Статья опубликована в журнале «Программирование». 2007. № 6, с. 65–80.

УДК 681.3.06

ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ДЛЯ ПОДДЕРЖКИ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

© В. С. Гуров[1], М. А. Мазин[1], А. С. Нарвский[1], А. А. Шалыто[2]

[1] *Компания eVelopers Corporation,*

197110, Санкт-Петербург, Б.-Разночинная 14-5

[2] *Санкт-Петербургский государственный университет информационных технологий, механики и оптики*

197101, Санкт-Петербург, Саблинская ул. 14

E-mail: vgurov@evelopers.com

Поступила в редакцию 01.08.2006 г.

В статье предлагается метод проектирования и реализации реактивных объектно-ориентированных программ с явным выделением состояний. Метод основан на использовании автоматного программирования (SWITCH-технология) и UML-нотации. Описывается базирующееся на этом методе инструментальное средство UniMod, являющееся встраиваемым модулем для платформы Eclipse.

1. ВВЕДЕНИЕ

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*) [2].

Это направление включает в себя «Разработку, управляемую моделями» (*Model-Driven Development, MDD*), которое может быть названо также «Проектирование на базе моделей» (*Model-Driven Design*) [3, 4]. Вариантом *MDD* является «Архитектура, управляемая моделями» (*Model-Driven Architecture, MDA*) [5], предложенная и развиваемая консорциумом *Object Management Group* (*OMG*).

При применении *MDA* модели программных систем представляются с помощью «Унифицированного языка моделирования» (*Unified Modeling Language, UML*) [6].

Если в течение ряда лет этот язык использовался только для представления моделей, то в последнее время все большую популярность приобретает идея *исполняемого UML* [7, 8]. 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 [9], «внятно» не сказано, как связывать статические диаграммы с динамическими.

В настоящее время, несмотря на наличие большого числа инструментальных средств для автоматического преобразования поведенческих диаграмм (диаграмм состояний) в код на различных языках программирования [10], в широко известных средствах моделирования, например *Sun Studio Enterprise* [11], такая функциональность отсутствует.

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

Ивар Якобсон, один из создателей языка моделирования *UML*, в докладе «Четыре основные тенденции в разработке программного обеспечения (ПО)» [12] перечислил важнейшие, по его мнению, направления развития процесса разработки ПО.

Он отметил, что технологическая база разработки объектно-ориентированного ПО, состоящая из языка *UML* и стандартного процесса разработки *Rational Unified Process* (*RUP*) [13], достаточно известна. По словам И. Якобсона, язык *UML* преподается более чем в 1000 университетах мира. По его мнению, следующим шагом должно стать широкое внедрение *UML* и *RUP*.

Применительно к тематике настоящей статьи из четырех тенденций, перечисленных И. Якобсоном, отметим две.

1. Исполняемый *UML*. В настоящее время *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 может быть достигнуто за счет непосредственной интерпретации модели.

2. Процесс разработки ПО должен быть активным. Существующие средства разработки требуют длительного времени для их изучения. И. Якобсон считает, что средства разработки 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*.

Признание многими ведущими в области разработки ПО фирмами того факта, что программы необходимо не писать «на авось» (как сказал по-русски на конференции

«Microsoft Research Academic Days in St.-Petersburg, April 21–23, 2004» создатель языка *Eiffel* Бертран Мейер), а проектировать, повышая уровень абстракции средств разработки, привело к появлению таких направлений создания ПО, как проектирование на базе моделей и визуальное конструирование программ [14].

2. РЕАКТИВНЫЕ СИСТЕМЫ

Широким классом программных систем являются реактивные системы – системы, выполняющие определенные действия в ответ на внешние события. В работе Д. Харела [15] показано, что для моделирования таких систем хорошо подходит расширение диаграмм переходов конечных автоматов (например, за счет применения вложенных состояний), названное «диаграммы состояний» (*Statechart)*. Для построения таких диаграмм и генерации кода по ним созданы инструментальные средства, многие из которых перечислены в [10]. В этой работе, в частности, упомянуты такие инструменты как *I-Logix Statemate* и *Rhapsody (*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/) [16–21].

Существуют также и другие инструменты для генерации кода по этим диаграммам (http://en.wikipedia.org/wiki/List_of_UML_tools), например, описанное в работе [22].

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**.

Из изложенного следует, что с помощью одних инструментальных средств можно построить статическую часть, а с помощью других – динамическую. Поэтому вопрос о создании инструментальных средств для разработки объектно-ориентированных программ в целом остается открытым.

3. ИСПОЛНЯЕМЫЙ UML

Для решения указанной проблемы, как отмечалось выше, ведутся работы по созданию *исполняемого UML*, в котором должны быть объединены статические и динамические диаграммы.

Одним из подходов к решению этой проблемы является разработка *виртуальной машины UML* [23–25].

В проекте [24] 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 [25] 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* [26]. 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*.

Проект *UniMod* (http://unimod.sourceforge.net), который лежит в основе настоящей работы, имеет открытый исходный код. Он представляет интерес как новая перспективная разработка, повышающая эффективность создания ПО и его качество.

Далее в статье описывается исполняемый графический язык, основанный на использовании *UML*-нотации, метод построения автоматных программ на его основе и инструментальное средство *UniMod* для поддержки этого метода.

4. ИСПОЛНЯЕМЫЙ ГРАФИЧЕСКИЙ ЯЗЫК И МЕТОД ПОСТРОЕНИЯ АВТОМАТНЫХ ПРОГРАММ НА ЕГО ОСНОВЕ

В работе [27] 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 [28], а потом и для объектно-ориентированных [29].

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 управляемых объектов к системе управления. В предлагаемом подходе система управления реализуется в виде системы взаимодействующих конечных автоматов, каждый из которых является структурным автоматом и имеет несколько входов и выходов [27].

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).

Авторами настоящей работы предлагается, сохранив автоматный подход, использовать *UML*-нотацию при построении диаграмм в рамках *SWITCH*-технологии. При этом, используя нотацию *UML*-диаграмм классов, строятся схемы связей автоматов, которые определяют интерфейс автоматов, а графы переходов строятся с помощью нотации *UML*-диаграммы состояний. При наличии нескольких автоматов их схема взаимодействия не строится, а все они изображаются на диаграмме классов. Диаграмма классов (как схема связей) и диаграммы состояний образуют предлагаемый графический язык.

Для проектирования программ на основе этого языка авторами предлагается следующий метод:

- на основе анализа предметной области разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними;
- в отличие от традиционных для объектно-ориентированного программирования подходов [9], 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 по командам от автоматов. 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;

- используя нотацию диаграммы классов, строится схема связей автоматов, задающая интерфейс каждого их них. На этой схеме слева отображаются 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;
- схема связей, кроме задания интерфейсов автоматов, выполняет функцию, характерную для диаграммы классов — задает объектно-ориентированную структуру программы;
- каждый объект управления содержит два типа методов, реализующих входные переменные (`xj`) и выходные воздействия (`zk`);
- для каждого автомата с помощью нотации диаграммы состояний строится граф переходов типа *Мура-Мили*, в котором дуги могут быть помечены событием (`ei`), булевой формулой из входных переменных и формируемыми на переходах выходными воздействиями;
- в вершинах могут указываться выходные воздействия, выполняемые при входе в состояние и имена вложенных автоматов, которые активны, пока активно состояние, в которое они вложены;
- кроме вложенности автоматы могут взаимодействовать по вызываемости. При этом вызывающий автомат передает вызываемому событие, что и указывается на переходе или в вершине в виде выходного воздействия. Во втором случае посылка события вызываемому автомату происходит при входе в состояние;
- каждый автомат имеет одно начальное и произвольное количество конечных состояний;
- 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;
- 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;
- каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования. Источники событий также реализуются вручную;
- 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.

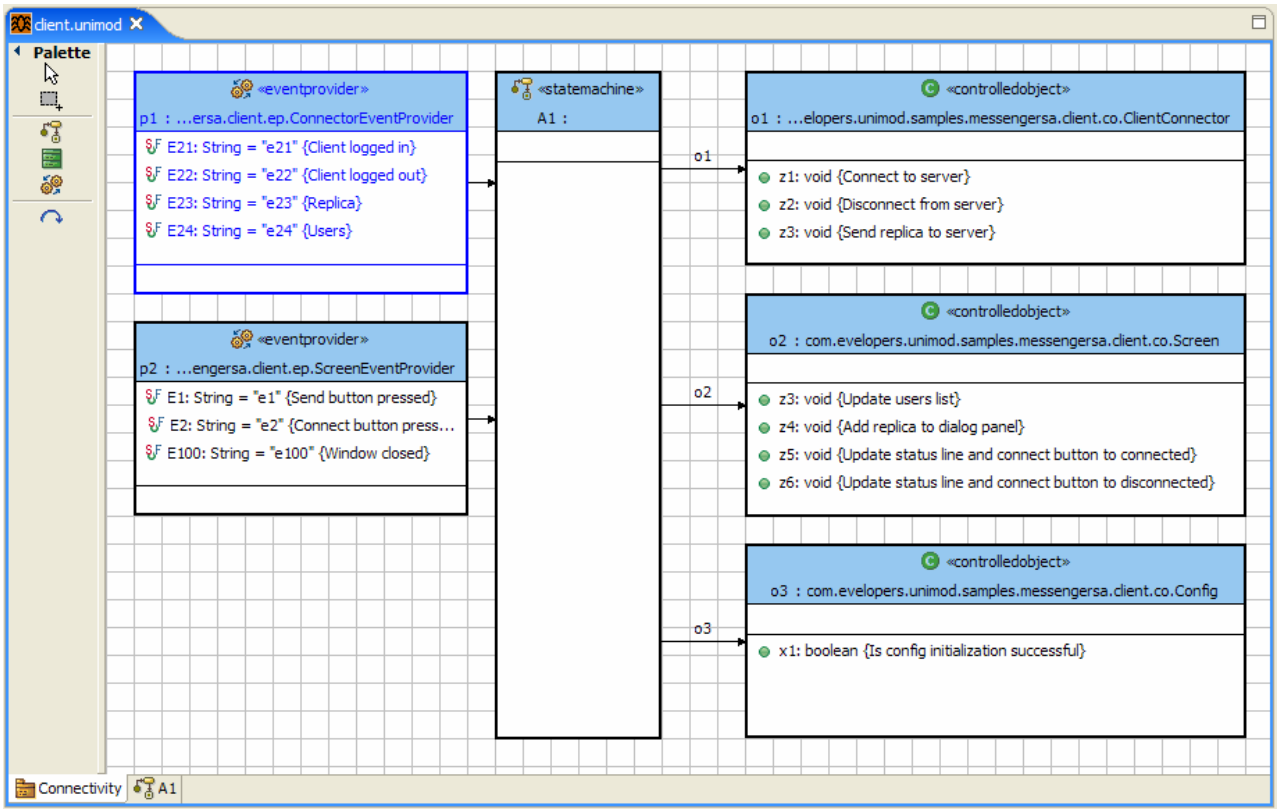

Рис. 1. Пример схемы связей автомата

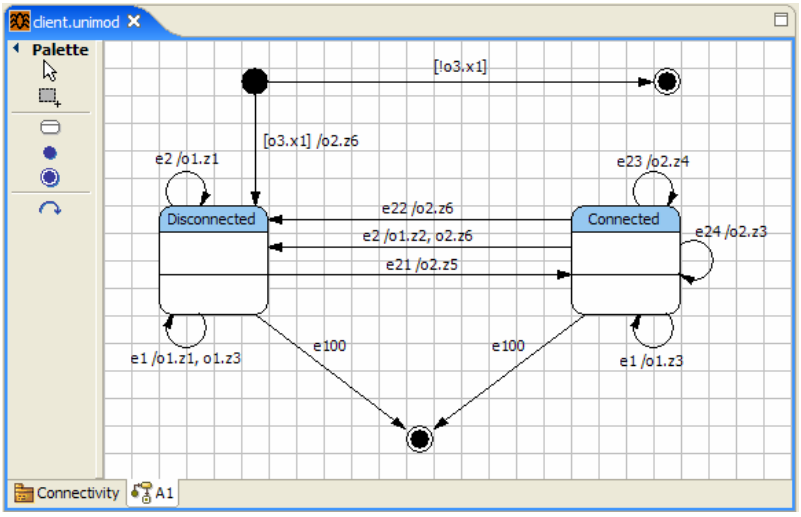

Рис. 2. Пример графа переходов автомата

Опишем синтаксис и операционную семантику предлагаемого графического языка.

4.1. Синтаксис

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)* [30].

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

Статическая модель системы в рамках проекта *UniMod* состоит из одной диаграммы классов, на которой изображаются классы со следующими стереотипами: «*EventProvider*» – источник событий, «*StateMachine*» – автомат и «*ControlledObject*» – объект управления. Между указанными стереотипами 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. На каждой из этих диаграмм могут изображаться начальные, конечные, простые и составные состояния. Составные состояния могут содержать внутри себя состояния любых других типов.

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) – булева формула.

Внутри простых состояний могут быть указаны действия, выполняемые при входе в состояние, которые записываются в виде строки. Например,

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

Действия, выполняемые при выходе из простых состояний, описываемый язык не поддерживает.

Как отмечалось выше, внутри простых состояний также может указываться и список вложенных автоматов.

UML-состояния с параллельными регионами, отражающими параллелизм, 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 (у нас – автоматов – прим. авторов)» [31].

4.2. Операционная семантика

Зададим операционную семантику модели системы, построенной описанным выше образом, которая позволяет:

- при запуске модели, инициализируются все источники событий и объекты управления. После этого источники событий начинают воздействовать на связанные с ними автоматы;
- каждый автомат начинает свою работу из начального состояния, а заканчивает — в одном из конечных;
- при получении события автомат выбирает все исходящие из текущего состояния переходы, помеченные символом этого события;
- автомат перебирает выбранные переходы и вычисляет булевы формулы, записанные на них, до тех пор, пока не найдет формулу со значением `true`;
- если переход с такой формулой найден, 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;
- если среди выходных воздействий встречается вызываемый автомат, то он вызывается с соответствующим событием;
- если переход не найден, то автомат продолжает поиск перехода у родительского состояния – состояния, в которое вложено текущее состояние;
- при переходе в конечное состояние автомат останавливает все источники событий. На этом работа системы завершается.

Более подробное и формальное описание операционной семантики приведено в работе [32].

Описав исполняемый графический язык на основе *UML*-нотации, его операционную семантику и метод его использования, перейдем к описанию инструментального средства для поддержки этого метода.

5. *UNIMOD* — ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ДЛЯ ПОДДЕРЖКИ АВТОМАТНОГО ПРОГРАМИРОВАНИЯ

Инструментальное средство *UniMod* обеспечивает разработку и выполнение автоматных программ. Он позволяет создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схемам связей и графам переходов.

Проектирование программ с использованием этого средства, как отмечено выше, 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.

5.1. Интерпретация

Интерпретационный подход реализует *виртуальную машину UML.*

На рис. 3 приведена структурная схема для интерпретационного подхода.

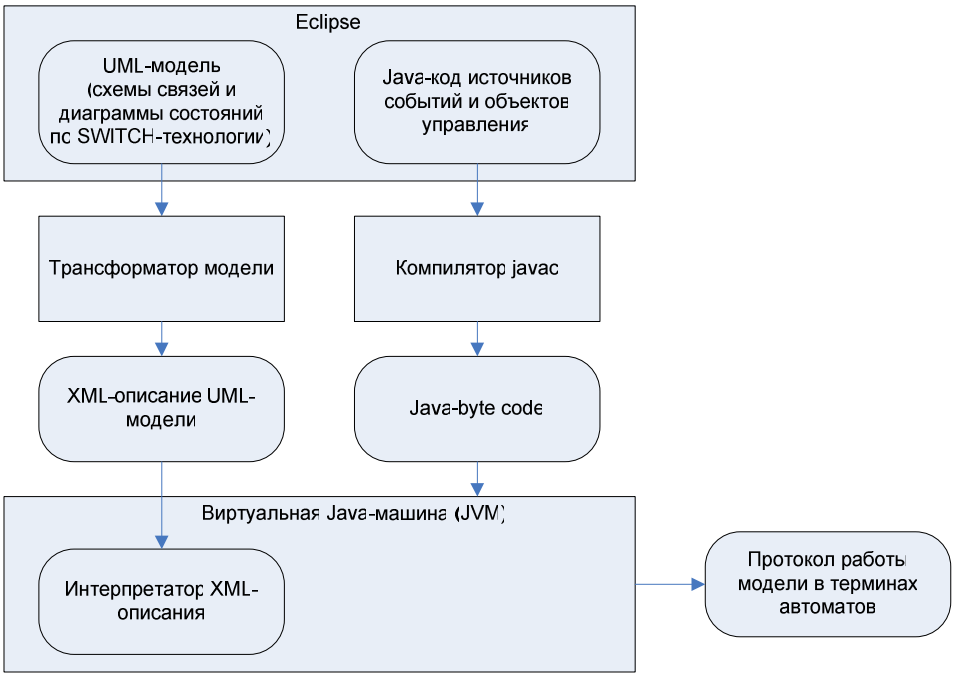

Рис. 3. Структурная схема интерпретационного подхода

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.

5.2. Компиляция

На рис. 4. приведена структурная схема для компилятивного подхода.

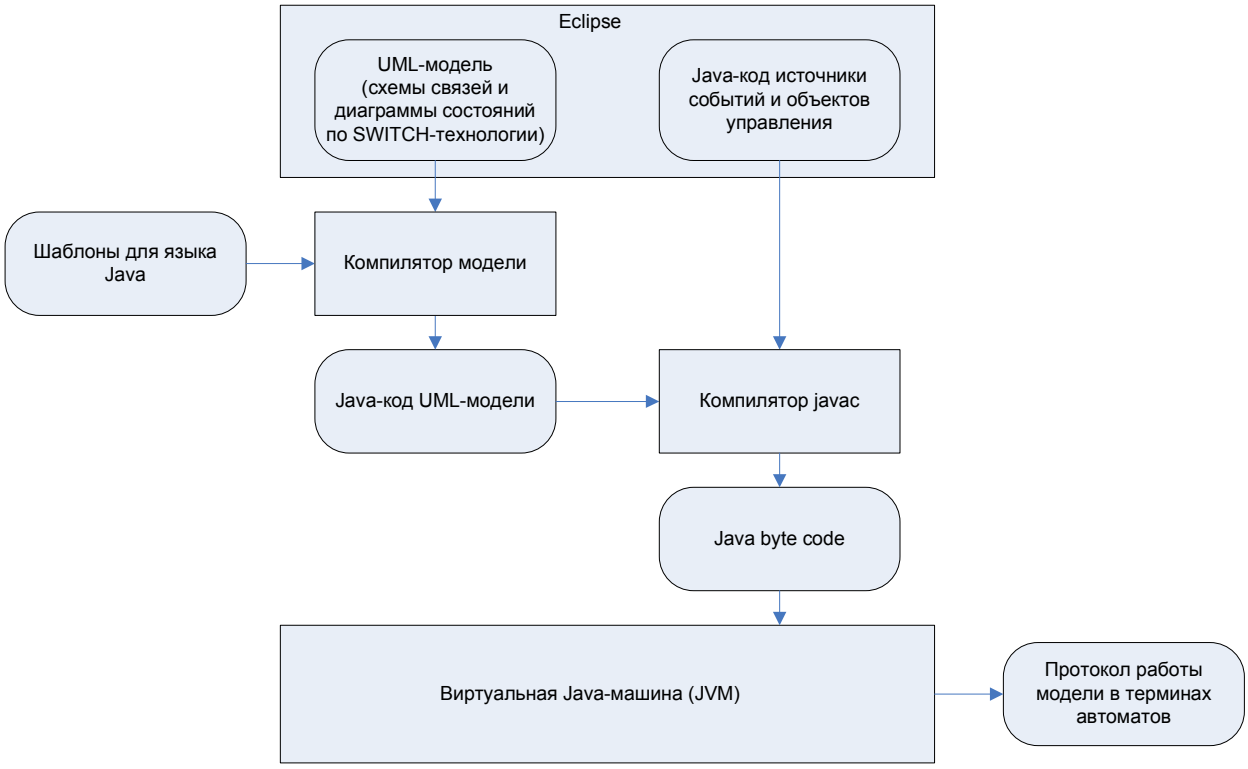

Рис. 4. Структурная схема компилятивного подхода

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 [33]. Это позволяет адаптировать компилятивный подход для языков программирования, отличных от языка *Java*, что и было сделано авторами для языка *C++*, применяемого при создании приложений для мобильных устройств (разд. 7).

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

6. 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 [34] кода;
- исполнение и отладка программы внутри среды разработки.

В английском языке эти возможности называются "*code assist*". При создании модуля для платформы *Eclipse* авторы реализовали указанные возможности для редактирования диаграмм.

6.1. Валидация модели

Для текстовых языков программирования редакторы осуществляют проверку принадлежности программы к заданному языку и выделяют (подсвечивают) места в коде, содержащие синтаксические ошибки. К семантическим ошибкам в текстовых языках программирования относится, например, вызовы несуществующих методов, некорректное приведение типов.

В стандарте на язык *UML* синтаксис и семантика диаграмм определяются набором ограничений, записанных на языке объектных ограничений (*Object Constraint Language*). Данный набор ограничений должен удовлетворяться для любой правильно построенной диаграммы. Именно на этих ограничениях и основана проверка синтаксиса и семантики диаграмм.

Авторами предлагается расширить множество ограничений следующим образом:

- все состояния на диаграмме состояний должны быть достижимы;
- множество исходящих переходов для любого состояния должно быть полно и непротиворечиво. Полнота означает, что для каждого события дизъюнкция охраняющих условий на всех дугах, исходящих из рассматриваемого состояния тождественно равна единице. Непротиворечивость означает, что при обработке любого события не существует переходов, которые могут быть выполнены одновременно (используются только детерминированные конечные автоматы).

Описанные ограничения задают условия корректности. Проверка корректности диаграмм происходит следующим образом. В фоновом режиме запускается процесс, который при любом изменении диаграммы, проверяет указанные условия. При нахождении ошибки некорректный элемент на диаграмме выделяется цветом. На рис. 5 приведен пример диаграммы с недостижимым состоянием.

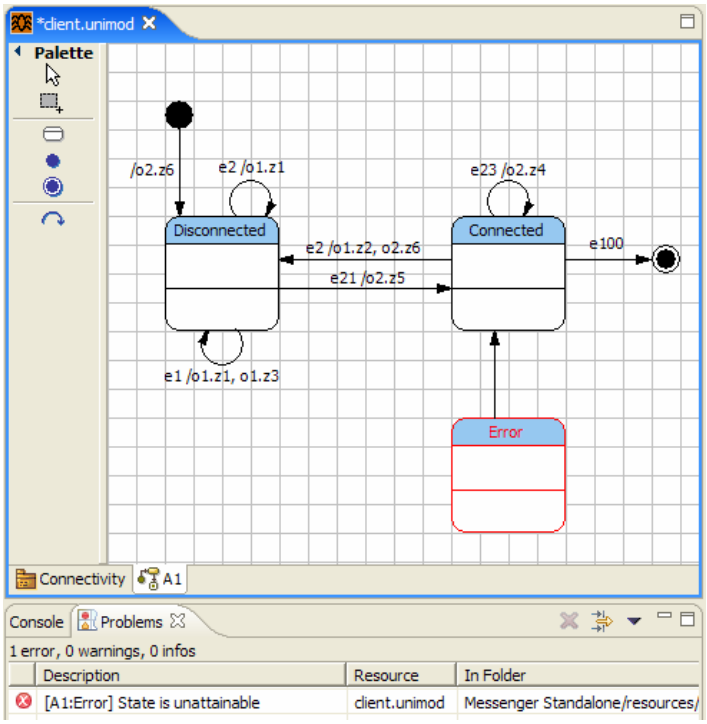

Рис. 5. Недостижимое состояние на графе переходов

6.2. 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, представленного на рис. 5, пользователю будет предложено добавить переход в это состояние из любого достижимого (рис. 6).

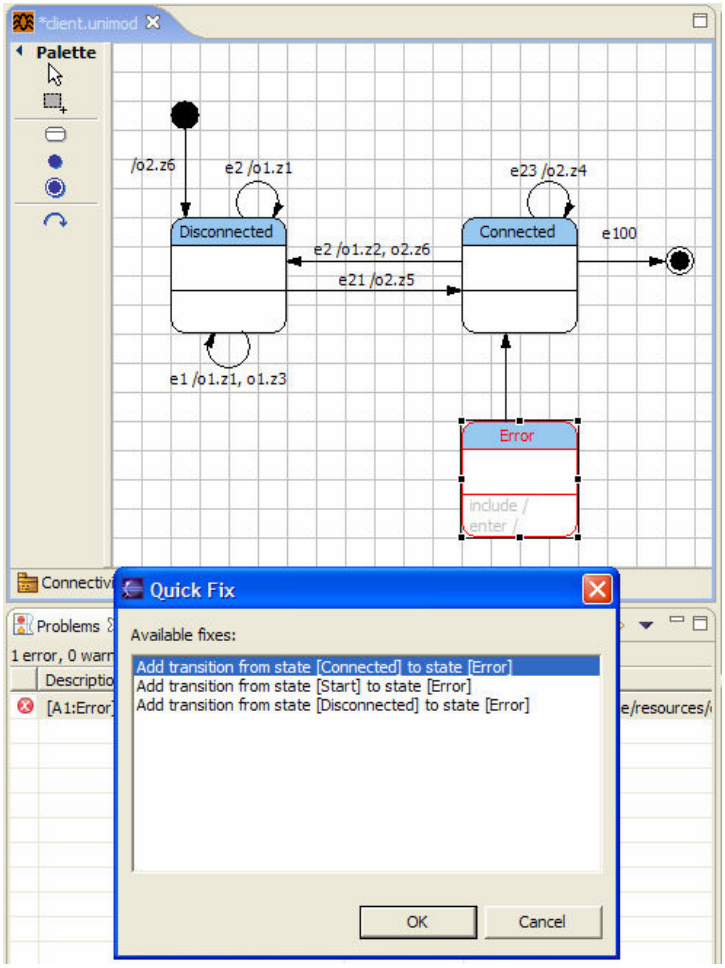

Рис. 6. Предлагаемые варианты исправления ошибки на диаграмме

6.3. Форматирование

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

Аналогом форматирования кода применительно к диаграммам, по мнению авторов, 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 [35], который дает удовлетворительные результаты, при необходимости улучшаемые вручную.

6.4. 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*, *Basic*).

Подобные решения применяются и для предлагаемого графического языка. Основными вариантами исполнения являются второй и третий. Они подробно описаны в разд. 5. В отдельных случаях (разд. 7) может применяться и первый подход.

6.5. Отладка модели

Традиционно отладка программ представляет собой трассировку программного кода оператор за оператором с одновременным анализом значений переменных.

Для графической автоматной модели предложена «графическая отладка» – трассировка графа переходов с анализом текущего состояния, событий и значений входных переменных, помечающих анализируемый переход. При необходимости возможна текстовая отладка кода входных переменных и выходных воздействий.

На рис. 7 приведен пример отладочной сессии в инструменте *UniMod*.

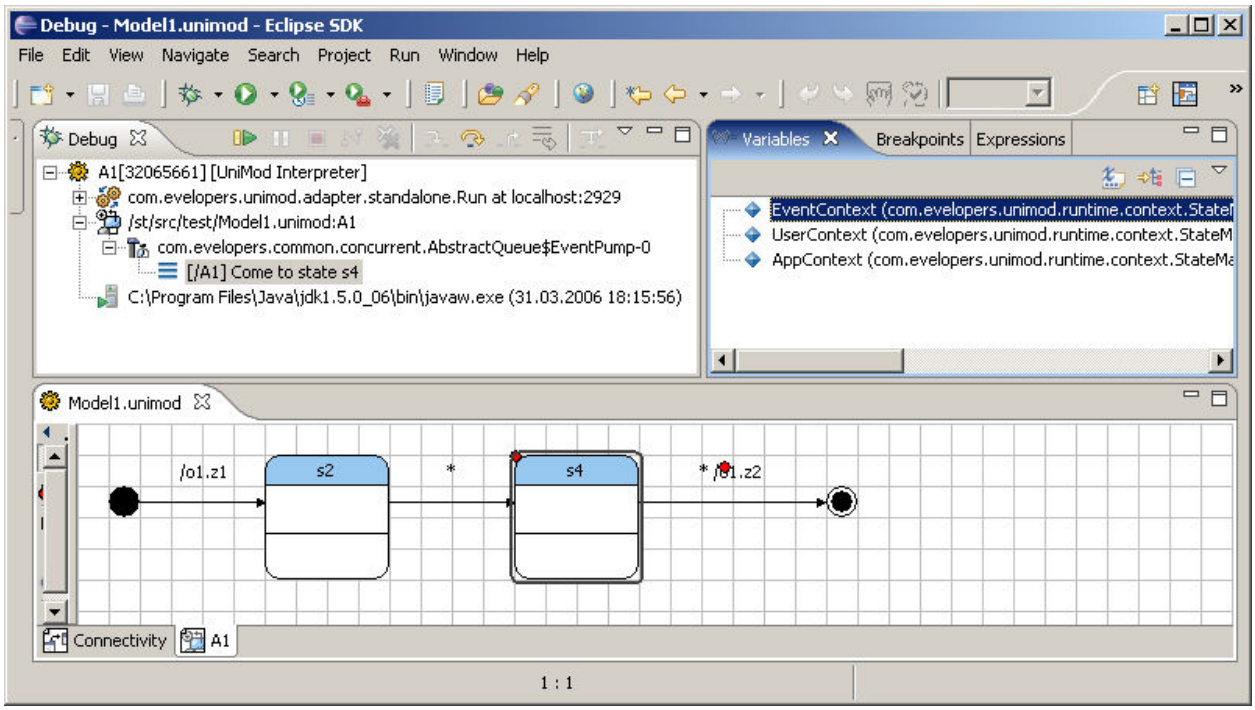

Рис. 7. Отладочная сессия

В нижней части рисунка показана отлаживаемая модель. На ней маленькими кружками указываются точки останова. Состояние, выделенное рамкой – текущая позиция отладчика.

7. ИСПОЛЬЗОВАНИЕ ПРЕДЛАГАЕМОГО ПОДХОДА НА ПРИМЕРЕ МОБИЛЬНЫХ УСТРОЙСТВ

Выше было сказано, что рассматриваемое инструментальное средство базируется на использовании языка *Java*. Однако в ряде случаев, например для мобильных устройств, для обеспечения требуемого быстродействия приходится использовать язык *C++*. При этом возможно использовать только компилятивный подход.

На рис. 8 показано, как изменяется структурная схема для компилятивного подхода (рис. 4) в случае применения языка *C++* для мобильной платформы *Symbian* (http://www.symbian.com).

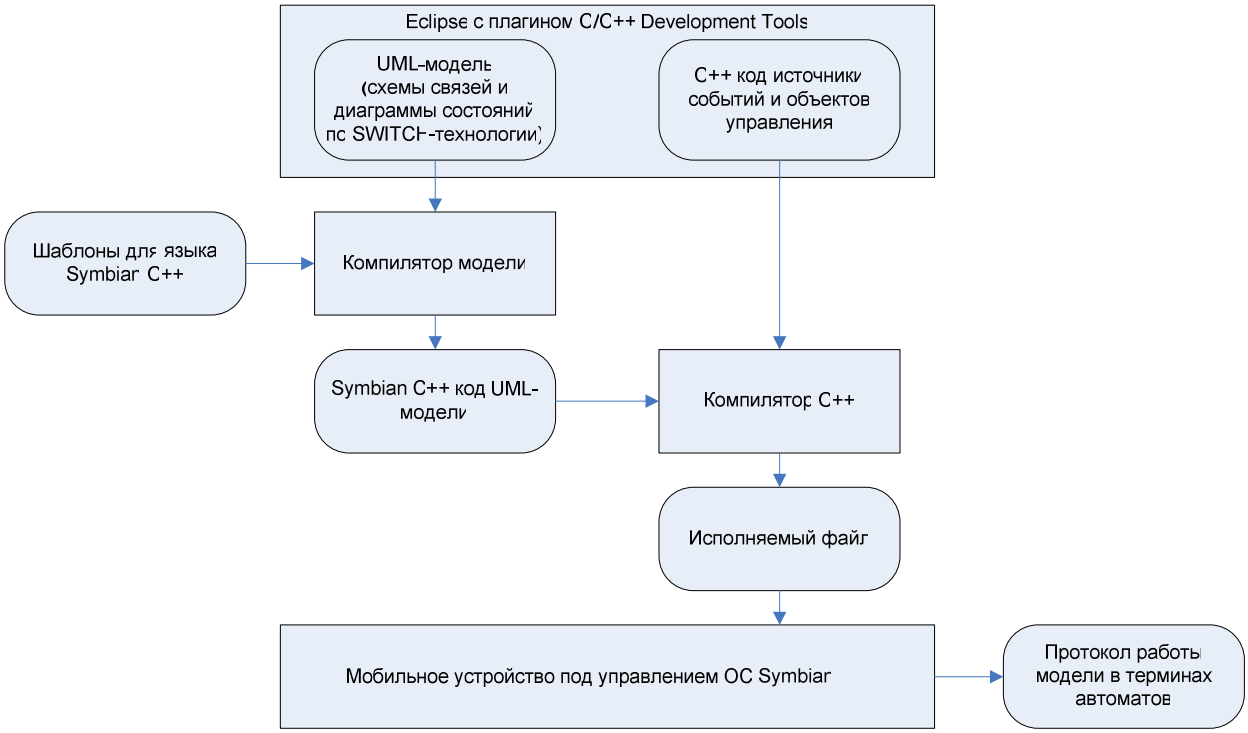

Рис. 8. Структурная схема компилятивного подхода при использовании *Symbian C++*

Обратим внимание, что **разработка** программы в этом случае выполняется как и для языка *Java*, но с единственным отличием – применяются шаблоны не для языка *Java*, а для языка *C++*. Пример разработки программы на языке *C++* с использованием инструментального средства *UniMod* опубликован по адресу http://is.ifmo.ru/science/MD-Mobile.pdf

На рис. 9 в качестве примера приведена диаграмма состояний основного автомата для автоответчика мобильного телефона. Эта диаграмма содержит все типы синтаксических конструкций, описанных выше.

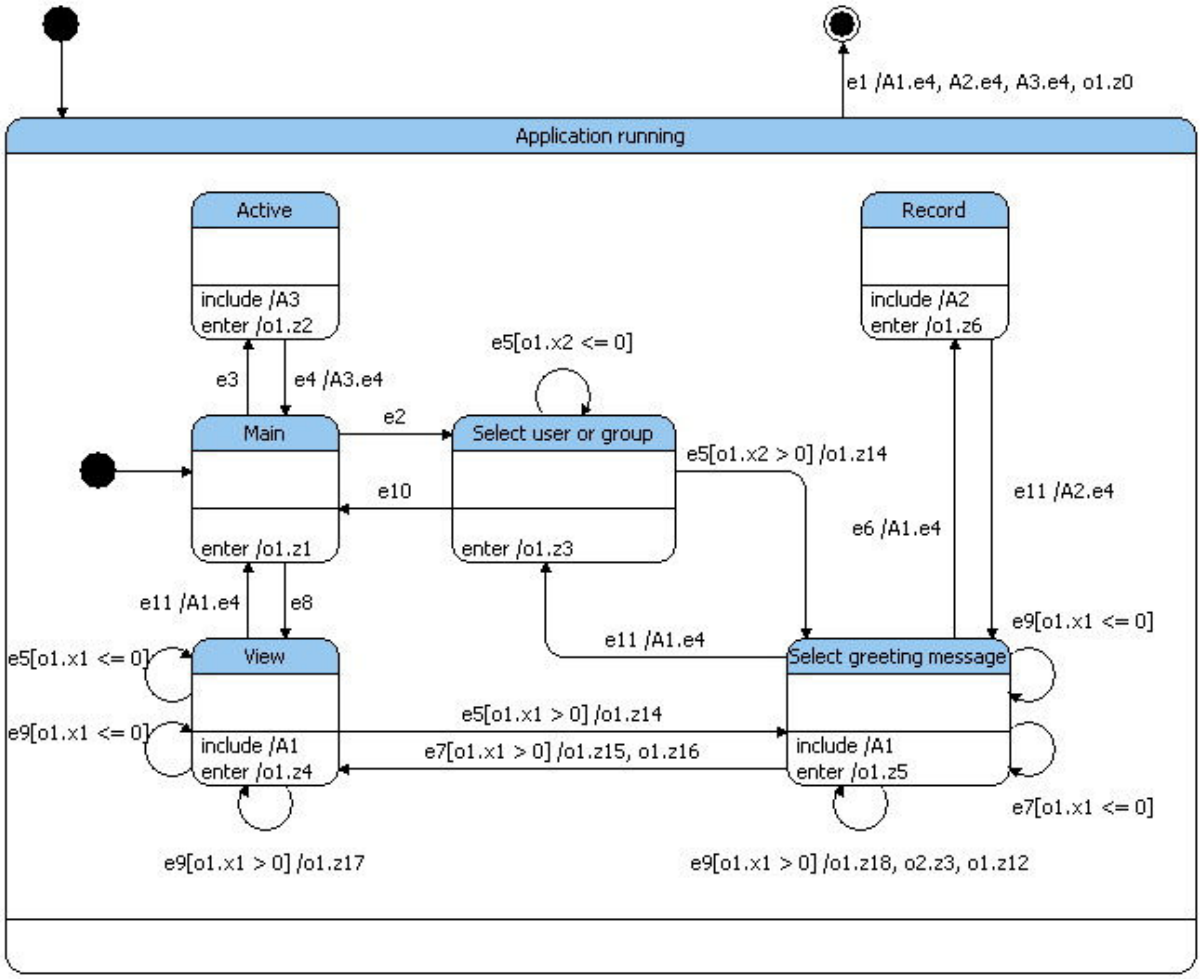

Рис. 9. Диаграмма состояний автоответчика мобильного телефона

Таким образом, можно утверждать, что, несмотря на то, что *Eclipse* и *UniMod* ориентированы на язык программирования *Java*, применение шаблонов при компиляции модели делает инструмент *UniMod* многоязыковой платформой. При этом для языков, отличных от *Java*, возможность графической отладки модели теряется.

ЗАКЛЮЧЕНИЕ

В статье излагается подход к созданию инструментального средства для поддержки автоматного программирования.

Этот подход позволяют:

- сокращать объем кода на текстовом языке программирования за счет использования графического языка программирования;
- строить предложенные в *SWITCH*-технологии схемы связей и графы переходов в *UML*-нотации диаграмм классов и диаграмм состояний соответственно, и включать их в проектную документацию [36];
- формально и наглядно описывать поведение программ и модифицировать их, изменяя, в большинстве случаев, только графы переходов;
- упростить сопровождение проектов вследствие повышения централизации логики программ.

Из изложенного следует, что предлагаемый подход обладает следующими преимуществами по сравнению с аналогами:

- в модели допускается использовать систему взаимосвязанных автоматов, что позволяет декомпозировать поведение сложной задачи на подзадачи. При этом отметим, что каждое состояние также осуществляет декомпозицию подзадачи, выделяя только те входные и выходные воздействия, которые с ним связаны;
- программы, создаваемые на основе изложенного метода, приспособлены к верификации по построению. Это объясняется тем, что при верификации поведения с использованием метода *Model Checking* [37] для программ, написанных традиционно, должны строиться модели, например, в виде системы переходов, а при автоматном программировании модели в виде графов переходов задаются при спецификации программ;
- структура автоматных программ, в которых функции входных и выходных воздействий практически полностью отделены от логики программ, делает практичным верификацию этих функций на основе формальных доказательств с использованием пред- и постусловий [38, 39];
- наряду с вложенными состояниями используются также вложенные автоматы, число и глубина вложения которых не ограничены;
- также автоматы могут взаимодействовать при помощи вызовов, выполняемых за счет посылки им соответствующих событий на переходах или при входе в состояние;
- наряду с компилятивным подходом имеется возможность интерпретировать модели. В этом случае «исходным кодом» являются и сами диаграммы;
- интерактивная валидация модели позволяет локализовать множество синтаксических, семантических и логических ошибок еще на стадии моделирования;
- структура программы (*framework*) задается инструментальным средством, и у пользователя нет необходимости разрабатывать ее для каждого приложения заново;
- предлагаемый метод позволяет создавать программу в целом;
- проект является открытым;
- опыт использования показывает, что при использовании компилятивного подхода при достаточно сложной логике более половины кода приложения строится по модели автоматически.

Предлагаемая операционная семантика является детерминированной за счет проверки отсутствия противоречивых переходов, что не выполняется, например, в таких инструментальных средствах, как *Rational Rose* и *Borland Together* [40]. Указанный недостаток существует также и в инструменте *VisualSTATE* [20], но устраняется с помощью исследовательского инструмента *SCOPE* [19].

Исходные тексты, документация и примеры использования программного пакета *UniMod* представлены на сайте http://unimod.sourceforge.net.

В заключении отметим, что данная работа базируется на работе [28], и развивает подход, описанный в работах [41, 42].

Также отметим, что в работе [43] сказано, что языка *UML* выполняется закон «20-80». Данная статья подтверждает этот закон: из всего многообразия типов *UML*-диаграмм авторам оказалось достаточно всего двух типов диаграмм для построения программ в

целом. Это соответствует «принципу Оккама», в соответствии с которым не следует размножать сущности без необходимости.

Разработанное средство используется для обучения на кафедре компьютерных технологий СПбГУ ИТМО. Студенческие проекты, содержащие проектную документацию, публикуются на сайте http://is.ifmo.ru в разделе «*UniMod*-проекты», например [44].

Дальнейшее совершенствование инструментального средства связано с верификацией моделей, создаваемых с его помощью, а также с дальнейшим документированием проекта.

По материалам настоящей работы были сделаны доклады на ряде научно-технических конференций, например, «Методы и средства обработки информации» (МСО 2005, МГУ) (http://lvk.cs.msu.su/mco/part9.html), «Open Source Forum 2005» (http://www.opensource-forum.ru/rbio_view.php?num=41), IEEE «International Conference «110 Anniversary of Radio Invention» (http://is.ifmo.ru/articles_en/_unimod.pdf) и «Software Engineering in Russia» (SECR 2005) (http://secr.ru/rus/program/schedule.html). При чем на последней из этих конференций Ивар Якобсон в своем докладе отметил новизну и оригинальность описанного подхода [45], что позволило авторам опубликовать статью, названную «Исполняемый *UML* из России» [46].

Инструментальное средство *UniMod* начало свое «движение по миру»: так в Польше вышла книга (http://helion.pl/ksiazki/juml2.htm) про *UML 2.0*, в которой среди инструментальных средств для платформы *Eclipse* указан *UniMod*, исходные тексты которого опубликованы на диске, прилагаемом к книге, а в политехническом институте Турина (http://is.ifmo.ru/unimod_en/_torino.pdf) и в корпорации *Borland* (http://www.pcweek.ru/?ID=504874) на средство *UniMod* обратили внимание как на многообещающий проект. Имеется информация по его применению и в других организациях (например, http://is.ifmo.ru/unimod_en/_unimoduser.pdf и http://portal.acm.org/citation.cfm?id=1108473.1108476).

Подход, изложенный в настоящей работе, близок к подходу, описанному в работе [47], который используется при проектировании программного обеспечения ответственных систем длительного использования.

Настоящая статья основана на результатах работ по государственному контракту «Технология автоматного программирования: применение и инструментальные средства» (http://www.fasi.gov.ru/fcp/technika/konkurs/it/izv-it-6.doc), который выполняется в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития науки и техники» на 2002-2006 годы.

СПИСОК ЛИТЕРАТУРЫ

1. *Соммервилл И.* Инженерия программного обеспечения. М.: Вильямс, 2002. – 623 с.
2. *Кузнецов С.* Обещания и просчеты UML 2.0 //Открытые системы. № 2, с. 75–79.
3. *1st European Conference on Model-Driven Software Engineering*. Germany. 2003. http://www.agedis.de/conference/
4. *International Workshop* “e-Business and Model Based in System Design”. IBM EE/A. SPb.: SPb ETU, 2004.
5. *OMG Model Driven Architecture*. http://www.omg.org/mda/
6. *Буч Г., Рамбо Г., Якобсон И.* UML. Руководство пользователя. М.: ДМК, 2000.
7. *Mellor S., Balcer M.* Executable UML: A Foundation for Model Driven Architecture. MA: Addison-Wesley, 2002.

8. *Raistrick C., Francis P., Wright J.* Model Driven Architecture with Executable UML. Cambridge University Press, 2004.
9. *Грехем И.* Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004. – 880 с.
10. *Wikipedia*. Finite state machine. Tools. http://en.wikipedia.org/wiki/Finite_automaton#Tools
11. *Sun Studio Enpterprise*. http://developers.sun.com/prodtech/javatools/jsenterprise/reference/techart/whatis.html
12. *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
13. *Якобсон И., Буч Г., Рамбо Дж.* Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002.
14. *Новиков Ф.* Визуальное конструирование программ //Информационно-управляющие системы, 2005, № 6 с. 9–22. http://is.ifmo.ru/works/visualcons/
15. *Harel D.*, Statecharts: A Visual Formalizm for Complex Systems //Science of Computer Programming 8, 1987, pp. 231–274.
16. *I-Logix Statemate.* http://ilogix.com/sublevel.aspx?id=74
17. *XJTek AnyState*. http://www.xjtek.com/anystates/
18. *StateSoft ViewControl.* http://www.statesoft.ie/products.html
19. *SCOPE*. http://www.itu.dk/~wasowski/projects/scope/
20. *IAR Systems visualSTATE.* http://www.iar.com/p1014/p1014_eng.php
21. *The State Machine Compiler*. http://smc.sourceforge.net/
22. *Jia X. et al.* Using ZOOM Approach to Support MDD. http://se.cs.depaul.edu/ise/zoom/papers/zoom/SERP_ZOOM.pdf
23. *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.
24. *Matilda UML Virtual Machine.* http://dssg.cs.umb.edu/projects/umlvm/
25. *Kennedy Carter iUML.* http://www.kc.com/products/iuml/index.html
26. *Telelogic TAU G2.* http://telelogic.com/corp/products/tau/g2/index.cfm
27. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
28. *Шалыто А.А., Туккель Н.И.* SWITCH-технология — автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. № 5, с. 45–62. http://is.ifmo.ru/works/switch/1/
29. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. № 2, с. 69–73. http://is.ifmo.ru/works/tanks_new/.
30. *MetaObject Facility Core Speification Version 2.0.* http://www.omg.org/technology/documents/formal/MOF_Core.htm
31. *Гома Х.* UML. Проектирование систем реального времени, распределенных и параллельных приложений. М.: ДМК, 2002.

32. *Гуров В.С., Мазин М.А., Шалыто А.А.* Операционная семантика *UML*-диаграмм состояний в программном пакете *UniMod* //Труды XII Всероссийской научно-методической конференции "Телематика- 2005". СПб.: СПбГУ ИТМО. Т.1, с.74–76. http://tm.ifmo.ru/tm2005/src/224as.pdf

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

34. *Фаулер М.* Рефакторинг. Улучшение существующего кода. М.: Символ-Плюс, 2003. .

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

36. *Шалыто А.А.* Новая инициатива в программировании. Движение за открытую проектную документацию //PC Week/RE. 2003. № 40, с. 38–42. http://is.ifmo.ru/works/open_doc/

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

38. *Дейкстра Э.* Заметки по структурному программированию / Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975.

39. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Русская редакция. 2005.

40. *Borland Together. http://www.borland.com/us/products/together/index.html*

41. *Горшкова Е.А., Новиков Б.А.* Использование диаграмм состояний и переходов для моделирования гипертекста //Программирование. 2004. № 1, с. 64–80.

42. *Горшкова Е.А., Новиков Б.А. Белов Д.Д., Гуров В.С., Спиридонов С.В.* Моделирование контроллера Web-приложений с использованием UML //Программирование. 2005. № 1, с. 44–51.

43. *Эккель Б.* Философия *Java*. СПб.: Питер. 2003.

44. *Параценко Д.А., Царев Ф.Н.. Шалыто А.А* Технология моделирования одного класса мультиагентных систем на основе автоматного программирования на примере игры «Соревнование летающих тарелок». http://is.ifmo.ru/unimod-projects/plates/.

45. *Шалыто А.* Две встречи с Иваром Якобсоном. http://is.ifmo.ru/aboutus/uml_ph/, http://is.ifmo.ru/belletristic/jacobson/.

46. *Гуров В., Нарвский А., Шалыто А.* Исполняемый *UML* из России //PC Week/RE. 2005. № 26, с. 18,19. http://is.ifmo.ru/works/_umlrus.pdf

47. *Риган П., Хемилтон С.* NASA: миссия надежна //Открытые системы. 2004. № 3, с. 24–32. http://www.osp.ru/os/2004/03/045_print.htm