МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
(МИНОБРНАУКИ)

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»
(СПбГУ ИТМО)

УТВЕРЖДАЮ
Ректор СПбГУ ИТМО,
докт. техн. наук, профессор
В. Н. Васильев

____ _____ 2006 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл.	

ТЕХНОЛОГИЯ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ: ПРИМЕНЕНИЕ И ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ И УКАЗАНИЯ ПО РАЗРАБОТКЕ, БАЗИРУЮЩИЕСЯ НА ПРИНЦИПАХ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

ЧАСТЬ 4. КЛИЕНТ-СЕРВЕРНЫЕ СИСТЕМЫ

Листов 28

Декан факультета «Информационных технологий и программирования»
докт. техн. наук, профессор
_____В.Г. Парфенов

Руководитель темы
заведующий кафедрой «Технологий программирования»,
докт. техн. наук, профессор
_____ А.А. Шалыто

Ответственный исполнитель
доцент кафедры «Технологий программирования», канд. физ.-мат. наук
_____Ф.А. Новиков

2006

СПИСОК ОСНОВНЫХ ИСПОЛНИТЕЛЕЙ

Руководитель темы д.т.н., профессор	А.А. Шалыто
Ответственный исполнитель к.ф.-м.н., с.н.с.	Ф.А. Новиков
Ведущие исполнители:	
Аспирант	В.С. Гуров
Магистрант	М.А. Мазин
В работе принимали участие:	
Ведущий научный сотрудник, д.т.н., профессор	В.В. Антипов
Ведущий научный сотрудник, к.т.н.	В.В. Киселев
Ведущий научный сотрудник, к.т.н.	Р.Н. Котляр
Ведущий научный сотрудник, к.т.н.	Ю.П. Московцев
Ведущий научный сотрудник, к.т.н., доцент	В.А. Третьяков
Ведущий научный сотрудник, к.т.н.	Г.М. Файкин
Ассистент, к.т.н.	Д.Г. Шопырин
Аспирант	И.М. Аничкин
Аспирант	М.А. Казаков
Аспирант	Г.А. Корнеев
Аспирант	П.Г. Лобанов
Ведущий инженер	К.В. Вавилов
Магистрант	М.А. Коротков
Магистрант	А.П. Лукьянова
Студент	Н.И. Поликарпова
Студент	Б.М. Ярцев

АННОТАЦИЯ

Описывается методика разработки клиент-серверных приложений на основе автоматного подхода, разработанная в процессе проведения опытно-конструкторских работ по **теме (Шифр ИТ-13.4/004): «Технология автоматного программирования: применение и инструментальные средства» (VI очередь),** выполняемой в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития науки и техники» на 2002-2006 годы по государственному контракту № 02. 435.11.1009 от 01.08.2005, заключенному между Федеральным агентством по науке и инновациям и Государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на основании решения Конкурсной комиссии Роснауки № 3 (протокол от 01 августа 2005 г. № 17).

Приведено общее описание методики, описаны основные понятия, элементы и положения автоматного программирования для рассматриваемого класса задач.

Описаны четыре этапа методики: постановка задачи, создание статической модели, создание динамической модели, создание *XML*-описания автоматов и программного кода. Применение методики описано по шагам на примере создания отладчика для диаграмм состояний *UML*.

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

Настоящая методика разработана в процессе проведения опытно-конструкторских работ по **теме (Шифр ИТ-13.4/004): «Технология автоматного программирования: применение и инструментальные средства» (VI очередь),** выполняемой в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития науки и техники» на 2002-2006 годы по государственному контракту № 02. 435.11.1009 от 01.08.2005, заключенному между Федеральным агентством по науке и инновациям и Государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на основании решения Конкурсной комиссии Роснауки № 3 (протокол от 01 августа 2005 г. № 17).

В рамках указанных опытно-конструкторских работ подготовлены методические рекомендации и указания по разработке, базирующиеся на принципах автоматного программирования, для создания программного обеспечения в следующих областях приложений:

a) системы управления с повышенными требованиями к надежности функционирования;
b) встроенные системы;
c) мобильные системы;
d) клиент-серверные системы;
e) интернет-системы;
f) визуализаторы алгоритмов;
g) тренажеры;
h) симуляторы.

В данном документе изложена четвертая часть методических рекомендаций и указаний — методические рекомендации и указания по разработке для клиент-серверных систем.

Отличие данной методики от общепринятых методик объектно-ориентированного анализа и проектирования [1] состоит в том, что она определяет формальный подход к проектированию динамических аспектов программных систем, а также устраняет необходимость ручного перехода от динамической модели системы к коду на целевом языке программирования. Необходимость этого перехода устраняется, так как разработанное авторами инструментальное средство позволяет исполнять *XML*-описание автоматов, задающих поведение программы.

2. НАЗНАЧЕНИЕ, ОБЛАСТЬ ПРИМЕНЕНИЯ И ОСНОВНЫЕ ПОЛОЖЕНИЯ МЕТОДИКИ

Данная методика описывает процесс создания клиент-серверных приложений на языке *Java*.

Отличие данной методики от общепринятых методик объектно-ориентированного анализа и проектирования [1] состоит в том, что она определяет формальный подход к проектированию динамических аспектов программных систем, а также устраняет необходимость перехода от динамической модели системы к коду на целевом языке программирования. Необходимость этого перехода устраняется, так как разработанный авторами интерпретатор исполнять *XML*-описание автоматов, задающих поведение программы.

При взаимодействии между серверным и клиентским приложениями в качестве транспортного протокола предлагается использовать протокол *TCP,* который позволяет пересылать массивы байт. Для *обмена сообщениями* на более высоком (прикладном) уровне необходимо создать другой протокол, что является одной из составных частей методики.

2.1. Основные положения методики

Для проектирования клиент-серверных приложений на основе автоматного подхода предлагается следующая методика:

1. Постановка задачи.
    - В виде неформального текстового описания задается набор требований к программе.
    - Описывается протокол взаимодействия в виде набора сообщений, которыми должны обмениваться клиент и сервер.
2. Технический дизайн.
    2.1. Создание статической модели системы.
    - На основе анализа требований разрабатывается статическая модель системы, определяющая сущности и отношения между ними. В модели явно выделяются клиентская и серверные части, указываются классы, отвечающие за сетевое взаимодействие.
    - Создается модель сообщений, которыми обмениваются клиентская и серверная части.

    2.2. Создание динамической модели системы.
    - В отличие от традиционных для объектно-ориентированного программирования подходов, в рамках предлагаемого подхода сущности делятся на 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 текущего состояния воздействуют на объекты управления, переходя в новое состояние.
    - Канал связи между клиентом и сервером моделируется одним источником событий и одним объектом управления для каждой из сторон. Эти объекты управления содержат по выходному воздействию на каждый тип сообщения, которое может быть отправлено. Выходные воздействия могут вызываться на переходах и в состояниях автоматов, свя-

занных с объектом управления. Каждому типу сообщения, которое может прийти по каналу связи, соответствует событие источника событий. Такой источник событий при получении сообщения преобразует его в событие и передает автомату.

- 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.
- 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 при необходимости выходными воздействиями. Дуги могут помечаться также именами вызываемых автоматов с указанием соответствующих событий. В вершинах могут указываться выходные воздействия и имена вложенных автоматов. При этом вложенность автоматов, в отличие от вызываемости, на схеме связей не отражается. 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 наличие дуги, исходящей из такого состояния, заменяет однотипные дуги 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 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.
- Выполняется проверка корректности построенной динамической модели. При этом проверяются следующие свойства: достижимость всех состояний на графах переходов, полнота и непротиворечивость множеств переходов, соответствующих каждой из вершин.

3. Реализация.

- Каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на языке программирования *Java*. 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.

- После создания модели программы выполняется генерация *XML*-описания автоматов модели. Это описание впоследствии может передаваться на вход интерпретатора, который инициализирует источники событий, объекты управления и автоматы, описанные в модели. Интерпретатор, используя *XML*-описание автоматов, обрабатывает события от источников событий. Например, такими событиями могут быть сообщения, описанные в протоколе взаимодействия клиента и сервера. В процессе обработки событий интерпретатор выводит на консоль протокол работы автоматов. Описанный подход назван «интерпретационным».
- Другой подход к реализации приложения, который состоит в том, что по модели и коду, написанному вручную, генерируется код приложения, назван «компилятивным».

4. Тестирование и отладка
    - Отладка диаграмм состояний, созданных в процессе проектирования модели, выполняется с помощью анализа протокола работы автоматов. В случае обнаружения некорректного поведения модели, в нее вносятся исправления. После этого выполняется перегенерация *XML*-описания.
    - Отладка кода, написанного вручную, выполняется стандартными методами, используемыми при создании *Java*-приложений.

3. ПРИМЕР ПРИМЕНЕНИЯ МЕТОДИКИ

В качестве примера применения методики рассматривается разработка отладчика диаграмм состояний в пакете *UniMod.*

3.1. Постановка задачи

В работе [2] описан программный пакет *UniMod* (http://unimod.sf.net), поддерживающий SWITCH-технологию [3,4]. Этот пакет предлагает, сохранив автоматный подход, использовать *UML*-нотацию при построении моделей в рамках SWITCH-технологии. При этом, используя нотацию *UML*-диаграмм классов, строятся схемы связей автоматов, определяющие их связи с источниками событий и объектами управления. Графы переходов строятся с помощью модифицированной нотации *UML*-диаграмм состояний.

Программный пакет *UniMod* реализован на языке *Java* в виде дополнения (plug-in) для платформы *Eclipse* (http://www.eclipse.org).

Модель программы, описанная с помощью двух указанных выше типов диаграмм, должна быть дополнена кодом на целевом языке программирования, реализующим источники событий и объекты управления. После этого полученная система может быть запущена в интерпретационном или компилятивном режиме.

Несмотря на наличие в пакете *UniMod* встроенных средств для проверки корректности модели, семантические 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. 1 показана предполагаемая архитектура графического отладчика. Отладчик должен базироваться на операционной семантике, описанной в работе [5].

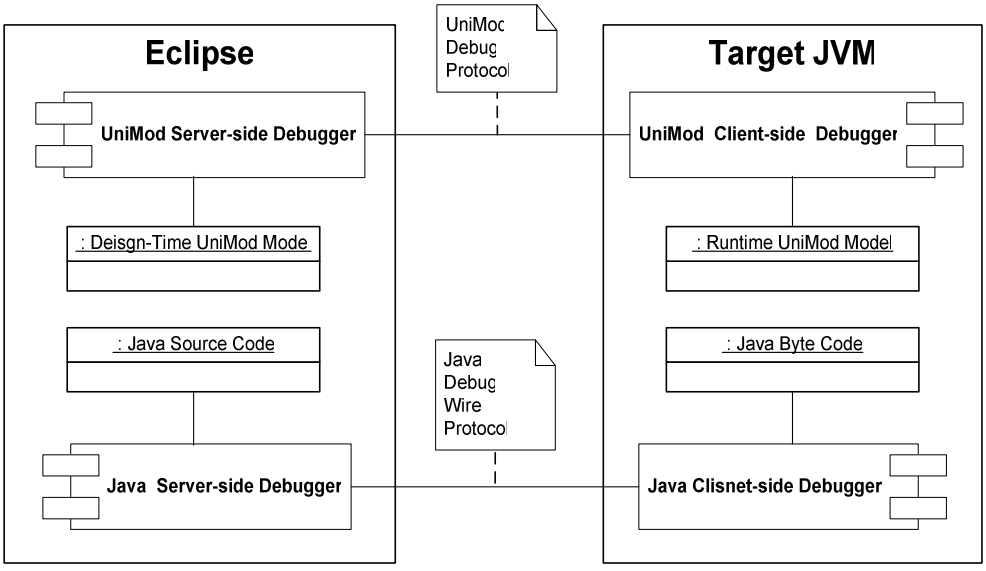

Рис. 1. Архитектура графического отладчика

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* [6].

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

1. При старте *UniMod Client-side Debugger* приостанавливает исполнение модели, создает серверный сокет [7] и ожидает присоединения к нему *UniMod Server-side Debugger.*
2. После установления соединения *UniMod Client-side Debugger* ожидает получения списка точек останова, возможно пустого.
3. После получения списка точек останова *UniMod Client-side Debugger* регистрирует их и возобновляет исполнение модели.
4. Каждый шаг исполнения модели контролируется и при достижении точки останова *UniMod Client-side Debugger* приостанавливает исполнение модели, а также информирует об этом событии *UniMod Server-side Debugger.*
5. *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 *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 *UniMod Client-side Debugger.*

Из приведенного выше регламента следует, что *UniMod Server-side Debugger* и *UniMod Client-side Debugger* взаимодействуют за счет посылки команд и извещений о событиях.

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

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

Таблица 1. Команды сервера

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

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

Таблица 2. События отладчика

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.

3.2. Технический дизайн

3.2.1. Статическая модель системы

На рис. 2 показана статическая модель системы. 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 нее 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, которое будет описано далее и реализовано в виде системы взаимодействующих автоматов.

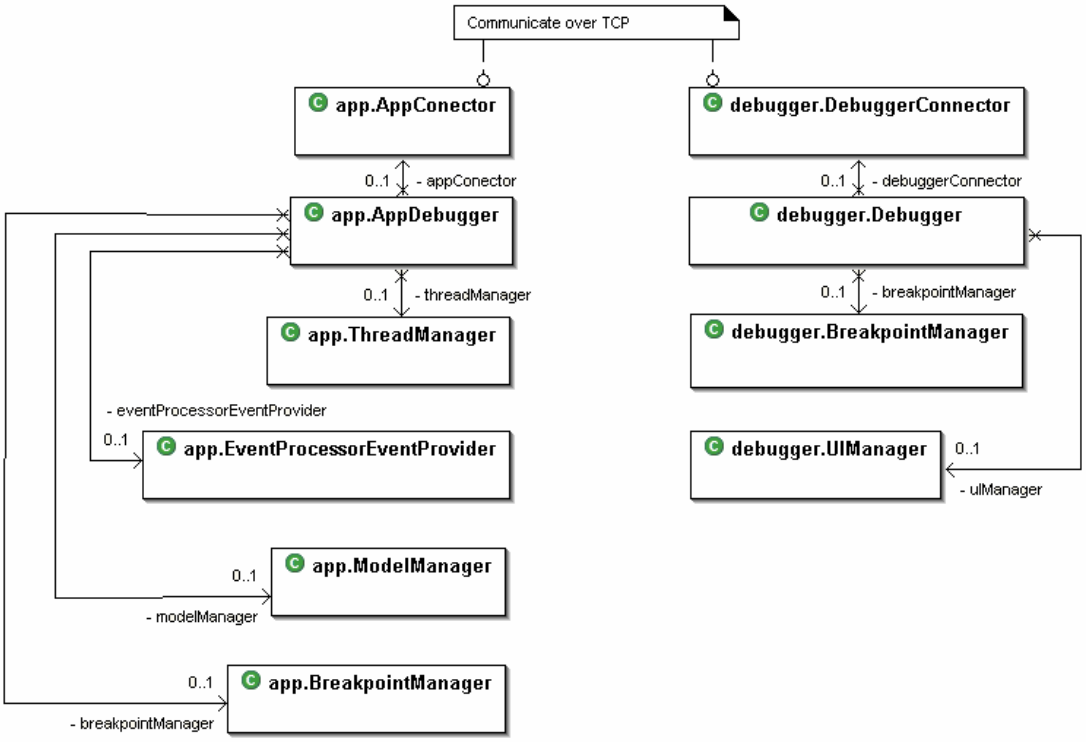

Рис. 2. Статическая модель системы

На рис. 3 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 табл. 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 точках останова.

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

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

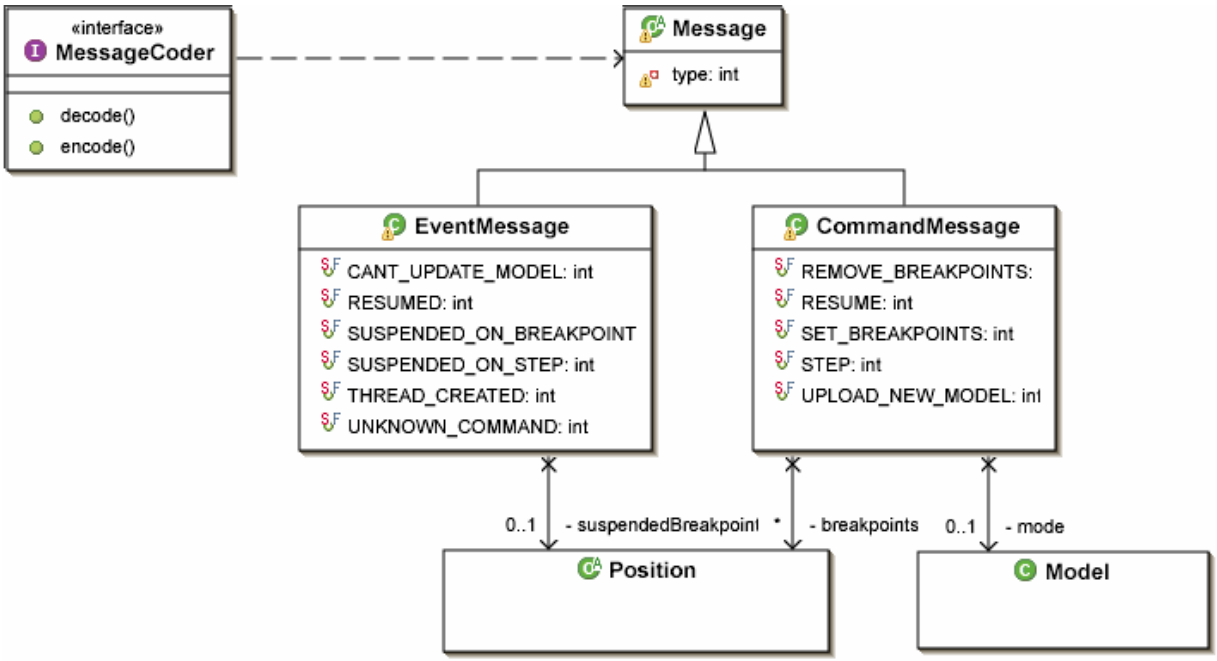

Рис. 3. Модель сообщений

3.2.2. Динамическая модель системы

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

На рис. 4 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. 4 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. 5 представлена диаграмма переходов автомата *app*. Этот автомат реализует поведение серверной части системы в целом.

На рис. 6 представлена диаграмма переходов автомата *A2*, отвечающего за управление точками останова.

На рис. 7 представлена диаграмма переходов автомата *A3*, управляющего потоками выполнения.

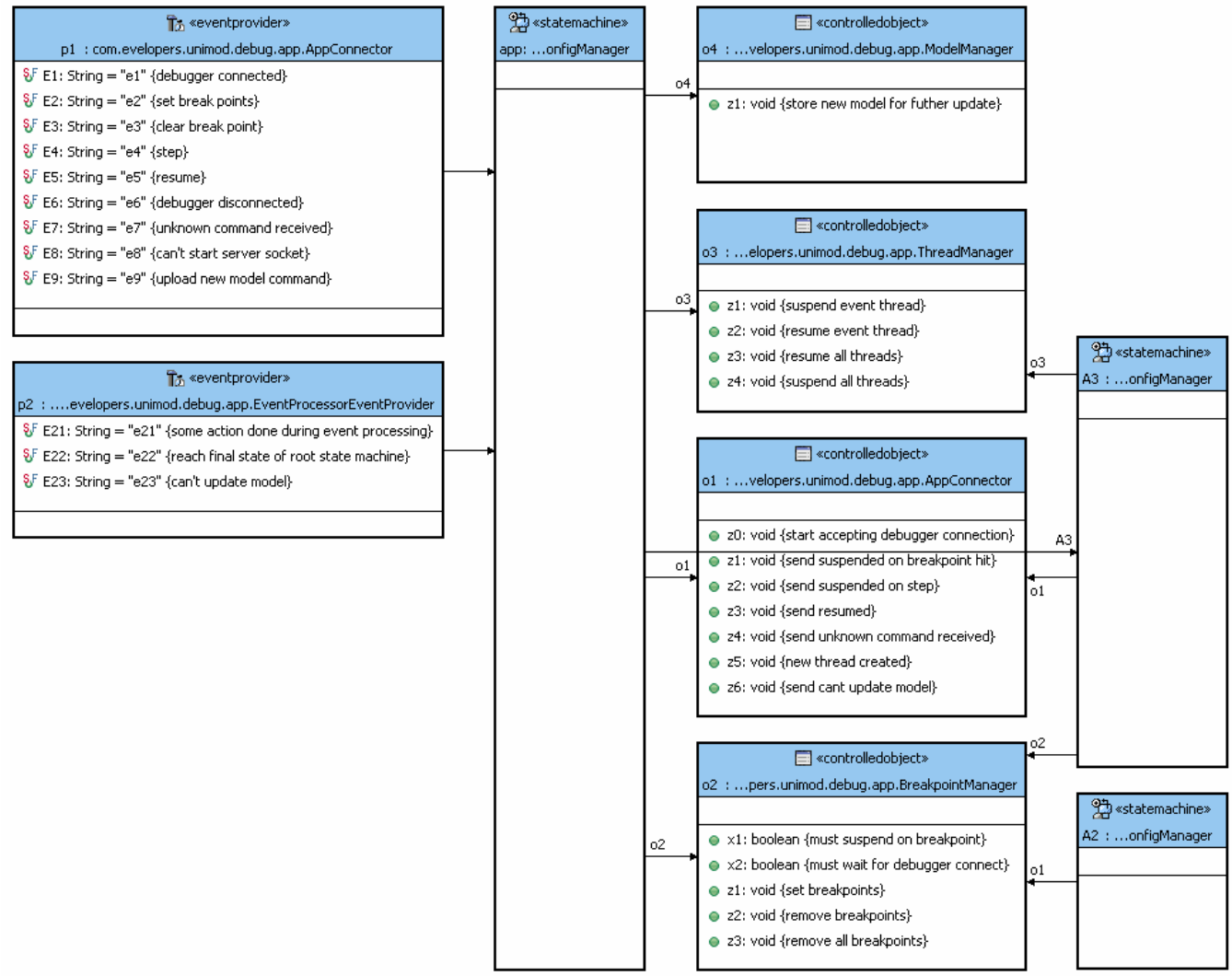

Рис. 4. Схема связей автоматов серверной части системы

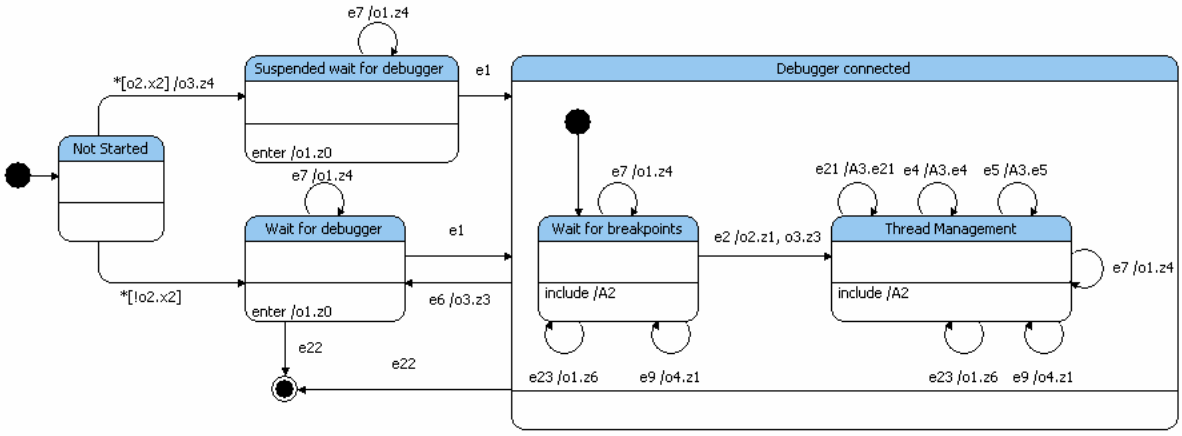

Рис. 5. Диаграмма переходов автомата *app*

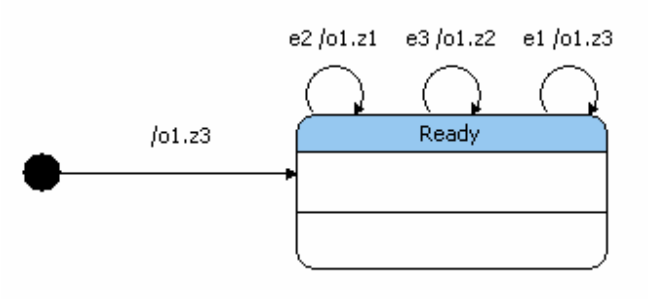

Рис. 6. Диаграмма переходов автомата *A2*

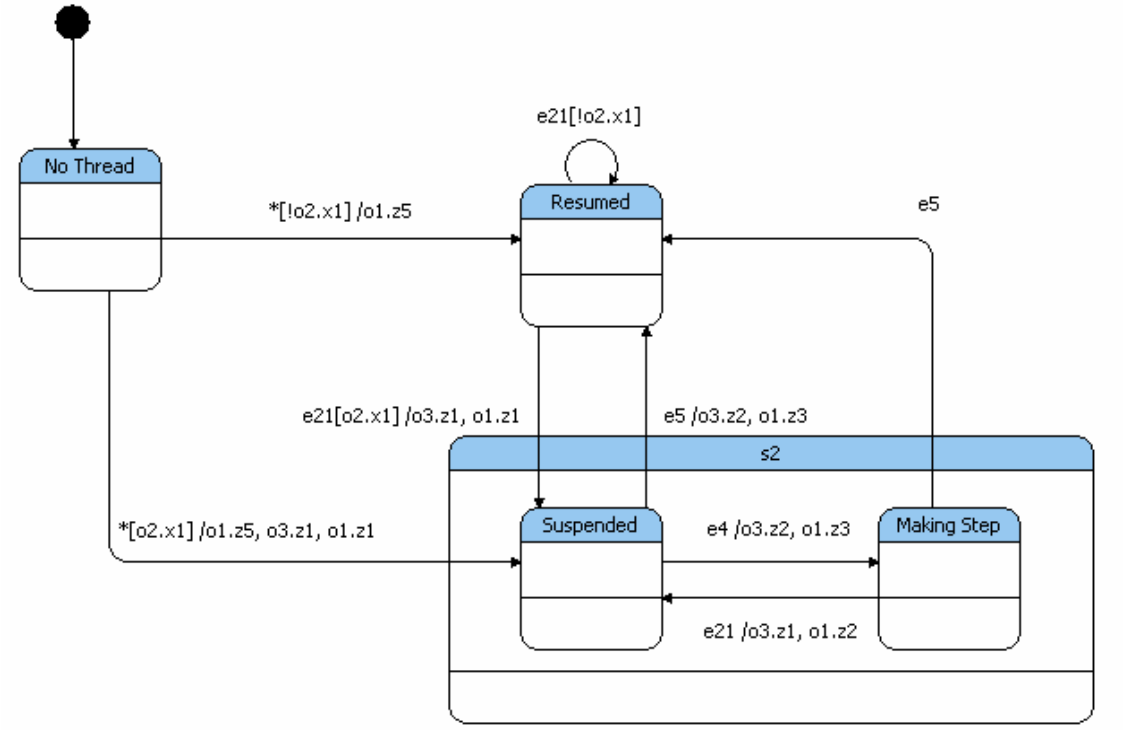

Рис. 7. Диаграмма переходов автомата *A3*

На рис. 8 представлена схема связей автоматов клиентской части системы. Классу статической модели debugger.*Debugger* соответствуют три автомата *debugger, A2 и A3*.

Класс *debugger.DebuggerConnector* на рис. 8, 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. 9 представлена диаграмма переходов автомата *debugger*. Этот автомат реализует поведение клиентской части системы в целом.

На рис. 10 представлена диаграмма переходов автомата *A2*, отвечающего за управление точками останова.

На рис. 11 представлена диаграмма переходов автомата *A3*, управляющего потоками выполнения.

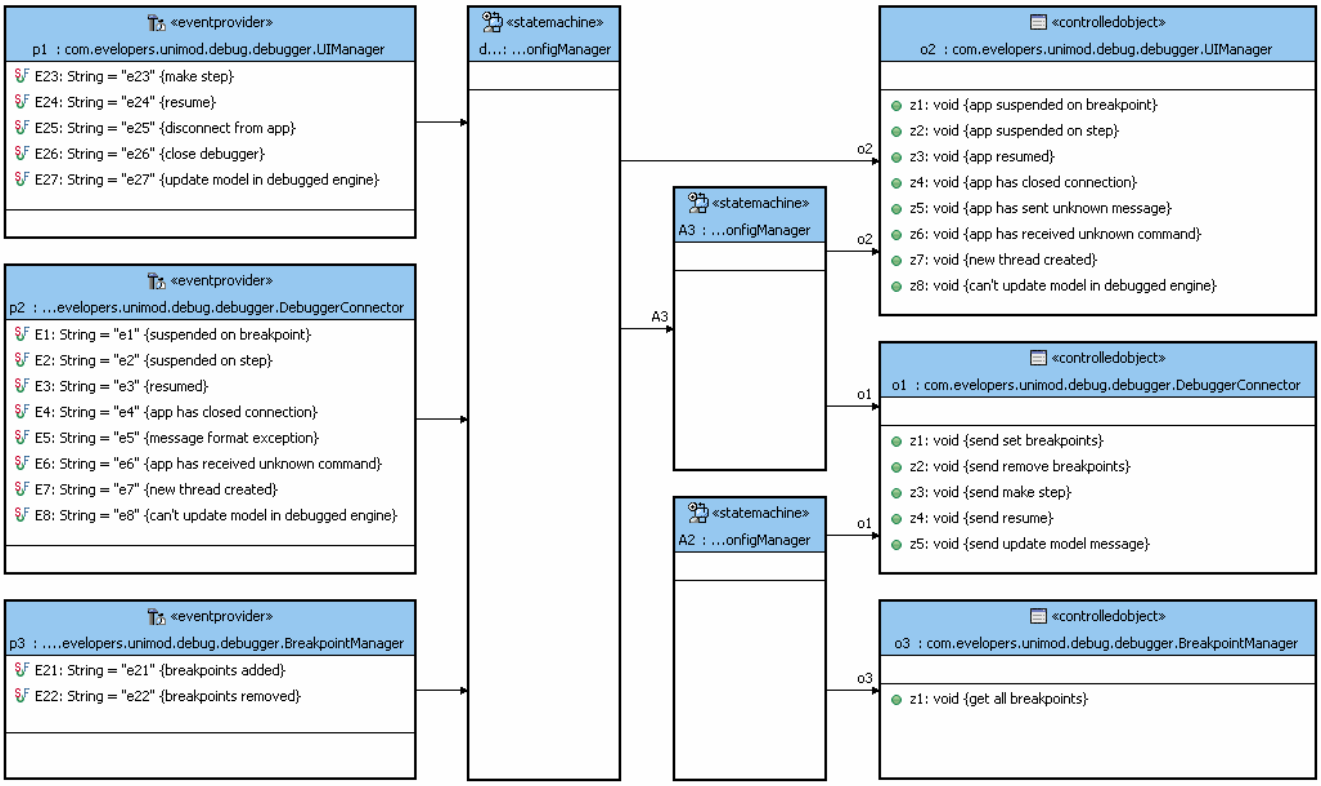

Рис. 8. Схема связей автоматов клиентской части системы

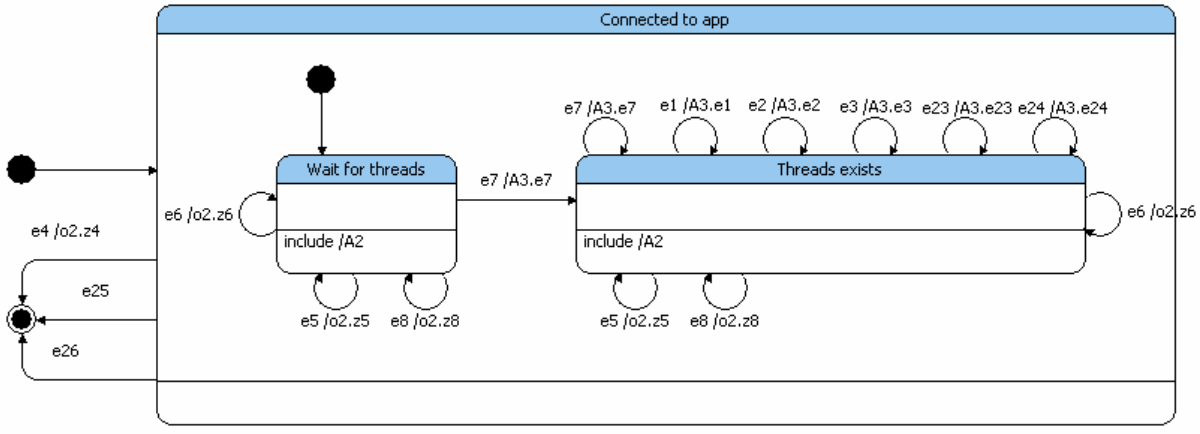

Рис. 9. Диаграмма переходов автомата *debugger*

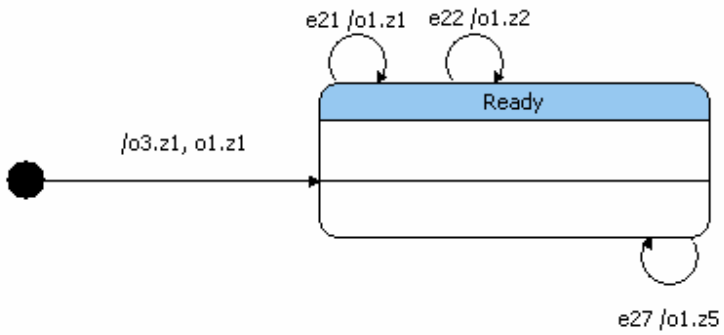

Рис. 10. Диаграмма переходов автомата *A2*

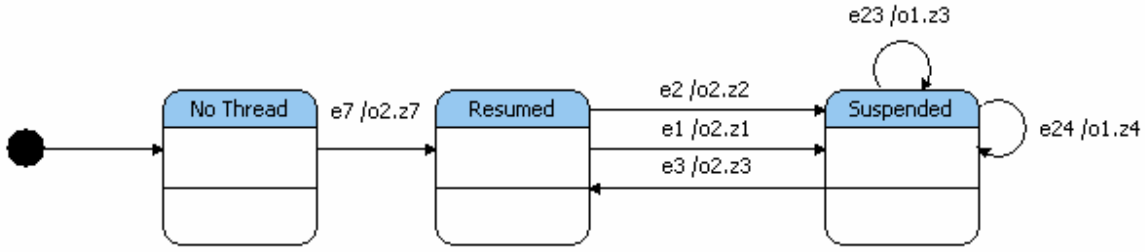

Рис. 11. Диаграмма переходов автомата *A3*

3.3. Реализация

При реализации используем интерпретационный подход. При этом по автоматной модели системы генерируется ее *XML*-описание, которое совместно с *Java*-кодом, написанным вручную для источников событий и объектов управления, исполняется интерпретатором. Полный текст *XML*-описания приведен в приложении. Ниже приведена часть *XML*-описания автоматной модели серверной части системы:

```
<model name="AppModel">

<!-- Ниже описываны объекты управления -->

  <controlledObject class="app.AppConnector" name="o1"/>
  <controlledObject class="app.BreakpointManager" name="o2"/>
  <controlledObject class="app.ThreadManager" name="o3"/>
  <controlledObject class="app.ModelManager" name="o4"/>

<!-- Источники собый -->

  <eventProvider class="app.AppConnector" name="p1">
    <association clientRole="p1" targetRef="app"/>
  </eventProvider>

  <eventProvider class="app.EventProcessorEventProvider" name="p2">
    <association clientRole="p2" targetRef="app"/>
  </eventProvider>

<!-- Корневой автомат -->
```

```
  <rootStateMachine>
    <stateMachineRef name="app"/>
  </rootStateMachine>

<!-- Описание автомата app -->

  <stateMachine name="app">

<!-- Связи автомата app с объектами управления и другими автоматами -->

    <association clientRole="app" supplierRole="o1" targetRef="o1"/>
    <association clientRole="app" supplierRole="o4" targetRef="o4"/>
    <association clientRole="app" supplierRole="o2" targetRef="o2"/>
    <association clientRole="app" supplierRole="A3" targetRef="A3"/>
    <association clientRole="app" supplierRole="o3" targetRef="o3"/>
<!-- Описание состояний автомата app -->

    <state name="Top" type="NORMAL">

      <state name="Debugger connected" type="NORMAL">
        <state name="Thread Management" type="NORMAL">

<!-- В состояние Thread Management вложен автомат A2 -->

          <stateMachineRef name="A2"/>
        </state>
        <state name="s4" type="INITIAL"/>
        <state name="Wait for breakpoints" type="NORMAL">
          <stateMachineRef name="A2"/>
        </state>
      </state>

    ....
    ....

    </state>

<!-- Описание переходов автомата app -->

      <transition event="e6" sourceRef="Debugger connected"
      targetRef="Wait for debugger">
      <outputAction ident="o3.z3"/>
    </transition>

    ....
    ....

  </stateMachine>

<!-- Описание автомата A2 -->

  <stateMachine name="A2">

    ....
    ....

  </stateMachine>
```

```
<!-- Описание автомата A3 -->

  <stateMachine name="A3">

    ....
    ....

  </stateMachine>

</model>
```

После создания автоматной модели системы источники событий и объекты управления реализуются вручную. Ниже приведен фрагмент класса *app.AppConnector*:

```
/**
 * Класс реализует интерфейсы ControlledObject и EventProvider так как он является и
источником собыий и объектом управления.
 */
public class AppConnector implements ControlledObject, EventProvider {

/**
      * Объявления событий, поставляемых данным классом автомату.
      */

      /**
      * @unimod.event.descr debugger connected
      */
      public static final String E1 = "e1";

      /**
      * @unimod.event.descr set break points
      */
      public static final String E2 = "e2";

....
....


/**
      * Этот метод вызывается интерпретатором XML-описания при запуске
      * автоматной модели.
      *
      */
    public void init(ModelEngine engine) throws SystemException {
/**
      * Сохранение указателя на объект - приемник событий.
      */
        handler = engine.getEventManager();

        try {
/**
      * Создание серверного сокета.
      */
            socket = new ServerSocket(port);
        } catch (IOException e) {
            throw new SystemException(e);
```

```
        }
    }

/**
     * Этот метод вызывается интерпретатором XML-описания при завершении
     * работы автоматной модели.
     *
     */
    public void dispose() {
        if (socketReader == null) {
            return;
        }

        try {
            InputStream is = socketReader;
            socketReader = null;
            is.close();

            //socketReader.close();
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


      /**
      * Метод, вызываемый автоматом, для начала приема соединений от клиентов
      *@unimod.action.descr start accepting debugger connection
      */
      public void z0(StateMachineContext context) throws
            MessageCoderException, IOException {

            /*
            * Создание нового потока, в котором будут приниматься сообщения от * клиента
      */
       new SocketListenerThread().start();
      }

      /**
      * Метод, вызываемый автоматом, для извещения клиента о том,
* что достигнута точка останова
      *@unimod.action.descr send suspended on breakpoint hit
      */
      public void z1(StateMachineContext context) throws
      MessageCoderException, IOException {

      Position b =
            (Position)context.getEventContext().getParameter(
                  Params.Event.POSITION);

            ThreadInfo ti =
            (ThreadInfo)context.getEventContext().getParameter(
                  Params.Event.THREAD_INFO);

            sendMessage(EventMessage.createSuspendedOnBreakpoint(b, ti));
      }
```

```
      /**
      * Служебный метод, который отправляет сообщение по каналу связи клиенту
*/
      private void sendMessage(Message m)
      throws MessageCoderException, IOException {

/*
* Преобразование сообщения в массив байт и запись массива
* в канал связи
*/
    coder.encode(m, socketWriter);
    socketWriter.flush();
}


/**
* Класс, реализующий поток, в котором принимаются сообщения от клиента.
* После получения сообщение расшифровывается и преобразуется в событие
* для автомата
*/
      private class SocketListenerThread extends Thread {

        public void run() {

            while (true) {
                Socket client = null;
                try {
                    // Примем соединения от клиента
                    client = socket.accept();

                    socketWriter = client.getOutputStream();
                    socketReader = client.getInputStream();

                    // Клиент присоединен. Извещение автомата об этом событии.
                    handler.handle(new Event(E1),
                     StateMachineContextImpl.create());

                    while (true) {
                        CommandMessage m = null;

                          // Декодирование сообщения от клиента
                            m = (CommandMessage) coder.decode(socketReader);

                        // Посылка автомату события, соответствующего типу
                        //принятого сообщения
                        switch (m.getType()) {
                        case CommandMessage.SET_BREAKPOINTS:
                            handler.handle(new Event(E2,
                                    new Parameter(Params.Event.BREAKPOINTS, m
                                            .getBreapoints())),
                                    StateMachineContextImpl.create());
                            break;

                        case CommandMessage.REMOVE_BREAKPOINTS:
                            handler.handle(new Event(E3,
                                    new Parameter(Params.Event.BREAKPOINTS, m
                                            .getBreapoints())),
                                    StateMachineContextImpl.create());
```

```
                        break;

                    ....
                    ....

                        break;
                    }
                }

            } catch (IOException e) {
                // В случае закрытия канала связи – извещение
          // автомата о том, что клиент отсоединился
                handler.handle(new Event(E6),
              StateMachineContextImpl.create());
            }
        }
    }

}
```

4. ЗАКЛЮЧЕНИЕ

Предлагаемая методика создания клиент-серверных приложений позволяет описывать динамическую модель системы в виде множества взаимодействующих конечных автоматов, задаваемых в форме графа переходов. При этом описанная модель в общем случае не нуждается в дальнейшей реализации на целевом языке программирования, так как может быть представлена в виде *XML*-описания и непосредственно выполнена интерпретатором.

Подход, предлагаемый в настоящей работе, позволяет использовать автоматы при спецификации клиент-серверного взаимодействия, его реализации и протоколировании.

Компоненты *UniMod Server-side Debugger* и *UniMod Client-side Debugger*, вошедшие в очередную версию пакета *UniMod,* реализованы с помощью предыдущей версии пакета *UniMod.* Поэтому часть проектной документация была получена «автоматически», так как диаграммы, созданные с помощью *UniMod*-редактора, могут рассматриваться как автоматные программы. Они могут быть включены в проектную документацию без изменений. Разработка последующих версий средств разработки с помощью предыдущих (раскрутка) является общепринятой практикой и позволяет говорить о зрелости программного продукта.

5. ЛИТЕРАТУРА

1. *Грехем И.* Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004. 880 с.
2. *Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А.* UML. SWITCH-Технология. Eclipse //Информационно-управляющие системы. 2005. № 6, с. 12–17. http://is.ifmo.ru/works/uml-switch-eclipse/
3. *Шалыто А. А.* SWITCH-технология. Алгоритмизация программирования задач логического управления. СПб.: Наука, 1998. 628 с. http://is.ifmo.ru/books/switch/1
4. *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/
5. *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
6. *Java Debug Wire Protocol*. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-spec.html.
7. *Java Networking Features*. http://java.sun.com/j2se/1.5.0/docs/guide/net/index.html.
8. *Гуров В.С., Мазин М.А.* Создание системы автоматического завершения ввода с использованием пакета UniMod /Вестник II межвузовской конференции молодых ученых. Т.1. СПб.: СПбГУ ИТМО. 2005, с.73–87.

6. ПРИЛОЖЕНИЕ

6.1. XML-описание серверной автоматной модели

```
<model name="AppModel">
  <controlledObject class="app.AppConnector" name="o1"/>
  <controlledObject class="app.BreakpointManager" name="o2"/>
  <controlledObject class="app.ThreadManager" name="o3"/>
  <controlledObject class="app.ModelManager" name="o4"/>
  <eventProvider class=" app.AppConnector" name="p1">
    <association clientRole="p1" targetRef="app"/>
  </eventProvider>
  <eventProvider class=" app.EventProcessorEventProvider" name="p2">
    <association clientRole="p2" targetRef="app"/>
  </eventProvider>
  <rootStateMachine>
    <stateMachineRef name="app"/>
  </rootStateMachine>
  <stateMachine name="app">
    <configStore class="config.DistinguishConfigManager"/>
    <association clientRole="app" supplierRole="o1" targetRef="o1"/>
    <association clientRole="app" supplierRole="o4" targetRef="o4"/>
    <association clientRole="app" supplierRole="o2" targetRef="o2"/>
    <association clientRole="app" supplierRole="A3" targetRef="A3"/>
    <association clientRole="app" supplierRole="o3" targetRef="o3"/>
    <state name="Top" type="NORMAL">
      <state name="Debugger connected" type="NORMAL">
        <state name="Thread Management" type="NORMAL">
          <stateMachineRef name="A2"/>
        </state>
        <state name="s4" type="INITIAL"/>
        <state name="Wait for breakpoints" type="NORMAL">
          <stateMachineRef name="A2"/>
        </state>
      </state>
      <state name="Suspended wait for debugger " type="NORMAL">
        <outputAction ident="o1.z0"/>
      </state>
      <state name="Not Started" type="NORMAL"/>
      <state name="s3" type="FINAL"/>
      <state name="Wait for debugger" type="NORMAL">
        <outputAction ident="o1.z0"/>
      </state>
      <state name="s5" type="INITIAL"/>
    </state>
    <transition event="e6" sourceRef="Debugger connected" targetRef="Wait for debugger">
      <outputAction ident="o3.z3"/>
    </transition>
    <transition event="e22" sourceRef="Debugger connected" targetRef="s3"/>
    <transition event="e7" sourceRef="Thread Management" targetRef="Thread Management">
      <outputAction ident="o1.z4"/>
    </transition>
    <transition event="e21" sourceRef="Thread Management" targetRef="Thread Management">
      <outputAction ident="A3.e21"/>
    </transition>
    <transition event="e4" sourceRef="Thread Management" targetRef="Thread Management">
      <outputAction ident="A3.e4"/>
    </transition>
    <transition event="e5" sourceRef="Thread Management" targetRef="Thread Management">
      <outputAction ident="A3.e5"/>
    </transition>
    <transition event="e9" sourceRef="Thread Management" targetRef="Thread Management">
      <outputAction ident="o4.z1"/>
    </transition>
    <transition event="e23" sourceRef="Thread Management" targetRef="Thread Management">
      <outputAction ident="o1.z6"/>
    </transition>
    <transition sourceRef="s4" targetRef="Wait for breakpoints"/>
    <transition event="e2" sourceRef="Wait for breakpoints" targetRef="Thread Management">
      <outputAction ident="o2.z1"/>
```

```
    <outputAction ident="o3.z3"/>
  </transition>
  <transition event="e7" sourceRef="Wait for breakpoints" targetRef="Wait for breakpoints">
    <outputAction ident="o1.z4"/>
  </transition>
  <transition event="e9" sourceRef="Wait for breakpoints" targetRef="Wait for breakpoints">
    <outputAction ident="o4.z1"/>
  </transition>
  <transition event="e23" sourceRef="Wait for breakpoints" targetRef="Wait for breakpoints">
    <outputAction ident="o1.z6"/>
  </transition>
  <transition event="e1" sourceRef="Suspended wait for debugger " targetRef="Debugger connected"/>
  <transition event="e7" sourceRef="Suspended wait for debugger " targetRef="Suspended wait for debugger ">
    <outputAction ident="o1.z4"/>
  </transition>
  <transition event="*" guard="o2.x2" sourceRef="Not Started" targetRef="Suspended wait for debugger ">
    <outputAction ident="o3.z4"/>
  </transition>
  <transition event="*" guard="!o2.x2" sourceRef="Not Started" targetRef="Wait for debugger"/>
  <transition event="e22" sourceRef="Wait for debugger" targetRef="s3"/>
  <transition event="e7" sourceRef="Wait for debugger" targetRef="Wait for debugger">
    <outputAction ident="o1.z4"/>
  </transition>
  <transition event="e1" sourceRef="Wait for debugger" targetRef="Debugger connected"/>
  <transition sourceRef="s5" targetRef="Not Started"/>
</stateMachine>
<stateMachine name="A2">
  <configStore class="config.SharedStateConfigManager"/>
  <association clientRole="A2" supplierRole="o1" targetRef="o2"/>
  <state name="Top" type="NORMAL">
    <state name="s1" type="INITIAL"/>
    <state name="Ready" type="NORMAL"/>
  </state>
  <transition sourceRef="s1" targetRef="Ready">
    <outputAction ident="o1.z3"/>
  </transition>
  <transition event="e2" sourceRef="Ready" targetRef="Ready">
    <outputAction ident="o1.z1"/>
  </transition>
  <transition event="e3" sourceRef="Ready" targetRef="Ready">
    <outputAction ident="o1.z2"/>
  </transition>
  <transition event="e1" sourceRef="Ready" targetRef="Ready">
    <outputAction ident="o1.z3"/>
  </transition>
</stateMachine>
<stateMachine name="A3">
  <configStore class="      .debug.ThreadConfigManager"/>
  <association clientRole="A3" supplierRole="o1" targetRef="o1"/>
  <association clientRole="A3" supplierRole="o2" targetRef="o2"/>
  <association clientRole="A3" supplierRole="o3" targetRef="o3"/>
  <state name="Top" type="NORMAL">
    <state name="s2" type="NORMAL">
      <state name="Suspended" type="NORMAL"/>
      <state name="Making Step" type="NORMAL"/>
    </state>
    <state name="Resumed" type="NORMAL"/>
    <state name="s3" type="INITIAL"/>
    <state name="No Thread" type="NORMAL"/>
  </state>
  <transition event="e4" sourceRef="Suspended" targetRef="Making Step">
    <outputAction ident="o3.z2"/>
    <outputAction ident="o1.z3"/>
  </transition>
  <transition event="e5" sourceRef="Suspended" targetRef="Resumed">
    <outputAction ident="o3.z2"/>
    <outputAction ident="o1.z3"/>
  </transition>
  <transition event="e21" sourceRef="Making Step" targetRef="Suspended">
    <outputAction ident="o3.z1"/>
    <outputAction ident="o1.z2"/>
  </transition>
  <transition event="e5" sourceRef="Making Step" targetRef="Resumed"/>
```

```
    <transition event="e21" guard="o2.x1" sourceRef="Resumed" targetRef="Suspended">
      <outputAction ident="o3.z1"/>
      <outputAction ident="o1.z1"/>
    </transition>
    <transition event="e21" guard="!o2.x1" sourceRef="Resumed" targetRef="Resumed"/>
    <transition sourceRef="s3" targetRef="No Thread"/>
    <transition event="*" guard="!o2.x1" sourceRef="No Thread" targetRef="Resumed">
      <outputAction ident="o1.z5"/>
    </transition>
    <transition event="*" guard="o2.x1" sourceRef="No Thread" targetRef="Suspended">
      <outputAction ident="o1.z5"/>
      <outputAction ident="o3.z1"/>
      <outputAction ident="o1.z1"/>
    </transition>
  </stateMachine>
</model>
```

6.2. XML-описание клиентской автоматной модели

```
<model name="DebuggerModel">
  <controlledObject class="debugger.DebuggerConnector" name="o1"/>
  <controlledObject class="debugger.UIManager" name="o2"/>
  <controlledObject class="debugger.BreakpointManager" name="o3"/>
  <eventProvider class="debugger.UIManager" name="p1">
    <association clientRole="p1" targetRef="debugger"/>
  </eventProvider>
  <eventProvider class="debugger.DebuggerConnector" name="p2">
    <association clientRole="p2" targetRef="debugger"/>
  </eventProvider>
  <eventProvider class="debugger.BreakpointManager" name="p3">
    <association clientRole="p3" targetRef="debugger"/>
  </eventProvider>
  <rootStateMachine>
    <stateMachineRef name="debugger"/>
  </rootStateMachine>
  <stateMachine name="debugger">
    <configStore class="config.DistinguishConfigManager"/>
    <association clientRole="debugger" supplierRole="A3" targetRef="A3"/>
    <association clientRole="debugger" supplierRole="o2" targetRef="o2"/>
    <state name="Top" type="NORMAL">
      <state name="s3" type="INITIAL"/>
      <state name="s2" type="FINAL"/>
      <state name="Connected to app" type="NORMAL">
        <state name="Wait for threads" type="NORMAL">
          <stateMachineRef name="A2"/>
        </state>
        <state name="s1" type="INITIAL"/>
        <state name="Threads exists" type="NORMAL">
          <stateMachineRef name="A2"/>
        </state>
      </state>
    </state>
    <transition sourceRef="s3" targetRef="Connected to app"/>
    <transition event="e4" sourceRef="Connected to app" targetRef="s2">
      <outputAction ident="o2.z4"/>
    </transition>
    <transition event="e25" sourceRef="Connected to app" targetRef="s2"/>
    <transition event="e26" sourceRef="Connected to app" targetRef="s2"/>
    <transition event="e5" sourceRef="Wait for threads" targetRef="Wait for threads">
      <outputAction ident="o2.z5"/>
    </transition>
    <transition event="e6" sourceRef="Wait for threads" targetRef="Wait for threads">
      <outputAction ident="o2.z6"/>
    </transition>
    <transition event="e7" sourceRef="Wait for threads" targetRef="Threads exists">
      <outputAction ident="A3.e7"/>
    </transition>
    <transition event="e8" sourceRef="Wait for threads" targetRef="Wait for threads">
      <outputAction ident="o2.z8"/>
    </transition>
    <transition sourceRef="s1" targetRef="Wait for threads"/>
    <transition event="e5" sourceRef="Threads exists" targetRef="Threads exists">
      <outputAction ident="o2.z5"/>
```

```
  </transition>
  <transition event="e6" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="o2.z6"/>
  </transition>
  <transition event="e7" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="A3.e7"/>
  </transition>
  <transition event="e1" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="A3.e1"/>
  </transition>
  <transition event="e2" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="A3.e2"/>
  </transition>
  <transition event="e3" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="A3.e3"/>
  </transition>
  <transition event="e23" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="A3.e23"/>
  </transition>
  <transition event="e24" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="A3.e24"/>
  </transition>
  <transition event="e8" sourceRef="Threads exists" targetRef="Threads exists">
    <outputAction ident="o2.z8"/>
  </transition>
</stateMachine>
<stateMachine name="A2">
  <configStore class="config.SharedStateConfigManager"/>
  <association clientRole="A2" supplierRole="o3" targetRef="o3"/>
  <association clientRole="A2" supplierRole="o1" targetRef="o1"/>
  <state name="Top" type="NORMAL">
    <state name="s1" type="INITIAL"/>
    <state name="Ready" type="NORMAL"/>
  </state>
  <transition sourceRef="s1" targetRef="Ready">
    <outputAction ident="o3.z1"/>
    <outputAction ident="o1.z1"/>
  </transition>
  <transition event="e21" sourceRef="Ready" targetRef="Ready">
    <outputAction ident="o1.z1"/>
  </transition>
  <transition event="e22" sourceRef="Ready" targetRef="Ready">
    <outputAction ident="o1.z2"/>
  </transition>
  <transition event="e27" sourceRef="Ready" targetRef="Ready">
    <outputAction ident="o1.z5"/>
  </transition>
</stateMachine>
<stateMachine name="A3">
  <configStore class="debug.ThreadConfigManager"/>
  <association clientRole="A3" supplierRole="o1" targetRef="o1"/>
  <association clientRole="A3" supplierRole="o2" targetRef="o2"/>
  <state name="Top" type="NORMAL">
    <state name="s1" type="INITIAL"/>
    <state name="No Thread" type="NORMAL"/>
    <state name="Suspended" type="NORMAL"/>
    <state name="Resumed" type="NORMAL"/>
  </state>
  <transition sourceRef="s1" targetRef="No Thread"/>
  <transition event="e7" sourceRef="No Thread" targetRef="Resumed">
    <outputAction ident="o2.z7"/>
  </transition>
  <transition event="e3" sourceRef="Suspended" targetRef="Resumed">
    <outputAction ident="o2.z3"/>
  </transition>
  <transition event="e23" sourceRef="Suspended" targetRef="Suspended">
    <outputAction ident="o1.z3"/>
  </transition>
  <transition event="e24" sourceRef="Suspended" targetRef="Suspended">
    <outputAction ident="o1.z4"/>
  </transition>
  <transition event="e2" sourceRef="Resumed" targetRef="Suspended">
    <outputAction ident="o2.z2"/>
```

```
    </transition>
    <transition event="e1" sourceRef="Resumed" targetRef="Suspended">
      <outputAction ident="o2.z1"/>
    </transition>
  </stateMachine>
</model>
```

Лист регистрации изменений

Номера листов (страниц)					*Всего листов (страниц) в докум.*	*№ документа*	*Входящий № сопроводительного докум. и дата*	*Подп.*	*Дата*
Изм.	*Измененных*	*Замененных*	*Новых*	*Аннулированных*					