Министерство образования и науки Российской Федерации

Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Б. З. Хасянзянов

Метод создания отладчиков для доменно-ориентированных языков программирования на основе технологии *Eclipse Modeling*

Бакалаврская работа

Руководитель: Гуров В. С.

Санкт-Петербург

2007

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Моделирование предметной области является одним из принципов при разработке программного обеспечения. На основе *доменных моделей* (моделей предметной области, *Domain Models* [1]) строятся как само программное обеспечение, так и программные средства, применяемые при его разработке.

Отдельным направлением в рамках «модельного» подхода являются *доменные языки* (*Domain-Specific Languages*, *DSL* [1]) и *порождающее программирование* (*Generative Programming* [1]).

Доменный язык (*Domain-Specific Languages*, *DSL* [1]) – это язык программирования, предназначенный для решения задач определенной предметной области.

Доменная модель (*Domain Models* [1]) – это подробное представление общих и изменяемых свойств системы, существующих в рамках данной предметной области, семантики этих свойств и понятий этой области, а также зависимостей между изменяемыми свойствами.

В рамках данного подхода доменно-ориентированный язык программирования задается *UML*-диаграммой структуры элементов, представляющих доменный язык – его *метамоделью*. Далее на основе модели на доменно-ориентированном языке автоматически генерируется программный код.

При разработке программного продукта в рамках порождающего программирования могут возникнуть трудности с отладкой конечного кода приложения стандартными средствами отладки. Эти проблемы связаны, в основном, с особенностями генерации кода:

- проблемы с читаемостью автоматически сгенерированного кода – полученный код может быть сложным для понимания;
- большой объем сгенерированного кода.

Поэтому возникает задача – создать систему отладки для доменно-ориентированных языков программирования, которая позволяла бы производить отладку на уровне элементов доменного языка и абстрагироваться от средств языка реализации конечного приложения.

Цель настоящей работы – описать метод построения отладчиков для доменно-ориентированных языков программирования. Метод будет рассмотрен на примере создания отладчика для автоматных программ на базе платформы *Eclipse* [2] с использованием технологий *Eclipse Modeling Framework* (*EMF* [3]), *Graphical Modeling Framework* (*GMF* [4]) и *Eclipse Debug Core* [2].

Одним из средств проектирования доменных моделей является технология *EMF* (*Eclipse Modeling Framework* [3]), реализованная как расширение к интегрированной среде разработки приложений *Eclipse* [2]. С помощью *EMF* на основе языка *UML* [5] строится метамодель доменно-ориентированного языка, задающая элементы, которые будут использованы при моделировании.

Например, для диаграммы состояний конечного автомата метамоделью является диаграмма, описывающая отношения между такими базовыми элементами конечного автомата, как «состояние», «переход в состояние», «событие» и т.д. Моделью в данном случае будет диаграмма состояний конкретного конечного автомата.

Для построения редактора моделей, используемого отладчиком для отображения и управления процессом отладки, была использована технология *Graphical Modeling Framework* (*GMF*) [4]. Эта технология позволяет по описанию метамодели и дополнительных конфигураций построить полнофункциональный графический редактор, встраивающийся как расширение в среду разработки *Eclipse*. Конфигурации задают список элементов метамодели, которые будут использованы на диаграмме, а также их графическое представление и наборы связанных с ними элементов управления редактора.

Разработанная система позволяет в полной мере использовать все стандартные средства отладки, предусмотренные средой разработки приложений *Eclipse*:

- точки останова (*breakpoints*), позволяющие приостановить выполнение приложения в указанном месте (в конкретном состоянии конечного автомата, или на конкретном переходе, в случае отладки модели конечного автомата);
- пошаговое выполнение;
- просмотр значений переменных в конкретный момент исполнения приложения (*context variables*).

В главе 1 выполнен обзор существующих технологий моделирования и отладки для доменно-ориентированных языков программирования.

В главе 2 сформулировано требование к системе отладки и изложены основные идеи предложенного метода.

В Главе 3 рассмотрена реализация метода создания системы отладки для произвольного доменно-ориентированного языка.

ГЛАВА 1. АНАЛИЗ СУЩЕСТВУЮЩИХ ТЕХНОЛОГИЙ

1.1. Обзор существующих технологий отладки

При анализе существующих технологий систем отладки было рассмотрено средство отладки автоматных программ, приведенное в работе [6]. Предлагаемая система позволяет наблюдать за работой автоматов, вести протоколирование и расставлять "точки остановки" (*breakpoints*), позволяющие приостанавливать работу приложения в интересующем разработчика месте. У этой системы есть ряд недостатков. Она жестко привязана к реализации конечного автомата и к языку программирования. Кроме того, сама система отладки реализована в виде отдельного от среды разработки приложения, что затрудняет ее использование.

Рассмотрим следующие системы моделирования и отладки программных систем – *Telelogic Statemate* [7], *Telelogic Rhapsody* [8] и *Telelogic Tau* [9]. Эти средства были созданы специально для построения сложных реактивных систем, поведение которых заданно диаграммой состояний. Семантика, используемая в них, основана на конкретном доменном языке – на диаграммах состояний конечного автомата. Кроме того, средства отладки, предусмотренные в данных продуктах, недостаточно практичны. Системы предназначены для раннего отслеживания ошибок в моделях, когда они еще не проинтегрированы в приложение. Поэтому отладка модели происходит отдельно от остального кода приложения.

Также отметим, что программы *Statemate, Rhapsody* и *Tau* являются коммерческими продуктами. Поэтому не только исходные коды, но даже и исполняемые файлы не представлены в свободном доступе.

Проанализировав существующие технологии моделирования и отладки приложений, основанных на доменных языках, необходимо отметить, что на

сегодняшний день наблюдается дефицит таких программных средств, а существующие технологии обладают рядом недостатков:

- они, так или иначе, ограничены метамоделью языка и представляют собой решение для конкретного класса моделей;
- ограничены средствами языка программирования, для которых реализованы;
- отладка моделей изолирована от кода приложения, использующего их;
- автономны от сред разработки приложений;
- являются закрытыми коммерческими продуктами.

1.2. Обзор средств моделирования

С появлением универсального языка моделирования *UML* [2] началась активная разработка программных средств моделирования. В рамках проекта *Eclipse* разрабатывается библиотека *EMF* (*Eclipse Modeling Framework* [3]) – универсальное средство, позволяющее строить метамодели и модели на их основе, которое поддерживает *XMI* и генерацию кода. На сегодняшний день возможности этой библиотеки таковы, что она позволяет по любой метамодели построить примитивный пользовательский интерфейс для создания объектов модели и редактирования свойств.

На основе технологии *EMF* создано множество средств, используемых при моделировании и генерации кода в различных областях применения. Одним из таких средств является фреймворк *GMF* (*Graphical Modeling Framework*) [4]. Это расширение предоставляет возможность автоматического генерирования кода для полнофункционального графического редактора моделей по произвольной метамодели, созданной с помощью технологии *EMF*. Таким образом, эта технология может быть использована как универсальное средство, применимое для любого доменно-ориентированного языка программирования. Функциональность средства позволяет использовать его в качестве

вспомогательного инструмента для создания отладчика приложений, написанных на доменных языках, так как открытый код редактора, полученного с его помощью, может быть использован для отображения процесса отладки непосредственно на графической модели программы, написанной на доменном языке.

1.3. Выводы

1. Выполнен обзор существующих технологий отладки для доменно-ориентированных языков программирования.
2. Установлено, что существующие технологии обладают рядом недостатков и ограничений, затрудняющих их использование при разработке приложений.
3. Выполнен обзор вспомогательных средств моделирования, основанных на использовании доменных моделей.

ГЛАВА 2. ОТЛАДЧИК ДЛЯ ДОМЕННО-ОРИЕНТИРОВАННОГО ЯЗЫКА

Как было показано в предыдущей главе, в настоящее время существует множество ограничений, которые затрудняют использование тех или иных систем отладки программ, написанных на доменных языках. В данной главе рассматривается метод создания отладчиков для доменных языков программирования, позволяющий избавиться от этих ограничений.

В разд. 2.1. формулируется задача, а в разд. 2.2. рассматриваются пути решения поставленной задачи и приводятся базовые идеи предлагаемого метода.

2.1. Постановка задачи

Необходимо разработать отладчик для программ, написанных на произвольном доменно-ориентированном языке, позволяющий отображать отлаживаемый алгоритм в графическом представлении в виде элементов доменно-ориентированного языка.

Такой язык задается метамоделью, описываемой диаграммой классов *UML*.

Возможность отладки программной модели программы на доменно-ориентированном языке не должна ориентироваться только на один конкретный язык программирования приложения[1]. Отладка должна быть возможна для любых языков программирования конечного приложения, таких как *Java, C#, C++*, а также других языков, поддерживаемых средой разработки, в которую интегрирован отладчик.

[1] Язык программирования приложения – это тот язык программирования общего назначения, в который будет транслироваться программа, написанная на доменно-ориентированном языке программирования (Java, C++, C# и т.д.).

2.2. Основные идеи метода

Одно из ограничений существующих систем отладки для доменно-ориентированных языков – это ориентация этих систем только на конкретные языки, что сужает область их применения. Отсюда возникает первое требование к методу, сформулированное в разд. 2.1 – разработка универсального ядра отладчика, способного работать с произвольной доменной метамоделью. Этого можно добиться, лишь избавившись от зависимости отладчика от метамодели доменно-ориентированного языка. При этом код отладчика не должен непосредственно использовать классы метамодели. Этот результат достигается следующим образом.

Применение доменных языков предполагает генерацию кода конечного приложения на языке программирования общего назначения (*C, C++, C#, Java и т.д.*). Таким образом, под конкретный доменно-ориентированный язык программирования должен существовать компонент, генерирующий реализацию программы по модели, написанной на этом языке, в программный код на языке программирования общего назначения. Рассмотрение методов генерации конечного кода не является целью данной работы, однако предполагается наличие генератора для рассматриваемой доменной модели. В дальнейшем к генератору будет предъявлено требование, позволяющее реализовать независимость ядра отладчика от доменной метамодели.

Рассмотрим основные моменты процесса отладки приложения. Для выявления логических ошибок алгоритма, реализованного на процедурном языке программирования, необходимо локализовать место ошибки в программе и детально рассмотреть поведение системы в этом месте. Это достигается при помощи таких средств отладки, как «точки останова» (*breakpoints*) и «пошаговое выполнение», позволяющее детально проследить ход выполнения алгоритма. Для реализации этих средств отладки, необходимо строить соответствие между участком исполняемого кода и графическим представлением элемента алгоритма,

соответствующего этому коду. Следовательно, отладчик должен взаимодействовать с графическим представлением отлаживаемой модели и сопоставлять элементу этого представления, выбранному в качестве «точки останова», участок кода программы, в котором следует приостановить выполнение.

Отладчик приложения представлен двумя компонентами:

- компонентом отладки, интегрированным в среду разработки приложений;
- клиентской части отладчика, выполняющейся в контексте отлаживаемого приложения (например, подключенной в виде библиотеки).

Интегрированный в среду разработки компонент предназначен для обеспечения интерактивности процесса отладки:

- отображение текущей позиции выполнения алгоритма;
- установка «точек останова»;
- управление исполнением программы (пошаговое выполнение, продолжение выполнения программы после приостановки и т.д.).

Клиентская часть отладчика предоставляет следующую функциональность:

- приостановка выполнения программы, когда достигнута очередная «точка останова», либо выполнен шаг алгоритма в режиме пошагового выполнения программы;
- предоставление информации о внутреннем состоянии алгоритма (текущая позиция выполнения алгоритма и контекстная информация).

Рассмотрим способ, с помощью которого будут заданы точки останова – метод, задающий соответствие между элементом доменно-ориентированного языка и участком кода отлаживаемого приложения.

Отметим, что модель доменного языка, также как и метамодель, задается в виде *XMI*-описания. Каждый элемент *XMI*-ресурса имеет свой уникальный

идентификатор – *URI* (*Unified Resource Identifier*)[10]. Таким образом, соответствие между доменным языком и программой может быть построено на основе *URI-идентификаторов* элементов доменного языка.

Итак, сформулируем требование к генератору кода: при его генерации для элемента модели, который может быть представлен в виде атомарного шага алгоритма (например, *состояние автомата* в случае автоматной модели), необходимо вставить во фрагмент сгенерированного кода обращение к клиентской части отладчика, передав в него *URI*-идентификатор элемента.

Это требование позволяет избавиться от зависимости программного кода отладчика от программного кода, реализующего метамодель доменно-ориентированного языка программирования, так как связь между программной реализацией метамодели и реализацией отладчика на этапе компиляции не устанавливается.

Итак, «точки останова» могут быть переданы клиентской части отладчика приложения в виде строки, хранящей идентификатор, и текущая позиция в модели доменного языка также может быть представлена идентификатором элемента доменного языка, сохраненном в сгенерированном коде. Благодаря требованию, сформулированному к генератору, в код отлаживаемого приложения добавлены вызовы метода отладчика, в качестве параметра которого передается идентификатора *URI* базового элемента. Этот метод проверяет, соответствует ли текущая позиция в программе какой-либо «точке останова», и, при необходимости, приостанавливает выполнение программы.

Для того чтобы оптимизировать скорость сопоставления идентификатора *URI* текущей позиции в отлаживаемом алгоритме значениям списка «точек останова», рекомендуется хранить этот список в виде структуры данных, реализованной на основе хеш-функции от значения *URI* (например, в структуре *HashSet* в случае реализации на языке *Java*). Оценка времени, затрачиваемого на поиск элемента в такой структуре данных, составит *O(1)*.

Следующей реализованной функциональностью является отображение контекстной информации отлаживаемой модели (контекстные переменные, параметры вызова и т.п.). Контекстную информацию отладки можно хранить в универсальной структуре данных, передавая ее в часть отладчика, интегрированную в среду разработки приложения. Такой структурой данных может быть, например, структура, аналогичная *HashMap* в языке *Java*. Эта структура хранит множество пар «ключ-значение». В качестве «ключа» будет выступать название контекстной переменной, а в качестве «значения» – значение переменной. Состав контекстной информации зависит от конкретной доменной метамодели. Поэтому ее содержимое и заполнение актуальными данными должно быть реализовано генератором кода конечного выполняемого приложения.

2.3. Выводы

1. Сформулирована постановка задачи.
2. Детально рассмотрен процесс отладки приложений.
3. Предложены основные идеи метода создания отладчика, в котором устранены ограничения, описанные в главе 1, и решения задача, сформулированной в разд. 2.1.

ГЛАВА 3. РЕАЛИЗАЦИЯ МЕТОДА ПОСТРОЕНИЯ ОТЛАДЧИКА ДЛЯ ДОМЕННО-ОРИЕНТИРОВАННОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ

В этой главе будет рассмотрена реализация метода, предложенного в главе 2. Метод рассмотрен на примере создания отладчика программ, построенных на основе автоматного подхода. Доменный язык в этом случае представлен метамоделью диаграммы состояний конечного автомата (*statechart*), представленной в разд. 3.1.

Реализация рассмотренного в главе метода представляет собой расширение к среде разработки приложений *Eclipse*, построенное на основе технологий *EMF* [3], *GMF* [4] и *Eclipse Debug Core* [2].

В разд. 3.2. рассмотрен метод построения вспомогательного средства отладчика – графического редактора, используемого отладчиком для графического отображения процесса отладки. В разд. 3.3. предложена детальная реализация отладчика. В конце главы выполнен обзор и проиллюстрирована полученная функциональность отладчика.

3.1. Метамодель диаграммы конечного автомата

Для наглядного описания реализации рассматриваемого метода, приведем пример построения графического отладчика для моделей конечных автоматов. Доменный язык для этого примера задан на основе *UML*-модели структуры конечного автомата (*рис. 1*). Данная модель была построена при помощи технологии *EMF*.

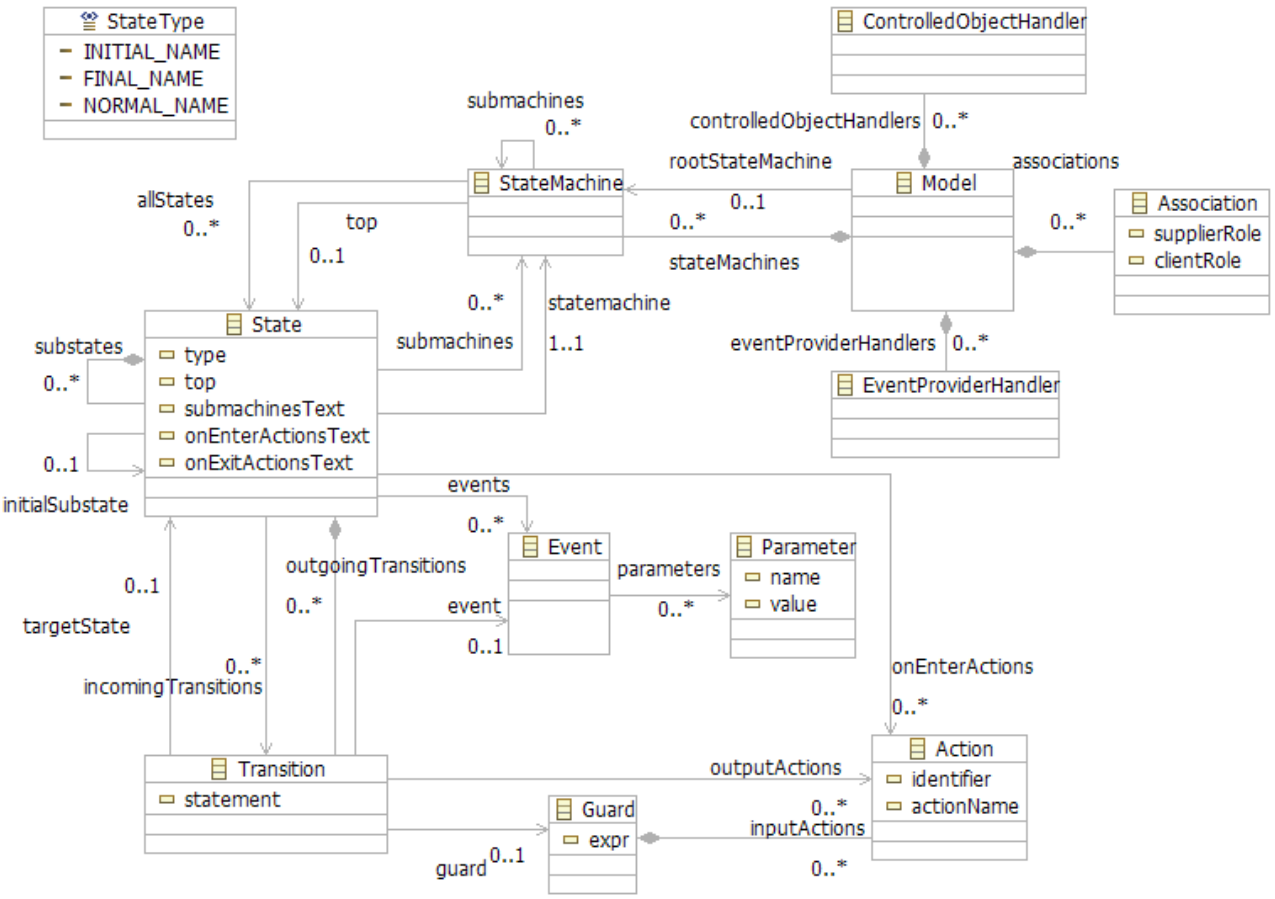

Рис. 1. Метамодель редактора диаграммы состояний конечного автомата

Приведем словесное описание данной модели.

Model – класс, представляющий собой описание структуры связей между экземплярами конечных автоматов (*StateMachine*), объектами-поставщиками событий (*EventProviderHandler*) и элементами управления приложения (*ControlledObjectHandler*), разрабатываемого в рамках автоматного подхода.

StateMachine – базовый класс модели, который хранит описание диаграммы переходов (*statechart*). Экземпляр этого класса хранит набор состояний и описание переходов между ними.

State – класс модели, описывающий состояние конечного автомата. Хранит тип состояния («начальное» – *initial*; «конечное» – *final*; «нормальное» – *normal*), список вложенных состояний, списки входящих и исходящих переходов (*transitions*).

Transition – класс модели, описывающий переход из одного состояния в другое. Хранит описание события на переходе, условие на переходе (*guard*) и список выходных воздействий (*output actions*).

Guard – класс модели, описывающий условие на переходе из одного состояния в другое. Хранит список входных воздействий (*input actions*), и булево выражение, представляющее собой условие, разрешающее переход по соответствующему ему событию.

3.2. Генерация кода графического редактора для диаграмм переходов конечного автомата

Для наглядного отображения процесса отладки конечного автомата, необходим компонент, предназначенный для графического отображения модели, который адаптирован под конкретную модель доменного языка. В качестве такого компонента будем использовать редактор, сгенерированный по метамодели доменного языка. Полученный редактор может быть использован не только для создания и редактирования моделей языка. Код сгенерированных компонентов и другие средства полученного редактора будут использованы для графического отображения процесса отладки.

Для этой цели может быть использовано расширение *GMF*, позволяющее по модели доменного языка сгенерировать код графического редактора, позволяющего создавать экземпляры моделей данного доменного языка. Расширение *Eclipse Graphical Modeling Framework* (*GMF*) является частью проекта *Eclipse Modeling Project* [11] и представляет собой фреймворк для генерации графических редакторов моделей, основанных на *EMF*.

Необходимо отметить, что полученный при помощи *GMF* редактор, а также построенный на его основе отладчик, будет реализован как новое расширение к среде разработки приложений *Eclipse.* Это позволит использовать его

непосредственно в процессе разработки конечных приложений. На *рис. 2* изображен схема общего процесса создания редактора при помощи технологии *GMF*.

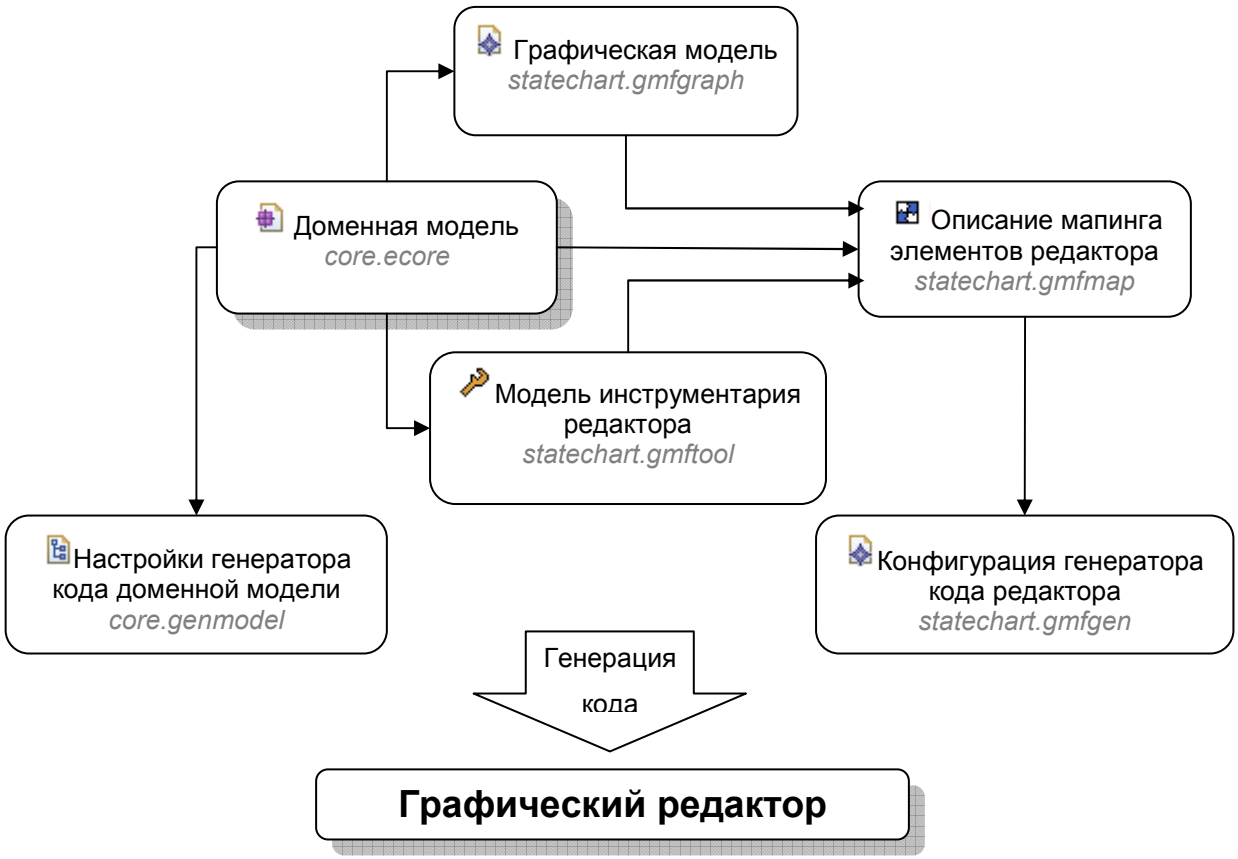

Рис. 2. Процесс генерации кода редактора по доменной модели при помощи *GMF*

Рассмотрим основные этапы процесса создания графического редактора на примере редактора диаграмм состояний конечного автомата.

Первым этапом создания редактора – генерация программного кода доменной модели – кода, реализующего классы *UML* метамодели конечного автомата. Для этого средствами *EMF* на основе описания метамодели создается модель, содержащая настройки генератора, и по ней генерируется программный код на языке *Java*. Этот код будет использован редактором моделей для хранения данных редактируемой модели и программным кодом конечного приложения. На *рис. 3* показано окно настроек генератора для автоматной модели.

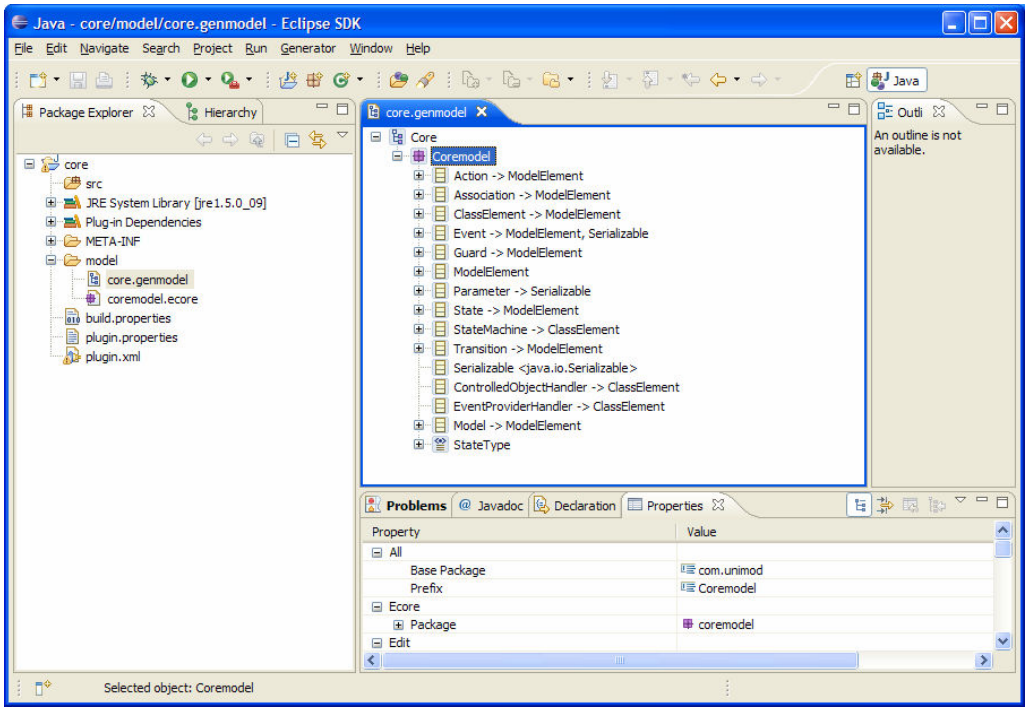

Рис. 3. Настройки генератора программного кода метамодели

Фрагмент сгенерированного кода класса, описывающего состояние конечного автомата, показан на *рис. 4*.

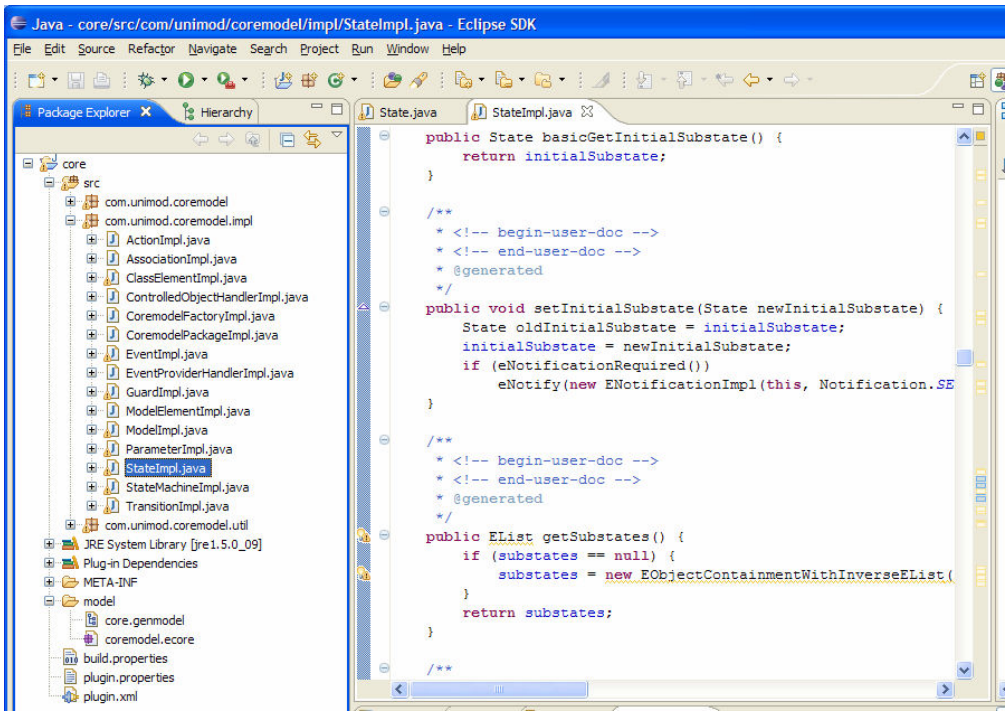

Рис. 4. Сгенерированный программный код метамодели

На втором этапе создается конфигурация создаваемого графического редактора, описывающая как элементы модели будут представлены графически, а также какие инструменты редактора будут использованы для создания и редактирования модели. На *рис. 5* изображены конфигурации графической модели, инструментария редактора и соответствия между графическими частями и инструментами редактора.

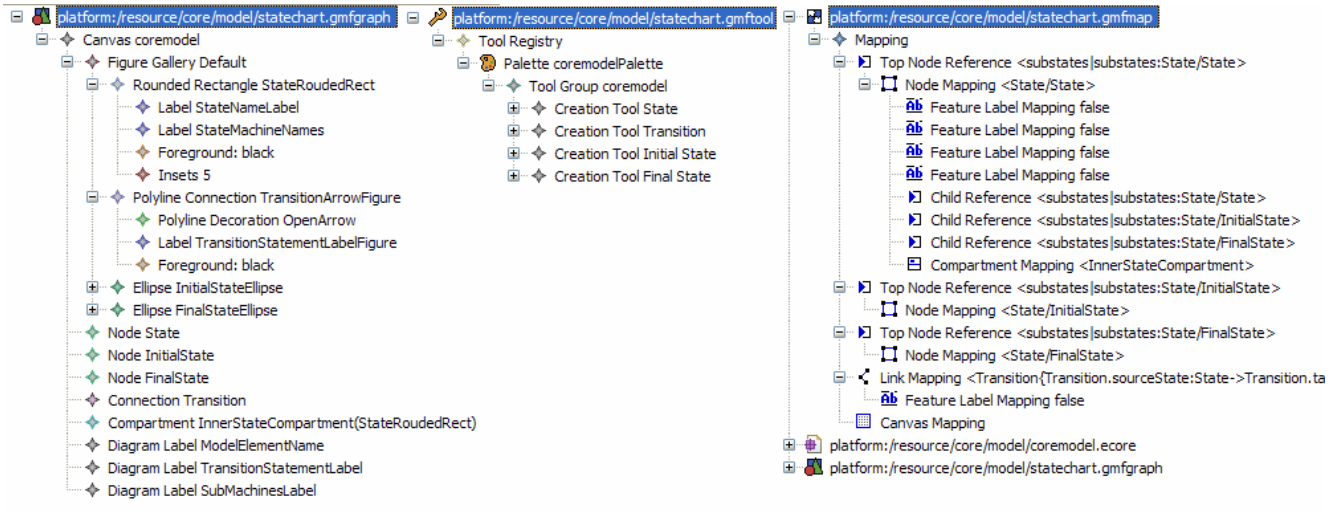

Рис. 5. Конфигурация редактора

На завершающем этапе по сформированной конфигурации с использованием технологии *GMF* построен графический редактор, который может быть применен в процессе отладки приложения, написанного на рассматриваемом доменном языке. Внешний вид редактора изображен на *рис. 6*.

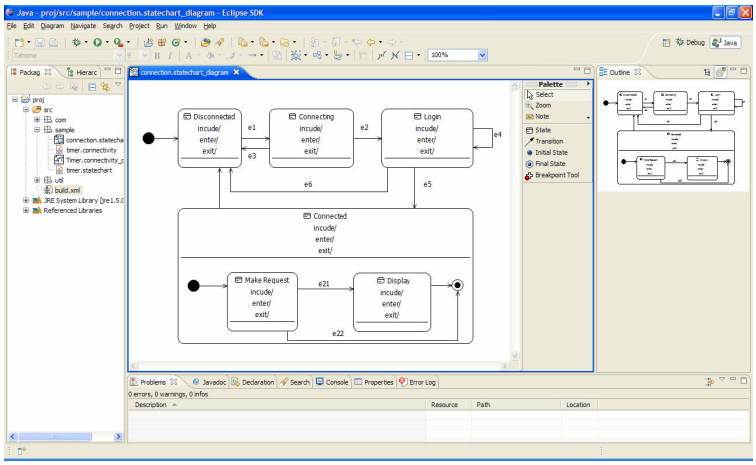

Рис. 6. Внешний вид графического редактора, полученного по доменной модели конечного автомата при помощи технологии *GMF*

Отметим, что универсальность этой технологии позволяет строить редактор по любой модели *EMF*. Таким образом, графический компонент отладчика может быть построен для любого доменно-ориентированного языка, заданного своей метамоделью в виде диаграммы классов *UML*.

Теперь, когда создан компонент, способный графически отображать состояние отлаживаемой программы, перейдем к описанию реализации основной части – ядра отладчика.

Оно должно иметь возможность интегрироваться с любым редактором, построенным описанным образом по метамодели доменного языка при помощи технологий *EMF* и *GMF*.

3.3. Реализация отладчика

Открытая технология среды разработки приложений *Eclipse* позволяет неограниченно расширять функциональность и сферы применения этой технологии. В данной работе для реализации функций отладчика будет использован компонент *Eclipse Debug Core* [2], предоставляющий общий интерфейс для создания отладчиков, интегрированных в среду разработки *Eclipse.* На основе этого интерфейса были реализованы следующие стандартные функции графического интерфейса *Eclipse,* доступные пользователю в режиме отладки приложения:

- создание конфигурации запуска приложения, написанного на доменном языке;
- регистрация «точек останова» (*breakpoints*) в среде разработки *Eclipse*;
- отображение стека вызовов в текущем состоянии выполнения программы;
- отображение информации о контексте выполнения программы (значения переменных).

Используя предоставленный интерфейс, разработанная система отладки интегрируется с инструментами отладки *Eclipse*. Теперь рассмотрим внутреннюю реализацию самого отладчика моделей на доменно-ориентированном языке.

Необходимо обеспечить возможность взаимодействия отладчика с отлаживаемым приложением. В разд. 2.2. был предложен метод, обеспечивающий такое взаимодействие. Метод состоит в разделении отладчика на два компонента:

- компонент отладки, интегрированный в среду разработки приложений;
- клиентская часть отладчика, выполняющаяся в контексте отлаживаемого приложения (например, в виде подключенной на этапе выполнения программы библиотеки).

Компонент отладки, интегрированный в среду разработки приложений, обеспечивает взаимодействие с интерфейсом пользователя и программными интерфейсами отладчика среды *Eclipse.* В его задачу входит оповещение об изменении списка «точек останова», передача команд отладки клиентской части отладчика, отображение текущего состояния отладки на графическом представлении модели отлаживаемой программы и отображение информации о внутреннем состоянии программы, получаемом от клиентской части.

Задачами клиентской части отладчика является хранение списка «точек останова», приостановка выполнения отлаживаемой программы по требованию пользователя или при достижении очередной точки останова и пересылка данных, хранящих описание внутреннего состояния программы. Таким образом, этот компонент является «посредником», обеспечивающим управление выполнением отлаживаемого приложения.

На *рис. 7* изображена схема, отображающая функции компонентов отладчика.

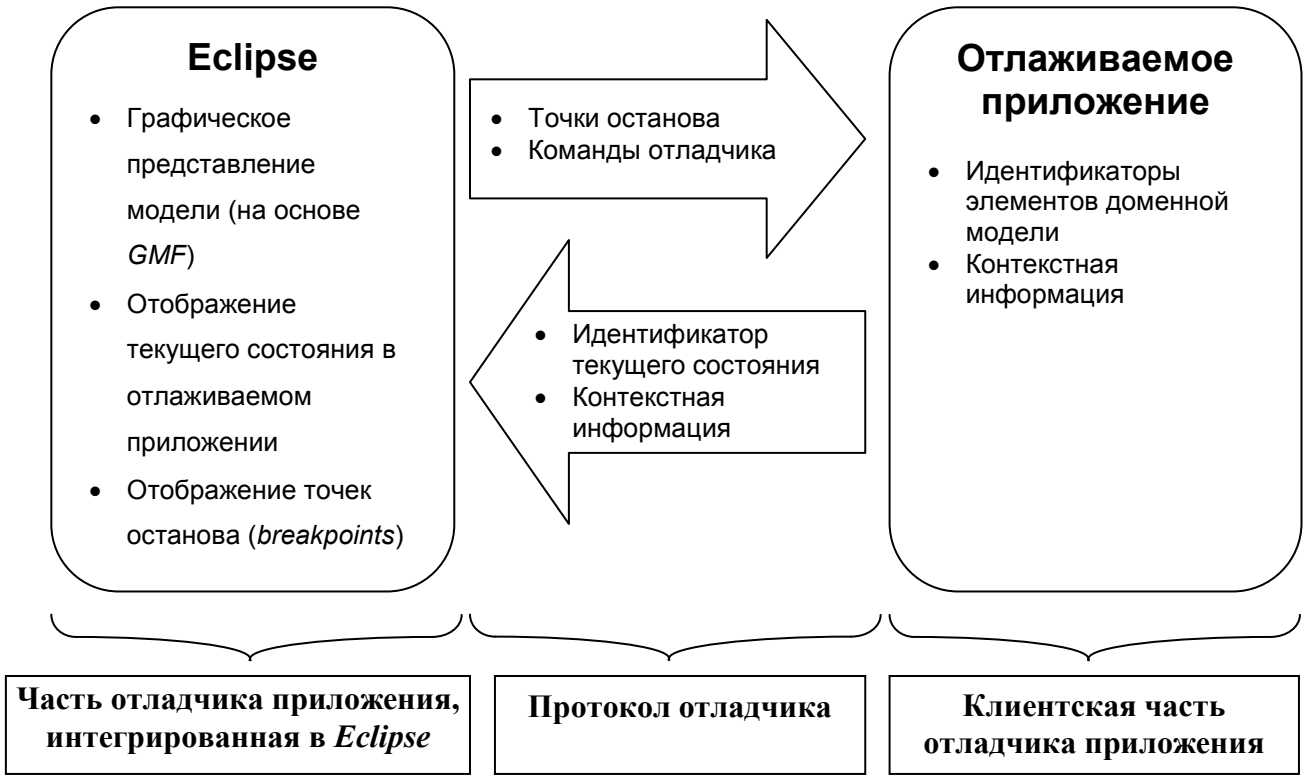

Рис. 7. Схема компонентов отладчика

Для передачи точек останова в контекст отлаживаемого приложения и для проверки соответствия текущей позиции исполнения программы точке останова реализован метод, идея которого была предложена в разд. 2.2. Метод заключается в передаче в сгенерированный код или в модель интерпретируемого языка *URI-идентификатора* элемента доменной модели, из которого был сгенерирован данный участок кода. Этот метод проиллюстрирован на *рис. 8*.

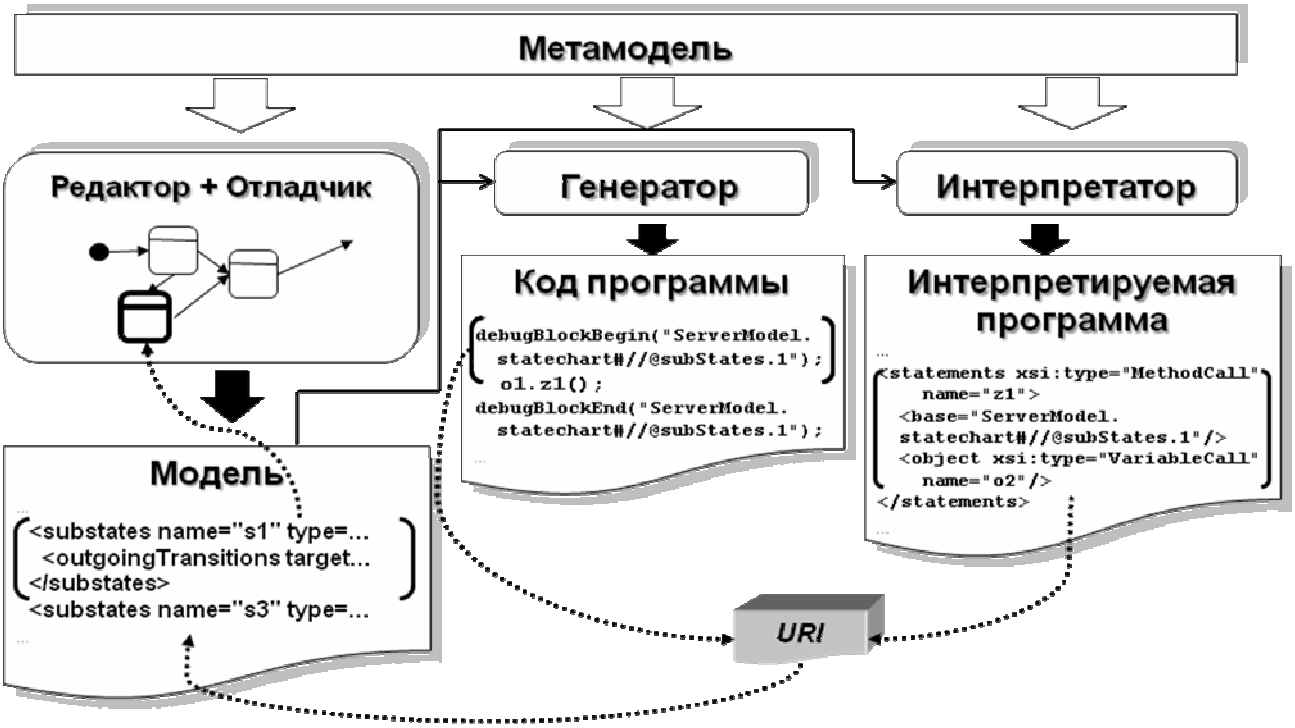

Рис. 8. Передача текущей позиции в отлаживаемой программе

На этом рисунке изображен процесс сопоставления соответствующего элемента доменной модели для участка выполняемого кода. Это сопоставление будет использоваться отладчиком для проверки того, соответствует ли выполняемая в данный момент времени инструкция программы какой-либо заданной заранее точке останова, а также для трассировки хода выполнения программы в графическом представлении модели.

Для преобразования модели программы на доменно-ориентированном языке в вид, пригодный для непосредственного выполнения на компьютере, может использоваться несколько подходов: генерация по модели исполнимого кода на языке общего назначения (например, *Java* или *C++*), либо интерпретация модели. В первом случае в сгенерированный исполняемый код вставляются вызовы вспомогательных инструкций из библиотеки отладчика. При вызове инструкции, в качестве параметра передается идентификатор элемента доменного языка, из которого был сгенерирован соответствующий участок кода. В случае интерпретации, идентификатор элемента читается непосредственно из

интерпретируемой модели и также передается библиотеке отладчика. Таким образом, при любом выбранном подходе библиотека отладчика имеет возможность производить трассировку хода выполнения программы по модели.

Таким образом, рассмотрен простой случай, когда модель построена с использованием одного доменно-ориентированного языка. Однако гораздо больший интерес представляет случай, когда модель построена на основе нескольких языков. Например, когда она реализована на базе автоматного языка, но в состояниях модели производится запрос на языке запросов к базам данных *SQL*. Запросы также построены на базе доменно-ориентированного языка *SQL*-запросов. Таким образом, необходимо работать с несколькими языками одновременно, и сталкиваемся с задачей реализации отладчика для этого случая.

Решим задачу следующим образом:

- введем промежуточный доменно-ориентированный язык, заданный своей метамоделью;
- зададим преобразования из языков, используемых в исходных моделях, в промежуточный язык, либо в другой используемый в модели язык, который, в свою очередь, может быть приведен в промежуточный.

На *рис. 9* рассмотрено применение метода для модели, построенной на трех языках, обозначенных как *Язык 1*, *Язык 2* и *Язык 3*. При этом предполагается, что существуют следующие преобразования:

1. Преобразование из *Языка 1* в *Язык 2*.
2. Преобразование из *Языка 2* в код на промежуточный язык.
3. Преобразование из *Языка 3* в промежуточный язык.

Так как не существует прямого преобразования из *Языка 1* в промежуточный язык, то соответствующий фрагмент исходной модели будет преобразован в код в несколько этапов: сначала он будет преобразован в *Язык 2* с сохранением ссылок на преобразованные элементы исходной модели, а затем полученный фрагмент на *Языке 2* будет преобразован в промежуточный язык.

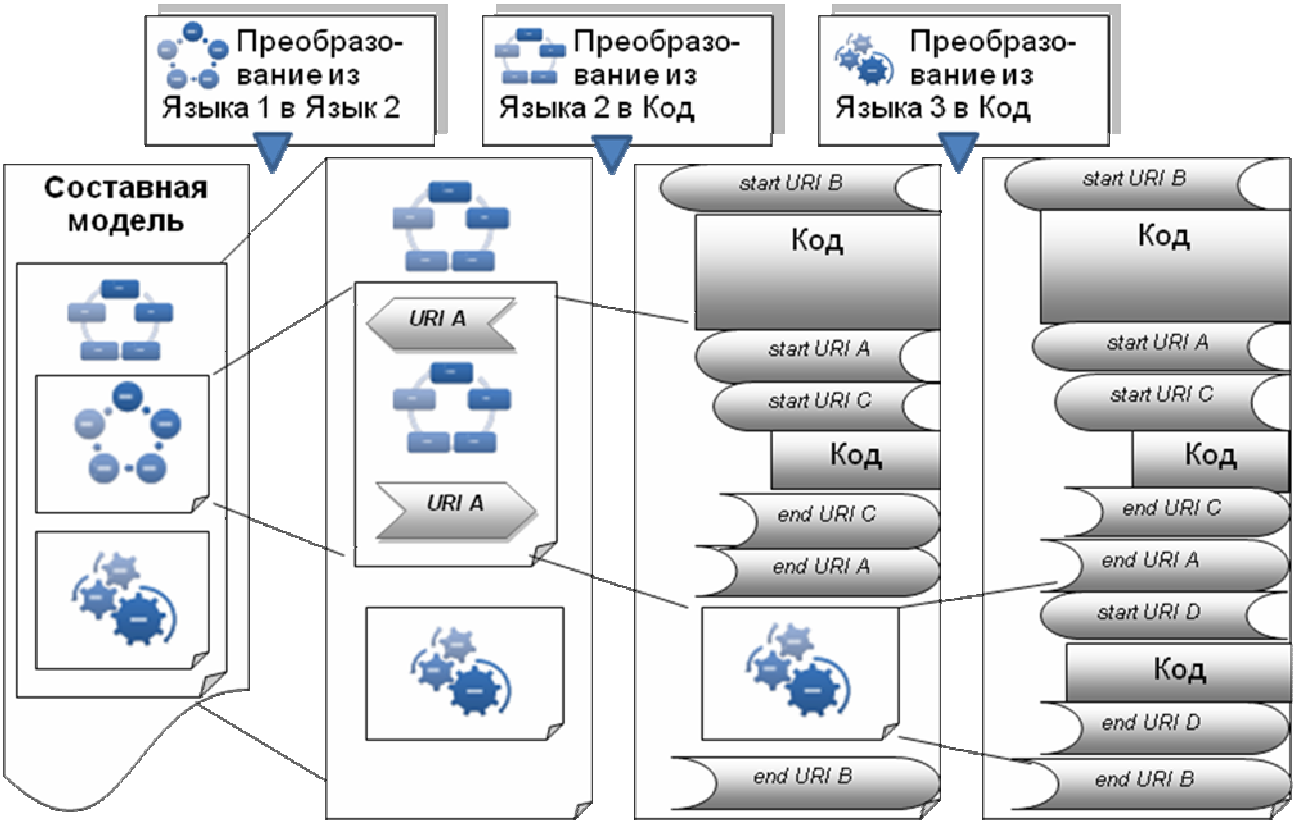

Рис. 9. Случай, когда модель построена с использованием нескольких языков

При преобразовании очередного фрагмента языка в промежуточный язык (на основе сохраненных идентификаторов преобразованных элементов) в модель промежуточного языка добавляются *маркеры*, обозначающие начало и конец фрагмента кода для соответствующего элемента исходной модели и содержащие его идентификатор. При помощи маркера начала и соответствующего ему маркера конца фрагмента, в коде создаются «операторные скобки», соответствующие элементу исходной модели.

Применив описанные преобразования, удается добиться того, что реализация отладчика для программной модели, использующей несколько доменно-ориентированных языков, сведена к реализации отладки для моделей только промежуточного языка.

Процесс визуализации текущего шага отладки в случае интерпретации программной модели будет происходить следующим образом. При выполнении кода будут встречаться маркеры начала и конца кода для соответствующего ему

элемента исходной модели. Маркер начала фрагмента в случае интерпретации инициирует выделение элемента языка на графическом представлении модели. Маркер конца фрагмента, в свою очередь, снимает выделение. Обработка «точек останова» происходит аналогичным образом.

В случае генерации кода на языке программирования, маркеры преобразуется в вызовы функций отладчика, реализующие те же действия. Схема процесса изображена на *рис. 10*.

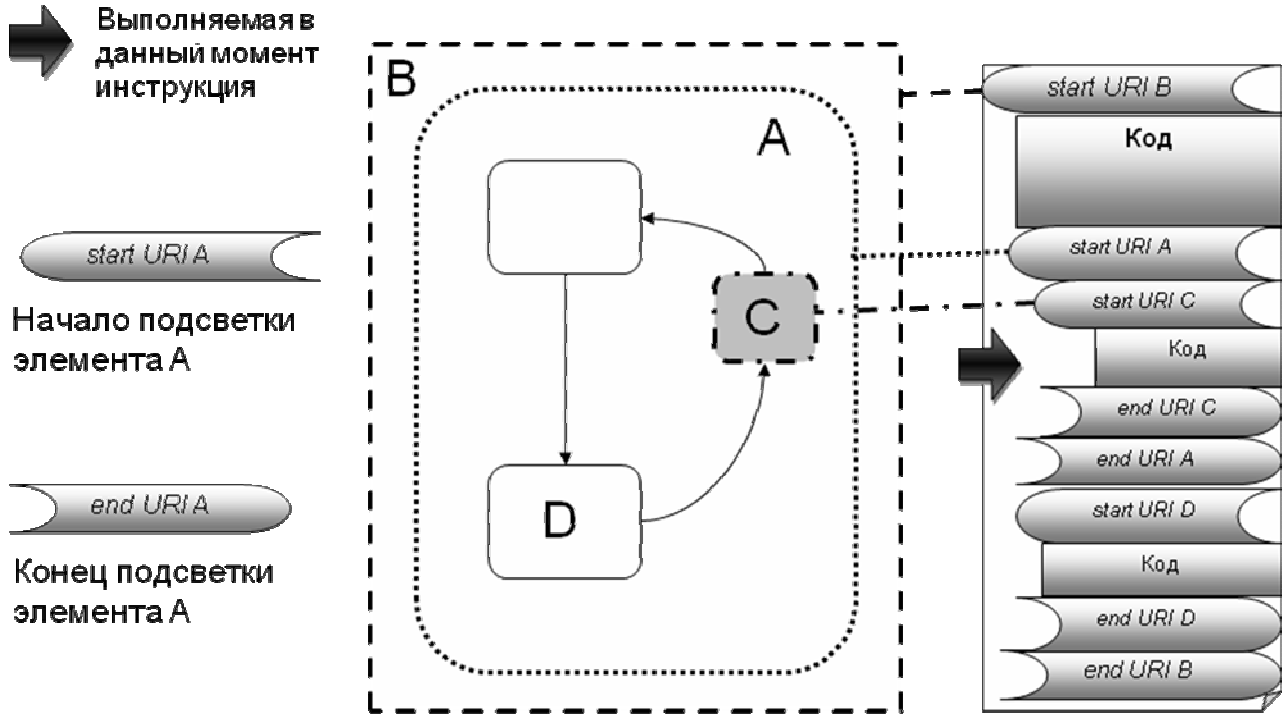

Рис. 10. Визуализация текущего шага отладки

Теперь рассмотрим непосредственно реализацию компонентов отладчика. Для реализации протокола взаимодействия интегрированной в среду разработки и клиентской частей отладчика используется автоматный подход [12]. Этот подход в данном случае позволяет наиболее четко формализовать алгоритм взаимодействия двух компонентов на основе протокола. Эта концепция была успешно применена в построении инструментального средства *UniMod* [13]. Далее будет детально рассмотрена автоматная реализация обоих компонентов отладчика.

3.3.1. Реализация клиентского компонента отладчика

Как уже было отмечено, клиентский компонент отладчика реализован на основе автоматного подхода. Ниже будет приведен список событий и объектов управления автоматов, реализующих логику этого компонента.

В клиентском компоненте отладчика имеется следующий набор событий:

e1 – подключение интегрированного в среду *Eclipse* компонента отладки;

e2 – передача списка «точек останова»;

e3 – очистка списка «точек останова»;

e4 – шаг отладки;

e5 – продолжить выполнение программы;

e6 – отключение интегрированного в среду *Eclipse* компонента отладки;

e7 – принята неопознанная команда;

e8 – невозможно создать подключение к компоненту отладки;

e21 – изменилось состояние отлаживаемой программы;

e22 – отлаживаемое приложение закончило работу.

Автоматы, реализованные в клиентской части отладчика, управляют следующими объектами:

- *o1* – модуль, распознающий команды, поступившие от интегрированного в *Eclipse* компонента. В его задачу входит также информирование о текущей позиции в отлаживаемом приложении и внутреннем состоянии приложения;
- *o2* – модуль, отвечающий за хранение «точек останова» и определяющий их достижение во время выполнения программы;
- *o3* – модуль, управляющий ходом выполнения программы. При необходимости приостанавливает или продолжает выполнение отлаживаемой программы (менеджер потоков).

Выходные воздействия объекта управления *o1*:

- *o1.z0* – осуществить подключение компонента отладчика;

- *o1.z1* – уведомление о том, что выполнение приостановлено в «точке останова»;
- *o1.z2* – уведомление о том, что выполнение приостановлено на очередном шаге в режиме пошагового выполнения;
- *o1.z3* – уведомление о том, что выполнение продолжено;
- *o1.z4* – уведомление о том, что команда не опознана;
- *o1.z5* – уведомление о создании нового потока в системном процессе отлаживаемой программы.

Входные воздействия объекта управления *o2*:

- *o2.x1* – текущая позиция соответствует точке останова.
- *o2.x2* – ожидается подключение компонента отладчика.

Выходные воздействия объекта управления *o2*:

- *o2.z1* – зарегистрировать «точку останова»;
- *o2.z2* – удалить «точку останова»;
- *o2.z3* – удалить все «точки останова».

Выходные воздействия объекта управления *o3*:

- *o3.z1* – приостановить поток;
- *o3.z2* – продолжить выполнения потока;
- *o3.z3* – продолжить выполнение всех потоков;
- *o3.z4* – приостановить выполнение всех потоков.

Рассматриваемый компонент отладчика включает в себя три конечных автомата. Диаграмма состояний основного автомата клиентской части отладчика изображена на *рис. 11*.

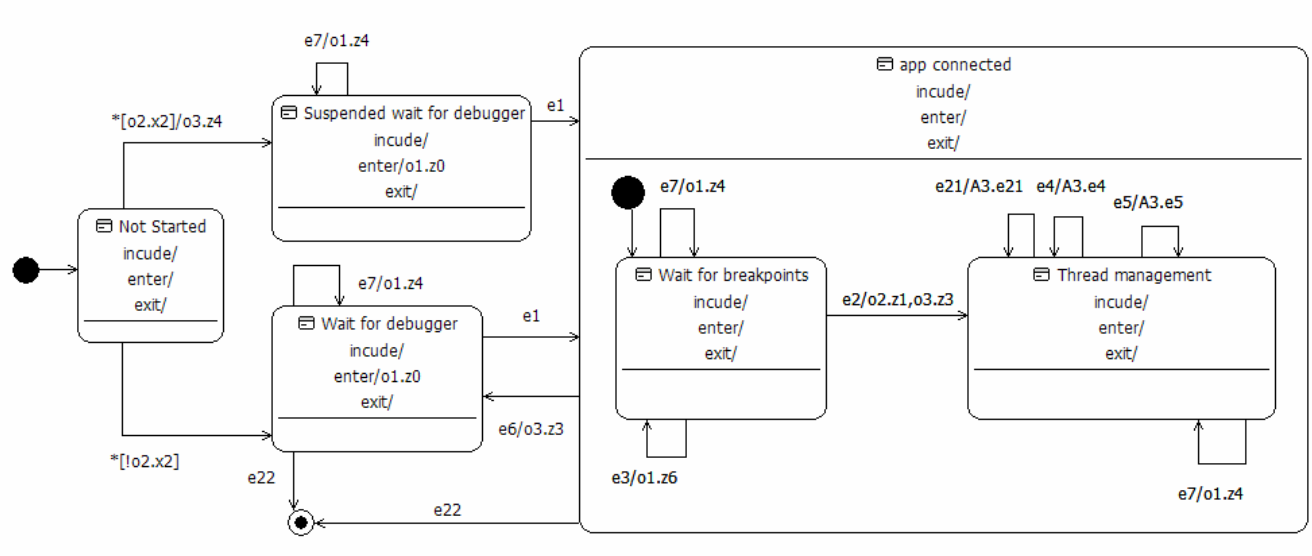

Рис. 11. Основной автомат клиентской части отладчика

Приведенный на *рис. 11* автомат реализует логику поведения клиентской части отладчика. В автомате имеется шесть состояний, два из которых вложенные. Рассмотрим подробнее работу этого автомата. В начале работы автомат находится в состоянии *Not Started*, из которого в зависимости от настройки, заданной при запуске, выполнение отлаживаемого приложения будет либо приостановлено (*o3.z4*) сразу при старте, и автомат перейдет в состояние *Suspended wait for debugger*, либо отлаживаемое приложение начнет свою нормальную работу, а автомат перейдет в состояние *Wait for debugger*. Находясь в любом из состояний *Suspended wait for debugger* или *Wait for debugger,* автомат будет ждать события *e1* подключения компонента отладчика, интегрированного в *Eclipse*. После получения сообщения *e1*, автомат переходит в состояние *app connected.* Это состояние соответствует процессу отладки приложения, и состоит из двух вложенных состояний: *Wait for breakpoints* и *Thread management*. Сразу после перехода в состояние *app connected* текущим состоянием автомата становится *Wait for breakpoints*, в котором происходит ожидание списка «точек останова». После получения списка «точек останова» выполнение продолжается в состоянии *Thread management*. Это основное состояние автомата, в котором производится управление ходом выполнения программы. Однако управление

ведется не непосредственно. Логика управления выполнением программы реализована автоматом *A3*, который рассмотрен ниже. Можно заметить, что оба состояния *Wait for breakpoints* и *Thread management* включают в себя автомат *A2*, управляющий модулем, хранящим список «точек останова» и обрабатывающий события добавления и удаления точек останова.

Диаграмма состояний этого автомата изображена на *рис. 12*.

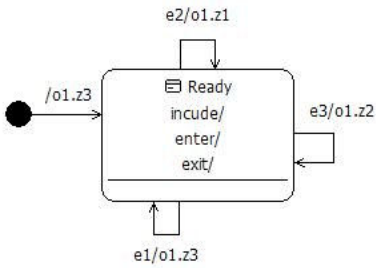

Рис. 12. Автомат клиентской части отладчика *A2*

Рассмотрим диаграмму состояний автомата *A3*. Как уже было сказано, этот автомат управляет выполнением программы. После старта автомат переходит в состояние *No Thread*. В этом состоянии находится до тех пор, пока не будет оповещен об изменении состояния отлаживаемой программы. Далее, если новое состояние программы соответствует «точке останова», то автомат приостанавливает поток и переходит в состояние *Suspended* и уведомляет об этом отладчик, интегрированный в *Eclipse*. В противном случае он переходит в состояние *Resumed*, и уведомляет менеджер потоков о создании нового потока. Из состояния *Resumed* автомат может перейти в состояние *Suspended* при достижении «точки останова». Из состояния приостановки выполнения может быть выполнен переход в состояние продолжения выполнения при получении соответствующей команды из *Eclipse.* Диаграмма состояний автомата *A3* изображена на *рис. 13*.

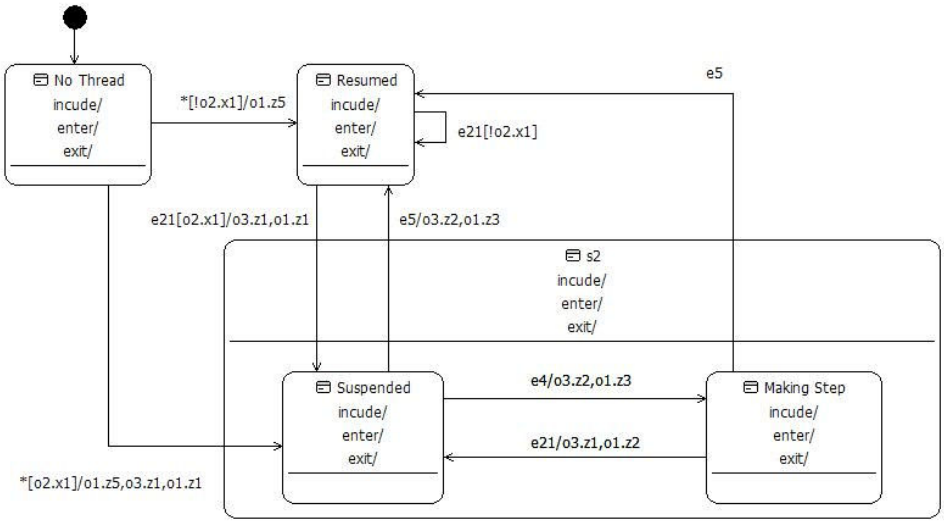

Рис. 13. Автомат клиентской части отладчика *A3*

Функцию приостановки приложения выполняет объект управления *o1* автомата *A3 – менеджер потоков*. Это достигается блокированием потоков отлаживаемого приложения (например, при помощи вызова метода *wait()* при реализации на языке *Java*).

3.3.2. Реализация компонента отладчика, интегрированного в *Eclipse*

Как уже было отмечено, компонент отладчика, интегрированный непосредственно в среду разработки приложений, так же, как и клиентский компонент, реализован на основе автоматного подхода. Ниже будет приведен список событий и объектов управления автоматов, реализующих логику этого компонента:

e1 – отлаживаемое приложение достигло «точки останова» и приостановило работу;

e2 – отлаживаемое приложение завершило шаг отладки и приостановило работу;

e3 – отлаживаемое приложение продолжило выполнение;

e4 – приложение завершило работу;

e5 – произошла ошибка;

e6 – получена неверная команда;

e7 – создан новый поток в системном процессе отлаживаемой программы;

e21 – пользователь добавил «точку останова»;

e22 – пользователь удалил «точку останова»;

e23 – команда пользователя произвести одиночный шаг отладки;

e24 – команда пользователя продолжить выполнение программы.

e25 – команда пользователя завершить отладку;

e26 – отладка завершена.

Автоматы, реализованные в интегрированной в *Eclipse* части отладчика, управляют следующими объектами:

- *o1* – модуль связи с клиентской частью отладчика. В его задачу входит передача списка «точек останова» и команд отладки, поступивших от пользователя;
- *o2* – модуль, взаимодействующий с интерфейсом пользователя и отвечающий за отображение процесса отладки;
- *o3* – модуль, управляющий списком «точек останова».

Выходные воздействия объекта управления *o1*:

- *o1.z1* – команда на пересылку списка «точек останова» клиентскому компоненту;
- *o1.z2* – команда на очистку списка «точек останова» клиентского компонента;
- *o1.z3* – команда на выполнение одиночного шага отладки;
- *o1.z4* – команда на продолжение выполнения приложения;

Выходные воздействия объекта управления *o2*:

- *o2.z1* – уведомление о том, что приложение достигло «точки останова»;

- *o2.z2* – уведомление о том, что приложение совершило одиночный шаг отладки;
- *o2.z3* – уведомление о том, что приложение продолжило выполнение;
- *o2.z4* – уведомление о том, что приложение закончило выполнение;
- *o2.z5* – уведомление о том, что от клиентской части отладчика получено неопознанное сообщение;
- *o2.z6* – уведомление о том, что клиентская часть получила неопознанное сообщение;
- *o2.z7* – уведомление о том, что приложение создало новый поток;

Выходные воздействия объекта управления *o3*:

- *o3.z1* – получить список всех точек останова;

Рассматриваемый компонент отладчика включает в себя три конечных автомата. Диаграмма состояний основного автомата интегрированной в *Eclipse* части отладчика изображена на *рис. 14*.

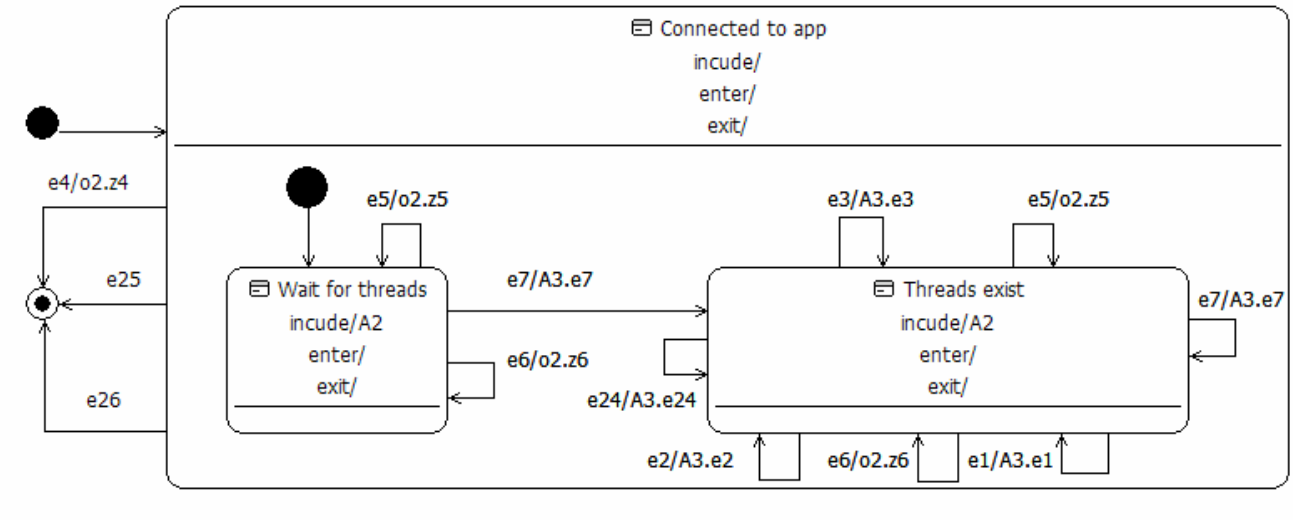

Рис. 14. Основной автомат компонента отладчика, интегрированного в *Eclipse*

Автомат, диаграмма состояний которого приведена на *рис. 11*, реализует логику поведения второй части отладчика – компонента, интегрированного в *Eclipse*. Рассмотрим работу этого автомата. Сразу после старта автомат переходит в состояние *Connected to app*, а затем во вложенное состояние *Wait for threads*. В

этом состоянии ожидается создание потока в системном процессе отлаживаемого приложения. При получении сообщения о создании потока, автомат посылает сообщение другому автомату *A3* (логика этого автомата будет рассмотрена ниже), реализующему управление средствами отладки и взаимодействующему с интерфейсом пользователя, и переходит в состояние *Threads exist*. В этом состоянии автомат получает сообщения, приходящие из пользовательского интерфейса отладки и переадресует их автомату *A3*. Выполнение основного автомата завершается при получении сообщения о завершении работы отлаживаемого приложения (*e4*), либо при получении команды пользователя на завершение отладки (*e25*).

Управление интерфейсом пользователя и отображением процесса отладки выполняет автомат *A3*, диаграмма которого изображена на *рис. 15*.

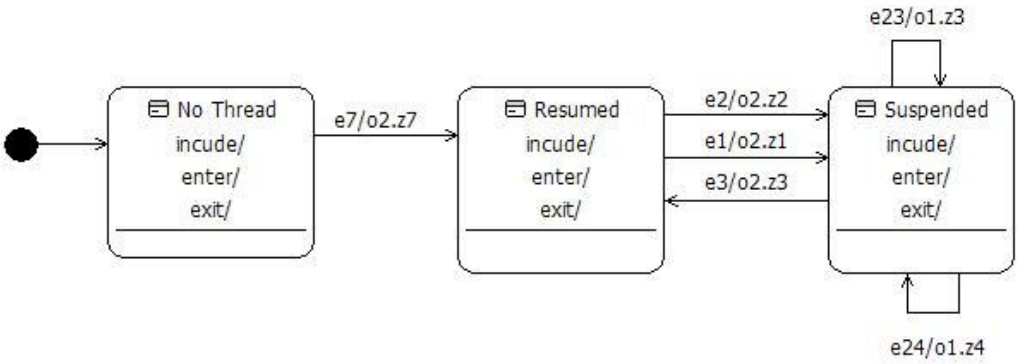

Рис. 15. Автомат *A3* компонента отладчика, интегрированного в *Eclipse*

Первоначально автомат находится в состоянии *No Thread.* При получении сообщения о создании потока в отлаживаемом приложении, выполняется переход в состояние *Resumed*. Это состояние означает то, что отлаживаемая программа находится в процессе выполнения. В процессе отладки автомат выполняет переход между двумя состояниями *Resumed* и *Suspended* в соответствии с сообщениями о текущем состоянии отладки, получаемыми от клиентской части отладчика. Если автомат находится в состоянии *Suspended*, то это означает, что

отлаживаемое приложение приостановлено на очередном шаге вследствие режима пошаговой отладки, либо достижения «точки останова».

Третий автомат рассматриваемого компонента – автомат *A3*, диаграмма состояний которого изображена на *рис. 16*.

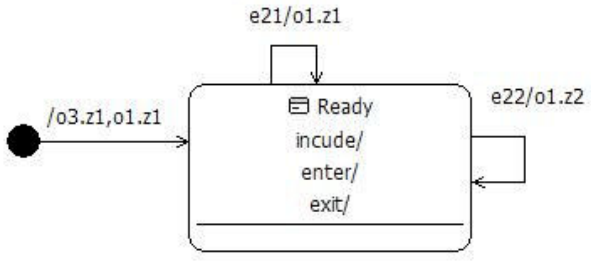

Рис. 16. Автомат *A2* компонента отладчика, интегрированного в *Eclipse*

Этот простой автомат управляет логикой менеджера «точек останова». В его задачу входит информировать клиентскую часть при добавлении (*e21*) и удалении (*e22*) «точки останова», а также пересылка списка «точек останова» при запуске процесса отладки.

3.4. Описание реализованной функциональности

Рассмотрим достигнутую в результате применения метода функциональность. На *рис. 17* изображен фрагмент пользовательского интерфейса добавления «точки останова».

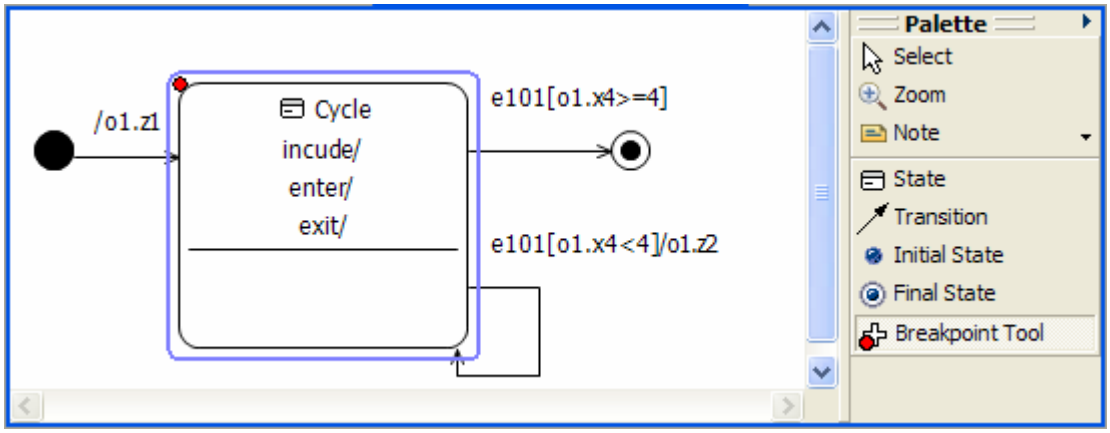

Рис. 17. Установка "точки останова" на состояние конечного автомата

Соответствующий инструмент добавлен на панель инструментов редактора. «Точка останова» отображена на графическом представлении модели в виде красного кружка. На *рис. 18* изображен процесс добавления «точки останова» на переход из одного состояния автомата в другое.

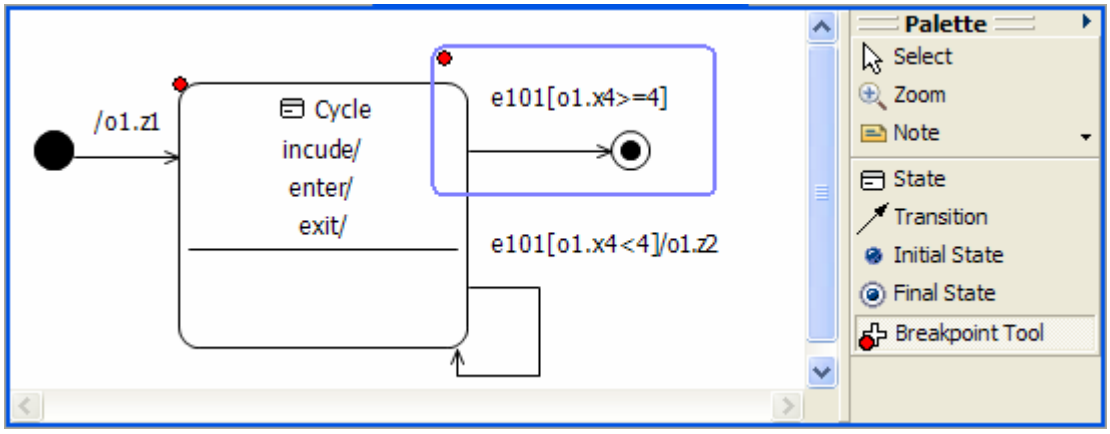

Рис. 18. Установка "точки останова" на переход конечного автомата

На *рис. 19* изображен общий интерфейс приложения *Eclipse* в режиме отладки и обозначены инструменты, доступные при отладке приложения:

1. Графическое представление модели с подсветкой элемента, соответствующего текущему положению в отлаживаемой программе.
2. Инструмент для добавления «точек останова».
3. Элементы управления процессом отладки (приостановка выполнения, одиночный шаг отладки, продолжение выполнения программы после приостановки).
4. Стек вызовов.
5. Отображение контекстной информации.
6. Протокол выполнения отлаживаемой программы.

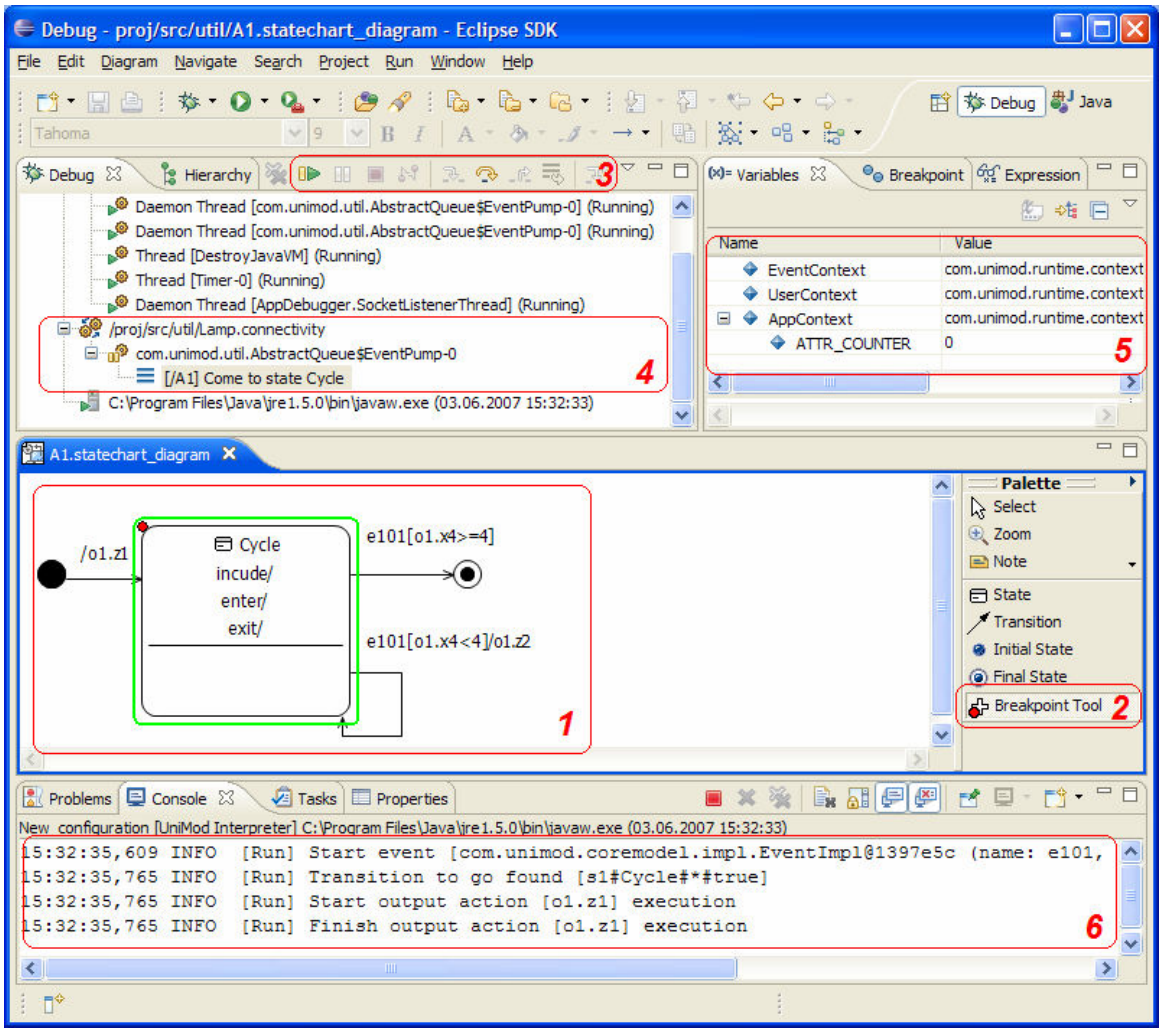

Рис. 19. Инструменты отладки

3.5. Внедрение метода в *UniMod*

Предложенная реализация отладчика, построенного на основе метамодели диаграммы состояний конечного автомата со всеми ее описанными возможностями, будет внедрена в инструментальное средство *UniMod 2* для реализации функции отладки моделей автоматного языка.

3.6. Выводы

1. Дано описание автоматной метамодели.
2. Рассмотрен процесс создания редактора моделей при помощи *GMF* на примере создания редактора автоматных моделей.
3. Рассмотрена реализация отладчика на основе автоматного подхода.
4. Описана полученная в результате применения метода функциональность отладчика:
    - задание «точек останова» (*breakpoints*) непосредственно в графическом представлении модели доменного языка;
    - отслеживание текущего состояния отладки приложения в графическом представлении модели доменного языка;
    - управление выполнением программы (пошаговое выполнение, приостановка выполнения программы в текущем состоянии, продолжение выполнения программы после приостановки);
    - протоколирование поведения отлаживаемой программы;
    - отображения информации о внутреннем состоянии программы (значения переменных).

ЗАКЛЮЧЕНИЕ

В данной работе получены следующие результаты:

1. Проведен анализ существующих средств отладки для доменно-ориентированных языков программирования.
2. Обоснована актуальность создания универсального средства отладки, применимого для произвольного доменно-ориентированного языка.
3. Предложен новый метод построения отладчиков для произвольного доменно-ориентированного языка, заданного своей метамоделью.
4. Построена система, реализующая стандартные функции отладки программ для произвольного доменно-ориентированного языка.
5. Построенная система отладки проинтегрирована в инструментальное средство *UniMod 2* для реализации функциональности отладки автоматных программ.

Возможные пути развития работы:

1. Поддержка дополнительных средств отладки: выражения, вычисляемые во время отладки приложения (*watch expressions*).
2. Поддержка дополнительных средств отладки: «точки останова» по условию (*conditional breakpoints*).

ИСТОЧНИКИ

1. *Чарнецки К., Айзенекер У.* Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005.
2. *Веб-сайт* проекта Eclipse. http://www.eclipse.org/.
3. *Веб-сайт* проекта Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/.
4. *Веб-сайт* проекта Eclipse GMF. http://www.eclipse.org/modeling/gmf/.
5. *Rumbaugh J., Jacobson I., Booch G.* The Unified Modeling Language Reference Manual. Addison-Wesley. 2004.
6. *Фельдман П.* Разработка средств для отладки автоматных программ, построенных на основе предложенной библиотеки. СПбГУ ИТМО. 2005. http://is.ifmo.ru/papers/aut_dlf/.
7. *Веб-сайт* проекта Telelogic Statemate: Embedded Systems Design Software. http://www.telelogic.com/products/statemate/index.cfm.
8. *Веб-сайт* проекта Telelogic Rhapsody: Model-Driven Development Software with UML 2.0. http://www.telelogic.com/products/rhapsody/index.cfm.
9. *Веб-сайт* проекта Telelogic Tau: Model-Driven Development of Complex Systems and Enterprise Applications. http://www.telelogic.com/products/tau/index.cfm.
10. *Буч Г., Якобсон И., Рамбо Дж.* UML. СПб.: Питер, 2006.
11. *Веб-сайт* проекта Eclipse Modeling. http://www.eclipse.org/modeling/.
12. *Шалыто А.А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
13. *Гуров В.С., Мазин М.А.* Веб-сайт проекта *UniMod*. http://unimod.sourceforge.net/.
14. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UML. SWITCH-технология. Eclipse. //Информационно-управляющие системы. 2004. № 6, с.12–17. http://is.ifmo.ru/works/uml-switch-eclipse/

15. *Гуров В. С.*, *Мазин М. А.*, *Шалыто А. А.* UniMod — инструментальное средство для автоматного программирования //Научно-технический вестник. Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий. СПбГУ ИТМО. 2006, с. 32–44. http://books.ifmo.ru/book/vip/210.pdf
16. *IBM Redbook*: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework http://publibb.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246302.html