Санкт-Петербургский государственный университет информационных технологий, механики и оптики

На правах рукописи

Гуров Вадим Сергеевич

Технология проектирования и разработки объектно-ориентированных программ с явным выделением состояний (метод, инструментальное средство, верификация)

Специальность 05.13.11. Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

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

Санкт-Петербург – 2008

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность проблемы. Современные программные системы, которые во многих случаях создаются с помощью объектно-ориентированных подходов, являются сложными. Для борьбы с этой сложностью непрерывно разрабатываются все новые средства, позволяющие увеличивать уровень абстракции и упрощать процесс программирования и проверки.

При создании программных систем обычно выделяют следующие фазы:

1. Постановка задачи – сбор требований и создание прототипа программы.
2. Проектирование – разработка проектной документации, отражающей структурные и поведенческие особенности создаваемой системы.
3. Реализация – создание на основе проекта кода для целевой программно-аппаратной платформы.
4. Тестирование – отладка кода и проверка соответствия реализации поставленной задаче.

Семантический разрыв при передаче знаний между проектированием и реализацией заключается в том, что разработчик обычно реализует систему в соответствии со своим пониманием проектной документации. Это приводит к ряду проблем:

1. Реализация системы не соответствует проектной документации ввиду неформальной связи фаз проектирования и реализации.
2. Проверка соответствия реализации проектной документации (верификация) может быть выполнена только вручную.
3. В случае необходимости изменений в системе, они вносятся в проектную документацию и в код программы независимо, что часто приводит к рассинхронизации документации и кода.

Причина указанных проблем кроется в том, что существуют методы проектирования объектно-ориентированных программ, которые позволяют моделировать их структуру, а также методы, позволяющие моделировать их поведение, но отсутствуют методы, которые обеспечивают связь статики и динамики в единую формальную модель.

Исследования, направленные на разработку таких методов и технологий для их поддержки, являются актуальными, так как позволят упростить процесс разработки и повысить качество создаваемых программ.

В настоящее время развивается автоматный подход к созданию программ, называемый автоматное программирование или программирование с явным выделением состояний, который обеспечивает возможность разработки указанных технологий, основанных, в том числе, и на графических языках программирования.

Цель диссертационной работы – разработка технологии проектирования и реализация объектно-ориентированных программ с явным выделением состояний.

Основные задачи исследования:

1. Создание метода проектирования объектно-ориентированных программ на основе автоматного подхода.
2. Разработка графического языка автоматного программирования.
3. Разработка методов верификации автоматных моделей программ.
4. Разработка инструментального средства для поддержки автоматного программирования.
5. Внедрение результатов работы в практику промышленной разработки программного обеспечения и в учебный процесс кафедры «Компьютерные технологии» СПбГУ ИТМО.

Научная новизна. На защиту выносятся следующие результаты, обладающие научной новизной:

1. Метод проектирования объектно-ориентированных программ с явным выделением состояний.

2. Графический язык для описания автоматных программ на основе *UML*-нотации.
3. Методы верификации автоматных моделей программ: метод верификации на модели (*Model Checking*), а также метод верификации полноты и непротиворечивости систем переходов автоматов.
4. Инструментальное средство для создания, верификации, отладки и запуска автоматных программ. При этом верификация на основе модели производится совместно с верификатором *Bogor*.

Перечисленные результаты получены в ходе выполнения в СПбГУ ИТМО научно-исследовательских и опытно-конструкторских работ по темам: «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода» (проводится по заказу Минобрнауки РФ с 2000 г. по настоящее время), «Разработка технологии автоматного программирования» (проводилась в 2002–2003 гг. по гранту Российского фонда фундаментальных исследований № 02-07-90114), «Разработка технологии объектно-ориентированного программирования с явным выделением состояний» (проводилась в 2005–2006 гг. по гранту Российского фонда фундаментальных исследований № 05-07-90011), «Технология автоматного программирования: применение и инструментальные средства» (государственный контракт, который выполнялся в 2005–2006 гг. в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития науки и техники»). Последняя работа вошла в список 15 наиболее перспективных проектов, выполняемых по этой программе.

Методы исследования. В работе использованы методы объектно-ориентированного проектирования, теории автоматов, теории формальных грамматик, теории графов, теории алгоритмов, теории верификации.

Достоверность научных положений и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием

постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами внедрения предложенной технологии.

Практическое значение полученных результатов состоит в том, что они успешно используются при разработке промышленных и учебных программных проектов на основе автоматного подхода.

Предложенные методы позволили устранить семантический разрыв между фазами проектирования и реализации за счет интерпретации автоматной модели программы или генерации изоморфного кода на целевом языке программирования.

За счет того, что автоматная модель системы является одновременно спецификацией и программой, процесс поддержания проектной документации в актуальном состоянии также значительно упростился.

Упрощается верификация на основе метода *Model Checking,* так как в автоматных моделях состояния явно выделены, и поэтому пространство состояний по сравнению с программами, построенными традиционным образом, резко сокращается.

Разработанная технология позволяет выявлять логические ошибки в автоматных программах на стадии проектирования, что уменьшает время разработки и время тестирования, и, как следствие, повышает качество программных продуктов.

Внедрение результатов работы. Результаты, полученные в диссертации, используются на практике в компании *eVelopers* (Санкт-Петербург) при разработке интернет-приложений для электронной коммерции и мобильных устройств, а также в компании *Intellij Labs* (Санкт-Петербург) при разработке мета-программирования *Meta Programming System*.

Полученные результаты используются также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при выполнении курсовых работ по курсу «Теория автоматов в программировании». При этом

на сайте http://is.ifmo.ru в разделе *UniMod*-проекты опубликовано 28 проектов, выполненных с помощью предлагаемой технологии, которые содержат, в том числе, и проектную документацию.

Апробация диссертации. Основные положения диссертационной работы докладывались на конференциях и семинарах: II конференции молодых ученых СПбГУ ИТМО (2005 г.); XXXIII, XXXV, XXXVI научных учебно-методических конференциях СПбГУ ИТМО «Достижения ученых, аспирантов и студентов СПбГУ ИТМО в науке и образовании» (2003, 2005, 2006 гг.); «Телематика-2003», «Телематика-2004», «Телематика-2005», «Телематика-2006», «Телематика-2007» (СПб.); на семинаре «Автоматное программирование» в рамках международной конференции «International Computer Symposium in Russia (CSR 2006)» (ПОМИ им. Стеклова, 2006 г.); на конференциях «Software Engineering Conference in Russia» – SECR 2005 (Москва), «The International Scientific Conference «110-Anniversary of Radio Invention» (СПбГЭТУ, IEEE, 2005 г.); Второй Всероссийской научной конференции «Методы и средства обработки информации» (МГУ, 2005 г.); Open Source Forum (М.: Форт-Росс, 2005 г.); международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы» МВУС-2007 (Таганрог, 2007 г.); научно-технической конференции «Научно-программное обеспечение в образовании и научных исследованиях» (СПб., 2008 г.).

Публикации. По теме диссертации опубликовано 23 печатные работы, в том числе в журналах из списка ВАК «Программирование», «Информационно-управляющие системы» и «Научно-технический вестник СПбГУ ИТМО», а также в журнале «Технология клиент-сервер» и материалах указанных конференций и семинаров.

Свидетельства об официальной регистрации программ для ЭВМ. На инструментальное средство, разработанное в рамках диссертации, получены свидетельства: «Ядро автоматного программирования»

№2006613249 от 14.09.2006, «Встраиваемый модуль автоматного программирования для среды разработки Eclipse» №2006613817 от 7.11.2006.

Структура диссертации. Диссертация изложена на 152 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит 114 наименований. Работа иллюстрирована 59 рисунками и содержит три таблицы.

В первой главе приведен обзор существующих методов создания программ на основе автоматного подхода и введена их классификация.

Во второй главе описан предлагаемый метод создания объектно-ориентированных программ на основе автоматного подхода, описан исполняемый графический язык программирования, его синтаксис и операционная семантика.

В третьей главе рассматриваются методы верификации автоматных программ на основе дедуктивного анализа и метода верификации на модели.

В четвертой главе описано инструментальное средство для поддержки разработанного метода проектирования, графического языка и методов верификации.

Пятая глава содержит описание результатов внедрения предложенных методов и инструментального средства. В этой главе также описан метод создания синтаксических анализаторов с эффективным восстановлением после ошибок.

ГЛАВА 1. ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ И РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

В последнее время для повышения уровня абстракции средств разработки программ развивается направление программной инженерии (*Software Engineering*) [1], которое называется «Инженерия, управляемая моделями» (*Model-Driven Engineering, MDE*) [9].

Это направление включает в себя «Разработку, управляемую моделями» (*Model-Driven Development, MDD*), которое может быть названо также «Проектирование на базе моделей» (*Model-Driven Design*) [10, 11]. Вариантом *MDD* является «Архитектура, управляемая моделями» (*Model-Driven Architecture, MDA*) [12, 13], предложенная и развиваемая консорциумом *Object Management Group* (*OMG*).

При применении *MDA* модели программных систем представляются с помощью «Унифицированного языка моделирования» (*Unified Modeling Language, UML*) [14].

Если в течение ряда лет этот язык использовался только для представления моделей, то в последнее время все большую популярность приобретает идея *исполняемого UML* [15, 16]. Это связано с тем, что практическое использование *UML* в большинстве случаев ограничивается моделированием только статической части программ с помощью диаграмм классов и генерацией по ним каркаса кода программы. Этого недостаточно для полноценного проектирования программ.

Моделирование динамических аспектов программ на языке *UML* затруднено в связи с отсутствием в стандарте на этот язык формального и однозначного описания правил интерпретации (операционной семантики) поведенческих диаграмм.

Кроме того, ни в одном из большого числа методов проектирования объектно-ориентированных систем, описанных в работе [1], «внятно» не сказано, как связывать статические диаграммы с динамическими.

Несмотря на наличие большого числа инструментальных средств для автоматического преобразования поведенческих диаграмм (диаграмм состояний) в код на различных языках программирования [17], в широко известных средствах моделирования, например *Sun Studio Enterprise* [18], такая функциональность отсутствует.

В некоторых инструментальных средствах графические редакторы для построения указанных диаграмм имеются, но кодогенерация по ним отсутствует.

Отметим две тенденции, активно развивающиеся в настоящее время [19]:

1. Исполняемый *UML*. На текущий момент *UML* применяется, в основном, как язык спецификации моделей систем. Существующие *UML*-средства позволяют строить различные диаграммы и автоматически создавать по диаграмме классов «скелет» кода на целевом языке программирования (например, языки *Java* и *C#*). Некоторые их этих средств также предоставляют возможность автоматически генерировать код поведения программы по диаграммам состояний.

   Однако в настоящее время указанная функциональность существует лишь в «зачаточном состоянии», так как известные инструменты не позволяют в полной мере эффективно связывать генерируемый код с моделью поведения, которую можно описывать с помощью четырех типов диаграмм (состояний, деятельностей, кооперации или последовательностей).

Отсутствие однозначной операционной семантики при традиционном написании программ приводит к различию описания поведения в модели и в программе, а также к произвольной интерпретации поведенческих диаграмм программистами. Более того, описание поведения в модели часто носит неформальный характер. Возможна и противоположная ситуация, когда строится формальная модель, а ее реализация выполняется эвристически. Часто формальная модель поведения строится архитектором, а программист при написании программы ее не использует, а пишет исходный текст программы, как считает нужным.

Появление операционной семантики зафиксирует однозначность понимания диаграмм и позволит создать исполняемый *UML*, для которого код (в привычном смысле этого слова) может не генерироваться. Это возможно за счет непосредственной интерпретации модели.

2. Процесс разработки ПО должен быть активным. Существующие средства разработки требуют длительного времени для их изучения. Поэтому они должны предсказывать действия разработчика и предлагать варианты решения возникших проблем в зависимости от текущего контекста. Отметим, что подобный подход реализован во многих современных средах разработки (например, *Borland JBuilder*, *Eclipse*, *IntelliJ IDEA*) для текстовых языков программирования, но не для языка *UML*.

1.1. Реактивные системы

Широким классом программных систем являются реактивные системы – системы, выполняющие определенные действия в ответ на внешние события. В работах Д. Харела [6–8] показано, что для

моделирования таких систем хорошо подходит расширение графов переходов конечных автоматов, названное «диаграммы состояний» (*Statechart)*, которые позволяют удобно и компактно описать реакцию системы на события.

Для построения диаграмм состояний и генерации кода по ним созданы инструментальные средства, многие из которых перечислены в [17]. В этой работе, в частности, упомянуты такие инструменты как *I-Logix Statemate (*http://ilogix.com/sublevel.aspx?id=74), *XJTek AnyState (*http://www.xjtek.com/anystates/), *StateSoft ViewControl* (http://www.statesoft.ie/products.html), *SCOPE* (http://www.itu.dk/~wasowski/projects/scope/), *IAR Systems visualSTATE (*http://www.iar.com/p1014/p1014_eng.php), *The State Machine Compiler (*http://smc.sourceforge.net/) [20–25].

Существуют также и другие инструменты для генерации кода по этим диаграммам, например, описанное в работе [26].

Недостаток этих инструментов состоит в том, что они позволяют строить и реализовать только поведенческую часть модели программы, не рассматривая их статику. Поэтому с помощью этих инструменты *программу в целом не построить.*

1.2. Классификация автоматных подходов

Указанные в предыдущем разделе инструменты для автоматно-ориентированной разработки программ, могут быть классифицированы по следующим признакам:

- целевой класс автоматных моделей:
  - имитационное моделирование для исследования свойств физических объектов;
  - приложения для встроенных систем (логические контроллеры и микроконтроллеры);

- приложения для мобильных устройств (например, для сотовых телефонов);
- прикладные программы (компиляторы, игры, системы автоматизации бизнес-процессов);

- способ задания и реализация автомата:
  - граф переходов реализуется с помощью оператора *Switch* или паттернов *State* [27] и *State Machine* [28] на целевом языке программирования. После этого код компилируется и запускается;
  - таблица переходов записывается в структурированный файл (например, в формате *XML*), который обрабатывается интерпретатором;
  - граф переходов задается с помощью визуальных средств моделирования в виде *UML*-диаграммы состояний или с использованием собственной графической нотации. Диаграммы реализуются либо путем их преобразования в код, который в дальнейшем компилируется, либо в файл для последующей интерпретации;
- способ получения кода из графически представленных графов переходов:
  - вручную;
  - автоматически;
- способ проверки корректности (верификации) графа переходов:
  - вручную;
  - автоматически;
- способ документирования поведенческой модели системы:
  - вручную;
  - автоматически;

- способ отладки автоматной программы:
  - в терминах целевого языка программирования;
  - в терминах автоматов;

Выполненный далее обзор основан на предложенной классификации.

1.3. Гибридные автоматы

В последние годы поведение сложных физических систем стали описывать с помощью гибридных автоматов [29, 30]. При этом в системах выделяются состояния, для каждого из которых характерно некоторое непрерывное поведение. При наступлении события или выполнении некоторого условия система скачкообразно изменяет свое поведение и. переходит из одного состояния в другое. Автоматы, описывающие такое поведение, названы гибридными, так как сочетают в себе особенности дискретных и непрерывных систем.

В работе [31] описывается подход к имитационному моделированию физических систем с использованием гибридных автоматов на основе программного продукта *Model Vision Studium* [32]. Этот продукт используется также в качестве составной части другого программного продукта – *AnyLogic* [33].

Идея гибридных автоматов может быть использована и при программирования игр [34].

Еще один подход к описанию поведения и программированию сложных систем состоит *в разделении состояний на управляющие и вычислительные* [35]. При этом число состояний управляющего автомата обычно не велико, но он может управлять сколь угодно большим числом вычислительных состояний. Так, например, в известной задаче о ханойских башнях число состояний, которые могут принимать n дисков равно 2^n, а число управляющих состояний равно всего лишь трем [35]. Идея этого подхода основана на конструкции машины Тьюринга, в которой конечный автомат управляет бесконечной лентой [36].

1.4. Автоматное программирование встраиваемых систем

Конечные автоматы до последнего времени в основном использовались при аппаратных реализациях алгоритмов [37].

С появлением программируемых логических матриц, выполненных в виде сверх больших интегральных схем, появились инструментальные средства, позволяющие программировать эти матрицы, используя в качестве спецификации алгоритмов графы переходов [38, 39]. Эти средства обладают графическими редакторами графов переходов автоматов, реализуют алгоритмы проверки корректности построенных моделей, умеют верифицировать модель с помощью эмуляции целевого аппаратного обеспечения.

В дальнейшем была предложена *SWITCH*-технология [40], в которой для спецификации задач логического управления при их программной реализации используются системы взаимосвязанных графов переходов. При этом были разработаны формальные методы построения программ для логических контроллеров.

Некоторые фирмы создали инструментальные средства программирования логических контроллеров на основе графов переходов. Например, фирма *General Electrics* создала средство *State Logic* [41].

В системах логического управления в качестве входных воздействий используются только входные переменные, а в реактивных системах – еще и события.

Для описания поведения реактивных систем Д. Харелом в работе [7], как было отмечено выше, предложено использовать диаграммы *Statechart*, являющихся расширением графа переходов. Эти диаграммы были использованы в качестве языка спецификации поведения реактивных систем в инструментальном средстве *Statemate* компании *I-Logix* [8, 20]. Эта компания также использует указанные диаграммы в составе более современного пакета для разработки встроенных систем на базе моделей

Rhapsody [42]. Компания *I-Logix* вошла в состав компании *Telelogic*, которая, в свою очередь, входит в состав корпорации *IBM*.

Для моделирования реактивных систем фирмой *Mathworks* создано средство *Stateflow* [43], которое тесно интегрировано с такими известными пакетами, как *MATLAB* и *Simulink*.

Кроме подхода для построения реактивных систем, предложенного Д. Харелом, для их создания разработан и другой подход, основанный на использовании систем взаимосвязанных графов переходов [44], являющийся разновидностью *SWITCH*-технологии.

Диаграммы переходов начали применяться также и для программирования микроконтроллеров [45]. Так, например, фирмой *IAR Systems* создано средство *visualSTATE* [24].

В работе [46] описана методология, названная *Co-Deisgn*. Она ориентирована на совместное проектирование аппаратной и программной частей встраиваемой системы с использованием конечных автоматов. Для верификации построенной системы используется библиотека *VIS* [47].

1.5. Использование автоматного подхода при реализации прикладных программ

Исторически первой областью программирования, в которой использовались автоматы, были компиляторы [48, 49].

В классических алгоритмах автоматы используются крайне редко. Так в книге [50], среди многих алгоритмов приведен только один (поиск подстрок), в котором используются автоматы. Существуют и другие алгоритмы, в которых целесообразно применять автоматы, например, обход деревьев [51].

При создании программ на основе автоматного подхода существуют два варианта программной реализации: компилятивный и интерпретационный. Первый подход предполагает создание кода, реализующего автоматное поведение на целевом языке программирования.

Этот код компилируется и запускается. Интерпретируемый подход предполагает наличие некоторой виртуальной машины, которой передается описание автомата, выполненное на языке отличном от целевого. Эта виртуальная машина загружает описание автомата, преобразует его в объектное представление в памяти и затем интерпретирует его. Преимуществом интерпретируемого подхода является возможность изменения поведения системы без ее перекомпиляции, недостатком является меньшая скорость работы, по сравнению с компилируемым подходом.

При переходе к объектно-ориентированному программированию исследовались различные подходы к совместному использованию автоматов и объектов [52]. Так, в частности, одним из подходов к реализации автоматов, является создание библиотек, реализующих набор базовых классов. Наследуясь от этих классов, программист пишет программу в «автоматном» стиле. К таким библиотекам можно отнести, например, *Werken Blissed* [53] и *boost::fsm* [54], первая из которых предназначена для создания программ на языке *Java*, а вторая – на языке *C++*. Особенность применения таких библиотек состоит в том, что описание структуры автомата выполняется на целевом языке программирования.

Более совершенные библиотеки предоставляют пользователю возможность описывать автомат в текстовом конфигурационном файле, который затем преобразуется в код на целевом языке программирования. К таким библиотеками относятся, например, *Ninni FSM Generator* [55], *Finite State Machine generating software* [56], *FSM* [57], *The State Machine Compiler* [58], CHSM [59].

Форматы текстового описания варьируются: от таблицы переходов автомата Мура [55, 56, 58] до *XML*-описания смешанного автомата [57].

The State Machine Compiler [58] по текстовому описанию графа переходов генерирует код на языках *Java, C++, C#, VB.Net,* реализующий паттерн *State* [27]. Проверку корректности заданного автомата данная библиотека не производит. Библиотека имеет возможность генерировать

графическое представление автомата по заданному описанию, но данную функциональность нельзя считать обоснованной, так как при моделировании поведения системы графическое представление автомата должно является первичным.

При использовании библиотеки [56] на первом этапе текстовое описание автомата преобразуется в бинарное представление, а затем оно передается интерпретатору. В работе [59] предлагается описывать граф переходов автомата непосредственно в коде программы на языке *C/C++*, используя специальные макросы.

При создании приложений с графическим интерфейсом пользователя в последнее время все шире используется паттерн *Model-View-Controller* [27]. Его основная идея заключается в разделении приложения на модель данных, контроллер и представление данных. Модель данных уведомляет контроллер об изменении данных. При этом контроллер обновляет представление данных. В этом случае оказывается удобным реализовывать контроллер с помощью конечного автомата, так как контроллер выполняет обновление представления данных на основе их текущего состояния – анализируя, например, какое окно в данный момент открыто. На сайте [60] приведен список программных продуктов, реализующих описанный подход. Среди них выделим *ViewControl* [22] компании *StateSoft*. Этот продукт ориентирован на разработку Интернет-приложений на основе платформы *J2EE* [62] и позволяет создавать графы переходов, используя *UML*-нотацию диаграммы состояний и собственный графический редактор.

В заключение раздела отметим, что в начале 90-х годов в Мичиганском университете Ю. Гуревичем [63] разработана теория машин абстрактных состояний (*ASM – Abstract State Machine*). В дальнейшем под его руководством в компании *Microsoft* на основе этой теории был разработан язык исполняемых спецификаций *AsmL* (*Abstract State Machine Language*) [64], который в настоящее время используется только для верификации.

1.6. Программные продукты для графического моделирования конечных автоматов

Функцию переходов конечного автомата можно задавать различными способами. Наиболее популярными являются табличное представление и представление в виде графа переходов [65].

Библиотека *Finite State Kernel Creator* [66] предоставляет пользователю графический интерфейс для редактирования таблиц переходов конечных автоматов.

Более наглядным и удобным является описание переходов конечного автомата в виде графа переходов. Существует множество нотаций для представления графа переходов автомата. Наиболее популярной сегодня является нотация, используемая в *UML*-диаграмме состояний, которая является модифицированной версией *Statechart* [7].

На сайте [67] приведен список компаний и их продуктов, предназначенных для создания моделей на языке *UML*. Кроме продуктов, указанных на этом сайте, следует отметить также *UML*-редактор *Real* [68].

UML-редакторы предоставляют средства для моделирования, как статической структуры программы, так и ее поведения. Для описания поведения программ могут использоваться различные диаграммы, в том числе и указанные выше диаграммы состояний.

Большинство *UML*-редакторов поддерживает возможность генерации только части исходного кода, соответствующего диаграмме классов. В отличие от других средств автоматного программирования, описанных в предыдущем разделе и ориентированных на написание и использование текстов программ, *UML*-редакторы предназначены для создания графической модели поведения системы. Однако большинство из них не позволяют автоматически создавать код по диаграммам состояний.

Создание кода не всегда является обязательным. Как отмечалось выше, возможно и другое использование поведенческих диаграмм – их интерпретация после представления на некотором промежуточном языке.

Здесь также следует отметить, что возможность запуска поведенческих *UML*-диаграмм с помощью генерации кода для них или с помощью их интерпретации привело к появлению нового направления, названного *Executable UML* [15]. Среди программных продуктов, реализующих идеи этого направления отметим инструмент *Nucleus UML Suite* [69] компании *Accelerated Technology* и инструмент *iUML* [90] компании *Kennedy Carter*.

Кроме вопросов построения кода по модели или непосредственного ее исполнения, также актуальным является автоматическая проверка формальной корректности модели. Спецификация *UML* [70] содержит описание ограничений, которым должна удовлетворять корректная модель. Такие ограничения описаны и для диаграммы состояний.

В указанной выше спецификации ограничения описываются с помощью *языка объектных ограничений* (*Object Constraint Language*). Предполагается, что *UML*-редакторы должны проверять правильность построения диаграмм с учетом этих ограничений. Существует ряд программных продуктов, ориентированных на проверку *OCL* ограничений [70, 72]. Однако в работе [73] показано, что некоторые из ограничений, описанных в спецификации на язык, противоречат друг другу. В этой работе также показано на примере популярных *UML*-редакторов, что очень мало ограничений реально проверяется. Отметим, что спецификация *UML* допускает присутствие в диаграммах состояний противоречивых переходов и неполноту множества исходящих из состояния переходов.

Еще одним недостатком *UML* является отсутствие полного формального описания операционной семантики для диаграмм состояний – правил их исполнения. Устранению данного недостатка посвящены такие работы как [74–76] и стандарт *ITU-T Recommendation Z.109* [77]. В работе [74] предлагается объединить язык *UML* с языком *SDL* [78], так как для последнего формально определена операционная семантика.

Практически в любой программе имеются состояния, которые фиксируют предысторию выполнения. Однако обычно такие состояния явно не выделяются и принадлежат общему пространству состояний системы, размерность которого обычно очень велика. Как отмечалось выше множество состояний можно декомпозировать на управляющие и вычислительный. Первые из них могут быть выделены явно и их число обычно невелико. Говоря в дальнейшем о явном выделении состояний будем иметь ввиду *выделение именно управляющих состояний*.

В работе [79] в рамках SWITCH-технологии предложен метод проектирования событийных объектно-ориентированных программ с *явным выделением состояний*. Особенность этого подхода состоит в том, что поведение объектов описывается с помощью конечных автоматов, представляемых в форме графов переходов с нотацией, предложенной в работе [44]. *SWITCH*-технология, как она описана в работе [40], несмотря на то, что она не содержит визуальных средств моделирования и библиотек, предлагает методологию перехода от поведенческой модели к коду на целевом языке.

Инструментальным средством для поддержки *SWITCH*-технологии является программа *Visio2Switch* [80], которая позволяет преобразовывать графы переходов, созданные в *Microsoft Visio,* в код на языке *C*.

Далее описаны примеры программных продуктов, позволяющих создавать графы переходов автоматов.

1.6.1. Finite State Machine Editor

Редактор *Finite State Machine Editor* [81] использует ряд идей из *SWITCH*-технологии и реализует редактор для графа переходов конечного автомата. Граф переходов может быть преобразован в код на языках *C++* или *Python*.

Перечислим недостатки данного продукта:

- использование собственной нотации для представления графа переходов;
- использование только компилятивного подхода;
- отсутствие возможности создания вложенных состояний или групп состояний;
- отсутствие вложенных автоматов;
- отсутствие автоматической проверки корректности графа переходов.

1.6.2. Среда разработки Флора

В среде разработки *Флора* [82] было предложено строить объектно-ориентированные программы путем «размещения» объектов из предварительно построенных библиотек на дереве. При этом вручную писалась только та часть программы, которая реализовала ее логику. После знакомства с автоматным программированием [40], в это средство была добавлена возможность описания логики с помощью графов переходов автоматов, которые автоматически исполняются.

Созданную модель системы в среде *Флора* можно «запустить» без генерации кода на целевом языке, что достигается за счет интерпретации созданной модели с помощью встроенной в среду объектной машины. Такой подход также позволяет изменять логику поведения системы прямо во время ее работы.

Недостатком данного продукта является необходимость установки всей среды для запуска разработанных в ней автоматных приложений, а также отсутствие средств проверки корректности модели. Также в среде реализован очень неудобный редактор графов переходов автоматов.

1.6.3. XJTek AnyState

Инструментальное средство *XJTek AnyState* [21] содержит редактор графов переходов и редактор исходного *Java*-кода. При изменении графа переходов выполняется его синхронизация с исходным кодом.

К особенностям данного продукта можно отнести:

- использование *UML* нотации диаграммы состояний;
- сохранение графической информации о диаграмме состояний (координаты состояний, цвета, тип шрифтов) прямо в исходном коде в виде комментариев;
- проверка выполнения некоторых ограничений. Информация о найденных ошибках записывается непосредственно в код;

Недостатком является неудобный графический редактор.

1.6.4. IAR Systems visualSTATE

Инструмент *IAR Systems visualSTATE* [24] предназначен для создания приложений для микроконтроллеров. Этот продукт реализует:

- редактор графа переходов автомата в виде *UML*-диаграммы состояний;
- проверку правильности построения графа переходов с помощью собственного алгоритма;
- интерпретатор созданной модели с помощью интегрированного эмулятора различных микроконтроллеров;
- генерацию программного кода;
- автоматическое создание некоторой документации.

Достоинством продукта является то, что модели, созданных в нем диаграмм, являются также и программами. Переход к коду на целевом языке выполняется только после отладки диаграммы на симуляторе.

Недостаток продукта – отсутствие методологии проектирования встроенных приложений.

1.6.5. Telelogic Tau2

Инструментальное средство *Telelogic Tau2* [83] является редактором диаграмм, поддерживающим стандарт *UML* версии 2 [84]. Средство позволяет проверять корректность построенной модели и запускать ее. При запуске существует возможность использовать встроенный отладчик.

При создании диаграмм состояний существует возможность описывать действия, выполняемые на переходах и в состояниях с помощью как внутреннего *C*-подобного языка, так и на целевых языках программирования, в которые входят *C*, *C++* и *Java*.

При запуске модели система позволяет пользователю посылать ей внешние события. При этом существует возможность автоматического построения диаграммы последовательности вызовов, которая в дальнейшем может быть использованы как тестовый сценарий.

Отличительной особенностью данного программного продукта является возможность генерации программного кода на целевом языке программирования для поведенческих диаграмм.

К недостаткам следует отнести ограниченную поддержку языка *Java,* отсутствие возможности удаленной отладки модели и неудобный графический редактор. Проверка корректности модели выполняется не во время редактирования, а при запуске модели, что также является недостатком.

1.6.6. Borland Together Architect

Пакет *Borland Together Architect* [85] является одним из самых популярных и удобных инструментов для создания *UML*-моделей. В нем существует возможность генерации кода по диаграмме классов для языков *Java*, *C++* и *C#* и обратная генерация – создание диаграммы классов по коду. Обе эти возможности вместе называются *round-trip* [86], и в указанном инструменте они работают синхронно – при изменении кода сразу изменяется модель, а при изменении модели – код.

Также данный пакет позволяет создавать диаграммы последовательности по коду метода класса и, наоборот, создавать код метода класса по диаграмме последовательности.

Недостатком *Borland Together Architect* является его ориентация, в первую очередь, на создание кода, а не на создание модели. Возможность синхронизации кода и диаграммы классов позиционируется создателями инструмента как удобное средство для рефакторинга [87] – улучшения существующего кода.

Еще одним недостатком рассматриваемого пакета являются завышенные требования к ресурсам рабочей станции, на которой предполагается использовать программный продукт, что является следствием реализации упомянутой выше технологии *round-trip*.

1.7. Исполняемый UML

Одним из принципиально новых подходов, развивающихся в настоящее время, как отмечалось выше, является *исполняемый UML* [15, 16], который объединяет статические и динамические диаграммы. Одним из вариантов к реализации этого подхода является разработка *виртуальной машины UML* [88–90].

В проекте [89] модель программной системы предлагается строить следующим образом: структура программы моделируется с помощью *UML*-диаграммы классов, а поведение – с помощью описания каждого метода каждого класса в виде *UML*-диаграммы последовательностей. Такой подход при сложной логике приложения крайне неудобен, так как приводит к очень громоздким моделям.

В проекте [90] предлагается расширить *UML* текстовым платформенно-независимым императивным языком для описаний действий, что приводит к перегрузке графических диаграмм текстовой информацией.

Среди промышленных разработок идея *исполняемого UML* реализована в проекте *Telelogic TAU2* [83]. Однако так как этот проект

является закрытым, то весьма трудно выполнить анализ решений, принятых при его создании. Также закрытыми являются и инструментальные средства *IBM Rational Rose* и *Borland Together*.

1.8. SWITCH-технология

В работе [40] был предложен метод проектирования программ с явным выделением состояний, названный «*SWITCH*-технология» (http://ru.wikipedia.org/wiki/Switch-технология) или «автоматное программирование» (http://ru.wikipedia.org/wiki/Автоматное программирование). В дальнейшем этот метод был развит для событийных систем [44], а потом и для объектно-ориентированных [79].

Особенность этого метода состоит в том, что программы предлагается строить также, как выполняется автоматизация технологических (и не только) процессов, в ходе которой первоначально строится схема связей, содержащая источники информации, систему управления, объекты управления и обратные связи от объектов к системе управления. В предлагаемом подходе система управления реализуется в виде системы взаимодействующих конечных автоматов, каждый из которых является структурным автоматом [91] и имеет несколько входов и выходов. Это отличает их от автоматов с одним входом и одним выходом, традиционно используемых в программировании (например, при разработке компиляторов), которые в теории автоматов называются абстрактными [91].

SWITCH-технология определяет для каждого автомата два типа диаграмм (схема связей и граф переходов) и их операционную семантику. При наличии нескольких автоматов предложено также строить схему их взаимодействия. Для каждого типа диаграмм предложена соответствующая нотация (http://is.ifmo.ru/?i0=science&i1=minvuz2).

Выводы по главе 1

1. Основным недостатком описанных средств автоматного программирования является отсутствие диаграмм, отображающих в явном виде связи между конечными автоматами и объектами, поведение которых моделируется. Отметим, что спецификация *UML* позволяет задавать подобные связи с помощью диаграммы классов, однако ни в одной известной методологии [1–5], такой подход не описан.
2. Указанный недостаток устранен в *SWITCH*-технологии, которая вводит в процесс проектирования диаграмму связей автомата, для описания его интерфейса – входных и выходных воздействий. В *UML*-нотации такую диаграмму удобно представлять в виде диаграммы классов.
3. На основании изложенного и выполненного обзора можно сформулировать задачи, решение которых актуально в настоящее время:
    - разработка методологии моделирования поведения программных систем на основе совместного применения *SWITCH*-технологии и языка *UML*;
    - разработка инструментального средства для поддержки *SWITCH*-технологии с использованием нотации *UML;*
    - обоснование выбора средств разработки, с помощью которых должны быть созданы инструментальные средства;
    - разработка подходов к созданию компилируемых и интерпретируемых исполняемых моделей поведения;
    - реализация методов верификации создаваемых моделей;
    - разработка средств отладки моделей в терминах автоматов.

ГЛАВА 2. РАЗРАБОТКА МЕТОДА ПОСТРОЕНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ АВТОМАТНОГО ПОДХОДА

2.1. Исполняемый графический язык автоматного программирования и метод построения программ на его основе

В настоящей работе предлагается при построении диаграмм в рамках *SWITCH*-технологии сохранить автоматный подход, но перейти к стандартной *UML*-нотации. При этом предлагается, используя нотацию *UML*-диаграмм классов, строить схемы связей автоматов, а графы переходов – используя нотацию *UML*-диаграмм состояний. При наличии нескольких автоматов их схема взаимодействия не строится, а они все изображаются на диаграмме классов. Диаграммы классов (как схема связей) и диаграммы состояний образуют предлагаемый графический язык для описания структуры и поведения программ.

Для проектирования программ с использованием этого языка предлагается следующий метод:

1. На основе анализа предметной области в виде *UML*-диаграммы классов разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними.
2. В отличие от традиционных для объектно-ориентированного программирования подходов [1], из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны – они по собственной инициативе воздействуют на автоматы. Объекты управления пассивны – они выполняют действия, которые вызываются автоматами. Объекты управления также могут формировать значения входных переменных для автоматов. Автоматы

активируются источниками событий и на основании значений входных переменных и текущих состояний воздействуют на объекты управления, переходя в новые состояния.

3. Используя нотацию диаграммы классов, строится схема связей автоматов, которая задает интерфейс каждого их них. На этой схеме слева изображаются источники событий, в центре – автоматы, а справа – объекты управления. Источники событий с помощью *UML*-ассоциаций связываются с автоматами, которым они поставляют события. Автоматы связываются с объектами, которыми они управляют, а также с другими автоматами, которые они вызывают или которые вложены в их состояния.

4. Схема связей, кроме задания интерфейсов автоматов, выполняет функцию, характерную для диаграммы классов – задает объектно-ориентированную структуру программы.

5. Каждый объект управления содержит два типа методов, реализующих входные переменные (`xj`) и выходные воздействия (`zk`). При этом отметим, что объект управления инкапсулирует вычислительные состояния системы, которые обычно явно не выделяются из-за большого их числа.

6. Для каждого автомата с помощью нотации диаграммы состояний строится граф переходов типа *Мура-Мили*, в котором дуги могут быть помечены событием (`ei`), логической формулой из входных переменных и формируемыми на переходах выходными воздействиями.

7. В каждом состоянии могут указываться выходные воздействия, выполняемые при входе и имена вложенных автоматов, которые активны, пока активно состояние, в которое они вложены.

8. Кроме вложенности, автоматы могут взаимодействовать по вызываемости. При этом вызывающий автомат передает вызываемому событие, что указывается на переходе или в состоянии в виде выходного воздействия. Во втором случае посылка события вызываемому автомату происходит при входе в состояние.
9. Каждый автомат имеет одно начальное и произвольное число конечных состояний.
10. Состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Основной особенностью сложных состояний является то, что дуга, исходящая из такого состояния, заменяет однотипные дуги, исходящие из каждого вложенного состояния.
11. Все сложные состояния неустойчивы, а все простые, за исключением начального – устойчивы. При наличии сложных состояний в автомате, появление события может привести к выполнению более одного перехода. Это происходит в связи с тем, что, как отмечено выше, сложное состояние является неустойчивым, и автомат выполняет переходы до тех пор, пока не достигнет первого из простых (устойчивых) состояний. Отметим, что если в графе переходов сложные состояния отсутствуют, то, как и в *SWITCH*-технологии, при каждом запуске автомата выполняется не более одного перехода.
12. Каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования. Источники событий также реализуются вручную.

13. Использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем. Смысл таких символов задает схема связей. При наведении курсора на соответствующий символ на графе переходов во всплывающей подсказке отображается его текстовое описание.

Предлагаемый метод **позволяет спроектировать программу в целом**. На рис. 1 приведен пример схемы связей автомата, а на рис. 2 – его граф переходов.

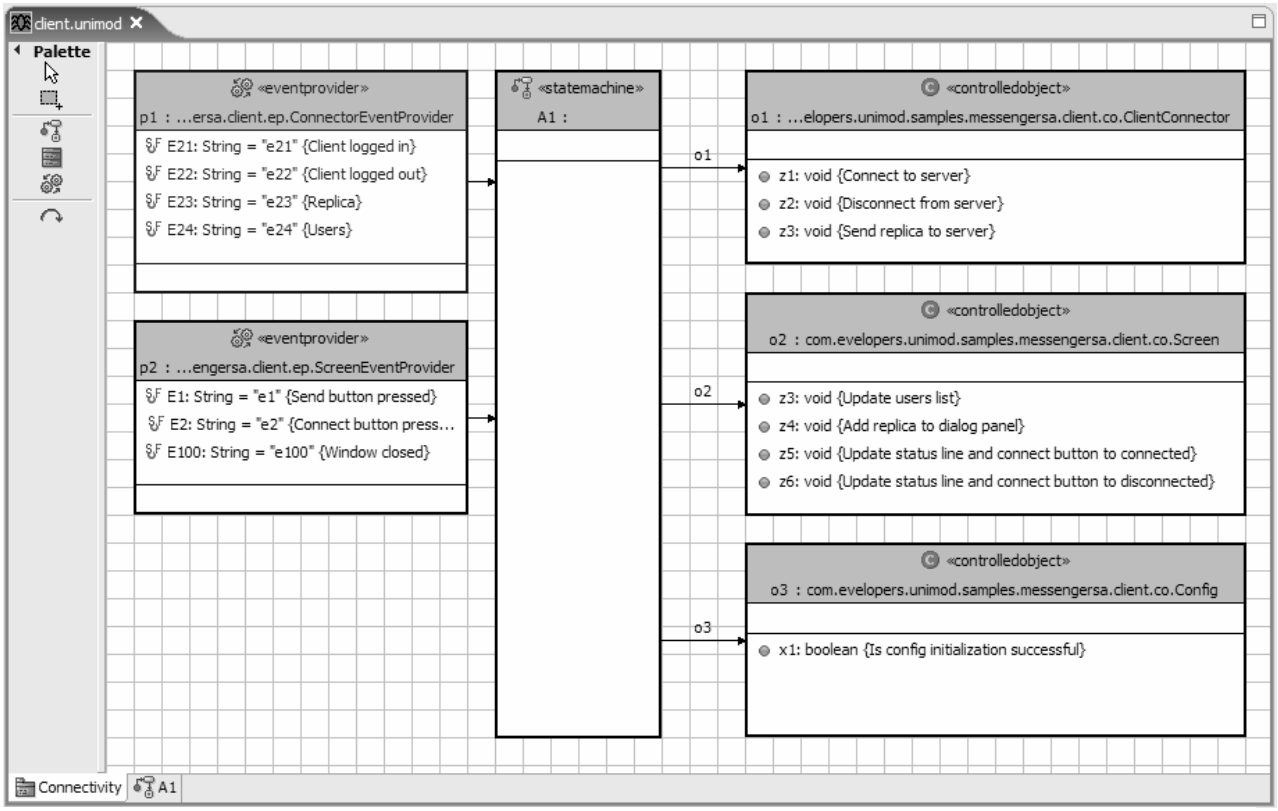

Рис. 1. Пример схемы связей автомата

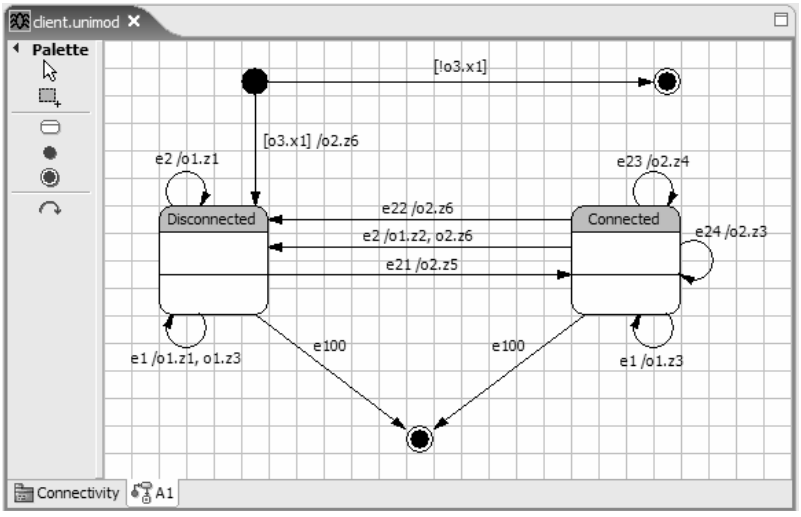

Рис. 2. Пример графа переходов автомата

Опишем синтаксис и операционную семантику предлагаемого графического языка.

2.2. Синтаксис графического языка

Синтаксис созданного графического языка основан на *UML*-нотации. Для текстовых языков программирования синтаксис обычно описывают с помощью формальных грамматик. *UML* является графическим языком и использует другой подход: описывается мета-модель, задающая множество правильных моделей, а затем определяются графические примитивы, соответствующие элементам мета-модели. Диаграммы строятся из указанных примитивов. Сама мета-модель *UML* описана с помощью высокоуровнего средства задания мета-моделей – *MetaObject Facility* (*MOF)* [92].

Предлагаемый графический язык использует только два типа *UML*-диаграмм, а, следовательно, не все элементы мета-модели. Формальное описание используемого подмножества *UML* мета-модели является списком элементов этой модели. Такое описание было бы трудно читаемым. Поэтому далее приводится содержательное описание указанного подмножества.

Статическая модель системы состоит из одной диаграммы классов, на которой изображаются классы со следующими стереотипами: «*EventProvider*» – источник событий, «*StateMachine*» – автомат и

«*ControlledObject*» – объект управления. Между такими классами возможно наличие направленных ассоциаций (дуга со стрелкой определенного вида) трех типов: от источника событий к автомату, от автомата к объекту управления и от автомата к автомату. Ассоциации должны быть помечены метками – идентификаторами.

Для каждого автомата, изображенного на диаграмме классов, необходимо создать диаграмму состояний. Совокупность диаграмм состояний образуют динамическую модель системы.

Диаграмма состояний содержит следующие типы элементов: *начальное*, *нормальное* и *конечное* состояния и переходы между ними. Нормальные состояния на диаграмме могут быть простыми и сложными. Если в нормальное состояние вложено другое состояние, то оно называется сложным. В противном случае – простым. Наличие дуги, исходящей из сложного состояния, заменяет однотипные дуги из каждого вложенного состояния. В каждое сложное состояние вложено ровно одно начальное состояние.

Каждая диаграмма состояний содержит одно *головное* – сложное состояние, содержащее все остальные состояния.

У нормального состояния может быть произвольное число входящих и исходящих переходов. У конечного состояния может быть произвольное число входящих переходов, но не должно быть исходящих. У начального состояния должен быть ровно один исходящий переход.

Переходы между состояниями могут иметь пометки вида:

```
e1[o1.x1 && o2.x3 > 10]/o1.z1, o2.z2, A2.e2
```

Здесь `e1` – название события; `o1`, `o2` – идентификаторы, помечающие ассоциации, которые ведут к первому и второму объектам управления; `x1`, `x3` – методы объектов управления, возвращающие значение типа `boolean` или `int`; `z1`, `z2` – методы объектов управления; `A2` – идентификатор,

помечающий ассоциацию, которая ведет к вызываемому автомату; `e2` – событие, посылаемое вызываемому автомату `A2`. В квадратных скобках задается условие срабатывания перехода (охранное условие) – логическая формула.

В качестве события на переходе может быть использовано либо событие, определенное в одном из источников событий, связанных с данным автоматом, либо специальное событие «*», означающее любое событие.

В качестве переменных в условиях на переходах используются имена методов объектов управления, связанных с автоматом. Действия на переходах задаются списком имен методов объектов управления.

Далее приведена *LL(1)* [48] грамматика для охранного условия:

```
S  -> else | T S'
S' -> or T S' | □
T  -> L T'
T' -> and L T' | □
L  -> not L | P
P  -> '(' S ')' | int rel N | bool | N P'
P' -> rel int | □
N  -> id dot id
```

Терминал **id** соответствует идентификатору, терминал **int** – целочисленной константе, терминал **bool** – булевской константе, а терминал **rel** – бинарному отношению (‘>’, ‘<’, ‘>=’, ‘<=’, ‘=’, ‘≠’).

Внутри нормальных состояний (простых и сложных) могут быть указаны действия, выполняемые при входе в состояние, которые записываются в виде списка имен методов объектов управления. Например:

```
o1.z1, o2.z2
```

Действия, выполняемые при выходе из нормальных состояний, описываемый язык не поддерживает. Внутри нормальных состояний также может указываться и список вложенных автоматов.

UML-состояния с параллельными регионами также не поддерживаются. Это связано с тем, что «проектирование объектов с одним потоком управления является достаточно простым, а для отражения параллелизма следует использовать несколько параллельно исполняемых объектов» [76]. Применительно к описываемому языку, в этой цитате слово «объект» необходимо читать как «автомат».

2.3. Операционная семантика графического языка

Для модели системы, построенной описанным выше образом и состоящей из статической и динамической моделей, зададим операционную семантику:

1. При запуске модели, инициализируются все источники событий и объекты управления. После этого источники событий начинают воздействовать на связанные с ними автоматы.
2. Каждый автомат начинает свою работу из начального состояния, а заканчивает – в одном из конечных.
3. При получении события автомат выбирает все исходящие из текущего состояния переходы, помеченные символом этого события.
4. Автомат перебирает выбранные переходы и вычисляет логические формулы, записанные на них, до тех пор, пока не найдет формулу со значением `true`.
5. Если переход с такой формулой найден, то автомат выполняет выходные воздействия, записанные на дуге, и переходит в новое состояние. В нем автомат выполняет выходные

воздействия, а также запускает вложенные автоматы. Если новое состояние оказалось составным, осуществляется переход из начального состояния, находящегося внутри данного составного состояния.

6. Если среди выходных воздействий встречается вызываемый автомат, то он вызывается с соответствующим событием.
7. Если переход не найден, то автомат продолжает поиск перехода у родительского состояния – состояния, в которое вложено текущее состояние.
8. При переходе в конечное состояние автомат останавливает все источники событий. На этом работа системы завершается.

Правила интерпретации диаграмм состояний представлены в виде *UML*-диаграммы деятельности на рис. 3. Обрабатываемое событие обозначено на диаграмме символом *e*.

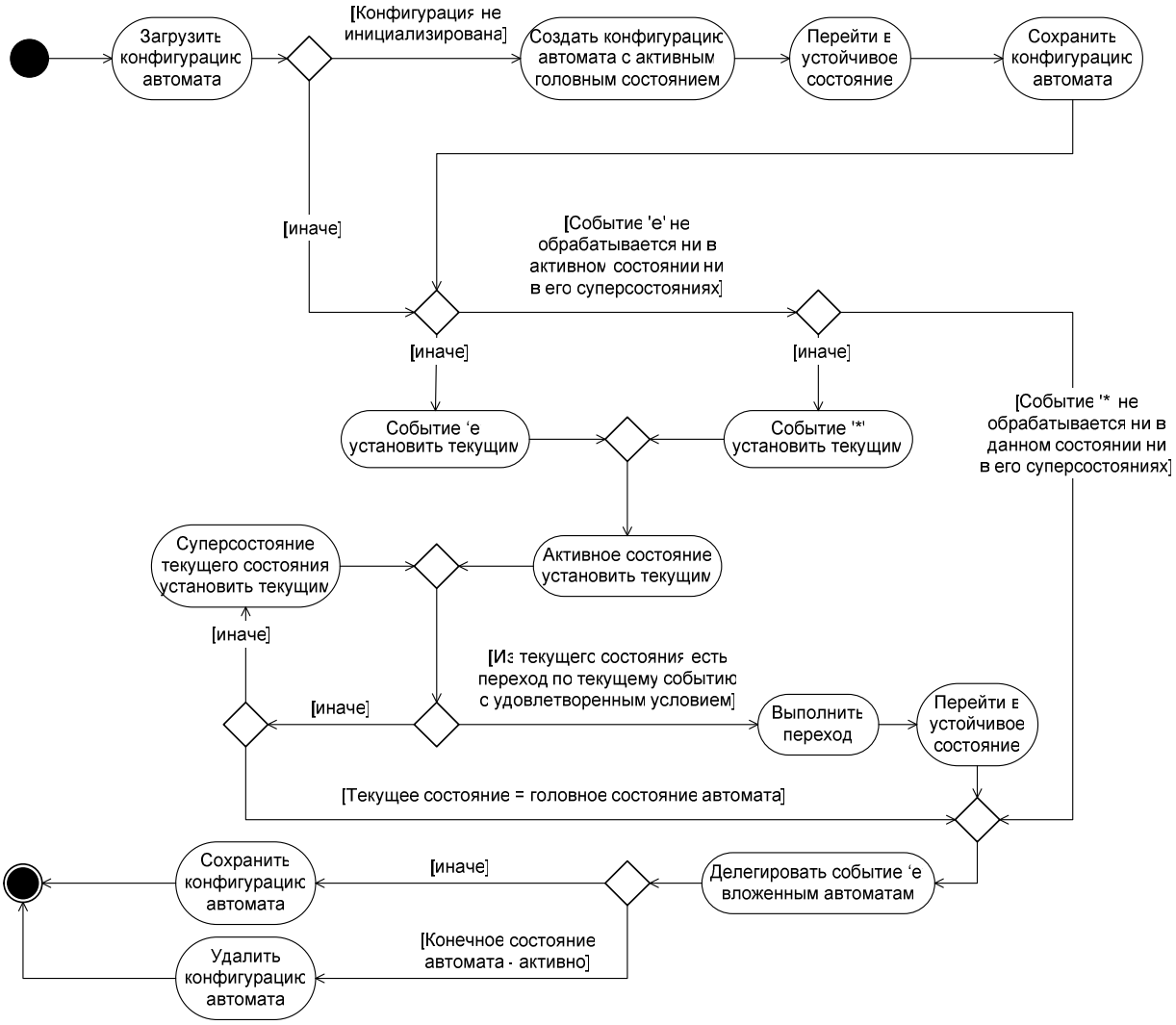

Рис. 3. Правила интерпретации диаграмм состояний

Обработка события автоматом начинается с загрузки конфигурации автомата. *Конфигурация* – это устойчивое состояние автомата, сохраняемое после окончания обработки события. Устойчивыми состояниями являются все простые нормальные и финальные состояния.

Активным состоянием в процессе обработки события называется состояние, переходы из которого анализируются в данный момент.

При обработке первого поступившего события конфигурация автомата не инициализирована, так как она еще ни разу не была сохранена. В этом случае в качестве активного выбирается головное состояние автомата, а затем осуществляется переход в устойчивое состояние. Деятельность «перейти в устойчивое состояние» показана на рис. 4.

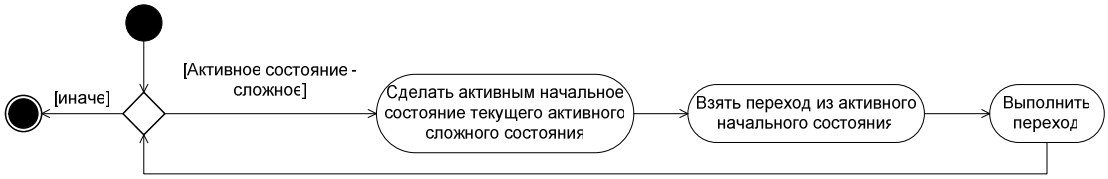

Рис. 4. Деятельность «Перейти в устойчивое состояние»

Если для пришедшего события найден переход, условие на котором удовлетворено, то автомат выполняет этот переход (рис. 5), изменяя активное состояние.

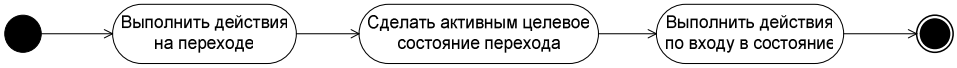

Рис. 5. Деятельность «Выполнить переход»

Переходы, помеченные событием ‘*’, анализируются в том случае, если пришедшее событие *e* явно не обрабатывается в данном состоянии. Целевое состояние выбранного перехода может оказаться сложным. Сложные состояния не являются устойчивыми. Поэтому при достижении такого состояния осуществляется переход в устойчивое состояние (рис. 4).

Описанные правила обработки события автоматом задают операционную семантику диаграммы состояний. Наличие синтаксиса и операционной семантики у диаграмм позволяет использовать их как программы. Существует два варианта реализации таких программ: интерпретационный и компиляционный. В первом случае содержимое диаграмм преобразуется в *XML*-описание и передается интерпретатору, во втором – содержимое преобразуется в код на целевом языке программирования и компилируется.

Выводы по главе 2

1. На основе *SWITCH*-технологии создан метод для разработки реактивных объектно-ориентированных систем.
2. На основе *UML*-нотации создан графический язык для поддержки созданного метода.
3. Описан синтаксис и операционная семантика графического языка.

ГЛАВА 3. ВЕРИФИКАЦИЯ МОДЕЛЕЙ АВТОМАТНЫХ ПРОГРАММ

Практически во всех программах есть ошибки. Их наличие порой приводит к драматическим последствиям. Поэтому в последнее время предлагается множество методов верификации программ.

При традиционном подходе к разработке программного обеспечения задается спецификация задачи, основываясь на которой пишется программа. При этом задача верификации состоит в следующем: доказать, что написанная программа соответствует спецификации. Из изложенного следует, что если спецификация задана неформально, то и доказать в математическом смысле ничего невозможно. Если же спецификация формальна, то некоторые свойства программы доказать можно. В общем случае многие свойства программ доказаны быть не могут [1].

При использовании описанного в предыдущей главе графического языка программирования, создаваемые модели являются и формальной спецификацией и программой одновременно, поэтому их можно подвергнуть формальной верификации.

Общепринятыми подходами к верификации программ являются:

- тестирование;
- имитационное моделирование;
- дедуктивный анализ;
- верификация на модели.

Тестирование является самым простым и распространенным методом проверки работы систем. Общий принцип тестирования заключается в том, чтобы работающей системе подавать на вход определенные входные значения, и проверять, что на выходе получаются требуемые выходные значения. Положительными качествами тестирования являются, во-первых, простота а, во-вторых, надежность, так как тестируется обычно сама работающая система. Гарантируется, что если тесты выполнялись при

тестировании, то они будут выполняться и при реальной работе. Недостатком тестирования является неполнота. Тесты проверяют функциональность системы лишь на некоторых примерах, что, естественно, не гарантирует, что система будет работать на *всех* примерах. С увеличением числа тестов можно быть *достаточно уверенным* в корректности работы программы. Однако *гарантии* корректности при этом дать нельзя. Обычно стремятся к тому, чтобы тесты, по крайней мере, покрывали все переходы в спроектированной системе. Например, желательно, чтобы каждая строка кода выполнялась хотя бы в одном тесте. Для эффективного тестирования создана технология программирования, называемая *Test Driven Development*, и существуют утилиты, указывающие на части кода, которые не выполняются ни при одном тесте. При этом улучшается качество тестирования, однако не решается проблема проверки правильности работы управляющей системы в общем случае.

Имитационное моделирование сходно с тестированием, однако проверяется не работающая система, а модель, имитирующая ее работу. Отсюда и название метода. Имитационное моделирование используется в тех случаях, когда тестирование реальной системы не представляется возможным или требуется проверить корректность проекта до создания прототипа. Таким образом, при неточной модели возможно возникновение ситуации, когда некоторый тест прошел этап имитационного тестирования, но на реальной системе не выполнится. Поэтому в этом методе очень важно точно перенести логику работы системы в ее модель.

Дедуктивный анализ – это формальное доказательство свойств системы. Для управляющей системы строится набор аксиом, из которых затем с помощью формальной логики пытаются доказать выполнимость этих свойств. Преимущество дедуктивного анализа в том, что в случае успешного доказательства можно с точностью утверждать, что свойство выполняется *всегда*. Недостаток дедуктивного анализа состоит в том, что он требует большой ручной работы и высокой квалифицированности специалистов, его

применяющих. Тем не менее, для автоматных программ, рассматриваемых в данной работе, некоторые свойства можно проверить с помощью дедуктивного анализа автоматически (например, полноту и непротиворечивость систем переходов автоматов).

Верификация на модели является практически полностью автоматическим методом для проверки свойств программ с *конечным* числом состояний. Как следует из названия метода, он работает не с реальной программой, а с ее *моделью*. Для проверяемой программы сначала строится формальная модель, описывающая ее поведение. Затем для нее формулируется спецификация – утверждения, истинность которых требуется проверить. После этого выполняется автоматическая верификация, в результате которой либо доказывается, что модель удовлетворяет спецификации, либо это опровергается. Опровержение представляет собой набор действий над моделью, которые приводят к нарушению спецификации.

Отметим также, что при верификации существует еще одна проблема: проверка соответствия формальной спецификации неформально поставленной задаче. Данная проблема может быть решена с помощью тестирования программы, при котором постановщик задачи создает набор входных воздействий и описывает ожидаемую реакцию программы. В случае несоответствия фактической и ожидаемой реакций программы, изменения вносятся в формальную спецификацию и процесс тестирования повторяется.

Далее описаны предлагаемые автором два метода формальной верификации автоматных моделей, использующие дедуктивный анализ и верификацию на модели.

3.1. Дедуктивный анализ автоматных моделей

В стандарте языка *UML* синтаксис и семантика диаграмм определяется набором ограничений, записанных на языке объектных ограничений (*Object Constraint Language*). Этот набор ограничений должен удовлетворяться для любой правильно построенной диаграммы. Предлагается расширить

множество ограничений следующим образом: множество исходящих переходов для любого состояния должно быть полно и непротиворечиво. Это означает, что при обработке любого события не должно быть альтернативных переходов и хотя бы один переход должен выполняться всегда.

UML-диаграммы состояний графами не являются. Для того чтобы распространить теорию графов на эти диаграммы, их необходимо преобразовать.

Определение 1. Определим *UML*-диаграмму состояний, как тройку $D = (S_s, S_c, T)$, где S_s – множество простых состояний, S_c – множество сложных состояний, T – множество переходов между состояниями. Каждый переход задается парой (s_1, s_2), где оба элемента принадлежат объединению множеств простых и сложных состояний.

Будем говорить, что диаграмма состояний $D' = (S_s, S_c', T')$ получена из диаграммы $D = (S_s, S_c, T)$ исключением сложного состояния s_c, если

$S_c' = S_c \setminus \{s_c\}$,

$T' = T \cup \left(\{(s_1, s_2) : s_1 \in s_c, (s_c, s_2) \in T\} \cup \{(s_1, s_2) : s_2 = \text{init}(s_c), (s_1, s_c) \in T\} \right) \setminus \left(\{(s_1, s_c)\} \cup \{(s_c, s_2)\} \right)$,

где $\text{init}(s_c)$ начальное состояние в s_c.

Также будем говорить, что псевдограф $G = (V, E)$ получен из диаграммы состояний $D = (S_s, S_c, T)$ путем исключения всех сложных состояний, если диаграмма $D' = (S_s', \varnothing, T')$ получена из диаграммы $D = (S_s, S_c, T)$ последовательным исключением всех сложных состояний, $V = S_s'$, а $E = T'$.

Утверждение 1. Для каждой диаграммы состояний существует единственный псевдограф, полученный из нее путем исключения всех сложных состояний. Таким образом, процесс исключения всех сложных состояний приводит к одному и тому же результату вне зависимости от порядка исключения состояний.

Все возможные диаграммы состояний можно разбить на классы эквивалентности. Две диаграммы эквивалентны, если им соответствует один и тот же псевдограф. Полученные результаты позволяют распространить теорию графов на диаграммы состояний.

Определение 2. Простое состояние на *UML*-диаграмме состояний называется достижимым, если в псевдографе, полученном исключением всех сложных состояний, соответствующая ему вершина достижима из вершины соответствующей начальному состоянию на *UML*-диаграмме, которое вложено в сложное состояние, которое, в свою очередь, не вложено ни в какое другое сложное состояние.

Множество достижимых состояний можно построить обходом графа переходов «в глубину» [50]. Время, затрачиваемое на обход графа, пропорционально числу вершин в нем – $O(V)$.

Единственность перехода для каждого набора входных воздействий (непротиворечивость), означает, что для каждого состояния условия на всех переходах попарно ортогональны. Таким образом, если c_i – условие на `i-ом` переходе, то

$$i \neq j \Rightarrow c_i \wedge c_j = 0.$$

Полнота множества переходов для состояния, означает, что

$$c_1 \vee c_2 \vee ... \vee c_N = 1.$$

В следующем разделе показано, как решить задачу проверки полноты и непротиворечивости графа переходов конечного автомата.

3.1.1. Разбор, минимизация и интерпретация булевских формул

Для проверки полноты и непротиворечивости множества переходов, для каждой вершины необходимо решать задачу проверки тождественного равенства единице или нулю логической формулы. Это *NP*-полная задача [93].

Для ее решения можно использовать два подхода:

- вычисление значений формулы на всех возможных значениях переменных – построение таблицы истинности;
- минимизация булевской формулы. Для тождественно истинных и тождественно ложных формул минимизация приводит к получению константного значения.

Второй подход обладает рядом преимуществ. Во-первых, он позволяет расширить класс используемых логических формул, дополнив его одноместными предикатами [94]. Во-вторых, в случае, если формула не является тождеством, то ее минимизированное представление можно использовать для облегчения процесса создания корректных диаграмм.

Алгоритм минимизации реализован для грамматики охранных условий, описанной в разд. 2.2.

Построение синтаксического анализатора для данной грамматики осуществлялось с помощью библиотеки *ANTLR* [95], позволяющей автоматически по заданной *LL(k)*-грамматике [48, 49] строить транслятор входного потока в синтаксическое дерево разбора. Дерево, полученное в результате трансляции, можно использовать для вычисления значений формул во время выполнения программы, а также трансформировать в другое дерево.

Реализованный процесс минимизации логической формулы основан на трансформации синтаксического дерева разбора и состоит из следующих этапов:

1. Приведение формулы к дизъюнктивной нормальной форме (ДНФ).
2. Упрощение термов ДНФ с помощью законов идемпотентности, операций булевской переменной и ее отрицания и операций булевской переменной и констант.
3. Нахождение интервалов целочисленных переменных, входящих в терм, при которых терм не обращается в ложь.

Приведение формулы к ДНФ реализуется с помощью последовательности трансформаций, сохраняющих тождественное равенство соответствующих формул. Формула, приведенная к ДНФ, представляет собой дизъюнкцию термов, каждый из которых конъюнкция литералов, а каждый литерал – предикат, булевская переменная или ее отрицание.

Для приведения синтаксического дерева к дереву, соответствующему ДНФ, необходимо и достаточно удовлетворить следующие условия:

- узлы, соответствующие логическому отрицанию, должны быть родительскими только по отношению к узлам, соответствующим вхождениям переменных;
- ни один узел, соответствующий конъюнкции, не должен быть предком по отношению к узлам, соответствующим дизъюнкции.

Для выполнения первого условия дерево трансформируется по закону де Моргана:

$$\neg(a \wedge b) = \neg a \vee \neg b,$$

$$\neg(a \vee b) = \neg a \wedge \neg b$$

и закону отрицания отрицания:

$$\neg(\neg a) = a.$$

На рис. 6 приведена трансформация дерева по закону де Моргана для операции ИЛИ. Для операции И соответствующая трансформация приведена на рис. 7. Преобразование по закону отрицания отрицания показано на рис. 8.

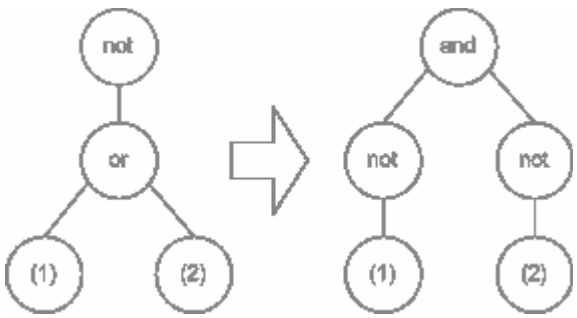

Рис. 6. Преобразование по закону де Моргана для операции ИЛИ

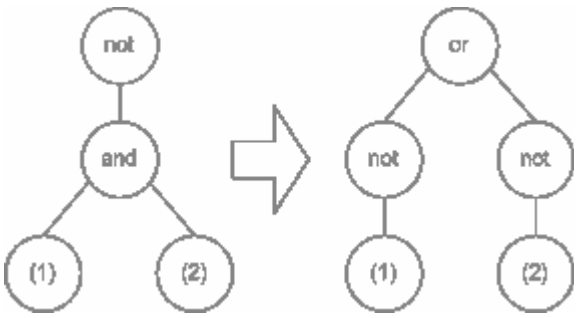

Рис. 7. Преобразование по закону де Моргана для операции И

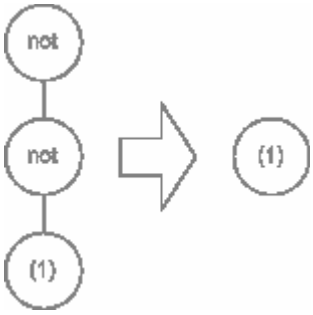

Рис. 8. Преобразование по закону отрицания отрицания

Выполнение второго условия достигается трансформацией согласно дистрибутивному закону алгебры логики:

$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$,

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

Эти соотношения иллюстрируются на рис. 9.

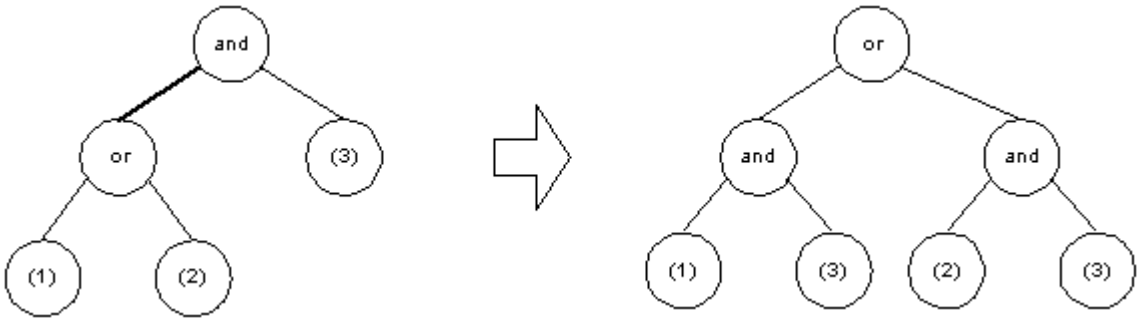

Рис. 9. Преобразование по дистрибутивному закону

Важно отметить, что порядок применения преобразований не безразличен – сначала должно быть удовлетворено первое условие, а лишь затем второе. Если сначала удовлетворить второе условие, а затем первое, то, в общем случае, результирующая формула не будет иметь вид ДНФ, и

необходимо будет применять преобразования до тех пор, пока формула не будет приведена к ДНФ. Это существенно понизит производительность алгоритма.

После приведения формулы к ДНФ поиск минимальной ДНФ не осуществляется, так как такой поиск является крайне ресурсоемкой операцией. К тому же оптимизация термов для тождественно ложных формул гарантированно приводит к тому, что формула обращается в ложь.

Упрощение термов основывается на применении для булевских переменных законов идемпотентности

$a \vee a = a$,

$a \wedge a = a$,

операций переменной и ее отрицанием

$a \vee \neg a = 1$,

$a \wedge \neg a = 0$

и операций с константами

$a \vee 0 = a$,

$a \vee 1 = 1$,

$a \wedge 0 = 0$,

$a \wedge 1 = a$.

Из формулы исключаются термы, в которые хотя бы одна переменная входит вместе со своим отрицанием. Множественные вхождения одного литерала в терм заменяются единственным вхождением.

Для упрощения термов, в которые входят предикаты над целочисленными переменными используется специальная техника, описанная далее.

Определение 3. Ядром предиката P назовем множество значений переменной, на котором предикат P обращается в истину $\mathrm{Ker}\, P = \{x : P(x) = 1\}$.

Если в терм входят предикаты $P_1, P_2, \ldots, P_n$ от переменной x_0, то терм обращается в тождественную ложь, когда

$$\bigcap_{i=1}^{n} \mathrm{Ker}\left(P_i\right) = \varnothing$$

– когда при любом значении переменной x_0 хотя бы один предикат ложен.

Упомянутая техника состоит в том, чтобы для каждой переменной терма построить пересечение ядер предикатов, в которые она входит, и в случае, если хотя бы одно пересечение пусто, исключить терм из формулы. Заметим, что эта техника не использует целочисленной природы переменных. Поэтому может быть распространена на предикаты заданные над произвольным множеством.

Теорема 1. Если ни одно правило для упрощения термов не привело к исключению терма из ДНФ, то существует комбинация значений переменных, обращающая терм в истину.

Доказательство. То, что терм не исключен из формулы ни по одному правилу, означает, что ни одна булевская переменная не входит в формулу со своим отрицанием, и для всех целочисленных переменных пересечение ядер, соответствующих предикатов, не пусто.

В качестве значения булевской переменной возьмем истину, если переменная входит в формулу без отрицания, и ложь – в противном случае. Для целочисленной переменной в качестве значения выберем любое из пересечения ядер, соответствующих ей предикатов. Выбранные значения переменных обращают терм в истину. ■

Как правило, формулы, соответствующие каждому переходу, не слишком длинны. Поэтому, если предварительно привести формулы для условий на каждом переходе к ДНФ, и принять за один шаг алгоритма перемножение термов, то можно оценить время, затрачиваемое на проверку полноты и непротиворечивости.

Оценка сложности алгоритма проверки непротиворечивости. Пусть для данного состояния с данным событием связано N переходов, каждый из которых помечен формулой

$c_i \quad (i = 1,\ldots,N)$.

Число термов в `i`-ой формуле обозначим через `mi`. При этом

$c_i = t_i^1 \vee \ldots \vee t_i^{m_i}$.

Для проверки непротиворечивости `i`-ого и `j`-ого условий необходимо привести к ДНФ их конъюнкцию:

$$c_i \wedge c_j = \left(t_i^1 \vee \ldots \vee t_i^{m_i}\right) \wedge \left(t_j^1 \vee \ldots \vee t_j^{m_j}\right).$$

Эта операция займет $m_i \times m_j$ шагов. Число термов в каждой формуле не превосходит числа $m = \max_i \left(m_i\right)$. Поэтому число шагов для проверки непротиворечивости всей системы переходов можно оценить как

$O(C_N^2 \times m^2)$.

Оценка сложности алгоритма проверки полноты. Для проверки полноты необходимо привести к ДНФ отрицание дизъюнкции всех формул $c_i \quad (i = 1, \ldots, N)$:

$$\neg\left(c_1 \vee \ldots \vee c_N\right) = \neg\left(\left(t_1^1 \vee \ldots \vee t_1^{m_1}\right) \vee \ldots \vee \left(t_N^1 \vee \ldots \vee t_N^{m_N}\right)\right).$$

Из изложенного следует, что приведение этой формулы к ДНФ с помощью описанной выше процедуры и последующая оптимизация термов имеет экспоненциальную от числа формул, записанных на переходах автомата, трудоемкость. Увеличить производительность алгоритма позволяет эвристика, основанная на дополнительной оптимизации термов. Предлагаемая процедура проверки полноты состоит из следующих этапов:

1. Построение дизъюнкции формул условий на переходах.
2. Оптимизация термов в дизъюнкции.
3. Приведение к ДНФ отрицания формулы, полученной на шаге (2).
4. Оптимизация термов в формуле, полученной на шаге (3).
5. Равенство нулю результирующей формулы означает полноту системы формул.

Число термов за счет оптимизации на шаге (2) сокращается. Поэтому приведение к ДНФ на шаге (3) происходит значительно быстрее.

В случае если система не полна, минимизированная формула, полученная на шаге (4), может быть использована в качестве условия на переходе, который необходимо добавить, для обеспечения полноты.

Теорема 2. Формула, полученная на шаге (4), ортогональна каждой формуле $c_i \quad (i = 1,\ldots,N)$, и система условий, дополненная этой формулой, полна.

Доказательство. Формула, полученная на шаге (4), тождественно равна

$$\neg(c_1 \vee \ldots \vee c_N),$$

ее конъюнкция с формулой c_i:

$$\neg(c_1 \vee \ldots \vee c_i \vee \ldots \vee c_N) \wedge c_i = (\neg c_1 \wedge \ldots \wedge \neg c_i \wedge \ldots \wedge \neg c_N) \wedge c_i =$$

$$= \neg c_1 \wedge \ldots \wedge \neg c_i \wedge c_i \wedge \ldots \wedge \neg c_N = \neg c_1 \wedge \ldots \wedge 0 \wedge \ldots \wedge \neg c_N = 0.$$

Полнота системы следует из того, что дизъюнкция формулы и ее отрицания тождественно истинна. ■

Эта теорема используется при построении инструмента для редактирования диаграмм состояний. В случае если введенная пользователем система переходов не полна, инструмент указывает, какое условие следует задать на переходе для исправления ошибки. Подобным образом может быть использована и формула, полученная в результате проверки непротиворечивости двух условий.

Теорема 3. Пусть c' формула, полученная в результате минимизации конъюнкции формул c_i и c_j. Тогда формулы $c_i \wedge \neg c'$ и c_j ортогональны, и если система $\{c_1,\ldots,c_i,\ldots,c_N\}$ полна, то система $\{c_1,\ldots,c_i \wedge \neg c',\ldots,c_N\}$ также полна.

Доказательство. Проверка ортогональности:

$$(c_i \wedge \neg c') \wedge c_j = (c_i \wedge \neg(c_i \wedge c_j)) \wedge c_j =$$

$$= c_i \wedge c_j \wedge (\neg c_i \vee \neg c_j) = (c_i \wedge c_j \wedge \neg c_i) \vee (c_i \wedge c_j \wedge \neg c_j) =$$

$$(0 \wedge c_j) \vee (c_i \wedge 0) = 0$$.

Перейдем к доказательству полноты:

$$c_i \wedge \neg c' = c_i \wedge \neg(c_i \wedge c_j) = (c_i \wedge \neg c_i) \vee (c_i \wedge \neg c_j) = c_i \wedge \neg c_j$$.

Таким образом, дизъюнкция формул равна

$$c_1 \vee \ldots \vee (c_i \wedge c') \vee \ldots \vee c_j \vee \ldots \vee c_N =$$

$$c_1 \vee \ldots \vee (c_i \wedge \neg c_j) \vee \ldots \vee c_j \vee \ldots \vee c_N,$$

но

$$(c_i \wedge \neg c_j) \vee c_j = (c_i \vee c_j) \wedge (\neg c_j \vee c_j) = c_i \vee c_j$$.

Поэтому

$$c_1 \vee \ldots \vee (c_i \wedge \neg c_j) \vee \ldots \vee c_j \vee \ldots \vee c_N = c_1 \vee \ldots \vee c_i \vee \ldots \vee c_j \vee \ldots \vee c_N$$.

Последняя формула тождественно истинна в силу полноты системы переходов $\{c_1,\ldots,c_i,\ldots,c_N\}$. ■

Теорема 3 также используется при построении инструмента редактирования диаграмм – инструмент подсказывает пользователю пути устранения противоречий в системе условий на переходах.

3.2. Верификация на модели

3.2.1. Метод верификации

Автоматная программа может рассматриваться как реактивная система, состоящая из трех компонент:

- источники событий – инициируют работу системы;
- объекты управления – выполняют действия и формируют входные переменные;
- управляющая система – модуль, который принимает события и значения входных переменных и вызывает действия объектов управления. В автоматной программе данный модуль в общем

случае может быть реализован, как система иерархически связанных конечных автоматов.

Верификация на модели автоматных программ состоит в проверке того, что управляющая система работает корректно. Корректность автоматной программы определяется выполнением темпоральных утверждений вида «если произошло событие `e1`, то когда-нибудь будет вызвано действие `z1`» или «если всегда неверно `x1` (`x1` всегда `false`), то автомат никогда не попадет в состояние `s2`». Утверждения, которые требуется проверить, называют *требованиями*, а их совокупность – *спецификацией*. В том случае, если система автоматов удовлетворяет спецификации, считается, что верификация завершена успешно. Если же хотя бы одно из требований не выполняется, то существует последовательность действий, которая приводит к нарушению этого требования. Такая последовательность называется *контрпример*.

У программ, написанных без использования автоматного подхода, отсутствует явное выделение управляющих состояний. Поэтому для таких программ возникает проблема значительного увеличения пространства состояний при верификации. Этого не происходит в автоматных программах, так как в них управляющие состояния выделяются уже на этапе проектирования. Поэтому для автоматных программ могут быть предложены эффективные методы верификации.

Для решения задачи верификации автоматных программ было выполнено несколько работ, в том числе работа [96], в которой был предложен метод верификации автоматных программ с помощью верификатора *SPIN*. В настоящей работе предлагается метод верификации автоматных программ, основанный на *эмуляции* (или имитации) работы автоматной программы. Этот метод позволяет значительно снизить сложность преобразований исходной автоматной программы, необходимых для верификации.

Верификация программ на основе алгоритма двойного поиска в глубину

Верификация программ может выполняться с использованием алгоритма двойного поиска в глубину [97]. Этот алгоритм используется во многих верификаторах, в том числе в верификаторах *SPIN* [98] и *Bogor* [99]. Верификация с применением алгоритма двойного поиска в глубину выполняется следующим образом.

Сначала для верифицируемой программы строится модель *Крипке* [97] – неявно заданный граф элементарных (вычислительных) состояний программы и переходов между ними. Модель *Крипке* является подробной схемой работы программы, в которой в каждом состоянии четко определены элементарные свойства программы.

Требования к программе формулируются в виде формул темпоральной логики. Такие формулы позволяют специфицировать работу программы *во времени*. Темпоральные формулы состоят из *предикатов* – элементарных утверждений о программе, логических операторов («не», «и», «или») и темпоральных операторов – операторов, описывающих выполнение утверждений во времени. Существует несколько, различных по выразительности, типов темпоральных логик. В данной работе используется *LTL* (*Linear Temporal Logic*) [97]. В этой логике допустимы следующие темпоральные операторы:

- **X** (neXt) – «$X\,p$» верно тогда, когда в следующий момент времени в программе будет выполняться предикат p;
- **G** (Globally) – «$G\,p$» верно, если во время работы программы всегда выполняется p;
- **F** (Future) – «$F\,p$» верно, если в будущем наступит момент, когда выполнится p;
- **U** (Until) – «$p\ U\ q$» верно, если в программе в каждый момент времени выполняется p до тех пор, пока не выполнится q. При этом q обязательно должно когда-либо выполниться;

- **R** (Release) – «*q R p*» верно, если *p* выполняется до тех пор, пока не станет выполняться *q* (включая момент, когда выполнится *q*), или всегда, если *q* не выполнится никогда.

Формула *LTL*, описывающая требования к программе, преобразуется в автомат *Бюхи* [97] – конечный автомат над бесконечными словами. Переходы автомата *Бюхи* помечаются предикатами из исходной *LTL*-формулы. Поскольку задача верификатора – найти контрпример, если он существует, то автомат *Бюхи* строится для *отрицания* исходной *LTL*-формулы. Такой автомат *Бюхи* допускает любые последовательности значений предикатов, которые не удовлетворяют требованиям.

Далее модель *Крипке*, построенная для исходной программы, также преобразуется в автомат *Бюхи.* После этого строится его пересечение с автоматом *Бюхи*, построенным по отрицанию *LTL*-формулы. Это пересечение также является автоматом *Бюхи,* и для него запускается алгоритм *двойного поиска в глубину* [97], который находит допускающую последовательность предикатов. Если эта последовательность существует, то:

- она допускается автоматом *Бюхи*, построенным по модели *Крипке*. Следовательно, эта последовательность является историей работы исходной программы;
- последовательность допускается автоматом *Бюхи*, построенным из отрицания *LTL*-формулы. Следовательно, эта последовательность является историей, нарушающей проверяемые требования.

Найденная последовательность предикатов является контрпримером.

В настоящей работе используется верификатор *Bogor*, который интегрируется с разработанным в рамках настоящей работы инструментальным средством *UniMod.*

В верификаторе *Bogor* явно не строятся: модель *Крипке*, автомат *Бюхи* для модели *Крипке* и его пересечение с автоматом *Бюхи* для *LTL*-формулы. Верифицируемая программа записывается на входном языке верификатора.

Для выполнения двойного поиска в глубину верификатору *Bogor* необходимо выполнить следующие действия:

1. Вычислить глобальное состояние программы. Оно должно однозначно определять поведение программы.
2. Совершить элементарный шаг работы программы. Такой шаг является переходом программы из одного глобального состояния в другое без посещения иных глобальных состояний.
3. Откатить назад элементарный шаг работы программы. При этом программа возвращается в предыдущее состояние.
4. В каждом состоянии определить возможные элементарные шаги.
5. Определить значения набора предикатов программы, используемых в требованиях.

Далее описано как данные действия реализуются при верификации автоматной модели.

Верификация автоматных программ с использованием алгоритма двойного поиска в глубину

В данной работе было создано расширение входного языка *Bogor* в виде нового класса – `AutomataModel`. Объект этого класса представляет собой автоматную модель. Для него задано лишь одно действие: `step` (шаг), которое совершает обработку очередного события в модели. Также у объекта этого класса можно получать различную информацию о состояниях автоматов, вызванных в ходе шага выходных воздействиях и других свойствах, которые могут понадобиться для формулировки верифицируемого требования к модели.

За счет создания класса `AutomataModel` спецификация автоматной модели во входном файле верификатора *Bogor* сводится всего лишь к одному бесконечному циклу, в котором совершается шаг автоматной модели. С точки зрения верификатора, обработка одного события в автоматной модели происходит атомарно.

В методе эмуляции указанные в предыдущем разделе действия осуществляются следующим образом:

1. Глобальное состояние программы складывается из набора текущих состояний каждого автомата.

   Верификатор использует инструментальное средство *UniMod,* которое описано в следующей главе, для работы с автоматной программой. Это средство хранит текущие состояния каждого автомата, выполняет по команде разработанного верификатора обработку события и т.д. При обработке события информация о сделанных переходах в автоматах, о выполненных действиях и о новом глобальном состоянии передается из инструментального средства *UniMod* в верификатор. Теоретически для построения модели *Крипке* для системы автоматов необходимо было бы построить один автомат как пересечение всех автоматов системы. Однако это делается неявно в самом инструментальном средстве *UniMod*. При этом верификатор сразу получает информацию о глобальном состоянии системы автоматов, как набор состояний каждого автомата.
2. Элементарный шаг работы программы – это обработка системой автоматов одного события. В результате обработки может смениться набор состояний автоматов.
3. Поведение автоматной программы определяется набором состояний автоматов. Поэтому для отката назад достаточно вернуть автоматы в те состояния, в которых они находились до выполнения шага.
4. Для автоматной программы строго определена схема работы системы автоматов, включая последовательность передачи управления между автоматами. Кроме того отметим, что система работает в одном потоке. Поэтому

недетерминированность в работе системы возникает лишь в результате разных последовательностей входных событий, а также в результате различных возможных значений переменных, запрашиваемых у объектов управления. В методе эмуляции возможные элементарные шаги определяются следующим образом: перед совершением очередного элементарного шага определяется набор событий, которые система автоматов может обработать в текущем глобальном состоянии, и каждое из этих событий затем используется для создания одной из историй работы программы. Аналогично, при необходимости вычислить условие перехода, в котором участвуют переменные объектов управления, такое условие принимается равным `True` или `False`. Оба варианта используются для создания двух различных историй работы программы.

5. Для вычисления предикатов при совершении элементарного шага сохраняется следующая информация:
    - состояния автоматов до выполнения шага;
    - обрабатываемое событие;
    - список вычисленных значений условий на переходах;
    - список вызванных действий у объектов управления.

    Эта информация позволяет вычислять значения предикатов. В методе эмуляции поддерживается следующий набор предикатов:
    - `wasEvent(e)` – возвращает `True`, если в последнем шаге было выбрано для обработки событие `e`;
    - `wasInState(sm, s)` – возвращает `True`, если перед последним шагом автомат sm, находился в состоянии s;

- `isInState(sm, s)` – возвращает *True*, если после совершения последнего шага автомат sm находится в состоянии s;
- `cameToState(sm, s)` – возвращает `True`, если после совершения последнего шага автомат `sm` сменил свое состояние на `s`. Это то же самое, что `(isInState(sm, s) && !wasInState(sm, s))`;
- `cameToFinalState()` – возвращает `True`, если после совершения шага корневой автомат модели перешел в конечное состояние. Это означает, что автоматная программа завершила работу.
- `wasAction(z)` – возвращает `True`, если в ходе выполнения шага было вызвано выходное воздействие `z`;
- `wasFirstAction(z)` – возвращает `True`, если в ходе выполнения шага первым вызванным действием было `z`;
- `wasLastAction(z)` – возвращает `True`, если в ходе выполнения шага последним вызванным действием было `z`;
- `getActionIndex(z)` – возвращает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Этот предикат предназначен для того, чтобы формулировать утверждения, задающие порядок вызова действий объектов управления в автоматной программе;
- `wasTrue(g)` – возвращает `True`, если в ходе выполнения последнего шага один из переходов был помечен условием `g`, и его значение было определено

как `True`. Условие `g`, в общем случае, может не являться значением одной переменной. Например,

```
g = !o1.x1 && o1.x2;
```

- `wasFalse(g)` – возвращает `True`, если в ходе выполнения последнего шага на одном из переходов встречается условие `g`, и его значение было определено как `False`.

Далее приводится сравнение данного метода с другими методами верификации автоматных моделей, а также приводится развернутый пример использования метода.

3.2.2. Сравнение метода эмуляции с методом верификации автоматных программ, известным из литературы

В работе [96] верификация системы автоматов производится следующим образом:

1. Система автоматов преобразуется в модель *Крипке*, записанную на входном языке верификатора *SPIN*.
2. Требования к системе автоматов переводятся в термины построенной модели.
3. Модель верифицируется верификатором *SPIN*. В случае ошибки выдается контрпример в терминах входного языка *SPIN*.
4. Контрпример переводится в термины исходной системы автоматов.

Эта схема верификации изображена на рис. 10.

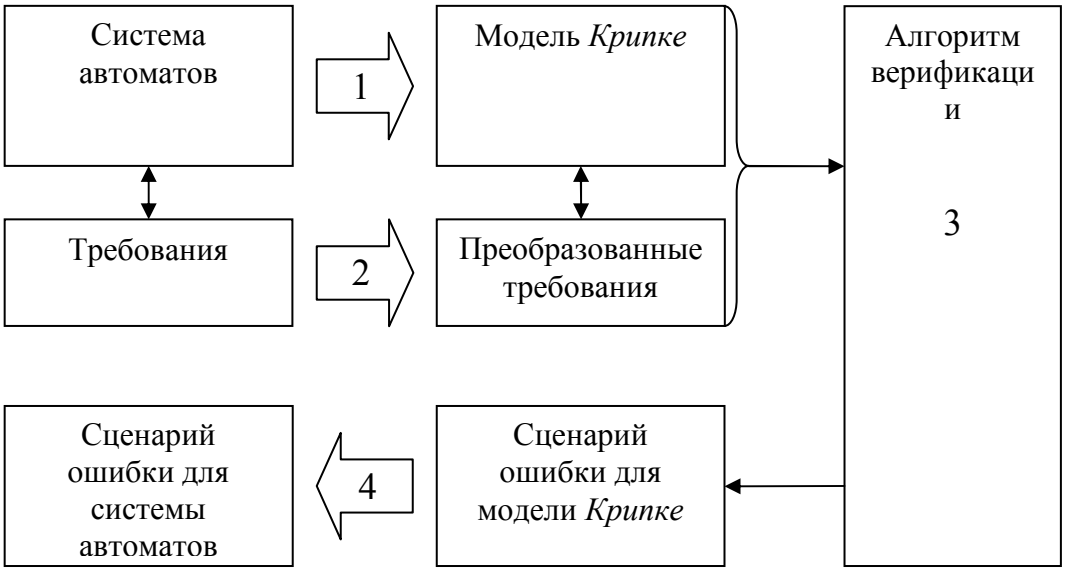

Рис. 10. Схема верификации с явным построением модели *Крипке*

Особенностью предлагаемого метода эмуляции является то, что он не требует дополнительных преобразований автоматной программы и полученного контрпримера. Модель *Крипке* не строится явно, а алгоритм верификации работает непосредственно с системой автоматов. В результате не требуются ни преобразование системы во входной язык верификатора, ни преобразование требований, ни преобразование сценария ошибки, поскольку сценарий сразу выдается в терминах состояний и переходов в системе автоматов. Схема верификации по методу эмуляции изображена на рис. 11.

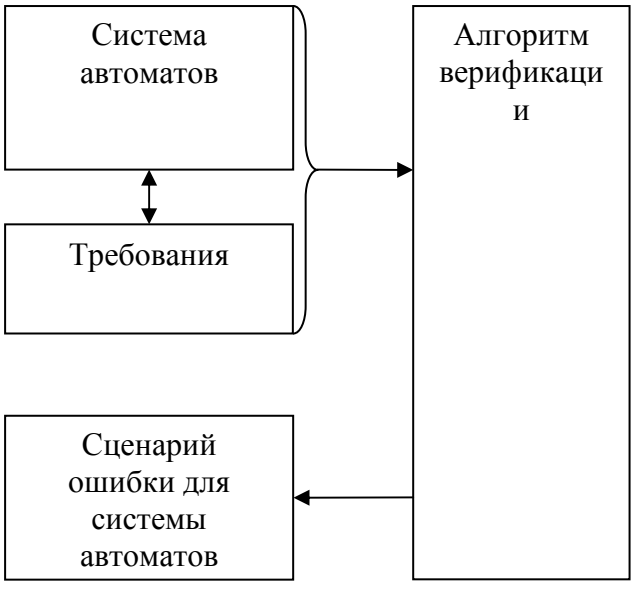

Рис. 11. Схема верификации методом эмуляции

3.2.3. Применение верификатора

Далее описывается применение верификатор для верификации автоматной программы, моделирующей работу банкомата, с целью проверки корректности работы программы.

Описание банкомата

Банкомат – это устройство, автоматизирующее операции по выдаче и переводу денег, хранящихся в банке, лицу, которому они принадлежат. Идентификация каждого клиента происходит с помощью имеющейся у него кредитной карты банка и соответствующего карте секретного *PIN*-кода. Поэтому человек может вне банка снимать деньги или оплачивать определенные услуги.

Сформулируем основные требования к устройству банкомата. Он должен:

- идентифицировать клиента;
- выполнять операции «Показать доступные средства» и «Снять определенную сумму денег»;
- уметь связываться с банком.

Клиентская программа запускается и предлагает пользователю выполнять различные операции с его личной картой.

Первое, что требуется от пользователя – вставить карту. Далее пользователь вводит свой личный *PIN*-код. Если на сервере не найдется запись о счете с введенным номером и *PIN*-кодом, то работа с этой картой прекращается. Если же *PIN*-код и номер счета был введён правильно, то пользователю предлагается выполнить одну из следующих операций:

- «Забрать карту» – возврат карты. Все текущие операции отменяются, и карта возвращается на руки пользователю;
- «Баланс» – отображает текущий остаток на счете, выводя его на экран и предоставляя возможность распечатать на чеке;

- «Снятие денег» – производит операцию снятия денег с карты. Для этого пользователь должен ввести сумму, которую он хочет снять. Клиент пошлёт запрос на сервер о текущем балансе, и получит ответ. Если на карте есть достаточно денег, то операция на сервере завершится успешно, и банкомат выдаст требуемую сумму денег. Также при нажатии на кнопку «Печать» будет напечатан чек по данной операции. Если обнаружится, что на карте недостаточно денег, то с карточки ничего не снимется, и клиент выводит соответствующее сообщение на экран.

После возврата карты, пользователь может вставить её снова либо уйти, нажав кнопку «Выход».

Модель банкомата

Программа банкомата состоит из двух частей – клиентской и серверной. В клиентской части реализован пользовательский интерфейс (`AClient`), а также интерфейс отправки запросов на сервер (`AServer`). Серверная часть производит операции со счетами.

Роль сервера исполняет класс `Server`, написанный и запускающийся отдельно. Поведение клиента моделируется автоматом `AClient` и вложенным в него автоматом `AServer`.

Источники событий:

`HardwareEventProvider` – системные события, генерируемые оборудованием;

`HumanEventProvider` – события, инициируемые пользователем;

`ServerEventProvider` – ответы на запросы, поступающие от сервера;

`ClientEventProvider` – запросы, поступающие на сервер.

Объекты управления:

`FormPainter` – визуализация работы;

`ServerQuery` – отправляет запросы на сервер;

`ServerReply` – отвечает на клиентские запросы.

На рис. 12 изображена схема связей автоматов `AClient` и `AServer`.

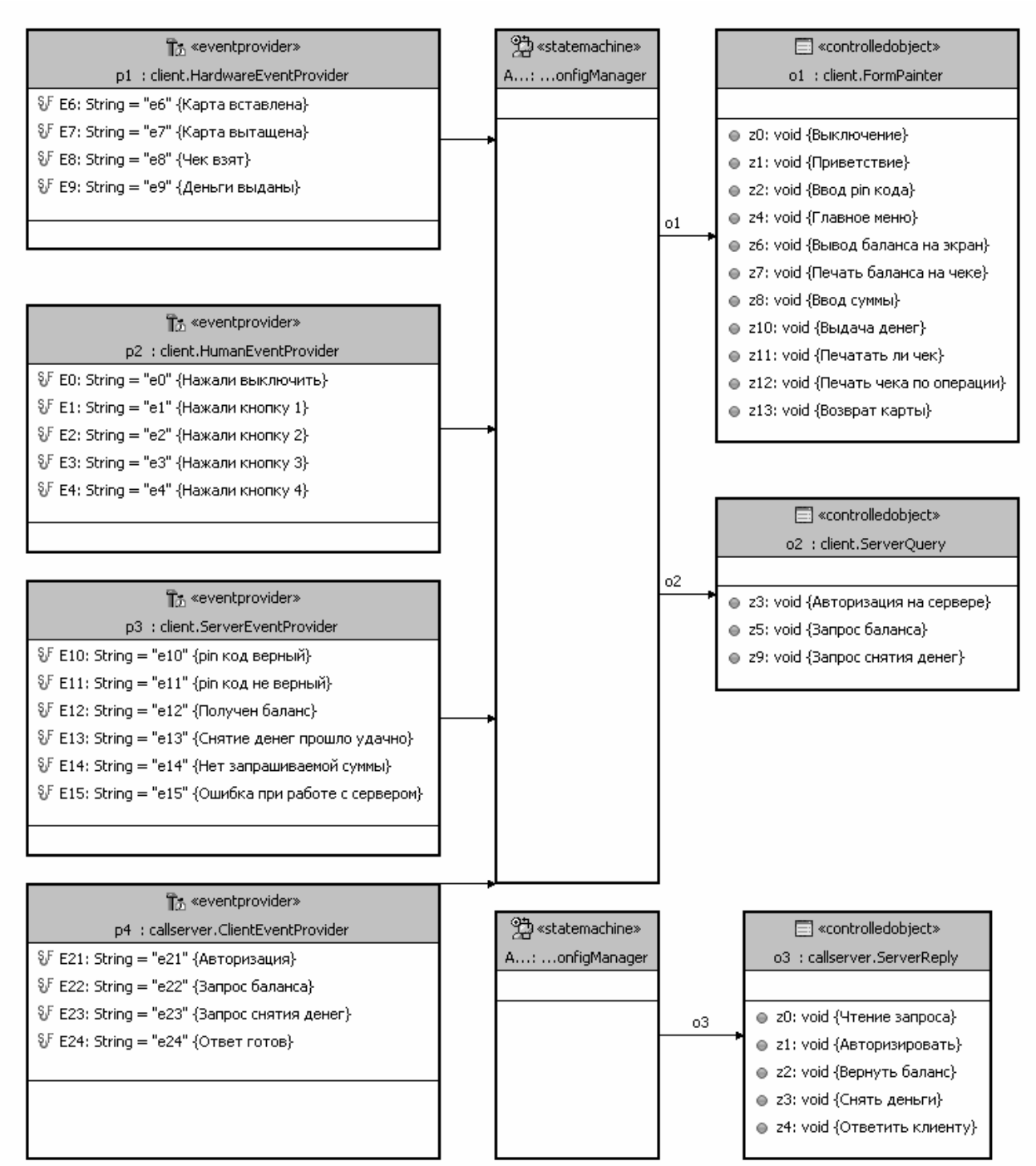

Рис. 12. Схема связей. Несмотря на отсутствие связи между автоматами, `AServer` вложен в `AClient`

На рис. 13 приведен граф переходов автомата `AClient`.

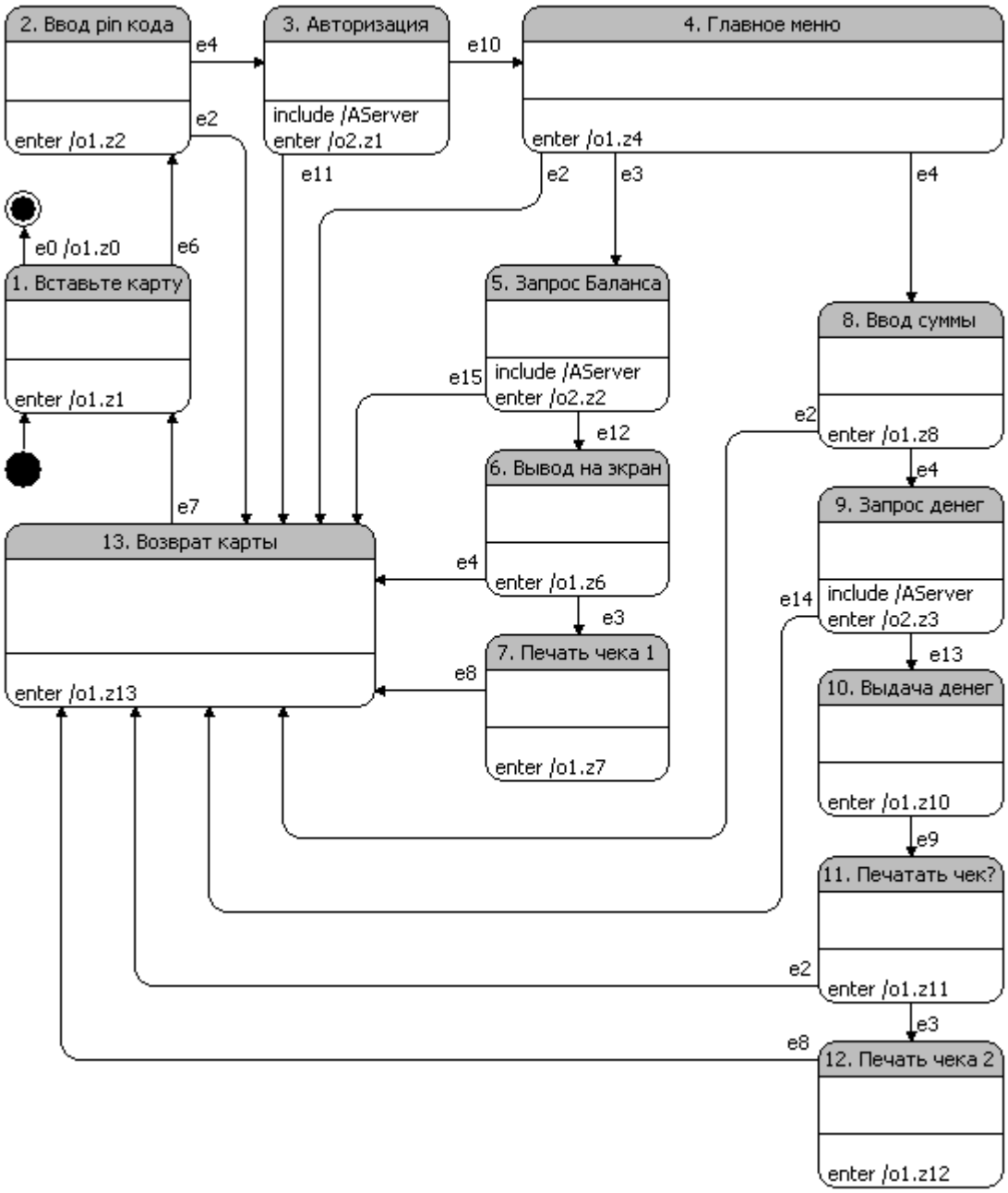

Рис. 13. Автомат `AClient`

На рис. 14 приведен граф переходов автомата `AServer`, посылающего запросы на сервер.

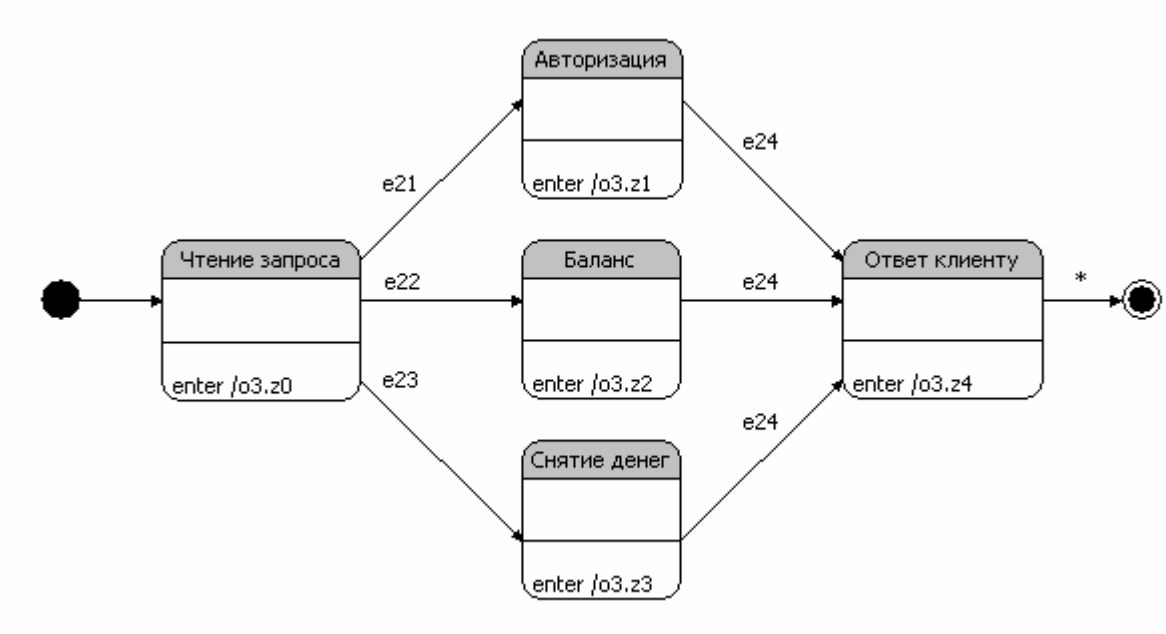

Рис. 14. Автомат `AServer`

Верификация банкомата

Банкомат выдает деньги только после авторизации. Проверим, что пользователь банкомата не получит денег, пока не введет правильный *PIN*-код. Словесная формулировка требования непосредственно переводится в темпоральную *LTL*-логику:

`[не выдадут деньги] U [введет правильный` *PIN*`-код],`

где `U` – темпоральный оператор `Until` – «пока не». Как показано на рис. 12, банкомате выдача денег происходит действием `o1.z10`, а правильно введенный *PIN*-код характеризуется событием `e10`. Таким образом, формула для верификации принимает следующий вид:

```
!o1.z10 U e10,
```

где предикат `o1.z10` означает, что было выполнено действие `o1.z10`, а предикат `e10` означает, что произошло событие `e10`. Запишем эту *LTL*-формулу на входном языке *BIR* верификатора *Bogor*:

```
LTL.temporalProperty(
  Property.createObservableDictionary(
```

```
        Property.createObservableKey(«correct_pin»,
            AutomataModel.wasEvent(model, «e10»)),
        Property.createObservableKey(«give_money»,
            AutomataModel.wasAction(model,
            «o1.z10»))
      ),
      LTL.weakUntil(
        LTL.negation(LTL.prop(«give_money»)),
        LTL.prop(«correct_pin»)
      )
    );
```

Здесь предикат «было выполнено действие `o1.z10`» записан в виде `AutomataModel.wasAction(model, «o1.z10»)` и сохранен под ключом «`give_money`». Аналогично, предикат «`произошло событие e10`» записан в виде `AutomataModel.wasEvent(model, «e10»)` и сохранен под ключом «`correct_pin`». Эти ключи затем использованы для записи самой темпоральной формулы. Стоит заметить, что вместо темпорального оператора `Until` здесь используется его модификация `weakUntil`. Разница между ними в том, что `p Until q` требует, чтобы q когда-нибудь выполнилось, в то время как `p weakUntil q` этого не требует. Действительно, это оправдано в данном случае, так как не гарантировано, что пользователь когда-нибудь введет правильный *PIN*-код.

При верификации созданной формулы верификатор выдает следующий результат:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0, Used Memory: 2MB
Transitions: 63, States: 41, Matched States: 22, Max Depth: 14, Errors found: 0, Used Memory: 1MB
Total memory before search: 765a008 bytes (0,73 Mb)
Total memory after search: 1a202a688 bytes (1,15 Mb)
Total search time: 703 ms (0:0:0)
States count: 41
Matched states count: 22
Max depth: 14
Done!
Verification successful!
```

Таким образом, требование, чтобы банкомат не выдавал деньги до введения правильного *PIN*-кода, выполняется в системе автоматов, управляющих банкоматом.

Деньги не выдаются, пока не сделан соответствующий запрос. Проверим, что деньги не выдаются, пока не сделан соответствующий запрос. Переведем формулировку в LTL-формулу:

```
[не выдаются деньги] U [сделан запрос на выдачу денег]
```

Деньги выдаются с выполнением действия `o1.z10`, а запрос выдачи денег посылается на сервер автоматом *AServer* при событии `e23`, в соответствии со схемой автомата, изображенной на рис. 14. Тогда формула принимает следующий вид:

```
!o1.z10 U e23.
```

На языке *BIR* эта формула запишется следующим образом:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey(«money_requested»,
        AutomataModel.wasEvent(model, «e23»)),
    Property.createObservableKey(«give_money»,
        AutomataModel.wasAction(model, «o1.z10»))
  ),

  LTL.weakUntil (
    LTL.negation(LTL.prop(«give_money»)),
    LTL.prop(«money_requested»)
  )
);
```

Однако, хотя, на первый взгляд, формула кажется выполняющейся в автомате, верификатор выдает ошибку со следующим сценарием:

```
Model [ step [0] event [null] guards [null] transitions [null] actions [null] states [null] ]
fsaState [T0_init]

Model [ step [0] event [] guards [] transitions [] actions [] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (Top)] ] fsaState [T0_init]

Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте карту#*#true] actions [o1.z1]
states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState [T0_init]

Model [ step [2] event [e6] guards [true->true] transitions [1. Вставьте карту#2. Ввод pin
кода#e6#true] actions [o1.z2] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса);
(/AClient:5. Запрос Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение
запроса); (/AClient) - (2. Ввод pin кода)] ] fsaState [T0_init]

Model [ step [3] event [e4] guards [true->true] transitions [2. Ввод pin кода#3.
Авторизация#e4#true] actions [o2.z3] states [(/AClient:9. Запрос денег/AServer) - (Чтение
запроса); (/AClient:5. Запрос Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer)
- (Чтение запроса); (/AClient) - (3. Авторизация)] ] fsaState [T0_init]

Model [ step [4] event [e10] guards [true->true] transitions [3. Авторизация#4. Главное
меню#e10#true] actions [o1.z4] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса);
(/AClient:5. Запрос Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение
запроса); (/AClient) - (4. Главное меню)] ] fsaState [T0_init]

Model [ step [5] event [e4] guards [true->true] transitions [4. Главное меню#8. Ввод
суммы#e4#true] actions [o1.z8] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса);
(/AClient:5. Запрос Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение
запроса); (/AClient) - (8. Ввод суммы)] ] fsaState [T0_init]

Model [ step [6] event [e4] guards [true->true] transitions [8. Ввод суммы#9. Запрос
денег#e4#true] actions [o2.z9] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса);
(/AClient:5. Запрос Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение
запроса); (/AClient) - (9. Запрос денег)] ] fsaState [T0_init]

Model [ step [7] event [e13] guards [true->true] transitions [9. Запрос денег#10. Выдача
денег#e13#true] actions [o1.z10] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса);
(/AClient:5. Запрос Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение
запроса); (/AClient) - (10. Выдача денег)] ] fsaState [bad$accept_all]
```

Как видно, на седьмом шаге выполнилось действие `o1.z10`, хотя за всю историю не происходило события `e23`. Посмотрим, что привело к выдаче денег. На шаге 6 автомат `AClient` попал в состояние «9. Запрос денег», в которое вложен автомат `AServer`. При этом выполнилось действие `o2.z3`, которое, как изображено на рис. 12, означает «Запрос снятия денег». Далее произошло событие `e13`, которое означает «Снятие денег прошло удачно». Возникает вопрос, каким образом произошел запрос снятия денег, если не происходило события `e23`. На самом деле, объект управления `o2` (`ServerQuery`) предназначен для того, чтобы создавать события генератора

событий p4 (ClientEventProvider). Объект управления o2 реализован так, что при вызове действия o2.z3 («Запрос снятия денег») генерируется событие e23 («Запрос снятия денег»).

Верификатор не учитывает логику внутри методов объектов управления. Поэтому он «не знает», что если выполнилось событие o2.z3, то обязательно произойдет событие e23. Для того чтобы все же верифицировать рассматриваемое свойство банкомата, добавим необходимую логику прямо в верифицируемое свойство, в следующем виде: «при условии, что выполняется необходимая логика, верифицируемое свойство тоже выполняется».

Итак, интересующая в данный момент логика объекта управления o2 заключается в том, что если было выполнено действие o2.z3, то будет сгенерировано событие e23. Запишем это в *LTL*-логике:

```
G (o2.z3 → X e23),
```

где X – *LTL*-оператор Next. Добавим теперь полученное условие в верифицируемую формулу, используя оператор следования, как было предложено выше:

```
( G (o2.z3 → X e23) ) → ( !o1.z10 U e23 ).
```

Теперь запишем эту формулу на языке *BIR*:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
  Property.createObservableKey(«server_request_money»,
        AutomataModel.wasAction(model, «o2.z3»)),
    Property.createObservableKey(«money_requested»,
        AutomataModel.wasEvent(model, «e23»)),
    Property.createObservableKey(«give_money»,
        AutomataModel.wasAction(model, «o1.z10»))
  ),
```

```
  LTL.implication(
    /* Ограничение: o2.z3 генерирует e23 */
    LTL.always (LTL.implication (
      LTL.prop(«server_request_money»),
      LTL.next(
        LTL.prop(«money_requested»)
      )
    )),

    /* Свойство для проверки */
    LTL.weakUntil (
      LTL.negation(LTL.prop(«give_money»)),
      LTL.prop(«money_requested»)
    )
  )
);
```

Верификация данной формулы оказывается удачной, что означает, что управляющая система банкомата действительно выполняет выдачу денег лишь после того, как она запросила наличие денег у сервера.

Если произойдет ошибка, то карта будет возвращена. Проверим, что если произойдет ошибка взаимодействия банкомата с сервером, то карта будет возвращена пользователю. В темпоральной логике такое свойство можно записать следующим образом:

```
G ([произошла ошибка] → F [карта будет возвращена]),
```

где `G` – темпоральный оператор *Globally*, а `F` – темпоральный оператор *Future*. Событие «Ошибка при работе с сервером» кодируется в банкомате как `e15`, а возвращение карты – это действие `o1.z13`. Тогда формула принимает следующий вид:

```
G (e15 → F o1.z13).
```

На языке *BIR* формула выглядит следующим образом:

```
LTL.temporalProperty (
```

```
  Property.createObservableDictionary (
    Property.createObservableKey(«error»,
        AutomataModel.wasEvent(model, «e15»)),
    Property.createObservableKey(«card_returned»,
        AutomataModel.wasAction(model, «o1.z13»))
  ),

  LTL.always (LTL.implication (
    LTL.prop(«error»),
    LTL.eventually (LTL.prop («card_returned»))
  ))
);
```

Верификация данной формулы успешна.

Безусловная выдача денег. Верифицируем заведомо ложное свойство банкомата, для того чтобы проверить способность верификатора находить ошибки. Например, проверим, что «пользователь всегда получает деньги». В темпоральной логике оно запишется следующим образом:

```
G F [пользователь получает деньги]
```

Деньги выдаются при выполнении действия `o1.z10`, так что формула в этом случае примет следующий вид:

```
G F o1.z10.
```

На языке *BIR*, соответственно, это записывается как

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey(«give_money»,
        AutomataModel.wasAction(model,
        «o1.z10»))
  ),
```

```
        LTL.always (LTL.eventually (LTL.prop
              («give_money»)))
    );
```

В результате верификации этой формулы верификатор выдает следующий контрпример:

```
Model [ step [0] event [null] guards [null] transitions [null] actions [null] states [null] ]
fsaState [T0_init]

Model [ step [0] event [] guards [] transitions [] actions [] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (Top)] ] fsaState [T0_init]

Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте карту#*#true] actions [o1.z1]
states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState [T0_init]

Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте карту#s2#e0#true]
actions [o1.z0] states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ]
fsaState [T0_init]

Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте карту#s2#e0#true]
actions [o1.z0] states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ]
fsaState [bad$accept_S2]

Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте карту#s2#e0#true]
actions [o1.z0] states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ]
fsaState [bad$accept_S2]
```

В этом контрпримере автоматная система совершает лишь два шага: переход из начального состояния в состояние «1. Вставьте карту», а затем переход по нажатию кнопки «Выключить» (событие `e0`) в конечное состояние главного автомата `AClient`. Как и предполагалось, выдачи денег не происходит в этой истории работы банкомата.

Заметим, что в контрпримере строка с шагом «2» повторяется три раза. Это связано с тем, что после совершения второго шага автоматная система перестает работать, и шаги совершает лишь автомат *Бюхи*: из состояния `T0_init` в состояние `bad$accept_S2`.

Выводы по главе 3

1. На основе преобразований логических формул разработан метод верификации автоматных программ, позволяющий проверять полноту и непротиворечивость систем переходов автоматов. В случае нахождения ошибок метод позволяет автоматически предлагать пути корректировки модели. При этом в отличие от метода верификации на модели, никакой дополнительной информации от пользователя не требуется.
2. На основе подхода к верификации на модели разработан метод верификации автоматных программ. Выполнен эксперимент по проверке эффективности разработанного метода на модели банкомата. Он показал, что метод адекватно проверяет заданные свойства.
3. Разработанные методы позволяют находить логические и семантические ошибки на стадии проектирования, что повышает качество разрабатываемых программ, одновременно уменьшая трудозатраты.

ГЛАВА 4. ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ДЛЯ ПОДДЕРЖКИ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ UNIMOD

Инструментальное средство *UniMod* обеспечивает разработку, верификацию и выполнение автоматных программ в соответствии с методами, описанными в предыдущих главах. Это инструментальное средство позволяет создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схемам связей и графам переходов автоматов.

Проектирование программ с использованием этого средства состоит в следующем: поведение приложения описывается системой взаимодействующих автоматов, заданных в виде набора указанных выше диаграмм, построенных с использованием *UML*-нотации. Источники событий и объекты управления реализуются вручную на целевом языке программирования.

Рассматриваемое инструментальное средство поддерживает два основных типа обработки построенных диаграмм – интерпретацию и компиляцию.

4.1. Интерпретация

Интерпретационный подход реализует *виртуальную машину UML.* На рис. 15 приведена структурная схема для интерпретационного подхода.

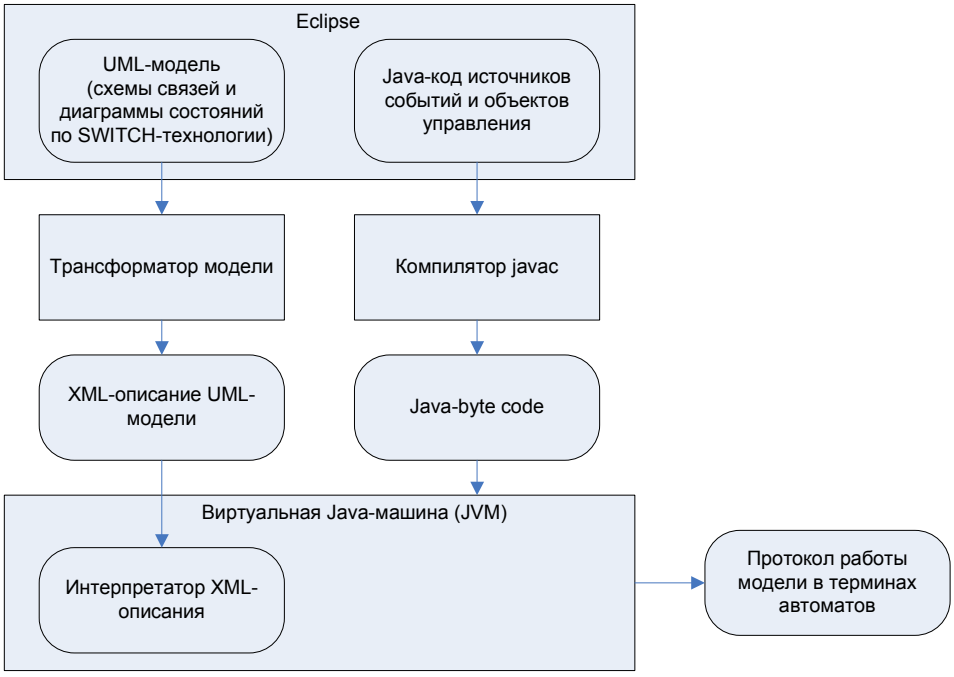

Рис. 15. Структурная схема интерпретационного подхода

Из структурной схемы следует, что при использовании интерпретационного подхода **исходным кодом являются *UML*-модель** (схемы связей и диаграммы состояний по *SWITCH*-технологии) **и *Java*-код источников событий и объектов управления**.

При запуске программы интерпретатор, входящий в состав средства *UniMod,* загружает в оперативную память *XML*-описание модели и создает экземпляры источников событий и объектов управления. Указанные источники формируют события и направляют их интерпретатору, который обрабатывает их в соответствии с логикой, описываемой автоматами. При этом автоматы вызывают методы объектов управления, реализующие входные переменные и выходные воздействия.

4.2. Компиляция

На рис. 16. приведена структурная схема для компилятивного подхода.

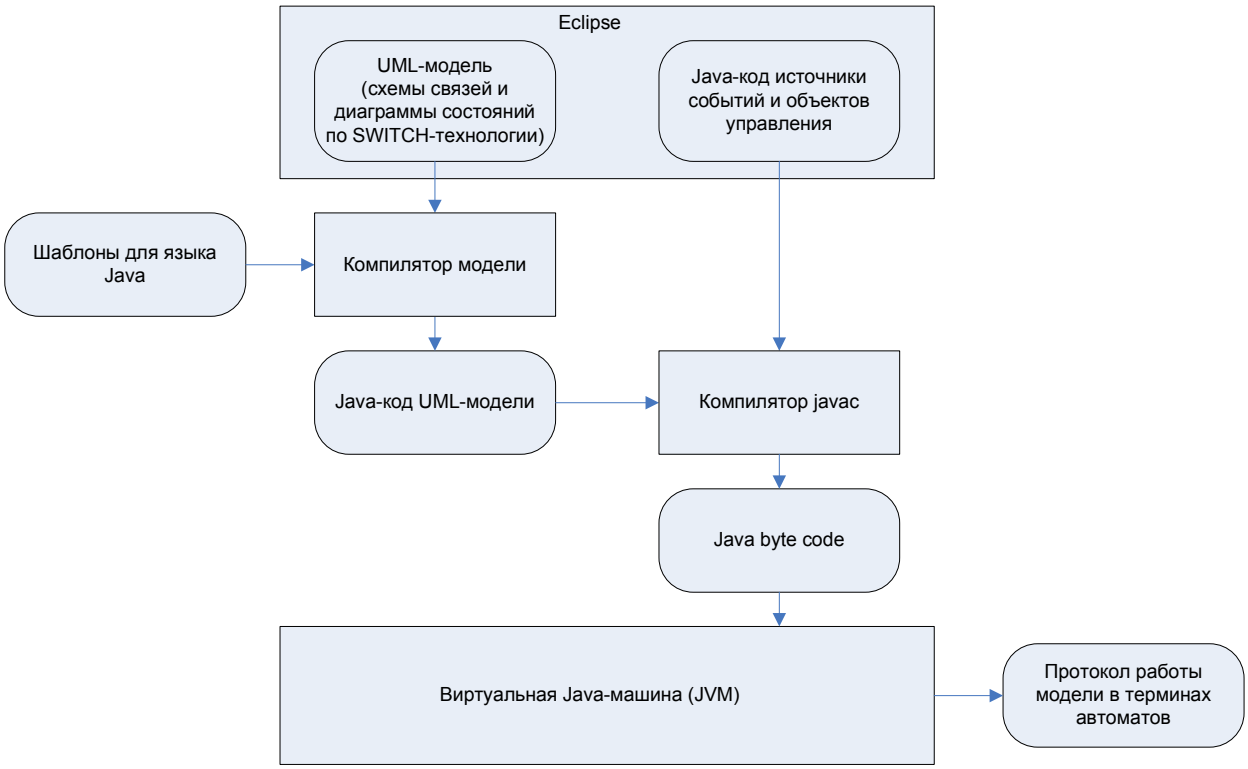

Рис. 16. Структурная схема компилятивного подхода

При использовании компилятивного подхода *UML*-модель **непосредственно** преобразуется в код на целевом языке программирования, который впоследствии компилируется и запускается. Для преобразования в код применяются *Velocity*-шаблоны [101]. Это позволяет адаптировать компилятивный подход для языков программирования, отличных от языка *Java*, например для *C++*.

Указанный подход целесообразно применять для устройств с ограниченными ресурсами. Этот подход является типичным для «классической» *SWITCH*-технологии.

4.3. Реализация редактора диаграмм на платформе *Eclipse*

Редактор для создания указанных диаграмм является встраиваемым модулем (*plug-in*) для платформы *Eclipse* (http://www.eclipse.org). Эта платформа обладает рядом преимуществ перед такими продуктами, как, например, *IntelliJ IDEA* или *Borland JBuilder*, так как:

- является бесплатным продуктом с открытым исходным кодом;

- содержит библиотеку для разработки графических редакторов – *Graphical Editing Framework*;
- активно поддерживается фирмой *IBM* и уже сейчас обладает не меньшей функциональностью, чем упомянутые выше аналоги.

Для обеспечения процесса активной разработки программ на текстовых языках в перечисленных выше средствах разработки реализованы:

- подсветка семантических и синтаксических ошибок;
- завершение ввода и исправление ошибок ввода;
- форматирование и рефакторинг [87] кода;
- исполнение и отладка программы внутри среды разработки.

В английском языке эти возможности называются «*code assist*». При создании модуля для платформы *Eclipse* указанные возможности были реализованы для редактирования диаграмм.

4.3.1. Завершение ввода и исправление ошибок ввода

Традиционно для текстовых языков программирования завершение ввода состоит в том, чтобы по заданному началу лексемы определить набор допустимых конструкций, префиксом которых данное начало является. При этом пользователю предлагается выбрать одну из лексем.

В текстовых языках исправление ошибок ввода состоит в том, чтобы для каждой найденной ошибки указать пользователю варианты ее исправления.

В предлагаемом графическом языке оба эти подхода использованы при редактировании пометок переходов.

В виду того, что предлагаемый язык наряду с текстовой информацией содержит также и графическую информацию, дополнительно выполняется исправление графических ошибок ввода. Так для недостижимого состояния пользователю будет предложено добавить переход в это состояние из любого достижимого (рис. 17).

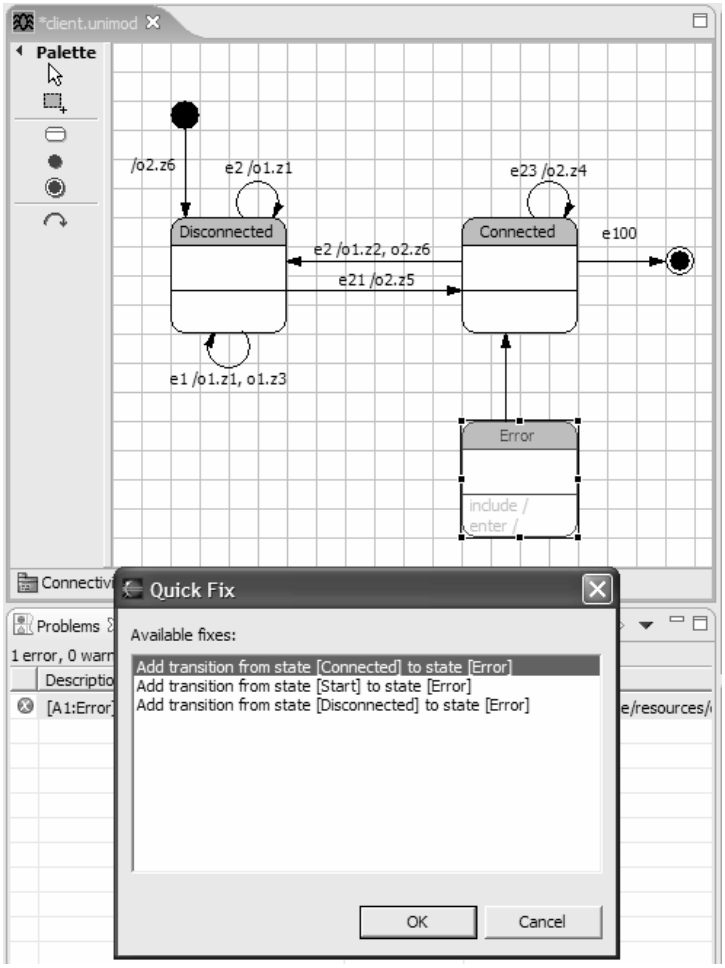

Рис. 17. Предлагаемые варианты исправления ошибки на диаграмме

4.3.2. Форматирование

Форматирование кода облегчает его чтение. Многие текстовые редакторы позволяют автоматически форматировать код.

Аналогом форматирования кода применительно к диаграммам является их укладка (*layout*). Задача укладки диаграмм является существенно более сложной, чем форматирование кода, так как общепринятые эстетические критерии качества укладки отсутствуют. В проекте *UniMod* раскладка диаграмм выполняется методом отжига [102], который дает удовлетворительные результаты, при необходимости улучшаемые вручную.

4.3.3. Исполнение модели

Традиционно используются следующие варианты исполнения программ, написанных на текстовых языках программирования:

- текст программы компилируется в код, исполняемый операционной системой (*Pascal, C++*);
- текст программы компилируется в код, исполняемый виртуальной машиной (*Java, C#*);
- текст программы непосредственно исполняется интерпретатором (*JavaScript*, *Perl*).

Подобные решения применяются и для предлагаемого графического языка. Основными вариантами исполнения являются второй и третий.

4.4. Отладка модели

Несмотря на наличие в пакете *UniMod* встроенных средств для верификации модели, некоторые логические ошибки не могут быть найдены автоматически. Поэтому возникает необходимость отладки *UML*-диаграмм состояний.

Обычно после локализации ошибки отладка представляет собой трассировку программного кода оператор за оператором с одновременным анализом значений переменных.

Для графической автоматной модели отладка – это трассировка графа переходов с анализом текущего состояния, событий и значений входных переменных. При необходимости возможна отладка текстового кода входных и выходных воздействий.

На рис. 18 показана архитектура графического отладчика.

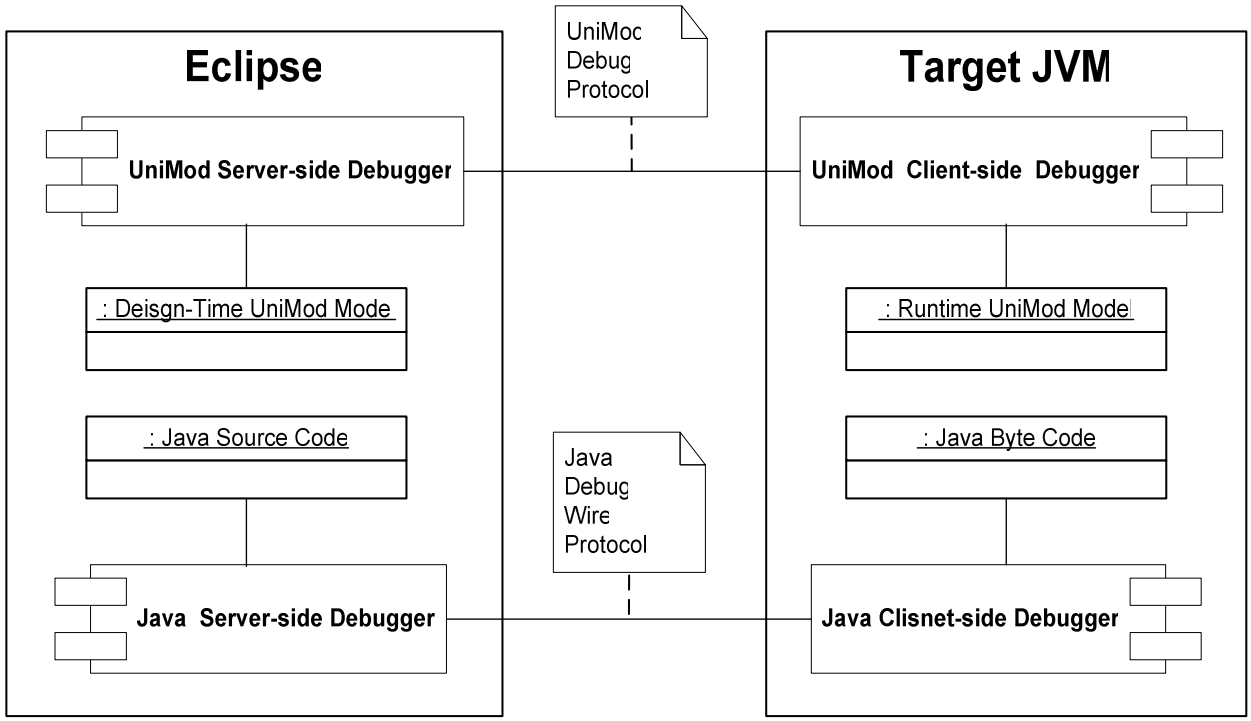

Рис. 18. Архитектура отладчика

При запуске модели в режиме отладки, внутри платформы *Eclipse* должен создаваться компонент `UniMod Server-side Debugger`, а в целевой виртуальной *Java* машине (*JVM*) должен создаваться компонент `UniMod Client-side Debugger`. Эти компоненты взаимодействуют, используя протокол *UniMod Debugger Protocol*. Для поддержки возможности отладки текстового *Java* кода объектов управления между платформой *Eclipse* и целевой *JVM* также устанавливается соединения посредством стандартного протокола *Java Debug Wire Protocol* [103].

UniMod Debugger Protocol должен поддерживать следующий регламент взаимодействия между указанными выше компонентами:

1. При старте компонента `UniMod Client-side Debugger` приостанавливает исполнение модели, создает серверный сокет [104] и ожидает присоединения к нему компонента `UniMod Server-side Debugger`.
2. После установления соединения эта компонента ожидает получения списка точек останова, возможно пустого.

3. После получения списка точек останова компонента UniMod Client-side Debugger регистрирует их и возобновляет исполнение модели.
4. Каждый шаг исполнения модели контролируется и при достижении точки останова компонент UniMod Client-side Debugger приостанавливает исполнение модели, а также информирует об этом событии компонент UniMod Server-side Debugger.
5. Компонент UniMod Server-side Debugger при получении события о достижении точки останова, графически выделяет соответствующий элемент на диаграмме состояний и ожидает команды от пользователя. При этом пользователь имеет возможность, как возобновить исполнение модели до следующей точки останова, так и выполнить только один шаг исполнения модели. О выбранном пользователем действии извещается UniMod Client-side Debugger.
6. Во время отладочной сессии у пользователя существует возможность вносить изменения в отлаживаемую модель. В этом случае, сразу после внесения изменений, компонент UniMod Server-side Debugger пересылает новую модель в целевую *JVM*. Такой подход позволяет ускорить цикл разработки, так как при нахождении ошибки в модели во время отладочной сессии, эту ошибку можно исправить и продолжить отладку уже новой модели без перезапуска целевой *JVM*.
7. Также в процессе отладочной сессии пользователь может устанавливать новые точки останова и удалять существующие. Об этих действиях извещается компонент UniMod Client-side Debugger.

Из приведенного выше регламента видно, что компоненты UniMod Server-side Debugger и UniMod Client-side Debugger взаимодействуют посредством посылки команд и извещений о событиях.

Отметим, что так как серверный сокет создает компонент UniMod Client-side Debugger, то с точки зрения архитектуры клиент-сервер, он и будет являться сервером данной системы, а компонент UniMod Server-side Debugger – клиентом.

При этом компонент UniMod Server-side Debugger должен посылать команды, описанные в табл. 1.

Таблица 1. Команды, посылаемые компонентом UniMod Server-side Debugger

Команда	**Описание**
SET_BREAKPOINTS	Установить точки останова
REMOVE_BREAKPOINTS	Удалить точки останова
STEP	Выполнить один шаг исполнения модели
RESUME	Возобновить исполнение модели до следующей точки останова
UPLOAD_NEW_MODEL	Загрузить новую модель

Компонент UniMod Client-side Debugger извещает о событиях, описанных в табл. 2.

Таблица 2. События, формируемые компонентом `UniMod Client-side Debugger`

Событие	Описание
THREAD_CREATED	Создан новый поток, в котором автомат обрабатывает события
SUSPENDED_ON_BREAKPOINT	Исполнение модели приостановлено на точке останова
SUSPENDED_ON_STEP	Выполнен один шаг исполнения модели и исполнение модели приостановлено
RESUMED	Исполнение модели возобновлено
CANT_UPDATE_MODEL	Невозможно обновить модель
UNKNOWN_COMMAND	Получена неизвестная команда

Ниже приведены возможные типы точек останова:

- достижение состояния на диаграмме состояний;
- выполнение перехода между состояниями;
- получение значения входной переменной при вычислении охранного условия на переходе;
- вызов выходного воздействия на переходе;
- вызов выходного воздействия по входу в состояние;
- вызов вложенного автомата.

4.4.1. Статическая модель отладчика

На рис. 19 показана статическая модель отладчика. Классы слева описывают структуру компонента `UniMod Client-side Debugger`, а классы справа – `UniMod Server-side Debugger`.

Классы `app.AppConnector` и `debugger.DebuggerConnector` реализуют сетевое взаимодействие.

Классы `app.BreakpointManager` и `debugger.BreakpointManager` управляют точками останова.

Класс `app.ThreadManager` приостанавливает и возобновляет поток выполнения в отлаживаемой модели.

Класс `app.EvetProcessorEventProvider` следит за процессом обработки событий в отлаживаемой модели.

Класс `app.ModelManager` сохраняет новую версию модели до того момента, когда на ее можно будет заменить старую версию.

Класс `debugger.UIManager` отвечает за взаимодействие с пользователем системы.

Классы `app.AppDebugger` и `debugger.Debugger` композируют остальные классы системы и их поведение далее будет описано и реализовано в виде системы взаимодействующих автоматов.

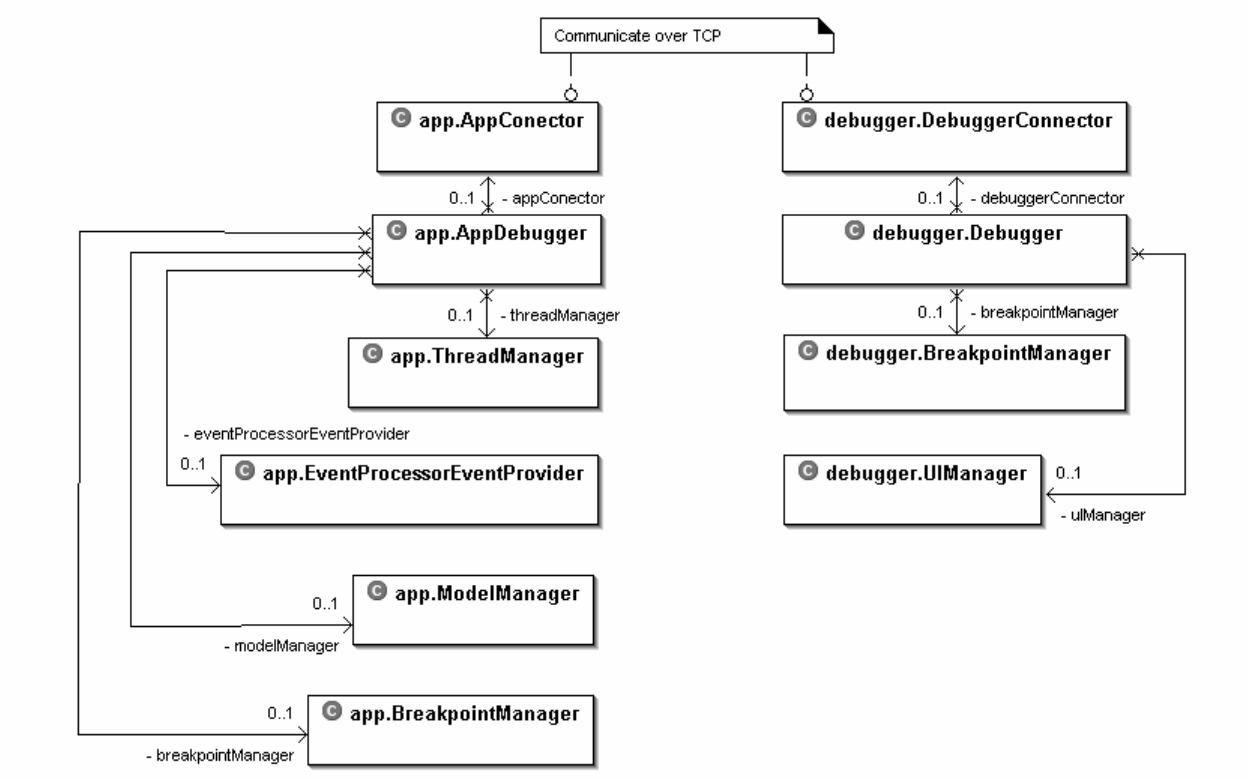

Рис. 19. Статическая модель системы

На рис. 20 показана модель сообщений, которыми обмениваются клиент и сервер. Класс `EventMessage` представляет сообщения, которые посылает компонент `UniMod Client-side Debugger`, а класс

`CommandMessage` – сообщения посылаемые компонентом `UniMod Server-side Debugger`. Внутри этих классов показаны константы, которые соответствуют типам сообщений, определенным в таблицах 1 и 2.

Класс `Position` определяет точку останова в отлаживаемой модели.

Класс `CommandMessage` имеет ассоциацию с классом `Position` для пересылки точек останова, установленных пользователем в отлаживаемую модель. Класс `EventMessage` имеет ассоциацию с классом `Position` для извещения пользователя о достигнутых токах останова.

Класс `CommandMessage` также имеет ассоциацию с классом `Model` для пересылки в отлаживаемую модель новой версии модели.

Интерфейс `MessageCoder` определяет методы для кодирования сообщения в массив байт для пересылке по протоколу TCP и для декодирования сообщения из массива байт.

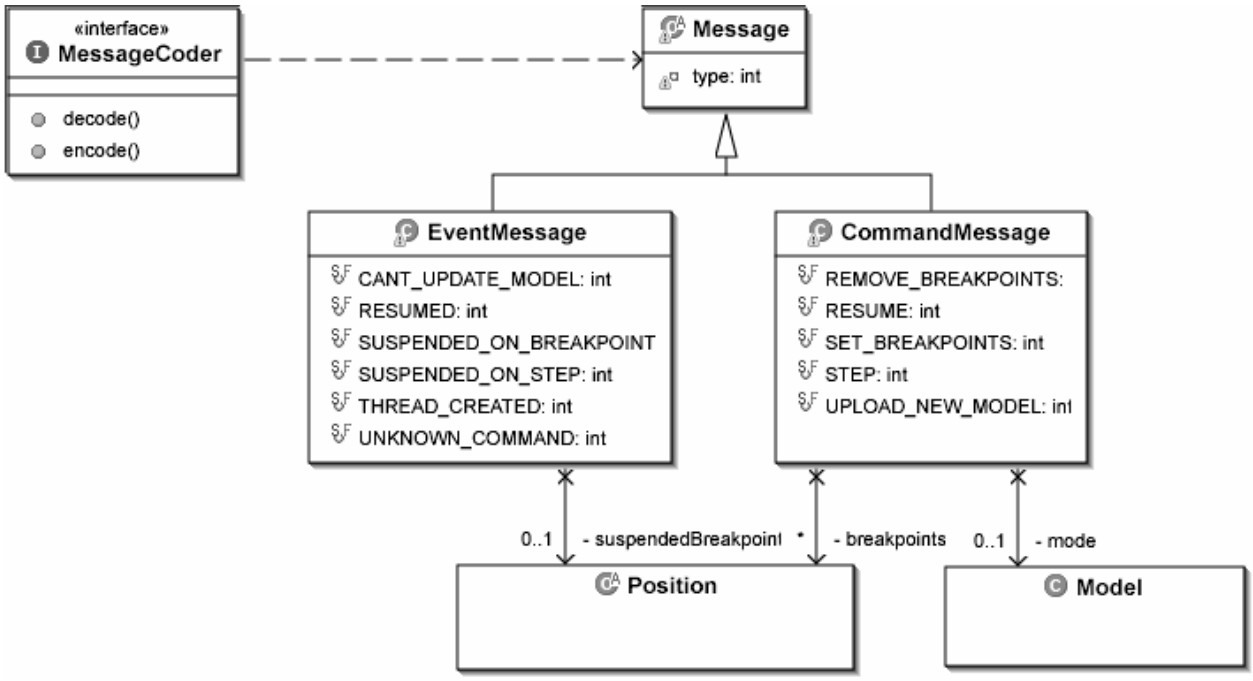

Рис. 20. Модель сообщений

4.4.2. Динамическая модель отладчика

Динамическая модель отладчика декомпозирована на две модели – серверную (реализует поведение компонента `UniMod Client-side Debugger`) и клиентскую (реализует поведение компонента `UniMod Server-side Debugger`).

На рис. 21 представлена схема связей автоматов серверной части системы. При этом классу статической модели `app.AppDebugger` соответствуют три автомата `app`, `A2` *и* `A3`.

Класс `app.AppConnector` на рис. 21 играет роль и источника событий и объекта управления. Это вызвано тем, что при получении сообщения от клиента `app.AppConnector` извещает об этом автомат с помощью посылки события. При необходимости отправки сообщения клиенту автомат вызывает методы объекта управления `app.AppConnector`.

На рис. 22 представлена диаграмма переходов автомата `app`. Этот автомат реализует поведение серверной части системы в целом.

На рис. 23 представлена диаграмма переходов автомата `A2`, отвечающего за управление точками останова.

На рис. 24 представлена диаграмма переходов автомата `A3`, управляющего потоками выполнения.

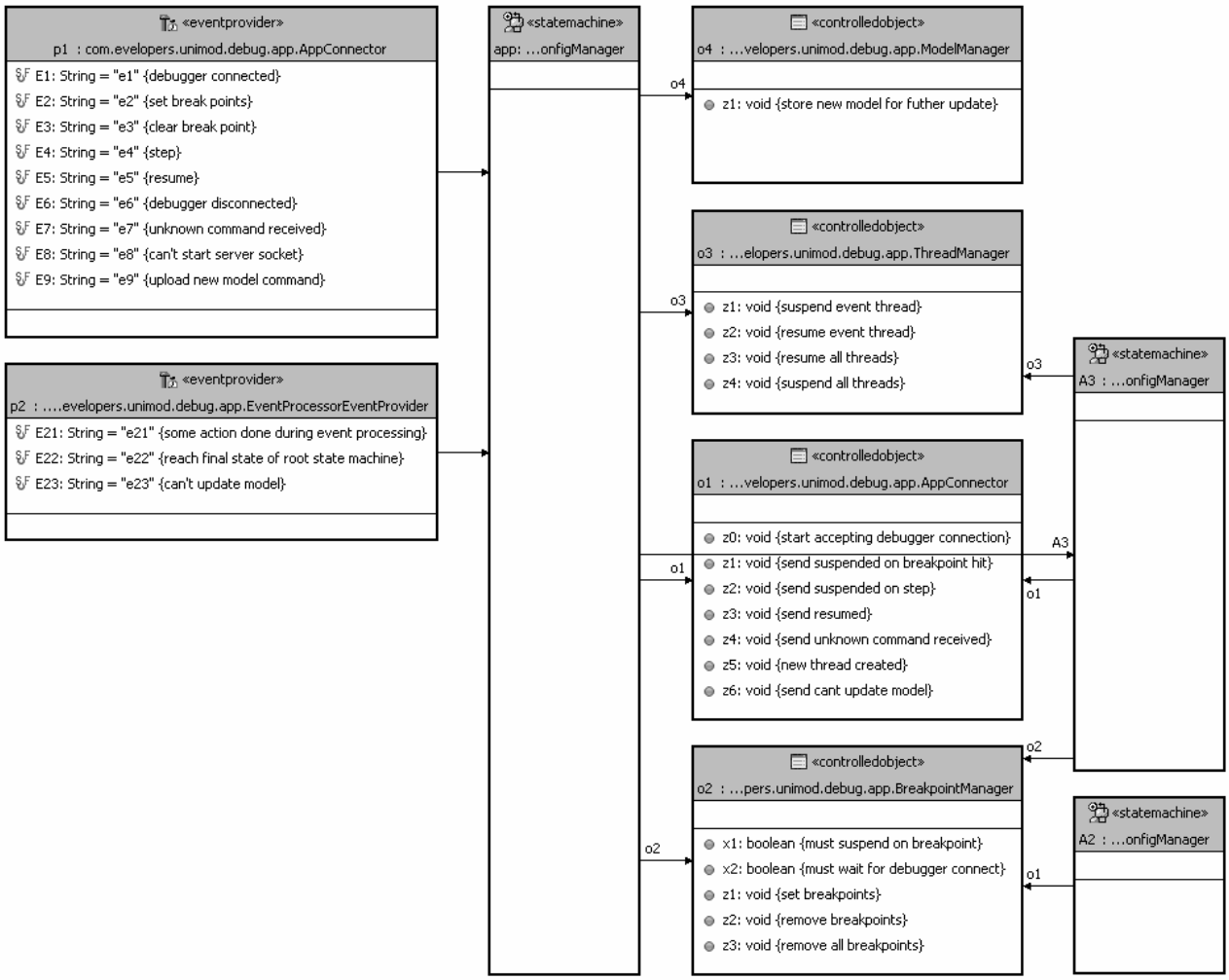

Рис. 21. Схема связей автоматов серверной части системы

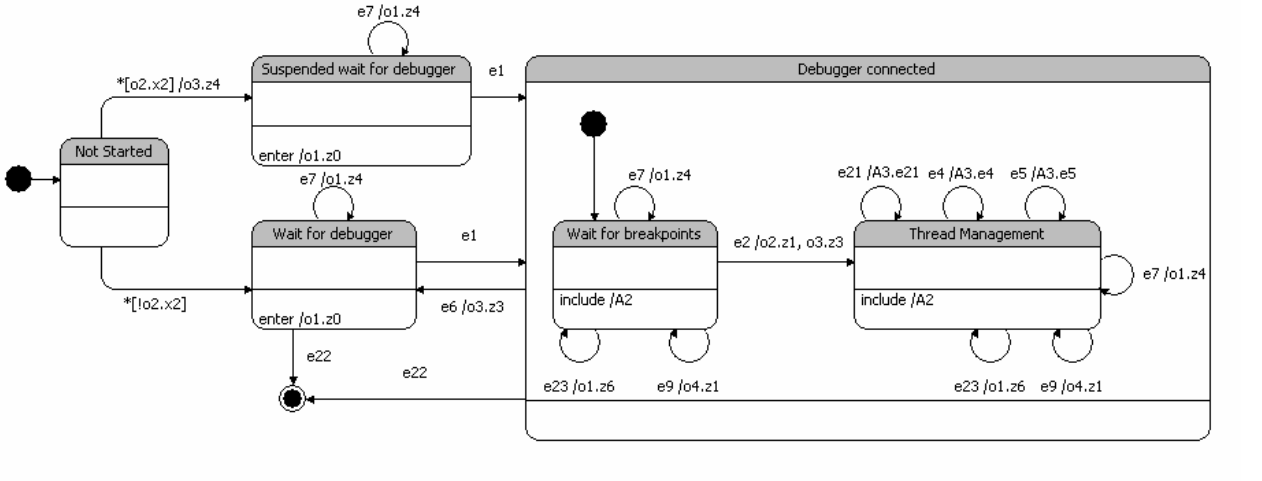

Рис. 22 Диаграмма переходов автомата app

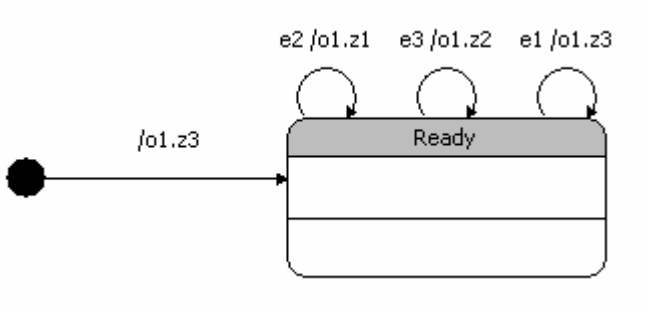

Рис. 23 Диаграмма переходов автомата A2

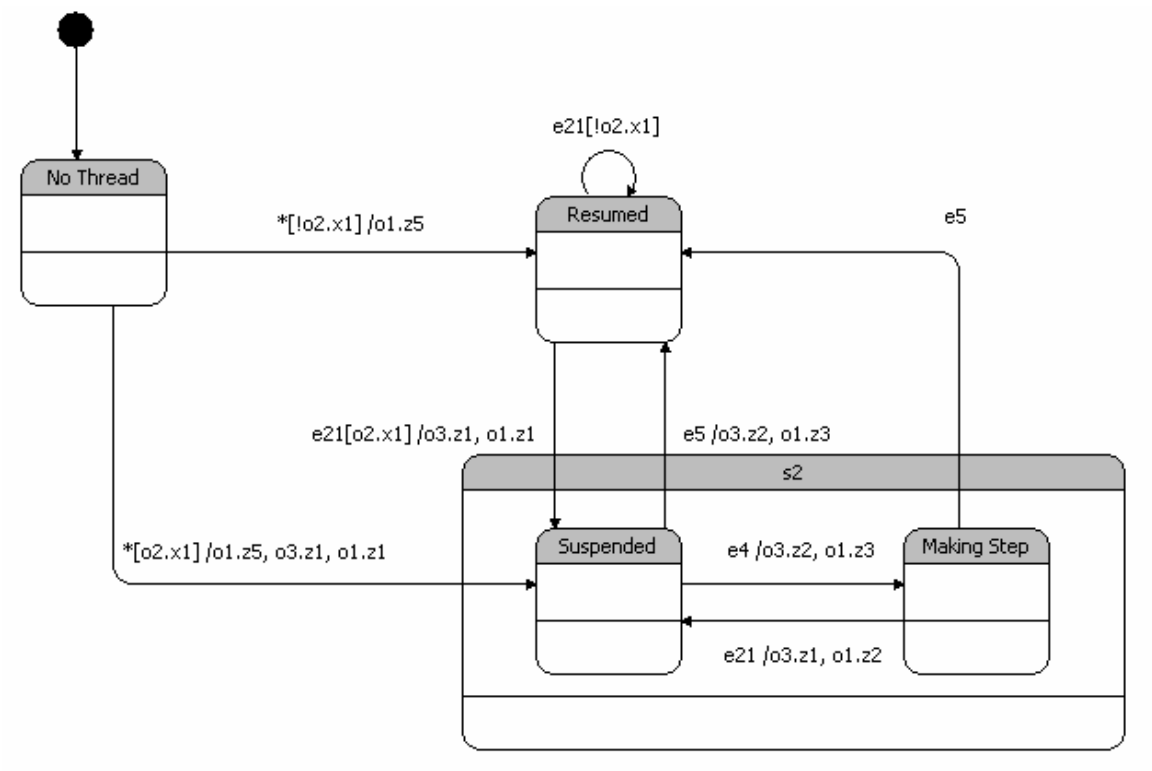

Рис.24. Диаграмма переходов автомата A3

На рис. 25 представлена схема связей автоматов клиентской части системы. Классу статической модели `debugger.Debugger` соответствуют три автомата `debugger`, `A2` и `A3`.

Класс `debugger.DebuggerConnector` на рис. 25, также как класс app.AppConnector, играет роль и источника событий и объекта управления.

Класс `debugger.UManager` также является и источником событий и объектом управления, так как он, с одной стороны, поставляет автомату события от пользователя, а с другой – позволяет автомату управлять пользовательским интерфейсом отладчика.

Класс `debugger.BreakpointManager`, как источник событий, извещает автомат о новых точках останова и об удалении старых. Как объект управления этот же класс позволяет автомату получить информацию о всех точках останова, установленных в отлаживаемой модели.

На рис. 26 представлена диаграмма переходов автомата `debugger`. Этот автомат реализует поведение клиентской части системы в целом.

На рис. 27 представлена диаграмма переходов автомата `A2`, отвечающего за управление точками останова.

На рис. 28 представлена диаграмма переходов автомата `A3`, управляющего потоками выполнения.

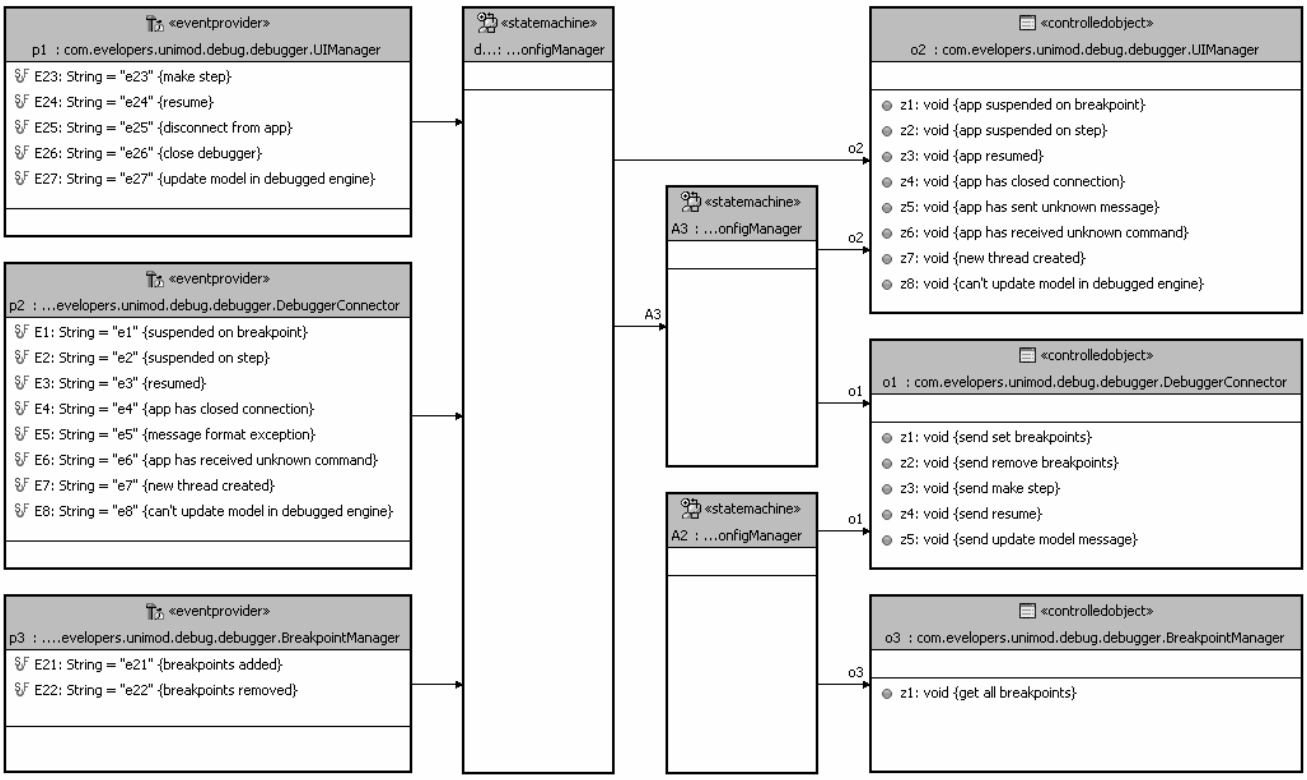

Рис. 25. Схема связей автоматов клиентской части системы

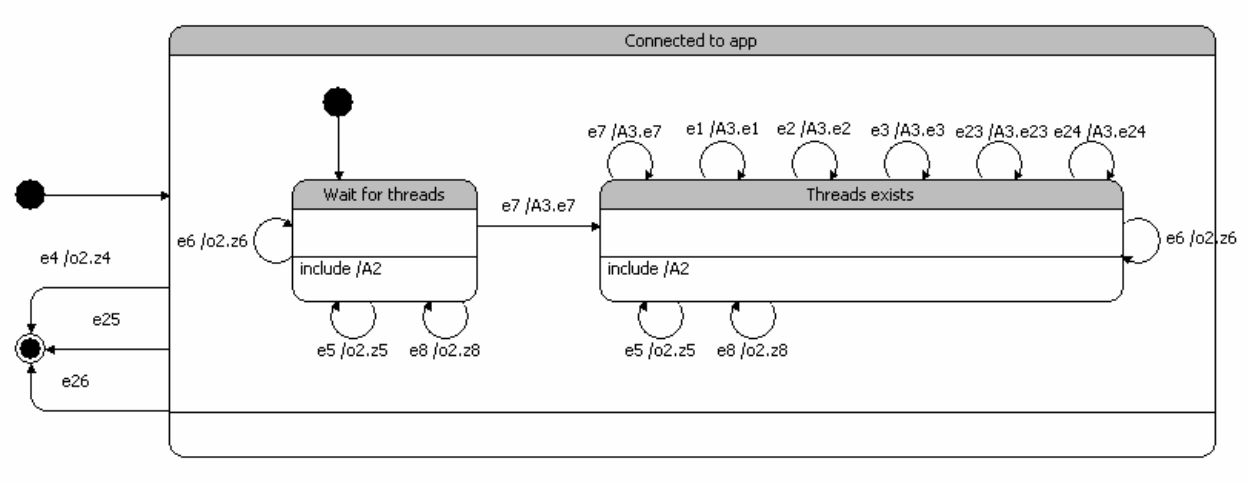

Рис. 26. Диаграмма переходов автомата `debugger`

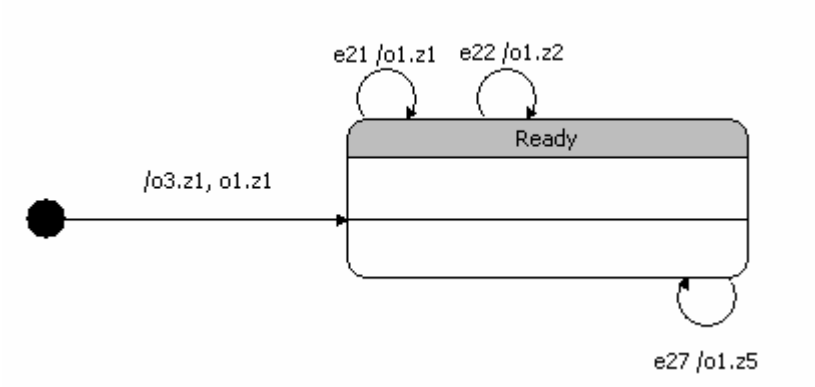

Рис. 27. Диаграмма переходов автомата `A2`

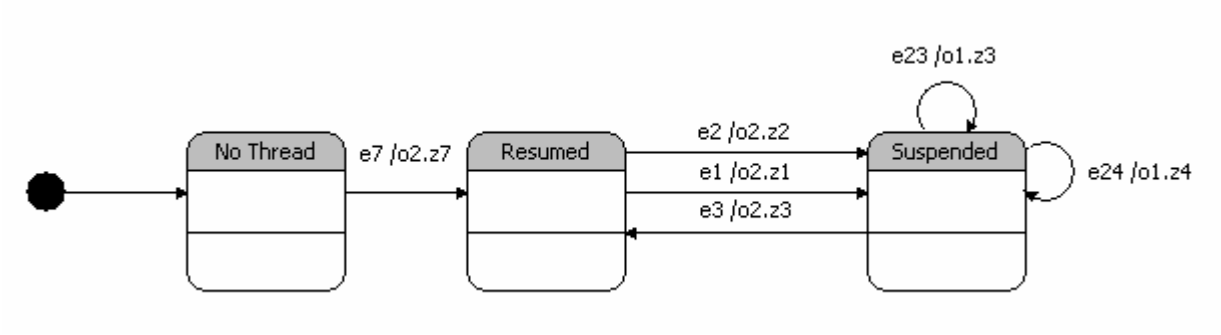

Рис. 28. Диаграмма переходов автомата `A3`

По автоматной модели системы сгенерировано ее *XML*-описание, которое исполняется интерпретатором. Источники событий и объекты управления реализованы вручную.

Выводы по главе 4

1. На основе платформы *Eclipse* разработано инструментальное средство для поддержки методов проектирования и верификации объектно-ориентированных автоматных программ.
2. Инструментальное средство поддерживает два подхода к запуску программ: интерпретационный и на основе генерации кода и компиляции.
3. При вводе пометок на переходах автомата инструментальное средство поддерживает автоматическое завершение ввода и исправление ошибок ввода.
4. Инструментальное средство позволяет производить отладку диаграмм состояний в терминах автоматов. Отладчик диаграмм реализован с помощью самого разработанного инструментального средства, что свидетельствует о высокой технической зрелости этого средства.

ГЛАВА 5. ВНЕДРЕНИЕ ПРЕДЛОЖЕННЫХ РЕЗУЛЬТАТОВ РАБОТЫ В ПРАКТИКУ ПРОЕКТИРОВАНИЯ

Методы и инструментальное средство *UniMod*, предложенные в данной работе, использовались:

- при разработке этого средства;
- в учебном процессе при выполнении студенческих курсовых проектов на кафедре «Компьютерных технологий» в СПбГУ ИТМО (http://is.ifmo.ru в разделе *UniMod*-проекты);
- в компании *eVelopers* (http://www.evelopers.com) для разработки интернет-приложений и мобильных приложений;
- в компании *JetBrains* (http://www.jetbrains.com) при разработке системы мета-программирования *Meta Programming System (MPS)*.

5.1. Создание системы автоматического завершения ввода

В рамках создания очередной версии пакета *UniMod* встала задача реализации системы автоматического завершения ввода при редактировании условий на переходах на *UML*-диаграмме состояний.

Автоматическим завершением ввода, применительно к редактированию программных текстов, традиционно называют технологию, позволяющую пользователю получить список строк, при добавлении которых в текст после позиции курсора, программа будет синтаксически верна.

Например, на рис. 29 показано, как среда разработки *Eclipse* предлагает варианты автоматического завершения ввода для текущей позиции курсора.

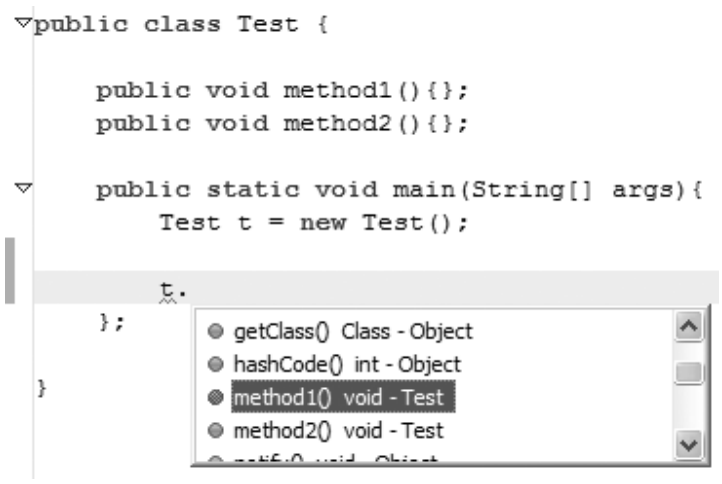

Рис. 29. Пример автоматического завершения ввода

Сформулируем требования к проектируемой системе автоматического завершения ввода. Пусть задан язык L и на вход системы подана строка α.

1. Если поданная на вход строка α является префиксом предложения языка L $(\exists\omega:\alpha\omega\in L)$, то система должна возвращать множество строк $C(\alpha)=\{\beta_i\}_{i=1..n}$, любая из которых может являться продолжением данной α $(\forall i\in[1..n]\exists\gamma:\alpha\beta_i\gamma\in L)$;
2. Если поданная на вход строка α не является префиксом предложения на заданном языке $(\neg\exists\omega:\alpha\omega\in L)$, то система должна с помощью дополнения строки недостающими символами или с помощью удаления лишних символов трансформировать строку в правильный префикс предложения языка. Число дополнений и удалений должно быть как можно меньше.

5.1.1. Описание предлагаемой технологии

Если исходный язык L задан формальной порождающей грамматикой, то для построения такой системы необходимо использовать методы проектирования компиляторов [48]. Известны инструменты для автоматического создания компиляторов по заданной грамматике (http://www.kulichki.net/kit/tools/java.html).

На рис. 30 приведена обобщенная структура компилятора.

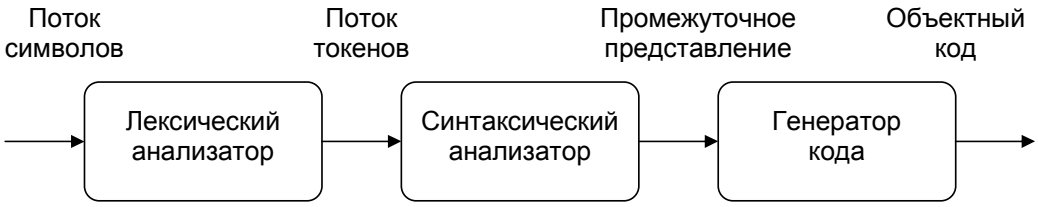

Рис. 30. Обобщенная структура компилятора

В проекте *UniMod* трансляция выражений на переходах выполняется с помощью, так называемого, «компилятора компиляторов» *ANTRL* [95]. Он по заданной *LL(k)*-грамматике строит код на языке *Java*, реализующий лексический анализатор и рекурсивный нисходящий синтаксический анализатор. Построенный синтаксический анализатор может быть использован и как распознаватель принадлежности выражения заданному грамматикой языку, и как транслятор выражений в абстрактное синтаксическое дерево. Данный анализатор не может быть использован для построения системы автоматического завершения ввода, так как в случае подачи ему на вход префикса для выражения на заданном языке вместо законченного выражения, он выдает ошибку.

Одним из возможных вариантов реализации требуемой системы может быть использование нерекурсивного нисходящего синтаксического анализатора, явно использующего стек и управляемого таблицей разбора. Таблица разбора представляет собой двумерный массив $M[A,a]$, где A – нетерминал, а a – терминал (лексема) или символ конца потока *$*. В ячейках таблицы записываются правила грамматики, с помощью которых заменяются нетерминалы на вершине стека, а пустые ячейки таблицы соответствуют ошибкам. Подробно работа такого анализатора описана в работе [48].

При подаче на вход описанному выше анализатору незавершенной строки α без символа конца потока, анализатор остановится, имея какой-то нетерминал на вершине стека. В этом случае множество терминалов $C(\alpha)$, ожидаемых вслед за обработанной строкой, может быть определено как

$\{\beta : M[T,\beta] \neq \varnothing\}$, где T – нетерминал на вершине стека после остановки анализатора.

Для реализации восстановления после ошибок в «режиме паники», таблица разбора может быть дополнена синхронизирующими символами, которые вписываются в некоторые пустые ячейки. При получении неожидаемого терминала, анализатор пропускает символы входного потока до тех пор, пока не будет обнаружен терминал, соответствующий синхронизирующему символу. Для восстановления на уровне фразы в некоторые пустые ячейки вписываются указатели на подпрограммы обработки ошибок, которые могут изменять, вставлять или удалять терминалы входного потока или элементы стека.

В работе [105] показано как создать программу нерекурсивного нисходящего синтаксического анализатора, используя автоматно-ориентированный подход. Таблица разбора в этой работе оставлена и выступает в роли объекта управления.

Далее описывается технология создания системы автоматического завершения ввода, позволяющая исключить таблицу разбора нисходящего нерекурсивного синтаксического анализатора и использующая гибкий алгоритм восстановления после ошибок на уровне фразы.

Технология основывается на том, чтобы для заданной *LL(1)*-грамматики построить конечный автомат типа Мили, который будет являться синтаксическим анализатором. Автомат должен реагировать на события, которые поставляет ему лексический анализатор. Каждому событию соответствует терминал. В работах [48, 106] приведено описание подхода для создания нисходящего синтаксического анализатора на основе диаграмм переходов. При этом предлагается записывать по одной диаграмме для каждого правила вывода грамматики. Описываемая в настоящей работе технология предлагает сворачивать все диаграммы в одну, при необходимости удаляя рекурсию с помощью метода, описанного в работе [107]. Такой подход позволяет избавиться от упоминания нетерминалов на

диаграммах переходов и, следовательно, разорвать семантическую связь с исходной грамматикой. Такой разрыв позволит описывать язык только с помощью диаграммы переходов и автоматически получать реализацию распознавателя для данного языка.

При подаче на вход системе, построенной описанным выше образом, незавершенной строки, автомат, реализующий синтаксический анализатор, останавливается в каком-то состоянии. События, заданные на переходах из состояния, в котором остановился автомат, определяют множество терминалов, которые могут следовать за последним терминалом, извлеченным из входной строки. После построения такого множества терминалов, каждый терминал преобразуется в строку символов.

5.1.2. Построение диаграммы переходов синтаксического анализатора

Пусть *LL(1)*-грамматика для нашего примера задана следующим множеством правил вывода:

1) `S -> else | T S'`
2) `S' -> or T S' | ε`
3) `T -> L T'`
4) `T' -> and L T' | ε`
5) `L -> not L | P`
6) `P -> '(' S ')' | int rel N | bool | N P'`
7) `P' -> rel int | ε`
8) `N -> id dot id`

Терминал **id** соответствует идентификатору, терминал **int** – целочисленной константе, терминал **bool** – булевской константе, а терминал **rel** – бинарному отношению (‘>’, ‘<’, ‘>=’, ‘<=’, ‘=’, ‘≠’). Опишем формальный процесс построения автомата типа Мили для данной грамматики.

На рис. 31 приведены диаграммы переходов для каждого нетерминала заданной грамматики, построенные с помощью метода, описанного в работе [48]. Единственным отличием указанных диаграмм, от диаграмм предлагаемых указанной работе, является наличие выделенных начального и

конечного состояний, отображаемых закрашенным кругом и закрашенным кругом внутри окружности соответственно. Из начального состояния существует всегда только один переход, в конечное состояние также ведет только один переход.

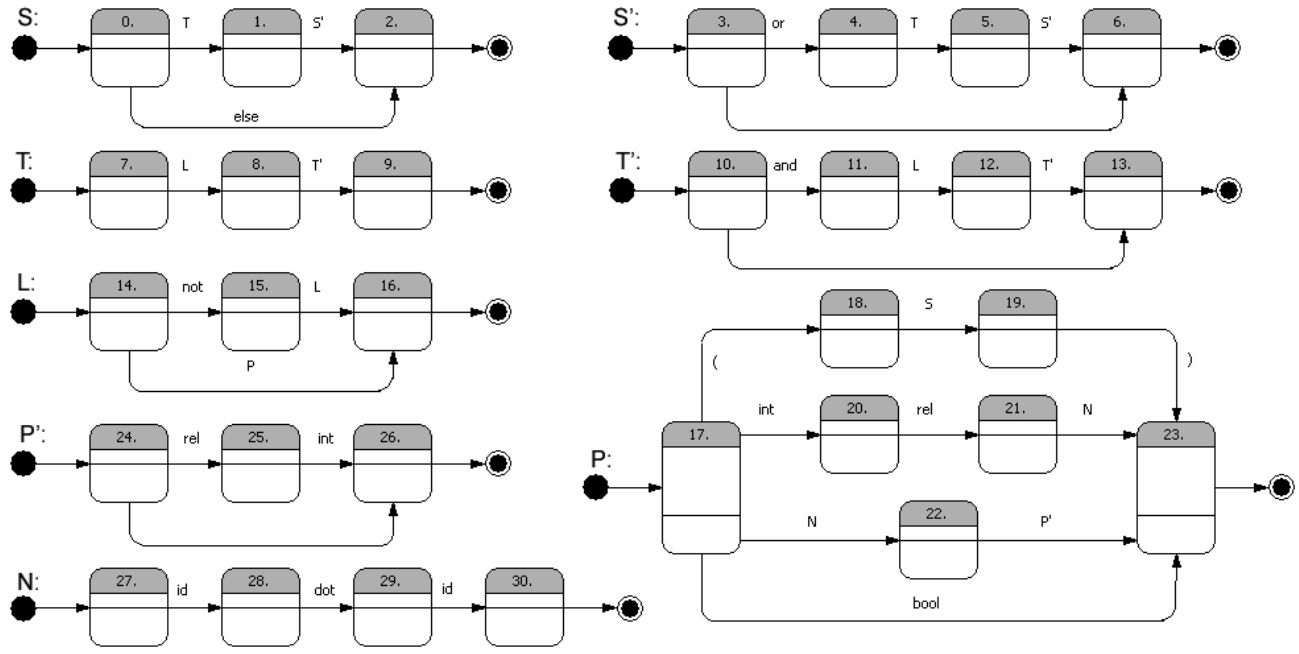

Рис. 31. Диаграммы переходов для каждого нетерминала грамматики

Состояния соответствуют позициям [49] в правилах вывода, а метки на переходах – терминалам и нетерминалам, отделяющим позиции друг от друга. Если нетерминал выводит ε-правило, то из состояния, соответствующего начальной позиции, существует непомеченный переход в состояние, соответствующее конечной позиции. Непомеченные переходы также называются немотивированными.

Далее множество диаграмм, представленных на рис. 31, можно преобразовать в одну диаграмму состояний, на которой все переходы будут помечены только терминалами. Процесс такого преобразования предполагает выполнение следующих шагов:

1. Удаление правой рекурсии.
2. Удаление немотивированных переходов.
3. Подстановка диаграмм переходов друг в друга.
4. Удаление срединной рекурсии.

Опишем каждый шаг подробно.

5.1.3. Удаление правой рекурсии

Наличие праворекурсивного правила вывода, означает, что на диаграмме, соответствующей некоторому терминалу *N*, есть переход, помеченный тем же нетерминалом *N*, ведущий в состояние, соответствующее конечной позиции.

Например, наличие праворекурсивного правила (2) влечет наличие перехода из состояния *5* в состояние *6* на рис. 31. Для устранения правой рекурсии этот переход должен быть заменен немотивированным переходом в состояние, соответствующее начальной позиции – в состояние *3* (рис. 32).

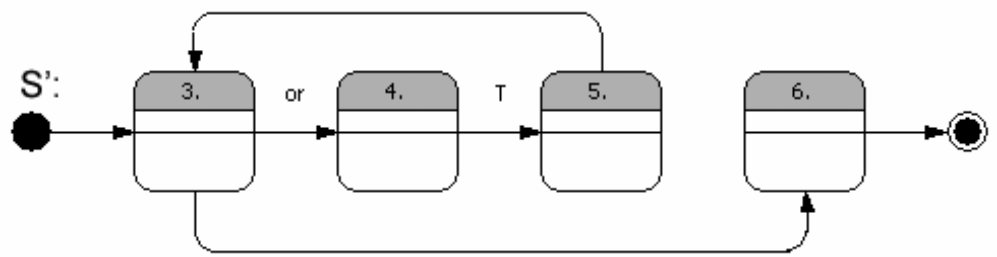

Рис. 32. Удаление правой рекурсии на диаграмме состояний для нетерминала S'

5.1.4. Удаление немотивированных переходов

Наличие немотивированного перехода из состояния S_1 в состояние S_2, означает, что за позицией, соответствующей состоянию S_1, могут следовать те же терминалы и нетерминалы, что и за позицией, соответствующей состоянию S_2.

Для устранения немотивированного перехода выполняются следующие операции:

1. Создать сложное состояние $S_{1,2}$.
2. Поместить состояния S_1 и S_2 внутрь состояния $S_{1,2.}$
3. Все переходы из состояния S_2 заменить аналогичными переходами из состояния $S_{1,2}$.

На рис. 33 показано удаление немотивированного перехода из состояния *5* в состояние *3*, присутствующего на рис. 32.

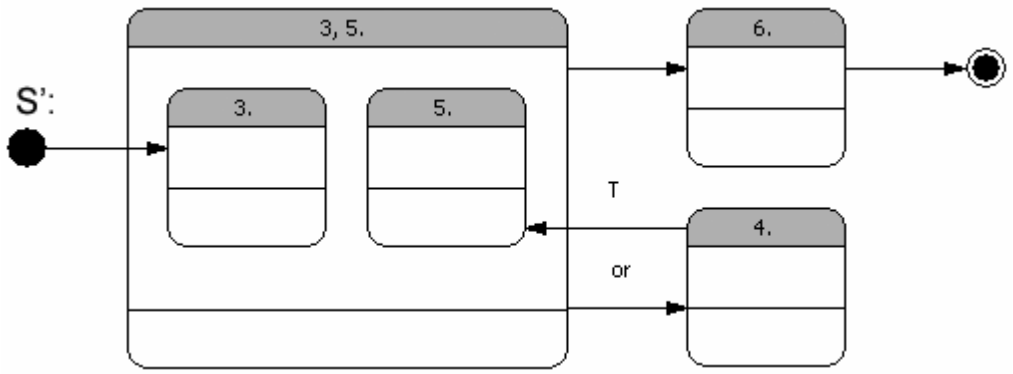

Рис. 33. Удаление немотивированного перехода из состояния 5 в состояние 3

Будем говорить, что исходящий переход из состояния S_1 совпадает с исходящим переходом из состояния S_2, если он помечен тем же символом и ведет в тоже состояние.

После выделения группового состояния $S_{1,2}$ множества переходов, исходящих из состояний S_1 и S_2 могут совпасть. В этом случае из каждой пары совпадающих переходов следует оставить только один и его начало переместить в состояние $S_{1,2}$. Все переходы, входящие в состояния S_1 и S_2, перенаправить в состояние $S_{1,2}$. Состояния S_1 и S_2 ликвидировать.

Отметим, что данный алгоритм аналогичен вычеркиванию одинаковых записей из таблицы, задающей функцию переходов автомата [50].

Для диаграммы состояний (рис. 33) описанный алгоритм можно применить для состояний *3* и *5*. Результирующая диаграмма показана на рис. 34.

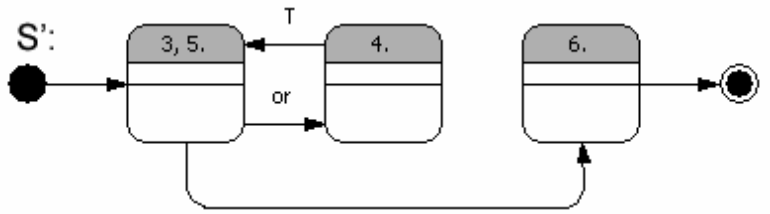

Рис. 34. Упрощение диаграммы состояний, показанной на рис. 33

На рис. 35 показано удаление немотивированного перехода из состояния *3, 5* в состояние *6*.

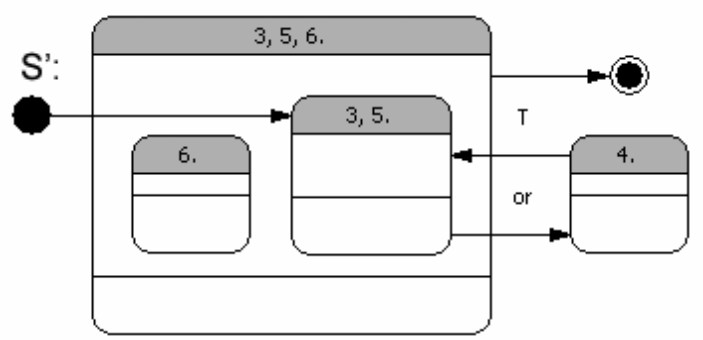

Рис. 35. Удаление немотивированного перехода из состояния 3,5 в состояние 6

Состояние *6* на рис. 35 не имеет входящих переходов и, следовательно, не достижимо. Поэтому оно может быть удалено. После удаления состояния *6* в состояние *3,5,6* будет вложено единственное состояние *3,5*. Начала переходов, исходящих из состояния *3,5,6* следует перенести в состояние *3,5*, а само состояние *3,5,6* удалить. На рис. 36 приведена диаграмма состояний для нетерминала `S'` после всех описанных модификаций.

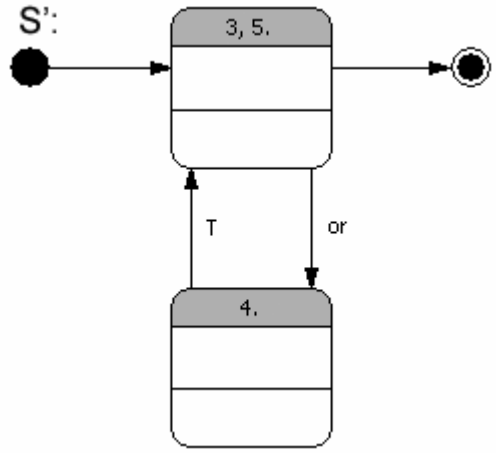

Рис. 36. Преобразованная диаграмма состояний для нетерминала S'

5.1.5. Подстановка диаграмм переходов друг в друга

Диаграммы переходов могут быть упрощены подстановкой одних диаграмм в другие. Предположим, что на диаграмме для нетерминала N_1 существует переход из состояния S_1 в состояние S_2, помеченный нетерминалом N_2. Заменим такой переход на немотивированный из состояния S_1 в состояние, следующее за начальным на диаграмме переходов для нетерминала N_2. Добавим переход из состояния, предшествующего

конечному, на диаграмме переходов для нетерминала N_2 в состояние S_2. Отметим, что указанную подстановку необходимо выполнять, только если $N_1 \neq N_2$, так как в противном случае имеет место срединная рекурсия, удаление которой будет описано ниже.

После выполнения такой подстановки, возникшие немотивированные переходы следует устранить описанным ранее способом. При этом сначала устраняется немотивированный переход из состояния S_1, а затем переход в состояние S_2.

На рис. 37 продемонстрирована подстановка диаграммы переходов для нетерминала L в диаграмму переходов для нетерминала T'. На рис. 38 приведена диаграмма переходов после устранения немотивированных переходов.

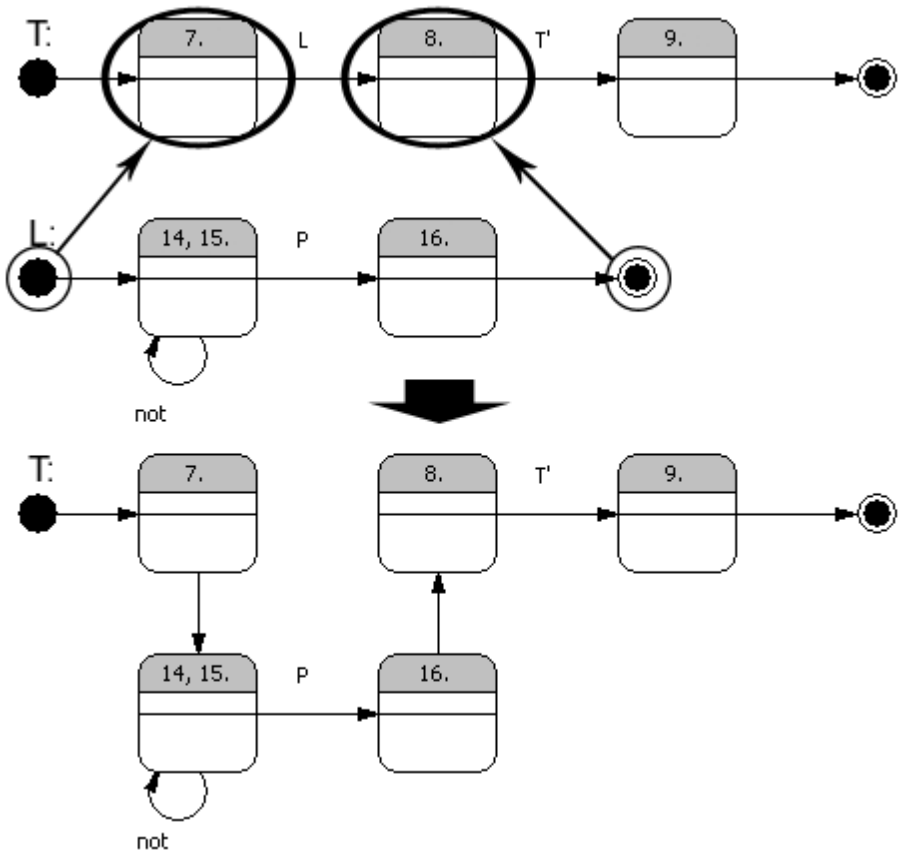

Рис. 37. Подстановка диаграмм переходов друг в друга

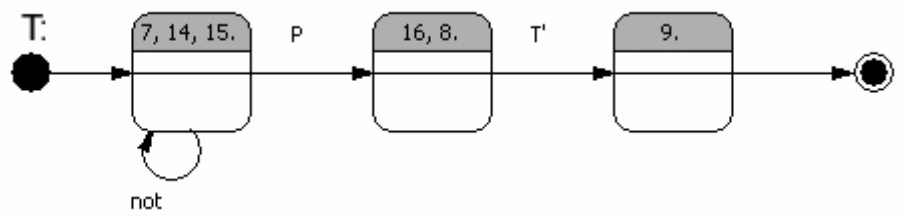

Рис. 38. Устранение немотивированных переходов после подстановки

Если некоторый нетерминал N_1 неоднократно присутствует на диаграмме для нетерминала N_2, то каждый переход, помеченный N_1, необходимо заменить соответствующей диаграммой переходов. В результате число однотипных подграфов на диаграмме N_2 чрезмерно возрастает.

Предлагается преобразовать диаграмму N_2 таким образом, чтобы нетерминал N_1 встречался на ней минимальное число раз. Для этого используется следующий метод: если несколько переходов помеченных нетерминалом N_1 ведут в одно и тоже состояние S, то можно выделить составное состояние, и заменить все эти переходы единственным переходом, помеченным нетеминалом $N_{1,}$, который исходит из группового состояния и входит в состояние S.

На рис. 39 данный метод применен для диаграммы переходов нетерминала S.

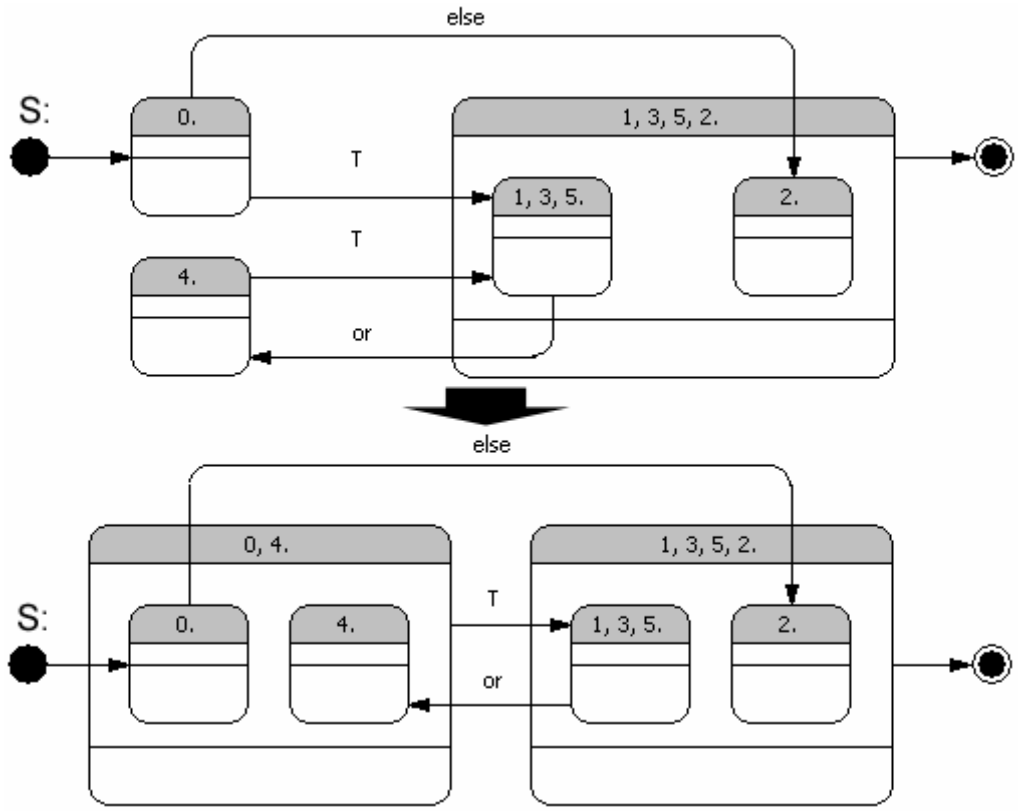

Рис. 39. Удаление двух переходов помеченных нетерминалом T на диаграмме переходов для нетерминала S

5.1.6. Удаление срединной рекурсии

В результате выполнения предыдущих шагов исходное множество диаграмм переходов преобразуется во множество диаграмм, возможно, имеющих срединную рекурсию. Для описанной выше грамматики исходные диаграммы, показанные на рис. 31, преобразуются в одну диаграмму, приведенную на рис. 40. На этой диаграмме присутствует срединная рекурсия – переход из состояния *18* в состояние *19*.

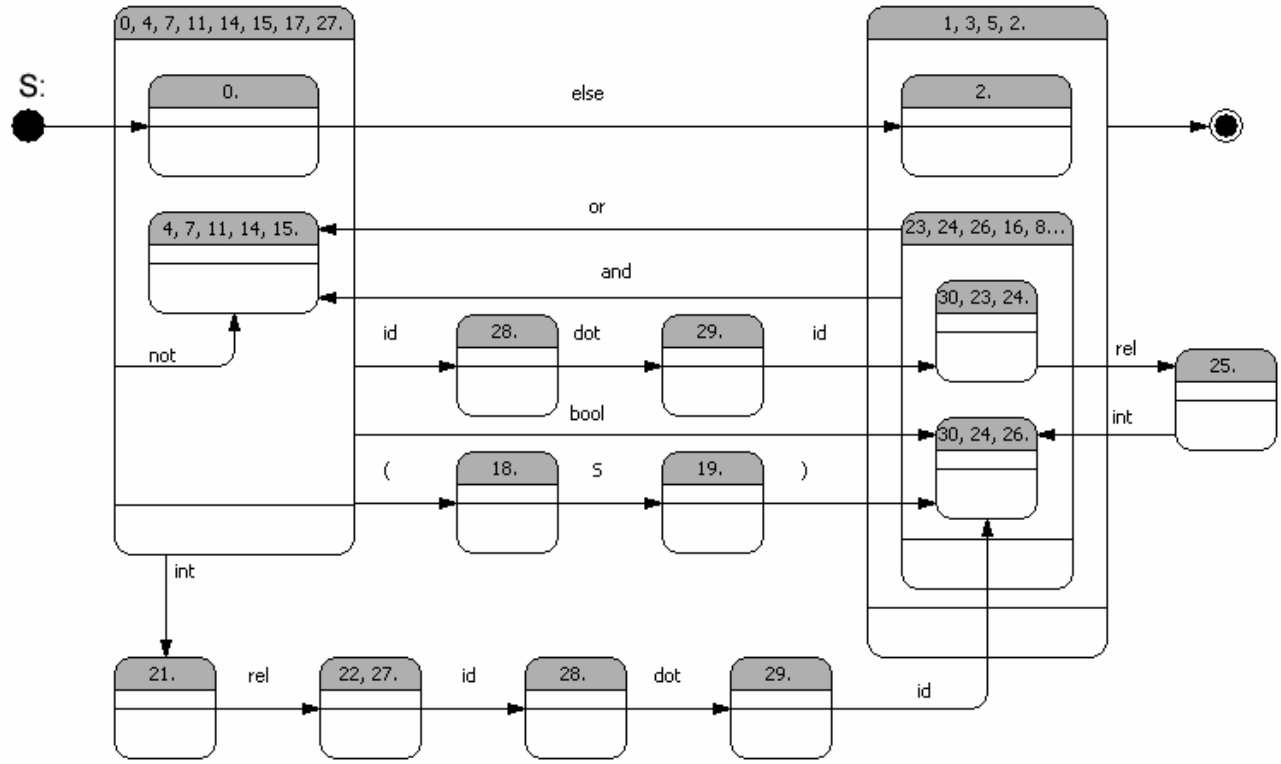

Рис. 40. Диаграмма состояний со срединной рекурсией

Для исключения переходов, образующих срединную рекурсию, предлагается использовать метод, предложенный в работе [107]:

- пусть на диаграмме переходов для нетерминала N существует рекурсивный переход из состояний S_1 в состояние S_2, помеченный нетерминалом N;
- заменим такой переход двумя немотивированными переходами. При этом первый из них должен выходить из состояния S_1, а входить в состояние, следующее за начальным состоянием на диаграмме. Второй переход должен выходить из состояния, предшествующего конечному, а входить в состояние S_2;
- на первом переходе должно выполняться действие по добавлению в стек метки M_{S2}, соответствующей исходному целевому состоянию S_2, а на втором переходе – действие по извлечению метки M_{S2} из стека, при условии, что метка M_{S2} находится на вершине стека.

На рис. 41 показана диаграмма, приведенная на рис. 40, с удаленной срединной рекурсией.

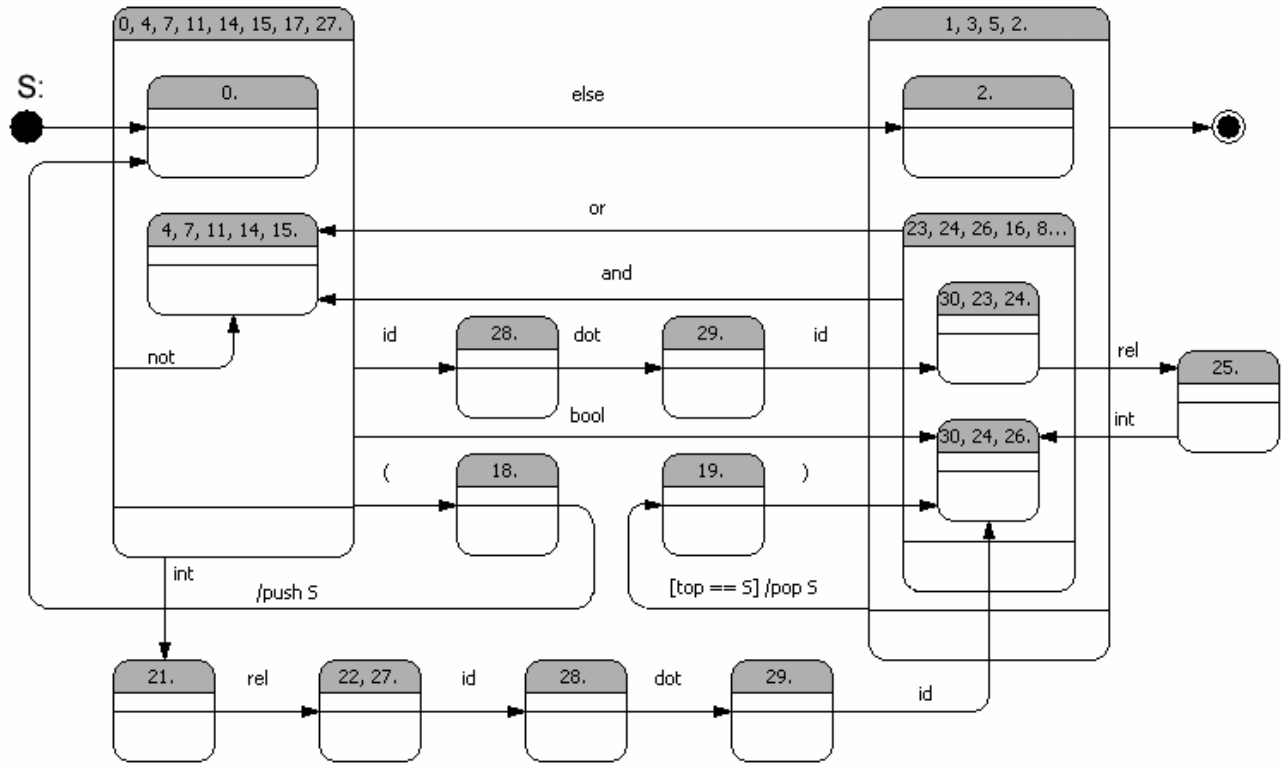

Рис. 41. Диаграмма состояний с удаленной срединной рекурсией

Отметим, что если преобразованное множество диаграмм состоит более чем из одной диаграммы, то после удаления срединной рекурсии в каждой из них, необходимо продолжить процесс преобразования с шага 3 – подстановка диаграмм переходов друг в друга.

5.1.7. Модель разрабатываемой системы

Описанные выше шаги приводят к построению диаграммы переходов для автомата типа Мили. Поставим в соответствие каждому терминалу событие, поставляемое лексическим анализатором, и создадим схему связей автомата в виде *UML*-диаграммы классов.

На рис. 42 приведена такая схема. При этом на ней слева показан объект *p1*, соответствующий лексическому анализатору, в центре – объект *A*, соответствующий автомату Мили, а справа – объект управления *o1*, соответствующий стеку.

На рис. 43 показана *UML*-диаграмма состояний автомата, построенная на основе диаграммы, приведенной на рис. 41, с помощью замены терминалов событиями и замены действий на переходах ссылками на методы объекта управления *o1*. Состояния 18 и 19 на рис. 43 отсутствуют из-за удаления немотивированных переходов.

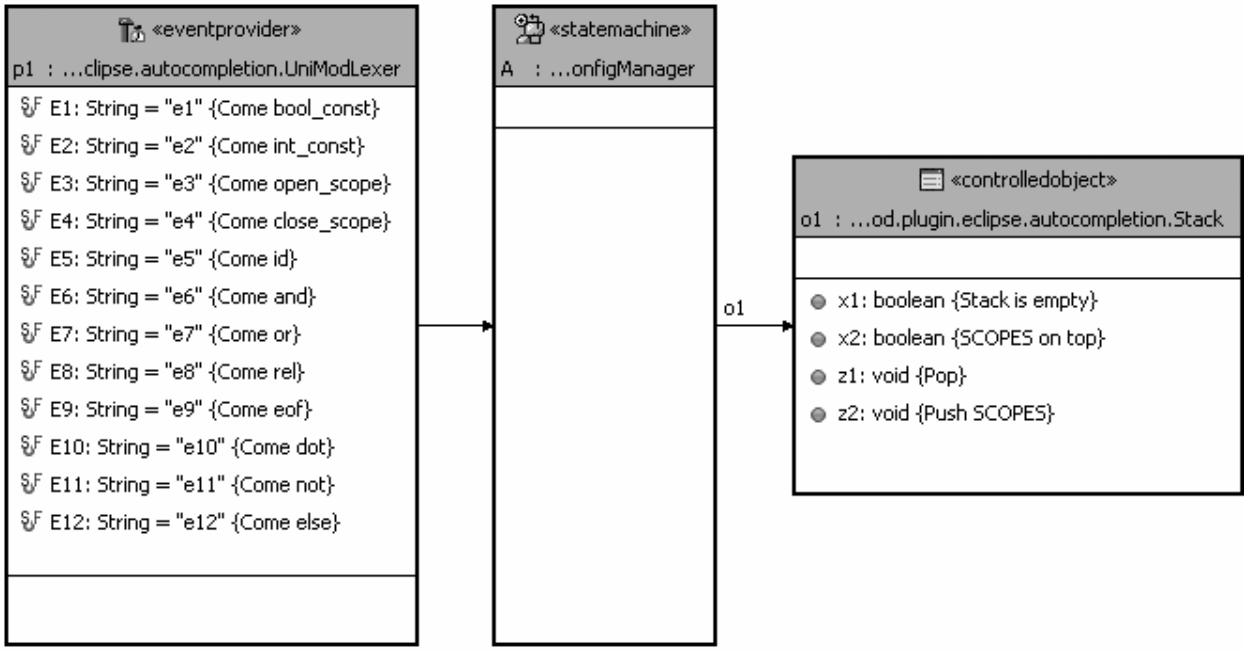

Рис. 42 Схема связей автомата

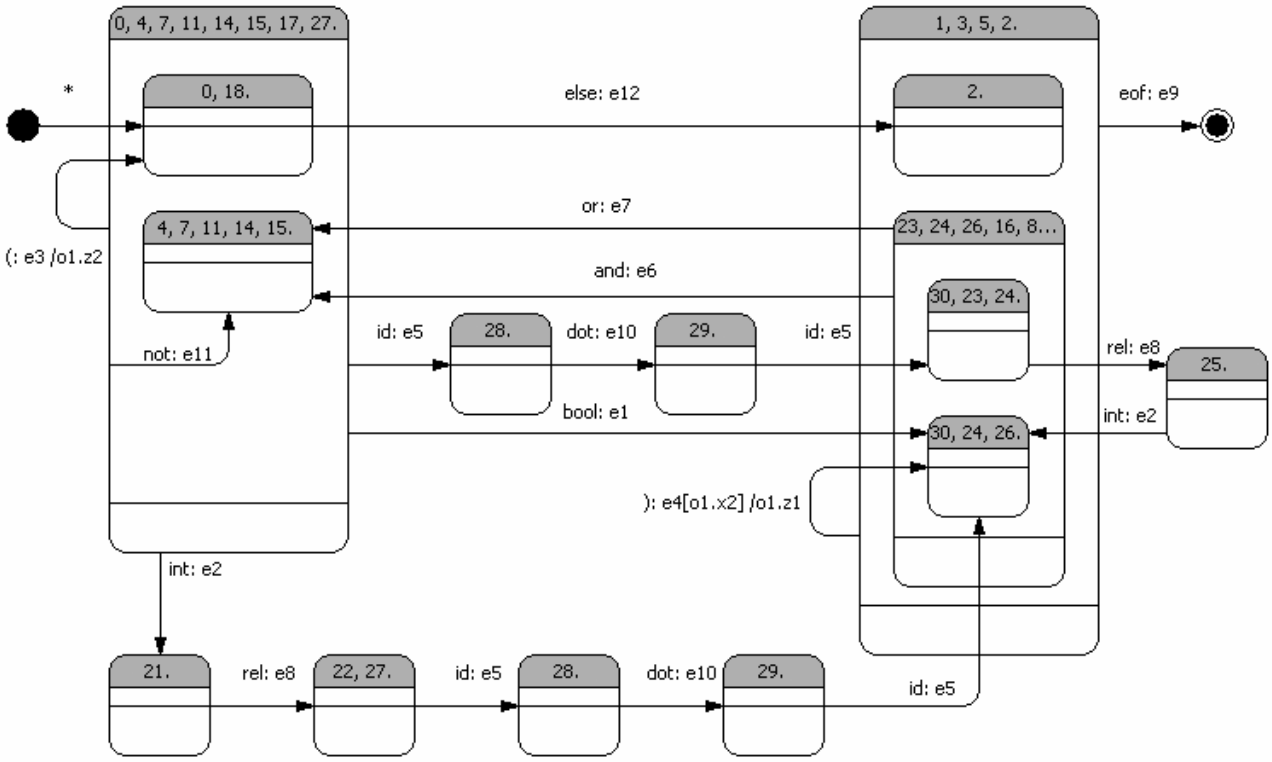

Рис. 43. Диаграмма состояний автомата

Полученная модель системы состоит из двух *UML*-диаграмм (рис. 42, 43) и описывает распознаватель для языка, заданного приведенной выше грамматикой. Отметим, что информация о приоритете операций была потеряна в ходе преобразований, и поэтому модель может быть использована для распознавания принадлежности выражений языку, но не для трансляции выражений.

Выражение, принадлежащее языку, поданное на вход распознавателю, приводит автомат *A* в финальное состояние. При подаче на вход выражения, являющегося префиксом какого-либо выражения принадлежащего языку, автомат остановится в каком-то состоянии, исходящие переходы из которого определяют возможные следующие терминалы.

Если выражение языку не принадлежит и не является префиксом какого-либо выражения, принадлежащего языку, то автомат *A* остановится в состоянии, в котором было получено событие, для которого не существовало исходящих переходов при текущих значениях входных переменных. В этом случае совокупность возможных следующих терминалов можно определить только для последнего правильно обработанного терминала.

5.1.8. Восстановление после ошибок

Перейдем к реализации второго, предъявляемого к системе, требования – обработка ошибочных строк. Для его реализации автомат *A* необходимо модифицировать таким образом, чтобы он корректно восстанавливался в случае подачи на вход выражения, не являющегося префиксом какого-либо выражения, принадлежащего языку. При этом автомат должен всегда останавливаться в состоянии, в которое существует переход по событию, соответствующему последнему терминалу, извлеченному из поданного на вход выражения.

Существует несколько возможных вариантов реализации восстановления автомата после ошибки [48]. Например, можно для каждого состояния добавить такой исходящий переход, который ведет в конечное

«ошибочное» состояние, что он будет срабатывать в случае отсутствия какого-либо другого исходящего перехода для пришедшего события и текущих значений входных переменных. Это приведет к тому, что при появлении в процессе распознавания первого же ошибочного терминала, автомата завершит работу в «ошибочном» состоянии. Однако из «ошибочного» состояния нет исходящих переходов, и, следовательно, множество возможных последующих терминалов будет пустым.

Предлагается использовать альтернативный алгоритм обработки ошибочных ситуаций, основанный на локально оптимальной коррекции входного потока терминалов от лексического анализатора. Такой подход также называют восстановлением на уровне фразы.

Пусть из состояния S нет исходящего перехода для пришедшего события e, соответствующего некоторому терминалу. Тогда коррекция потока может осуществляться двумя способами: дополнением потока недостающими терминалами; пропуском лишних терминалов в потоке.

Для того, чтобы автомат A, находясь в состоянии S, пропустил в потоке терминал, соответствующий пришедшему событию e, необходимо добавить в автомат петлю в состоянии S по событию e. Тогда, находясь в состоянии S и получив событие e, автомат останется в состоянии S – проигнорирует пришедшее событие, и, как, следствие, пропустит терминал в потоке.

Для того чтобы автомат A, находясь в состоянии S, дополнил поток недостающими терминалами, следует выполнить указанные ниже операции:

1. Найти достижимое из S состояние S_h, такое, что в нем существует исходящий переход по событию e. Если из состояния S достижимы несколько таких состояний, то выберем ближайшее из них.
2. Если из ближайшего найденного состояния S_h есть переход в некоторое состояние S_t по событию e при условии c, то необходимо добавить в автомат переход из состояния S в

состояние S_t по событию e при условии c. Отметим, что отсутствие условия трактуется как тождественная истина.

Последовательность терминалов, соответствующих событиям, которыми помечен кратчайший путь из состояния S в состояние S_h, можно использовать для вставки в поток перед терминалом, соответствующим пришедшему событию e.

Если лексический анализатор позволяет заглядывать на произвольное число терминалов вперед, то можно применять оба способа коррекции одновременно, выбирая оптимальный способ в процессе разбора.

Для выбора оптимального способа предлагается использовать следующее правило:

1. При получении ошибочного терминала в текущем состоянии вычислим число терминалов, которыми необходимо дополнить поток.
2. Вычислим число терминалов, которое необходимо пропустить в потоке, до следующего обрабатываемого в текущем состоянии терминала.
3. Выполним операцию, требуемое число терминалов для которой минимально.

Для реализации этого способа коррекции к автомату распознавателя в качестве объекта управления добавим лексический анализатор (рис. 44). Он предоставляет автомату распознавателя целочисленную входную переменную *o2.x1*. Ее значение равно числу терминалов, которые необходимо пропустить в потоке до следующего терминала, обрабатываемого в текущем состоянии. Если входной поток вообще не содержит терминалов, обрабатываемых в текущем состоянии, то значение переменной *o2.x1* больше любого наперед заданного целого числа.

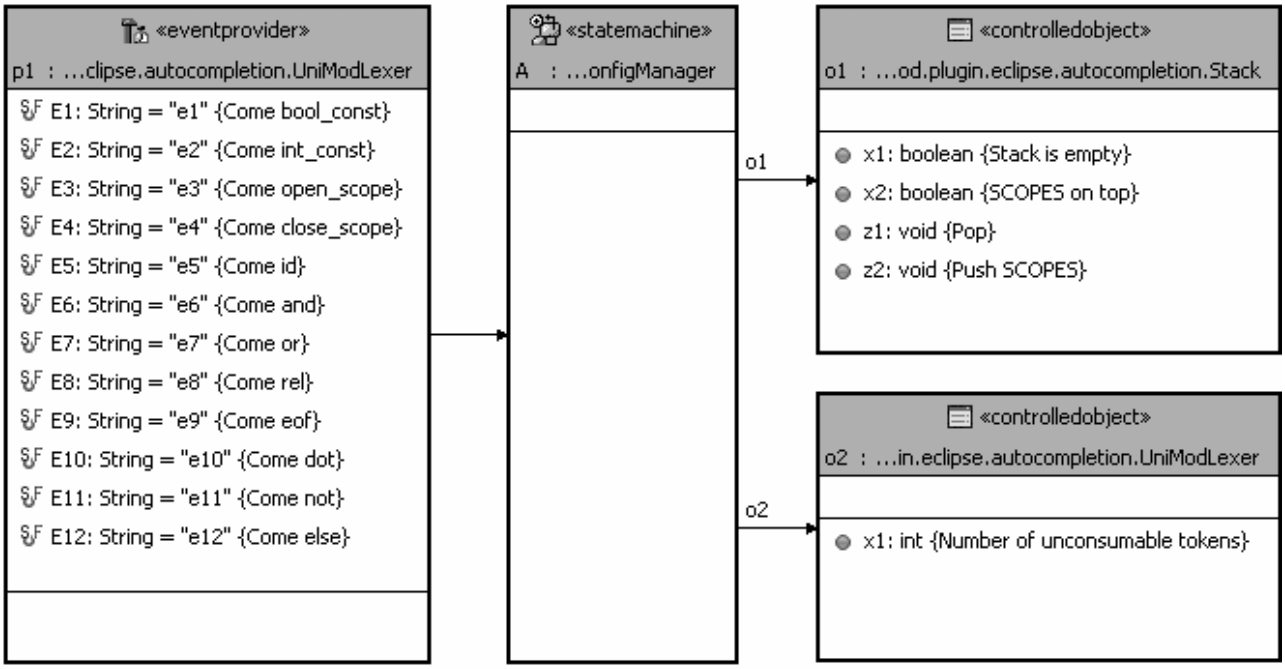

Рис. 44. Схема связей модели распознавателя с лексическим анализатором в качестве объекта управления

В автомат *A* добавляются переходы, реализующие и добавление и пропуск терминалов в потоке. Переходы, реализующие дополнение потока, помечаются следующим условием: длина пути из состояния *S* в состояние S_h меньше или равна значению входной переменной *o2.x1*. Переходы, реализующие пропуск терминалов, помечаются отрицанием того же условия. Если для состояния *S* не существует состояния S_h, то переход, удаляющий лексему, выполняется безусловно.

Например, в состоянии *21* нет переходов по событию *e10* – в этом состоянии появление во входном потоке терминала **`dot`** (точка) не ожидается. Для того чтобы обработать ошибочное появление этого терминала, необходимо добавить два перехода, исходящих из состояния *21* (рис. 45). Ближайшее состояние, в котором обрабатывается событие *e10* – состояние *28*. Длина пути из состояния *21* в состояние *28* равна двум. Поэтому условие на петле в состоянии *21* по событию *e10* имеет вид $o1.x1 < 2$. Таким образом, в случае, если сразу за терминалом **`dot`** в потоке следует терминал **`rel`** (отношение), то терминал **`dot`** игнорируется. Если следует какой-нибудь другой терминал, то целесообразно сразу перейти в

состояние *29* – добавить в поток отсутствующие терминалы **`rel`** и **`id`** (идентификатор).

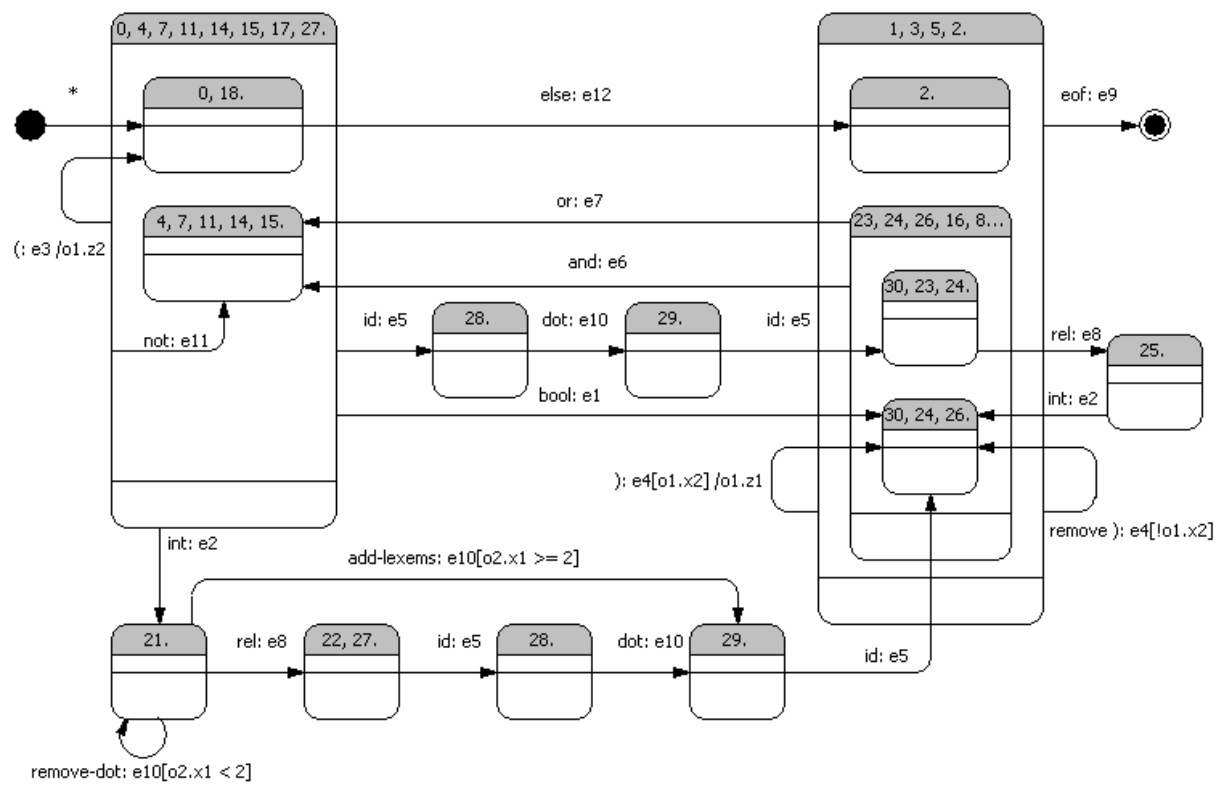

Рис. 45. Добавление переходов, корректирующих поток, в состоянии 21

Описанные выше преобразования могут быть выполнены автоматически для любой диаграммы переходов, так как ближайшее состояние, в котором обрабатывается неожидаемый терминал для данного состояния, может быть вычислено, используя, например, алгоритм Флойда-Уоршала [50].

5.1.9. Получение множества строк для автоматического завершения ввода

Предлагаемый алгоритм коррекции входного потока позволяет вычислить варианты завершения как для выражений, являющихся префиксами принадлежащих языку выражений, так и для ошибочных выражений.

После того как автомат распознавателя, дополненный корректирующими переходами, обработает все терминалы, извлеченные из поданного на вход выражения, он окажется в некотором состоянии *S*. Для

построения вариантов завершения следует определить переходы, исходящие из состояния *S*, условия на которых при текущих значениях входных переменных истинны. Терминалы, соответствующие событиям, которыми помечены эти переходы, должны быть преобразованы обратно во множество строк. Например, терминал **id** должен быть преобразован в имена переменных, а терминал **and** – в строку «&&». Полученная совокупность строк образует варианты завершения.

5.1.10. Пример работы системы

Приведем пример построения вариантов завершения.

Пусть на вход распознавателю подана строка:

```
! o1.x1 &&
```

Лексический анализатор преобразует ее в поток терминалов:

not id dot id and,

которому соответствует последовательность событий:

```
e11, e5, e10, e5, e6
```

В процессе обработки этих событий автомат изменяет состояния в следующем порядке:

```
(0,18)→(4,7,11,14,15)→(28)→(29)→(30,23,24)→(4,7,11,14,15)
```

Состояние `(4,7,11,14,15)`, в котором остановился автомат, содержит исходящие переходы для событий:

```
e1, e2, e3, e5, e11.
```

Этим событиям соответствуют терминалы:

```
bool, int, '(', id, not,
```

которые преобразуются в строки:

```
«true», «false», «(«, «o1», «o2», «o22», «!».
```

Эти строки и формируют варианты завершения для строки, поданной на вход распознавателю.

На рис. 46 показан фрагмент среды разработки с встроенной системой автоматического завершения ввода, описанной в настоящей статье.

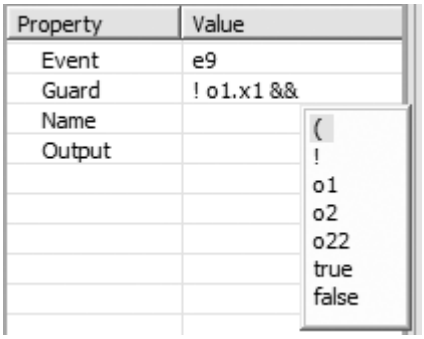

Рис. 46. Пример автоматического завершения ввода

В работе [48] отмечено, что нерекурсивные нисходящие синтаксические анализаторы можно строить, используя диаграммы переходов, записанные для каждого нетерминала исходной грамматики. Изложенный подход обеспечивает построение всего одной диаграммы переходов для исходной грамматики. На базе построенной диаграммы реализуется система автоматического завершения ввода. Также отметим, что в известной автору литературе, описание формального метода построения подобных систем отсутствует.

Реализация системы автоматического завершения ввода для следующей версии проекта *UniMod* выполнена с помощью предыдущей версии проекта. Поэтому часть проектной документация для этого средства была получена «автоматически», так как диаграммы, созданные с помощью *UniMod*-редактора, являются автоматными программами и могут быть включены в проектную документацию без изменений. Разработка последующих версий средств разработки с помощью предыдущих является общепринятой практикой и позволяет говорить о зрелости программного продукта.

5.2. Внедрение в учебном процессе

С помощью описанного метода построения автоматных программ и инструментального средства *UniMod* было реализовано 28 студенческих проектов. Среди них игры и банкоматы, стиральные машины и автомобильные сигнализации.

Полный список проектов приведен в табл. 3.

Таблица 3. Студенческие проекты, реализованные на основе *UniMod*

Название проекта	**Авторы**
Моделирование работы распределенной системы банкоматов	И. А. Балтийский, С. И. Гиндин
Моделирование работы электронной автомобильной сигнализации	А. А. Борисенко, Д. М. Пенкин
Моделирование работы системы интеллектуального здания	Г. М. Рыбаков
«Устройство» для карточной игры Блэкджек»	Р. В. Наумов, А. В. Якушев
Моделирование проезда нерегулируемых перекрёстков равнозначных дорог	А. С. Никитин, М. Ю. Чураков
Автоматный подход к моделированию эволюции микроорганизмов с использованием генетических алгоритмов	И. С. Гунич, А. В. Иринев
Имитация работы причального контейнерного крана	В. В. Каширин
Игра «Побег»	К. В. Егоров, П. М. Райков
Карточная игра «Triple Triad»	А. А. Зезюкин
Моделирование работы стиральной машины	И. И. Гниломёдов
Моделирование устройства для продажи проездных билетов	Д. И. Кулагин, А. В. Суслов
Моделирование многоэтажной автоматической стоянки на основе автоматного программирования	Ф. В. Подтелкин
Моделирование автомата класса «Однорукий бандит»	Д. Д. Захаров, С. С. Косухин
Моделирование устройства для обмена валюты	А. Ю. Законов, А. А. Клебанов

Автоматизированная система оплаты мобильного телефона	В. Р. Данилов, И. О. Варвалюк
Система сетевого файлообмена (аналог системы NetBIOS)	И. А. Вотинов, Б. З. Хасянзянов
Моделирование цифрового фотоаппарата на основе автоматного подхода к программированию	А. С. Потёмкин, М. И. Меретяков
Система управления пассажирским лифтом	Е. О. Решетников, М. В. Смачных
Система управления автомобильной сигнализацией	А. Х. Киракозов, Б. Р. Яминов
Моделирование работы «умного» светофора	Т. Г. Магомедов, А. Б. Островский
Моделирование работы автоматического путеукладчика на основе автоматного программирования	В. Г. Лашманов
Моделирование работы банкомата	В. А. Козлов, О. А. Комалёва
Реализация классической игры «Ним» на основе автоматного подхода	А. В. Яковлев, М. А. Лукин
Имитация работы автоматической коробки передач	А. В. Вокин, И. А. Пименов
Реализация алгоритма Лампорта на основе автоматного подхода	Р. В. Сатюков, И. А. Синёв
Технология моделирования одного класса мультиагентных систем на основе автоматного программирования на примере игры «Соревнование летающих тарелок»	Д. А. Паращенко, Ф. Н. Царев
Моделирование устройства для продажи газированной воды на инструментальном средстве UniMod	И. И. Колыхматов, О. О. Рыбак

«Устройство» для карточной игры Покер	С. М. Вишняков, Д. Ю. Кочелаев

5.3. Создание мобильного приложения

В качестве примера применения методики и инструментального средства в компании *eVelopers*, рассматривается разработка автоответчика для мобильного телефона *Nokia 6600*. На данном мобильном телефоне установлена операционная система *Symbian* (http://www.symbian.com/) и разработка приложений выполняется на языке *C++*.

Для языка *C++* были созданы шаблоны для генерации кода, что позволило применять пакет *UniMod* при использовании языка *C++* как целевого языка программирования. Прямая интерпретация автоматной модели не реализована при использовании языка *C++*. При этом отметим, что интерпретационный подход является более медленным и ресурсоемким по сравнению с компилятивным подходом, поэтому его использование нецелесообразно для мобильных устройств.

Для разработки *C++* приложений с использованием пакета *UniMod* для платформы *Symbian* необходимо установить следующее программное обеспечение:

- *Eclipse SDK 3.1.2* (http://www.eclipse.org)
- *Eclipse C/C++ Development Tools (CDT) 3.0.2* (http://www.eclipse.org/cdt/)
- *UniMod 1.3.1.36* (http://unimod.sourceforge.net/)
- *Symbian OS development plug-in for Eclipse* (http://www.newlc.com/Eclipse-plug-in-for-Symbian-C.html)
- *Active Perl* (http://www.activestate.com/Products/ActivePerl/)
- *S60 Platform SDK for Symbain OS, for C++* (http://www.forum.nokia.com/)
- *Microsoft .NET Software Development Kit* (http://www.microsoft.com/downloads/)

- *Microsoft Visual C++ Toolkit 2003* (http://www.microsoft.com/downloads/)
- *Microsoft Windows Platform SDK* (http://www.microsoft.com/downloads/)

Цикл разработки состоит из следующих шагов:

1. Создать новый *Java* проект.
2. Создать в этом проекте *UniMod* модель.
3. Сгенерировать классы «заглушки» для объектов управления и источников событий.
4. При этом будет сгенерирован код, реализующий автоматы из *UniMod* модели на языке *Symbian C++*.
5. Создать новый *Symbian OS* проект.
6. Перенести в этот проект код сгенерированный на шаге 4.
7. Реализовать на языке *Symbian C++* объекты управления, источники событий и вспомогательные классы для инициализации приложения, создания экземпляров объектов управления, источников событий и сгенерированных автоматных классов.

Для *UniMod* модели генерируются пара файлов: заголовочный *Symbian C++* файл и файл, содержащий реализацию методов классов, объявленных в заголовочном файле. Для каждой автоматной модели генерируется несколько классов:

- класс, реализующий «движок» автоматный модели, этот класс является интерфейсом между автоматной частью программы, сгенерированной *UniMod* и частью программы, которую необходимо запрограммировать вручную;
- вспомогательный базовый автоматный класс, содержащий общие для всех автоматов модели поля и методы;
- по классу на каждый автомат модели.

Класс, сгенерированный для одного автомата, модели содержит:

- конструктор, принимающий на вход экземпляр класса «движка» модели;
- метод *HandleL*, который реализует логику обработки событий;
- методы для установки ссылок на экземпляры объектов управления и вызываемых автоматов;
- перечисление (*enum*) всех состояний, объявленных в автомате;
- поле, соответствующее переменной состояния;
- ссылки на все связанные с автоматом объекты управления и вызываемые автоматы;
- метод *IsStable*, вычисляющий является ли состояние, переданное ему в качестве формального параметра устойчивым;
- метод *GetSuperstate*, вычисляющий состояние, в которое вложено состояние, переданное ему в качестве формального параметра;
- метод *TransiteOnEventL*, реализующий переход из текущего активного состояния, в зависимости от события и значений входных переменных;
- метод *TransiteToStable*, реализующий переход из состояния заданного входным параметром в соответствующее ему устойчивое состояние;
- метод *EnterStateL*, реализующий вход автомата в состояние;
- метод *InvokeSubmachineL*, реализующий вызов автоматов, вложенных в заданное состояние.

Объекты управления и источников событий должны быть запрограммированы вручную. Класс, реализующий «движок» автоматной модели, содержит методы двухфазной инициализации [109], которые принимают на вход ссылки на экземпляры объектов управления, и инициализируют автоматы модели.

Для посылки события автоматам модели, источников событий должны вызывать метод «движка» автоматной модели *HandleL*, который, в свою очередь, делегирует вызов головному автомату модели. Источники событий могут передать параметры события, используя контекст, ссылку на который они могут получить у «движка» автоматной модели, используя метод *Context*.

Для того чтобы запустить автоматную модель требуется вручную запрограммировать создание объектов управления, источники событий и «движка» автоматной модели. При создании «движка» автоматной модели ему передаются ссылки на объекты управления.

Предполагается, что источники событий запрограммированы таким образом, чтобы подписываться и обрабатывать системные события, и транслировать их затем в вызовы метода *HanldeL* «движка» автоматной модели. Так как в приложениях на *Symbian C++* используется невытесняющая многозадачность, проблемы с одновременной обработкой автоматом нескольких событий не возникает – новое событие начинает обрабатываться только по завершении обработки текущего.

5.3.1. Постановка задачи

Опишем требования:

1. Разработать программу-автоответчик для мобильного телефона *Nokia 6600*.
2. Программа должны быть написана на языке *C++* для платформы *Symbian.*
3. Программа должна поддерживать два режима работы: режим настройки голосовых сообщений и режим ответа на звонки.
4. В режиме настройки голосовых сообщений программа должна позволять пользователю записать произвольное количество голосовых сообщений для определенного абонента или группы абонентов из встроенной в телефон телефонной книги.

5. В режиме ответа на звонки программа должна автоматически снимать трубку, определять номер абонента и проигрывать предназначенное для этого абонента сообщение.

На рис. 47–51 показан прототип приложения в виде экранных форм. Каждый рисунок показывает состоит из двух частей: слева показана экранная форма, а справа – контекстное меню, определяющее набор возможных действий.

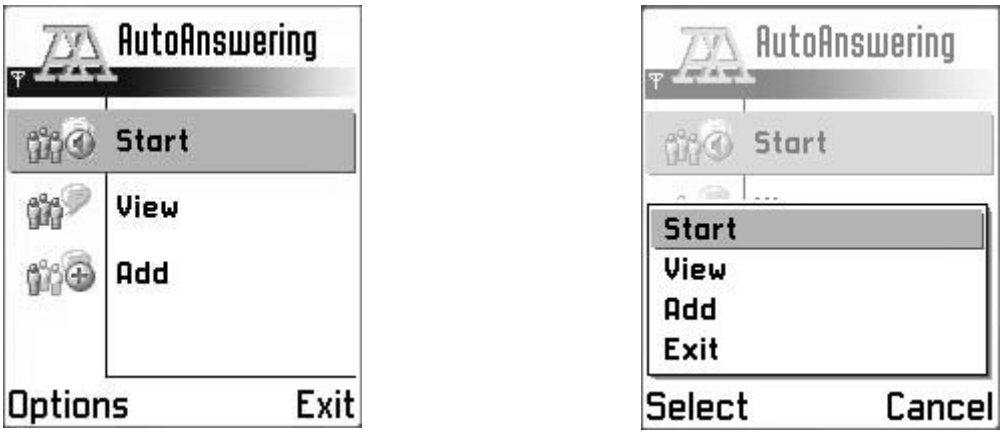

Рис. 47. Главная экранная форма (*Main*)

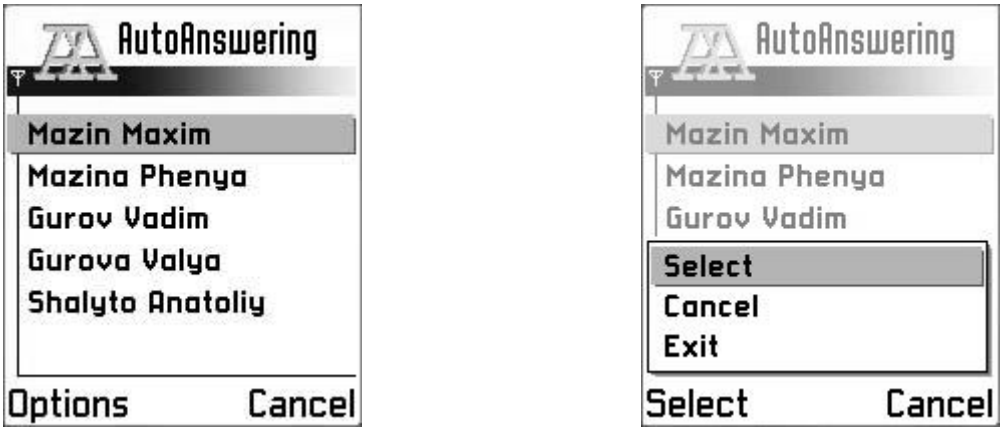

Рис. 48. Форма выбора абонента или группы абонентов (*Select user or group*)

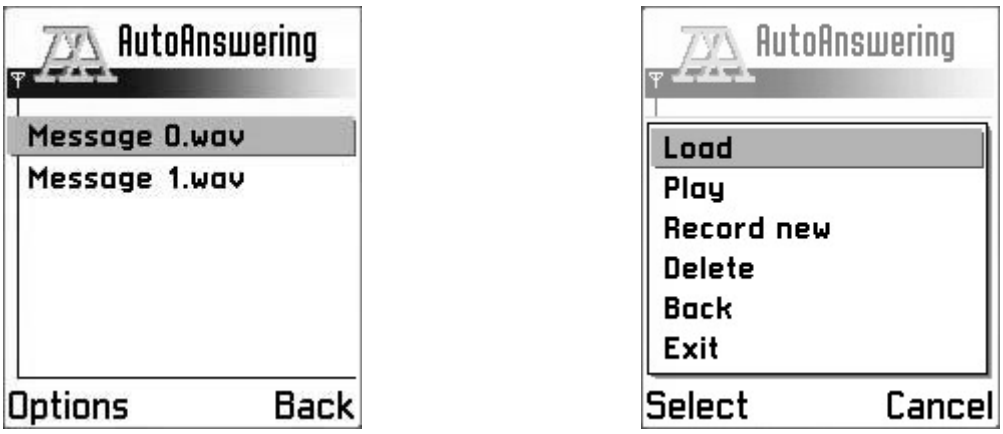

Рис. 49. Форма выбора записанного сообщения (*Select greeting message*)

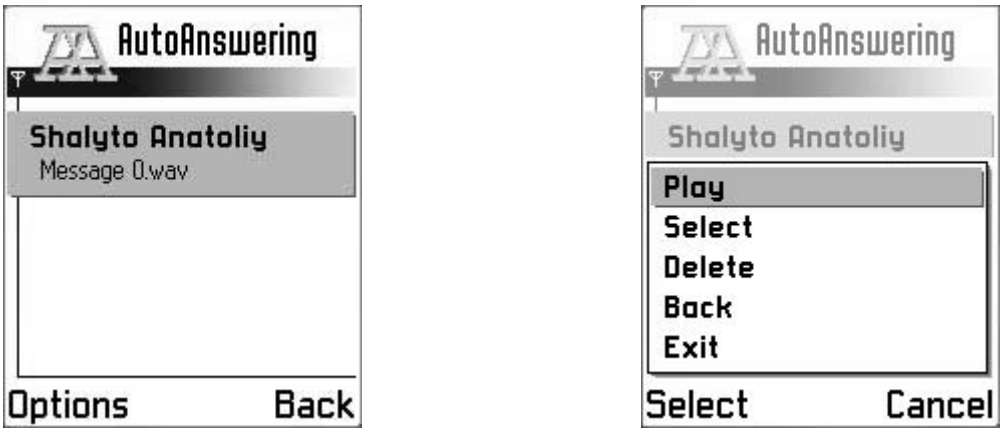

Рис. 50. Форма просмотра абонентов и предназначенных для них сообщений (*View*)

Рис. 51. Форма записи нового сообщения (*Record*)

На рис. 52 представлена диаграмма состояний, описывающая переходы между экранными формами. Названия состояний соответствуют названиям экранных форм в подписях под рис. 47–51. Для состояния *Active*

экранная форма отсутствует, так как в нем приложение переходит в фоновый режим ответа на звонки.

События на переходах на рис. 52 соответствуют пунктам контекстного меню на рис. 47–51.

Составное состояние *Application Running* введено того, чтобы сделать диаграмму более компактной, так как оно позволяет только один раз определить переход в финальное состояние, который возможен из любого вложенного состояния.

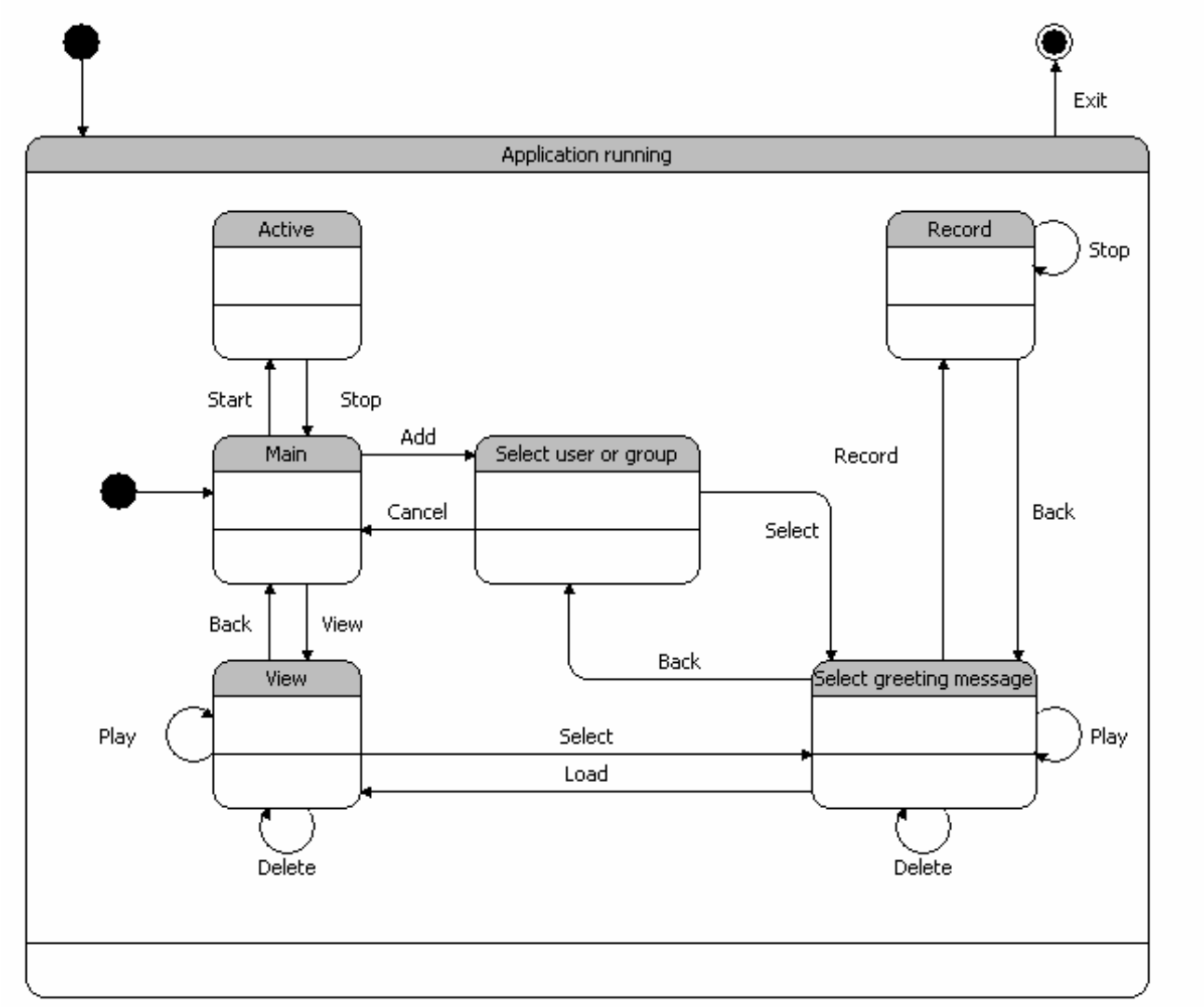

Рис. 52. Диаграмма состояний, описывающая переходы между экранными формами

5.3.2. Статическая модель системы

На рис. 53 представлена статическая модель системы. На ней с помощью стереотипов *<<eventprovider>>*, *<<controlledobject>>* и *<<statemachine>>* выделены источники событий, объекты управления и автоматы соответственно.

Стереотипом *<<datatype>>* помечены классы, которые являются контейнером для данных.

Стереотипом *<<service>>* помечены классы, являющиеся составной частью операционной системы *Symbian* и предоставляющие приложению различные сервисные функции для взаимодействия с окружающей средой.

Классы, не помеченные стереотипами, показаны для удобства представления предметной области и не имеют соответствующей им реализации. Так класс `Phone` логически агрегирует в себе все сервисы, предоставляемые запущенным на нем приложениям, а класс `User` соответствует пользователю, работающему с телефоном.

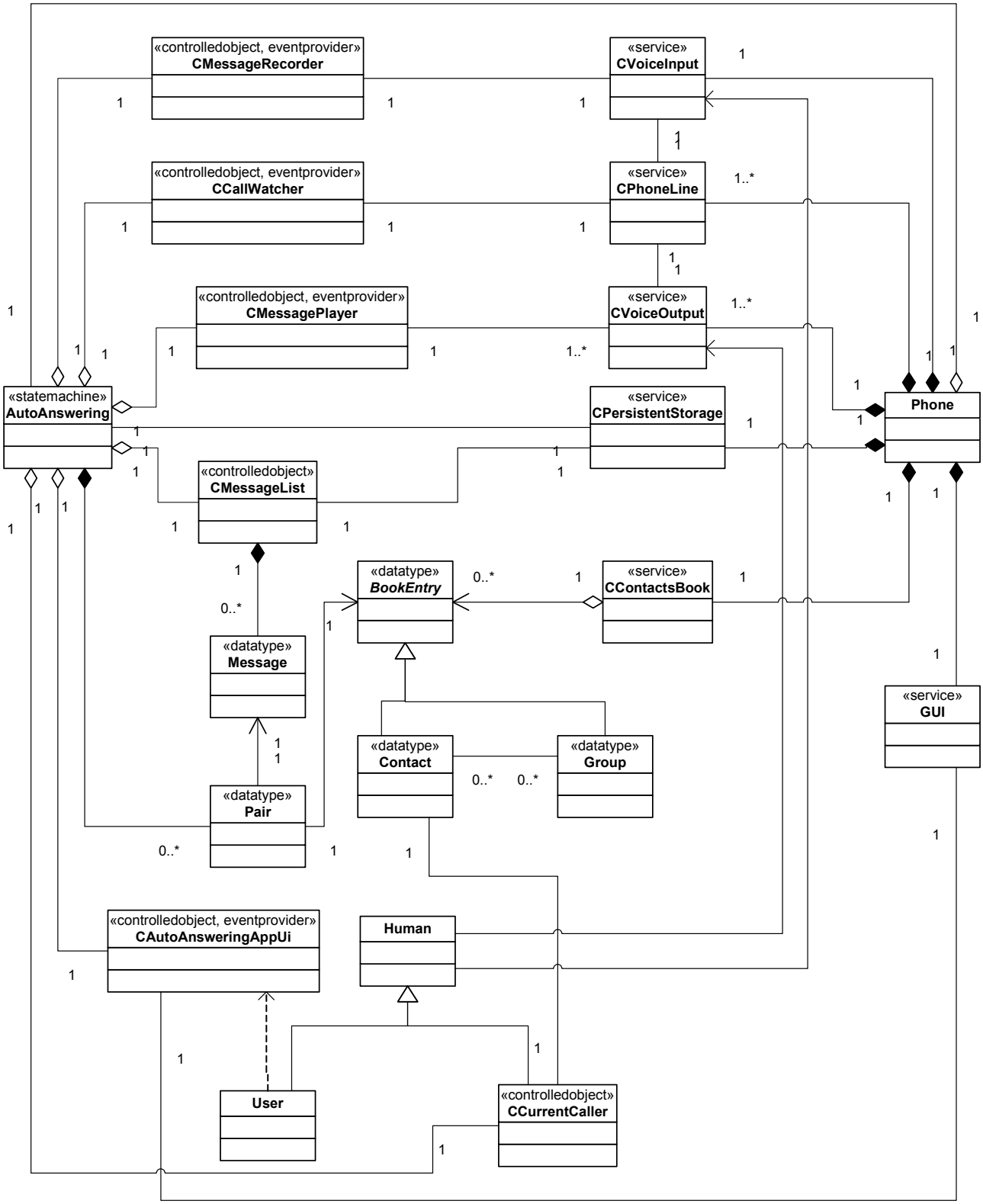

Рис. 53. Статическая модель системы

5.3.3. Динамическая модель системы

На рис. 54 представлена схема связей автоматов. Слева на ней расположены источники событий, а справа – автоматы и объекты

управления. Отметим, что такие классы статической модели системы, как `CAutoAnsweringApi`, `CMessagePlayer`, `CMessageRecorder`, `CCallWatcher`,`CMessageList`, `CCurrentWatcher` в динамической модели играют роль и источников событий и объектов управления, поэтому на схеме связей они показаны дважды.

Классы со стереотипами <<*datatype*>> и <<*service*>> не показаны на схеме связей, так как они не являются частью динамической модели системы.

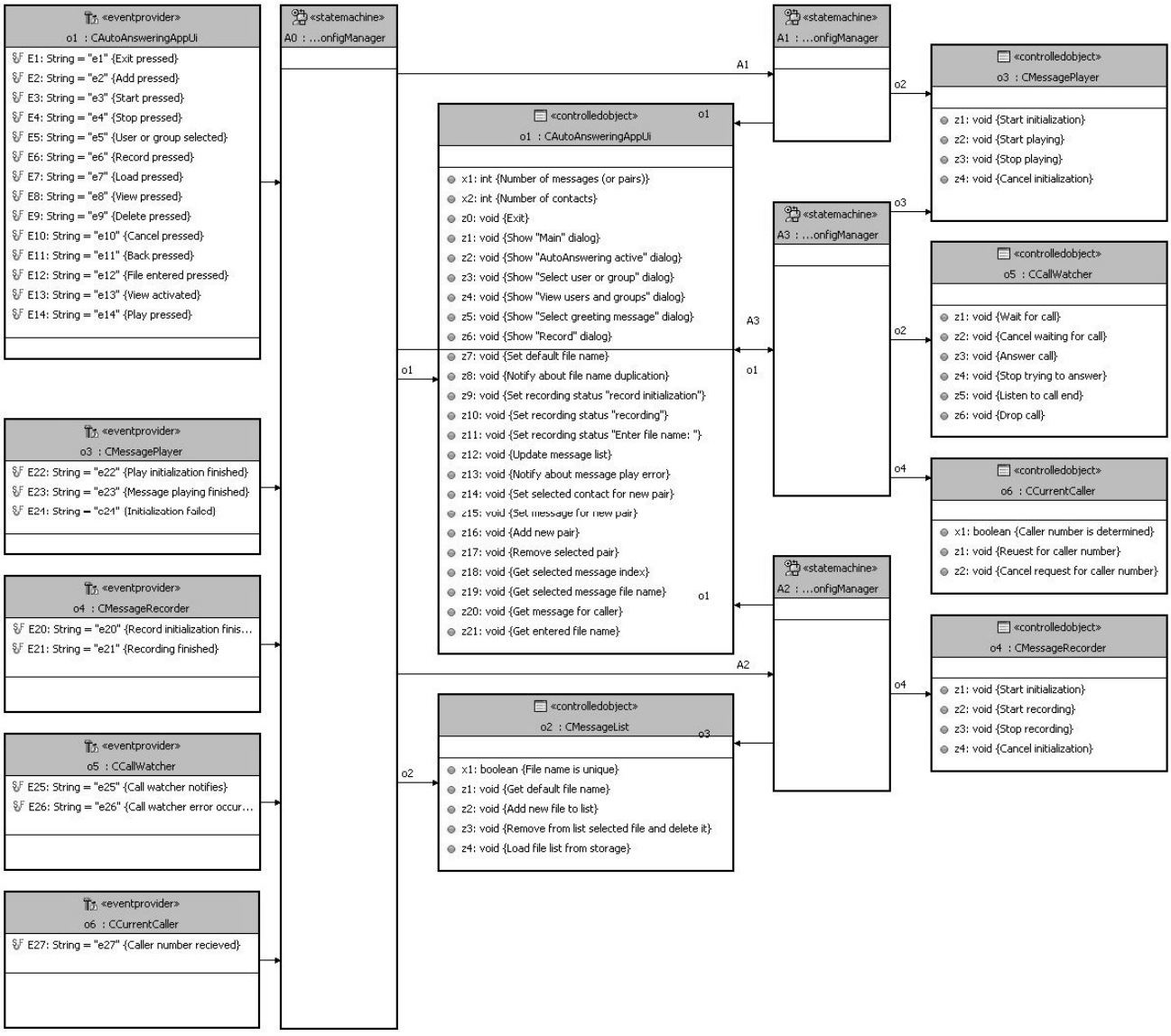

Рис. 54. Схема связей автоматов

Класс статической модели `AutoAnswering` в динамической модели декомпозирован на четыре автомата:

- *A0*. Построен на основе диаграммы переходов между экранными формами (рис. 55). Управляет режимом настройки голосовых сообщений.
- *A1*. Управляет проигрыванием сообщений.
- *A2*. Контролирует запись новых сообщений.
- *A3*. Управляет режимом ответов на звонки.

На рис. 55–58 показаны диаграммы переходов автоматов *A0–A3*.

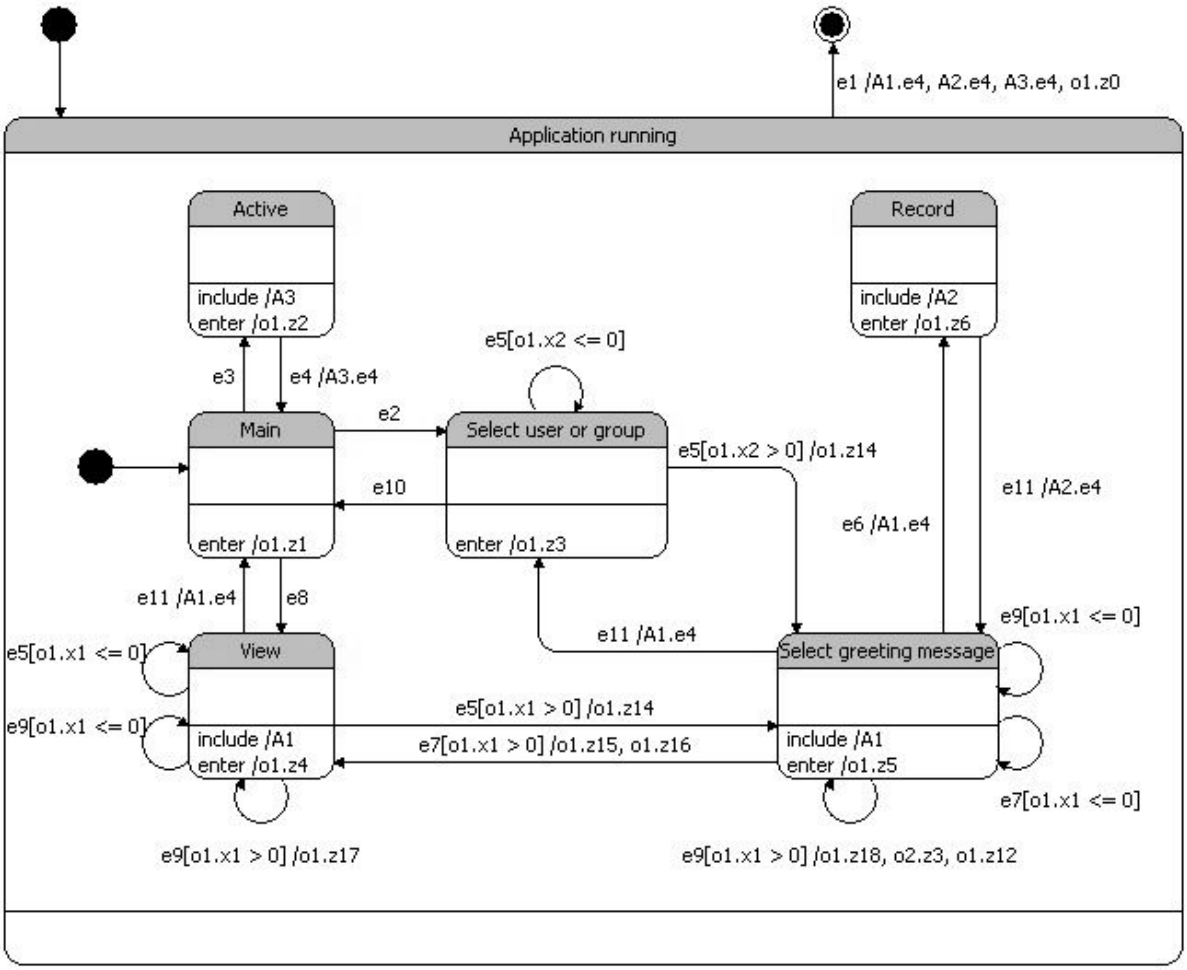

Рис. 55. Диаграмма переходов автомата *A0*

Автомат *A0* взаимодействует с автоматами *A1–A3* с помощью вложенности (например, автомат *A3* вложен в состояние автомата *A0 Active*) и вызываемости (например, в автомате *A0* присутствует переход *e11/A2.e4*). Вложенность означает, что, например, все события, полученные автоматом

A0 в состоянии *Active* будут также делегированы автомату *A3*. Вызываемость означает явную посылку события другому автомату.

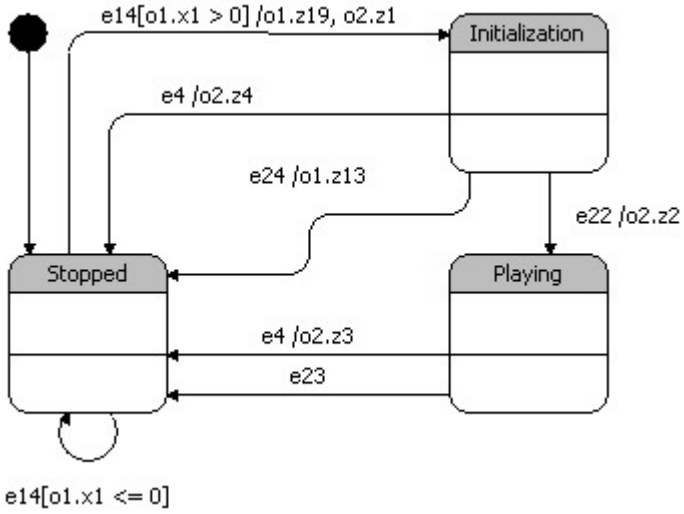

Рис. 56. Диаграмма переходов автомата *A1*

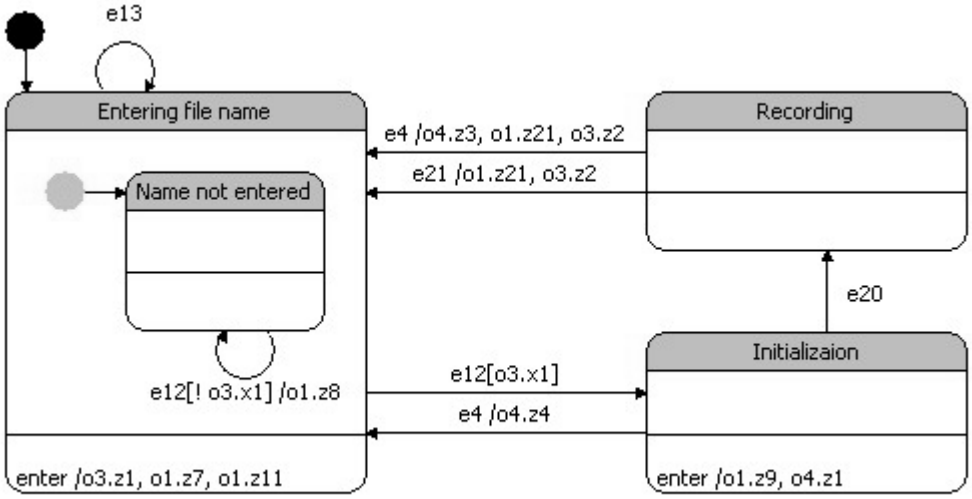

Рис. 57. Диаграмма переходов автомата *A2*

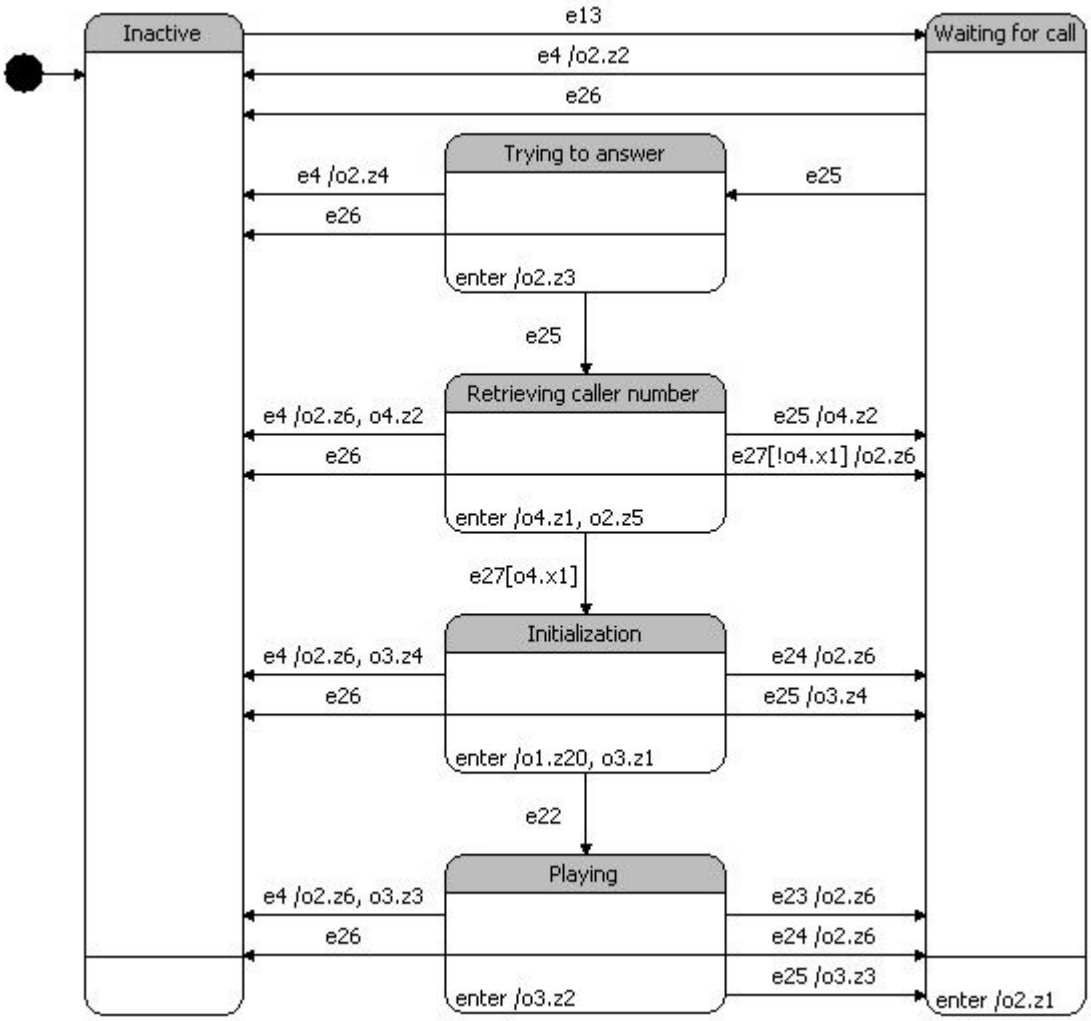

Рис. 58. Диаграмма переходов автомата *A3*

5.3.4. Создание кода

Для того чтобы источник событий имел возможность посылать события автоматной модели, у него должна быть ссылка на экземпляр «движка» автоматной модели. Для класса `CMessagePlayer` метод, реализующий установку ссылки на «движок» автоматной модели, имеет следующий вид:

```
void CMessagePlayer::SetModelEngine(CAutoAnswererModelEngine* aModelEngine)
    {
    iModelEngine = aModelEngine;
    }
```

Для посылки события автомату, источник событий должен вызвать метод `HandleL` у объекта «движка» автоматной модели. Например,

источник событий CMessagePlayer посылает автомату событие о завершении инициализации следующим образом:

```
void CMessagePlayer::MapcInitComplete(TInt aError, const TTimeIntervalMicroSeconds&
/*aDuration*/)
    {
    iLogger.LogVariableL(«MessagePlayer.Error», aError);
    if (iModelEngine)
        {
        if (aError == KErrNone)
            {
            TRAPD (err, iModelEngine->HandleL(CAutoAnswererModelEngine::E23));
            if (err)
                {
                Stop();
                }
            }
        else
            {
            iStatus = aError;
            iModelEngine->HandleL(CAutoAnswererModelEngine::E24);
            }
        }
    }
```

Для объектов управления методы, реализующие входные и выходные переменные, программируются вручную. Так для объекта управления CMessageList выходная переменная *z3* имеет вид:

```
void CMessageList::Z3L(TStateMachineContext& aContext)
    {
    if (aContext.iMessageFileIndex >= 0 && aContext.iMessageFileIndex < iList->Count())
        {
        TBuf<256> fileName;
        StringLoader::Load(fileName, R_AUTOANSWERING_MSG_DIR_PATH);
        fileName.Append(iList->At(aContext.iMessageFileIndex).iName);
        BaflUtils::DeleteFile(iFs, fileName);
        iList->Delete(aContext.iMessageFileIndex);
        CreateDefaultNameL();
        }
    }
```

Входная переменная *x1* для того же объекта управления реализована следующим образом:

```
TBool CMessageList::X1L(TStateMachineContext& aContext)
    {
    TInt numberMessages(iList->Count());

    for (TInt i = 0; i < numberMessages; i++)
        {
        const TMessage& message = iList->At(i);

        if(message.iName.Compare(*aContext.iFileName) == 0)
            {
            return EFalse;
            }
        }
    return ETrue;
    }
```

Код, сгенерированный для метода TransiteOnEventL автомата *A2*, реализующий переход из текущего активного состояния, в зависимости от события и значений входных переменных:

```
TA2::EState TA2::TransiteOnEventL(CAutoAnswererModelEngine::EEvent aEvent, TStateMachineContext&
aContext)
    {
    TA2::EState targetState = UNKNOWN_STATE;
    for (TA2::EState sourceState = iState; targetState == UNKNOWN_STATE && sourceState != TOP;
sourceState = GetSuperstate(sourceState))
        {
        switch (sourceState)
            {
            /* State: Top */
            case TOP:
                switch (aEvent)
                    {
                    default: ;// Do nothing if no event was triggered
                    }
                break;
            /* State: Recording */
            case RECORDING:
                switch (aEvent)
                    {
                    case CAutoAnswererModelEngine::EEvent::E21:
                        iO1->Z21L(aContext);
                        iO3->Z2L(aContext);
                        targetState = ENTERING_FILE_NAME;
                        EnterStateL(targetState, aContext);
                        break;
                    case CAutoAnswererModelEngine::EEvent::E4:
                        iO4->Z3L(aContext);
                        iO1->Z21L(aContext);
                        iO3->Z2L(aContext);
                        targetState = ENTERING_FILE_NAME;
                        EnterStateL(targetState, aContext);
                        break;
                    default: ;// Do nothing if no event was triggered
                    }
                break;
            /* State: Initializaion */
            case INITIALIZAION:
                switch (aEvent)
                    {
                    case CAutoAnswererModelEngine::EEvent::E20:
                        targetState = RECORDING;
                        EnterStateL(targetState, aContext);
                        break;
                    case CAutoAnswererModelEngine::EEvent::E4:
                        iO4->Z4L(aContext);
                        targetState = ENTERING_FILE_NAME;
                        EnterStateL(targetState, aContext);
                        break;
                    default: ;// Do nothing if no event was triggered
                    }
                break;
            /* State: Entering file name */
            case ENTERING_FILE_NAME:
                switch (aEvent)
                    {
                    case CAutoAnswererModelEngine::EEvent::E12:
                        if (iO3->X1L(aContext))
                            {
                            targetState = INITIALIZAION;
                            EnterStateL(targetState, aContext);
                            }
                        break;
                    case CAutoAnswererModelEngine::EEvent::E13:
                        targetState = ENTERING_FILE_NAME;
                        EnterStateL(targetState, aContext);
                        break;
                    default: ;// Do nothing if no event was triggered
                    }
                break;
            /* State: Name not entered */
            case NAME_NOT_ENTERED:
                switch (aEvent)
                    {
                    case CAutoAnswererModelEngine::EEvent::E12:
                        if (!iO3->X1L(aContext))
                            {
                            iO1->Z8L(aContext);
                            targetState = NAME_NOT_ENTERED;
```

```
                            EnterStateL(targetState, aContext);
                            }
                        break;
                    default: ;// Do nothing if no event was triggered
                    }
                break;
            }
        }

    // Look for default transition
    for (TA2::EState sourceState = iState; targetState == UNKNOWN_STATE && sourceState != TOP;
sourceState = GetSuperstate(sourceState))
        {
        switch (sourceState)
            {
            /* State: Top */
            case TOP:
                break;
            /* State: Recording */
            case RECORDING:
                break;
            /* State: Initializaion */
            case INITIALIZAION:
                break;
            /* State: Entering file name */
            case ENTERING_FILE_NAME:
                break;
            /* State: Name not entered */
            case NAME_NOT_ENTERED:
                break;
            }
        }

    // If no transition was found stay in current state
    return targetState != UNKNOWN_STATE ? targetState : iState;
    }
```

Для инициализации и запуска автоматной модели необходимо создать объекты управления, источники событий и автоматы:

```
    iMessageList = CMessageList::NewL(Document().Fs());

    iContactEngine = CPbkContactEngine::NewL();

    iActiveView = CAutoAnsweringViewActive::NewL();
    AddViewL(iActiveView); // Transfer ownership to base class

    iMainView = CAutoAnsweringViewMain::NewL();
    AddViewL(iMainView); // Transfer ownership to base class

    iRecordView = CAutoAnsweringViewRecord::NewL();
    AddViewL(iRecordView); // Transfer ownership to base class

    iSelectMsgView = CAutoAnsweringViewSelectMsg::NewL(*iMessageList);
    AddViewL(iSelectMsgView); // Transfer ownership to base class

    iSelectUserView = CAutoAnsweringViewSelectUser::NewL(*iContactEngine);
    AddViewL(iSelectUserView); // Transfer ownership to base class

    iViewView = CAutoAnsweringViewView::NewL(Document().ContactMessageMap(), *iContactEngine);
    AddViewL(iViewView); // Transfer ownership to base class

    iCallWatcher = CCallWatcher::NewL(*iLogger);
    iMessageRecorder = CMessageRecorder::NewL(Document().Fs());
    iMessagePlayer = CMessagePlayer::NewL(Document().Fs(), iCallWatcher->Phone(), *iLogger);
    iMessagePlayer->SetUseLoudSpeaker(ETrue);
    iLineMessagePlayer = CMessagePlayer::NewL(Document().Fs(), iCallWatcher->Phone(), *iLogger);
    iLineMessagePlayer->SetUseLoudSpeaker(EFalse);
    iCurrentCaller = CCurrentCaller::NewL(*iContactEngine);

    iModelEngine = CAutoAnswererModelEngine::NewL(
        this, iMessageList, iMessagePlayer, iMessageRecorder,
        iCallWatcher, iCurrentCaller);
    iMessagePlayer->SetModelEngine(iModelEngine);
    iMessageRecorder->SetModelEngine(iModelEngine);
    iCallWatcher->SetModelEngine(iModelEngine);
```

```
// ToDo find out what should do LineMessagePlayer
iLineMessagePlayer->SetModelEngine(iModelEngine);
iCurrentCaller->SetModelEngine(iModelEngine);
```

Программа, построенная на основе изложенного подхода, может быть загружена с сайта http://www.evelopers.com.

5.4. Текстовый язык для автоматного программирования

С использованием предлагаемого метода и инструментального средства *UniMod* был выполнен ряд проектов, которые показали эффективность применения автоматного программирования и средства *UniMod* при реализации систем со сложным поведением. Тем не менее многие программисты предпочитают работать с текстовым представлением программы, несмотря на то, что диаграммы позволяют представлять информацию более компактно и обозримо.

Для расширения функциональных возможностей инструментального средства *UniMod* был разработан прототип текстового языка автоматного программирования с помощью системы метапрограммирования *JetBrains Meta Programming System* (*MPS*) [110, 111], которая позволяет быстро создавать проблемно-ориентированные языки (*Domain Specific Language — DSL*) [111, 112]. Для задания языка в системе *MPS* требуется разработать:

- структуру абстрактного синтаксического дерева (*АСД*) [48] для разрабатываемого языка. Узлам *АСД* могут соответствовать такие понятия как «объявление класса», «вызов метода», «операция сложения» и т.п.;
- модель текстового редактора для каждого типа узла *АСД*. Задание редактора для узла *АСД* равноценно заданию конкретного синтаксиса для этого узла. При этом, если для традиционных текстовых языков программирования создание удобного редактора – отдельная сложная задача, то для языков, созданных с помощью средства *MPS*, редакторы являются частью языка. Эти редакторы

поддерживают автоматическое завершение ввода текста и проверку корректности программы;

- модель ограничений на экземпляры *АСД*;
- модель системы типов [113] для языка;
- модель трансформации программы на задаваемом языке в исполняемый код.

Система *MPS* позволяет как создавать новые языки, так и расширять языки, уже созданные с помощью этой системы.

В отличие от традиционных, языки, созданные с помощью системы *MPS*, не являются текстовыми в традиционном понимании, так как при программировании на них пользователь пишет не текст программы, а вводит ее в виде *АСД* с помощью специальных редакторов. Структура и внешний вид этих редакторов таковы, что работа с моделью программы для пользователя выглядит, как традиционная работа с текстом программы. Отказ от традиционного текстового ввода программ значительно упрощает создание новых языков [114] – исчезает необходимость в разработке лексических и синтаксических анализаторов, и, как следствие, перестают действовать ограничения на класс грамматик языков. Недостатком такого подхода является зависимость языков от системы *MPS* – невозможно разрабатывать программы без этой системы. Однако подобное ограничение присуще и традиционным, чисто текстовым языкам, которые зависят от компиляторов. Впрочем, после трансляции программы, написанной на языке, созданном в системе *MPS*, исполняемый код перестает зависеть от этой системы.

С помощью *MPS* созданы два варианта текстового языка для автоматного программирования. Первый язык выполнен в виде расширения языка *Java*, второй является самостоятельным языком. Эти языки позволяют описывать состояния и логику переходов автоматов, а также события, обрабатываемые автоматами. При этом, также как и в инструментальном

средстве *UniMod*, функции входных и выходных переменных реализуются на другом языке программирования.

Язык, который расширяет язык *Java*, позволяет в одном *Java*-классе совмещать автоматные и не автоматные аспекты. При этом диаграммы состояний строятся автоматически по мере набора текста и доступны только в режиме просмотра, но не редактирования. После написания программный код сначала транслируется в *Java*-код, уже без автоматного расширения, а потом компилируется стандартным *Java*-компилятором. Преимуществом этого языка является простота его использования в объектно-ориентированных приложениях, написанных на языке *Java*. При этом поведенческие аспекты программы описываются в *Java*-коде явно, с помощью специального удобного для редактирования автоматного языка, осуществляющего проверку корректности программы на стадии ее написания, а не в процессе ее компиляции.

В то же время для приложений, написанных на других языках, предложенный язык не подходит. Для таких приложений предпочтительно использовать независимую от языка *Java* версию автоматного языка. Эта версия менее зависима от платформы и полностью соответствует понятию запускаемых спецификаций.

Использование специального языка для автоматного программирования упрощает разработку автоматов, избавляя программиста от необходимости описывать их средствами языков общего назначения.

При этом диаграммы состояний, являющиеся более наглядными, по сравнению с текстовым представлением автоматов, строятся по текстовому описанию автоматически и на лету.

На рис. 59 слева показан пример текстовой автоматной программы на разработанном языке программирования первого типа, распознающей цепочки символов вида *a*b*c**. Автоматически построенная диаграмма состояний изображена на рис.59 справа.

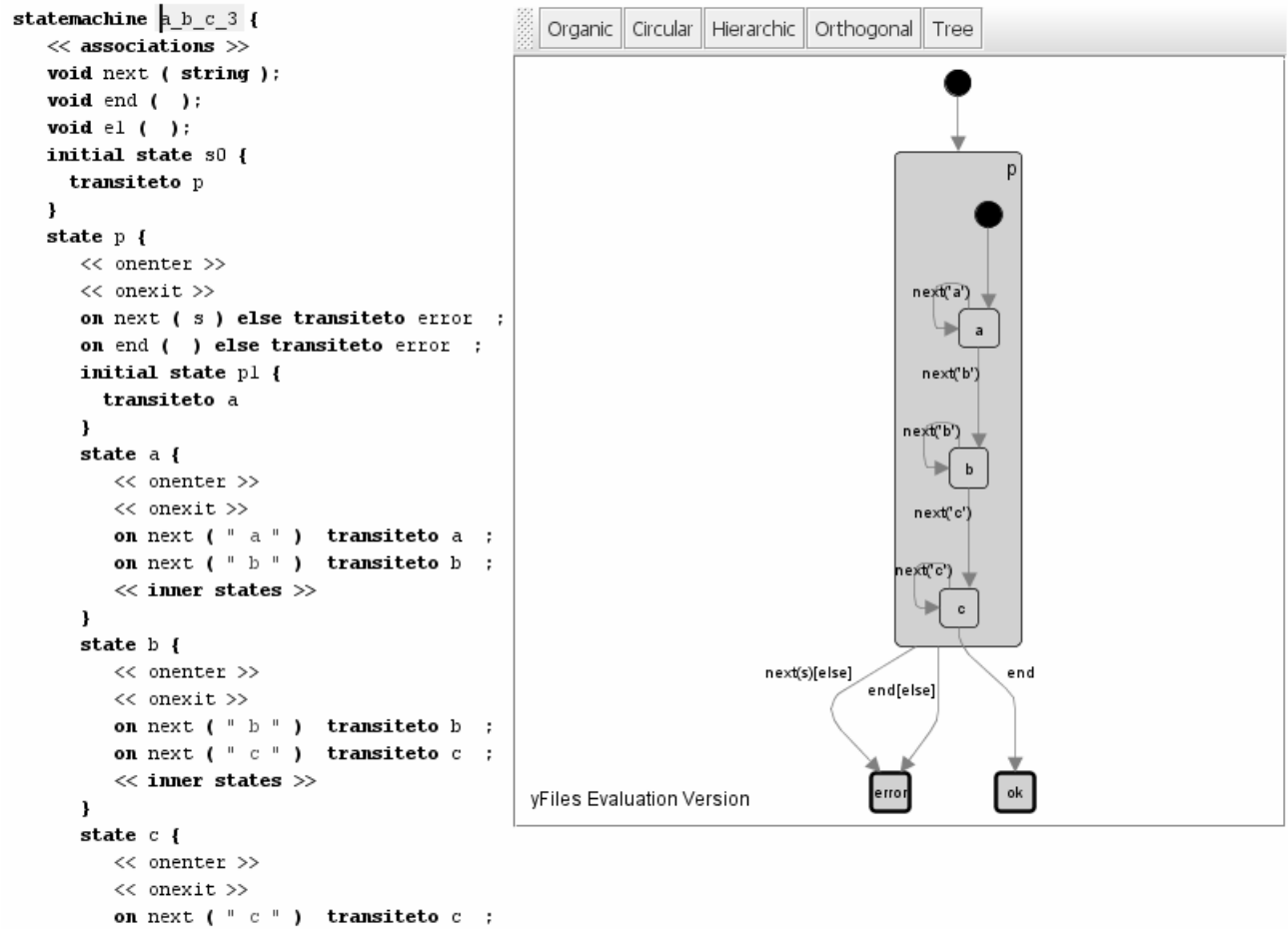

Рис. 59. Пример программы на текстовом языке

Примеры, выполненные с помощью прототипа предложенного текстового языка автоматного программирования, продемонстрировали его эффективность. Поэтому в рамках работ по созданию следующей версии инструментального средства *UniMod2* [115] в него будет включена реализация этого языка.

Выводы по главе 5

1. При разработки системы автоматического завершения ввода в инструментального для инструментального средства *UniMod* использовано само это средство – применен так называемый «метод раскрутки».
2. Разработанные методы и инструментальное средство успешно применены для выполнения студенческих проектов по курсу

«Автоматное программирование» на кафедре «Компьютерных технологий» СПбГУ ИТМО.

3. Разработанные методы и инструментальное средство внедрено в компании *eVelopers* для промышленной разработки программного обеспечения. Приведен пример разработки автоответчика для мобильного телефона.
4. Разработан прототип текстового языка автоматного программирования в среде быстрой разработки языков *MPS*. В рамках работ по созданию следующей версии инструментального средства *UniMod2* в него будет включена реализация этого языка.

ЗАКЛЮЧЕНИЕ

В настоящей работе разработан метод проектирования объектно-ориентированных программ с явным выделением состояний. В рамках этого метода модель программы предлагается строить с помощью двух *UML*-диаграмм – диаграмм классов и состояний. Разработанный метод апробирован при создании ряда приложений и показал свою эффективность. Он позволяет:

- сократить объем ручного программирования;
- при наличии библиотеки источников событий и объектов управления для определенной предметной области, полностью отказаться от ручного программирования;
- использовать диаграммы классов и графы переходов в составе проектной документации;
- формально и наглядно описывать логику поведения программы и модифицировать ее, изменяя только графы переходов;
- упростить сопровождение проекта вследствие повышения централизации логики программы.

Для поддержки разработанного метода было создано инструментальное средство *UniMod.* Оно позволяет визуально создавать модели программ, запускать их в режиме интерпретации или компиляции, визуально отлаживать их. Инструментальное средство также позволяет верифицировать создаваемые модели, для некоторых типов найденных ошибок автоматически предлагаются варианты их исправлений.

Исходные тексты, документация и примеры использования программного пакета *UniMod* представлены на сайте http://unimod.sourceforge.net. За все время существования проекта было произведено более 45000 скачиваний.

Предложенный метод и инструментальное средство были успешно внедрены в учебном процессе для создания курсовых проектов, а также в

коммерческих организациях для создания промышленных приложений для мобильных устройств и интернет.

Также в ходе внедрения для нужд самого инструментального средства был разработан новый подход к созданию систем автоматического завершения ввода на основе теории построения трансляторов. Особенностью подхода является то, что транслятор целиком строится в виде диаграмм классов и состояний. Это позволило также создать эффективный метод для восстановления транслятора после ошибок, который заключается в том, что каждое состояние на диаграмме состояний транслятора автоматически дополняется переходами для пропуска лишних и вставки недостающих символов во входную последовательность. Выбор одного из этих переходов осуществляется динамически во время обработки входной последовательности.

Для расширения функциональных возможностей инструментального средства *UniMod* был разработан прототип текстового языка автоматного программирования. Разработка осуществлялась с помощью системы метапрограммирования *JetBrains MPS*. В рамках работ по созданию следующей версии инструментального средства *UniMod2* в него будет включена реализация этого языка.

Полученные результаты позволяют ускорить процесс разработки программного обеспечения, а также повысить качество создаваемых программ.

ИСТОЧНИКИ

1. *Соммервилл И.* Инженерия программного обеспечения. М.: Вильямс, 2002.
2. *Грехем И.* Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004.
3. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. СПб.: Невский диалект, 2001.
4. *Ларман К.* Применение UML и шаблонов проектирования. М.: Вильямс, 2001.
5. *Коуд П., Норт Д., Мейфилд М.* Объектные модели. Стратегии, шаблоны и приложения. М.: Лори, 1999.
6. *Harel D., Politi M.* Modelling Reactive Systems with Statecharts. NY: McGraw-Hill, 1998.
7. *Harel D.* Statecharts: A Visual Formalizm for Complex Systems //Science of Computer Programming. 1987. № 8, pp. 231–274.
8. *Harel D. et al.* STATEMATE: A Working Environment for the Development of Complex Reactive Systems //IEEE Transactions on Software Engineering. 1990. № 4, pp. 234 – 252. http://csdl.computer.org/comp/trans/ts/1990/04/e0403abs.htm
9. *Кузнецов С.* Обещания и просчеты UML 2.0 //Открытые системы. 2006. № 2, с. 75–79.
10. *1st European Conference on Model-Driven Software Engineering*. Germany. 2003. http://www.agedis.de/conference/
11. *International Workshop* «e-Business and Model Based in System Design». IBM EE/A. SPb ETU, 2004.
12. *OMG Model Driven Architecture*. http://www.omg.org/mda/
13. *Frankel D*. Model Driven Architecture: Applying MDA to Enterprise Computing. NJ: Wesley, 2003.

14. *Буч Г., Рамбо Г., Якобсон И.* UML. Руководство пользователя. М.: ДМК, 2000.

15. *Mellor S., Balcer M.* Executable UML: A Foundation for Model Driven Architecture. MA: Addison-Wesley, 2002.

16. *Raistrick C., Francis P., Wright J.* Model Driven Architecture with Executable UML. Cambridge University Press, 2004.

17. *Wikipedia*. Finite state machine. http://en.wikipedia.org/wiki/Finite_state_machine#External_links

18. *Sun Studio Enpterprise*. http://developers.sun.com/prodtech/javatools/jsenterprise/reference/techart/whatis.html

19. *Jacobson I.* Four Macro Trends in Software Development Y2004. http://www.ivarjacobson.com/postnuke/html/modules.php?op=modload&name=UpDownload&file=index&req=getit&lid=9

20. *I-Logix Statemate.* http://ilogix.com/sublevel.aspx?id=74

21. *XJTek AnyState*. http://www.xjtek.com/anystates/

22. *StateSoft ViewControl.* http://www.statesoft.ie/products.html

23. *SCOPE*. http://www.itu.dk/~wasowski/projects/scope/

24. *IAR Systems visualSTATE.* http://www.iar.com/p1014/p1014_eng.php

25. *The State Machine Compiler*. http://smc.sourceforge.net/

26. *Jia X. et al.* Using ZOOM Approach to Support MDD. http://se.cs.depaul.edu/ise/zoom/papers/zoom/SERP_ZOOM.pdf

27. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.

28. *Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А.* State Machine – новый паттерн объетно-ориентированного проектирования //Информационно-управляющие системы. 2004. № 5, с. 32–36. http://is.ifmo.ru/works/pattern/
29. *The Hybrid Systems Project*. http://control.ee.ethz.ch/~hybrid/
30. *Ferrari-Trecate G. at el*. Mixed Logic Dynamical Model of a Hydroelectric Power Plant. http://control.ee.ethz.ch/research/publications/publications.msql?banner=hybrid&action=Showdetails&id=859
31. *Бенькович Е., Колесов Ю., Сениченков Ю*. Практическое моделирование динамических систем. СПб.: БХВ, 2002.
32. *Model Vision Studium*. http://www.exponenta.ru/soft/others/mvs/mvs.asp
33. *XJTek AnyLogic*. http://www.xjtek.com/anylogic/
34. *Беляев А. В., Суясов Д. И., Шалыто А. А*. Компьютерная игра космонавт. Проектирование и реализация //Компьютерные инструменты в образовании. 2004. № 4, с. 75–84. http://is.ifmo.ru/works/_cosmo_article.pdf
35. *Шалыто А., Туккель Н., Шамгунов Н.* Ханойские башни и автоматы //Программист. 2002. № 8, с. 82–90. http://is.ifmo.ru/works/hanoy/
36. *Шалыто А., Туккель Н.* От тьюрингова программирования к автоматному //Мир ПК. 2002. № 2, с. 144–149. http://is.ifmo.ru/works/turing/
37. *Шалыто А. А.* Логическое управление. Методы аппаратной и программной реализации алгоритмов. СПб.: Наука, 2000.
38. *Xilinx StateCAD*. http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?sGlobalNavPick=PRODUCTS&sSecondaryNavPick=Design+Tools&key=dr_dt_statemachine
39. *Altera Quartus II*.http://www.altera.com/products/software/sfw-index.jsp
40. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

41. *State Logic*. http://www.geindustrial.com/cwc/products?pnlid=2&id=sl
42. *I-Logix Rhapsody*. http://www.ilogix.com/rhapsody/rhapsody.cfm
43. *Mathworks Stateflow*. http://www.mathworks.com/products/stateflow/
44. *Шалыто А. А., Туккель Н. И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем. //Программирование. 2001. № 5, с. 42–54. http://is.ifmo.ru/works/_avtomatnij_podhod_k_sozdaniju_programmnogo_obespechenija.djvu
45. *Gibson D.* Finite State Machines. Making simple work of complex functions. http://www.microconsultants.com/tips/fsm/fsmarticl.pdf
46. *A Framework for Hardware-Software Co-Design of Embedded Systems*. http://www-cad.eecs.berkeley.edu/~polis/
47. *VIS (Verification Interacting with Synthesis)*. http://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html
48. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001.
49. *Хантер Р.* Основные концепции компиляторов. М.: Вильямс. 2002.
50. *Кормен Т., Лайзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦМНО, 2000.
51. *Корнеев Г. А., Шамгунов Н. Н., Шалыто А. А.* Обход деревьев на основе автоматного подхода //Компьютерные инструменты в образовании. 2004. № 3, с. 32–37. http://is.ifmo.ru/works/traverse/
52. *Шалыто А. А., Наумов Л. А.* Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов //Искусственный интеллект. 2004. № 4, с. 756–762.
53. *Werken Blissed. Java State-Machine Framework.* http://blissed.werken.com/
54. *boost::fsm. C++ library for finite state machines.* http://boost-sandbox.sourceforge.net/fsm.zip.
55. *Ninni FSM Generator.* http://nunnifsmgen.nunnisoft.ch/en/home.jsp

56. *Finite State Machine generating software.* http://fsmgenerator.sourceforge.net/
57. *Finite State Machine (FSM).* http://finsm.sourceforge.net/
58. *The State Machine Compiler.* http://smc.sourceforge.net/
59. *Duval P-Y. et al.* Evaluation of CHSM (Concurrent Hierarchical State Machine) language system. http://atddoc.cern.ch/Atlas/Notes/012/Note012-1.html
60. *Open Source Page Flow Written in Java.* http://www.manageability.org/blog/stuff/open-source-statemachine-for-user-interfaces-written-in-java
61. *StateSoft ViewControl.* http://www.statesoft.ie/products.html
62. *Java 2 Platform, Enterprise Edition (J2EE).* http://java.sun.com/j2ee/index.jsp
63. *Gurevich Y.* Evolving Algebra 1993: Lipari Guide in «Specification and Validation Methods». Oxford University Press, 1995.
64. *AsmL for Microsoft .NET.* http://research.microsoft.com/fse/asml/doc/StartHere.html
65. *Карпов Ю. Г.* Теория автоматов. СПб.: Питер, 2002.
66. *Finite State Kernel Creator.* http://fskc.sourceforge.net/
67. *UML Products By Company.* http://www.objectsbydesign.com/tools/umltools_byCompany.html
68. *Терехов А. Н., Романовский К. Ю., Кознов Д. В. и др.* REAL: Методология и CASE-средство разработки информационных систем и программного обеспечения //Программирование. 1999. № 3, с. 18–24.
69. *Nucleus UML Suite.* http://www.projtech.com/embedded/nuc_modeling.html
70. *UML Specification 1.5.* http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf
71. *A Validation Toolset for UML.* http://www.eecs.umich.edu/~wwshen/tool/tool.html

72. *IBM OCL Parser.* ftp://ftp.software.ibm.com/software/websphere/awdtools/standards/ocl-parser-03.zip
73. *Андреев Н.* Автоматическая верификация модели UML. СПб ГТУ, 2002. http://bicamp.aanet.ru/2003/papers/sectionIT/AndreevND.pdf
74. *Maller-Pedersen B.* Specification and Description Language. SDL + UML. //Telektronik. 2000. № 4, pp. 22–30. http://www.item.ntnu.no/fag/ttm4115/UMLandSDL/Telek4_2000%20SDL-UML.pdf
75. *Douglas B.* Real-Time UML: Developing Efficient Objects for Embedded Systems. MA: Addison-Wesley, 1998.
76. *Гома Х.* UML. Проектирование систем реального времени, параллельных и распределенных приложений. М.: ДМК Пресс, 2002.
77. ITU-T. *SDL combined with UML* (Z.109). Geneva. ITU-T, 2000.
78. *Specification and Design Language (SDL).* http://www.sdl-forum.org/SDL/index.htm
79. *Шалыто А. А., Туккель Н. И.* Автоматы и танки //BYTE/Россия. 2003. № 2, с. 28–32. http://is.ifmo.ru/works/tanks_new/
80. *Visio2Switch.* http://www.softcraft.ru/auto/switch/v2s.shtml
81. *Finite State Machine Editor.* http://fsme.sourceforge.net/
82. *Ваганов С.* Ускоритель разработки приложений //Открытые системы. 2004. № 6, с. 40–44.
83. *Telelogic Tau2*. http://www.telelogic.com/products/tau/tg2.cfm
84. *Rambaugh J., Jacobson I., Booch G.* The Unified Modeling Language Reference Manual. Addison-Wesley, 2005.
85. *Borland Together Architect*. http://www.borland.com/together/architect/index.html
86. *Фаулер М.* UML. Основы. СПб.: Символ-Плюс, 2004.

87. *Фаулер М.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2002.

88. *Riehle D., Fraleigh S., Bucka-Lassen D., Omorogbe N.* The Architecture of a UML Virtual Machine / Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ’01). ACM Press, 2001.

89. *Matilda UML Virtual Machine.* http://dssg.cs.umb.edu/projects/umlvm/

90. *Carter K. iUML.* http://www.kc.com/products/iuml/index.html

91. *Глушков В.М.* Синтез цифровых автоматов. М.: Физматгиз, 1962.

92. *MetaObject Facility Core Speification Version 2*.0. http://www.omg.org/technology/documents/formal/MOF_Core.htm

93. *Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.

94. *Верещагин Н., Шень А.* Лекции по математической логике и теории алгоритмов. Часть 2. Языки и исчисления. М.: МЦНМО, 2000.

95. *Parr T. J., Quong R. W.* ANTRL: A Predicated-LL(k) Parser Generator // Software – Practice And Experience. 1995, №25(7), pp. 789–810.

96. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. 2007. Т. 14. № 1, с. 3–14.

97. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002

98. *Holzmann G.* The Model Checker Spin. IEEE Trans. on Software Engineering, Vol. 2. 1997, No. 5, pp. 279–295.

99. *Robby, Dwyer M., Hatcliff J.* Bogor: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp 3–22.

100. *Robby, Dwyer M., Hatcliff J.* Bogor: An Extensible and Highly-Modular Model Checking Framework /In the Proceedings of the Fourth Joint Meeting

of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003).

101. *Velocity*. *Java*-based template engine. http://jakarta.apache.org/velocity/index.html

102. *Fruchterman T. M. J., Reingold E. M.* Graph Drawing by Force Directed Placemen // Software – Practice and Experience. 1991. № 21(11), pp. 1129–1164.

103. Java Debug Wire Protocol. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-spec.html

104. Java Networking Features. http://java.sun.com/j2se/1.5.0/docs/guide/net/index.html

105. *Шалыто А. А., Штучкин А. А.* Совместное использование теории построения компиляторов и SWITCH-технологии (на примере построения калькулятора). СПбГУ ИТМО. 2003. http://is.ifmo.ru/projects/calc/

106. *Легалов А. И.* Основы разработки трансляторов. Использование диаграмм Вирта для представления динамически порождаемых конечных автоматов, распознающих КС(1) грамматику. http://softcraft.ru/translat/lect/t08-04.shtml

107. *Шалыто А. А., Туккель Н. И., Шамгунов Н. Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода. //Телекоммуникации и информатизация образования. 2002. № 5, с. 52–68. http://is.ifmo.ru/works/recurse/

108. *Акимов О.Е.* Дискретная математика: логика, группы, графы. М.: Лаборатория Базовых Знаний. 2003.

109. *Harrison R*. Symbian OS C++ for Mobile Phones. John Wiley & Sons, 2003.

110. *Fowler M.* Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html

(Фаулер М. Языковой инструментарий: новая жизнь языков предметной области. http://www.maxkir.com/sd/languageWorkbenches.html)

111. *Dmitriev S.* Language Oriented Programming: The Next Programming Paradigm //onBoard. 2005. № 2. (Дмитриев С. Языково-ориентированное программирование: следующая парадигма //RSDN Magazine. 2005. № 5).
112. *Ward M.* Language Oriented Programming //Software – Concepts and Tools. 1994. 15.
113. *Luo Z.* Computation and Reasoning: A Type Theory for Computer Science. Oxford University Press, 1994.
114. *Simonyi C.* The Death of Computer Languages, the Birth of Intentional Programming /The Future of Software. Univ. of Newcastle upon Tyne, England, Dept. of Computing Science, 1995.
115. *Kachelaev D., Khasanzyanov B., Yaminov B., Shalyto A.* Instrumental Tool for Automata Based Software Development UniMod 2 /Proceeding of the Second Spring Young Researchers' Colloquium on Software Engineering. V. 1. SPbSU. 2008, pp. 55–58.