ в систему взаимодействующих конечных автоматов.
2. На основе указанного выше метода разработан метод, позволяющий формально преобразовывать программу в систему взаимодействующих конечных автоматов, обеспечивающую трассировку исходной программы в прямом и обратном направлениях.
3. Разработан язык описания визуализаторов алгоритмов дискретной математики.
4. На основе предложенных методов преобразования программ к автоматному виду разработана технология, автоматизирующая построение визуализаторов рассматриваемого класса.

Методы исследования. В работе использованы методы теории автоматов, теории графов, теории алгоритмов, теории компиляторов и языков программирования.

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

Практическое значение работы состоит в том, что все полученные результаты могут быть использованы и в настоящее время уже используются на практике в учебном процессе в СПбГУ ИТМО и других учебных учреждений..

Предложенные методы позволили упростить процесс создания визуализаторов алгоритмов и внесение изменений в них. При этом за счет предложенных методов и их автоматизации уменьшается количество ошибок, допускаемых при разработке визуализаторов алгоритмов.

Эти методы реализованы в системе визуализации *Vizi*, существенно упрощающей и ускоряющей процесс создания визуализаторов алгоритмов за счет автоматизации некоторых операций.

Результаты данной работы могут быть использованы двояко: во-первых, учащиеся могут пользоваться визуализаторами, созданными по предложенной технологии, а, во-вторых, учащиеся могут сами создавать визуализаторы.

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

1. В учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсу «Дискретная математика» и выполнении курсовых работ по этому курсу.
2. В учебном процессе в Лицее «Физико-техническая школа» (Санкт-Петербург).
3. В учебном процессе в Специализированном учебно-научном центре МГУ (Москва).

Научно-исследовательские работы. Результаты диссертации были получены в ходе выполнения научно-исследовательских работ по гранту конкурса персональных грантов для студентов и аспирантов вузов и академических институтов Санкт-Петербурга; по теме «Разработка технологии программного обеспечения систем управления на основе автоматного подхода», выполняемой по заказу Министерства образования РФ в 2001-2006 гг.; по теме «Разработка технологии автоматного программирования», выполненной по гранту РФФИ № 02-07-90114 в

2002-2003 гг.; по теме «Разработка технологии объектно-ориентированного программирования с явным выделением состояний», выполняемой по гранту РФФИ № 05-07-90011; по государственному контракту «Технология автоматного программирования: применение и инструментальные средства», выполняемому в рамках ФЦНТП «Исследования и разработки по приоритетным направлениям развития науки и техники на 2002 – 2006 годы».

Апробация результатов работы. Основные положения диссертационной работы докладывались на Второй Всероссийской научной конференции «Методы и средства обработки информации» (Москва, МГУ, 2005 г.); II и III конференции молодых ученых СПбГУ ИТМО (Санкт-Петербург, 2005, 2006 гг.); Политехническом симпозиуме «Молодые ученые — промышленности Северо-Западного региона» (Санкт-Петербургский государственный политехнический университет, 2005 г.); Software Engineering Conference in Russia — SECR-2005 (Москва, 2005 г.); научно-методических конференциях «Телематика-2001», «Телематика-2003» (Санкт-Петербург); XXXV научной и учебно-методической конференция СПбГУ ИТМО «Достижения ученых, аспирантов и студентов СПбГУИТМО в науке и образовании» (Санкт-Петербург, 2006 г.); на семинаре «Автоматное программирование» в рамках международной конференции «International Computer Science Symposium in Russia CSR 2006» (Санкт-Петербург, 2006 г.).

Публикации. По теме диссертации опубликовано 11 печатных работ, в том числе в Научно-техническом вестнике СПбГУ ИТМО (входит в «Перечень ведущих рецензируемых научных журналов и изданий, выпускаемых в Российской Федерации»); сборниках трудов конференций «Методы и средства обработки информации», «Телематика», «Межвузовская конференция молодых учёных», Политехнического симпозиума «Молодые ученые — промышленности Северо-Западного региона»; журналах «Телекоммуникации и информатизация образования», «Компьютерные инструменты в образовании».

Структура диссертации. Диссертация изложена на 181 странице и состоит из списка терминов, введения, шести глав, заключения и двух

приложений (на 11 страницах). Список литературы содержит 103 наименования. Работа иллюстрирована 50 рисунками и содержит 12 таблиц.

Глава 1 содержит описание текущего состояния в области создания и применения визуализаторов и систем визуализации. В частности, рассмотрено применение визуализаторов в учебном процессе и предъявляемые к ним требования.

В этой главе также приводится анализ существующих 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* предлагается процесс построения визуализаторов алгоритмов, и рассматриваются пути его автоматизации. Вначале выделяется структура визуализатора: основные части и связь между ними. Далее анализируются подходы к построению основных частей визуализатора, и обосновывается необходимость автоматизации их построения. В частности, рассматривается автоматный подход к построению логики визуализаторов алгоритмов. В конце главы формулируются задачи, решаемые в диссертационной работе.

В *главе 3* рассматриваются преобразования, позволяющие упростить дальнейшее преобразование программы в систему взаимодействующих конечных автоматов. В начале предлагается метод построения модели данных по программе, что позволяет разделить вычислительные и управляющие состояния. Затем рассматривается преобразование программы к приведенной форме, что позволяет далее рассматривать только программы, записанные в ней.

В *главе 4* предлагаются методы преобразования программ в систему взаимодействующих автоматов. При этом рассматриваются как неформальные, так и формальные методы преобразования. В начале главы разрабатывается метод построения системы взаимодействующих конечных автоматов, позволяющей исполнять программу в прямом направлении. Далее предлагается метод построения по программе системы автоматов, поддерживающей

обратимое исполнение. В заключение для предложенных методов доказана корректность и другие свойства.

В *главе 5* предлагается язык описания визуализаторов, основанный на *XML*. При этом отдельно рассматриваются структуры описаний визуализируемого алгоритма и конфигурации визуализатора.

В *главе 6* описываются результаты внедрения разработанных методов. В начале рассматривается система визуализации *Vizi*, построенная на основе методов и подходов, разработанных в предыдущих главах. Далее приводится пример построения визуализатора на основе системы *Vizi*. Затем производится сравнение полученных результатов с существующими подходами и приводится информация о практическом внедрении системы *Vizi* и визуализаторов, построенных на ее основе.

ГЛАВА 1. СИСТЕМЫ ВИЗУАЛИЗАЦИИ АЛГОРИТМОВ ДИСКРЕТНОЙ МАТЕМАТИКИ

Визуализаторы алгоритмов широко используются в мире при обучении программированию и дискретной математике. Рассмотрим материал, накопленный по этой теме, с различных точек зрения.

В разделе 1.1 рассмотрено применение визуализаторов в учебном процессе. При этом в разделе 1.1.2 сформулированы требования к визуализаторам, применяемым при обучении.

Далее в разделе 1.2 выполнен анализ визуализаторов алгоритмов на примере визуализаторов сортировок и отмечены их недостатки.

В разделе 1.3 рассмотрены имеющиеся системы визуализации и также указаны их недостатки.

1.1. Применение визуализаторов в учебном процессе

Визуализаторы алгоритмов широко применяются для обучения алгоритмам дискретной математики, в том числе при обучении автоматизированному проектированию [10, 11, 25, 36, 97]. Исследования [49] показали, что применение визуализаторов помогает учащимся глубже и быстрее понять объясняемый материал. При этом также выявлена большая заинтересованность учащихся в лекциях с применением визуализаторов, чем без них.

1.1.1. Варианты применения визуализаторов

На основе анализа литературы и многолетнего опыта применения визуализаторов на кафедре «Компьютерные технологии» в СПбГУ ИТМО выделены основные варианты применения визуализаторов алгоритмов в учебном процессе.

В начале визуализаторы алгоритмов использовались как вспомогательный материал на лекциях [32]. При этом преподаватель заранее отбирал визуализаторы, соответствующие теме, и готовил к ним наборы

входных данных. На лекции при помощи выбранных визуализаторов преподаватель демонстрировал работу алгоритмов, поясняя их.

Также визуализаторы могут быть использованы для повышения интереса учащихся к излагаемому на лекциях материалу. Одним из вариантов такого применения является отображение некоторого состояния алгоритма, и предложение учащимся определить какое действие выполнит алгоритм на следующем шаге. Полученные ответы обучаемых проверяются на визуализаторе путем совершения шага вперед. Визуализатор также позволяет подробно разобрать, почему было осуществлено именно это действие, а не другое. В таком режиме визуализаторы могут быть использованы и для проверки знаний.

Третий способ применения визуализаторов — начальное знакомство с алгоритмом. Учащийся сначала работает с визуализатором, составляя для себя общее представление об алгоритме. Впоследствии преподаватель дает полное описание алгоритма. После этого учащийся может вернуться к работе с визуализатором.

Четвертый способ заключается в том, что учащиеся сами строят визуализаторы, досконально изучая при этом визуализируемые алгоритмы. Полученные таким образом визуализаторы могут быть опубликованы в сети *Internet* и использованы приведенными выше способами.

Важной областью использования визуализаторов являются самостоятельное и дистанционное обучение. В этих случаях большое значение приобретает возможность задания пользователем входного набора данных. Таким 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. *Возможность ввода данных, на которых демонстрируется алгоритм (интерактивность)* — учащиеся должны иметь возможность задавать наборы входных данных и рассматривать работу алгоритма на них.
2. *Двунаправленность* — при работе с визуализатором должна быть возможность совершать шаги алгоритма как вперед, так и назад.
3. *Автоматический режим работы* — наряду с пошаговым режимом, визуализаторы должны предоставлять возможность работы без вмешательства пользователя.
4. *История* — визуализатор должен обеспечивать возможность пошагового возврата назад, вплоть до начального состояния.
5. *Отображение хода выполнения алгоритма* — визуализатор должен иметь возможность отображать не только изменения в данных, производимые визуализируемым алгоритмом, но и другие действия, например, сравнения при сортировке.
6. *Наличие комментариев для шагов алгоритма (комментарии)* — на каждом шаге алгоритма должны отображаться комментарии, поясняющие производимое действие.
7. *Простота использования* — интерфейс визуализатора должен быть интуитивно понятен.
8. *Свободный доступ* (*доступность*) — у учащихся должна быть возможность доступа к визуализаторам не только в аудиториях.
9. *Платформонезависимость* — визуализатор должен работать на распространенных аппаратных конфигурациях (*IBM-PC* совместимые компьютеры, *Apple Macintosh* и т.д.) и операционных системах (*Windows*, *Unix*, *Linux*, *Mac OS* и т.д.).
10. *Автономность* — при работе визуализатор не должен требовать подключения к вычислительным сетям.

В скобках приведены сокращенные названия требований, которые будут употребляться в дальнейшем.

Системы визуализации должны позволять строить визуализаторы, удовлетворяющие указанным требованиям. Кроме того, к системам визуализации предъявляются дополнительные требования.

11. *Удобство создания визуализаторов* — простота использования системы визуализации при разработке визуализаторов.
12. *Скорость разработки визуализаторов* — система визуализации должна позволять достаточно быстро разрабатывать визуализаторы, в частности, на основе уже существующих компонент.

Поясним обоснованность указанных требований.

В работе [49] показано, что интерактивность, двунаправленность и история позволяют более глубоко изучать работу алгоритмов, по сравнению с визуализаторами, не обладающими этими свойствами.

Простота использования важна при самостоятельном обучении, так как в этом случае 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 автономность позволяют использовать визуализаторы как на лекциях и практических занятиях, так для самостоятельного обучения.

1.2. Обзор визуализаторов на примере алгоритмов сортировок

Произведем обзор визуализаторов алгоритмов на примере алгоритмов сортировок, так как они являются одной из основополагающих тем при обучении дискретной математике и входят как важная составная часть во многие алгоритмы.

1.2.1. Подходы к визуализации алгоритмов сортировки

Рассмотрим несколько подходов к визуализации алгоритмов сортировки:

- «числовой» — сортируемый массив отображается набором содержащихся в нем чисел. При сравнении и обмене элементов они подсвечиваются. В некоторых случаях применяется анимация обмена элементов [36];
- «гистограммный» — сортируемый массив изображается в виде гистограммы. При этом обмен элементов визуализируется как физическое перемещение столбцов, соответствующих числам [40];
- «исторический» — весь процесс сортировки представляется в виде одного изображения. При этом обычно значения элементов отображают цветом [36].

На рис. 1 приведены примеры изображений, генерируемых визуализаторами, использующими описанные подходы.

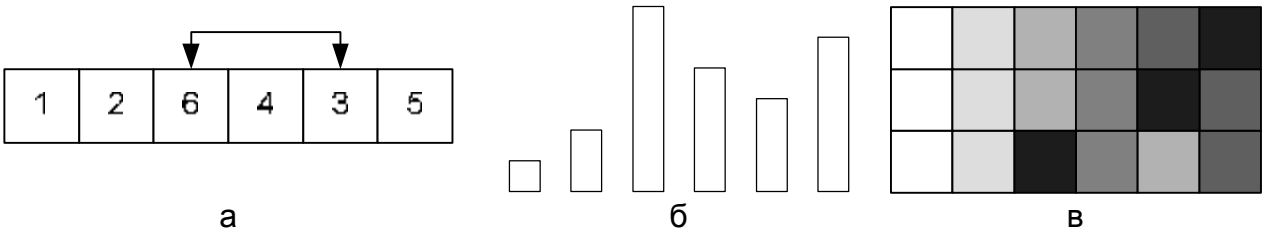


Рис. 1. Подходы к визуализации алгоритмов сортировки «числовой» (а), «гистограммный» (б) и «исторический» (в)

Обычно визуализируется следующий «стандартный» набор алгоритмов сортировки: обменная, пузырьковая, простого выбора, простой вставки, пирамидальная, слиянием и быстрая [18].

1.2.2. Обзор визуализаторов алгоритмов сортировок

В приводимом ниже обзоре в заглавии подразделов приводится название университета, в котором выполнен визуализатор.

Brown University (BALSA)

Система визуализации *BALSA* является одним из первых опытов создания визуализаторов алгоритмов (раздел 1.3.1). Принципы построения этой системы описаны работе [36].

Эта система позволяет проводить визуализацию одновременно на основе нескольких подходов, например, «гистограммного» и «числового». При этом соответствующие изображения обновляются синхронно с изменением данных. Также имеется возможность использовать «исторический» подход, при котором на одном изображении показывается процесс сортировки в целом.

Визуализаторы, в основном, демонстрируют изменения в сортируемых данных. При этом отображение алгоритма и комментариев отсутствует, а процесс визуализации комментируется преподавателем. Сама система позволяет отображать комментарии и шаги алгоритма, но это практически не используется.

Визуализация выполняется с помощью заранее написанных программ на языке сценариев. Возможность ручного ввода данных не предусмотрена.

Визуализаторы выполняются в виде модулей, встраиваемых в систему *BALSA*, и не являются платформонезависимыми. Это также ограничивает их доступность.

State University of New York, College at Brockport

Коллекция визуализаторов сортировок расположена по адресу [87]. Представлен «стандартный» набор визуализаторов сортировок, созданных на общем ядре. При этом для алгоритма пирамидальной сортировки использовано специальное решение.

При визуализации применяется «гистограммный» подход. Визуализация выполняется на заданном при создании визуализатора наборе данных. Существует возможность осуществлять шаги назад по алгоритму, но проверка, выполненная автором, показала, что она не работает на некоторых наборах входных данных.

Присутствуют комментарии к текущим действиям, но алгоритмы представлены только на сопроводительных страницах.

Визуализаторы выполнены в виде *Java*-апплетов, что определяет их доступность и платформонезависимость. Для визуализации связи с сервером не требуется. Таким образом, эти визуализаторы являются автономными.

Princeton University

Визуализаторы исполнены в виде единого *Java*-апплета, доступного по адресу [84]. В дополнение к «стандартному» набору визуализаторов сортировки визуализируются алгоритмы построения выпуклых оболочек.

Для визуализации применяется как «числовой», так и «гистограммный» подходы с возможностью выбора. Исходный набор данных может быть задан только случайно.

При визуализации алгоритмы не отображаются, так как использована визуализация данных (раздел 1.3.2). По той же причине производимые действия не комментируются. Третий недостаток — невозможность осуществлять шаги назад по алгоритму.

Hope College

Визуализаторы сортировки, выполненные в единой оболочке, доступны по адресу [79]. Для визуализации используется «гистограммный» подход. Ввод исходных данных невозможен. Трассировка алгоритма в обратном направлении не предусмотрена.

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

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

University of Joensuu (Jeliot)

В примерах к системе визуализации *Jeliot* [80] приведены только визуализаторы пузырьковой и быстрой сортировок.

При выполнении алгоритмов используется низкоуровневая визуализация, отображающая не только выполнение операторов, но и вычисление выражений. При этом подсвечивается текущий оператор.

Ввод исходных данных выполняется через специальные классы ввода-вывода. Трассировка алгоритма в обратном направлении невозможна.

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

СПбГУ ИТМО (Казаков М.А.)

В работе [14] описан визуализатор пирамидальной сортировки, доступный по адресу [69]. Данный визуализатор основан на автоматном подходе, изложенном в работе [29].

При визуализации на экране отображаются состояние массива, пирамида и комментарий к выполняемому действию. Ввод исходных данных не предусмотрен. Алгоритм трассируется только в прямом направлении.

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

1.2.3. Анализ визуализаторов алгоритмов сортировок

Рассмотрим описанные визуализаторы с точки зрения требований, изложенных в разделе 1.1.2.

Сравнительная характеристика рассмотренных визуализаторов приведена в таблице 1. Ее столбцы соответствуют свойствам, указанным в разделе 1.1.2. При этом используются следующие обозначения:

«+» — свойство выполняется;

«–» — свойство не выполняется;

«±» — свойство выполняется частично.

Таблица 1. Сравнительные характеристики визуализаторов алгоритмов сортировок

Визуализаторы	1	2	3	4	5	6	7	8	9	10
Brown University	–	+	±	+	±	±	–	–	–	–
SUNY Brockport	–	±	+	±	–	+	–	+	+	+
Princeton University	±	–	+	–	–	–	±	+	+	+
Hope College	–	–	+	–	+	–	+	+	+	+
Jeliot	+	–	–	–	+	–	+	+	±	+
СПбГУ ИТМО (Казаков М.А.)	–	–	+	–	–	+	+	+	+	+

Из приведенной таблицы следует, что не один из рассмотренных визуализаторов не удовлетворяет всем выдвинутым требованиям. Поэтому необходимо разработать новую систему визуализации, позволяющую строить визуализаторы, для которых все указанные свойства выполняются.

1.3. Обзор систем визуализации

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

1.3.1. Развитие систем визуализации

История создания визуализаторов алгоритмов насчитывает более двадцати пяти лет. Первый визуализатор был разработан в *Bell Telephone Laboratories* в 1966 году для пояснения работы связных списков [46, 47]. Широкое распространение визуализаторы стали получать в начале 80-х.

В 1981 году в Университете Торонто был создан фильм о сортировках [32], 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*) [36] и *TANGO* (*Transition-based Animation GeneratiOn*) [62]. Эти системы оказали большое влияние на последующие разработки в данной области и легли в основу книги [34].

Система визуализации алгоритмов *BALSA* разработана М. Брауном и Р. Седжвиком в *Brown Univesity* (США). Она стала первой широко

распространенной системой визуализации. Система *BALSA* может одновременно отображать как несколько видов одной программы, так и виды разных программ. В последствии М. Браун и Р. Седжвик приняли участие в разработке других систем визуализации, в которых были исправлены некоторые недостатки *BALSA*, и, в частности, уменьшены требования к вычислительным ресурсам.

Система визуализации алгоритмов *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* [63] (версия *TANGO* для *XWindows*), в которой была реализована плавная анимация. Другим последователем *TANGO* явилась система визуализации *Polka* [65], оптимизированная для визуализации параллельных программ. В последствии в систему *Polka* была добавлена возможность трехмерной визуализации.

Распространенными системами визуализации являются также *Zeus* [35], *Leonardo* [37], *CATAI* [43] и *Mocha* [33]. Первая из них была разработана М. Брауном и являлась развитием системы *BALSA*. Она поддерживает возможность отображения нескольких синхронизированных видов программы и может визуализировать параллельные программы. Система *Leonardo* является интегрированной средой для разработки визуализаторов на языке *C*. *CATAI* также предназначена для визуализации программ на языке *C*. В ней объединены возможности разработки и визуализации программ. Система *Mocha* была разработана в Университете им. Брауна (США). В ней использован клиент-серверный подход. Клиенты были реализованы как на языке *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 платформозависимыми. В последствии, с распространением языка *Java*, был осуществлен переход на него. Таким образом, системы визуализации стали платформонезависимыми. Применение языка *Java* также позволила легко

выкладывать визуализаторы в сети *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 системы могут быть использованы для исследования и создания новых алгоритмов. Ряд систем достаточно просты (например, *Animal* [60] и *Agat* [71]), другие же весьма сложны (например, *Leonardo*).

В 2001 году в работе [15] было предложено использовать автоматный подход для построения визуализаторов. На основе этого подхода было разработано несколько визуализаторов [12−14]. Предложенный подход был основан на ручном построении графов переходов автоматов, что является трудоемкой и нетривиальной задачей.

1.3.2. 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 применяться 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 создаваемых 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 событий* (interesting events) [40]. При подготовке визуализатора в интересные точки программы вставляются вызовы процедур визуализации, генерирующие события. Таким образом, это метод визуализации является императивным.

В дальнейшем будем называть интересные точки программы *интересными состояниями*, что связано с применением в настоящей работе автоматного подхода для построения логики визуализаторов.

Визуализаторы данных 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, [40]), напрямую не поддерживается и выполняется искусственными методами. Вторым недостатком данного подхода является невозможность отображения комментариев к выполняемым 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.

Сравнительный обзор визуализаторов программ и данных выполнен в работе [40].

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

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 [72] и описанная в работе [60]. Опыт использования системы *Animal* в *University of Siegen* (Германия) изложен в работе [54].

Система создавалась для удовлетворения следующим требованиям [60]:

- платформонезависимость;
- доступность;
- простота использования;
- поддержка визуализации программ.

Визуализация в системе *Animal* создается как последовательность кадров, отображающих действия алгоритма. Кадры задаются в специальном редакторе. Также имеется возможность описания движения элементов кадра, что позволяет сделать визуализацию более наглядной.

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

Существенным недостатком этой системы визуализации является ручное задание 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 сети *Internet*. Подробное описание системы *JAWAA* дано в работе [55]. Использование *JAWAA* в *Duke University* (США) описано в работах [58, 59].

Визуализаторы в системе *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 языках *C/C++*. Таким образом, система *JAWAA* может быть использована как визуализатор программ (раздел 1.3.2).

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

1.3.4. Обзор систем визуализации алгоритмов

Рассмотрим существующие системы визуализации с точки зрения соответствия требованиям, выдвинутым в разделе 1.1.2. С более подробным обзором можно ознакомиться в работе [67].

BALSA

BALSA является первой системой визуализации, получившей широкую известность. Она была разработана в *Brown University* (США) и основана на визуализации программ. Система *BALSA* и опыт ее использования подробно описаны в работах [34, 36]. При просмотре визуализатора можно открыть несколько видов программы — различных изображений структур данных и текущей выполняемой операции. Например, доступны «исторические» виды (раздел 1.2).

Визуализаторы описываются на специально разработанном языке описания сценариев. Для этого требуется достаточно высокая квалификация, что признавалось и самими авторами [36].

Недостатком системы *BALSA* и систем *BALSA-II* и *Zeus*, построенных на той же идеологии, является сложность описания визуализаторов и ручное задание действий, осуществляемых при обратном проходе по алгоритму. Другими недостатками является слабая переносимость и невозможность задания входных данных пользователем.

Tango и XTango

Tango была первой системой визуализации данных. Она разработана в *Brown Univesity* (США). В последствии была создана система *XTango* — версия системы *Tango*, работающая под *XWindow*. Системы *Tango* и *XTango* подробно описаны в работах [62, 63, 64].

Визуализация в этих системах 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 *University of Joensuu* (Финляндия). Она является *Java*-версией системы визуализации *Eliot*, разработанной там же. Система доступна по адресу [80]. Описание системы *Jeliot* приведено в работах [52, 53], а описание системы *Eliot* — в работе [48].

Для создания визуализируемых программ применяется язык *Java*. При визуализации Java-код преобразуется во внутреннее представление. Визуализация низкоуровневая, в частности, показывается вычисление всех выражений, без возможности их пропуска. Выполняется подсветка текущего оператора и отображается стек вызовов. Также поддерживается анимационный режим.

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

Основным недостатком системы *Jeliot* является отсутствие возможности контроля визуализации — возможности влиять на изображения, которые показываются пользователю. Это, в совокупности с низкоуровневой визуализацией, делает систему *Jeliot* малопригодной для визуализации сложных алгоритмов.

Polka и Samba

Polka — система визуализации, разработанная в *Georgia Institute of Technology* (США) [65]. *Samba* является графическим интерфейсом к системе *Polka* и образует с ней единую (более развитую) систему визуализации. Поэтому они рассматриваются совместно. Системы *Polka* и *Samba* доступны по адресам [83] и [86] соответственно.

Система визуализации данных *Polka* (развитие системы *XTango*), специализирована на визуализации параллельных программ. С этой целью пользователь может одновременно открыть несколько окон с графической информацией. Описание визуализатора выполняется на специальном языке описания сценариев, программы на котором можно создавать с помощью системы *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 системы *Polka* является платформозависимость созданных на ее базе визуализаторов. Также в этой системе 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 [81]. Принципы ее работы и построения описаны в работах [37, 38].

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, построенные при помощи системы *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 описана в работе [71] и доступна по адресу [74].

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

На встроенном языке сценариев 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 анализировать алгоритмы и разрабатывать на основе полученных данных новые алгоритмы.

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

1.3.5. Анализ систем визуализации

Обобщим материал разделов 1.3.3 и 1.3.4, сравнив системы визуализации на основе требований, изложенных в разделе 1.1.2. Столбцы таблицы 2 соответствуют требованиям, как указано в разделе 1.1.2. При этом «+» значит что требование удовлетворено, «–» — не удовлетворено, а «±» — возможно удовлетворить требование путем сложной доработки визуализатора.

Таблица 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 систем визуализации не удовлетворяет всем выдвинутым требованиям. Наиболее полно им удовлетворяет система *Animal*, однако визуализаторы, созданные с ее использованием, не обеспечивают:

- ввод данных пользователем;
- двунаправленность;
- автоматизацию создания логики визуализаторов.

Таким образом, задача разработки системы визуализации, удовлетворяющей всем перечисленным в разделе 1.1.2 требованиям, является актуальной.

Выводы по главе 1

1. Сформулированы требования к визуализаторам алгоритмов дискретной математики.
2. Ни одна из рассмотренных систем визуализаторов не позволяет создавать визуализаторы, которые полностью удовлетворяют требованиям, изложенным в разделе 1.1.2.
3. Требуется реализовать систему визуализации, удовлетворяющую всем выдвинутым требованиям.
4. Визуализаторы данных обладают меньшими выразительными способностями по сравнению с визуализатором программ. Поэтому новая система визуализации должна обеспечивать возможность визуализации программ.

ГЛАВА 2. ПРОЦЕСС ПОСТРОЕНИЯ ВИЗУАЛИЗАТОРОВ

Процесс построения визуализатора алгоритма достаточно сложен, поэтому разобьем его на отдельные составляющие и рассмотрим с нескольких точек зрения.

В разделе 2.1 описываются основные варианты использования визуализатора и на их основе разрабатывается структура визуализатора. Затем предлагается порядок «ручной» разработки визуализаторов и выделяются этапы, которые могут быть автоматизированы (раздел 2.2). Далее рассматриваются подходы к построению основных компонент, требующихся для автоматизации процесса построения визуализатора (разделы 2.3−2.5). В конце главы формулируются теоретические и практические задачи, решаемые в диссертационной работе (раздел 2.6).

2.1. Структура визуализатора

Визуализатор является сложным программным продуктом, поэтому для его построения необходимо произвести анализ, позволяющий выделить отдельные части визуализаторы, которые могут быть рассмотрены самостоятельно.

В разделе 2.1.1 выделяются основные варианты использования визуализатора алгоритма. На их основе разрабатывается структура визуализатора (раздел 2.1.2).

2.1.1. Варианты использования визуализатора

Диаграмма вариантов использования визуализатора, построенная в результате анализа материала, изложенного в главе 1, приведена на рис. 2. Основным вариантом использования, включающим в себя несколько подвариантов, является «работа с визуализатором».

В таблице 3 приведены источники вариантов использования. При этом варианты использования «Запуск алгоритма» и «Анализ состояния алгоритма» были получены как обобщение других вариантов использования.

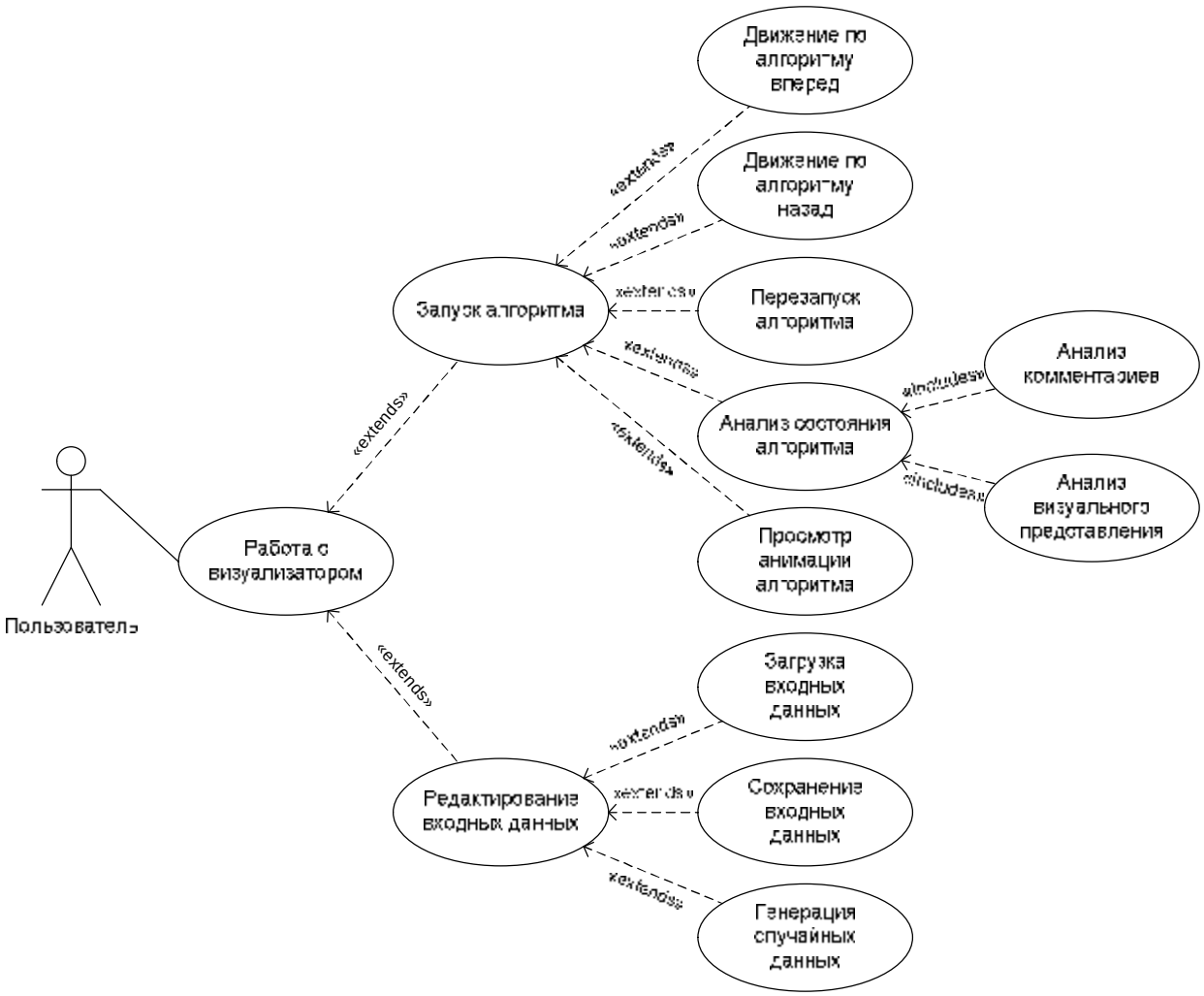


Рис. 2. Диаграмма вариантов использования визуализатора

Таблица 3. Источники вариантов использования

Вариант использования	Источник
Работа с визуализатором	
Запуск алгоритма	
Движение по алгоритму вперед	раздел 1.1.2, требование 2
Движение по алгоритму назад	раздел 1.1.2, требования 2 и 4
Перезапуск алгоритма	раздел 1.1.2, требования 2 и 4
Анализ состояния алгоритма	
Анализ комментариев	раздел 1.1.2, требование 6
Анализ визуального представления	раздел 1.1.2, требование 5
Просмотр анимации алгоритма	раздел 1.1.2, требование 3
Редактирование входных данных	раздел 1.1.2, требование 1
Загрузка входных данных	раздел 1.1.1
Сохранения входных данных	раздел 1.1.1
Генерация случайных входных данных	раздел 1.1.1

2.1.2. Выделение основных частей визуализатора

На основании имеющегося опыта и вариантов использования из визуализатора можно выделить следующие основные части, диаграмма компонентов которых приведена на рис. 3:

- *логика визуализатора*;
- *модель данных*;
- *визуальное представление*;
- *набор комментариев*;
- *элементы управления*;
- *интерфейс визуализатора*;
- *проектная документация*.

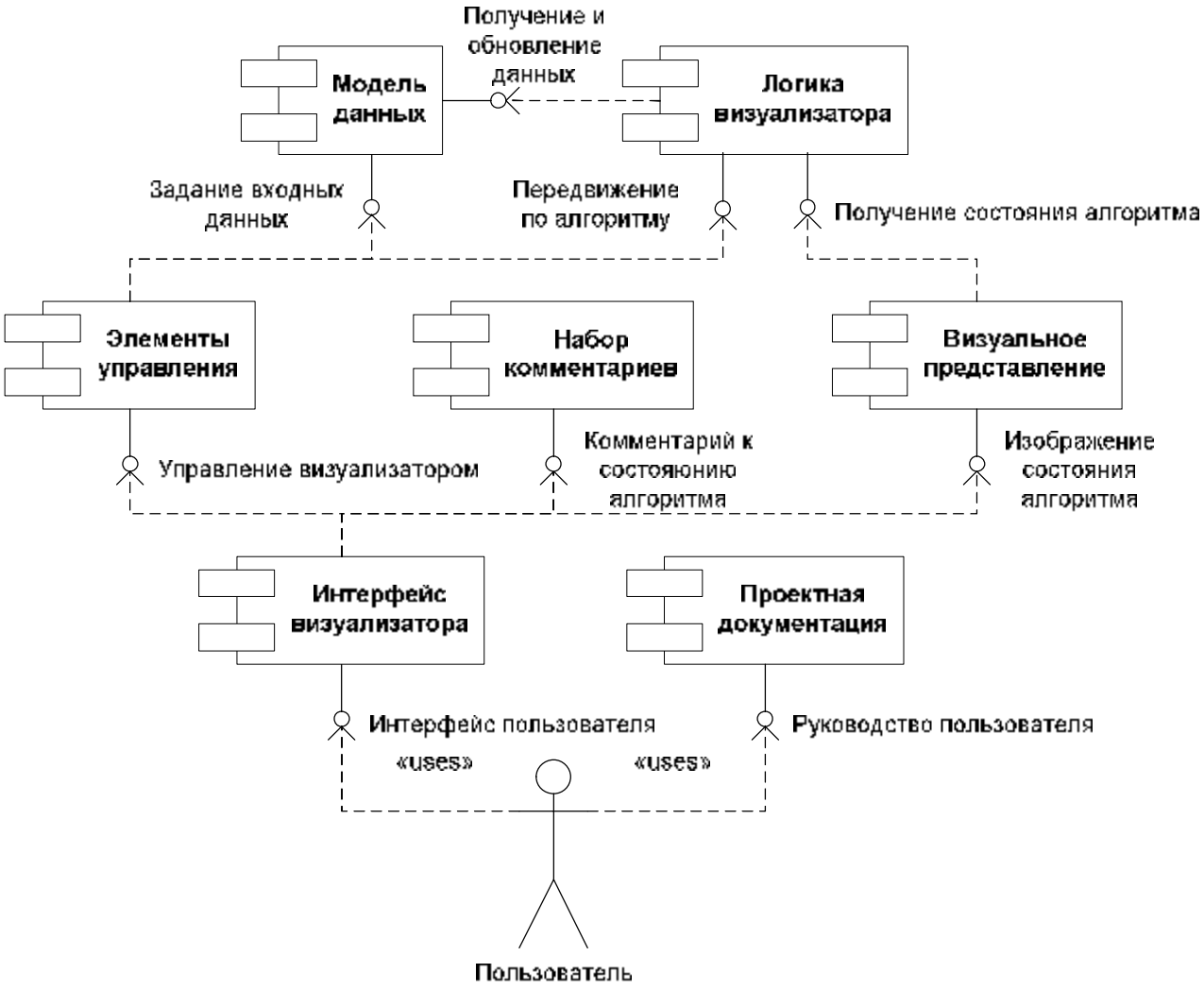


Рис. 3. Диаграмма компонентов основных частей визуализатора

Опишем подробнее предназначение каждой из частей визуализатора.

Логика визуализатора — часть визуализатора, обеспечивающая возможность трассировки алгоритма. При этом логика визуализатора обновляет модель данных так, чтобы она отражала текущее вычислительное состояние алгоритма (значения переменных).

В алгоритме выделяются интересные состояния, в которых информация отображается пользователю.

Например, в алгоритме поиска максимального потока в сети вряд ли стоит визуализировать поиск кратчайшего пути между вершинами. Такие «подалгоритмы» не содержат интересных состояний и отображаются как одна операция.

В соответствии с требованиями, выдвинутыми в разделе 1.1.2, логика визуализатора должна обеспечивать возможность движения по алгоритму на произвольное число шагов как вперед, так и назад.

Модель данных — часть визуализатора, хранящая значения переменных, используемых в программе, и предоставляющая к ним доступ другим частям визуализатора.

Отметим, что в обычных программах только сама программа имеет доступ к своим данным. Таким образом, в них явное выделение модели данных не требуется.

Визуальное представление — часть визуализатора, определяющая, что и как будет отображаться пользователю в интересных состояниях. Обычно визуальное представление задается набором слайдов (схем слайдов), которые отображаются пользователю в процессе визуализации. При этом слайды могут отображать информацию, предоставляемую моделью данных.

Визуальное представление предназначено для облегчения понимания визуализируемого алгоритма. Например, 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 должно быть решено, как

будет отображаться граф. Например, при визуализации алгоритма Флойда естественным является представление графа матрицей весов, а для алгоритма Краскала — списками ребер.

Набор комментариев — часть визуализатора, определяющая какие комментарии будут отображаться пользователю в каждом интересном состоянии. Комментарий, отображаемый пользователю, может включать в себя данные, предоставленные моделью данных.

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 даже без визуального представления.

Элементы управления — часть визуализатора, через которую пользователь управляет визуализатором. При этом определяются набор действий, которые пользователь может выполнять с визуализатором, и связь этих действий с логикой визуализатора. Примерами таких действий могут служить трассировка в прямом и обратном направлениях (здесь могут быть использованы малые и большие шаги), изменение количества элементов (вершин в графе, размерность массива), генерация случайных исходных данных и другие возможности.

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

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

2.2. Разработка визуализаторов

В настоящем разделе рассматривается порядок «ручной» разработки визуализаторов (раздел 2.2.1) и возможные пути его автоматизации (раздел 2.2.2).

2.2.1. «Ручная» разработки визуализаторов

В целях выделения шагов, поддающихся автоматизации, рассмотрим порядок «ручной» разработки визуализаторов, состоящий из 12 этапов. Для наглядности некоторые этапы дополнительно разбиты на отдельные шаги.

1. Постановка задачи и анализ литературы.
2. Создание визуализируемой программы:
    - реализация алгоритма;
    - отладка программы, реализующей алгоритм.
3. Проектирование визуализатора:
    - выделение «интересных» состояний;
    - проектирование визуального представления;
    - проектирование набора комментариев;
    - проектирование элементов управления.
4. Построение модели данных.
5. Построение и отладка логики визуализатора.
6. Реализация набора комментариев.
7. Реализация визуального представления.
8. Реализация элементов управления.
9. Интеграция модели данных, логики визуализатора, визуального представления, набора комментариев и элементов управления.
10. Создание интерфейса визуализатора.
11. Общая интеграция и отладка визуализатора.
12. Оформление проектной документации.

Приведем подробные описания отдельных шагов.

На первом этапе производится постановка задачи и анализ литературы. При этом рассматриваются модификации алгоритма и одна из них выбирается для визуализации.

На втором этапе создается и отлаживается программа, реализующая выбранную модификацию алгоритма. При этом выделяются шаги алгоритма, требующие особого внимания. Полученная таким образом информация используется на следующем этапе.

На третьем этапе выполняется проектирование визуализатора. Для этого сначала выделяются те шаги алгоритма, которые представляют наибольший интерес, и соответствующие «интересные» состояния. Затем проектируется концепция визуального представления и ее конкретизация для каждого выделенного шага. Одновременно с проектированием визуального представления для каждого шага пишутся комментарии, поясняющие действия, выполняемые алгоритмом.

После этого проектируются элементы управления визуализатором, в частности, определяется какие параметры и в каких пределах сможет регулировать пользователь.

На четвертом этапе производится построение модели данных визуализируемой программы. При этом программа преобразуется таким образом, чтобы использовать модель данных.

На пятом этапе выполняются построение и отладка логики визуализатора. Построение логики визуализатора может оказаться не только трудоемким, но и сложным с идейной точки зрения, так как требуется создание программы, позволяющей производить трассировку, как в прямом, так и в обратном направлении.

На шестом, седьмом и восьмом этапах выполняется реализация проекта визуализатора, разработанного на третьем этапе.

На девятом этапе производится интеграция результатов четвертого — восьмого этапов. При этом могут возникнуть сложности, для решения которых придется вернуться на один из предыдущих этапов.

Десятый этап может быть выполнен один раз при разработке набора визуализаторов. Это может существенно уменьшить затраты времени.

На одиннадцатом этапе производится окончательная интеграция визуализатора и его отладка. Найденные ошибки для их исправления могут потребовать возврата к предыдущим этапам.

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

Отметим, что по сравнению с работой [12] предлагаемый порядок построения визуализаторов не ориентирован на конкретную технологию.

2.2.2. Автоматизация разработки визуализаторов

Выделим из предложенного порядка разработки визуализаторов шаги и этапы, которые можно автоматизировать.

Первые три этапа построения визуализатора по своей сути являются творческими, и поэтому автоматизированы быть не могут.

Выделение модели данных может производиться автоматически по исходному тексту программы, реализующей алгоритм. Таким образом, четвертый этап может быть автоматизирован.

Автоматизация построения логики визуализатора является трудоемким процессом, но она необходима, так как на данном этапе часто допускают ошибки, особенно при реализации обратного хода алгоритма. Таким образом, автоматизация четвертого этапа позволит существенно сократить время разработки визуализатора.

Реализация набора комментариев зависит от модели данных и логики визуализатора. При этом механизм выдачи комментариев к текущему состоянию может быть разработан один раз. Таким образом, шестой этап может быть автоматизирован, при условии автоматизации четвертого и пятого этапов.

Как показывают исследования, полная автоматизация построения визуального представления возможна, но получаемые визуализаторы часто не

понятны и малоинформативны [39]. С другой стороны, данный этап может быть упрощен за счет применения библиотек компонентов визуального представления.

Элементы управления могут быть поделены на элементы, применяемые в большинстве визуализаторов (такие как, «Сделать шаг вперед (назад)»), и элементы, специфичные для конкретного визуализатора (например, графическое задание входных данных). При этом элементы управления первого типа могут предоставляться системой визуализации.

Интеграция модели данных, логики визуализатора, набора комментариев визуального представления и элементов управления, предоставляемых системой визуализации, может быть автоматизирована за счет разработки языка описания визуализатора. Интеграция элементов управления, специфичных для визуализатора, автоматизирована быть не может.

Основной интерфейс визуализатора может быть создан один раз, а затем применяться для построения многих визуализаторов. При этом общая интеграция и отладка визуализатора может быть автоматизирована.

Таким образом, шаги 4−6 и 10 могут быть автоматизированы полностью, а шаги 7−9 и 11 — частично. Шаги 1−3 и 12 автоматизированы быть не могут. Для автоматизации требуется разработать:

- метод автоматизации построения модели данных по программе;
- метод построения программ, обеспечивающий трассировку программы в прямом и обратном направлениях;
- язык описания визуализаторов.

2.3. Модель данных визуализатора

Модель данных и логика визуализатора тесно взаимодействуют друг с другом. При этом модель данных отвечает за хранение данных о текущем состоянии визуализатора.

Рассмотрим требования, предъявляемые к модели данных различными частями визуализатора (раздел 2.3.1), и предлагаемый способ организации переменных в модели данных (раздел 2.3.2).

2.3.1. Требования к модели данных

С точки зрения логики визуализатора от модели данных требуется:

1. Хранение данных о текущем состоянии визуализатора.
2. Возможность изменения хранимых данных.
3. Хранение информации, достаточной для обеспечения обратной трассировки программы.

Модель также должна предоставлять данные для визуального представления и комментариев. При этом к ней предъявляется дополнительное требование:

4. Удобство доступа к данным о текущих значениях переменных.

Отметим, что построение логики визуализаторов должно быть автоматизировано, поэтому удобство доступа к модели данных с этой точки зрения не играет особой роли.

2.3.2. Подходы к построению модели данных

Традиционными являются следующие подходы к размещению локальных переменных:

1. *Статическое выделение памяти.* Для каждой переменной выделяется фиксированное место в памяти [56].
2. *Стековое выделение памяти.* Все данные хранятся в едином стеке. При входе в процедуру создается новый кадр стека, в котором располагаются локальные переменные [57].

Первый подход не может быть применен для рекурсивных программ. Поэтому его использование для построения модели данных невозможно.

Второй подход позволяет реализовывать рекурсивные программы, но удобный доступ обеспечивается только к переменным процедуры, исполняемой в данный момент.

Данный недостаток может быть исправлен путем комбинации двух рассмотренных подходов. При этом для каждой локальной переменной выделяется фиксированное место в памяти. При рекурсивном вызове процедуры, локальные переменные предыдущего экземпляра процедуры сохраняются в стеке. Таким образом, с одной стороны обеспечивается поддержка рекурсивных программ, а с другой — удобный доступ к переменным активного последнего экземпляра каждой процедуры.

2.4. Логика визуализатора

Логика визуализатора отвечает за переходы между «интересными» состояниями визуализатора.

В настоящем разделе рассматриваются требования, предъявляемые к логике визуализатора (раздел 2.4.1), методы решения задачи об обратимом исполнении программ (раздел 2.4.2) и построение логики визуализатора на основе автоматного подхода (раздел 2.4.3).

2.4.1. Требования к логике визуализатора

Логика визуализатора должна обеспечивать:

1. Трассировку алгоритма в прямом направлении с остановками в «интересных» состояниях.
2. Трассировку алгоритма в обратном направлении с остановками в «интересных» состояниях.
3. Предоставлять доступ к информации о текущем «интересном» состоянии.
4. Сообщать другим частям визуализатора об изменении текущего состояния визуализатора.

Первые два требования позволяют пользователю визуализатора производить трассировку в прямом и обратном направлениях.

Третье и четвертое требования обеспечивают взаимодействие модели данных с другими частями визуализатора.

2.4.2. Подходы к реализации обратимого исполнения

При построении визуализаторов наибольшую сложность представляет обеспечение возможности обратной трассировки программ. Большинство проанализированных автором визуализаторов либо вовсе не имеют указанной возможности, либо ее реализация содержит ошибки. Таким образом, визуализатор должен решать задачу обратимого исполнения программ [68].

Существуют различные подходы к задаче обратимого исполнения.

1. Построение процессоров, поддерживающих обратимое исполнение программ [61].
2. Эмуляция процессоров, поддерживающих обратимое исполнение программ [37].
3. Построение программ, поддерживающих обратимое исполнение [68].

Первый подход не применим для построения визуализаторов, так как широко распространенные процессоры не поддерживают обратимое исполнение программ.

Второй подход может быть применен для построения визуализаторов алгоритмов, но он требует написания специализированных компиляторов и эмулятора процессора. При этом эмулятор существенно тормозит исполнение программы и требует для своего выполнения большого количества памяти.

В рамках третьего подхода по исходной программе строится другая программа, которая обеспечивает обратимое исполнение исходной программе на обычных процессорах. В рамках данного подхода возможны следующие варианты:

1. *Пошаговое сохранение*. Для каждого шага сохраняются значение переменных на данном шаге. При совершении шага назад значения, сохраненные на предыдущих шагах, восстанавливаются.
2. *Перезапуск программы*. При движении вперед запоминается количество шагов, сделанное программой. Для совершения шага

назад программа перезапускается и останавливается, сделав на один шаг меньше.

3. *Обращение программы*. Для передвижения в обратную сторону строится программа, умеющая выполнять исходную программу в обратном направлении с любого места.

Первый вариант является простым для реализации, но требует большого количества памяти и времени для запоминания состояний программы.

Второй вариант прост в реализации и экономичен по памяти, но при выполнении программы в обратном направлении время растет квадратично по количеству шагов, что неприемлемо для сложных программ.

Третий вариант является наиболее сложным с точки зрения реализации, но позволяет добиться экономии памяти и времени по сравнению с первыми двумя методами.

Подход, заключающийся в построении программы, обратной к исходной, является наиболее перспективным.

2.4.3. Автоматный подход к построению логики визуализаторов

Начало формальному изучению систем с конечным числом состояний положено в работе [50].

Различные модели детерминированных конечных автоматов, которые обычно называют «конечными автоматами» или просто «автоматами», были разработаны в середине 50-х годов прошлого века [6, 42, 51, 23]. При этом если первые две из этих работ были посвящены синтезу схем (аппаратуры), то остальные носили более абстрактный характер. Считается, что итог этапу становления теории автоматов был подведен выпуском сборника статей [1], который, в частности, содержал работы [42, 23]. Недетерминированные автоматы и их эквивалентность детерминированным автоматам были рассмотрены в работе [24].

В дальнейшем применительно к построению аппаратуры теория автоматов «распалась» две взаимосвязанных теории: абстрактную и

структурную [7], для первой из которых характерно последовательная обработка информации, а для второй — параллельная.

В программировании автоматы начали применяться после появления работы [16], в которой были введены регулярные выражения и приведено доказательство их эквивалентности конечным автоматам. При этом абстрактная теория автоматов используется в основном для обработки текстов [2, 28, 66], а структурная — для программного управления [29].

С точки зрения автоматного подхода, выделяются *управляющие* и *вычислительные* состояния [31]. Их различие может быть проиллюстрировано на следующем примере «При создании вычислительной системы для банка имеет смысл выделить режимы нормальной работы и банкротства в разные управляющие состояния, так как в этих режимах поведение банка может существенно отличаться. В то же время, конкретные суммы денег в банковском балансе будут представлять вычислительное состояние» [19].

В работе [15] было предложено применить автоматный подход к построению визуализаторов. Эта идея была развита в работе [96] и показала свою состоятельность.

Для применения автоматного похода требуется выделить управляющие состояния. В случае визуализатора в качестве управляющего состояния можно рассматривать текущий исполняемый оператор, а вычислительному состоянию — значения переменных, хранимые в модели данных. При этом в зависимости от входных данных одному управляющему состоянию может соответствовать множество вычислительных состояний.

Отметим, что в случае одной нерекурсивной процедуры в качестве состояния достаточно взять номер оператора. В случае рекурсивной программы или программы из нескольких процедур требуется использование более сложной техники указания текущего оператора, например текстуальными индексами [41].

Отметим, что так как «интересное» состояние — это некоторое место в визуализируемой программе, то ему соответствует управляющее состояние. С

другой, стороны, не все управляющие состояния являются интересными. Поэтому требуется способ пометки «интересных» управляющих состояний.

Выделение управляющих состояний при построении визуализатора имеет следующие преимущества:

1. Комментарии и визуальное представление «привязаны» к состоянию, что упрощает их отображение.
2. По описанию автомата можно доказывать его свойства.
3. Для построения обратимой программы требуется обращать только действия на переходах.
4. Исследования, проводимые в СПбГУ ИТМО при участии автора, показали, что автоматный подход удобен при построении логики визуализаторов алгоритмов.

Таким образом, автоматный подход является наиболее перспективным для построения логики визуализаторов алгоритмов. При этом требуется дополнить существующие подходы с тем, чтобы они обеспечивали обратимое выполнение программ.

2.5. Язык описания визуализаторов

Язык описания визуализаторов должен обеспечивать:

1. Запись логики визуализаторов, близкую к обычной, и, в частности, поддержку:
   - процедур;
   - локальных и глобальных переменных;
   - основных операторов, в том числе:
     - оператора присваивания;
     - последовательность операторов (составной оператор);
     - укороченный оператор ветвления (`if-then`);
     - полный оператор ветвления (`if-then-else`);
     - цикл с предусловием (`while`);
     - вызов процедуры.

2. Указание комментариев к интересным состояниям.
3. Указание визуального представления к интересным состояниям.
4. Описание параметров визуализатора, необходимых для автоматизации построения визуализатора.
5. Удобство редактирование описания визуализатора.

Рассмотрим эти требования подробнее.

Как показывает опыт, только небольшое количество алгоритмов может быть выражено в виде одной процедуры [17, 18]. Таким образом, поддержка процедур требуется для реализации сложных, в частности, рекурсивных алгоритмов. Поддержка локальных и глобальных переменных также требуется из-за их частого применения.

Список поддерживаемых операторов составлен на основе теоремы структурирования [8, 21], в соответствии с которой любую программу можно преобразовать так, что она будет содержать только три типа управляющих конструкций: последовательность операторов, присваивание, цикл с предусловием. Номенклатура указанных операторов достаточна для записи любой программы. С другой стороны, для удобства записи программ в список добавлены операторы ветвления, а для поддержки процедурного программирования — оператор вызова процедуры. Будем назвать *приведенной формой* программы, такую ее запись, в которой используются только перечисленные выше операторы.

Отметим, что если программа содержит операторы других типов, то такие операторы должны быть заменены на допустимые операторы или их последовательности. Поэтому требуется разработка метода, позволяющего преобразовывать программы к приведенной форме.

Возможность указания комментариев к интересным состояниям и указания для них визуального представления требуется для автоматизации построения визуализатора.

Удобство редактирования является важным требованием, так как процесс построения визуализатора итеративен, и описание визуализатора часто изменяется.

2.6. Задачи, решаемые в диссертационной работе

Для автоматизации построения визуализаторов необходимо разработать подходы для решения следующих теоретических задач:

1. Автоматизированное выделение модели данных.
2. Автоматизированное преобразование программы к приведенной форме.
3. Автоматизированное преобразования программы в систему взаимодействующих конечных автоматов, которая будет обеспечивать трассировку программы в прямом и обратном направлениях.

Для практического внедрения теоретических результатов требуется решить следующие задачи:

1. Разработать язык описания визуализаторов, позволяющий автоматизировать их построение.
2. Разработать САПР для построения визуализаторов, реализующий разработанные методы и язык.

Результаты работы необходимо внедрить в учебный процесс по курсу «Дискретный анализ».

Выводы по главе 2

1. Разработана структура визуализаторов алгоритмов дискретной математики.
2. Предложен процесс «ручного» построения визуализаторов.
3. Выполнен анализ предложенного процесса и выделены этапы, которые будут автоматизированы.
4. Разработаны требования и проанализированы подходы к построению модели данных.

5. Разработаны требования и проанализированы подходы к построению логики визуализаторов алгоритмов.
6. Разработаны требования к языку описания визуализаторов.
7. Сформулированы теоретические и практические задачи, решаемые в диссертационной работе.

ГЛАВА 3. ПОСТРОЕНИЕ МОДЕЛИ ДАННЫХ И ПРЕОБРАЗОВАНИЕ ПРОГРАММЫ К ПРИВЕДЕННОЙ ФОРМЕ

В данной главе разрабатываются методы, позволяющие автоматизировать построение модели данных по исходному коду программы и преобразование программы к приведенной форме.

Модель данных предназначена для хранения вычислительных состояний визуализатора. Она выделяется для того, чтобы отделить вычислительные состояния визуализатора от управляющих состояний, хранимых логикой визуализатора.

Преобразование программы к приведенной форме позволяет упростить процесс построения системы взаимодействующих конечных автоматов, обеспечивающей трассировку программы в прямом и обратном направлениях.

В разделе 3.1 предлагается построение модели данных разбить на два этапа, и разрабатываются требования к исходной программе, которые позволят автоматизировать построение модели данных. В разделе 3.2 разрабатывается метод построения модели данных по итеративной программе. Этот метод адаптируется для построения модели данных по рекурсивной программе (раздел 3.3). Предложенные методы иллюстрируются на примерах.

В конце главы разрабатывается метод, позволяющий автоматизировать преобразование программы к приведенной форме (раздел 3.4).

3.1. Построение модели данных

Для автоматизации построения модели данных требуется разработать формальный подход, позволяющий построить модель данных по реализации алгоритма.

В начале рассмотрим разбиение процесса построения модели данных на этапы (раздел 3.1.1). Затем разработаем требования к исходной программе (раздел 3.1.2).

3.1.1. Этапы построения модели данных

Построение модели данных по программе можно разбить на два этапа:

1. Создание модели данных по программе.
2. Модификация программы к виду, использующему модель данных.

На первом этапе в модель данных включаются все переменные, используемые в программе. Для каждой переменной модели указывается ее имя и тип данных. Отметим, что имена переменных не могут совпадать.

На втором этапе в исходную программу включается определение модели данных, удаляются объявления переменных, а все обращения к переменным заменяются на обращения к модели данных.

В разделе 2.3.2 было отмечено, что при построении модели памяти по итеративной программе можно использовать статическое распределение памяти, а для рекурсивной программы требуется применение более сложного смешанного распределения памяти. Поэтому в начале построим метод для более простого первого случая (раздел 3.2), а затем на его основе построим метод для решения задачи в целом (раздел 3.3).

3.1.2. Требования к исходной программе

Сформулируем требования, предъявляемые к исходной программе.

1. Программа должна быть написана на императивном языке программирования и быть структурированной.
2. Параметры и возвращаемые значения процедур должны передаваться по значению (это не исключает передачу ссылок или указателей).
3. В идентификаторах не должен использоваться символ подчеркивания («_»).
4. Используемые выражения не должны иметь побочных эффектов — значения переменных должны изменяться только оператором присваивания.

5. Все процедуры должны иметь различные имена.

Алгоритмы дискретной математики традиционно формулируются на императивных (псевдо-)языках. Таким образом, первое требование не вносит существенных ограничений.

Второе требование 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 к виду, не использующему данную особенность, путем использования указателей.

Третье требование наложено для того, чтобы символ подчеркивания мог быть использован для образования составных идентификаторов. Вместо подчеркивания возможно использование любого другого символа, который может быть частью идентификатора.

Четвертое требование исключает применение цепных операторов присваивания, а также операций 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.

В последующих разделах рассматриваются программы, удовлетворяющие приведенным требованиям.

3.2. Построение модели данных по итеративной программе

В соответствии с синтаксисом большинства процедурных языков программирования все переменные, используемые в реализации алгоритмов, 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 локальные переменные объявлены в разных процедурах, то они могут иметь одинаковые имена. Такие переменные не могут быть вынесены в модель данных с сохранением имен. 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. Создание модели данных

В итеративной программе для каждой локальной переменной, формального параметра процедуры и возвращаемого значения можно выделить статический участок памяти [56]. Фактически это и выполняется при создании модели данных.

Для итеративных программ создание модели данных можно разбить на четыре шага, на каждом из которых в модель добавляются переменные. В начале она полагается пустой.

На первом шаге в модель выносятся глобальные переменные. Так как имена всех глобальных переменных различны, то их можно непосредственно вынести в модель. Этот шаг может быть формализован 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 выполняется выделение локальных переменных в модель. Так как их имена могут совпадать, то при добавлении в модель данных им требуется сопоставить новые имена. При построении модели будем использовать 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 формализован следующим образом.

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

   <имя процедуры>_<имя переменной>

   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. Это 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.

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 по имени переменной модели однозначно восстанавливать, какой переменной из какой процедуры она порождена. Правила такого восстановления изложены в разделе 3.3.4.

Отметим, что для уникальности имен, построенных по приведенным правилам, требуется чтобы:

- все процедуры имели различные имена;
- локальные переменные процедур в рамках одной процедуры имели разные имена;

- аргументы каждой процедуры имели разные имена;
- имена процедур и переменных не содержали символа подчеркивания.

Первые три пункта обычно связаны с синтаксисом языка. Четвертый пункт соответствует третьему требованию к исходной программе (раздел 3.1.2).

3.2.2. Модификация программы

Модифицируем программу, так чтобы она использовала только переменные модели данных. В дальнейшем предполагается, что экземпляр модели доступен всем процедурам, как переменная с именем `data`.

В начале выполним действия, связанные с глобальными переменными.

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.`».

Теперь модифицированная программа содержит только одну глобальную переменную `data` и все обращения к глобальным переменным заменены на обращения к модели данных.

Перейдем к выполнению действий с локальными переменными:

4. Если локальная переменная инициализируется одновременно с ее объявлением, то разобьем каждое объявление на две части: собственно объявление и инициализацию.
5. Удалим все объявления локальных переменных.
6. Ко всем обращениям к локальным переменным добавим префикс

   `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 ту же процедуру более одного раза, разбиваются так, чтобы 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*>`_`

Для записи преобразований процедур будем использовать следующие обозначения: $f(a_1, a_2, \ldots, a_N)$ — вызов процедуры с именем f и N реальными аргументами $a_1, a_2, \ldots, a_N$. Имена соответствующих формальных параметров обозначим через $p_1, p_2, \ldots, p_N$.

9. Каждый вызов процедуры заменяется на N операторов вида:

   `data.`<f>`_`<p_M> `=` <a_M>`;` (M=1..N)

   и оператор вызова 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 двенадцати шагов, но и после выполнения первых трех или первых шести из них. Это может быть использовано для проверки правильности производимых преобразований при выполнении их в ручную.

3.2.3. Упрощенная запись (@-нотация)

После модификации программы в ней во многих местах имеются ссылки на переменную, содержащую модель данных, что затрудняет чтение программы. Для повышения удобства записи и чтения программ введем *@-нотацию* обращения к модели данных.

Обращение к переменной будем записывать в виде:

`@<`*Имя Переменной*`>`

Например, `@a` означает обращение к переменной `a`.

Если переменная `a` является глобальной, то запись `@a` эквивалентна

```
data.a
```

Если же `a` — локальная переменная, то `@a` означает ссылку на соответствующую локальную переменную, вынесенную в модель данных. Например, если в процедуре `calcSum` была определена переменная `a`, то в ней запись `@a` будет эквивалентна записи

```
data.calcSum_a
```

Таким образом, применение @-нотации позволяет приблизить запись программ с вынесенной моделью данных к исходной записи.

В дальнейшем, во всех примерах для обращения к переменным модели применяется @-нотация.

3.2.4. Пример построения модели данных

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];
  }
}
```

```
/** Проверяет 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);
}
```

Создадим 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
}
```

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

```
Model data; // 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];
  }
}
```

При этом процедура `isMin` осталась без изменений.

Выполнение шагов 4–7 модифицирует 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);
}
```

После выполнения восьмого и девятого шагов получим:

```
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);
}
```

Таким образом, по исходной программе построена модель данных и программа преобразована так, чтобы использовать выделенную модель.

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 данных ей соответствует только одна переменная. Таким

образом, возникает проблема, связанная с тем, что при рекурсивном вызове по ошибке могут быть изменены локальные переменные не только текущего, но других экземпляров рекурсивной процедуры. Для учета этой особенности необходимо изменить правила, изложенные в разделе 3.2.

3.3.1. Построение модели данных

Если для программы с явной рекурсией применить правила, изложенные в разделе 3.2, то при вызове процедуры может получиться так, что одна и та же переменная будет требоваться и для вычисления значения аргумента и для записи полученного значения. Поэтому в модель требуется ввести дополнительные переменные, предназначенные для временного хранения аргументов процедуры. Для этого к четырем шагам, приведенным в разделе 3.2.1, добавим пятый шаг:

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.

Отметим, что для создания модели данных в соответствии с изложенными правилами достаточно произвести анализ таблицы символов исходной программы. При этом все шаги однозначны и могут быть произведены программно.

3.3.2. Модификация программы

Для хранения локальных переменных и реальных аргументов процедур требуется не только модель, но и дополнительная память. При вызове процедуры будем помещать новые значения аргументов в соответствующие переменные модели памяти. Замещаемые данные будем сохранять в стеке для

последующего восстановления, которое выполняется при выходе из процедуры. При этом сохраненные значения извлекаются из стека и записываются в переменные модели данных.

Для записи этого механизма введем оператор *обратимого присваивания* (в программах он будет обозначаться «@=»), который не только изменяет значение своего левого аргумента, но и сохраняет замещенное значение в стеке. Также введем оператор извлечения значений, сохраненных оператором *обратимого 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 девятый шаг заменяется в зависимости от типа рекурсивного вызова.

Если вызов не является явной рекурсией, то девятый шаг выглядит следующим образом:

9. Каждый вызов процедуры заменяется на:
   - N операторов присваивания вида:
     `data.`<f>`_`<p_M>`@ =` <a_M>`;` (M=1..N)
   - Вызов процедуры без аргументов.
   - N операторов восстановления значений, сохраненных операторами обратимого присваивания:
     `data.`<f>`_`<p_M> `?=;` (M=N..1)

Если же вызов является явной рекурсией, то девятый шаг имеет вид:

9. Каждый вызов процедуры заменяется на:
   - N операторов присваивания вида:
     `data.`<f>`_`<p_M>`_ =` <a_M>`;` (M=1..N)
   - N операторов обратимого присваивания, копирующие значения реальных аргументов:
     `data.`<f>`_`<p_M> `@= data.`<f>`_`<p_M>`_;` (M=1..N)
   - Вызов процедуры без аргументов.

- N операторов восстановления значений, сохраненных операторами обратимого присваивания:

  `data.`<f>`_`<p_M>` ?=;` (M=N..1)

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

Отметим, что для модификации программы в соответствии с изложенными правилами достаточно произвести анализ дерева разбора исходной программы. При этом все шаги могут быть произведены программно.

3.3.3. Пример выделения модели и модификации программы

Рассмотрим изложенный подход 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:

```
/** 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) {
    return 1;
  } else {
    return a * fact(a – 1);
  }
}
```

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 процедуры `calc` не является прямой рекурсией, а вызов процедуры `fact` из себя самой ей является. Поэтому, 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

процедуру, в которой она была объявлена. Соответствующие правила приведены в таблице 4.

Таблица 4. 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. Преобразование программы к приведенной форме

Для записи описания визуализатора программа должна находиться в приведенной форме (раздел 2.5). Поэтому требуется разработать метод, позволяющий преобразовывать программу к приведенной форме путем сокращения количества типов используемых операторов.

Отметим, что похожие задачи уже решались, например:

1. При создании структурного программирования была решена задача структуризации произвольных схем алгоритмов (блок-схем) [9].
2. В процессе компиляции для процессоров с RISC-архитектурой [45] доступно ограниченное количество типов команд.

При этом в рассматриваемом случае на входе задана уже структурированная программа, что позволяет сократить число типов используемых операторов проще, чем в первом из указанных задач. С другой стороны, в результате преобразования должна получиться структурированная программа. Поэтому методы, разработанные для второй задачи, также не подходят.

3.4.1. Типы операторов

Рассмотрим типы операторов, определенных в императивных языках (в скобках приведено обозначение в нотации типичной для языков *Java*, *C* и *C++*):

1. Выражение (`expression`);
2. Составной оператор (`{ ... }`);
3. Укороченный оператор ветвления (`if-then`);
4. Полный оператор ветвления (`if-then-else`);
5. Цикл с предусловием (`while`);
6. Оператор вызова процедуры;
7. Пустой (`;`);
8. Цикл с постусловием (`do`);
9. Цикл со счетчиком (`for`);
10. Оператор продолжения цикла (`continue`);
11. Оператор выхода из цикла (`break`);
12. Оператор возврата из процедуры (`return`);
13. Оператор выбора (`switch`);

При этом требуется преобразовать программу так, чтобы она использовала только операторы первых шести типов.

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

Исключение других операторов более сложно и рассматривается в разделах 3.4.2–3.4.7. В заключительном разделе 3.4.8 описывается порядок, в котором необходимо исключать операторы из программы.

3.4.2. 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 дополнительную переменную типа `boolean`, инициализировав ее значением `true`. После этого 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.

3.4.3. Оператор цикла со счетчиком

В языках *Java*, *C* и *C++* Оператор цикла со счетчиком описывается следующей грамматикой:

ЦиклСоСчетчиком ::= `for (` *Инициализация*? `;` *Условие*? `;` *Шаг*? `)` *Оператор*

Инициализация ::= *СписокВыражений*

| *ОпределениеПеременной*

Шаг ::= *СписокВыражений*

СписокВыражений ::= *Выражение*

| *СписокВыражений* `,` *Выражение*

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

```
Инициализация
while ( Условие ) {
  Оператор
  Шаг
}
```

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

3.4.4. Оператор продолжения цикла

Рассмотрим цикл, в котором используется оператор продолжения цикла. Для исключения оператора продолжения цикла потребуется флаг продолжения — дополнительная переменная логического типа (`boolean`), объявленная до заголовка цикла и инициализированная значением ложь (`false`). Условие выхода при истинности этого флага добавляется в заголовок цикла. Будем преобразовывать оператор снизу вверх (от самого вложенного блока к самому внешнему).

В общем виде такой цикл может быть записан следующим образом:

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

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

Оператор продолжения цикла заменим на присваивание флагу выхода значения истина (`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 опустить.

Преобразование продолжается до тех пор, пока не будет достигнут оператор цикла, на который действует оператор продолжения.

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 дополнительную переменную `continueFlag`. При этом строка

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

заменяется на строку:

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

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

```
boolean continueFlag; // Значение false по умолчанию
while ((true) && !continueFlag) {
  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.

3.4.5. Оператор выхода из цикла

Оператор выхода из цикла преобразуется аналогично оператору продолжения цикла, только в место флага продолжения заводится флаг выхода. После этого к заголовку цикла добавляется условие вида:

`!` *ФлагВыхода*

Это условие ответственно за выход из цикла после выполнения оператора выхода из цикла.

Например, фрагмент из предыдущего раздела преобразуется к следующему виду:

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

Здесь `breakFlag` — флаг выхода.

3.4.6. Оператор возврата из процедуры

Если процедура не имеет возвращаемого значения, 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.

3.4.7. Оператор выбора

В соответствии со спецификацией языков *Java* [44], *C* и *C++* оператор выбора может иметь очень сложную семантику. 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.

Так как по семантике языка программирования значение выражения нельзя вычислять несколько раз, то 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
}
```

Здесь *КонстантноеВыражения* — выражения из соответствующих меток.

Например, оператор выбора

```
boolean value;
switch (key) {
  case 'Y': case 'y':
    value = true;
    break;
  case 'N': case 'n':
    value = false;
    break;
  default:
    // Действия при неправильном вводе
}
```

преобразуется в программу вида:

```
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
}
```

3.4.8. Порядок преобразования операторов

Не все из приведенных преобразований операторов независимы. Некоторые преобразования необходимо выполнять после завершения других. Такие зависимости изображены на рис. 4.

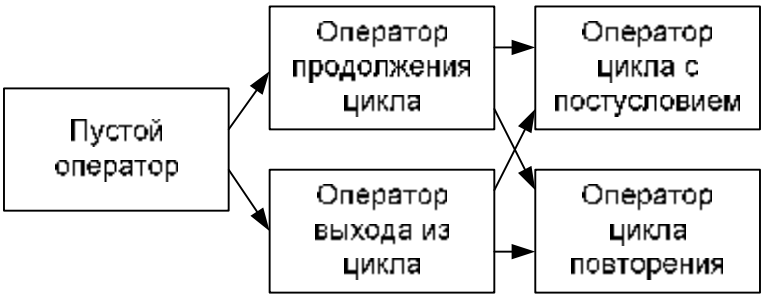


Рис. 4. Зависимости преобразований

Для модификации программы в соответствии с изложенными правилами достаточно произвести анализ дерева разбора исходной программы. При этом все шаги могут быть произведены программно.

Выводы по главе 3

1. Разработан формальный метод, позволяющий автоматизировать построение модели данных по итеративной программе.
2. Разработан формальный метод, позволяющий автоматизировать построения модели данных по рекурсивной программе.
3. Разработан формальный метод, позволяющий автоматизировать преобразование программ к приведенной форме.

ГЛАВА 4. ПРЕОБРАЗОВАНИЕ ПРОГРАММЫ В СИСТЕМУ ВЗАИМОДЕЙСТВУЮЩИХ КОНЕЧНЫХ АВТОМАТОВ

Задача формального построения конечных автоматов по программам на императивных процедурных языках программирования не является новой. В работе [30] был предложен метод преобразования (не содержащих рекурсии) программ. В дальнейшем это метод был развит, что позволило преобразовывать рекурсивные программы [26]. В области аппаратного обеспечения эта задача рассматривалась уже в 70-х годах и ее решение для одной процедуры приведено в работе [4].

Отметим, что во всех указанных работах строился один автомат, который позволял осуществлять трассировку только в прямом направлении, кроме того, рассматривались только программы, состоящие из одной процедуры. Этого недостаточно для применения данных методов в визуализаторах алгоритмов.

Задача преобразования программы в систему взаимодействующих конечных автоматов, поддерживающих обратимое исполнение достаточно сложна, поэтому будем решать ее в несколько этапов. В разделе 3.4 был разработан метод преобразования программ к приведенной форме (раздел 2.5), поэтому в дальнейшем рассматриваются только приведенные программы.

В разделе 4.1 вводятся основные понятия, рассматриваемые в настоящей главе. В частности, фиксируется грамматика, описывающая программы.

В разделе 4.2 разрабатывается метод, позволяющий по процедуре, записанной в приведенной форме, построить автомат, обеспечивающий ее трассировку в прямом направлении. На основе этого метода в разделе 4.3 разрабатывается метод, позволяющий по процедуре в приведенной форме построить пару автоматов, обеспечивающих двунаправленную трассировку.

В разделе 4.4 предлагаются методы, позволяющие объединить автоматы, построенные по отдельным процедурам в систему взаимодействующих автоматов.

В разделе 4.5 доказываются корректность предложенных методов, а также свойства автоматов, получаемых при их использовании.

4.1. Основные понятия

Для формального описания метода преобразования программы в систему взаимодействующих конечных автоматов требуется описать исходную программу (раздел 4.1.1) и ввести понятие *фрагмент автомата*, а также операции *замыкания* фрагментов автоматов (4.1.2).

4.1.1. Исходная программа

Программы в приведенной форме содержат только следующие операторы (раздел 2.5):

- оператор присваивания;
- последовательность операторов;
- полный и укороченный операторы ветвления;
- цикл с предусловием;
- вызов процедуры.

Структура программы может быть описана 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* ::= *ИмяПроцедуры*( )

При этом каждую процедуру можно рассматривать как дерево, листами которого являются операторы присваивания и вызова процедуры, внутренними узлами — управляющие операторы, а корнем — процедура в целом.

Построение автомата по процедуре будем осуществлять постепенно — от листьев дерева к корню. Для такого построения требуется ввести новые термины.

4.1.2. Фрагменты автоматов

Фрагмент автомата — набор состояний и переходов. При этом у некоторых переходов может быть не определено начальное или конечное состояние. Такие переходы называются *входами* и *выходами* фрагмента соответственно. Отметим, что переход, у которого не определены ни начальное, ни конечное состояния, одновременно является входом и выходом.

Входы и выходы фрагмента позволяют соединять его с другими фрагментами посредством операции *замыкания* — вход одного фрагмента и выход другого объединяются в один переход, у которого определены и начальное (начальное состояние выхода второго фрагмента) и конечное состояния (конечное состояние входа первого фрагмента).

Отметим, что для преобразования фрагмента в автомат требуется замкнуть все входы и выходы фрагмента и указать начальное состояние.

В дальнейшем будем рассматривать фрагменты автоматов с одним входом и одним выходом.

На рисунках входы и выходы фрагмента автомата будем представлять в виде входящих и исходящих стрелок, у которых начало (для входа) и конец (для выхода) не связаны ни с одним состоянием.

Для преобразования процедуры в конечный автомат будем строить для каждого оператора фрагмент автомата. Из фрагментов автомата,

соответствующих отдельным операторам, будем строить фрагменты, соответствующие последовательностям операторов и т.д.

4.2. Преобразование процедуры в автомат

Рассмотрим правила преобразования различных операторов.

4.2.1. Оператор присваивания

Оператор присваивания преобразуем во фрагмент автомата, содержащий одно состояние, в котором в качестве действия, выполняется присваивание. Например, оператор

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

преобразуется в состояние, как изображено на рис. 5.

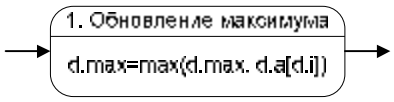


Рис. 5. Пример оператора присваивания

Здесь и далее над чертой, разделяющей на две части изображения состояния, записаны номер состояния и его название, а под ней — действия, выполняемые в состоянии.

В общем случае продукция

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

порождает фрагмент автомата, изображенный на рис. 6.

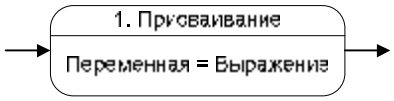


Рис. 6. Оператор присваивания

4.2.2. Последовательность операторов

Преобразование последовательности из двух операторов выполняется замыканием выхода фрагмента автомата, соответствующего первому оператору, и входа фрагмента автомата, соответствующего второму оператору. Преобразование последовательности из большего числа операторов выполняется аналогично.

Например, последовательность операторов

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

преобразовывается во фрагмент автомата, изображенный на рис. 7.

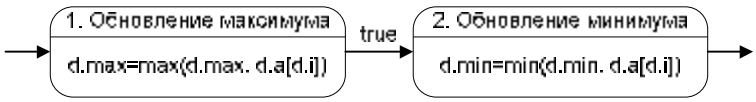


Рис. 7. Пример последовательности состояний

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 фрагмент автомата содержит один вход и один выход, то будем обозначать его, как показано на рис. 8.

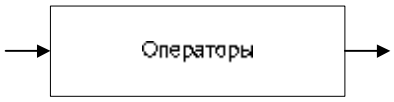


Рис. 8. Фрагмент автомата

В общем случае продукция

5 *Операторы* ::=

порождает фрагмент автомата, состоящий из одного перехода, являющегося одновременно и входом, и выходом, а продукция

4 *Операторы* ::= *Операторы Оператор*

порождает фрагмент, приведенный на рис. 9, где «Операторы» и «Оператор» — фрагменты, соответствующие нетерминалам правой части продукции.

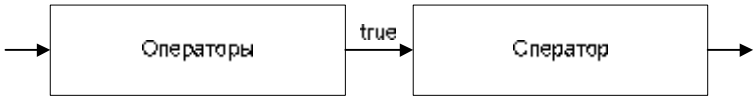


Рис. 9. Последовательность состояний

4.2.3. Оператор вызова процедуры

Оператор вызова процедуры преобразуем в состояние, в котором выполняется вызов автомата, соответствующего вызываемой процедуре. Например, вызов процедуры `findMaximum` преобразуем, как показано на рис. 10.

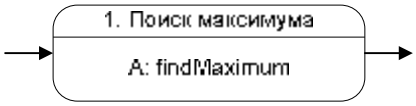


Рис. 10. Пример вызова процедуры

Здесь префикс «A:» обозначает вызов автомата. Более подробно вызовы процедур и автоматов будут рассмотрены в разделе 4.4.

В общем случае, продукция

13 *ВызовПроцедуры* ::= *ИмяПроцедуры()*

порождает фрагмент автомата, изображенный на рис. 11.

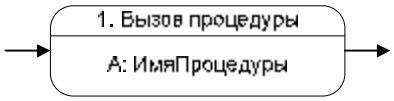


Рис. 11. Оператор вызова процедуры

4.2.4. Оператор ветвления

В продукции оператора ветвления

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

нетерминал *Выражение* задает условие ветвления. Будем преобразовывать ветвления во фрагмент автомата, изображенный на рис. 12, где фрагменты «Операторы 1» и «Операторы 2» соответствуют, последовательностям операторов, образующих ветви.

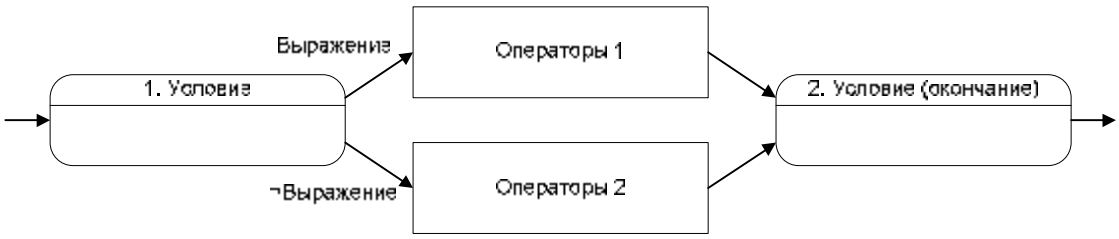


Рис. 12. Оператор ветвления

Из состояния «Условие» выполняется тот переход, условие для которого истинно. ¬*Выражение* обозначает отрицание условия, заданного *Выражением*.

Состояние «Условие (окончание)» введено для того, чтобы фрагмент автомата имел только один выход.

4.2.5. Цикл с предусловием

Будем преобразовывать циклы с предусловием, заданные продукцией

12 *ОператорЦикла* ::= *Выражение Операторы*

во фрагмент автомата, изображенный на рис. 13, где фрагмент «Операторы» соответствует последовательностям операторов, образующих тело цикла.

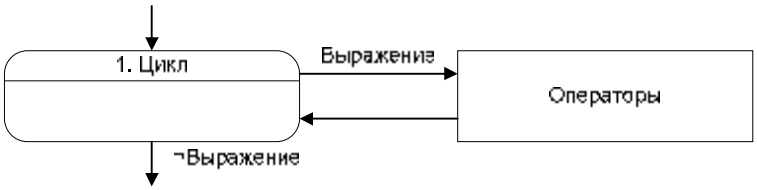


Рис. 13. Цикл с предусловием

4.2.6. Завершение построения автомата

Для завершения построения автомата к фрагменту, соответствующему телу процедуры, добавим начальное и конечное состояния. Начальное состояние связывается со входом фрагмента, а конечное — с его выходом, как показано на рис. 14.

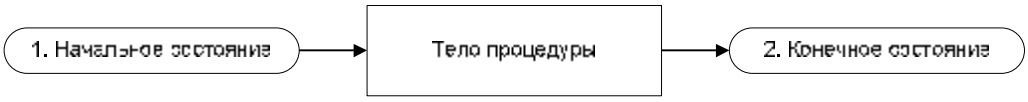


Рис. 14. Завершение построения автомата

Отметим, что 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».

4.2.7. 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.

Проиллюстрируем изложенный метод на примере построения автомата для процедуры поиска максимума в массиве натуральных чисел.

Процедура поиска максимума может быть реализована следующим образом:

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

Создадим модель данных по программе (раздел 3.3.1):

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

Модифицируем программу, так чтобы она использовала модель данных (раздел 3.3.2).

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

Заменим цикл `for` на цикл `while` (раздел 3.4.3):

```
@max = 0;
@i = 0;
while (@i < @a.length) {
  if (@max < @a[@i]) @max = @a[@i];
  @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. 15.

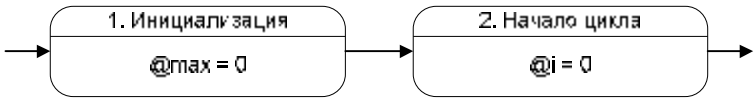


Рис. 15. Начальная инициализация

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. 16.

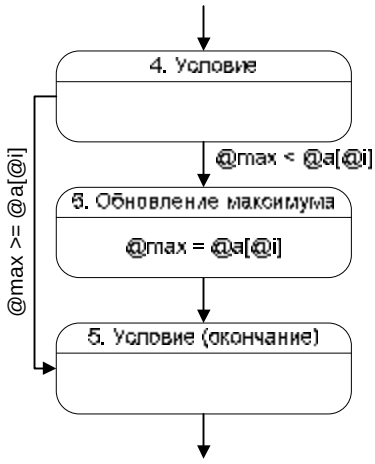


Рис. 16. Оператор ветвления

Присоединив к полученному фрагменту (состояния 4–6 на рис. 17) состояние 7, соответствующее оператору `@i++`, получим преобразованное тело цикла (состояния 4–7). Заканчивая преобразование цикла `while`, добавим состояние 3.

Присоединяя к фрагменту, соответствующему циклу `while` (рис. 13), состояния 1 и 2, а также, вводя начальное и конечное состояния (0 и 8), получим автомат, изображенный на рис. 17.

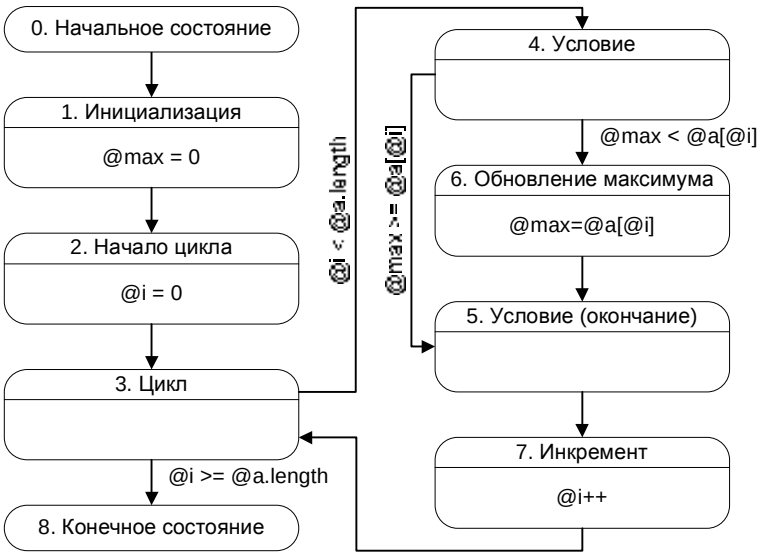


Рис. 17. Автомат поиска максимума

4.3. Построение обратного автомата

В разделе 4.2 был разработан метод, позволяющий построить по процедуре автомат, обеспечивающий трассировку программы в прямом направлении. Модифицируем этот метод так чтобы в обеспечивалась трассировка, как в прямом, так и в обратном направлениях.

4.3.1. Обратные автоматы

Для обеспечения обратимого исполнения будем строить *пары автоматов*, состоящие из прямого и обратного автоматов. *Прямой автомат* будет обеспечивать трассировку программы в прямом направлении, а *обратный автомат* — трассировку в обратном направлении.

Прямой и обратный автоматы будут иметь общее множество состояний, отличаясь только переходами. При этом для трассировки вперед будет применяться граф переходов прямого автомата, а для трассировки назад — граф переходов обратного автомата. Таким образом, пара автоматов может быть рассмотрена как один автомат с двумя функциями переходов.

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

При прямом проходе остановка автомата для визуализации выполняется перед каждым переходом (сразу после выполнения действия в состоянии). Соответственно, при обратном проходе остановку необходимо осуществлять непосредственно перед каждым состоянием. При использовании этого соглашения текущему состоянию прямого и обратного автоматов можно присваивать один номер состояния.

4.3.2. Обращение операторов

Для построения обратного автомата требуется обращать действия, выполняемые алгоритмом. Это можно сделать двумя способами:

1. *Непосредственный способ.* Состояние автомата и значения переменных модели данных вычисляются непосредственно по их значениям на следующем шаге.

2. *Косвенный способ.* Состояние автомата и значения переменных модели данных восстанавливаются по информации, сохраненной при прямом проходе, как описано ниже.

Обращение непосредственным способом позволяет экономить память, однако, не всегда возможно и для своего применения требует неформального подхода.

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

В дальнейшем вместо терминов «обращение непосредственным способом» и «обращение косвенным способом» будем писать «*непосредственное обращение*» и «*косвенное обращение*» соответственно.

Перейдем к построению пар автоматов для различных конструкций программы.

4.3.3. Обращение оператора присваивания

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

Косвенное обращение оператора присваивания выполняется путем помещения замещаемого значения переменной в стек при прямом проходе и извлечения его из стека при обратном проходе. Стек подходит для сохранения замещаемых значений, так как они используются в порядке, обратном 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`. Отметим, что *общий стек* и стек, введенный в разделе 3.3.2, могут совпадать.

Для стеков будем использовать следующие операции:

1. `push(`*выражение*`)` — поместить значение *выражения* на вершину стека;
2. `peek()` — прочитать значение на вершине стека;
3. `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 стека.

Приведем пример косвенного обращения оператора присваивания для оператора, осуществляющего обновление максимума (из процедуры поиска максимума в разделе 4.2.7):

```
@max = @a[@i];
```

Этот оператор обращается, как показано на рис. 18.

Рис. 18. Пример оператор присваивания (а) и его косвенного обращения (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 то же время непосредственное обращение оператора

```
@i++;
```

из процедуры поиска максимума в разделе 4.2.7 выполняется оператором

```
@i--;
```

Таким образом, при прямом проходе единица прибавляется к значению переменной `i`, а при обратном проходе — вычитается. Обращение данного оператора изображается, как показано на рис. 19.

а б

Рис. 19. Пример оператор присваивания (а) и его непосредственного обращения (б)

В целом, непосредственное обращение оператора присваивания вида

$$x = f(x, y_1, y_2, \ldots y_n)$$

может быть выполнено, если существует обратная к f функция

$$g(x, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots z_m),$$

такая что для всех x

$$g(f(x, y_1, y_2, \ldots y_n), y_1, y_2, \ldots, y_n, z_1, z_2, \ldots z_m)=x.$$

В этом случае для вычисления значения x на обратном проходе применяется функция g. Отметим, что функция g может зависеть не только от тех переменных, от которых зависит функция f, но и от любых других переменных, используемых в программе.

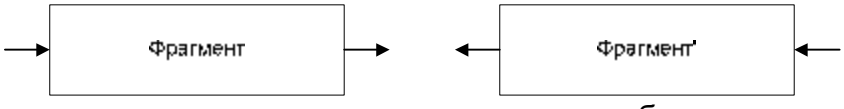
В общем случае оператор присваивания может быть обращен косвенным способом. Соответствующий фрагменты автоматов изображены на рис. 20.

а б

Рис. 20. Оператор присваивания (а) и его косвенное обращение (б)

4.3.4. Обращение последовательности операторов

Будем обозначать фрагмент обратного автомата, также как и соответствующий фрагмент прямого автомата, но с добавлением к имени штриха (рис. 21).



а б

Рис. 21. Фрагмент прямого автомата (а) и соответствующий ему фрагмент обратного автомата (б)

Последовательность операторов обращается путем выполнения обращений операторов в противоположном порядке (рис. 22).

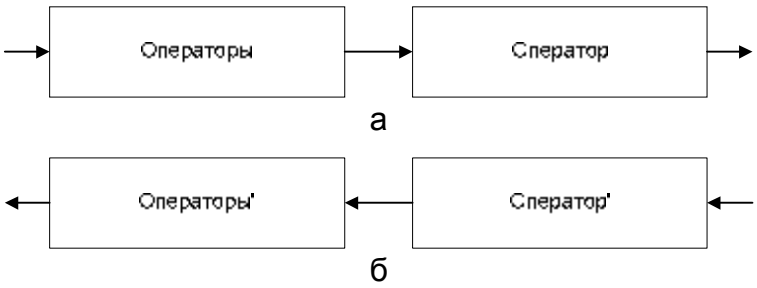


Рис. 22. Последовательность операторов (а) и ее обращение (б)

Таким образом, обращение последовательности операторов сводится к обращению отдельных операторов.

4.3.5. Обращение оператора вызова

Оператор вызова процедуры преобразуем во фрагмент автомата, состоящий из одного состояния, в котором выполняется вызов обратного автомата для соответствующей процедуры. Полученный фрагмент изображен на рис. 23, а соответствующий ему фрагмент прямого автомата — на рис. 11.

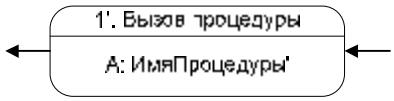


Рис. 23. Последовательность состояний

Более подробно вызовы процедур и автоматов будут рассмотрены в разделе 4.4.

4.3.6. Обращения операторов ветвления

Обращение оператора ветвления затруднено тем, что решение о выборе ветви для обратного прохода невозможно принять в состоянии «Условие» при прямом проходе, так как в этот момент выполнение обращенной ветви должно быть уже закончено. Поэтому, выбор ветви, исполняемой при обратном проходе, выполняется в состоянии «Условие (окончание)».

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 обращении оператора ветвления получаются фрагменты автоматов, приведенные на рис. 24 (соответствующий прямой автомат приведен на рис. 12).

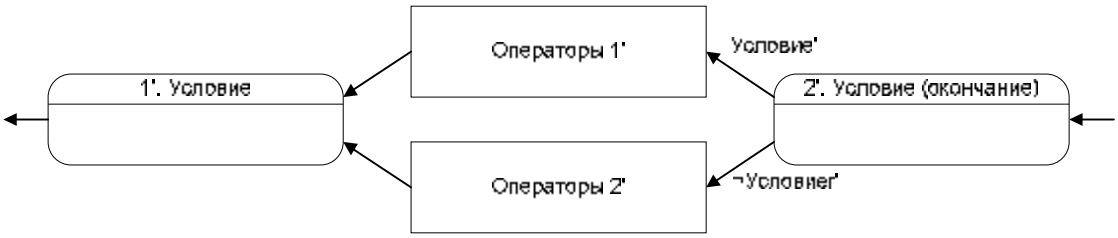


Рис. 24. Непосредственное обращение оператора ветвления

Здесь и далее под пометкой «*Условие'*» подразумевается условие, позволяющее выбрать ветвь для обратного прохода. Отметим, что данное условие строится неформально.

При косвенном способе необходимо запоминать ветвь, выбранную при прямом проходе.

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. 25.

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.

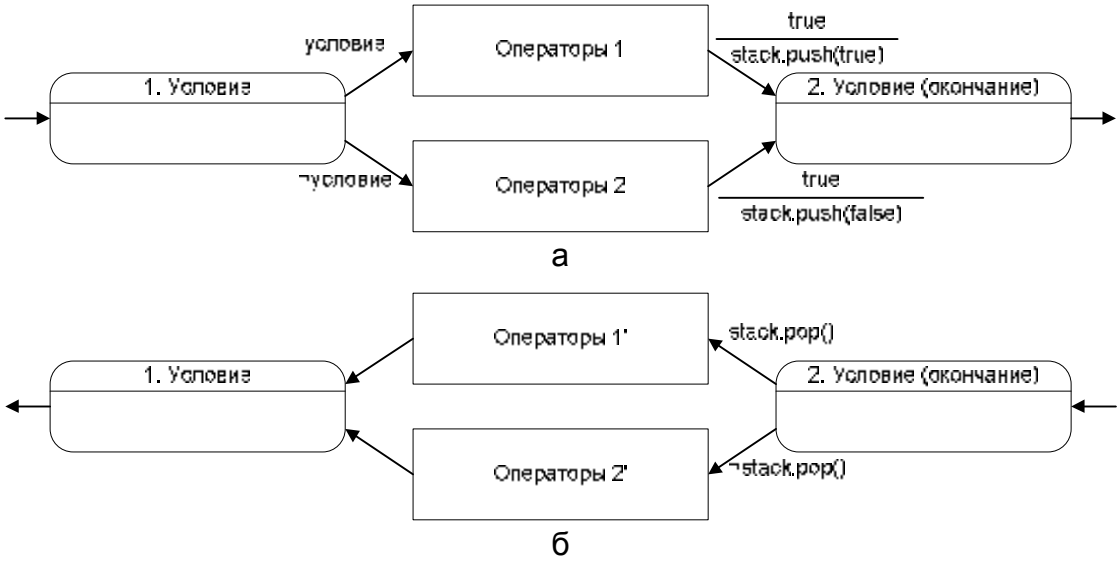


Рис. 25. Косвенное обращения оператора ветвления.
Фрагменты прямого (а) и обратного (б) автоматов

Таким образом строятся фрагменты смешанного автомата [29], в котором, в отличие от автомата Мура, действия могут выполняться не только в состояниях, но и на переходах.

Косвенное обращение не требует дополнительных данных и может выполняться автоматически.

4.3.7. 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. 26 (соответствующий фрагмент прямого автомата приведен на рис. 13).

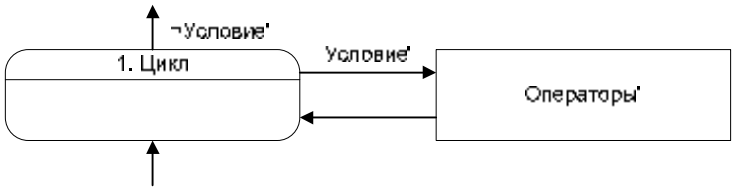


Рис. 26. Непосредственное обращение цикла с предусловием

При косвенном обращении цикла с предусловием требуется сохранять дополнительную информацию. Для этого при прямом проходе будем сохранять в стеке значение `false` (ложь), если переход в состояние цикла совершен извне,

и `true` (истина), если переход выполняется из тела цикла. Соответственно, при обратном проходе данную информацию можно будет использовать для определения того, было ли тело цикла выполнено при прямом проходе.

Фрагменты прямого и обратного автоматов для косвенного обращения цикла с предусловием приведены рис. 27.

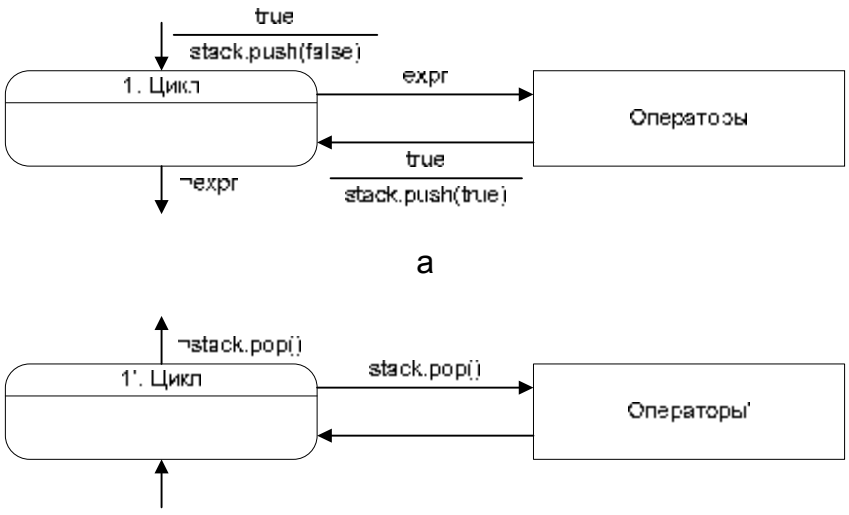


Рис. 27. Косвенное обращение цикла с предусловием.
Фрагменты прямого (а) и обратного (б) автоматов

4.3.8. Варианты построения обратного автомата

В разделах 4.3.3–4.3.7 рассмотрены различные варианты построения обратного автомата. Обобщим сведения о предложенных вариантах в таблице 5.

Таблица 5. Варианты построения прямого и обратного автоматов

Особенности обращения	Непосредственное обращение	Косвенное обращение		
		Оператор присваивания	Ветвления и циклы	Последователь-ности операторов
1	2	3	4	5
Формализм	неформальный	формальный	формальный	формальный
Тип автоматов	Мура	Мура	смешанный	Мура
Стек	не требуется	один		не требуется
Модификация прямого автомата	не требуется	требуется		

Построить пару автоматов по процедуре можно как формальным, так и неформальным способом. При формальном построении применяется косвенно обращение (столбцы 3–5). При этом требуется дополнительная память в виде стека. В случае неформального построения можно применять как прямое, так и косвенное обращение. При этом в случае прямого обращения дополнительная память не требуется.

Отметим, что прямой и обратный автоматы, построенные посредством предложенных методов, имеют одинаковую структуру переходов. При этом прямой и обратный автоматы имеют одни и те же переходы, отличающиеся направлениями, условиями и действиями, но не соединяемыми вершинами.

4.3.9. Пример построения обратного автомата

В разделе 4.2.7 был построен прямой автомат для процедуры поиска максимума. Модифицируем этот автомат и построим обратный автомат, как изложено в разделе 4.3.

Оператор, осуществляющий обновление максимума,

```
@max = @a[@i];
```

обратим косвенным способом (раздел 4.3.3 и столбец 3 таблицы 5), так как его обращение непосредственным способом требует просмотра всех предыдущих значений массива.

По тем же причинам оператор ветвления обратим косвенным способом (раздел 4.3.4 столбец 4 таблицы 5). При этом получим смешанный автомат. Обращенный оператор ветвления приведен на рис. 28.

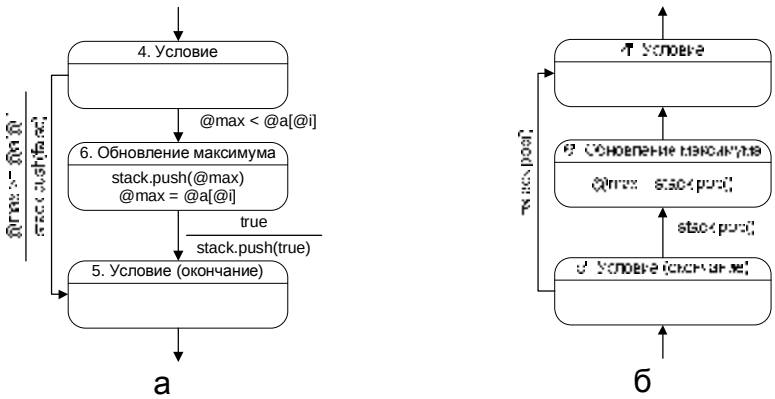


Рис. 28. Оператор ветвления (а) и его обращение (б)

Оператор, осуществляющий переход к следующему элементу, обратим непосредственным способом (раздел 4.3.3 и столбец 2 таблицы). Также непосредственным способом обратим цикл. Обращенный цикл соответствует состояниям 3–7 на рис. 29 и 30.

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

Завершим построение обратного автомата добавлением начального и конечного состояний. Полученная пара автоматов изображена на рис. 29 и 30.

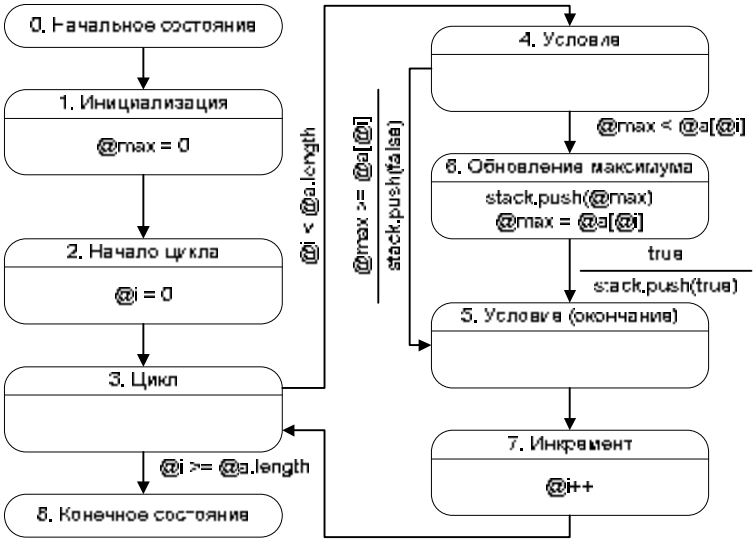


Рис. 29. Прямой автомат поиска максимума

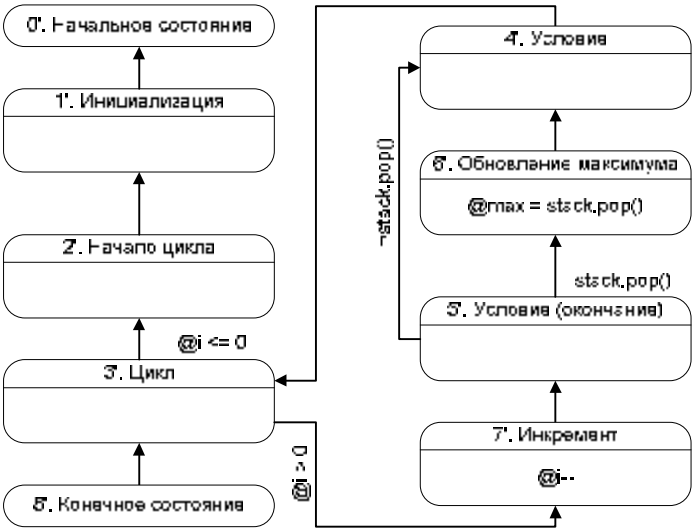


Рис. 30. Обратный автомат поиска максимума

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

Пара автоматов, построенная при использовании методов,столбцов 3 и 4 таблицы 5, приведена на рис. 31 и 32.

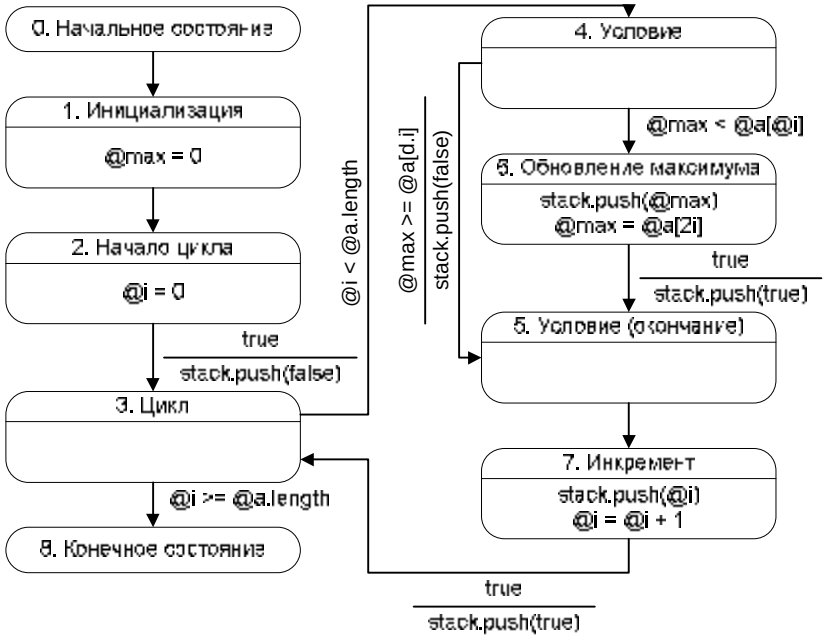


Рис. 31. Прямой автомат поиска максимума (второй вариант)

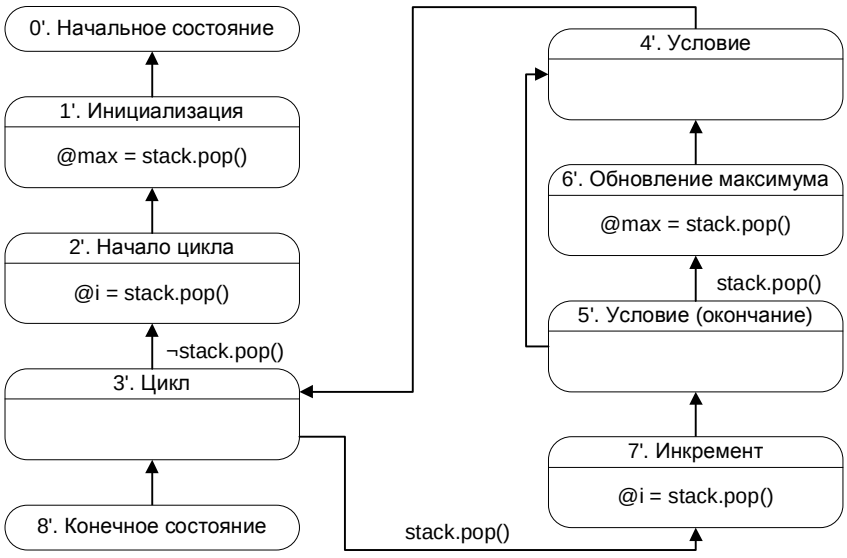


Рис. 32. Обратный автомат поиска максимума (второй вариант)

4.4. 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:

1. Итеративные программы.
2. Рекурсивные программы.

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

4.4.1. Итеративные программы

Преобразование итеративной программы

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

Введем несколько условий для связи между автоматами:

1. `state(Ai)==j` — автомат *Ai* находится в состоянии с номером *j*.
2. `isAtStart(Ai)` — автомат *Ai* находится в начальном состоянии.
3. `isAtEnd(Ai)` — автомат *Ai* находится в конечном состоянии.

Так как автоматы *Ai* и *Ai'* всегда находятся в одном и том же состоянии (раздел 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 вызывающей процедуры. Составим *список вызовов* автомата *Ai*. Он состоит из пар (*Aj*, *k*), где *Aj* — автомат, который выполняет вызов автомата *Ai*, а *k* — номер состояния, в котором выполняется 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 через *R(Ai)*.

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

1. К переходу из начального состояния добавим условие *R(Ai)*.
2. Введем безусловный переход из конечного состояния в начальное.

Для обратного автомата *Ai'*:

3. К переходу из конечного состояния добавим условие *R(Ai)*.
4. Введем безусловный переход из начального состояния в конечное.

После этого преобразуем состояния, в которых выполняется вызов автоматов. Пусть в состоянии *j* выполняется вызов автомата *Ak* (рис. 33).

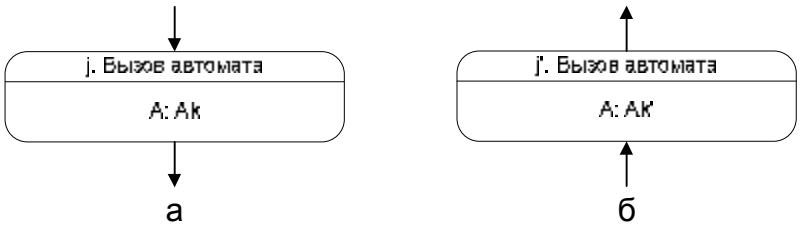


Рис. 33. Состояние прямого (а) и обратного (б) автоматов, из которых вызывается другой автомат

Преобразуем состояния прямого и обратного автоматов, как показано на рис. 34.

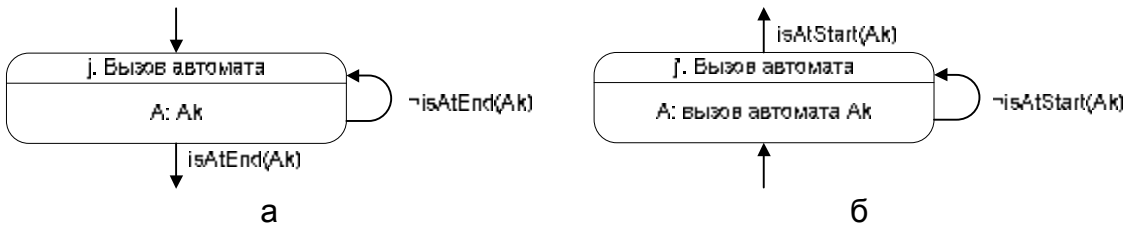


Рис. 34. Преобразованные состояния прямого (а) и обратного (б) автоматов, из которых вызывается другой автомат

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

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

Пример преобразования итеративной программы

Проиллюстрируем рассмотренные преобразования на (весьма надуманном) примере, изображенном на рис. 35.

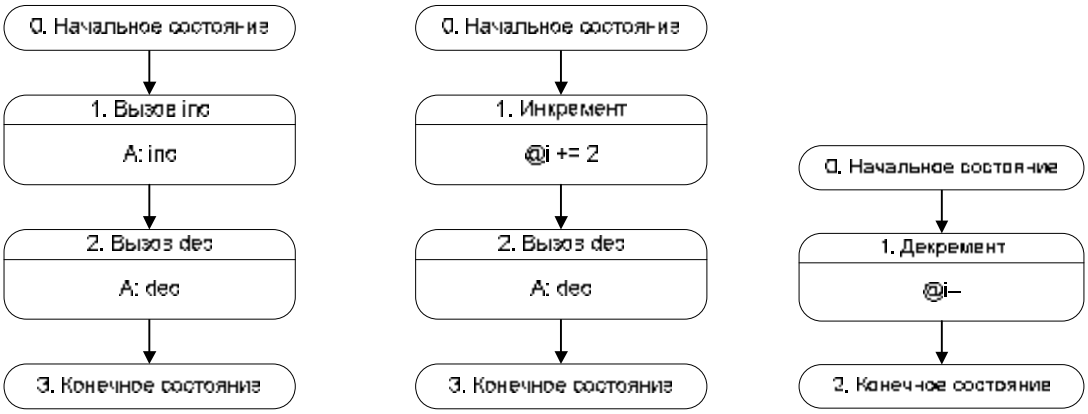
```
void main() {
  inc();
  dec();
}
```
а

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

```
void dec() {
  d.i--;
}
```
в

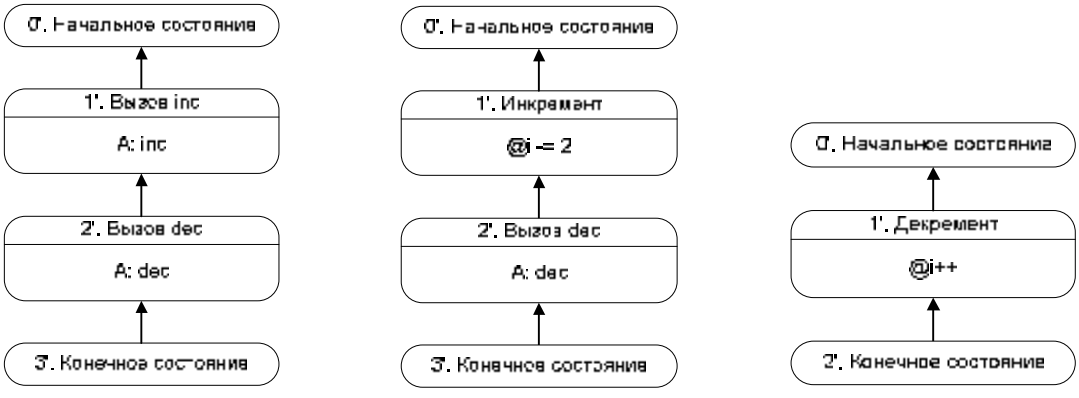
Рис. 35. Преобразуемая программа. Главная процедура (а), процедура inc (б), процедура dec (в)

Соответствующие им прямые автоматы и обратные автоматы изображены на рис. 36 и рис. 37



а б в

Рис. 36. Прямые автоматы для процедур main (а), inc (б) и dec (в)



а б в

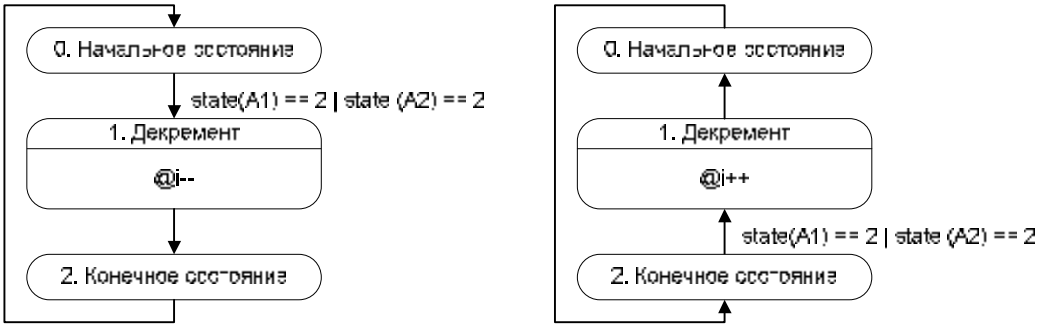
Рис. 37. Обратные автоматы для процедур main (а), inc (б) и dec (в)

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 этом условие *R(A3)* записывается следующим образом:

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

где «|» —дизъюнкция. Преобразованные автоматы *A3* и *A3'* приведены на рис. 38.



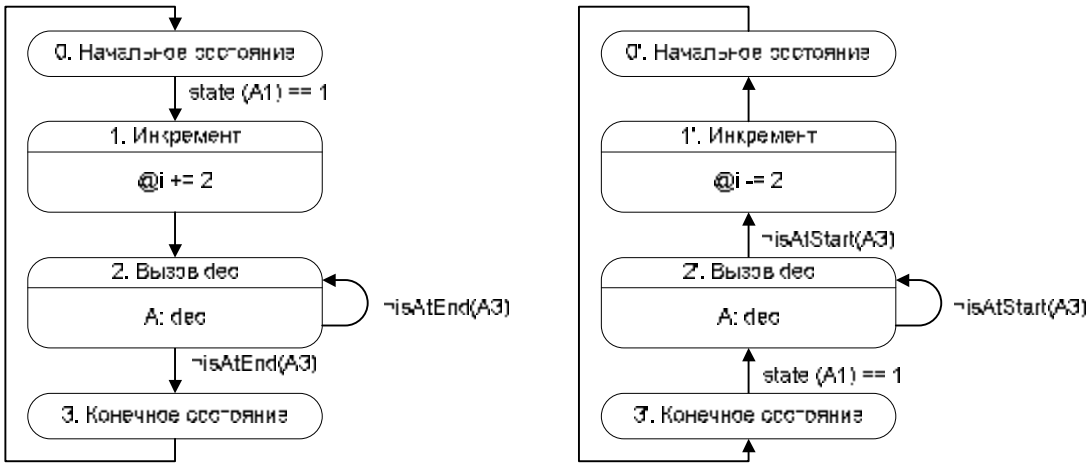
а б

Рис. 38. Преобразованные автоматы A3 (а) и A3' (б)

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

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

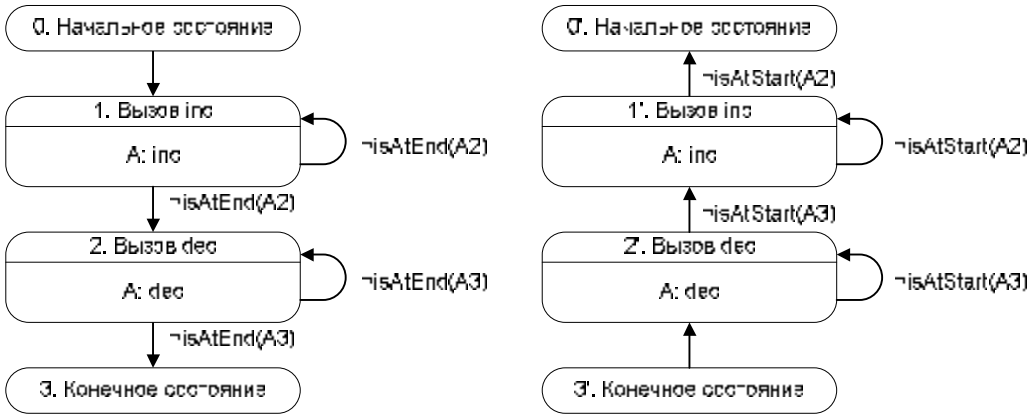
При этом в автоматах *A2* и *A2'* необходимо преобразовать состояния 2 и 2', как показано на рис. 34. Таким образом, строятся автоматы, изображенные на рис. 39.



а б

Рис. 39. Преобразованные автоматы A2 (а) и A2' (б)

Аналогично преобразуем автоматы *A1* и *A1'* (рис. 40).



а б

Рис. 40. 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) à (1, 0, 0) à (1, 1, 0) à (1, 2, 0) à
(1, 2, 1) à (1, 2, 2) à (1, 3, 0) à (2, 0, 0) à
(2, 0, 1) à (2, 0, 2) à (3, 0, 0),
```

а при обратном проходе:

```
(3, 0, 0) à (2, 3, 2) à (2, 3, 1) à (2, 3, 0) à
(1, 3, 2) à (1, 2, 2) à (1, 2, 1) à (1, 2, 0) à
(1, 1, 3) à (1, 0, 3) à (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 активными в рассматриваемый момент времени, а остальные процедуры — пассивными.

При прямом проходе автоматы, соответствующие пассивным 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.

4.4.2. Рекурсивные программы

Преобразование рекурсивной программы

Вопрос о преобразовании рекурсивных программ в автоматные рассматривался в статье [26]. Однако, этот метод применим только для преобразования программ, состоящих из одной процедуры. В данной работе предложен метод, применимый для программ, содержащих несколько процедур. Этот метод позволяет преобразовывать программы, как с явной, так и с косвенной рекурсией. Предлагаемый метод основан 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, выполняющей вычисление факториала:

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

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

Для отражения этого факта введем понятие экземпляр автомата. Как указано в разделе 4.3.1, пара из прямого и обратного автоматов может быть рассмотрена, как один автомат с двумя функциями переходов. *Экземпляр автомата* — объект, хранящий состояние такого автомата. При этом все экземпляры автомата имеют общую логику, определяемую графом переходов. Таким образом, экземпляру автомата соответствует одно число — номер его состояния.

Определим для экземпляра автомата следующие функции:

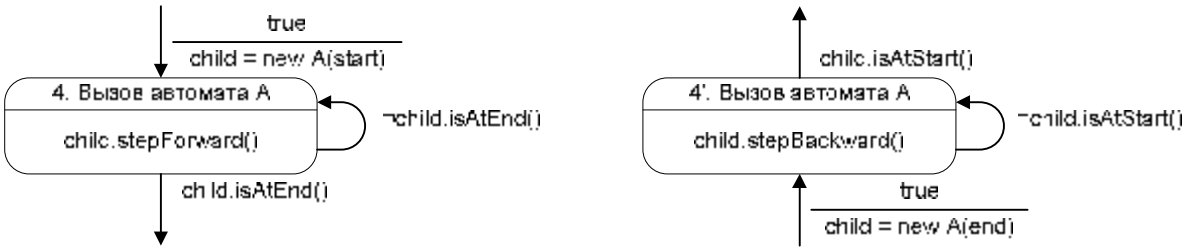
- `stepForward()` — сделать шаг вперед (изменить состояние и выполнить действия в соответствии с функцией переходов прямого автомата);
- `stepBackward()` — сделать шаг назад (изменить состояние и выполнить действия в соответствии с функцией переходов обратного автомата);
- `isAtStart()` — проверка, находится ли экземпляр автомата в начальном состоянии;
- `isAtEnd()` — проверка, находится ли экземпляр автомата в конечном состоянии;

- `new автомат(start)` — создать новый экземпляр автомата, находящийся в начальном состоянии;
- `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, выполняющих вызов автомата *A*, приведены на рис. 41.



а б

Рис. 41. Фрагменты прямого (а) и обратного (б) автоматов, вызывающие автомат A

Этими фрагментами заменяются состояния, вызывающие автоматы *A* и *A'* в автоматах, построенных с помощью предложенного метода.

Пример преобразования рекурсивной программы

Проиллюстрируем предложенный метод на примере функции, вычисляющей факториал (исходная программа приведена в начале настоящего раздела). Данная функция использует явную рекурсию.

Построим модель данных по программе и преобразуем ее (раздел 2.2.2). Передача параметра будем производиться через переменную `a`, а возврат результата — через переменную `r`:

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

Прямой и обратные автоматы, построенные по данной программе, изображены на рис. 42.

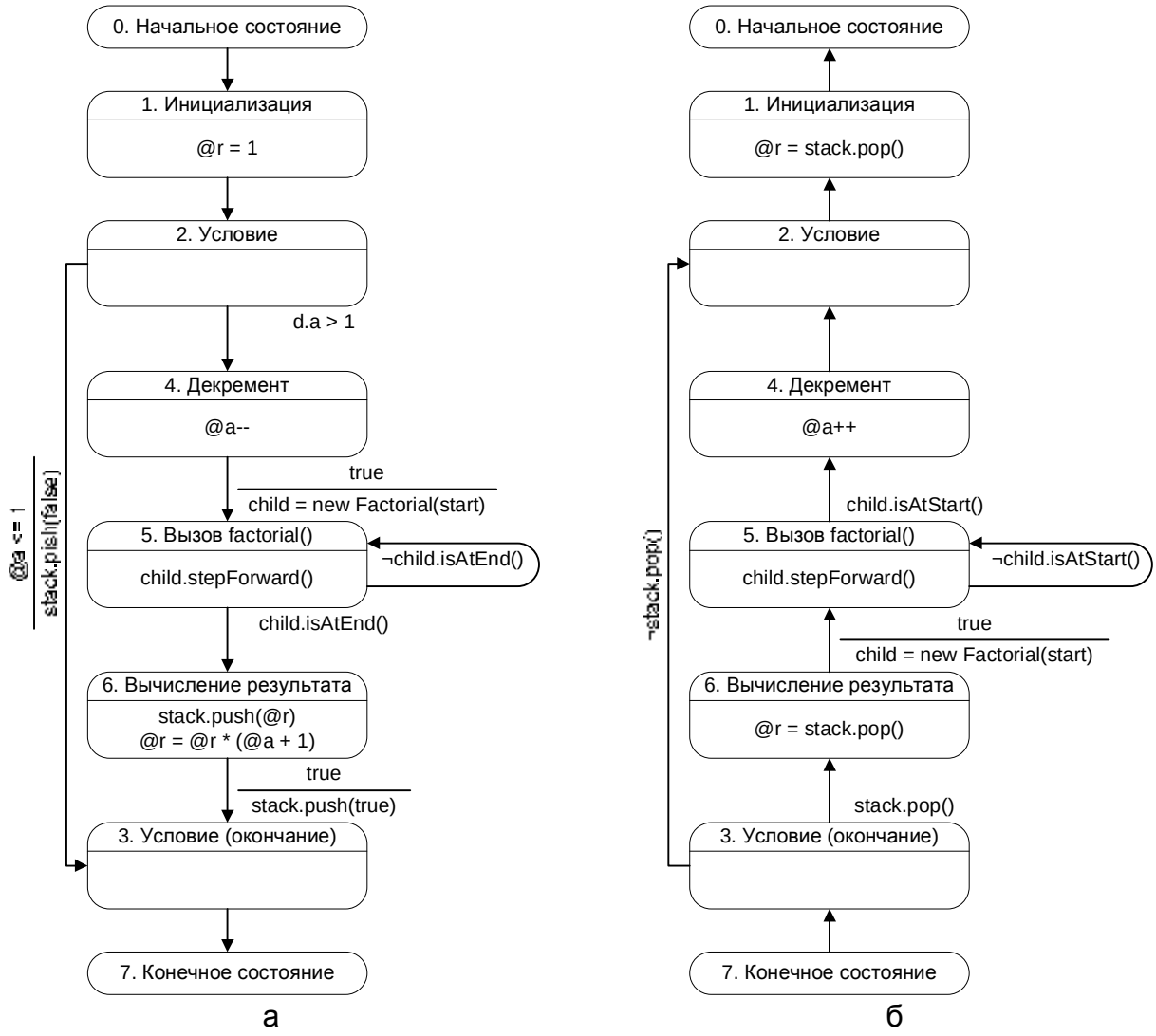


Рис. 42. Прямой (а) и обратный (б) автоматы для процедуры вычисления факториала

Преобразование программ с косвенной рекурсией производится аналогично.

4.5. Формализация преобразования программы

Опишем формально процедуру построения системы взаимодействующих автоматов по программе, основанную на рассуждениях, приведенных в разделах 4.2−4.4.

Построенный автомат может взаимодействовать со стеком посредством операций, описанных в разделе 4.3.3.

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

4.5.1. Свойства автоматов

Для каждого построенного автомата будем доказывать следующие основные свойства:

1. *Адекватность* — система автоматов выполняет те же действия, что и исходная программа.
2. *Полнота* — для каждого состояния, кроме конечного, при любых значениях переменных условие на одном из переходов должно быть истинно (дизъюнкция всех условий на переходах из состояния является тавтологией).
3. *Непротиворечивость* — условия на переходах из одного состояния не могут быть истинными одновременно (все попарные конъюнкции условий на переходах должны быть невыполнимы).
4. *Отсутствие недостижимых состояний* — любое состояние может быть достигнуто по переходам из начального состояния.

5. *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.

При проверке на отсутствие недостижимых состояний, не будем требовать для каждого состояния наличия такой комбинации переменных, чтобы можно было достигнуть этого состояния, так как в общем случае эта задача алгоритмически не разрешима. Будем проверять только принципиальную возможность достижения этой вершины из начальной. При этом автомат рассматривается как ориентированный граф, вершинами которого являются состояния, а дугами — переходы.

Указанные свойства будем доказывать с помощью структурной индукции. Будем предполагать, что эти свойства выполняются для фрагментов, полученных преобразованием потомков вершин дерева, и доказывать эти свойства для фрагмента, полученного преобразованием текущей вершины.

Сформулируем перечисленные свойства автоматов для фрагментов.

1. *Адекватность фрагмента автомата* — фрагмент выполняет такие же действия, что и оператор (последовательность операторов), из которого он получен.
2. *Полнота фрагмента автомата* — для каждого состояния дизъюнкция условий на переходах из состояния является тавтологией.
3. *Непротиворечивость фрагмента автомата* — для каждого состояния все попарные конъюнкции условий на переходах должны быть невыполнимы.
4. *Отсутствие недостижимых состояний во фрагменте* — каждое состояние и выход фрагмента могут быть достигнуты по переходам из входа фрагмента.
5. Обратимость фрагмента автомата — при 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.

4.5.2. Текстовая нотация

Для описания состояний автоматов будем использовать текстовую нотацию, определенную следующей грамматикой:

Состояние ::= *ПереходыВ* < *ИмяСостояния*: *Действия* > *ПереходыИз*

Действия ::= *Действие*; *Действия*
|

ПереходыВ ::= *ПереходВ*; *ПереходыВ*
|

ПереходВ ::= *ИмяСостояния* {/ *Действие*}

ПереходыИз ::= *ПереходИз*; *ПереходыИз*
|

ПереходИз ::= *ИмяСостояния* [*Условие*] {/ *Действие*}

Здесь *ИмяСостояния* — произвольный идентификатор, а *Действие* — произвольное действие, записываемое на языке программирования.

Автоматы будем описывать как множества состояний. При этом запись является корректной, если автомат содержит все состояния, в которые (из которых) ведут его переходы.

Для описания фрагментов автоматов введем два псевдосостояния с именами *Вход* (начальное состояние для входов фрагмента) и *Выход* (конечное состояние для выходов фрагмента).

Действие на переходе можно добавить как при определении входящего, так и при определении исходящего перехода. Следовательно, на переходе может выполняться 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 выполнять на границах переходов — когда действия, указанные на выходах уже выполнены, а на входах еще нет.

Начнем доказательство с отдельных операторов (продукции 10–13, грамматики приведенной в 4.1.1), затем перейдем к последовательностям операторов (продукции 4–9), а от них — к процедурам (продукции 1–3).

4.5.3. Преобразование оператора присваивания

Оператор присваивания преобразуется, как указано в разделах 4.2.1 и 4.3.3. 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 вершина стека содержит значение, сохраненное 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. ◄

4.5.4. Преобразование оператора ветвления

Оператор ветвления преобразуется, как указано в разделах 4.2.4 и 4.3.6.

Преобразуемая продукция:

11 *ОператорВетвления* ::= *Условие* $Операторы_1$ $Операторы_2$

Для прямого автомата добавляются два состояния:

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

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

Для обратного автомата также 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, действиям, 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 выполняется переход во фрагмент *Да'*. В противном случае после снятия значения со стека происходит переход во фрагмент *Нет'*. По индуктивному предположению, на входе во фрагмент автомата в вершине стека содержится значение, помещенное туда при прямом проходе. Таким образом, фрагмент обратного автомата выбирается верно. После этого указанное 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 выполняются переходы во фрагменты *Да* и *Нет* (*Да'* и *Нет'*), от входов которых по индуктивному предположению достижимы все определенные в них состояния, а также выходы. Следовательно, состояние 2 (1') также достижимо. Поэтому, также достижим и выход. ◄

4.5.5. Преобразование оператора цикла

Оператор цикла преобразуется как указано в разделах 4.2.5 и 4.3.7.

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 значение `false`. После этого, пока *Условие* истинно, осуществляется переход во фрагмент *Операторы*, на выходе из которого при каждой итерации цикла в вершину стека помещается значение `true`. Таким образом, по индуктивному предположению тело цикла выполняется пока истинно условие. При этом стек содержит столько значений `true`, сколько было итераций цикла и один элемент со значением `false`. ◄

Обратимость. ► При обратном проходе, пока на вершине стека лежит значение `true`, оно снимается и выполняется переход во фрагмент *Операторы'*. При этом по индуктивному предположению, при каждом попадании в добавленное состояние вершина стека будет содержать 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 из него. ◄

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. ◄

4.5.6. Преобразование оператора вызова процедуры

Оператор вызова процедуры преобразуется, как указано в разделе 4.4.2. При этом возможная нерекурсивность некоторых процедур не используется.

Преобразуемая продукция:

13 *ВызовПроцедуры* ::= *Процедура*()

Состояние, добавляемое к прямому автомату:

Вход, child = new *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 4.5.8.

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

Обратимость. ► При обратном проходе, выполняются обратные шаги экземпляра автомата, пока тот не придет в исходное состояние. По дополнительному предположению после каждого такого шага значения переменных восстанавливают свои значения, и поэтому 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. ◄

4.5.7. Преобразование последовательностей операторов

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

Продукция

5 *Операторы* ::=

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

Адекватность и Обратимость. ► Действия не выполняются. ◄

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*, а также выход *Ф2'* и вход *Ф1*.

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

Обратимость. ► По индуктивному предположению, после выхода из фрагмента *Ф2'* в переменных будут восстановлены промежуточные значения. Следовательно, и после выхода из фрагмента *Ф1'* переменные 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 также его выход. ◄

4.5.8. 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. ◄

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. ◄

4.5.9. 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 доказаны 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 индуктивному предположению.

Заметим, что, так как дерево вызовов конечно, то последовательно «обрывая листья» можно оставить только корень, для которого также 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*. ◄

В таблице 6 указаны продукции, порождающие автоматы, состояния или переходы. Из приведенных данных следует, что количество создаваемых автоматов, состояний и переходов линейно зависит от суммы количества операторов и процедур в исходной программе.

Таблица 6. Автоматы, состояния и переходы

Продукция	Добавляемые		
	Автоматы	Состояния	Переходы
1	2	2	
2	2		
3		2	
4			1
10		1	2
11		2	2
12		1	2
13		1	3

Таким образом, описанные методы позволяют автоматически строить по программе систему взаимодействующих конечных автоматов, допускающую

трассировку либо только в прямом (раздел 4.2), либо в прямом и обратном направлениях (раздел 4.3).

Выводы по главе 4

1. Разработан метод преобразования программ в систему взаимодействующих конечных автоматов, обеспечивающую трассировку в прямом направлении.
2. Разработан метод преобразования программ в систему взаимодействующих конечных автоматов, обеспечивающую трассировку в прямом и обратном направлениях.
3. Доказана корректность предложенных методов.
4. Доказаны свойства адекватности, обратимости, полноты, непротиворечивости и достижимости всех состояний для автоматов, получаемых разработанными методами.

ГЛАВА 5. ЯЗЫК ОПИСАНИЯ ВИЗУАЛИЗАТОРОВ

Для автоматизации построения визуализаторов алгоритмов требуется разработать язык, позволяющий описывать визуализаторы. Описание визуализатора имеет сложную структуру. Поэтому в качестве основы при разработке языка описания визуализаторов был выбран язык *XML* [77], предназначенный для платформонезависимого хранения и передачи структурированных данных.

5.1. Структура языка

Описание визуализатора алгоритма можно разбить на две основные части:

1. Описание визуализируемого алгоритма, позволяющее автоматизировать реализацию:
   - модели данных;
   - логики визуализатора;
   - набора комментариев;
   - связи с визуальным представлением
2. Описание конфигурации визуализатора, в частности:
   - сообщений, выдаваемых пользователю;
   - цветовой схемы визуализатора;
   - …

Первая часть описания создается один раз в процессе разработки визуализатора и после этого не изменяется. Вторая — позволяет настраивать визуализатор под конкретное окружение, например, языковое.

Структура описания визуализатора представлена на рис. 43. Здесь и далее для представления структуры XML-описания применяется диаграмма элементов, основанная на диаграмме классов *UML* [3]. При этом классы соответствуют XML-элементам, а их атрибуты — атрибутам элементов. Атрибуты, имеющие стереотип «text», соответствуют вложенному тексту

элемента. Стереотип «i18n» поясняется ниже. Абстрактные элементы применяются для отображения общности элементов. При этом конкретные XML-элементы им не соответствуют.

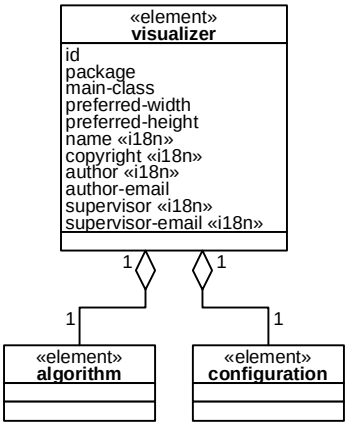


Рис. 43. Диаграмма элементов описания визуализатора

Рассмотренные части описания визуализатора могут разрабатываться независимо друг от друга.

5.2. Описание визуализируемого алгоритма

В предлагаемом языке алгоритм рассматривается как набор процедур и глобальных переменных. Среди процедур выделяется главная процедура, запускаемая при выполнении алгоритма. Эта процедура должна иметь имя `main`.

Процедура рассматривается как последовательность операторов следующих типов:

- оператор присваивания;
- оператор ветвления;
- оператор цикла с предусловием;
- оператор вызова процедуры;
- блочный оператор.

Каждому типу оператора, кроме блочного, соответствует XML-элемент. Блочные операторы записываются неявно.

Отметим, что операторы вызова процедуры позволяют задавать алгоритмы, как с явной, так и с косвенной рекурсией. Оба случая обрабатываются корректно.

Операторы могут использовать как глобальные переменные, определенные на уровне алгоритма, так и локальные переменные, определенные в рамках процедуры.

Для шага может быть указан его уровень, шаблон комментария и связь с визуальным представлением.

Уровень шага определяет в каком случае визуализатор останавливается при исполнении данного шага. Уровень задается целым числом. При этом значение `-1` обозначает пропуск шага, `0` — остановку при просмотре алгоритма маленькими шагами, `1` — остановку при просмотре алгоритма большими шагами.

Шаблон комментария определяет вид и параметры комментария, отображаемого на данном шаге.

Связь с визуальным представлением позволяет обновлять визуальное представление при отображении шага.

На рис. 44 приведена диаграмма XML-элементов, применяемых при описании логики визуализаторов.

5.2.1. Описание алгоритма

Как следует из рис. 44, описание алгоритма задается элементом `algorithm`, содержащим описания процедур (элемент `auto`), глобальных переменных (элемент `variable`), а также вспомогательных конструкций (элементы `import`, `toString` и `method`).

Описание процедур будет рассмотрено в разделе 5.2.2.

Описание глобальных переменных

Для каждой глобальной переменной указывается ее описание (атрибут `description`), имя (`name`), тип (`type`) и значение по умолчанию (`value`).

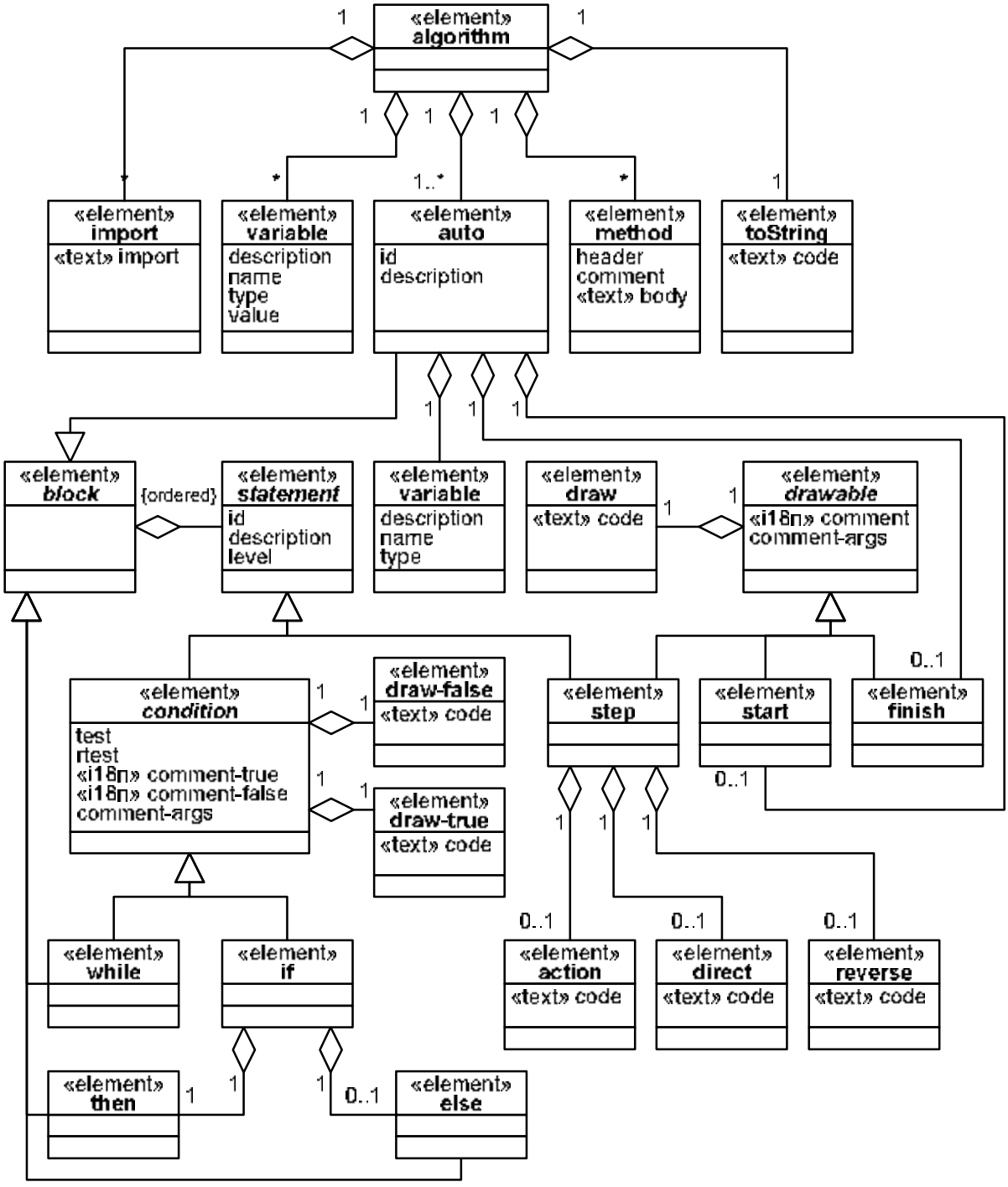


Рис. 44. Диаграмма элементов описания визуализируемого алгоритма

Описание переменной применяется при отладке алгоритма и не используется при генерации логики визуализатора.

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

Значение по умолчанию применяется при автоматизированной проверке корректности построения логики визуализатора (раздел 6.1.3).

Описание вспомогательных конструкций

Элемент `import` позволяет описывать связь визуализатора с другими типами аналогично конструкции `import` языка *Java* [44]. Имя импортируемого класса (пакета) задается обычным образом и записывается внутри элемента `import`. Автоматически импортируются пакет `ru.ifmo.vizi.base.auto` и класс `Java.util.Locale`.

Элемент `toString` задает метод, представляющий значения основных переменных визуализатора в виде строки. Этот метод используется при автоматизированной проверке корректности построения логики визуализатора (раздел 6.1.3). При этом значения, выдаваемые этим методом при прямом и обратном проходах, сравниваются, и при их неравенстве выдается сообщение об ошибке.

Элемент `method` позволяет задавать не визуализируемые методы, которые могут применяться, например, для построения визуального представления.

Для метода указывается сигнатура метода (атрибут `header`), комментарий (`comment`) и тело метода (текст элемента). Сигнатура и тело метода записываются также, как в языке *Java*.

5.2.2. Описание процедур

Описание процедуры задается элементом `auto`. При этом указываются имя (атрибут `name`) и описание (`description`). Имя процедуры должно быть корректным *Java*-идентификатором и не содержать символов подчеркивания.

В рамках процедуры описываются локальные переменные, начальное и конечное состояния, а также шаги алгоритма.

Описание локальных переменных.

Локальные переменные описываются элементом `auto`. Для каждой локальной переменной указываются ее словесное описание (атрибут `description`), имя (`name`) и тип (`type`). Имена переменных должны быть корректными *Java*-идентификаторами и не содержать символов подчеркивания.

Отметим, что для локальных переменных значения по умолчанию не указываются. Таким образом, исходное значение локальной переменной не определено.

Описание начального и конечного состояний

Описания начального (элемент `start`) и конечного (элемент `finish`) состояний позволяют задать комментарий и визуальное преставление, отображаемые пользователю при входе и выходе из процедуры. Они обычно применяются в главной процедуре алгоритма.

Комментарий задается шаблоном комментария (атрибут `comment`) и параметрами комментария (`comment-args`).

Отметим, что атрибут `comment` имеет стереотип «`i18n`». Это означает, что в целях интернационализации комментарии могут быть указаны на различных языках. При этом вместо атрибута `comment` указываются атрибуты `comment-ru` и `comment-en`, содержащие комментарии на русском и английском языках соответственно.

Шаблон комментария содержит ссылки на параметры, записываемые в виде:

{*номер аргумента*}

Параметры комментария перечисляются через запятую. Каждый параметр представляет собой выражение, вычисляемое в момент отображения комментария. Таким образом, комментарии могут включать в себя значения переменных алгоритма.

Связь с визуальным представлением указывается во вложенном элементе `draw`, содержащем код, исполняемый при отображении визуального представления пользователю. Таким образом, визуальное представление может зависеть не только от текущего состояния, но и от значений переменных.

5.2.3. Описание операторов

С точки зрения удобства визуализации, операторы (абстрактный элемент `statement` на рис. 44) разделены следующим образом:

- неотображаемые операторы;
  - оператор вызова процедуры;
  - блочный оператор (абстрактный элемент `block` на рис. 44).
- отображаемые операторы:
  - оператор присваивания;
  - условные операторы (абстрактный элемент `condition` на рис. 44):
    - оператор ветвления;
    - оператор цикла с предусловием.

Оператор вызова процедуры

Оператор вызова процедуры описывается элементом `call-auto`. При этом указывается имя вызываемой процедуры (атрибут `id`).

Блочный оператор

Предлагаемый язык не содержит выделенной конструкции для описания блочных операторов. Вместо этого тело процедуры, цикла с предусловием и ветвей оператора ветвления являются блочными операторами.

Отображаемые операторы

Для отображаемых операторов указываются идентификаторы (атрибут `id`), описание (`description`) и уровень оператора (`level`), указывающий остановится ли визуализатор на данном операторе.

Значение уровня оператора по умолчанию равно нулю.

Оператор присваивания

В операторе присваивания могут быть изменены значения одной или нескольких переменных.

Для трассировки в обратном направлении требуется выполнить обращение операторов — построить код, исполняемый при обратном проходе (раздел 4.3).

При автоматическом обращении выполняемые действия записываются в элементе `action`. При этом для изменения переменных применяются

операторы обратимого присваивания, имеющие вид `@=`. При обратном проходе все изменения, выполненные операторами обратимого присваивания, автоматически «откатываются» (раздел 4.3).

В случае ручного обращения отдельно указываются действия, выполняемые при прямом (элемент `direct`) и обратном (элемент `reverse`) проходах. В этом случае проверка верности обращения может быть автоматизирована (раздел 6.1.3).

В случае отображения пользователю, для оператора присваивания указываются комментарий и связь с визуальным представлением, как это было отмечено выше для начального и конечного состояний.

Условные операторы

Для условного оператора указывается условие (атрибут `test`), позволяющее выбрать следующий оператор при прямом проходе.

Выбор следующего оператора при обратном проходе может выполняться как автоматически, так и вручную. Во втором случае соответствующее условие указывается в атрибуте `rtest`.

Для условных операторов также записываются комментарии и связь с визуальным представлением. При этом шаблоны комментариев и действия по обновлению визуального представления указываются отдельно для истинного (атрибут `comment-true` и элемент `draw-true`) и ложного (`comment-false`, `draw-false`) значений условия.

Оператор ветвления

Оператор ветвления задается элементом `if`, в который вложены описания ветвей, исполняемых при истинности (элемент `then`) и ложности (`else`) условия. Для каждой ветви указывается набор операторов, составляющих эту ветвь.

Отметим, что в случае укороченного оператора ветвления ветвь `else` опускается.

Оператор цикла с предусловием

Оператор цикла с предусловием задается элементом `while`, в который вложены описания операторов, составляющих тело цикла.

5.2.4. Переменные

При записи описания логики визуализатора переменные делятся на глобальные и локальные. Глобальные переменные доступны во всех процедурах, а локальные — только в той процедуре, в которой они объявлены.

Для доступа к переменным применяется @-нотация. При этом настоящее имя переменной не используется в явном виде, а подставляется автоматически.

Доступ к переменной осуществляется при помощи выражения вида:

```
@<имя переменной>
```

Например, оператор

```
@max = @a[@i]
```

выполняет присваивание переменной `max` значения `i`-го элемента массива `a`. Заметим, что при наличии в области видимости глобальной и локальной переменных с одним и тем же именем используется локальная переменная.

Для создания строкового представления автомата в процедуре `toString` введен синтаксис, позволяющий производить доступ к локальным переменным других процедур. Доступ к переменной, объявленной в другой процедуре, осуществляется следующим образом:

```
@<имя процедуры>@<имя переменной>
```

Например, оператор

```
buffer.append(@Main@i);
```

выполняет добавление к буферу значения локальной переменной `i` процедуры `Main`.

Для доступа к переменным модели из кода визуализатора применяются для глобальных переменных выражения вида:

```
<переменная модели>.<имя переменной>
```

а для локальных переменных:

<переменная модели>.<имя процедуры>_<имя переменной>

Например:

```
data.max = 0;
System.out.println(data.Main_i);
```

5.2.5. Пример описания визуализируемого алгоритма

Рассмотрим описание алгоритма визуализатора поиска максимума в массиве натуральных чисел.

Данная задача может быть решена следующей программой:

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

Здесь a — массив, в котором производится поиск максимума, а max — значение текущего максимума (после i-ой итерации — среди первых i элементов).

Отметим, что инициализация максимума нулем не приводит к ошибке, так как по условию задачи в массиве содержатся только натуральные числа.

На предложенном языке данный алгоритм может быть записан следующим образом:

```
1:    <algorithm>
2:      <variable name="a" value="new int[]{1, 2, 3, 1, 6}"
                  type="int[]" description="Массив для поиска"/>
3:      <variable name="max" value="0" type="int"
                  description="Текущий максимум"/>
4:      <auto id="Main" description="Ищет максимум в массиве">
5:        <variable name="i" type="int"
                  description="Переменная цикла"/>
6:        <start comment-ru="На экране изображен массив, в
                  котором будет осуществляться поиск максимума">
7:          <draw>@visualizer.updateArray(0, 0);</draw>
8:        </start>
9:        <step id="Initialization" description="Инициализация"
                  comment-ru="Инициализируем максимум нулем (так
                  как в массиве только натуральные числа).">
10:         <draw>@visualizer.updateArray(0, 0);</draw>
11:         <action>@max @= 0;</action>
12:       </step>
13:       <step id="LoopInit" description=":"
                  level="-1">:</step>
```

```
14:       <while id="Loop" description="Цикл"
                 test="@i &lt; @a.length"
                 rtest="@i &gt;= 0" level="-1">
15:         <if id="Cond" description="Условие"
                 test="@max &lt; @a[@i]"
                 true-comment-ru="{0} больше текущего максимума
                 ({1})"
                 false-comment-ru="{0} не больше текущего
                 максимума ({1})"
                 comment-args="new Integer(@a[@i]), new
                 Integer(@max)">
16:           <draw>@visualizer.updateArray(@i, 1);</draw>
17:           <then>
18:             <step id="newMax"
                  description="Обновление максимума"
                  comment-ru="Обновляем текущий максимум">
19:               <draw>@visualizer.updateArray(@i, 2);</draw>
20:               <action>@max @= @a[@i];</action>
21:             </step>
22:           </then>
23:         </if>
24:         <step id="inc" description=":" level="-1">:</step>
25:           <forward>@i = @i + 1;</forward>
26:           <reverse>@i = @i – 1;</reverse>
27:         </step>
28:       </while>
29:       <finish comment-ru="Максимум найден ({0})"
                 comment-args="new Integer(@max)">
30:         <draw>@visualizer.updateArray(0, 0);</draw>
31:       </finish>
32:     </auto>
33:   </algorithm>
```

Рассмотрим некоторые части этого описания подробнее.

Данный алгоритм содержит две глобальные переменные: a — массив, в котором выполняется поиск (объявлена во второй строке) и max — текущее значение максимума (объявлена в третьей строке).

В четвертой строке начинается описание основной процедуры main, которое заканчивается в строке 31. В процедуре объявлена локальная переменная — индекс текущего элемента массива (строка 4).

За описанием локальной переменной следует описание начального состояния (строки 6–8), включающее комментарий (строка 6) и связь с визуальным представлением (строка 7).

Далее приводится описание шагов алгоритма, начинающееся операторами присваивания начальных значений переменным `max` (строки 9–12) и `i` (строка 13). Отметим, что первое из этих присваиваний визуализируется, а второе — нет, так как для него указан уровень `-1`.

В строках с 14 по 28 описан оператор цикла с предусловием. Отметим, что для него применяется обращение вручную. Соответствующее условие указано в атрибуте `rtest`.

Тело оператора цикла включает два оператора: ветвления (строки 15–23) и присваивания (строки 24–27). При этом оператор присваивания использует косвенное обращение: в строке 25 указано действие, осуществляемое при прямом проходе, а в строке 26 — действие, выполняемое при обратном проходе.

5.3. Описание конфигурации визуализатора

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

- Описания групп, свойств и сообщений:
  - описание группы (элемент `group`);
  - описание свойств (элемент `property`):
  - описание сообщений (элемент `message`).
- Описания таблиц стилей:
  - описание стиля (элемент `style`);
  - описание таблицы стилей (элемент `style-set`);
  - описание шрифта (элемент `font`);
  - описание цвета (элемент `color`).
- Описание элементов управления:
  - описание панели (элемент `panel`);
  - описание кнопки (элемент `button`);
  - описание новой панели выбора (элемент `adjustablePanel`).

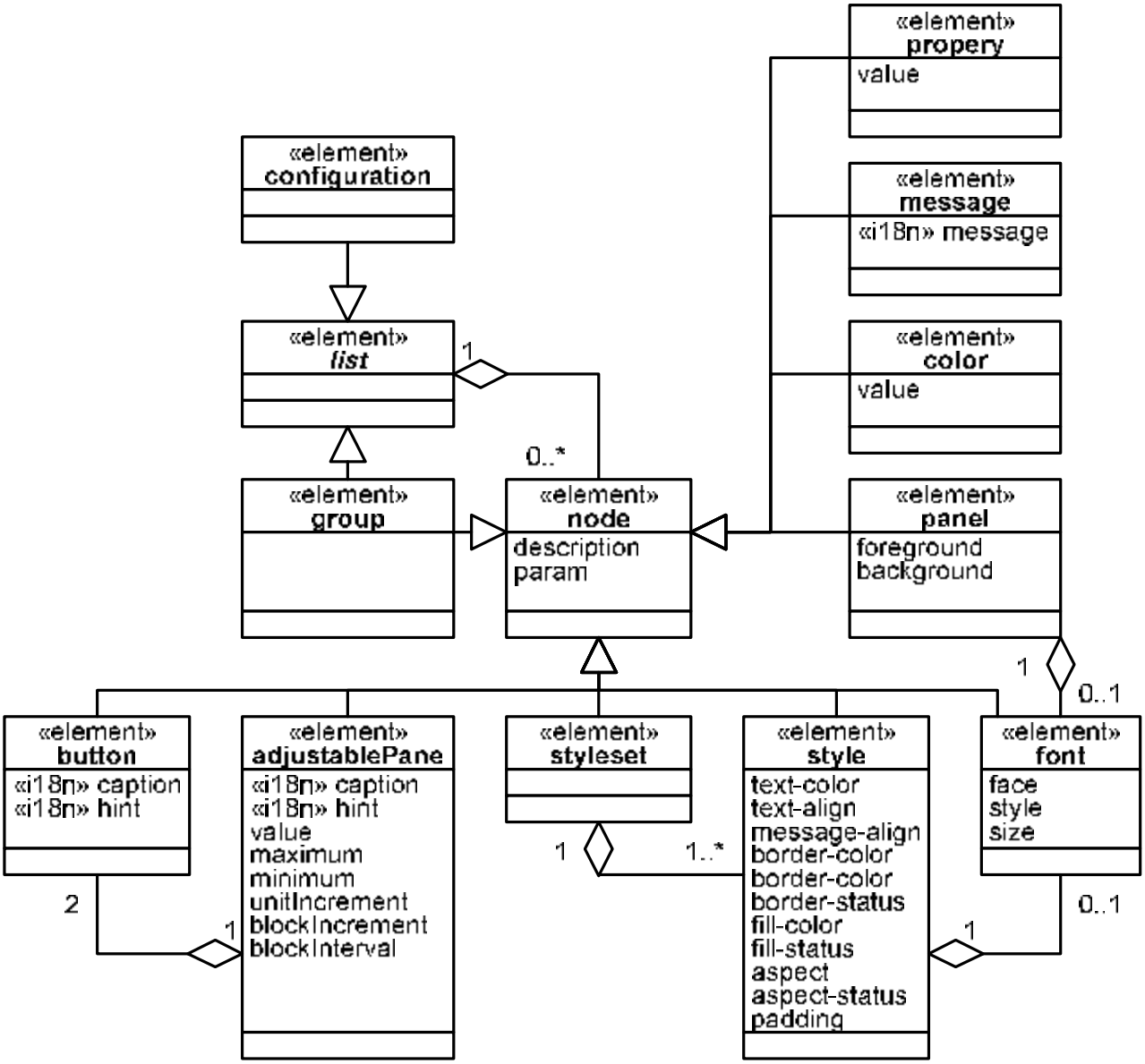


Рис. 45. Диаграмма элементов описания конфигурации визуализатора

Возможные комбинации этих элементов представлены на рис. 45.

5.3.1. Группы, свойства и сообщения

Группы (элемент `group`) служат для объединения свойств и сообщений, имеющих общий смысл, например, сообщения, отображаемые одним элементом управления. Группы также могут использоваться для конфигурирования нестандартных компонент.

Группа содержит (возможно пустой) список узлов конфигурации. Для каждого узла (абстрактный элемент `node` на рис. 45) указываются его описание (атрибут `description`) и локальное имя (атрибут `param`). Описания

предназначены для облегчения модификации конфигурации в будущем и напрямую не используются.

Имена предназначены для идентификации узлов конфигурации визуализатора. Каждый узел имеет локальное имя (атрибут `param`) и полное имя, составленное путем конкатенации имен групп, в которых входит этот узел. Например, узел с локальными именем `test`, входящий в группу `example1`, которая, в свою очередь, входит в группу `examples`, имеет полное имя

```
examples-example1-test
```

Свойства (элемент `property`) описывают конфигурацию, не зависящую от языкового окружения, например, минимальные и максимальные значения или интервалы времени. Значение свойства хранится в атрибуте `value`.

Сообщения (элемент `message`) описывают конфигурацию, зависящую от языкового окружения, например, сообщения, выводимые пользователю.

5.3.2. Таблицы стилей

Таблицы стилей предназначены для описания конфигурации элементов визуального представления.

Шрифты

Элемент шрифт (`font`) описывает шрифт, применяемый для вывода надписей и сообщений. Для шрифта указывается его начертания (атрибут `face`), размер в пункт пунктах (`size`) и стиль (`style`).

Определены следующие начертания шрифтов [78]:

- `Serif` — с засечками (*Times New Roman, Roman*).
- `SansSerif` — без засечек (*Arial, Helvetica*).
- `Symbol` — для специальных символов (*Symbol*).
- `Monospaced` — моноширинной (*Courier New, Courier New*).

Для стиля шрифта определены следующие значения:

- `Plain` — обычное начертание.
- `Bold` — полужирное начертание.
- `Italic` — курсивное начертание.

- `BoldItalic` — полужирный курсив.

Стили

Стиль (элемент `style`) описывает, как элемент визуального представления отображается на экране, в частности:

- цвет (атрибут `text-color`) и выравнивание (`text-align`) текста;
- отображать ли рамку элемента (`border-status`) и ее цвет (`border-color`);
- отображать ли фон элемента (`fill-status`) и его цвет (`fill-color`);
- сохранять ли при масштабировании ширины элемента к его длине (`aspect-status`) и само это отношение (`aspect`);
- выравнивание блока текста (`message-align`) и отступ от него до границы элемента (`padding`).

Шрифт, которым отображается текст компоненты задается вложенным элементом `font`.

Таблицы стилей

Таблица стилей (элемент `styleset`) задает набор стилей, используемых для отображения элемента визуального представления в зависимости от состояния. Стили нумеруются, начиная с нуля.

Если в стиле не определены некоторые свойства, то их значения выбираются из описания нулевого стиля в наборе. Если не определено свойство нулевого стиля, используется значение по умолчанию.

Описание цвета

Элемент `color` служит для описания цветов элементов, обычно он вложен в описание других элементов. Цвет (атрибут `value`) записывается в виде шестизначного шестнадцатеричного числа в формате `RRGGBB`.

5.3.3. Элементы управления

Описание панели

Элемент `panel` предназначен для описания вида панелей (наследников класса `Java.awt.Panel`). Для панели указываются цвета текста (атрибут `foreground`) и фона (`background`). Также для панели может быть указан шрифт (раздел 5.3.2).

Описание кнопки

Элемент `button` предназначен для описания кнопок (наследников класса `Java.awt.Button`). Для каждой кнопки указываются надпись (атрибут `caption`) и текст всплывающей подсказки (`hint`).

Описание новой панели выбора

Элемент `adjustablePanel` описывает конфигурацию панели выбора, позволяющей выбирать целые числа в некотором диапазоне. Этот элемент управления обычно применяется для задания входных данных визуализатора.

Для панели выбора указываются:

- описание (атрибут `description`);
- шаблон надписи (`caption`);
- текст всплывающей подсказки (`hint`);
- начальное (`value`), максимальное (`maximum`) и минимальное (`minimum`) значения;
- начальное (`value`), максимальное (`maximum`) и минимальное значение (`minimum`);
- приращение при маленьком (`unitIncrement`) и большом (`blockIncrement`) шагах;
- интервал, в течение которого используется большой шаг (`blockInterval`).

Большой шаг используется тогда, когда между двумя нажатиями на кнопку увеличения (уменьшения) значения проходит меньше `blockInterval` миллисекунд, в противном случае используется маленький шаг.

Выводы по главе 5

1. Разработан формат записи визуализируемого алгоритма.
2. Разработан формат записи конфигурации визуализатора.
3. Разработан язык описания визуализаторов алгоритмов.

ГЛАВА 6. ВНЕДРЕНИЕ ПРЕДЛОЖЕННЫХ МЕТОДОВ

В главе 1 показана актуальность создания новой системы визуализации алгоритмов дискретной математики.

На базе методов и подходов, предложенных в главах 2−4, была разработана система визуализации *Vizi* (раздел 6.1). Пример использования системы *Vizi* приведен в разделе 6.2.

Ее применение позволило существенно сократить время и квалификацию разработчиков, которые необходимы для построения визуализаторов сложных алгоритмов (раздел 6.3).

6.1. Система визуализации *Vizi*

Система визуализации *Vizi* [88] состоит из двух основных частей: динамической, функционирующей во время исполнения визуализатора, и статической, работающей на этапе создания визуализатора.

Динамическая часть состоит из библиотеки времени исполнения и кода, сгенерированного по описанию визуализатора. При этом динамическая часть фиксирует структуру визуализатора (раздел 6.1.1).

Статическая часть состоит из XSLT-скриптов, генерирующих код и конфигурационные файлы по описанию визуализатора (раздел 6.1.2). Кроме того статическая часть позволяет отлаживать описание алгоритма визуализатора (раздел 6.1.3).

Для системы визуализации *Vizi* разработан процесс построения визуализаторов (раздел 6.1.4).

6.1.1. Структура визуализатора

В разделе 2.1.2 выделены основные части визуализатора. В рамках системы визуализации *Vizi* предложенное разбиение на части было уточнено и дополнено (рис. 46). Серым цветом помечены компоненты, входящие в систему *Vizi*, штриховкой — компоненты, автоматически генерируемые по описанию визуализатора, а белым — компоненты, создаваемые вручную.

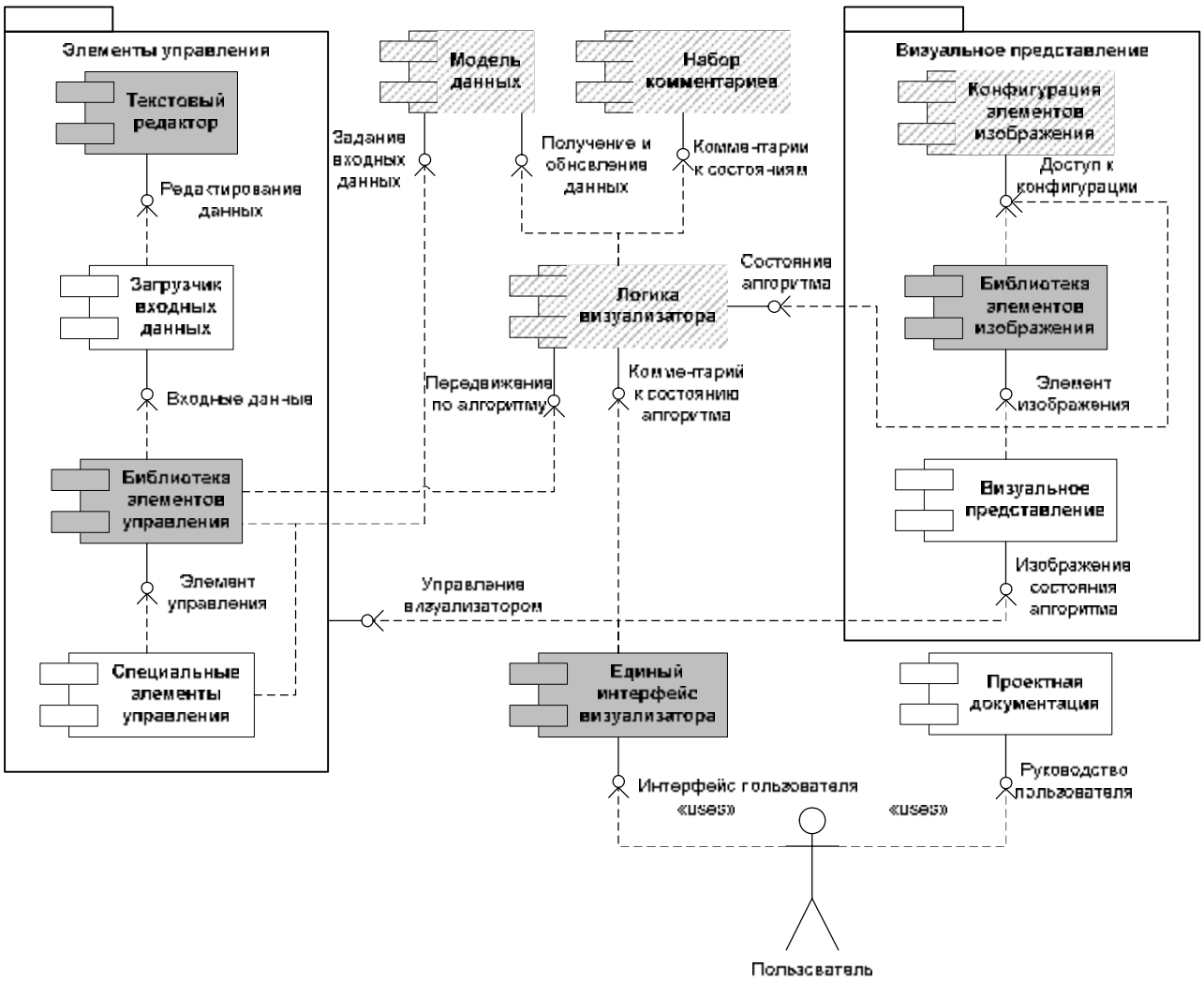


Рис. 46. Диаграмма компонентов визуализатора

В качестве интерфейса визуализатора предложено использовать *единый интерфейс визуализатора*.

Отметим, что единый интерфейс визуализатора не взаимодействует с набором комментариев напрямую, а использует для этого логику визуализатора. Такое разделение позволяет создавать визуализаторы, умеющие отображать комментарии на различных языках программирования.

Часть визуализатора *элементы управления* была уточнена и разбита на четыре компонента:

1. Библиотека элементов управления. Содержит часто используемые элементы управления, такие как: кнопки, списки, панели настройки.

2. Специальные элементы управления. Содержит элементы управления, созданные пользователем специально для этого визуализатора. Может отсутствовать.
3. Загрузчик входных данных. Отвечает за загрузку входных данных введенных пользователем.
4. Текстовый редактор. Позволяет пользователю вводить данные.

Часть визуализатора *визуальное представление* также разбита на несколько компонент:

1. Собственно визуальное представление. Отображает текущее состояние алгоритма с использованием библиотеки элементов изображения и конфигурации визуализатора.
2. Библиотека элементов изображения. Содержит часто используемые элементы управления, такие как:
   - надписи;
   - прямоугольники;
   - эллипсы.
3. Конфигурация элементов изображения. Содержит конфигурацию, заданную в описании визуализатора.

Отметим, что выделение компонентов *элементов управления* и *визуального представления* упрощает создание визуализаторов, так как некоторые из выделенных компонентов включаются в систему визуализации или генерируются автоматически.

Таким образом, единственной не изменившейся частью по сравнению диаграммой, приведенной в разделе 2.1.2, остается *проектная документация*, которая создается вручную при минимальной автоматизации.

6.1.2. Статическая часть

Статическая часть системы визуализации *Vizi* предназначена для обработки описания визуализатора и сборки проекта.

Диаграмма компонентов статической части изображена на рис. 47.

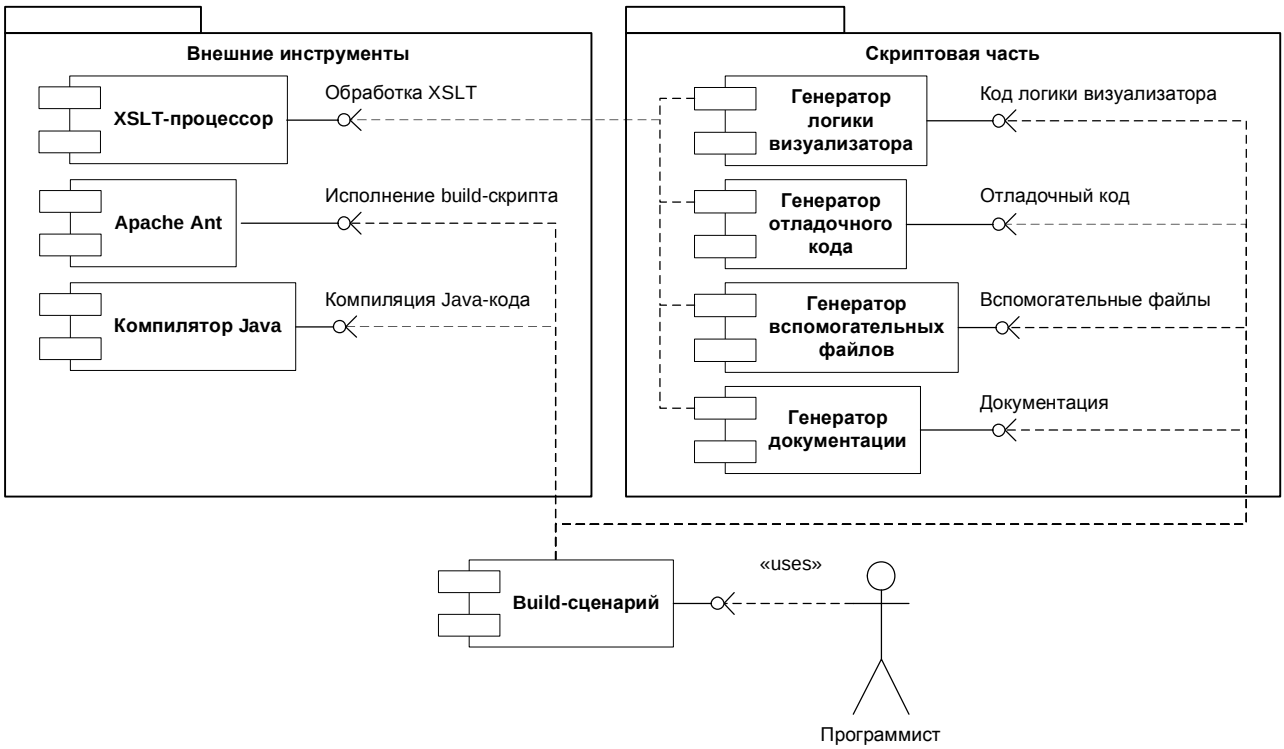


Рис. 47. Диаграмма компонентов статической части системы визуализации *Vizi*

Для построения проекта визуализатора программист запускает build-сценарий, исполняемый при помощи инструмента *Apache Ant* [73]. Процесс построения визуализатора состоит из нескольких этапов.

1. *Проверка корректности описания визуализатора*. Описание визуализатора проверяется с применением описаний типа документа *DTD* [77] и *XML Schema* [89–91].
2. *Генерация исходного кода логики визуализатора и модели данных*. При помощи языка *XSLT* [92] по описанию визуализатора генерируется исходный код системы взаимодействующих конечных автоматов на языке *Java*.
3. *Создание наборов комментариев для указанных языков*. При помощи языка *XSLT* по описанию визуализатора генерируются файлы свойств [85], содержащие наборы комментариев.
4. *Компиляция проекта визуализатора*.
5. *Упаковка (packaging* [44]*) визуализатора*.
6. *Создание файлов*, запускающих визуализатор.

По требованию пользователя может быть сгенерирована часть документации на визуализатор.

Отметим, что все указанные этапы полностью автоматизированы.

6.1.3. Отладка описания визуализатора

Статическая часть предоставляет возможности по отладке описания визуализатора, которая состоит из двух этапов:

1. Отладка визуализируемой программы.
2. Отладка программы визуализатора.

На первом этапе осуществляется поиск ошибок переноса визуализируемой программы в описание алгоритма, а на втором — отладка автоматически построенной программы визуализации.

Отладка визуализируемой программы

Для упрощения отладки визуализируемой программы предназначена цель `debug-source` [88], определенная в *build*-сценарии. При запуске этой цели строится реализация визуализируемой программы без использования конечных автоматов. При этом элементы `if` и `while` (раздел 5.2.3) описания визуализируемой программы преобразуются в соответствующие операторы, а элементы `step` и `call-auto` (раздел 5.2.2) — в части кода и вызовы процедур соответственно.

В полученный исходный код переносятся комментарии из описания алгоритма. В нем расставляются отступы, что может существенно упростить отладку. Сгенерированный код помещается в файл с именем, указанным в описании визуализатора, к которому добавлен суффикс `Debug`. Полученный исходный код может отлаживаться с использованием любого программного инструмента.

Отладка программы визуализатора

Отладка программы визуализации производится при помощи валидатора автоматов. Валидатор осуществляет последовательный запуск визуализатора на все большем количестве шагов (до тех пор, пока не дойдет до конца

визуализируемой программы) и проверяет правильность значений на обратном проходе. Запуск осуществляется на данных, заданных в описании визуализатора в качестве значений по умолчанию.

При обратном проходе сравниваются значения, выданные процедурой преобразования состояния автомата в строку (`toString`), и значения, выданные ей при прямом проходе. При неравенстве строк на соответствующих шагах выдается сообщение об ошибке.

Для использования валидатора служит цель `check` [88], определенная в *build*-сценарии. Запуск этой цели создает командный файл, предназначенный для запуска валидатора, и помещает его в каталог файлов визуализатора.

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 прямом проходе;
- результат функции `toString` при обратном проходе;
- результат функции `toString` для шага, до которого производился ошибочный запуск;
- результат функции `toString` для шага, предшествующего ошибке (шаг с номером на единицу больше).

Работа валидатора завершается с кодом возврата `1`, в отличие от нормального завершения, когда код 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 описании визуализатора 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 (раздел 5.2) проверять, а второй — с какого шага начинать проверку. Таким образом, проверка может выполняться следующим образом: сначала быстрая проверка на больших шагах, а затем при обнаружении ошибки она локализуется с использованием маленьких шагов.

6.1.4. Процесс построения визуализатора

Опишем порядок разработки визуализатора с применением технологии *Vizi*.

1. Постановка задачи и анализ литературы.
2. Создание визуализируемой программы:
    - реализация алгоритма;
    - отладка программы, реализующей алгоритм.
3. Проектирование визуализатора:
    - выделение «интересных» состояний;
    - проектирование визуального представления;
    - проектирование набора комментариев;
    - проектирование элементов управления.
4. Построение описания визуализируемой программы:
    - выделение модели данных;
    - преобразование программы;
    - запись описания отдельных процедур;
    - запись описания визуализируемой программы;
    - отладка описания визуализируемой программы;
    - интеграция набора комментариев в описание визуализируемой программы.
5. Реализация визуального представления.
6. Реализация элементов управления.
7. Интеграция и отладка визуализатора:

- интеграция визуального представления в описание визуализатора;
- генерация кода по описанию визуализатора;
- отладка визуализатора.

8. Оформление проектной документации.

Этапы 1–3 совпадают с первыми тремя этапами ручной разработки визуализатора, приведенными в разделе 2.2.1.

На четвертом этапе производится построение описания визуализируемой программы, созданной на втором этапе. Первоначально из программы выделяется модель данных для того, чтобы визуализатор имел доступ к переменным, используемых при визуализации. Так как рассматриваются только программы в приведенной форме (раздел 2.5), то предварительно программу необходимо преобразовать к такому виду (раздел 3.4). После этого записываются описания отдельных процедур, которые затем объединяются в описание визуализируемой программы. Затем созданное описание отлаживается при помощи средств, предоставляемых пакетом *Vizi*. Этап завершается добавлением к описанию визуализируемой программы комментариев, разработанных на третьем этапе.

Четвертый этап заменяет этапы 4–6 ручной разработки визуализатора.

На пятом и шестом этапах производится реализация визуального представления и элементов управления, в соответствии с концепцией, разработанной на третьем этапе. Эти этапы совпадают с седьмым и восьмым этапами ручной разработки визуализатора.

На седьмом этапе интегрируются результаты четвертого и пятого этапов, и производится отладка визуализатора. Данный этап заменяет этапы 9–11 ручной разработки визуализатора.

Уменьшение числа этапов достигнуто за счет их автоматизации, как указано в разделе 2.2.2.

6.2. Пример построения визуализатора

Проиллюстрируем применении системы визуализации *Vizi* и порядка построения визуализаторов (раздел 6.1.4) на простом примере — построении визуализатора алгоритма поиска максимума в массиве натуральных чисел.

6.2.1. Постановка задачи и анализ литературы

Задан массив из N натуральных чисел. Требуется найти в нем максимальное число.

Алгоритм решения этой задачи очень прост, и поэтому сразу перейдем к следующему этапу.

6.2.2. Создание визуализируемой программы

Задача поиска максимума в массиве решается следующей программой:

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

Здесь `a` — массив, в котором производится поиск максимума, `max` — значение текущего максимума (после `i`-ой итерации — среди первых `i` элементов).

Отметим, что инициализация максимума нулем не приводит к ошибке, так как по условию задачи в массиве содержатся только натуральные числа.

6.2.3. Проектирование визуализатора

Выделение «интересных» состояний

В визуализаторе поиска максимума наиболее интересным являются шаги, осуществляющие ветвление и связанное с ним обновления текущего максимума. Кроме того, должны быть выделены начальное и заключительное состояния.

Специально визуализировать переход к следующему элементу массива вряд ли имеет смысл. Список «интересных» состояний алгоритма приведен таблице 7.

Таблица 7. «Интересные» состояния алгоритма

Шаг	Пояснение
Начальное состояние	Описывается цель алгоритма
Инициализация максимума	Инициализация текущего максимума нулем и соответствующие пояснения
Проверка на обновление максимума	Проверяется необходимость обновить текущий максимум
Обновление текущего максимума	Присваивание текущему максимуму значения текущего элемента
Заключительное состояние	Отображение результатов вычисления

Проектирование визуального представления

Основными данными в визуализаторе являются элементы массива и значение текущего максимума. Их значения постоянно должны быть представлены на экране. Поэтому для визуального представления будем использовать схему визуального представления, изображенную на рис. 48.

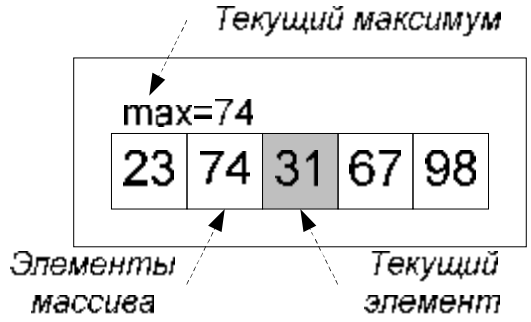


Рис. 48. Схема визуального представления

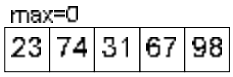
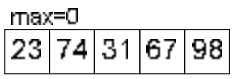
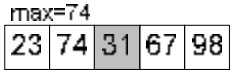
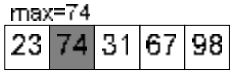
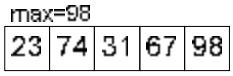
Отметим, что индекс текущего элемента в массиве явно не отображается. Вместо этого текущий элемент выделяется цветом.

Для визуализации выделенных «интересных» состояний используются слайды, представленные в таблице 8.

Проектирование набора комментариев

Для каждого из «интересных» состояний разрабатываются комментарии. Отметим, что, так как шаг «*Проверка на обновление максимума*» соответствует оператору ветвления, то для него должны быть разработаны два комментария: для истинного и ложного условий.

Таблица 8. Слайды для «интересных» состояний

Шаг	Пояснение	Слайд
Начальное состояние	Текущий элемент и максимум не подсвечивается	
Инициализация максимума	Текущий элемент не подсвечивается	
Проверка на обновление максимума	Текущий элемент подсвечивается зеленым цветом	
Обновление текущего максимума	Текущий элемент подсвечивается красным цветом	
Заключительное состояние	Текущий элемент не подсвечивается	

Отметим, что комментарии могут содержать параметры, места включения которых задаются следующим образом:

{*номер параметра*}

Значения параметров вычисляются и подставляются в процессе визуализации.

Приведем набор комментариев для визуализатора поиска максимума (таблица 9). Выражения, соответствующие параметрам визуализатора, указаны в столбце «Параметры».

Таблица 9. Набор комментариев

Шаг алгоритма	Комментарий	Параметры
Начальное состояние	На экране изображен массив, в котором будет осуществляться поиск максимума	
Инициализация максимума	Инициализируем максимум нулем (так как в массиве только натуральные числа)	
Проверка на обновление максимума (истина)	{0} больше текущего максимума ({1})	a[i], max
Проверка на обновление максимума (ложь)	{0} не больше текущего максимума ({1})	a[i], max
Обновление текущего максимума	Обновляем текущий максимум	
Заключительное состояние	Максимум найден ({0})	max

Проектирование элементов управления

Список элементов управления визуализатора приведен в таблице 10. Отметим, что только два из них не являются стандартными, а остальные входят в библиотеку *Vizi*.

Таблица 10. Элементы управления

Элемент управления	Назначение	Стандартный
<< и >>	Шаг назад и вперед	Да
Рестарт	Начало визуализации сначала	Да
Авто и Стоп	Вход в автоматический режим и выход из него	Да
<< Задержка: 1000 >>	Регулирование задержки между шагами в автоматическом режиме. Задержка измеряется в миллисекундах	Да
?	Вывод информации о визуализаторе	Да
Случайно	Генерация случайного набора данных	Нет
Сохранить/Загрузить	Загрузка/сохранения состояния визуализатора	Да
<< Элементов: 7 >>	Задание количество элементов в массиве	Нет

Элементы управления визуализатором компонуются, как показано на рис. 49.

Рис. 49. Область элементов управления

6.2.4. Построение описания визуализируемой программы

Выделение модели данных

В визуализируемой программе используются три переменные:

- `max` — значение текущего максимума;
- `a` — массив, в котором выполняется поиск;
- `i` — индекс текущего элемента массива `a`.

Эти переменные должны быть вынесены в модель данных для того, чтобы визуализатор имел доступ к ним. При этом значения переменных `max` и `a` будут визуализироваться, а значения переменной `i` — нет. Поэтому сделаем переменные `max` и `a` глобальными, а переменную `i` — локальной для процедуры `main` (раздел 5.2.4). Описание этих переменных имеет следующий вид:

```
<variable
  description = "Массив для поиска"
  name        = "a"
  type        = "int[]"
  value       = "new int[]{1, 2, 3, 1, 3, 5, 6}"
/>
<variable
  description = "Текущий максимум"
  name        = "max"
  type        = "int"
  value       = "0"
/>
<variable
  description = "Переменная цикла"
  name        = "i"
  type        = "int"
/>
```

Отметим, что для глобальных переменных указаны их начальные значения, которые могут быть использованы при отладке.

В модель данных также вынесем переменную, содержащую ссылку на экземпляр визуализатора. Она будет применяться для формирования визуального представления текущего шага. Опишем ее следующим образом:

```
<variable
  description = "Экземпляр апплета"
  name        = "visualizer"
  type        = "FindMaximumVisualizer"
  value       = "null"
/>
```

Для ссылок на переменные, вынесенные в модель данных, применяется @-нотация (раздел 3.2.3). После выделения модели данных программа имеет следующий вид:

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

Преобразование программы

Сначала преобразуем цикл со счетчиком в цикл с предусловием. После этого преобразования программа выглядит следующим образом:

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

Теперь преобразуем выражение `@i++` к виду простого присваивания. В соответствии с семантикой языка *Java* [44], это приводит к оператору

```
@i = @i + 1
```

Для автоматического построения кода, осуществляющего обратную трассировку алгоритма, требуется заменить операторы присваивания на операторы обратимого присваивания (раздел 3.3.2). В результате, визуализируемая программа будет записана следующим образом:

```
void main() {
  @max @= 0;
  @i @= 0;
  while (@i < @a.length)) {
    if (@max < @a[@i]) {
      @max @= @a[@i];
    }
    @i @= @i + 1;
  }
}
```

Запись описания отдельных процедур

Визуализируемая программа содержит одну процедуру `main`, описание которой будет иметь вид (разделы 5.2.2 и 5.2.3):

```
<auto id="Main" description="Ищет максимум в массиве">
  <variable description="Переменная цикла" name="i"
        type="int"/>
  <step id="Initialization" description="Инициализация">
    <action>@max @= 0;</action>
  </step>
  <step id="LoopInit" description="Начало цикла">
    <action>@i @= 0;</action>
  </step>
```

```
  <while  id="Loop" description="Цикл" test="@i &lt;
        @a.length">
    <if id="Cond" description="Условие" test="@max &lt;
        @a[@i]">
      <then>
        <step id="newMax" description="Обновление
        максимума">
          <action>@max @= @a[@i];</action>
        </step>
      </then>
    </if>
    <step id="inc" description="Инкремент">
      <action>@i @= @i + 1;</action>
    </step>
  </while>
</auto>
```

Отметим, что `i`, как локальная переменная процедуры `main`, описана в теле этой процедуры.

Запись описания визуализируемой программы

Для построения полного описания визуализируемой программы добавим к описанию процедуры `main` описания переменных модели данных и метода `toString`, который применяется при отладке (раздел 5.2.1):

```
<algorithm>
  <... описание переменных ...>

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

  <... описание процедуры main ...>
</algorithm>
```

Для получения описания визуализатора в целом, требуется включить описание программы в описание визуализатора:

```
<!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"
```

```
  name-ru="Поиск максимума"
  author-ru="Георгий Корнеев"
  author-email="kgeorgiy@rain.ifmo.ru"

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

  copyright-ru="Кафедра КТ, СПбГУ ИТМО, 2005"

  preferred-width="400"
  preferred-height="250"
>
  <... описание алгоритма ...>
  <configuration>
  </configuration>
</visualizer>
```

В атрибутах элемента `visualizer` описания визуализатора указываются пакет и имена классов визуализатора, а также информация об авторе.

Отладка описания визуализируемой программы

Отладка описания визуализатора выполняется при помощи средств, входящих в пакет *Vizi*.

Для этого вызовом *build*-скрипта с параметром `debug-source` [88] генерируется файл с реализацией алгоритма, заданного описанием:

```
/**
  * Ищет максимум в массиве.
  */
private final void Main() {
  // Инициализация
  d.max = 0;
  // Начало цикла
  d.Main_i = 0;
  // Цикл
  while (d.Main_i < d.a.length) {
    // Условие
    if (d.max < d.a[d.Main_i]) {
      // Обновление максимума
      d.max = d.a[d.Main_i];
    }
    // Инкремент
    d.Main_i = d.Main_i + 1;
  }
}
```

Данный файл позволяет убедиться в том, что описание корректно. Отметим, что сгенерированный код содержит комментарии с информацией,

перенесенной из атрибутов description описания визуализируемой программы.

Проверим также корректность автоматического обращения визуализируемой программы. Для этого вызовом *build*-скрипта с параметром debug-check создадим тест и запустим его командой CheckFindMaximum из каталога deploy [88]. Удостоверимся что, тест прошел успешно:

```
Check 1 steps...  OK ()
Check 2 steps...  OK ()
…
Check 36 steps...  OK ()
Check 37 steps...  OK ()
OK
```

Интеграция набора комментариев в описание визуализируемой программы

Добавим разработанные комментарии к описанию алгоритма.

Для отображения комментариев в начальном и конечном состояниях к описанию процедуры main требуется добавить шаги start и finish:

```
<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"/>
  <step id="Initialization" description="Инициализация"
      comment-ru="Инициализируем максимум нулем (так как
        в массиве только натуральные числа).">
    <action>@max @= 0;</action>
  </step>
  <step id="LoopInit" description="…" level="-1">…</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})"
      false-comment-ru="{0} не больше текущего максимума
        ({1})"
      comment-args="new Integer(@a[@i]), new
        Integer(@max)"
    >
      <then>
        <step id="newMax" description="Обновление
        максимума"
            comment-ru="Обновляем текущий максимум">
```

```
            <action>@max @= @a[@i];</action>
          </step>
        </then>
      </if>
      <step id="inc" description="…" level="-1">…</step>
        <action>@i @= @i + 1;</action>
      </step>
    </while>
    <finish comment-ru="Максимум найден ({0})"
        comment-args="new Integer(@max)"/>
  </auto>
```

Здесь и далее полужирным шрифтом выделены внесенные дополнения.

6.2.5. Реализация визуального представления

Разработанное визуальное представление весьма просто (таблица 8) и его отображение может производиться вызовом одного метода

`updateArray(<индекс элемента>, <номер цвета>)`

При вызове этого метода первым параметром передается индекс выделенного элемента, а вторым — способ его отображения: 0 — без выделения, 1 — зеленый, 2 — красный.

6.2.6. Реализация элементов управления

Визуализатор содержит два нестандартных элемента управления: кнопку генерации случайного набора данных и панель задания количества элементов. Первый их них может быть реализован на основе кнопки с подсказкой (`HintedButton`), а второй — на основе панели выбора (`AdjustablePanel`), входящих в пакет *Vizi* [88].

Отметим, что в результате воздействия на эти элементы управления изменяются входные данные. Поэтому при их использовании визуализатор должен переходить в начальное состояние.

6.2.7. Интеграция и отладка визуализатора

Интеграция визуального представления в описание визуализатора

Добавим вызовы, осуществляющие обновление визуального представления, к шагам, имеющим неотрицательный уровень, путем указания их в элементах `draw` (раздел 5.2.3):

```
<auto id="Main" description="Ищет максимум в массиве">
  <variable description="Переменная цикла" name="i"
        type="int"/>
  <start …> <draw>@visualizer.updateArray(0,
        0);</draw></start>
  <step …>
    <draw>@visualizer.updateArray(0, 0);</draw>
    <action>@max @= 0;</action>
  </step>
  <step …>…</step>
  <while …>
    <if …>
      <draw>@visualizer.updateArray(@i, 1);</draw>
      <then>
        <step …>
          <draw>@visualizer.updateArray(@i, 2);</draw>
          <action>@max @= @a[@i];</action>
        </step>
      </then>
    </if>
    <step …>…</step>
  </while>
  <finish …><draw>@visualizer.updateArray(0,
        0);</draw></finish>
</auto>
```

Генерация кода по описанию визуализатора

Код визуализатора генерируется вызовом *build*-скрипта с параметром `all` [88].

В результате этого по описанию будет сгенерирован код, реализующий, в том числе, и логику визуализатора (раздел 6.1.2). Для данного визуализатора будут автоматически построена пара конечных автоматов с девятью состояниями (раздел 4.3).

На рис. 50 приведен снимок экрана построенного визуализатора, запущенного на массиве значений `56 15 59 10 87`.

Отладка визуализатора

После построения визуализатора запустим его командой `FindMaximum_ru` из каталога `deploy` [88] и проверим его работоспособность. Если бы возникли ошибки при отображении состояний, то их устранение было бы произведено обычным способом.

Рис. 50. Визуализатор в одном из состояний

Оформление проектной документации

Материал раздела 6.2 может являться основой для создания проектной документации к визуализатору рассмотренного алгоритма.

6.2.8. Выводы

Из приведенного примера следует, что построение визуализаторов с применением технологии и пакета *Vizi* автоматизировано.

Полное описание визуализатора алгоритма поиска максимума приведено в приложении 1, а его исходный код — в приложении 2.

6.3. Сравнение с существующими подходами

6.3.1. Сравнение проектов визуализаторов

Данные о затратах времени, требующегося на построении визуализатора, обычно не публикуются, поэтому для его оценки приходится пользоваться косвенными данными. При этом считается, что время, затраченное на построение программы, примерно пропорционально размеру ее кода.

Для объективного сравнения размеров кода различных визуализаторов требуется исключить как можно больше несущественных факторов, таких как используемый язык программирования, библиотеки и т.п. Поэтому в качестве базы для сравнения было решено использовать визуализаторы, построенные на основе библиотеки *BaseApplet* [88], также разработанной автором, и

являющейся предшественником *Vizi*, но не содержащей модулей автоматизации построения визуализаторов.

Отметим сходства и различия *Vizi* и *BaseApplet*, влияющие на сравнение:

1. При построении визуализаторов используется один язык программирования — *Java* [44] и единый стиль программирования, описанный в работе [75].
2. Визуализаторы выполнялись людьми примерно одинаковой квалификации — студентами 1-2 курсов.
3. К проектам визуализаторов предъявлялись схожие требования.
4. *Vizi* содержит средства автоматизации построения визуализаторов, разработанные на основе предложенных методов.

Для сравнения выбирались визуализаторы алгоритмов, отображающие одинаковый объем информации, что позволило уменьшить влияния выбранного способа отображения на результат сравнения.

При сравнении учитывался размер исходного кода и конфигурации визуализатора после удаления комментариев и пробельных символов.

Из результатов сравнения, приведенных в таблице 11, следует, что применение системы визуализации *Vizi* позволяет сократить размер кода визуализатора. Отметим, что с усложнение визуализируемого алгоритма уменьшение размера кода в процентном отношении имеет тенденцию к увеличению.

6.3.2. Визуализаторы, построенные на основе *Vizi*

На основе системы визуализации *Vizi* с использованием автоматов для представления логики визуализаторов было построено более пятидесяти визуализаторов. В том числе такие простые, как алгоритм для работы с деревом отрезков (одна пара автоматов и 15 состояний), так и такие сложные как алгоритмы работы с 2-3 деревьями (семь пар автоматов и 195 состояний) и алгоритм Малхотры-Кумара-Махешвари (девять пар автоматов и 89 состояний).

Таблица 11. Сравнение размеров исходного кода визуализатора

Алгоритм	*BaseApplet*				*Vizi*				Уменьшение размера
	Автор	Размер кода (КБ)	Размер конфигурации (КБ)	Общий размер (КБ)	Автор	Размер кода (КБ)	Размер описания визуализатора (КБ)	Общий размер (КБ)	
Алгоритм Бойера-Мура	Пак С.	15.01	2.46	18.69	Наумов Р.	3.973	8.09	12.06	35.5%
Поиск двусвязных компонент графа	Кузнецов А.	22.20	2.89	26.54	Коломейцева О.	11.952	10.92	22.87	13.8%
Обходы в ширину и глубину	Прощенко Ю.	22.21	6.76	32.34	Гунич И.	13.835	11.36	25.20	22.1%
Алгоритм Форда-Фалкерсона	Штучкин А.	37.38	5.61	45.80	Кулев В.	19.095	7.48	26.57	42.0%
Алгоритм Форда-Беллмана	Гаврилов М.	44.44	4.64	51.39	Кулагин Д.	17.032	9.05	26.08	49.3%
Дерево отрезков	Дронь В.	47.95	8.71	61.01	Лагунов И.	15.897	11.73	27.63	54.7%
2-3 деревья	Суясов Д.	71.49	7.19	82.28	Постников Д.	12.79	20.41	33.20	59.6%
RB деревья	Краюхин Д.	83.18	8.02	95.20	Акишев И.	15.32	47.18	62.50	34.4%

Таблица 12. Визуализаторы, выполненные на основе *Vizi*

Алгоритм	Автор	A	C
Построение кратчайшего дерева в ориентированном графе	Пименов С.	2	26
Битонический алгоритм для задачи 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	Паращенко 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

Список визуализаторов, построенных на основе системы визуализации *Vizi* в 2003-2004 учебном году, представлен в таблице 12.

Здесь в столбце «A» обозначено количество пар автоматов, а в столбце «C» — суммарное количество состояний в них.

В общей сложности с 2003 по 2006 год на основе системы *Vizi* было построено более 80 визуализаторов алгоритмов, многие из которых опубликованы на сайтах [70] и [76].

6.3.3. Выполнение требований к визуализаторам

Покажем, что система визуализации *Vizi* позволяет создавать визуализаторы, соответствующие требованиям, приведенным в разделе 1.1.2.

1. *Возможность ввода данных, на которых демонстрируется алгоритм*. Библиотека времени исполнения предоставляет стандартный редактор для входных данных и средства, облегчающие их разбор.
2. *Двунаправленность*. Код, осуществляющий двунаправленное исполнение, генерируется автоматически по описанию визуализатора.
3. *Автоматический режим работы*. Обеспечивается стандартными элементами управления.
4. *История*. Код, осуществляющий двунаправленное исполнение, генерируется автоматически по описанию визуализатора.
5. *Отображение хода выполнения алгоритма*. Автоматически отображаются шаги визуализатора, указанные в его описании.
6. *Наличие комментариев для шагов алгоритма (комментарии)*. Комментарии указываются для каждого шага алгоритма визуализатора. Поддерживаются многоязычные комментарии.
7. *Простота использования*. Все визуализаторы имеют единый интерфейс, что облегчает их использование.
8. *Свободный доступ*. При построении визуализатора генерируются HTML-страницы, предназначенные для публикации на Интернет-сайтах.
9. *Платформонезависимость*. Визуализаторы реализуются на языке *Java*.
10. *Автономность*. При построении визуализатора генерируются исполняемые файлы, позволяющие запускать визуализатор без доступа к сети.

11. *Удобство создания визуализаторов*. Разработан пошаговый процесс построения визуализатора с применением системы визуализации *Vizi*.
12. *Скорость разработки визуализаторов.* Скорость построения визуализаторов существенно увеличивается за счет автоматизации некоторых этапов.

Таким образом, система визуализации *Vizi* позволяет выполнить все рассмотренные в разделе 1.1.2 требования.

6.4. Практическое использование результатов работы

Система визуализации *Vizi* (как и ее предшественник *BaseApplet*) была использована в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО. В течение ряда лет студенты первого курса выполняли визуализаторы алгоритмов, которые используются для преподавания дискретной математики и будут выложены на сайте Интернет-школы программирования.

Визуализаторы, созданные на основе системы *Vizi*, были внедрены в учебном процессе в Лицее «Физико-техническая школа» (Санкт-Петербург) и Специализированном учебно-научном центре МГУ (Москва).

Выводы по главе 6

1. Разработана система визуализации *Vizi*.
2. Выполнено сравнение с предшествующей системой построения визуализаторов, которое показало, что система визуализации *Vizi* позволяет сократить объем кода визуализатора.
3. Показано, что система визуализации *Vizi* позволяет выполнить все требования к визуализаторам, сформулированные в главе 1.
4. Система визуализации *Vizi* успешно внедрена в учебный процесс.

ЗАКЛЮЧЕНИЕ

В настоящей работе разработаны методы преобразования программ (в том числе и рекурсивных) в систему взаимодействующих конечных автоматов. При этом полученная система автоматов позволяет эмулировать выполнение программы как вперед, так и назад. Ранее системы автоматов позволяли двигаться по программе только в одном направлении.

Для описания логики визуализаторов разработан язык описания визуализаторов. В рамках этого формата удобно описывается как логика визуализатора (при этом описание повторяет структуру визуализируемого алгоритма), 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 предложенных методов и языка разработана технология автоматизированной разработки визуализаторов. Анализ зарубежных и отечественных результатов в области построения визуализаторов позволяет утверждать, что разработанная технология является новой и отличается формализованностью процесса построения логики визуализатора. Это позволяет строить на базе предложенной технологии системы визуализации на новом уровне.

На основе разработанной технологии была построена система визуализации *Vizi*, позволившая существенно упростить создание визуализаторов сложных алгоритмов и сократить сроки их разработки. Система визуализации *Vizi* напрямую поддерживает разработанный язык описания визуализатора и позволяет по описанию автоматически генерировать исходные коды программы визуализатора, а также 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 *Vizi* была внедрена на кафедре «Компьютерные технологии» СПбГУ ИТМО (2003−2006 гг.). Визуализаторы, созданные на ее основе внедрены в учебном процессе в Лицее «Физико-техническая школа» (Санкт-Петербург) и специализированном учебно-научном центре МГУ (Москва).

При этом с ее использование студентами первых курсов были выполнены визуализаторы таких сложных алгоритмов, как, например, поиск максимального потока в сети методами Диница и Малхотры-Кумара-Махешвари. Создание визуализаторов таких алгоритмов ранее было сопряжено с большими трудностями, которые устранены системой визуализации *Vizi*.

Таким образом, результаты, достигнутые в данной работе, существенно изменяют процесс построения визуализаторов. Формализация процесса построения визуализаторов позволяет использовать менее квалифицированные кадры, в частности, студентов, что полезно при изучении сложных алгоритмов, применяемых в САПР приборов и устройств.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Печатные издания на русском языке

1. Автоматы / Ред. Шеннона К.Э., МакКарти Дж. М.: Изд-во иностр. лит., 1956.
2. *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии и инструменты. М.: Вильямс. 2001.
3. *Буч Г., Якобсон А., Рамбо Дж.* UML. 2-е издание. СПб.: Питер, 2006, 736 с.
4. *Баранов С. И.* Синтез микропрограммных автоматов (граф-схемы и автоматы). Л.: Энергия, 1979.
5. *Батищев Д. И., Львович Я. Е., Фролов В. Н.* Оптимизация в САПР. Воронеж: Воронежский государственный университет, 1997.
6. *Гаврилов М. А.* Теория релейно-контактных схем. М.: Изд-во АН СССР, 1950.
7. *Глушков В. М.* Синтез цифровых автоматов. М.: Изд-во физ.-мат. лит., 1962.
8. *Грис Д.* Наука программирования. М.: Мир, 1984.
9. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. М.: Мир, 1975.
10. *Казаков М. А., Мельничук О. П., Парфенов В. Г.* Интернет-школа программирования в СПбГИТМО. Реализация и внедрение / Труды международной научно-методическая конференции «Телематика-2002». СПб.: СПбГИТМО (ТУ), 2002.
11. *Казаков М. А., Столяр С. Е.* Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования / Труды международной научно-методическая конференции «Телематика-2000» . СПб.: 2000.

12. *Казаков М. А., Шалыто А. А.* Использование автоматного программирования для реализации визуализаторов // Компьютерные инструменты в образовании. 2004. № 2.
13. *Казаков М. А., Шалыто А. А.* Реализация анимации при построении визуализаторов алгоритмов на основе автоматного подхода // Информационно-управляющие системы. 2005. № 4.
14. *Казаков М. А., Шалыто А. А.* Автоматный подход к реализации анимации в визуализаторах алгоритмов // Компьютерные инструменты в образовании. 2005. № 3.
15. *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.
16. *Клини С.* Представление событий в нервных сетях и конечных автоматах / В [1].
17. *Кнут Д.* Искусство программирования. Том 1. Основные алгоритмы. М.: Вильямс, 2000.
18. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦНМО, 1999.
19. *Корнеев Г А., Шалыто А. А., Шамгунов Н. Н.* Паттерн State Machine для объектно-ориентированного проектирования автоматов // Информационно-управляющие системы. 2004. № 5.
20. *Ли К.* Основы САПР (CAD/CAM/CAE). СПб.: Питер, 2004.
21. *Лингер Р., Миллс Х., Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982.
22. *Мелихов А. Н., Бернштейн Л. С., Курейчик В. М.* Применение графов для проектирования дискретных устройств. М.: Наука, 1974.
23. *Мур Э.* Умозрительные эксперименты с последовательными машинами / В [1].

24. *Рабин М.О., Скотт Д.* Конечные автоматы и задачи их разрешения // Кибернетический сборник. Вып. 4. М.: Изд-во иностр. лит., 1962.
25. *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.
26. *Туккель Н. И., Шалыто А. А., Шамгунов Н. Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2002. № 5.
27. *Фокс А., Пратт М.* Вычислительная геометрия: применение в проектировании и на производстве. М.: Мир, 1982.
28. *Хопкрофт Д., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2001.
29. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
30. *Шалыто А.А., Туккель Н. И.* Преобразование итеративных алгоритмов автоматные //Программирование. 2002. № 5.
31. *Шалыто А.А. Туккель Н.И.* От тьюрингова программирования к автоматному // Мир ПК. 2002. № 2.

Печатные издания на английском языке

32. *Baecker R.* Sorting out Sorting / SIGGRAPH 1981, Dallas, Texas.
33. *Baker J.E., Cruz I. F., Liotta G., Tamassia R.* The Mocha Algorithm Animation System / Proceedings of International Workshop on Advanced Visual Interfaces. 1996.
34. *Brown M.* Algorithm Animation. MIT Press, Cambridge, Massachussets, 1988.
35. *Brown M.* Zeus: a System for Algorithm Animation / Proceedings of 1991 IEEE Workshop on Visual Langauges. 1991.
36. *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.

37. *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, Academic Press, April 2000.
38. *Demetrescu C., Finocchi I.* Smooth animation of algorithms in a declarative framework // Journal of Visual Languages and Computing (JVLC). Volume 9. Issue 3. Special Issue devoted to selected papers from the 15th IEEE Symposium on Visual Languages. Academic Press, 2001.
39. *Demetrescu C., Finocchi I.* A technique for generating graphical abstractions of program data structures / Proceedings of the 3rd International Conference on Visual Information Systems. Amsterdam, 1999.
40. *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.
41. *Dijkstra E.W.* Go To Statement Considered Harmful // Communications of the ACM. Volume 11. No. 3, March 1968.
42. *Huffman D.A.* The Synthesis of Sequential Switching Circuits // J. Franclin Inst. 1954. V.257, № 3, 4.
43. *Italiano G., Cattaneo G., Ferraro U., Scarano V.* CATAI: Concurrent Algorithms and Data Types Animation over the Internet / Proceedings of 15th IFIP World Computer Congress. Vienna, August - September 1998.
44. *Joy B., Steele G., Gosling J., Bracha G.* Java Language Specification (Second Edition). Addison-Wesley. 2000.
45. *Kane G.* MIPS RISC architecture. Prentice-Hall, N.J., 1988.
46. *Knowlton K.* L6: Bell Telephone Laboratories Low-Level Linked List Language / 16-minute black-and-white film. Murray Hill, N.J., 1966.
47. *Knowlton K.* L6: Part II. An Example of L6 Programming / 30-minute black-and-white film, Murray Hill, N.J., 1966.

48. *Lahtinen S., Sutinen E., Tarhio J.* Automated Animation of Algorithms with Eliot // Journal of Visual Languages and Computing. 9 (3). 1998.
49. *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.
50. *McCulloch W., Pitts W.* A Logical Calculus of Ideas Immanent in Nervous Activity // Bull. Math. Biophysics. 1943, 5.
51. *Mealy G.* A Method for Synthesizing Sequential Circuits // Bell System Technical Journal. 1955. V.34. № 5.
52. *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.
53. *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.
54. *Naps T., Röβling 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. ACM Press, New York, 2003.
55. *Pierson W., Rodger S. H.* Web-based Animation of Data Structures Using JAWAA / Twenty-ninth SIGCSE Technical Symposium on Computer Science Education. 1998.
56. *Programming Language FORTRAN.* American National Standards Institute (ANSI). 1978.
57. *Programming Langauges* — C. ISO/IEC 9899:1999. 1999.
58. *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.

59. *Rodger S. H.* Introducing Computer Science Through Animation and Virtual Worlds / Thirty-third SIGCSE Technical Symposium on Computer Science Education. 2002.
60. *Rößling G., Freisleben B.* ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation // Journal of Visual Languages and Computing, Volume 13 issue 3. Elsevier, Amsterdam, Netherlands, 2002.
61. *Ruknet C.* The Design of the Processor Architecture Capable of forward and Reverse Execution / Procedings of IEEE SoutheastCon 91. V.2, 1991.
62. *Stasko J.* Tango: A Framework and System for Algorithm Animation // IEEE Computer. 23, 1990.
63. *Stasko J.* Animating Algorithms with XTANGO // SIGACT News, volume 23, issue 2, Spring 1992.
64. *Stasko J.* Tango: A Framework and System for Algorithm Animation // IEEE Computer, September 1990.
65. *Stasko J.* A methodology for building Application-Specific Visualizations of Parallel Programs // Journal of Parallel and Distributed Computing. 1993, № 18.
66. *Thompson K.* Regular expression Search Algorithm //Communications of the ACM. 1966. V.11. № 6.
67. *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.
68. *Zelkowitz M. V.* Reversible Execution // Communication of the ACM. 1973. V. 16, № 9.

Ресурсы сети Internet

69. Визуализатор пирамидальной сортировки / http://is.ifmo.ru/works/heapsort/
70. Кафедра «Технологии программирования» СПбГУ ИТМО / http://is.ifmo.ru.

71. Agat: Another Graphical Animation Tool / http://www-sop.inria.fr/cafe/Olivier.Arsac/agat/agat.html.
72. Animal Home Page / http://www.animal.ahrgr.de.
73. Apache Ant Home Page / http://ant.apache.org/.
74. *Arsac O., Lavirotte S.* The Agat Manual / http://www-sop.inria.fr/cafe/Olivier.Arsac/agat/Doc/refman.ps.gz.
75. Code conventions for Java programming language. / http://Java.sun.com/docs/codeconv/.
76. Дискретная математика: алгоритмы / http://rain.ifmo.ru/cat.
77. Extensible Markup Language (XML) Version 1.0 (Second Edition) / http://www.w3.org/TR/2000/REC-xml-20001006/.
78. Font class description (Java 2 Platform SE 1.4.2) / http://Java.sun.com/j2se/api/Java/awt/Font.html.
79. Hope College animator page / http://www.cs.hope.edu/~alganim/animator/Animator.html.
80. Jeliot Home Page / http://cs.joensuu.fi/jeliot/.
81. Leonardo home page / http://www.dis.uniroma1.it/~demetres/Leonardo/Leonardo.html.
82. MessageFormat class description (Java 2 Platform SE 1.4.2) / http://Java.sun.com/j2se/api/Java/text/MessageFormat.html.
83. Polka Animation System / http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html.
84. Princeton University animators collection / http://www.cs.princeton.edu/~ah/alg_anim/version1/Animator.html.
85. Properties class description (Java 2 Platform SE 1.4.2) / http://Java.sun.com/j2se/api/Java/util/Properties.html.
86. Samba algorithm animation system / http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html.
87. State University of New York College at Brockport sorting animators collection / http://www.cs.brockport.edu/cs/Javasort.html.

88. Vizi home page / http://ctddev.ifmo.ru/vizi
89. XML Schema Part 0: Primer / http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.
90. XML Schema Part 1: Structures / http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.
91. XML Schema Part 2: Data Types / http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.
92. XSL Transformations (XSLT) Version 1.0 / http://www.w3.org/TR/1999/REC-xslt-19991116

Публикации

93. *Корнеев Г.А., Шалыто А.А.* Преобразование программ в систему взаимодействующих конечных автоматов / Труды Второй Всероссийской Научной конференции «Методы и средства обработки информации». М.: МГУ. 2005, с. 385-387.
94. *Корнеев Г. А.* Метод преобразования программ в систему взаимодействующих автоматов / Труды II межвузовской конференции молодых учёных. СПб.: СПбГУ ИТМО. 2005, с. 65-72.
95. *Корнеев Г. А.* Технология разработки визуализаторов алгоритмов / Труды II межвузовской конференции молодых учёных. СПб.: СПбГУ ИТМО, 2005, с. 18-23.
96. *Казаков М.А., Корнеев Г.А., Шалыто А.А.* Разработка логики визуализаторов алгоритмов на основе конечных автоматов // Телекоммуникации и информатизация образования. 2003. № 6, с. 27-58.
97. *Корнеев Г.А., Васильев В.Н., Парфенов В.Г., Столяр С.Е.* Визуализаторы алгоритмов как основной инструмент технологии преподавания дискретной математики и программирования / Труды международной научно-методической конференции «Телематика-2001». СПб.: СПбГИТМО (ТУ). 2001, с. 119, 120.

98. *Корнеев Г.А., Шалыто А.А.* Реализация конечных автоматов с использованием объектно-ориентированного программирования / Труды X международной научно-методической конференции «Телематика-2003». СПб.: СПбГИТМО (ТУ). 2003, с. 377, 378.
99. *Корнеев Г.А., Казаков М.А., Шалыто А.А.* Построение логики работы визуализаторов алгоритмов на основе автоматного подхода / Труды X международной научно-методической конференции «Телематика-2003». СПб.: СПбГИТМО (ТУ). 2003, с. 378, 379.
100. *Корнеев Г.А., Шамгунов Н.Н., Шалыто А.А.* Обход деревьев на основе автоматного подхода // Компьютерные инструменты в образовании. 2004. № 3, с. 32-37.
101. *Корнеев Г.А.* Преобразование программы в систему взаимодействующих автоматов, допускающих двустороннюю трассировку / Материалы политехнического симпозиума 2005 «Молодые ученые — промышленности Северо-Западного региона». СПб.: Санкт-Петербургский государственный политехнический университет. 2005, с. 33, 34.
102. *Корнеев Г.А., Шалыто А.А.* Построение визуализаторов алгоритмов дискретной математики // Научно-технический вестник СПбГУ ИТМО. Выпуск 23. Высокие технологии в оптических и информационных системах. СПб.: СПбГУ ИТМО. 2005, с. 118-129.
103. *Корнеев Г.А., Шалыто А.А.* VIZI — язык описания логики визуализаторов алгоритмов // Научно-технический вестник СПбГУ ИТМО. Выпуск 23. Высокие технологии в оптических и информационных системах. СПб.: СПбГУ ИТМО. 2005, с. 130-138.

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. Исходный код визуализатора поиска максимума

```
package ru.ifmo.vizi.find_max;

import ru.ifmo.vizi.base.ui.*;
import ru.ifmo.vizi.base.*;
import ru.ifmo.vizi.base.widgets.Rect;
import ru.ifmo.vizi.base.widgets.ShapeStyle;

import Java.awt.*;
import Java.awt.event.AdjustmentEvent;
import Java.awt.event.AdjustmentListener;
import Java.util.Stack;
```

```
/**
 * Find maximum applet.
 * @author  Georgiy Korneev
 */
public final class FindMaximumVisualizer
  extends Base
  implements AdjustmentListener
{
  /** Find maximum automata instance. */
  private final FindMaximum auto;

  /** Find maximum automata data. */
  private final FindMaximum.Data data;

  /** Cells with array elements. Vector of {@link Rect}. */
  private final Stack cells;

  /** Number of elements in array. */
  private final AdjustablePanel elements;

  /** Maximal array value. */
  private final int maxValue;

  /** Maximal array value string. */
  private final String maxValueString;

  /** Rectangle that contains maximum value. */
  private final Rect rectMax;

  /** Max message tamplate. */
  private final String maxMessage;

  /** Array shape style set. */
  private final ShapeStyle[] styleSet;

  /** Save/load dialog. */
  private SaveLoadDialog saveLoadDialog;

  /**
   * Creates a new Find Maximum visualizer.
   *
   * @param parameters visualizer parameters.
   */
  public FindMaximumVisualizer(VisualizerParameters parameters) {
    super(parameters);
    auto = new FindMaximum(locale);
    data = auto.d;
    data.visualizer = this;
    cells = new Stack();

    maxMessage = config.getParameter("max-message");
```

```
  styleSet = ShapeStyle.loadStyleSet(config, "array");
  rectMax = new Rect(
      new ShapeStyle[]{new ShapeStyle(config,
         "max-style", styleSet[0])},
      "max"
  );
  clientPane.add(rectMax);
  rectMax.adjustSize();
  rectMax.setLocation(10, 10);

  elements = new AdjustablePanel(config, "elements");
  elements.addAdjustmentListener(this);

  maxValue = config.getInteger("max-value");
  maxValueString = config.getParameter("max-value-string",
          Integer.toString(maxValue));
  setArraySize(elements.getValue());
  randomize();

  createInterface(auto);
}

/**
 * This method creates panel with visualizer controls.
 *
 * @return controls pane.
 */
public Component createControlsPane() {
  Container panel = new Panel(new BorderLayout());
  panel.add(new AutoControlsPane(config, auto, forefather,
          false), BorderLayout.CENTER);

  Panel bottomPanel = new Panel();
  bottomPanel.add(new HintedButton(config, "button-random"){
    protected void click() {
      randomize();
    }
  });
  if (config.getBoolean("button-ShowSaveLoad")) {
    bottomPanel.add(new HintedButton(config, "button-SaveLoad") {
      protected void click() {
        saveLoadDialog.center();
        StringBuffer buffer = new StringBuffer();
        int[] a = auto.d.a;
        buffer.append("/* ").append(
          I18n.message(
            config.getParameter("ArrayLengthComment"),
            new Integer(elements.getMinimum()),
            new Integer(elements.getMaximum())
          )
        ).append(" */\n");
```

```
      buffer.append("ArrayLength = ")
        .append(a.length).append("\n");

      buffer.append("/* ").append(
        I18n.message(
          config.getParameter("ElementsComment"),
          new Integer(0),
          new Integer(maxValue)
        )
      ).append(" */\n");

      buffer.append("Elements = ");
      for (int i = 0; i < a.length; i++) {
        buffer.append(a[i]).append(" ");
      }

      buffer.append("\n/* ").append(
        config.getParameter("StepComment")
      ).append(" */\n");
      buffer.append("Step = ").append(auto.getStep());
      saveLoadDialog.show(buffer.toString());
    }
  });
}
bottomPanel.add(elements);
panel.add(bottomPanel, BorderLayout.SOUTH);

saveLoadDialog = new SaveLoadDialog(config, forefather) {
  public boolean load(String text) throws Exception {
    SmartTokenizer tokenizer =
      new SmartTokenizer(text, config);
    tokenizer.expect("ArrayLength");
    tokenizer.expect("=");
    int[] a = new int[tokenizer.nextInt(
      elements.getMinimum(),
      elements.getMaximum()
    )];

    tokenizer.expect("Elements");
    tokenizer.expect("=");
    for (int i = 0; i < a.length; i++) {
      a[i] = tokenizer.nextInt(0, maxValue);
    }

    tokenizer.expect("Step");
    tokenizer.expect("=");
    int step = tokenizer.nextInt();
    tokenizer.expectEOF();

    setArraySize(a.length);
    auto.d.a = a;
    auto.getController().rewind(step);
```

```
      return true;
    }
  };

  return panel;
}

/**
 * Adjusts array size to match current model size.
 */
private void adjustArraySize() {
  int size = auto.d.a.length;
  while (cells.size() < size) {
    Rect rect = new Rect(styleSet);
    cells.push(rect);
    clientPane.add(rect);
  }
  while (cells.size() > size) {
    clientPane.remove((Component) cells.pop());
  }
  clientPane.doLayout();
}

/**
 * Sets new array size.
 *
 * @param size new array size.
 */
private void setArraySize(int size) {
  auto.d.a = new int[size];
  elements.setValue(data.a.length);
  adjustArraySize();
}

/**
 * Randomizes array values.
 */
private void randomize() {
  for (int i = 0; i < data.a.length; i++) {
    data.a[i] = (int) (Math.random() * maxValue) + 1;
  }
  auto.getController().doRestart();
}

/**
 * Invoked on adjustment event.
 *
 * @param event event to process.
 */
public void adjustmentValueChanged(AdjustmentEvent event) {
  if (event.getSource() == elements) {
    setArraySize(event.getValue());
    randomize();
```

```
    }
  }

  /**
   * Updates array view.
   *
   * @param activeCell current active cell.
   * @param activeStyle style of active cell.
   */
  public void updateArray(int activeCell, int activeStyle) {
    rectMax.setMessage(I18n.message(maxMessage, new
            Integer(data.max)));
    rectMax.adjustSize();

    for (int i = 0; i < data.a.length; i++) {
      Rect rect = (Rect) cells.elementAt(i);
      rect.setMessage(Integer.toString(data.a[i]));
      rect.setStyle(i == activeCell ? activeStyle : 0);
    }
    update(true);
  }

  /**
   * Invoked when client pane shoud be layouted.
   *
   * @param clientWidth client pane width.
   * @param clientHeight client pane height.
   */
  protected void layoutClientPane(int clientWidth,
      int clientHeight
  ) {
    int n = cells.size();

    Rectangle mb = rectMax.getBounds();

    int width = Math.round(clientWidth / (n + 1));
    int height = Math.min(width, (clientHeight - mb.x - mb.height)
            * 10 / 13);
    int y = mb.x + mb.height + height / 10;
    int x = (clientWidth - width * n) / 2;

    for (int i = 0; i < n; i++) {
      Rect rect = (Rect) cells.elementAt(i);
      rect.setBounds(x + i * width, y, width+1, height+1);
      rect.adjustFontSize(maxValueString);
    }
  }
}
```