Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования

Кафедра «Компьютерные технологии»

И. А. Лагунов

Разработка текстового языка автоматного программирования и его реализация для инструментального средства *UniMod* на основе автоматного подхода

Бакалаврская работа

Руководитель – А. А. Шалыто

Санкт-Петербург
2008

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время в сфере разработки программного обеспечения широко используются объектно-ориентированные языки программирования. Однако для большого класса задач объектно-ориентированный подход является не самым оптимальным. Как показано в настоящей работе, *SWITCH*-технология, предложенная в работах [1, 2], является, пожалуй, наиболее естественным решением для задач управления событийными системами. Поэтому ее применение целесообразно для реализации подобных систем. Эта технология представляет собой автоматный подход к описанию поведения программ или программирование с явным выделением состояний.

К настоящему моменту было осуществлено немало попыток облегчить применение этой сравнительно молодой парадигмы – автоматного программирования. Создаются графические среды, позволяющие создавать и редактировать графы переходов автоматов. Одну из таких сред предоставляет проект *UniMod* [3]. Это надстройка для интегрированной среды разработки *Eclipse* [4], позволяющая строить графы переходов автоматов, интерпретировать или компилировать автоматные программы на одном из объектно-ориентированных языков, проверять корректность построения графов переходов, а также отлаживать созданную программу непосредственно в среде разработки. Однако предлагаемый способ описания конечных автоматов – с помощью графического редактора – весьма трудоемок. В то же время существующие текстовые языки автоматного программирования имеют ряд недостатков.

Цель настоящей работы – **разработка текстового языка *FSML* (*Finite State Machine Language*)** для представления конечных автоматов, а также **создание встраиваемого модуля (plug-in)** для среды разработки *Eclipse*, реализующего редактор (*Editor*) языка *FSML*. Разрабатываемый язык предназначен для использования совместно с инструментальным средством *UniMod*. Синтаксический анализатор языка и некоторые возможности

редактора *FSMLEditor* реализованы на основе *SWITCH*-технологии и инструментального средства *UniMod*.

Более подробно ознакомиться с этой технологией можно на сайте http://is.ifmo.ru, а с инструментальным средством *UniMod* – на сайте http://unimod.sourceforge.net.

ГЛАВА 1. ИССЛЕДОВАНИЕ ЗАДАЧИ

1.1. Цели работы

Первая цель работы – разработка текстового языка автоматного программирования *FSML*. Для этого требуется выполнить анализ и провести сравнение существующих языков автоматного программирования, выделить их сильные и слабые стороны, а также сформулировать требования к новому языку. Полученные требования будут лежать в основе разработки языка *FSML*.

Вторая цель работы – создание редактора *FSMLEditor* для разработанного языка. Для того чтобы создать эффективный инструмент разработки, необходимо составить требования к функциональности редактора, исходя из возможностей современных сред разработки программного обеспечения. Поскольку основной частью редактора является синтаксический анализатор языка, необходимо изучить подходы к построению анализаторов языков и выбрать оптимальный для данного случая.

Задача ставится так, что создаваемые язык и редактор будут использоваться совместно с инструментальным средством *UniMod*. Это накладывает определенные ограничения как на процесс разработки текстового языка автоматного программирования, так и на процесс проектирования его редактора.

1.2. Обзор языков автоматного программирования

В настоящее время существует несколько языков автоматного программирования. Безусловно, каждый из них имеет свою специфику в силу того, что авторы используют разные подходы и требования при разработке языков. Поэтому необходимо рассмотреть существующие языки автоматного программирования с целью выделения их особенностей, а затем сформулировать требования к разрабатываемому языку, что и будет сделано в следующем разделе. Для текстовых языков будем отмечать недостатки с учетом применимости их к инструментальному средству *UniMod*.

Перейдем к рассмотрению указанных языков.

1. *AsmL* (*Abstract State Machine Language*) – язык, разрабатываемый компанией *Microsoft* и предназначенный, в первую очередь, для спецификации функциональности компьютерных систем [5]. Язык основан на понятии *абстрактного состояния*, в качестве которого можно рассматривать вектор значений всех переменных. Это значит, что переход в новое состояние происходит при изменении переменных. Поэтому синтаксис языка не предусматривает явного задания графа переходов автомата. Этот язык разработан для платформы *.NET* с возможностью интеграции с другими языками программирования для этой платформы. Для этого языка можно выделить следующие недостатки:

- невозможность явного задания графа переходов автомата;
- ограничение платформой *.NET*.

2. *SMC (State Machine Compiler)* – язык автоматного программирования, в котором в отличие от языка *AsmL* явно описывается граф переходов автомата [6]. Состояния описываются в виде отдельных блоков, которые содержат правила переходов в соответствии с событиями автомата. Программа на языке *SMC* может быть транслирована в код на одном из многих распространенных языков программирования. В качестве недостатков можно отметить:

- отсутствие вложенных автоматов;
- бедность синтаксиса, что снижает читаемость кода программ.

3. *FSMGenerator (Finite State Machine Generator)* – язык описания шаблонов конечных автоматов и средство трансляции этих шаблонов в код на языках программирования *C++* или *Java* [7]. В шаблоне задаются свойства и сущности автомата, такие как имя автомата, множества состояний, событий, переходов и действий. Это приводит к излишней избыточности и неудобству использования в качестве языка программирования. Рассматриваемый подход достаточно удобен лишь для хранения данных автомата. Основные недостатки – избыточность и неудобство описания автоматов.

4. *State Machine* – язык, являющийся расширением языка программирования *Java* за счет введения в него автоматных конструкций [8]. Этот язык основывается на одноименном паттерне проектирования. Каждое состояние реализуется в виде отдельного класса, что позволяет сделать их независимыми друг от друга. Однако это приводит к избыточности кода, увеличению числа и размера классов. Таким образом, данный язык упрощает использование соответствующего паттерна, но практически исключает возможность эффективного описания сложных автоматов. Поэтому основной недостаток – избыточность кода.

5. *TABP* (*Textual Automata Based Programming*) – язык, сочетающий некоторые положительные качества описанных выше языков [9]. Данный язык предоставляет возможности неявного задания графа переходов, задания нескольких автоматов в рамках одной программы, а также наследования, параметризации и обусловливания событий. Более того, этот язык выходит за рамки предметно-ориентированного и является языком общего назначения. Таким образом, автором решена задача создания универсального языка автоматного программирования, и это неизбежно привело к его сложности. Будучи универсальным, язык *TABP* не требует интеграции с другими универсальными языками, однако с помощью интерфейсов может быть осуществлена связь с системой и другими приложениями. Таким образом, основной недостаток – сложность языка, обусловленная его универсальностью.

6. Предметно-ориентированный язык автоматного программирования на базе динамического языка *Ruby* с использованием библиотеки *STROBE* [10] призван решить проблему переноса диаграмм переходов автоматов, разработанных по *SWITCH*-технологии [1, 2], в исполняемый код. Особенностями этого подхода являются декларативная структура кода и его изоморфность исходной диаграмме. Данный язык поддерживает наличие нескольких экземпляров одного и того же автомата, обеспечение связей между ними, а также интеграцию с программами на других языках и возможность

управления физическими устройствами. Однако использование интерпретируемого языка *Ruby* [11] в качестве базового языка накладывает определенные ограничения на сферу применимости данного подхода. Среди этих ограничений можно выделить отсутствие поддержки потоков операционной системы, а также отсутствие компиляции в байткод и встроенной поддержки юникода. Правда, последние два ограничения обходятся использованием специальных компиляторов для компиляции в *Java* и *.NET* байткоды и использованием дополнительных библиотек. Однако это лишь уменьшает и без того невысокую скорость работы программ в этом случае. Таким образом, основной недостаток этого подхода – привязанность к языку программирования *Ruby*.

7. Также заслуживают внимания языки автоматного программирования, созданные с помощью системы метапрограммирования *JetBrains Meta Programming System (MPS)* [12]. Эта система позволяет создавать предметно-ориентированные языки путем задания определенных моделей и редакторов для языка. Структура и внешний вид этих редакторов таковы, что пользователь, работая с текстом программы, напрямую редактирует абстрактное синтаксическое дерево [13] программы, а следовательно модель конечного автомата в случае языка автоматного программирования. Это выводит языки рассматриваемой системы из категории текстовых. С помощью системы *MPS* созданы два языка автоматного программирования: первый – в виде самостоятельного языка, второй – в виде расширения языка *Java*. Эти языки позволяют описывать состояния и логику переходов автоматов по событиям, а также сами события. Кроме того, обеспечена возможность автоматического построения диаграммы состояний по мере набора текста. Заметим, что для рассмотренных выше языков не представляется подобная возможность. Таким образом, можно выделить два недостатка данного подхода:

- привязанность к системе *MPS* (до трансляции в язык общего назначения);
- доступность диаграмм состояний только в режиме просмотра.

8. Проект *UniMod* [3] предлагает визуальный язык автоматного программирования, поддерживая концепцию «Исполняемый *UML*» [14]. С помощью данного языка строятся два типа *UML*-диаграмм: диаграммы классов и диаграммы состояний [15]. Диаграмма классов изображает автоматы, поставщики событий, объекты управления и связи между ними. При этом автоматы описываются диаграммами состояний, а поставщики событий и объекты управления – классами на языке *Java*. Применение этого подхода для реализации систем со сложным поведением показало его эффективность, но также выявило недостаток – трудоемкость визуального ввода диаграмм.

1.3. Требования к языку автоматного программирования

Сформулируем требования к языку автоматного программирования с учетом анализа рассмотренных выше языков. Напомним, что данная работа ограничена созданием текстового языка, который дополнит инструментальное средство *UniMod*. Поэтому часть требований будет относиться к сочетанию этих двух средств разработки.

Текстово-визуальный подход к разработке автоматных программ не реализован полноценно ни в одном из рассмотренных выше языков. Только система метапрограммирования *MPS* и инструментальное средство *UniMod* отчасти реализуют этот подход.

Система *MPS* предлагает возможность просмотра диаграммы состояний создаваемого автомата. Однако на данный момент в ней невозможно создание графических редакторов. Это исключает возможность редактирования диаграммы состояний в виде графа. В то же время эта возможность может быть полезна для внесения быстрых изменений в структуру автомата, а текстовый ввод удобен для быстрого первоначального описания автомата.

Инструментальное средство *UniMod* наоборот значительно теряет в эффективности, не позволяя редактировать автомат в текстовом представлении. Именно этот недостаток призвана решить данная работа.

Явное задание графа переходов позволяет гарантировать его изоморфность программе, что избавит программиста от многих ошибок уже на этапе описания автомата. Кроме того, этого значительно облегчает проверку валидности графа переходов. Заметим, что большинство рассмотренных выше языков соответствовало этому требованию.

Интеграция с объектно-ориентированным кодом необходима для использования языка автоматного программирования на практике, поскольку за исключением языка *TABP* описанные выше языки не являются языками общего назначения. Однако большая часть описанных языков реализует интеграцию с помощью трансляции кода программы в код на одном из универсальных языков. При этом происходит разбор текста программы и преобразование ее в абстрактное синтаксическое дерево (АСД), по которому генерируется код на языке общего назначения (рис. 1).

Рис. 1. Стандартная схема интеграции с объектно-ориентированным кодом

Такая «непрозрачная» связь затрудняет разработку на этих языках, так как сгенерированный код намного сложнее читать, чем исходный код или диаграмму состояний. Подходом к выполнению этого требования выгодно отличается система *MPS*. Она позволяет с помощью ее языков автоматного программирования напрямую редактировать абстрактное представление программы, а следовательно модель конечного автомата в памяти. Затем эта модель может быть транслирована в любой из языков общего назначения или в графическое изображение диаграммы состояний.

Переиспользование компонентов кода позволяет решить проблему дублирования кода и является необходимым требованием при создании сложных систем. Однако в случае языка автоматного программирония это достаточно сильное требование, полноценная реализация которого может

свести почти на нет преимущества такого языка. Такая реализация использована в языке *State Machine*. В результате он является очень громоздким, требуя для каждого состояния создание отдельного класса, объявление и инициализацию дополнительных переменных. Поэтому требуется найти компромисс между удобством проектирования и удобством кодирования.

Краткость и понятность синтаксиса языка является простым, но немаловажным требованием. Стоит заметить, что большинство рассмотренных языков с явным заданием графа переходов имеет достаточно похожую структуру программы и отчасти синтаксис. В целом, от этого требования зависит эффективность использования языка автоматного программирования.

1.4. Требования к редактору языка

Создание редактора языка по сей день остается достаточно сложным делом, несмотря на попытки формализовать и упростить этот процесс. Одна из таких попыток, например, это упомянутая выше система *Meta Programming System* [16]. Однако она обладает существенным недостатком – языки и редакторы, разработанные на ней, невозможно в каком-либо виде отделить от нее и использовать как самостоятельные средства. Поэтому в настоящей работе выбирается традиционный способ разработки редактора языка. Для этого необходимо составить список требований к редактору.

Создаваемый редактор должен быть реализован в виде **встраиваемого модуля для интегрированной среды разработки *Eclipse***. Это требование вызвано необходимостью интеграции редактора с инструментальным средством *UniMod*, которое является модулем к той же среде.

Редактор должен предоставлять возможность для эффективной разработки программ. Это **автоматическое завершение ввода и исправление ошибок, валидация автоматной программы в процессе ее создания и отладка программы**. Многие современные среды разработки программного обеспечения имеют подобные возможности, так как их наличие позволяет значительно ускорить и упростить процесс создания программ.

Интеграция с инструментальным средством *UniMod* должна предоставить синхронизацию кода автоматной программы с диаграммой состояний соответствующего автомата. При этом код программы и диаграмма состояний будут связаны через модель конечного автомата, хранящуюся в памяти. Выполнение этого требования сделает полноценным текстово-визуальный подход к разработке автоматных программ, что было указано в разд. 1.3.

1.5. Подходы к построению анализатора языка

Рассмотрим основные подходы к построению синтаксического анализатора разрабатываемого языка. Нам потребуется выбрать оптимальный подход с учетом сформулированных выше требований. При этом сравним также их возможности для создания систем автоматического завершения ввода и исправления ошибок.

1.5.1. Рекурсивный нисходящий разбор

Простейшим способом реализации синтаксического анализатора является рекурсивный нисходящий разбор выражения. Для этого используются универсальные технологии, такие как «компилятор компиляторов» *ANTRL* [17], применяемый, в частности, в проекте *UniMod*. Он по заданной *LL(k)*-грамматике генерирует код, реализующий лексический анализатор и рекурсивный нисходящий синтаксический анализатор. Это позволяет проверить принадлежность выражения заданному грамматическому языку и построить по выражению абстрактное синтаксическое дерево.

Однако данный синтаксический анализатор не может быть использован для построения системы автоматического завершения ввода, поскольку в случае подачи ему на вход префикса выражения на заданном языке вместо законченного выражения он выдает ошибку.

1.5.2. Нерекурсивный нисходящий разбор

Другим вариантом реализации требуемой системы является применение нерекурсивного нисходящего синтаксического анализатора, использующего стек и управляемого таблицей разбора. Таблица разбора представляет собой двумерный массив, в котором содержатся продукции грамматики для каждой пары из нетерминала грамматики и терминала входной строки. Эти продукции могут быть использованы для замены нетерминалов на вершине стека в процессе разбора выражения. Пустые ячейки таблицы разбора означают ошибки. Такой анализатор подробно описан в работе [13].

Система автоматического завершения ввода реализуется довольно естественно: в качестве строк для завершения выбираются все терминалы, для которых соответствующие ячейки таблицы разбора непусты. Для реализации системы исправления ошибок таблица разбора может быть дополнена синхронизирующими символами и указателями на подпрограммы обработки ошибок, вписываемые в некоторые пустые ячейки.

1.5.3. Разбор с помощью конечного автомата

В работе [18] показано, как создать программу нерекурсивного нисходящего синтаксического анализатора, используя автоматно-ориентированный подход. При этом таблица разбора оставлена и выступает в роли объекта управления автомата.

В работе [19] предлагается технология создания системы автоматического завершения ввода, позволяющая исключить таблицу разбора нисходящего нерекурсивного синтаксического анализатора и использующая гибкий алгоритм восстановления после ошибок на уровне фразы. Данная технология основывается на построении конечного автомата типа Мили из диаграмм переходов, соответствующих правилам вывода *LL(1)*-грамматики. Построенный автомат будет синтаксическим анализатором для заданной грамматики, реагируя на события, которые поставляет ему лексический анализатор. Каждому такому событию соответствует терминальный символ.

При подаче на вход системе, описанной выше, незавершенной строки, автомат, реализующий синтаксический анализатор, останавливается в каком-то состоянии. События, заданные на переходах из состояния, в котором остановился автомат, определяют множество терминалов, которые могут следовать за последним терминалом, извлеченным из входной строки. Для исправления ошибок ввода используется алгоритм, автоматически добавляющий недостающие переходы из каждого состояния. При этом учитываются оба возможных варианта ошибок: либо некоторые лексемы были пропущены, либо введено что-то лишнее.

Выводы по главе 1

1. Сформулированы цели работы.
2. Выполнен обзор существующих языков автоматного программирования, выделены их недостатки.
3. Сформулированы требования к языку автоматного программирования и редактору этого языка.
4. Рассмотрены известные подходы к созданию синтаксического анализатора языка.

ГЛАВА 2. ОПИСАНИЕ ЯЗЫКА *FSML*

В ходе выполнения данной работы был разработан язык *FSML*. Он предназначен для текстового представления конечных автоматов и их связей. Его возможности позволяют с использованием синтаксиса, близкого к синтаксису языка программирования *Java*, описать события, состояния, вложенные автоматы, переходы и другие элементы автоматной модели. В этой главе приведем полное описание возможностей разработанного языка.

2.1. Пример *fsml*-программы

Описание языка начнем с примера, реализующего модель пешеходного светофора с таймером (Листинг 1).

Листинг 1. Код примера на языке *FSML*

```
1    uses trafficlight.provider.TrafficSignalProvider;
2
3    statemachine TrafficLightWithTimer {
4
5        trafficlight.object.RedLamp red;
6        trafficlight.object.GreenLamp green;
7        trafficlight.object.Timer timer;
8
9        initial Init {
10           transitto Inactive;
11       }
12       Active {
13           on switch transitto Inactive;
14
15           initial InitActive {
16               execute red.turnOn, timer.reset
17               transitto Red;
18           }
19           Red {
20               on tick if timer.value == 0
21                   execute red.turnOff, green.turnOn, timer.reset
22                   transitto Green;
23               on tick else
24                   execute timer.decrement;
25           }
26           Green {
27               on tick if timer.value < 5
28                   execute green.turnBlinking, timer.decrement
29                   transitto GreenBlinking;
30               on tick else
31                   execute timer.decrement;
32           }
33           GreenBlinking {
34               on tick if timer.value == 0
35                   execute green.turnOff, red.turnOn, timer.reset
36                   transitto Red;
```

```
37                on tick else
38                    execute timer.decrement;
39            }
40        }
41        Inactive {
42            on enter execute red.turnOff, green.turnOff, timer.turnOff;
43            on switch transitto Active;
44            on stop transitto Final;
45        }
46        final Final {
47        }
48    }
```

Этой программе соответствуют диаграмма связей и диаграмма состояний, представленные на рис. 2 и 3 соответственно. Эти диаграммы оптимизированы вручную для большей наглядности. В приложении 1 приведены те же диаграммы, сгенерированные и расположенные на плоскости **автоматически** средствами *FSML* и *UniMod*.

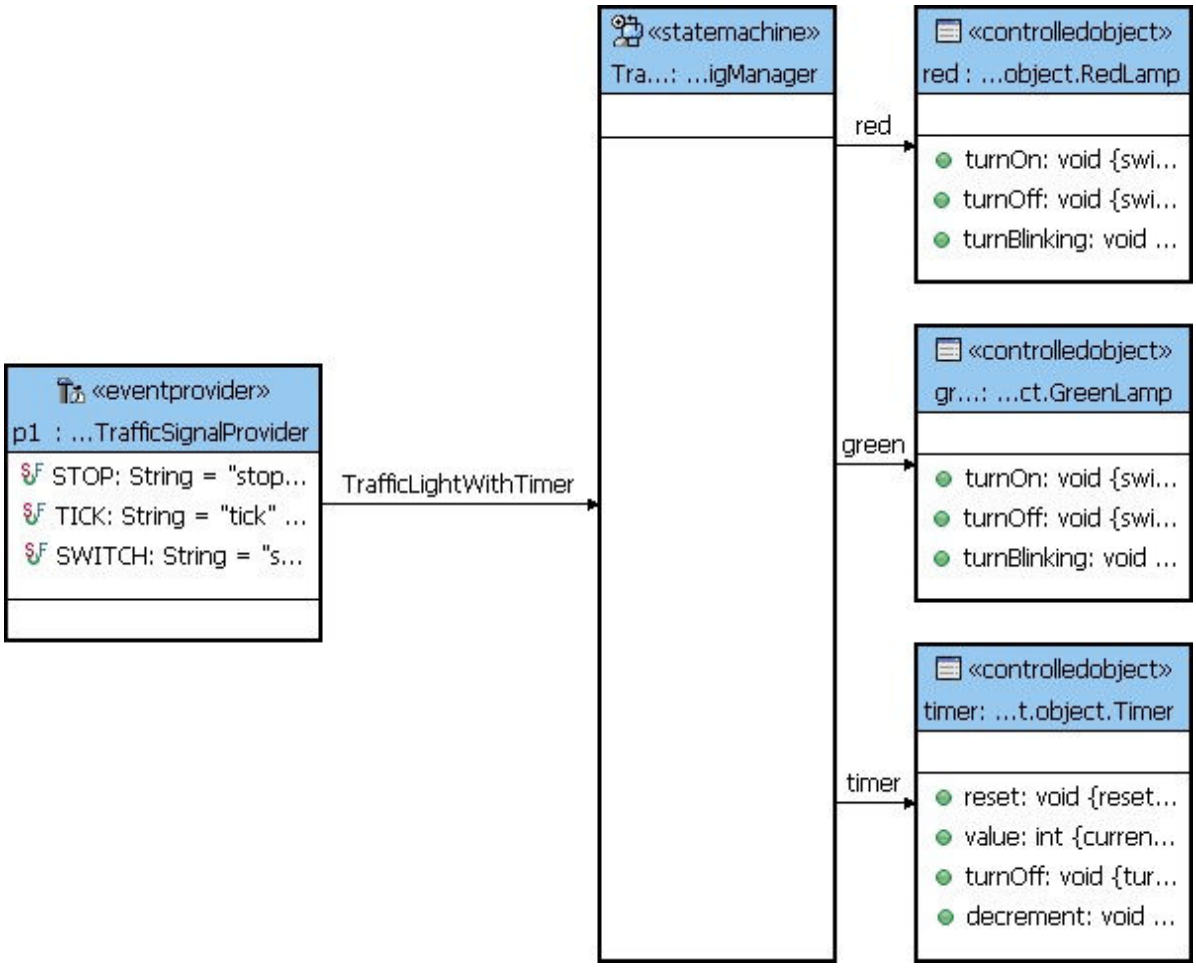


Рис. 2. Диаграмма связей для примера

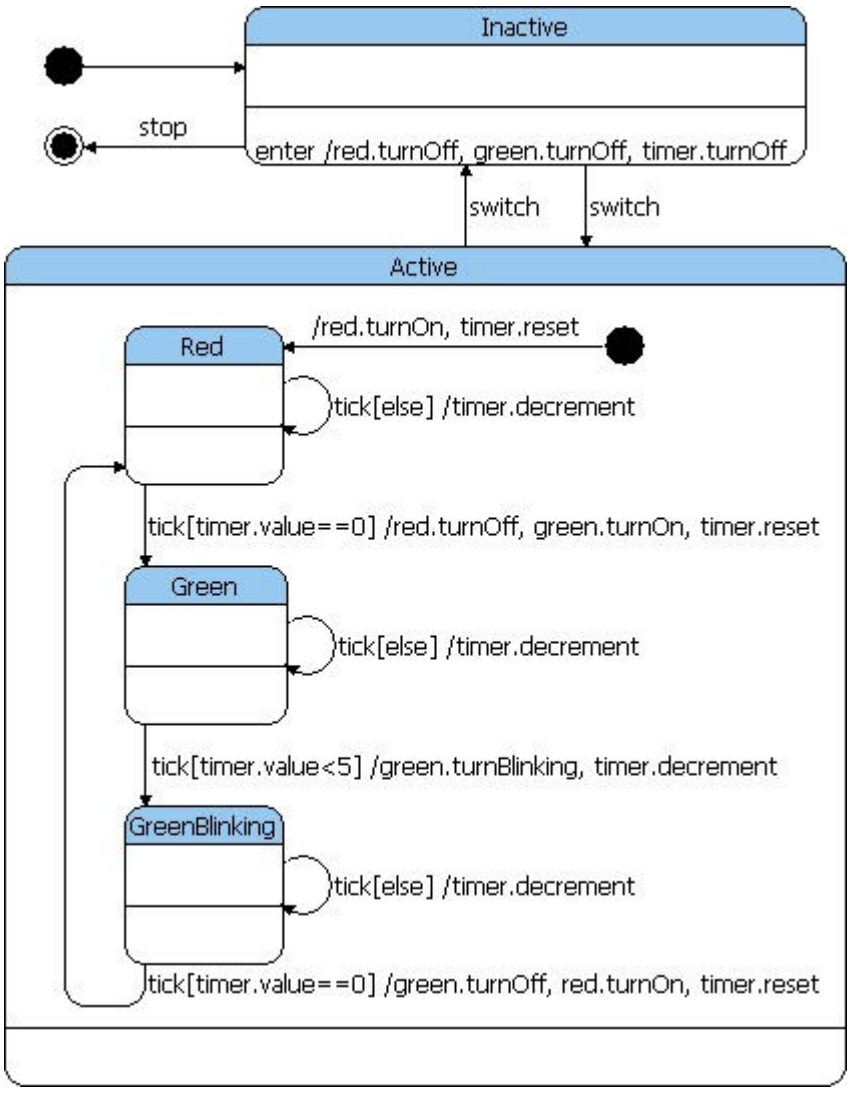


Рис. 3. Диаграмма состояний для примера

Итак, рассмотрим пример более подробно. Сначала объявляется поставщик событий данной системы:

- `trafficlight.provider.TrafficSignalProvider` – он предоставляет следующие события:
  - `switch` – реализует сигнал включения/выключения светофора;
  - `tick` – реализует сигнал от системного таймера;
  - `stop` – реализует сигнал завершения работы системы.

Дальше объявляется автомат `TrafficLightWithTimer` и его объекты управления. Каждой из физических частей светофора соответствует свой объект управления:

- `trafficlight.object.RedLamp red` – красная лампа. Этот объект управления имеет следующий интерфейс:
  - `turnOn` – включить лампу;
  - `turnOff` – выключить лампу;
  - `turnBlinking` – включить лампу в режиме мигания;
- `trafficlight.object.GreenLamp green` – зеленая лампа. Этот объект управления аналогичен предыдущему;
- `trafficlight.object.Timer timer` – табло таймера. Этот объект управления имеет следующий интерфейс:
  - `turnOff` – выключить табло таймера;
  - `decrement` – уменьшить значение таймера на единицу;
  - `reset` – сброс таймера к начальному значению.

Оставшуюся часть программы занимают описания состояний и переходов между ними. В рассмотренном примере помимо начального (**`initial`** `Init`) и конечного (**`final`** `Final`) состояний автомата присутствуют следующие:

- `Active` – светофор включен. Это сложное состояние, содержащее основной цикл работы светофора:
  - `InitActive` – начальное состояние после включения светофора;
  - `Red` – включен красный сигнал светофора;
  - `Green` – включен зеленый сигнал светофора;
  - `GreenBlinking` – включен зеленый мигающий сигнал светофора;
- `Inactive` – светофор выключен.

Заметим, что это далеко не единственная и не лучшая реализация автомата, управляющего светофором. Можно произвести декомпозицию и выделить содержимое состояния `Active` в отдельный автомат, сделав его вложенным в это состояние. Это позволит упростить как программу на языке

FSML, так и соответствующую диаграмму состояний. Синтаксис данной конструкции описан в разд. 2.2.5.

2.2. Синтаксис языка

В этом разделе описывается синтаксис языка программирования *FSML*, его ключевые слова и конструкции. В целом, по стилю данный язык похож на язык программирования *Java*: для выделения вложенных конструкций используются фигурные скобки, объекты управления и сторожевые условия на переходах аналогичны соответственно объектам и условиям языка *Java*.

2.2.1. Ключевое слово `uses` – объявление поставщика событий

Ключевое слово `uses` используется для объявления поставщика событий автомата. Поставщик событий является *Java*-классом, для идентификации которого применяется полный путь – строка вида: <имя пакета>.<имя класса>.

Общий синтаксис:

```
uses <имя пакета>.<имя класса>;
```

Пример:

```
uses trafficlight.provider.TrafficSignalProvider;
```

2.2.2. Ключевое слово `statemachine` – объявление автомата

Ключевое слово `statemachine` используется для объявления конечного автомата. Соответствующая конструкция создает новый уровень вложенности и содержит всю оставшуюся часть программы.

Общий синтаксис:

```
statemachine <имя автомата> { <описание автомата> }
```

Пример:

```
statemachine TrafficLightWithTimer { … }
```

2.2.3. Объявление объекта управления

Объявление объекта управления автомата аналогично объявлению переменной в языке программирования *Java*, за исключением необходимости указывать полный путь к *Java*-классу, реализующему данный объект. Все объекты управления должны быть объявлены в самом начале описания автомата – до описания состояний.

Общий синтаксис:

```
<имя пакета>.<имя класса> <имя объекта управления>;
```

Пример:

```
trafficlight.object.Timer timer;
```

2.2.4. Объявление состояния

Существует три типа состояний: начальное, обычное и конечное. Каждое состояние создает новый уровень вложенности и содержит внутри свое описание.

Общий синтаксис начального состояния:

```
initial <имя состояния> { <описание состояния> }
```

Общий синтаксис обычного состояния:

```
<имя состояния> { <описание состояния> }
```

Общий синтаксис конечного состояния:

```
final <имя состояния> { }
```

Примеры:

```
initial Init { … }
Active { … }
final Final { }
```

Каждый тип состояния имеет ограничения на свое описание. Начальное состояние должно содержать единственный переход без события и сторожевого условия. Обычное состояние может содержать объявления вложенных автоматов, переходы и вложенные состояния, однако не может быть пустым. Конечное состояние должно быть пустым и приводится только для полноты

явного задания диаграммы состояний автомата. Более подробное описание переходов, сторожевых условий и вложенных автоматов приводится в следующих пунктах.

Состояния должны идти в порядке, указанном выше: сначала одно начальное состояние, затем произвольное число обычных состояний и в конце одно конечное состояние. В случае вложенных состояний начальное и конечное состояния могут опускаться.

2.2.5. Ключевое слово `include` – объявление вложенных автоматов

Поддержка вложенных автоматов позволяет производить декомпозицию сложных автоматов, а также переиспользовать части автоматных программ. Для этого используется специальная конструкция, которая должна идти в начале описания состояния, содержащего вложенный автомат. Возможно объявление сразу нескольких вложенных автоматов в виде разделенного запятыми списка.

Общий синтаксис:

```
include <список вложенных автоматов>;
```

Пример:

```
include TrafficLightWithTimer, AnotherAutomaton;
```

2.2.6. Ключевые слова `on`, `if`, `else`, `execute`, `transitto` – объявление перехода

Объявление перехода в языке *FSML* задается сложной конструкцией и может состоять из нескольких частей: события, сторожевое условие, список действий и целевое состояние. Рассмотрим сначала общий синтаксис конструкции перехода, а затем обратимся к его составным частям.

Общий синтаксис:

```
on <события> <сторожевое условие>
execute <список действий>
```

```
transitto <целевое состояние>;
```

Пример:

```
on tick if timer.value < 5
execute  green.turnBlinking, timer.decrement
transitto GreenBlinking;
```

События могут быть нескольких типов:

- Обычное событие (от поставщика событий). Возможно задание сразу списка таких событий через запятую. Следовательно, одно объявление перехода на языке *FSML* может задавать несколько переходов на соответствующей диаграмме состояний.
- Событие при входе в состояние – **`enter`**.
- Любое обычное событие – **`any`**.
- Событие при выходе из состояния – **`exit`** (будет в версии для *UniMod 2*).

Примеры:

```
on tick, switch, stop
on enter
on any
```

Сторожевое условие добавляет зависимость перехода от объектов управления – переход выполняется только при выполнении указанного условия. Оно должно удовлетворять грамматике, записанной в листинге 2.

Листинг 2. Грамматика, задающая синтаксис сторожевого условия

```
S → 'else'
S → 'if' I0
I0 → I0 '||' I1
I0 → I1
I1 → I1 '&&' I2
I1 → I2
I2 → '!' I3
I2 → I3
```

```
I3 → '(' S ')'
I3 → I4
I4 → I5 rel I5
I4 → ident_bool
I4 → const_bool
I5 → ident_number
I5 → const_number
```

Здесь `rel` является строковым представлением отношения: `'>'`, `'<'`, `'>='`, `'<='`, `'!='`, `'=='`, `ident_bool` и `ident_number` – входные значения объекта управления типов *boolean* и *int* соответственно, `const_bool` – константа *'true'* или *'false'*, `const_number` – целочисленная константа. Сторожевое условие `else` – дополнение всех остальных условий для тех же событий.

Пример:

```
if (timer.value < 5) && (timer.value >= 0)
```

Список действий через запятую перечисляет выходные действия объектов управления, которые должны выполниться при данном переходе.

Пример:

```
execute  green.turnBlinking, timer.decrement
```

Целевое состояние определяет, куда будет совершен переход.

Пример:

```
transitto GreenBlinking
```

При объявлении перехода могут отсутствовать некоторые из вышеперечисленных частей. Рассмотрим все случаи (таблица), поставим '+', если часть должна присутствовать, '–' – если она должна отсутствовать, '?' – ее наличие опционально.

Таблица. Формат конструкции перехода для разных состояний

Состояние	События	Сторожевое условие	Список действий	Целевое состояние
Начальное	–	–	?	+
Обычное	+	?	?	?

В случае отсутствия целевого состояния в переходе из обычного состояния переход считается «петлей» – исходное и целевое состояния совпадают.

2.2.7. Комментарии

Данный язык позволяет использовать комментарии подобно языку программирования *Java*. Допустимы два типа комментариев: однострочные и многострочные.

Примеры:

```
// Однострочный комментарий
/* Многострочный
комментарий */
```

2.3. Интеграция с объектно-ориентированным кодом

Язык *FSML* совместно с инструментальным средством *UniMod* выгодно отличаются от других средств автоматного программирования. Можно выделить два главных отличия схемы на рис. 4 от схемы на рис. 1, соответствующей другим текстовым языкам автоматного программирования:

- сразу строится конкретная автоматная модель вместо абстрактного синтаксического дерева автоматной программы;
- существует и активно используется обратная связь с объектно-ориентированным кодом – через объекты управления автомата.

Эти два отличия в совокупности позволяют работать непосредственно с моделью автомата, запуская его в интерпретируемом режиме. В этом случае

отпадает необходимость трансляции автоматного кода в код на языке общего назначения. Однако такая возможность имеется на случай необходимости применения компилятивного подхода.

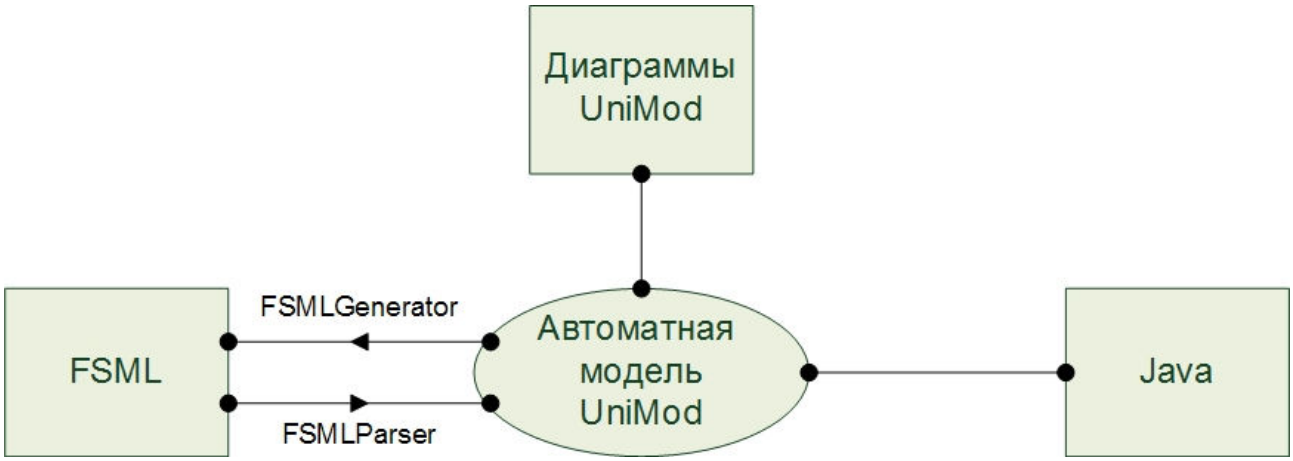


Рис. 4. Схема интеграции языка *FSML* с объектно-ориентированным кодом

Выводы по главе 2

1. Приведен и разобран пример программы на языке *FSML*.
2. Подробно описан синтаксис разработанного языка.
3. Оценены преимущества подхода к интеграции с объектно-ориентированным кодом по сравнению с другими языками.

ГЛАВА 3. ПРОЕКТИРОВАНИЕ РЕДАКТОРА ЯЗЫКА *FSML*

Данная глава посвящена подробному описанию созданного редактора языка *FSML*. Здесь в порядке подглав описываются возможности редактора, проектные решения, принятые на основе требований, которые были сформулированы в разд. 1.4. После этого приводится проектная документация, включающая подробное описание автоматной части проекта и обзор неавтоматной.

3.1. Обзор возможностей редактора

Редактор состоит из лексического и синтаксического анализаторов, генератора объектных автоматных моделей, системы автоматического завершения ввода и других средств. В перспективе планируется добавить отладчик кода программы, компоновщик, производящий укладку автомата на плоскости для отображения в виде графа, генератор *fsml*-программ из объектной автоматной модели.

Рассмотрим более подробно каждый элемент. *Лексический анализатор* осуществляет чтение входной цепочки символов и их группировку в элементарные конструкции, называемые лексемами. *Синтаксический анализатор* выполняет разбор исходной программы, используя поступающие лексемы, а также семантический анализ программы. *Генератор объектной автоматной модели* строит конечное представление автомата, сохраненного в *fsml*-программе. *Система автоматического завершения ввода* предоставляет пользователю список строк, при добавлении которых редактируемая программа будет синтаксически верна. *Отладчик* подсвечивает семантические и синтаксические ошибки в коде программы и предоставляет средства для интерактивного поиска нетривиальных семантических ошибок. *Компоновщик* укладывает сгенерированную объектную модель автомата в виде графа на плоскости для визуализации в пакете *UniMod*. *Генератор fsml*-программ осуществляет обратную связь, преобразуя отображаемые визуально объектные

модели автоматов в программы на языке *FSML*. Это позволит пользователю полноценно редактировать конечные автоматы, как в текстовом представлении, так и в визуальном.

3.2. Проектные решения

В этом разделе рассмотрим функции редактора языка *FSML* и соответствующие им проектные решения. Полученная структура редактора изображена на рис. 5.

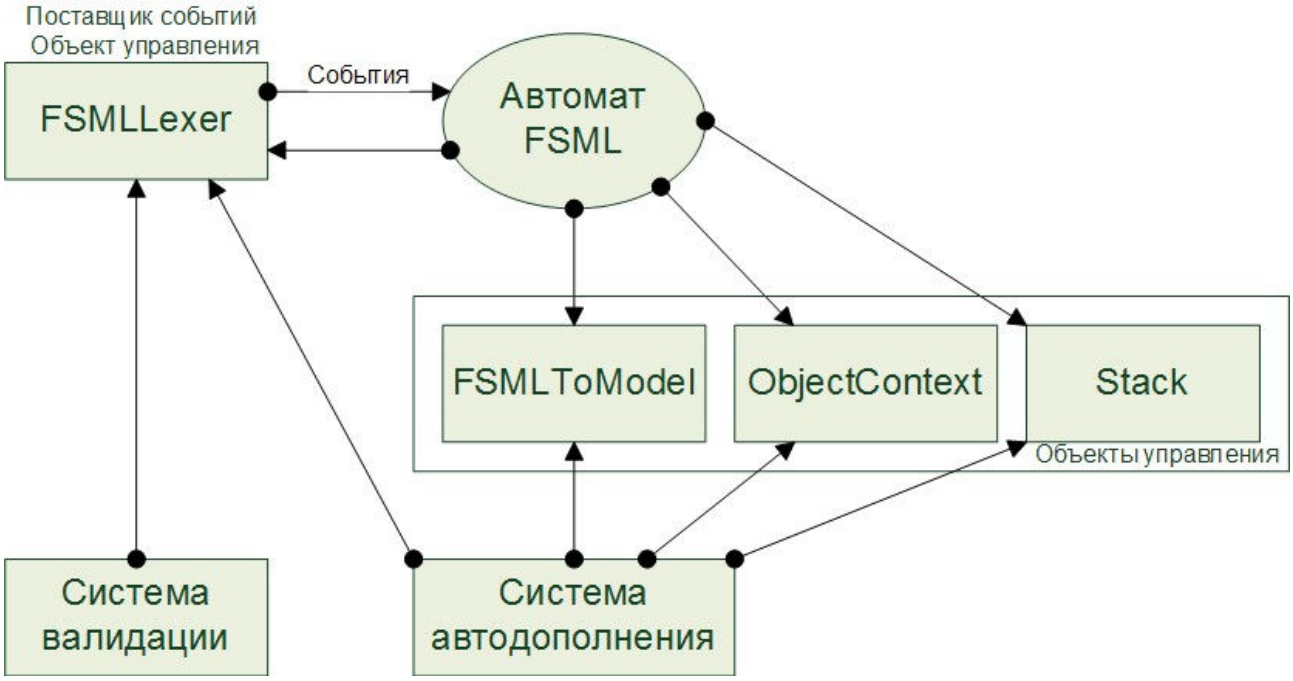


Рис. 5. Структура редактора языка *FSML*

1. Основной функцией данного редактора является **построение автоматной модели по программе на языке *FSML***. Автоматная модель – это внутреннее представление диаграмм состояний автоматов и схемы их связей с поставщиками событий и объектами управления. Для реализации этой функции редактор языка *FSML* включает в себя синтаксический анализатор и генератор объектной автоматной модели. Эти средства естественно представляются в виде событийной системы, поэтому было решено реализовать их на основе автоматного подхода с использованием пакета *UniMod* [3]. Синтаксический анализатор – это система, которая получает события в виде поступающих лексем и управляет генератором объектной модели. Таким образом,

синтаксический анализатор реализован в виде конечного автомата *FSML*, лексический анализатор – в виде поставщика событий-лексем *FSMLLexer*, генератор объектной автоматной модели – в виде объекта управления *FSMLToModel* для этого автомата. Для выполнения рассматриваемой функции автомату также требуются вспомогательные данные, реализованные в виде объектов управления *ObjectContext* и *Stack*. Поставщик событий подробно описан в разд. 3.4, объекты управления – в разд. 3.5, автомат – в разд. 3.6.

2. Дополнительная функция редактора – **валидация программы**. Она позволяет автоматически находить синтаксические и семантические ошибки в программе. Эта система использует данные, генерируемые лексическим и синтаксическим анализаторами, поэтому для нее достаточно единственной зависимости – от объекта *FSMLLexer*. В данной работе валидация реализована без поиска семантических ошибок, поскольку эта часть уже реализована со стороны инструментального средства *UniMod*.

3. Система автодополнения подразумевает **автоматическое завершение ввода пользователя**. Автоматный подход значительно упрощает реализацию этой функции. Для этого используется построенный автомат, который позволяет получить набор ожидаемых лексем в каждом состоянии. Поэтому система автодополнения имеет зависимости от объектов управления автомата, хранящих необходимые данные.

В разд. 3.8. приведен пример работы рассмотренных функций.

3.3. Диаграмма связей

Как отмечалось выше, проектирование программы выполнено с помощью инструментального средства *UniMod*, которое позволяет построить диаграмму классов программы (схему связей), а также автоматы, описывающие поведение системы.

При этом схема связей является диаграммой классов *UML*, однако классы располагаются не сверху вниз, а слева направо, так же, как обычно изображаются автоматизированные объекты управления [20]. Диаграмма связей

для решаемой задачи изображена в приложении 2. На этой схеме слева направо изображены: поставщик событий, автомат и объекты управления. В приложении 3 приведен граф переходов автомата *FSML*, в приложениях 4 и 5 – примеры работы, а в приложении 6 – исходные коды программы.

3.4. Поставщик событий

В этом разделе описывается поставщик событий (*Event provider*) *p1:FSMLLexer*. В приложении 6.1 приведен его исходный код. Этот поставщик реализует события в виде лексем, полученных после лексического анализа входной *fsml*-программы. Разбиение входного текста на лексемы производится регулярным выражением.

События, генерируемые поставщиком событий *FSMLLexer*:

- `e1` – на входе нераспознанная лексема;
- `e2` – на входе идентификатор типа;
- `e3` – на входе константа типа *boolean*;
- `e4` – на входе константа в виде целого числа;
- `e5` – на входе идентификатор;
- `e6` – на входе символы логического «и» '&&';
- `e7` – на входе символы логического «или» '||';
- `e8` – на входе символы отношения;
- `e9` – на входе символ конца ввода *EOF*;
- `e10` – на входе символ точки;
- `e11` – на входе символ логического отрицания '!';
- `e12` – на входе символ запятой;
- `e13` – на входе символ точки с запятой;
- `e14` – на входе символ открывающей фигурной скобки;
- `e15` – на входе символ закрывающей фигурной скобки;
- `e16` – на входе символ открывающей круглой скобки;

- `e17` – на входе символ закрывающей круглой скобки;
- `e18` – на входе тип состояния *initial*;
- `e19` – на входе тип состояния *final*;
- `e20` – на входе ключевое слово *execute*;
- `e21` – на входе ключевое слово *transitto*;
- `e22` – на входе ключевое слово *on*;
- `e23` – на входе ключевое слово *statemachine*;
- `e24` – на входе ключевое слово *if*;
- `e25` – на входе ключевое слово *else*;
- `e26` – на входе тип события *enter*;
- `e27` – на входе тип события *any*;
- `e28` – на входе тип события *exit*;
- `e29` – на входе ключевое слово *uses*;
- `e30` – на входе ключевое слово *include*;
- `e31` – на входе пробельные символы;
- `e32` – на входе комментарий.

3.5. Объекты управления

В данном разделе описываются объекты управления (*Controlled Objects*). В приложении 6.2 приведены соответствующие исходные коды.

Объект управления *o1:Stack*

Этот объект реализует стек для синтаксического анализатора. Он необходим для корректной обработки автоматом, например, вложенных скобочных выражений (в дальнейшем – вопросы вложенности). Методы, предоставленные объектом управления *Stack*, делятся на входные переменные:

- `x1` – стек пуст;
- `x2` – на вершине круглая скобка;
- `x3` – на вершине фигурная скобка;

- `x4` – на вершине символ, обозначающий начало группы состояний;

и выходные воздействия:

- `z1` – снять один элемент с вершины стека;
- `z2` – положить в стек круглую скобку;
- `z3` – положить в стек фигурную скобку;
- `z4` – положить в стек символ, обозначающий начало группы состояний.

Объект управления *o2:FSMLLexer*

Этот объект необходим для системы автоматического завершения ввода. Он предоставляет входную переменную `x1` – число лексем в потоке, неподходящих в данном состоянии.

Объект управления *o3:FSMLToModel*

Этот объект реализует генератор объектной модели автомата. Он создает соответствующие объекты этой модели. Методы, предоставленные объектом управления *FSMLToModel*, делятся на входные переменные:

- `x1` – текущее состояние является начальным;
- `x2` – текущее состояние является конечным;
- `x3` – текущее состояние является обычным (используется для краткости записи вместо одновременного использования двух предыдущих);

и выходные воздействия:

- `z1` – создать объект конечного автомата;
- `z2` – сохранить вложенный автомат в списке;
- `z3` – сохранить тип состояния;
- `z4` – создать объект состояния;
- `z5` – сохранить часть сторожевого условия;
- `z6` – создать переходы и действия в автомате;
- `z7` – добавить событие;
- `z8` – создать объект сторожевого условия;

- z9 – сохранить тип события;
- z10 – сохранить часть идентификатора действия;
- z11 – добавить действие;
- z12 – сохранить имя целевого состояния;
- z13 – сохранить часть имени класса поставщика событий;
- z14 – добавить поставщик событий;
- z15 – сохранить часть имени класса объекта управления;
- z16 – добавить объект управления;
- z17 – добавить список вложенных автоматов;
- z18 – перейти в суперсостояние.

Объект управления *o4:ObjectContext*

Этот объект представляет собой контекст синтаксического анализатора – хранит информацию о последних обработанных лексемах. Он используется системой автоматического завершения ввода. Объект управления *ObjectContext* предоставляет следующие методы с выходными воздействиями:

- z1 – сохранить последнюю лексему в стек;
- z2 – очистить стек;
- z3 – установить контекст по умолчанию;
- z4 – установить контекст события;
- z5 – установить контекст сторожевого условия;
- z6 – установить контекст действия;
- z7 – установить контекст поставщика событий;
- Z8 – установить контекст объект управления;
- Z9 – установить контекст состояния;
- Z10 – отметить последний идентификатор как завершенный.

Объект управления *o5:StreamErrorListener*

Это вспомогательный объект управления, он реализует обработчик ошибок в процессе работы синтаксического анализатора. Объект управления *StreamErrorListener* предоставляет следующие методы с выходными воздействиями:

- `z1` – лексема была пропущена;
- `z2` – лексемы были добавлены.

3.6. Автомат *FSML*

В этом разделе описывается используемый в работе автомат.

3.6.1. Описание

Автомат *FSML* реализует синтаксический анализатор – принимает поступающие лексемы от поставщика событий *p1:FSMLLexer* и осуществляет синтаксически-семантический анализ программы. Кроме того, автомат дополняется всеми недостающими переходами с помощью алгоритма создания системы автоматического завершения ввода, предложенного в работе [19].

3.6.2. Принцип работы

Получая последовательно лексемы в виде событий, синтаксический анализатор в зависимости от состояния формирует команды управления генератору объектной модели автомата *FSMLToModel*. Для разрешения вопросов вложенности используется объект управления *Stack*. Контекстная информация, необходимая алгоритму автоматического завершения ввода, обеспечивается объектом управления *ObjectContext*.

3.6.3. Состояния

Состояния автомата *FSML* разбиты на логические группы, в которых происходит семантический анализ какой-либо части *fsml*-программы. В квадратных скобках указан диапазон порядковых номеров состояний в группе.

1. `SM[1-4]` – объявление автомата в программе;

2. `EP[1-2]` – объявление поставщика событий автомата;
3. `CO[1-3]` – объявление объекта управления автомата;
4. `State[1-6]` – объявление состояния автомата;
5. `Submachine[1-2]` – объявление вложенного автомата;
6. `Transition[1-3]` – суперсостояния, объединяющие циклы обработки перехода;
7. `Event[1-5]` – объявление событий для перехода;
8. `Guard[1-12]` – объявление сторожевого условия для перехода;
9. `Action[1-4]` – объявление списка действий, которые требуется совершить для описанных событий;
10. `TargetState[1-2]` – объявление целевого состояния перехода;
11. `s[1-2]` – начальное и конечное состояния автомата.

3.6.4. Граф переходов

Граф переходов автомата *FSML* представлен в приложении 3.

3.7. Структура проекта

В этом разделе рассмотрена файловая структура проекта. Проект реализован на языке *Java* и содержит следующие пакеты:

- `com.unimod.fsml` – пакет, содержащий общие данные языка *FSML*:
  - `FSMLTokens.java` – интерфейс с ключевыми словами языка и соответствующими событиями автомата;
- `com.unimod.fsml.model` – пакет, содержащий модель анализатора языка *FSML* (автомат, поставщик событий, объекты управления и вспомогательные классы):
  - `ObjectContext.java` – объект управления, реализующий контекст синтаксического анализатора;
  - `Stack.java` – объект управления, реализующий стек;

  - `StreamErrorListener.java` – объект управления, обрабатывающий ошибки;
  - `ParserCOMap.java` – класс, хранящий объекты управления;
  - `FSMLAutoCompletion.java` – класс, реализующий систему автоматического завершения ввода и исправления ошибок;
  - `UnhandledTokensResolver.java` – класс, реализующий алгоритм дополнения автомата недостающими переходами [19];
  - `FSMLModelHelper.java` – класс, предоставляющий интерфейс работы с автоматной моделью;
- `com.unimod.fsml.model.lexer` – пакет, содержащий классы, необходимые для лексического анализатора:
  - `FSMLLexer.java` – лексический анализатор, является поставщиком событий и объектом управления;
  - `LexerParameters.java` – интерфейс с названиями параметров;
  - `Token.java` – класс лексемы;
- `com.unimod.fsml.model.transform` – классы, реализующие преобразование между объектной автоматной моделью и *fsml*-программами:
  - `FSMLToModel.java` – объект управления, реализующий генератор объектной модели автомата;
  - `UnfinishedTransaction.java` – класс, хранящий неполные данные перехода в автомате;
- `com.unimod.fsml.validation` – пакет, содержащий систему валидации *fsml*-программы (классы валидатора, ошибок валидации и вспомогательные);
- `com.unimod.fsml.util` – пакет с утилитными классами;

- `com.unimod.fsmleditor` – главный пакет редактора языка *FSML*, платформозависимая часть.

3.8. Пример работы редактора *FSML*

В данном разделе рассмотрим функции редактора языка *FSML*, перечисленные в разд. 3.2, на примере из разд. 2.1. Этот пример реализует модель пешеходного светофора с таймером. На данный момент редактор устроен так, что его работа состоит из двух этапов: написание или редактирование программы на языке *FSML* и генерация автоматной модели *UniMod* по этой программе. Далее опишем каждый из этих этапов.

3.8.1. Редактирование программы на языке *FSML*

На этапе редактирования используются две из рассмотренных функций: валидация программы и автоматическое завершение ввода. В приложении 4 приведено изображение среды разработки *Eclipse* с вышеупомянутым примером, открытым в редакторе языка *FSML*. Редактор подсвечивает ошибку валидации программы и предоставляет набор лексем для устранения найденной ошибки в строке:

```
on stop <положение курсора> Final;
```

Валидация запускается автоматически при сохранении текста программы. При этом происходит разбор программы: разбиение ее на лексемы и синтаксический анализ, выполняемый автоматом *FSML*. Перед обработкой очередной лексемы выполняется проверка ее валидности в текущем состоянии автомата *FSML*. Лексема является валидной, если из текущего состояния существует переход по ней с выполненным сторожевым условием и не являющийся обработчиком ошибки (такие переходы обозначаются меткой `error`).

Система автоматического завершения ввода запускается пользователем. По алгоритму работы она схожа с системой валидации, но синтаксический анализ программы выполняется до места, в котором требуется автоматическое

завершение ввода. В качестве вариантов завершения ввода предлагаются все валидные лексемы в текущем состоянии автомата *FSML*.

Проанализируем предлагаемый набор лексем в рассмотренном выше случае. После обработки начала соответствующей строки до места положения курсора автомат *FSML* находится в состоянии `Event4` (Приложение 3). Из этого состояния существуют переходы в следующие состояния:

- `TargetState1` по событию `e21` (лексема `'transitto'`);
- `Action1` по событию `e20` (лексема `'execute'`);
- `Guard1` по событию `e25` (лексема `'else'`);
- `Guard2` по событию `e24` (лексема `'if'`);
- `Event5` по событию `e12` (лексема `','`);
- `State6` по событию `e13` (лексема `';'`).

Этот список лексем и предлагается редактором для исправления найденной ошибки, так как все сторожевые условия на этих переходах выполняются. Заметим, что на всех этих переходах находятся константные лексемы. В случае наличия перехода по событию `e5` (на входе идентификатор) задача усложняется необходимостью выборки соответствующих значений. Для этого используется объект управления *o4:ObjectContext*, предоставляющий контекстную информацию.

3.8.2. Генерация автоматной модели *UniMod*

На данном этапе происходит разбор программы на языке *FSML* и построение автоматной модели с помощью рассмотренных в разд. 3.5 объектов управления. Так как объекты управления подробно описаны выше, рассмотрим тут только краткий пример их работы – в процессе разбора первой строки программы из примера:

```
uses trafficlight.provider.TrafficSignalProvider;
```

Опишем каждый шаг этого процесса:

1. Автомат *FSML* совершает безусловный переход из начального состояния `s1` в состояние `SM1` – начало объявления конечного автомата.

2. Переход в состояние `EP1` по событию `e29` (лексема `'uses'`). При этом выполняется действие `o4.z7` – объект управления *o4:ObjectContext* устанавливает контекст поставщика событий. Это позволяет автомату и системе автодополнения определять семантику последующих идентификаторов.

3. Переход в состояние `EP2` по событию `e5` (идентификатор `'trafficlight'`). При этом выполняется действие `o3.z13` – объект управления *o3:FSMLToModel* сохраняет часть имени класса поставщика событий. При входе в состояние `EP2` выполняется действие `o4.z1` – идентификатор сохраняется в стеке объекта управления *o4:ObjectContext*. Таким образом, *o3:FSMLToModel* формирует имя класса поставщика событий для его последующего сохранения в автоматной модели, а *o4:ObjectContext* запоминает информацию для системы автодополнения.

4. Переход в состояние `EP1` по событию `e10` (лексема `'.'`). При этом снова выполняется действие `o3.z13`. Также выполняется действие `o4.z10` – последний идентификатор отмечается как завершенный, что сообщает системе автодополнения о необходимости загружать названия подпакетов следующего уровня.

5. Повторно выполняются шаги 3 – 4 – 3 данного процесса, в результате чего объект управления *o3:FSMLToModel* сохраняет полное имя класса поставщика событий.

6. Переход в состояние `SM1` по событию `e13` (лексема `';'`). При этом выполняется действие `o3.z14` – объект управления *o3:FSMLToModel* добавляет поставщика событий в автоматную модель.

Таким способом происходит разбор всей программы на языке *FSML* и построение автоматной модели *UniMod*. Затем к построенной модели автоматически применяется встроенный в инструментальное средство *UniMod*

компоновщик, который производит укладку автоматной модели на плоскости в виде диаграмм связей и состояний. Сгенерированные им диаграммы для примера приведены в приложении 1. В приложении 5 показан способ запуска генерации автоматной модели через контекстное меню редактора.

3.9. Использование

Результатом данной работы является встраиваемый модуль (plug-in) для среды разработки *Eclipse*. Для работы модуля необходимо установить инструментальное средство *UniMod*, являющееся модулем к той же среде. При тестировании были использованы версии *Eclipse 3.3.0* и *UniMod 1.3.39*. Версия *UniMod 1.3.39* содержит критические изменения, необходимые для корректной работы редактора *FSML*.

Руководство по установке *Eclipse* и *UniMod* можно найти на сайте проекта *UniMod* http://unimod.sourceforge.net. Для установки модуля *FSMLEditor* распакуйте приложенный к данной работе архив **com.unimod.fsmleditor_1.0.0.zip** в директорию Eclipse/plugins и перезапустите *Eclipse.*

Выводы по главе 3

1. Приведен обзор возможностей созданного редактора.
2. Редактор реализован на основе автоматного подхода с использованием инструментального средства *UniMod*.
3. Описаны принятые проектные решения и приведена проектная документация.
4. Приведен пример и порядок использования реализованного инструментального средства.

ЗАКЛЮЧЕНИЕ

Итак, в ходе работы выполнен обзор существующих языков и средств автоматного программирования, рассмотрены основные подходы к построению синтаксического анализатора языка автоматного программирования. В результате были сформулированы требования к разработанному языку и инструментальному средству. **Для реализации поставленной задачи выбран автоматный подход**, имеющий явные преимущества в удобстве и эффективности использования перед традиционными способами нисходящего разбора, а именно:

- естественное представление анализатора языка в виде событийной системы со сложным поведением;
- удобство реализации вспомогательных задач, таких как реализация системы автоматического завершения ввода.

Автоматный подход при создании программного обеспечения реальных систем и их моделей помогает существенно облегчить процесс проектирования, отладки и модификации программного кода. Явное выделение состояний делает логику программы более простой и прозрачной для понимания, что в совокупности с протоколированием работы автоматов позволяет разработчику успешно следить за поведением программы, как в период разработки, так и во время сопровождения программного продукта.

В ходе данной работы создан текстовый язык *FSML* для представления конечных автоматов и реализован редактор этого языка *FSMLEditor*, предназначенный для использования совместно с инструментальным средством *UniMod*. В совокупности эти две компоненты имеют весомые преимущества перед существующими средствами автоматного программирования:

- текстово-визуальный подход к описанию конечных автоматов;
- естественное сочетание автоматного и объектно-ориентированного подходов при разработке программ;

- современные средства для работы с кодом, такие как подсветка ошибок и автоматическое завершение ввода.

Отметим, что текстовый редактор *FSMLEditor*, являющийся результатом этой работы, реализован с использованием инструментального средства *UniMod*. Таким образом, на основе автоматного подхода создан эффективный инструмент автоматного программирования.

СПИСОК ИСТОЧНИКОВ

1. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1

2. *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

3. *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.pdf

4. *Среда разработки Eclipse*. http://www.eclipse.org

5. *Язык AsmL*. http://research.microsoft.com/fse/asml

6. *Язык SMC*. http://smc.sf.net

7. *Язык FSMGenerator*. http://fsmgenerator.sf.net

8. *Шамгунов Н. Н.* Разработка методов проектирования и реализации поведения программных систем на основе автоматного подхода. Диссертация на соискание ученой степени кандидата технических наук. СПб.: СПбГУ ИТМО. 2004. http://is.ifmo.ru/disser/shamg_disser.pdf

9. *Цымбалюк Е. А.* Текстовый язык автоматного программирования. Магистерская диссертация. СПб.: СПбГУ ИТМО. 2008.

10. *Степанов О. Г., Шалыто А. А., Шопырин Д. Г.* Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby //Информационно-управляющие системы. 2007. № 4, с. 22–27. http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf

11. *Язык Ruby*. http://ru.wikipedia.org/wiki/Ruby

12. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования // Тезисы докладов международной научной конференции, посвященной памяти профессора А. М. Богомолова «Компьютерные науки и технологии». Саратов: СГУ. 2007. с. 66–69.

http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf

13. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс. 2003.

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

15. *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

16. *Дмитриев С.* Языково-ориентированное программирование: следующая парадигма // RSDN Magazine. 2005. № 5.
http://www.rsdn.ru/article/philosophy/LOP.xml

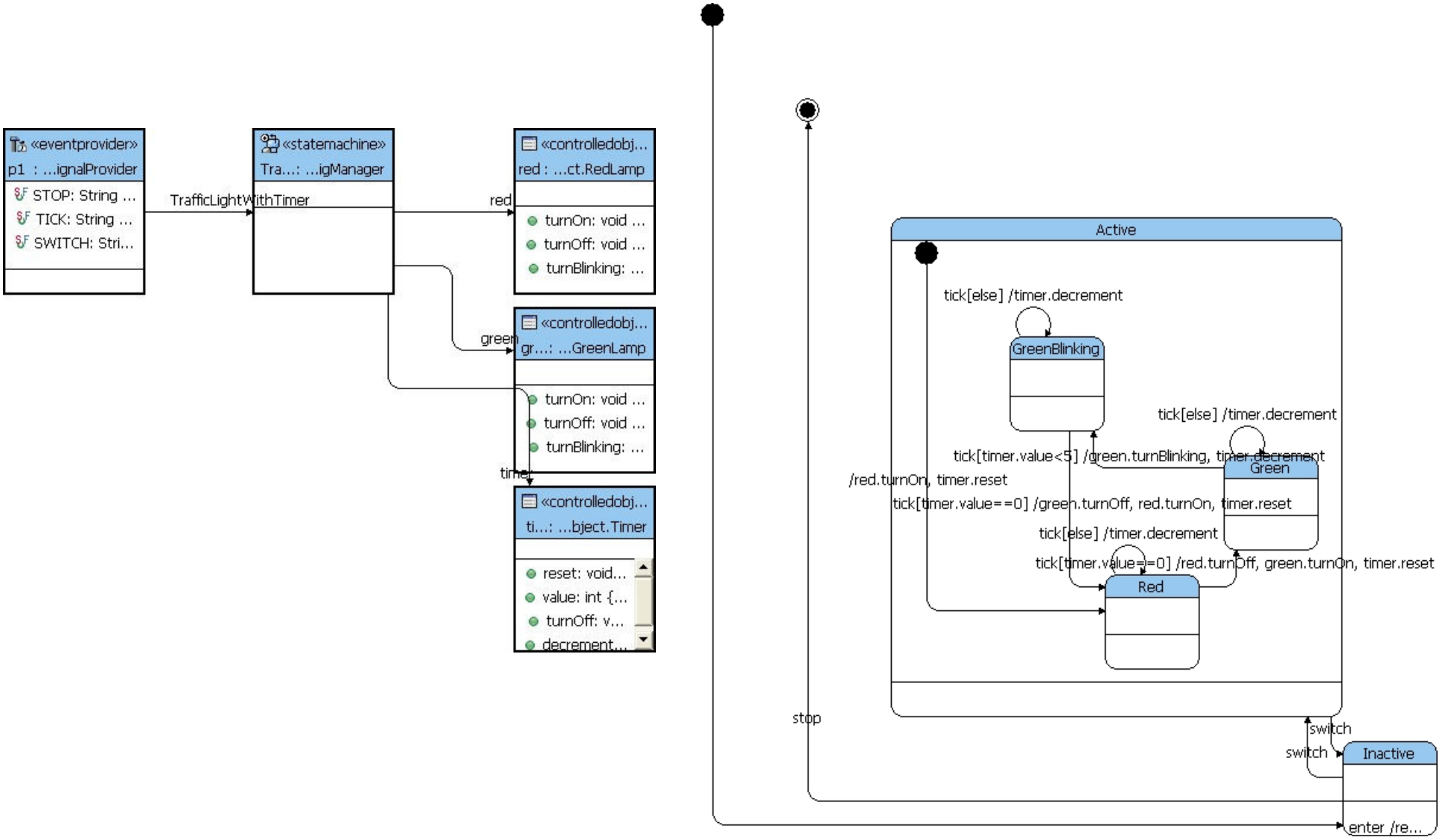
17. *Parr T. J., Quong R. W.* ANTRL: A Predicated-LL(k) Parser Generator // Software – Practice and Experience. 1995. № 25(7), pp. 789–810.

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

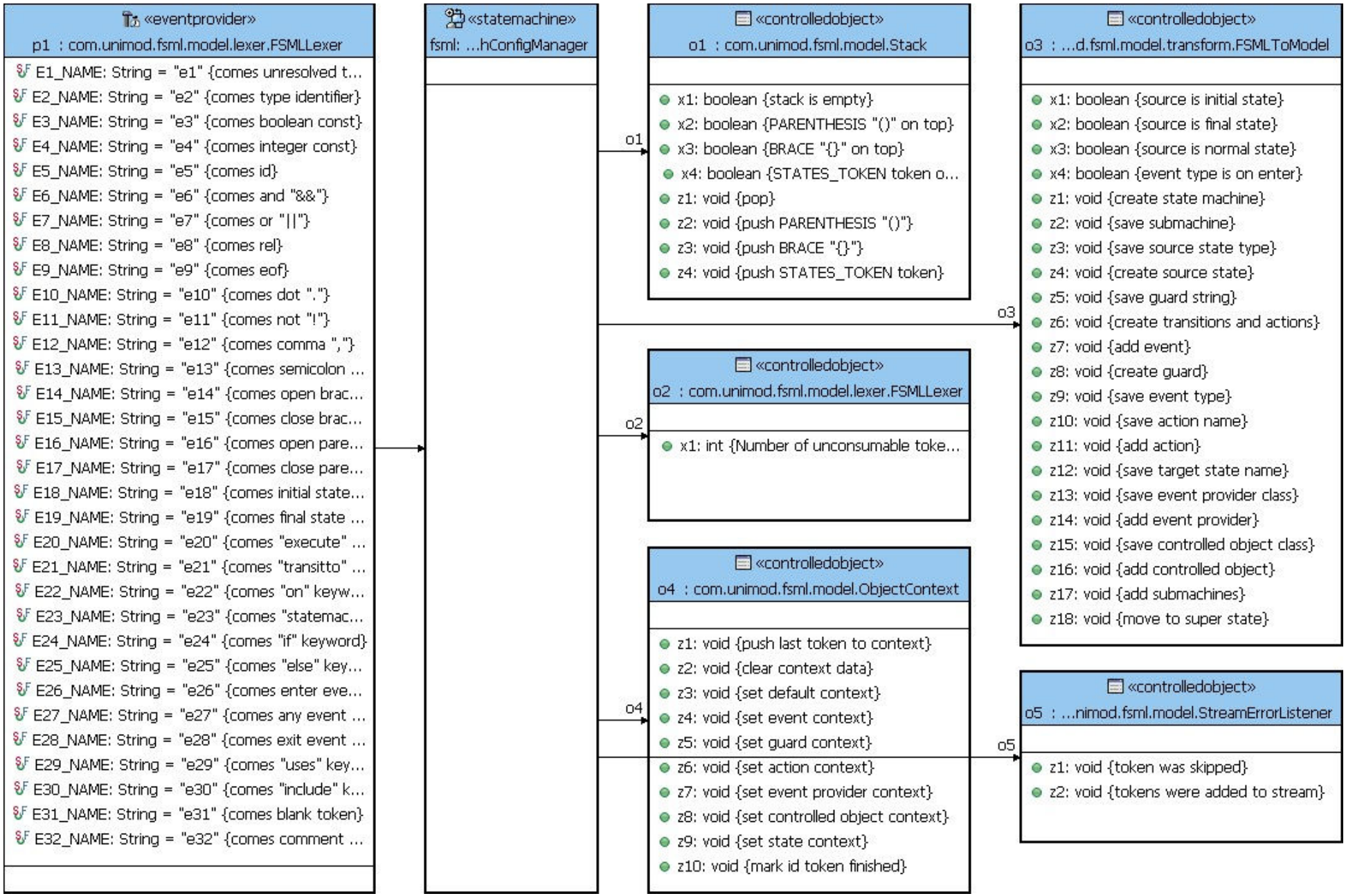
19. *Гуров В. С., Мазин М. А.* Создание системы автоматического завершения ввода с использованием пакета UniMod // Вестник II Межвузовской конференции молодых ученых. Т.1. СПб.: СПбГУ ИТМО. 2005, с. 73–87.

20. *Switch-технология*. http://ru.wikipedia.org/wiki/Switch-технология

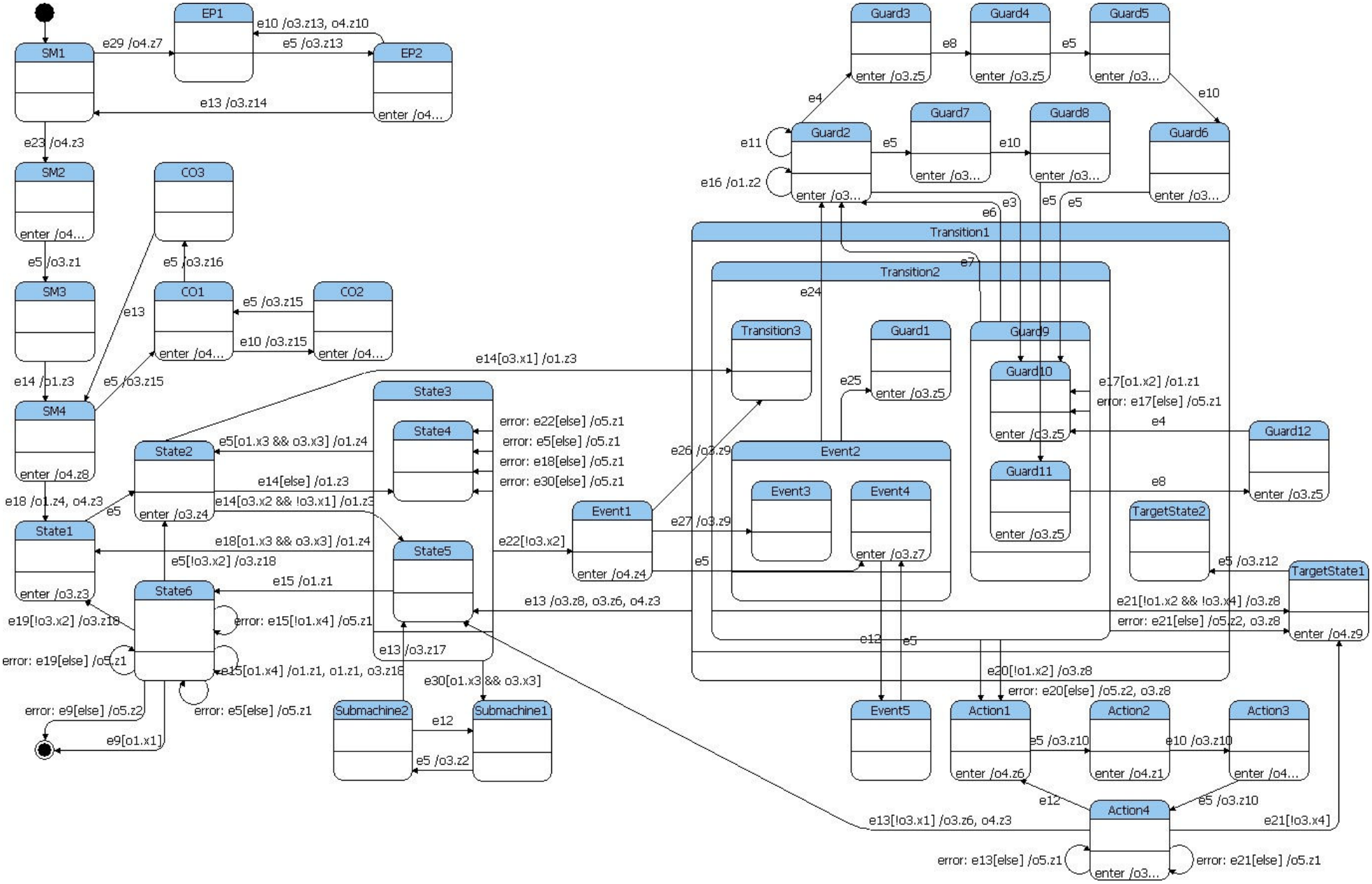
ПРИЛОЖЕНИЕ 1. Автоматически сгенерированные диаграммы для примера



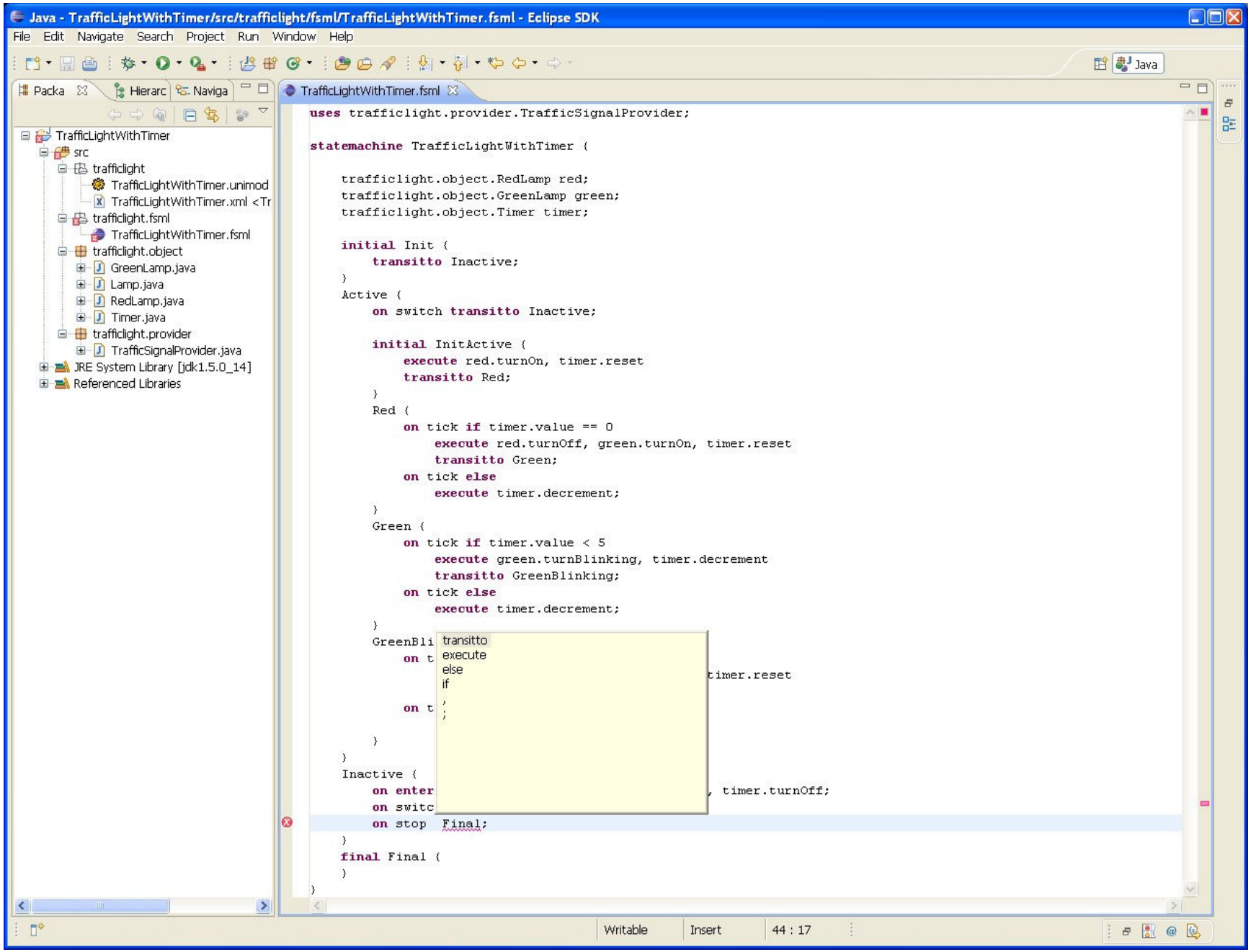
ПРИЛОЖЕНИЕ 2. Диаграмма связей анализатора языка *FSML*



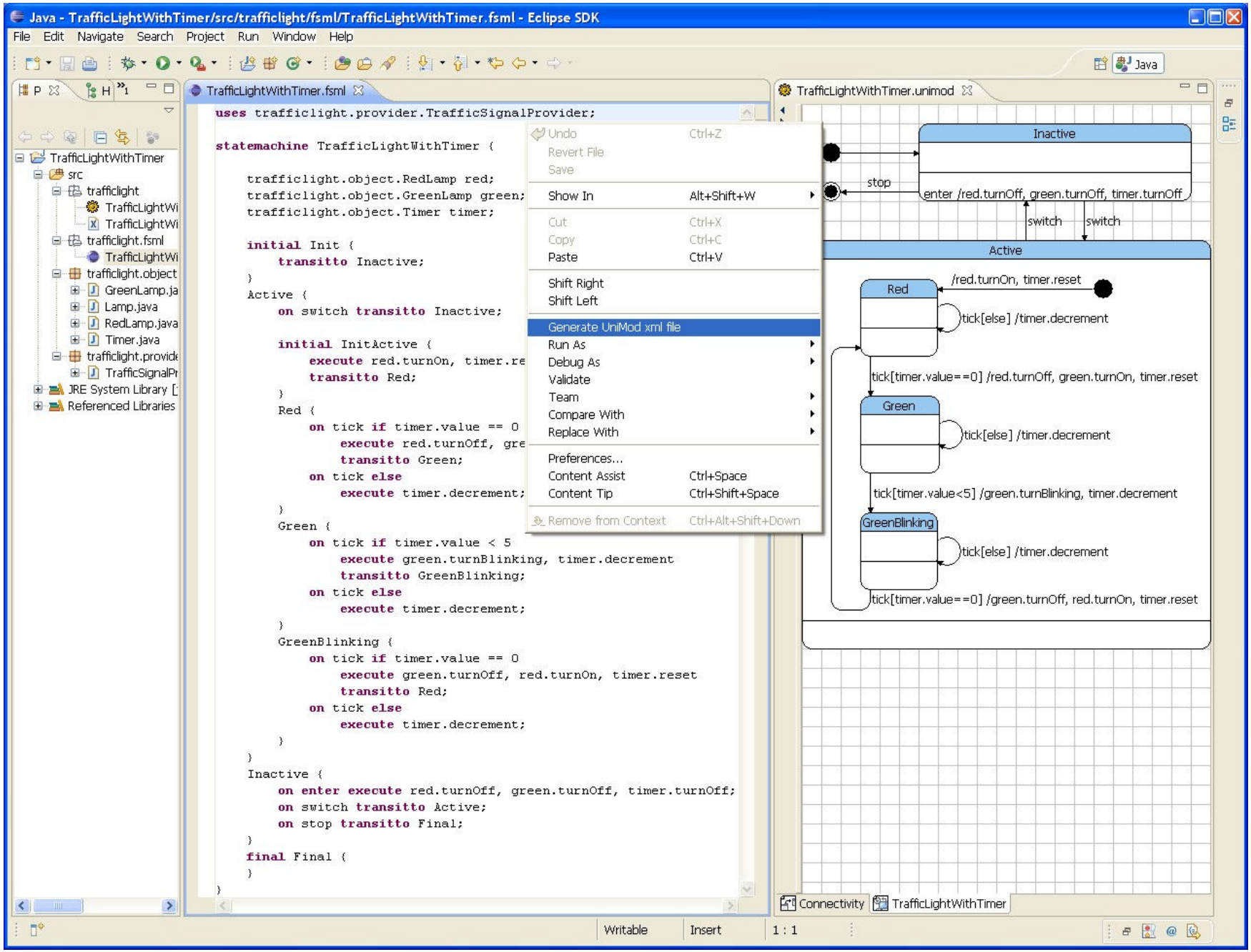
ПРИЛОЖЕНИЕ 3. Граф переходов автомата *FSML*



ПРИЛОЖЕНИЕ 4. Пример работы – редактирование



ПРИЛОЖЕНИЕ 5. Пример работы – генерация



ПРИЛОЖЕНИЕ 6. Исходные коды программы

6.1. Поставщик событий `FSMLLexer.java`

```
1    package com.unimod.fsml.model.lexer;
2
3    import java.util.ArrayList;
4    import java.util.Arrays;
5    import java.util.HashSet;
6    import java.util.Iterator;
7    import java.util.List;
8    import java.util.Set;
9    import java.util.regex.Matcher;
10   import java.util.regex.Pattern;
11
12   import org.apache.commons.lang.StringUtils;
13   import org.apache.commons.logging.Log;
14   import org.apache.commons.logging.LogFactory;
15
16   import com.evelopers.unimod.contract.CoreContract;
17   import com.evelopers.unimod.core.stateworks.Event;
18   import com.evelopers.unimod.core.stateworks.Guard;
19   import com.evelopers.unimod.core.stateworks.State;
20   import com.evelopers.unimod.core.stateworks.StateMachine;
21   import com.evelopers.unimod.core.stateworks.Transition;
22   import com.evelopers.unimod.parser.InterpreterException;
23   import com.evelopers.unimod.runtime.ControlledObject;
24   import com.evelopers.unimod.runtime.EventProcessorException;
25   import com.evelopers.unimod.runtime.EventProvider;
26   import com.evelopers.unimod.runtime.ModelEngine;
27   import com.evelopers.unimod.runtime.StateMachineConfig;
28   import com.evelopers.unimod.runtime.StateMachineConfigManager;
29   import com.evelopers.unimod.runtime.StateMachinePath;
30   import com.evelopers.unimod.runtime.context.Parameter;
31   import com.evelopers.unimod.runtime.context.StateMachineContext;
32   import com.evelopers.unimod.runtime.interpretation.ActionExecutor;
33   import com.evelopers.unimod.runtime.interpretation.InterpretationEventProcessor;
34   import
com.evelopers.unimod.runtime.interpretation.InterpretationEventProcessor.InterpretationInputActions;
35   import com.unimod.fsml.FSMLTokens;
36   import com.unimod.fsml.model.FSMLModelHelper;
37   import com.unimod.fsml.model.UnhandledTokensResolver;
38   import com.unimod.fsml.validation.ProblemToken;
39
40   /**
41    * This class represents lexer for FSML.
42    *
43    * @author Ivan Lagunov
44    */
45   public class FSMLLexer implements FSMLTokens, EventProvider, ControlledObject {
46       private static final Log log = LogFactory.getLog(FSMLLexer.class);
47
48       private StateMachineContext context;
49
50       private ModelEngine engine;
51       private StateMachine machine;
52       private List<Token> tokens;
53       private Iterator<Token> tokenIterator;
54       private Token prevToken;
55       private Token nextToken;
56       private State currentState;
57
58       /* Data for autocompletion */
59       private int offset;
60
61       /* Data for validation */
62       private List<ProblemToken> problemTokens;
63
64       /**
65        * This event provider is designed to be used with build in approach
66        * so it has no default constructor.
67        */
68       public FSMLLexer(StateMachine machine, StateMachineContext context) {
69           this.machine = machine;
```

```
70          this.context = context;
71      }
72
73      /* (non-Javadoc)
74       * @see
com.evelopers.unimod.runtime.EventProvider#init(com.evelopers.unimod.runtime.ModelEngine)
75       */
76      public void init(ModelEngine engine) {
77          this.engine = engine;
78      }
79
80      /* (non-Javadoc)
81       * @see com.evelopers.unimod.runtime.EventProvider#dispose()
82       */
83      public void dispose() {
84      }
85
86      public void initData(String expr, int offset) {
87          tokens = new ArrayList<Token>();
88          problemTokens = new ArrayList<ProblemToken>();
89          parse(expr);
90          tokenIterator = tokens.iterator();
91          this.offset = offset;
92      }
93
94      /**
95       * Returns the previous token before <code>nextToken</code>.
96       *
97       * @return the previous token.
98       */
99      public Token getPrevToken() {
100         return prevToken;
101     }
102
103     /**
104      * Returns the next token after those that were processed before offset.
105      *
106      * @return the next token.
107      */
108     public Token getNextToken() {
109         return nextToken;
110     }
111
112     /**
113      * Returns an iterator over the collection of found problem tokens.
114      *
115      * @return an iterator over the collection of found problem tokens.
116      */
117     public Iterator<ProblemToken> getProblemTokenIterator() {
118         return problemTokens.iterator();
119     }
120
121     /**
122      * @unimod.action.descr Number of unconsumable tokens
123      */
124     public int x1(StateMachineContext context) throws EventProcessorException,
InterpreterException {
125         Set<Integer> consumableTokens = getConsumableTokens();
126         int unconsumableTokensNumber = 1;
127         // TODO: check if tokenIterator is correct.
128         for (Iterator<Token> i = tokenIterator; i.hasNext();) {
129             Token token = i.next();
130             if (!consumableTokens.contains(token.getTypeId())) {
131                 unconsumableTokensNumber++;
132             } else {
133                 break;
134             }
135         }
136         return unconsumableTokensNumber;
137     }
138
139     /**
140      * Returns set of consumable tokens in current state.
141      *
142      * @return the set of consumable tokens.
143      * @throws EventProcessorException
144      * @throws InterpreterException
145      */
```

```
146      public Set<Integer> getConsumableTokens() throws EventProcessorException,
InterpreterException {
147          Set<Integer> consumableTokens = new HashSet<Integer>();
148          for (State cur = getActiveState(context); cur != null; cur = cur.getSuperstate()) {
149              for (Object o : cur.getOutgoingTransitions()) {
150                  Transition transition = (Transition) o;
151                  if (!isErroneousTransition(transition) && guardConditionIsMet(transition)) {
152                      consumableTokens.add(Events.getIdByName(transition.getEvent().getName()));
153                  }
154              }
155          }
156          return consumableTokens;
157      }
158
159      /**
160       * Checks if last token was valid according to state machine or not.
161       * If not valid, it should be changed.
162       *
163       * @param lastTokenType the type of last token.
164       * @return true if last token is valid.
165       */
166      public boolean isValidLastToken(String lastTokenType) {
167          Transition elseTransition = null;
168          try {
169              State prevState = currentState;
170              while ((prevState != null) && (null == elseTransition)) {
171                  for (Object o : prevState.getOutgoingTransitions()) {
172                      Transition transition = (Transition) o;
173                      if (lastTokenType.equals(transition.getEvent().getName())) {
174                          if (transition.getGuard().equals(Guard.ELSE)) {
175                              elseTransition = transition;
176                          } else if (guardConditionIsMet(transition)) {
177                              return !isErroneousTransition(transition);
178                          }
179                      }
180                  }
181
182                  prevState = prevState.getSuperstate();
183              }
184          } catch (InterpreterException e) {
185              log.error("Cannot check guard condition", e);
186          }
187          return (elseTransition != null) ? !isErroneousTransition(elseTransition) : false;
188      }
189
190      private boolean isErroneousTransition(Transition transition) {
191          return UnhandledTokensResolver.ERROR_TRANSITION_NAME.equals(transition.getName());
192      }
193
194      private boolean guardConditionIsMet(Transition transition) throws InterpreterException {
195          FSMLModelHelper modelHelper = FSMLModelHelper.getInstance();
196          InterpretationEventProcessor interpretationEventProcessor = new
InterpretationEventProcessor(modelHelper.getProgramModel());
197          // Prepare input actions pool
198          ActionExecutor actionExecutor = new ActionExecutor(modelHelper.getCOMap());
199          StateMachinePath path = getStateMachinePath();
200          InterpretationInputActions inputActions = interpretationEventProcessor.new
InterpretationInputActions(actionExecutor, path);
201
202          return interpretationEventProcessor.guardConditionIsMet(transition, context,
inputActions);
203      }
204
205      /**
206       * Returns token from the stream.
207       * If index equals to 0 then last token will be returned,
208       * if index equals to 1 then last but one token will be returned,
209       * etc.
210       *
211       * @param index index from the end of stream.
212       * @return peeked token.
213       */
214      public Token peekToken(int index) {
215          return 0 <= index && index < tokens.size() ? (Token) tokens.get(tokens.size() - (index
+ 1)) : null;
216      }
217
218      /**
```

```
219         * Processes tokens before the offset specified during <code>initData</code> call.
220         *
221         * @return whether there is any token left to process.
222         */
223        public boolean processTokensBeforeOffset() {
224            try {
225                currentState = getActiveState(context);
226            } catch (EventProcessorException e) {
227                log.error("Cannot load active state", e);
228            }
229
230            while (hasNextTokenBeforeOffset()) {
231                tick();
232            }
233            return (null == nextToken);
234        }
235
236        /**
237         * Processes the rest of tokens.
238         */
239        public void processTokensAfterOffset() {
240            if (nextToken != null) {
241                do {
242                    tick();
243                } while (hasNextTokenAfterOffset());
244            }
245        }
246
247        /**
248         * Checks if there is a token strictly before given offset and saves it.
249         *
250         * @return true - if next token exists, false - otherwise.
251         */
252        private boolean hasNextTokenBeforeOffset() {
253            prevToken = nextToken;
254            if (tokenIterator.hasNext()) {
255                nextToken = tokenIterator.next();
256
257                return (nextToken.getEnd() < offset) ||
258                    ((nextToken.getEnd() == offset) && (!nextToken.isSpaceFollowed()));
259            } else {
260                nextToken = null;
261                return false;
262            }
263        }
264
265        /**
266         * Checks if there is a token left and saves it.
267         *
268         * @return true - if next token exists, false - otherwise.
269         */
270        private boolean hasNextTokenAfterOffset() {
271            prevToken = nextToken;
272            if (tokenIterator.hasNext()) {
273                nextToken = tokenIterator.next();
274                return true;
275            } else {
276                nextToken = null;
277                return false;
278            }
279        }
280
281        private void tick() {
282            Parameter ptoken = new Parameter(LexerParameters.TOKEN, nextToken);
283            Parameter expr = new Parameter(LexerParameters.EXPRESSION, nextToken.getExpr());
284            Parameter parsedExpr = new Parameter(LexerParameters.PARSED_EXPRESSION,
nextToken.getParsedExpr());
285            Parameter tokenType = new Parameter(LexerParameters.TOKEN_TYPE, nextToken.getType());
286            Parameter tokenValue = new Parameter(LexerParameters.TOKEN_VALUE,
nextToken.getValue());
287
288            Event token = new Event(nextToken.getType(), new Parameter[] {
289                    ptoken, expr, parsedExpr, tokenType, tokenValue});
290
291            checkForProblem(nextToken);
292
293            engine.getEventManager().handle(token, context);
294
```

```
295           try {
296               currentState = getActiveState(context);
297           } catch (EventProcessorException e) {
298               log.error("Cannot load active state", e);
299           }
300       }
301
302       private void checkForProblem(Token lastToken) {
303           if (!isValidLastToken(lastToken.getType())) {
304               problemTokens.add(new ProblemToken(lastToken));
305           }
306       }
307
308       /**
309        * Returns active state by given {@link StateMachineContext context}.
310        *
311        * @param context state machine context.
312        * @return the active state.
313        * @throws EventProcessorException
314        */
315       private State getActiveState(StateMachineContext context) throws EventProcessorException {
316           StateMachinePath path = getStateMachinePath();
317           StateMachineConfigManager configManager =
engine.getEventProcessor().getModelStructure().getConfigManager(machine.getName());
318           StateMachineConfig config = configManager.load(path, context);
319
320           return CoreContract.decodeState(machine, config.getActiveState());
321       }
322
323       private StateMachinePath getStateMachinePath() {
324           return new StateMachinePath(machine.getName());
325       }
326
327       private static Pattern TOKEN_PATTERN = Pattern.compile("(-?\\d+)|([_A-Za-
z]\\w*)|(&&)|(\\|\\|)|(([><]=?)|([!=]=))|(\\.)|(\\!)|(\\,)|(\\;)|(\\{)|(\\})|(\\()|(\\))|(\\s+)|(//.
*?$|/\\*.*?\\*/)", Pattern.DOTALL | Pattern.MULTILINE);
328
329       private void parse(String expr) {
330           Matcher matcher = TOKEN_PATTERN.matcher(expr);
331           int pos = 0;
332           int lineNumber = 0;
333           int nextNewLineCharPos = -1;
334           String type = null;
335
336           while (matcher.find()) {
337               type = null;
338
339               if (matcher.group(1) != null) {
340                   type = E4_NAME;
341               } else if (matcher.group(2) != null) {
342                   String value = matcher.group();
343                   if (Arrays.asList(BOOL_CONSTS).contains(value)) {
344                       type = E3_NAME;
345                   } else if (EVENT_TYPE.ENTER.equals(value)) {
346                       type = E26_NAME;
347                   } else if (EVENT_TYPE.ANY.equals(value)) {
348                       type = E27_NAME;
349                   } else if (EVENT_TYPE.EXIT.equals(value)) {
350                       type = E28_NAME;
351                   } else if (STATE_TYPE.INITIAL.equals(value)) {
352                       type = E18_NAME;
353                   } else if (STATE_TYPE.FINAL.equals(value)) {
354                       type = E19_NAME;
355                   } else if (EXECUTE.equals(value)) {
356                       type = E20_NAME;
357                   } else if (TRANSITTO.equals(value)) {
358                       type = E21_NAME;
359                   } else if (ON.equals(value)) {
360                       type = E22_NAME;
361                   } else if (STATEMACHINE.equals(value)) {
362                       type = E23_NAME;
363                   } else if (IF.equals(value)) {
364                       type = E24_NAME;
365                   } else if (ELSE.equals(value)) {
366                       type = E25_NAME;
367                   } else if (USES.equals(value)) {
368                       type = E29_NAME;
369                   } else if (INCLUDE.equals(value)) {
```

```
370                     type = E30_NAME;
371                 } else {
372                     type = E5_NAME;
373                 }
374             } else if (matcher.group(3) != null) {
375                 type = E6_NAME;
376             } else if (matcher.group(4) != null) {
377                 type = E7_NAME;
378             } else if (matcher.group(5) != null) {
379                 type = E8_NAME;
380             } else if (matcher.group(8) != null) {
381                 type = E10_NAME;
382             } else if (matcher.group(9) != null) {
383                 type = E11_NAME;
384             } else if (matcher.group(10) != null) {
385                 type = E12_NAME;
386             } else if (matcher.group(11) != null) {
387                 type = E13_NAME;
388             } else if (matcher.group(12) != null) {
389                 type = E14_NAME;
390             } else if (matcher.group(13) != null) {
391                 type = E15_NAME;
392             } else if (matcher.group(14) != null) {
393                 type = E16_NAME;
394             } else if (matcher.group(15) != null) {
395                 type = E17_NAME;
396             } else if (matcher.group(16) != null) {
397                 // Blank token.
398 //              type = E31_NAME;
399             } else if (matcher.group(17) != null) {
400                 // Comment token.
401 //              type = E32_NAME;
402             }
403 
404             int tokenStart = matcher.start();
405             int tokenEnd = matcher.end();
406             while (tokenStart > nextNewLineCharPos) {
407                 lineNumber++;
408                 nextNewLineCharPos = StringUtils.indexOf(expr, '\n', nextNewLineCharPos + 1);
409                 if (-1 == nextNewLineCharPos) {
410                     nextNewLineCharPos = expr.length() + 1;
411                 }
412             }
413 
414             // Some part of text wasn't matched by pattern.
415             if (pos < tokenStart) {
416                 tokens.add(createUnresolvedToken(expr, pos, tokenStart, lineNumber));
417             }
418 
419             log.debug(String.format("Found token: [%s]", matcher.group()));
420 
421             if (type != null) {
422                 Token token = new Token(type, matcher.group(), tokenStart, tokenEnd,
lineNumber, expr);
423                 tokens.add(token);
424             }
425             pos = tokenEnd;
426         }
427 
428         if (pos < expr.length()) {
429             tokens.add(createUnresolvedToken(expr, pos, expr.length(), lineNumber));
430         }
431     }
432 
433     private Token createUnresolvedToken(String expr, int start, int end, int lineNumber) {
434         log.debug(String.format("Unresolved char sequence [%d,%d] = %s", start, end - 1,
expr.substring(start, end)));
435 
436         return new Token(E1_NAME, expr.substring(start, end), start, end, lineNumber, expr);
437     }
438 }
439 
```

```
72              buffer.addLast(STATES_TOKEN);
73          }
74      }
75
```

6.2.2. FSMLToModel.java

```
1   package com.unimod.fsml.model.transform;
2
3   import java.util.ArrayList;
4   import java.util.HashMap;
5   import java.util.HashSet;
6   import java.util.Iterator;
7   import java.util.List;
8   import java.util.Map;
9   import java.util.Set;
10  import java.util.Map.Entry;
11
12  import org.apache.commons.logging.Log;
13  import org.apache.commons.logging.LogFactory;
14
15  import com.evelopers.unimod.core.stateworks.Action;
16  import com.evelopers.unimod.core.stateworks.ControlledObjectHandler;
17  import com.evelopers.unimod.core.stateworks.Event;
18  import com.evelopers.unimod.core.stateworks.EventProviderHandler;
19  import com.evelopers.unimod.core.stateworks.Guard;
20  import com.evelopers.unimod.core.stateworks.Model;
21  import com.evelopers.unimod.core.stateworks.State;
22  import com.evelopers.unimod.core.stateworks.StateMachine;
23  import com.evelopers.unimod.core.stateworks.StateType;
24  import com.evelopers.unimod.core.stateworks.Transition;
25  import com.evelopers.unimod.plugin.eclipse.model.GNormalState;
26  import com.evelopers.unimod.plugin.eclipse.model.GStateMachineHandle;
27  import com.evelopers.unimod.plugin.eclipse.model.GSubmachineList;
28  import com.evelopers.unimod.plugin.eclipse.model.GTransition;
29  import com.evelopers.unimod.runtime.ControlledObject;
30  import com.evelopers.unimod.runtime.context.StateMachineContext;
31  import com.unimod.fsml.FSMLTokens;
32  import com.unimod.fsml.FSMLTokens.EVENT_TYPE;
33  import com.unimod.fsml.util.COHelper;
34
35  /**
36   * Transforms fsml files into {@link StateMachine state machine} model.
37   *
38   * @author Ivan Lagunov
39   */
40  public class FSMLToModel implements ControlledObject {
41      private static final Log log = LogFactory.getLog(FSMLToModel.class);
42
43      private Model model;
44      private StateMachine sm;
45      private StateType sourceStateType;
46      private State sourceState;
47      private String targetStateName;
48      private String guardString;
49      private Guard guard;
50      private String eventType;
51      private List<Event> eventList;
52      private String actionName;
53      private List<Action> actionList;
54      /**
55       * Map from state name to the list of unfinished transitions from this state.
56       */
57      private Map<String, List<UnfinishedTransition>> unfinishedTransitions;
58      /**
59       * Map from event provider class name to its instance name.
60       */
61      private Map<String, String> eventProviders = new HashMap<String, String>();
62      /**
63       * Map from controlled object instance name to its full class name.
64       */
65      private Map<String, String> controlledObjects = new HashMap<String, String>();
66      private StringBuilder className = new StringBuilder("");
67      private Set<String> submachines = new HashSet<String>();
68      private Set<String> states = new HashSet<String>();
69
70      public FSMLToModel(Model model) {
```

```
71          this.model = model;
72
73          this.guardString = "";
74          this.actionName = "";
75          this.sourceStateType = StateType.NORMAL;
76          this.unfinishedTransitions = new HashMap<String, List<UnfinishedTransition>>();
77      }
78
79      /**
80       * @unimod.action.descr create state machine
81       */
82      @SuppressWarnings("unused")
83      public void z1(StateMachineContext context) {
84          String stateMachineName = COHelper.getTokenValue(context);
85          sm = getStateMachine(stateMachineName);
86          sourceState = sm.getTop();
87
88          for (Entry<String, String> entry : eventProviders.entrySet()) {
89              EventProviderHandler epHandler = model.createEventProviderHandler(entry.getValue(),
entry.getKey());
90              sm.createIncomingAssociation(epHandler);
91          }
92      }
93
94      /**
95       * @unimod.action.descr save submachine
96       */
97      public void z2(StateMachineContext context) {
98          String innerStateMachineName = COHelper.getTokenValue(context);
99          StateMachine innerStateMachine = getStateMachine(innerStateMachineName);
100
101         sm.addStateMachine(innerStateMachine);
102         submachines.add(innerStateMachineName);
103     }
104
105     /**
106      * @unimod.action.descr save source state type
107      */
108     public void z3(StateMachineContext context) {
109         String stype = COHelper.getTokenValue(context);
110         if (stype.equals(FSMLTokens.STATE_TYPE.INITIAL)) {
111             sourceStateType = StateType.INITIAL;
112         } else if (stype.equals(FSMLTokens.STATE_TYPE.FINAL)) {
113             sourceStateType = StateType.FINAL;
114         }
115     }
116
117     /**
118      * @unimod.action.descr create source state
119      */
120     public void z4(StateMachineContext context) {
121         String stateName = COHelper.getTokenValue(context);
122         State newState = sm.createState(stateName, sourceStateType);
123         sourceState.addSubstate(newState);
124         sourceState = newState;
125         states.add(stateName);
126
127         sourceStateType = StateType.NORMAL;
128
129         // If there are some transitions to finish.
130         if (unfinishedTransitions.containsKey(stateName)) {
131             for (UnfinishedTransition unfinishedTransition :
unfinishedTransitions.remove(stateName)) {
132                 createTransitions(unfinishedTransition.getSourceState(), sourceState,
133                         unfinishedTransition.getGuard(),
134                         unfinishedTransition.getEvents(),
135                         unfinishedTransition.getActions());
136             }
137         }
138     }
139
140     /**
141      * @unimod.action.descr save guard string
142      */
143     public void z5(StateMachineContext context) {
144         String guardPart = COHelper.getTokenValue(context);
145
146         if (!guardPart.equals(FSMLTokens.IF)) {
```

```
147             guardString += guardPart;
148         }
149     }
150 
151     /**
152      * @unimod.action.descr create transitions and actions
153      */
154     @SuppressWarnings("unused")
155     public void z6(StateMachineContext context) {
156         if ((actionList != null) || (targetStateName != null)) {
157             if ((null == eventType) && (null == eventList)) {
158                 eventType = EVENT_TYPE.ANY;
159             }
160 
161             if (EVENT_TYPE.ANY.equals(eventType)) {
162                 eventType = null;
163                 eventList = new ArrayList<Event>();
164                 eventList.add(Event.ANY);
165             }
166 
167             if (eventType != null) {
168                 saveActionsForState(sourceState, actionList, eventType);
169                 eventType = null;
170             } else {
171                 saveTransitions(sourceState, targetStateName, guard, eventList, actionList);
172                 targetStateName = null;
173                 guard = null;
174             }
175 
176             actionList = null;
177         }
178         eventList = null;
179     }
180 
181     /**
182      * @unimod.action.descr add event
183      */
184     public void z7(StateMachineContext context) {
185         if (null == eventList) {
186             eventList = new ArrayList<Event>();
187         }
188 
189         String eventName = COHelper.getTokenValue(context);
190         eventList.add(new Event(eventName));
191     }
192 
193     /**
194      * @unimod.action.descr create guard
195      */
196     @SuppressWarnings("unused")
197     public void z8(StateMachineContext context) {
198         if (guardString.length() != 0) {
199             guard = sm.createGuard(guardString);
200             guardString = "";
201         }
202     }
203 
204     /**
205      * @unimod.action.descr save event type
206      */
207     public void z9(StateMachineContext context) {
208         eventType = COHelper.getTokenValue(context);
209     }
210 
211     /**
212      * @unimod.action.descr save action name
213      */
214     public void z10(StateMachineContext context) {
215         actionName += COHelper.getTokenValue(context);
216     }
217 
218     /**
219      * @unimod.action.descr add action
220      */
221     @SuppressWarnings("unused")
222     public void z11(StateMachineContext context) {
223         if (null == actionList) {
224             actionList = new ArrayList<Action>();
```

```
225          }
226
227          actionList.add(sm.createAction(actionName));
228          actionName = "";
229      }
230
231      /**
232       * @unimod.action.descr save target state name
233       */
234      public void z12(StateMachineContext context) {
235          targetStateName = COHelper.getTokenValue(context);
236      }
237
238
239      /**
240       * @unimod.action.descr save event provider class
241       */
242      public void z13(StateMachineContext context) {
243          className.append(COHelper.getTokenValue(context));
244      }
245
246      /**
247       * @unimod.action.descr add event provider
248       */
249      public void z14(StateMachineContext context) {
250          String epClassName = className.toString();
251          if (!eventProviders.containsKey(epClassName)) {
252              String epName = generateEventProviderName();
253              eventProviders.put(epClassName, epName);
254              // Event providers will be created on state machine creation (z1).
255          }
256          className = new StringBuilder("");
257      }
258
259      /**
260       * @unimod.action.descr save controlled object class
261       */
262      public void z15(StateMachineContext context) {
263          className.append(COHelper.getTokenValue(context));
264      }
265
266      /**
267       * @unimod.action.descr add controlled object
268       */
269      public void z16(StateMachineContext context) {
270          String coName = COHelper.getTokenValue(context);
271          String coClassName = className.toString();
272          controlledObjects.put(coName, coClassName);
273          ControlledObjectHandler coHandler = model.createControlledObjectHandler(coName,
coClassName);
274          sm.createOutgoingAssociation(coHandler);
275          className = new StringBuilder("");
276      }
277
278      /**
279       * @unimod.action.descr add submachines
280       */
281      public void z17(StateMachineContext context) {
282          if (sourceState instanceof GNormalState) {
283              GNormalState gns = (GNormalState) sourceState;
284              for (Object o : gns.getSubMachineHandles().asCollection()) {
285                  GStateMachineHandle gsmh = (GStateMachineHandle) o;
286                  submachines.add(gsmh.getName());
287              }
288              gns.setSubMachineHandles(new GSubmachineList(submachines.toArray(new String[0])));
289          } else {
290              log.error("z2: Adding submachine to not normal state.");
291          }
292          submachines.clear();
293      }
294
295      /**
296       * @unimod.action.descr move to super state
297       */
298      public void z18(StateMachineContext context) {
299          sourceState = sourceState.getSuperstate();
300      }
301
```

```
302       /**
303        * Finds existing or creates new state machine in model.
304        *
305        * @param stateMachineName name of state machine.
306        * @return state machine with the given name.
307        */
308       private StateMachine getStateMachine(String stateMachineName) {
309           StateMachine stateMachine = model.getStateMachine(stateMachineName);
310           if (null == stateMachine) {
311               stateMachine = model.createStateMachine(stateMachineName);
312           }
313           return stateMachine;
314       }
315
316       /**
317        * Creates transitions if target state was already added to state machine,
318        * saves transitions as unfinished otherwise.
319        *
320        * @param sourceState the source state of each transition.
321        * @param targetStateName the name of target state of each transition.
322        * @param guard the guard condition on each transition.
323        * @param eventList the list of events for transitions.
324        * @param actionList the list of actions for each transition.
325        */
326       private void saveTransitions(State sourceState, String targetStateName, Guard guard,
List<Event> eventList, List<Action> actionList) {
327           State targetState;
328           if (null == targetStateName) {
329               targetState = sourceState;
330           } else {
331               targetState = sm.findState(targetStateName);
332           }
333
334           if (targetState != null) {
335               createTransitions(sourceState, targetState, guard, eventList, actionList);
336           } else {
337               // Saves transition until targetState is initialized.
338               UnfinishedTransition unfinishedTransition = new UnfinishedTransition(sourceState,
eventList, actionList, guard);
339
340               if (unfinishedTransitions.get(targetStateName) == null) {
341                   unfinishedTransitions.put(targetStateName, new
ArrayList<UnfinishedTransition>());
342               }
343
344               unfinishedTransitions.get(targetStateName).add(unfinishedTransition);
345           }
346       }
347
348       /**
349        * Creates transitions using specified data.
350        *
351        * @param sourceState the source state of each transition.
352        * @param targetState the target state of each transition.
353        * @param guard the guard condition on each transition.
354        * @param eventList the list of events for transitions.
355        * @param actionList the list of actions for each transition.
356        */
357       private void createTransitions(State sourceState, State targetState, Guard guard,
List<Event> eventList, List<Action> actionList) {
358           for (Event event : eventList) {
359               GTransition transition = (GTransition) sm.createTransition(sourceState,
targetState, guard, event);
360
361               saveActionsForTransition(transition, actionList);
362               transition.getLabel().update();
363           }
364       }
365
366       /**
367        * Registers actions for specified transition.
368        *
369        * @param transition last parsed transition.
370        * @param actionList list of actions to register.
371        */
372       private void saveActionsForTransition(Transition transition, List<Action> actionList) {
373           if (actionList != null) {
374               for (Action action : actionList) {
```

6.2. Объекты управления

6.2.1. `Stack.java`

```
1   package com.unimod.fsml.model;
2
3   import java.util.LinkedList;
4
5   import com.evelopers.unimod.runtime.ControlledObject;
6   import com.evelopers.unimod.runtime.context.StateMachineContext;
7
8   /**
9    * @author Ivan Lagunov
10   */
11  @SuppressWarnings("unused")
12  public class Stack implements ControlledObject {
13      final static private String PARENTHESIS = "parenthesis";
14      final static private String BRACE = "brace";
15      final static private String STATES_TOKEN = "states_token";
16      private LinkedList<String> buffer = new LinkedList<String>();
17
18      /**
19       * @unimod.action.descr stack is empty
20       */
21      public boolean x1(StateMachineContext context) {
22  //      Logger.CONSOLE.info(buffer);
23          return buffer.isEmpty();
24      }
25
26      /**
27       * @unimod.action.descr PARENTHESIS "()" on top
28       */
29      public boolean x2(StateMachineContext context) {
30          return !buffer.isEmpty() && PARENTHESIS.equals(buffer.getLast());
31      }
32
33      /**
34       * @unimod.action.descr BRACE "{}" on top
35       */
36      public boolean x3(StateMachineContext context) {
37          return !buffer.isEmpty() && BRACE.equals(buffer.getLast());
38      }
39
40      /**
41       * @unimod.action.descr STATES_TOKEN token on top
42       */
43      public boolean x4(StateMachineContext context) {
44          return !buffer.isEmpty() && STATES_TOKEN.equals(buffer.getLast());
45      }
46
47      /**
48       * @unimod.action.descr pop
49       */
50      public void z1(StateMachineContext context) {
51          buffer.removeLast();
52      }
53
54      /**
55       * @unimod.action.descr push PARENTHESIS "()"
56       */
57      public void z2(StateMachineContext context) {
58          buffer.addLast(PARENTHESIS);
59      }
60
61      /**
62       * @unimod.action.descr push BRACE "{}"
63       */
64      public void z3(StateMachineContext context) {
65          buffer.addLast(BRACE);
66      }
67
68      /**
69       * @unimod.action.descr push STATES_TOKEN token
70       */
71      public void z4(StateMachineContext context) {
```

```
375                 transition.addOutputAction(action);
376             }
377         }
378     }
379
380     /**
381      * Registers actions for specified state.
382      *
383      * @param state last parsed state.
384      * @param actionList list of actions to register.
385      * @param eventType the type of both event and actions.
386      */
387     private void saveActionsForState(State state, List<Action> actionList, String eventType) {
388         if (eventType.equals(EVENT_TYPE.ENTER)) {
389             for (Action action : actionList) {
390                 state.addOnEnterAction(action);
391             }
392         } else if (eventType.equals(EVENT_TYPE.EXIT)) {
393             //TODO: this event type will be only in Unimod 2.
394         }
395     }
396
397     /**
398      * @unimod.action.descr source is initial state
399      */
400     @SuppressWarnings("unused")
401     public boolean x1(StateMachineContext context) {
402         if (null == sourceState) {
403             log.warn("FSMLToModel: source state is null");
404             return false;
405         }
406
407         return sourceState.getType().equals(StateType.INITIAL);
408     }
409
410     /**
411      * @unimod.action.descr source is final state
412      */
413     @SuppressWarnings("unused")
414     public boolean x2(StateMachineContext context) {
415         if (null == sourceState) {
416             log.warn("FSMLToModel: source state is null");
417             return false;
418         }
419
420         return sourceState.getType().equals(StateType.FINAL);
421     }
422
423     /**
424      * @unimod.action.descr source is normal state
425      */
426     @SuppressWarnings("unused")
427     public boolean x3(StateMachineContext context) {
428         if (null == sourceState) {
429             log.warn("FSMLToModel: source state is null");
430             return false;
431         }
432
433         return sourceState.getType().equals(StateType.NORMAL);
434     }
435
436     /**
437      * @unimod.action.descr event type is on enter
438      */
439     @SuppressWarnings("unused")
440     public boolean x4(StateMachineContext context) {
441         return EVENT_TYPE.ENTER.equals(eventType);
442     }
443
444     /**
445      * Returns an iterator over saved event providers.
446      *
447      * @return an iterator over saved event providers.
448      */
449     public Iterator<String> getEventProvidersIterator() {
450         return eventProviders.keySet().iterator();
451     }
452
```

```
453      /**
454       * Returns an iterator over saved controlled objects' names.
455       *
456       * @return an iterator over saved controlled objects' names.
457       */
458      public Iterator<String> getControlledObjectsIterator() {
459          return controlledObjects.keySet().iterator();
460      }
461
462      /**
463       * Returns the controlled object class name by given instance name.
464       *
465       * @return the class name of controlled object.
466       */
467      public String getControlledObjectClass(String coName) {
468          return controlledObjects.get(coName);
469      }
470
471      /**
472       * Returns an iterator over created states.
473       *
474       * @return an iterator over created states.
475       */
476      public Iterator<String> getStates() {
477          return states.iterator();
478      }
479
480      private String generateEventProviderName() {
481          return String.format("p%d", eventProviders.size() + 1);
482      }
483  }
484
```

6.2.3. ObjectContext.java

```
1    package com.unimod.fsml.model;
2
3    import java.util.Iterator;
4    import java.util.Stack;
5
6    import org.eclipse.jdt.internal.core.util.Util;
7
8    import com.evelopers.unimod.runtime.ControlledObject;
9    import com.evelopers.unimod.runtime.context.StateMachineContext;
10   import com.unimod.fsml.util.COHelper;
11
12   /**
13    * @author Ivan Lagunov
14    */
15   public class ObjectContext implements ControlledObject {
16
17       public enum ContextType {
18           DEFAULT, EVENT_PROVIDER, CONTROLLED_OBJECT, EVENT, GUARD, ACTION, STATE
19       }
20
21       private Stack<String> s = new Stack<String>();
22       private boolean idTokenFinished = false;
23       private ContextType contextType = ContextType.DEFAULT;
24
25       /**
26       * @unimod.action.descr push last token to context
27       */
28       public void z1(StateMachineContext context) {
29           s.push(COHelper.getTokenValue(context));
30           idTokenFinished = false;
31       }
32
33       /**
34       * @unimod.action.descr clear context data
35       */
36       public void z2(StateMachineContext context) {
37           clearStack();
38       }
39
40       /**
41        * @unimod.action.descr set default context
42        */
```

```
43      public void z3(StateMachineContext context) {
44          clearStack();
45          contextType = ContextType.DEFAULT;
46      }
47
48      /**
49      * @unimod.action.descr set event context
50      */
51      public void z4(StateMachineContext context) {
52          clearStack();
53          contextType = ContextType.EVENT;
54      }
55
56      /**
57      * @unimod.action.descr set guard context
58      */
59      public void z5(StateMachineContext context) {
60          clearStack();
61          contextType = ContextType.GUARD;
62      }
63
64      /**
65       * @unimod.action.descr set action context
66       */
67      public void z6(StateMachineContext context) {
68          clearStack();
69          contextType = ContextType.ACTION;
70      }
71
72      /**
73       * @unimod.action.descr set event provider context
74       */
75      public void z7(StateMachineContext context) {
76          clearStack();
77          contextType = ContextType.EVENT_PROVIDER;
78      }
79
80      /**
81       * @unimod.action.descr set controlled object context
82       */
83      public void z8(StateMachineContext context) {
84          clearStack();
85          contextType = ContextType.CONTROLLED_OBJECT;
86      }
87
88      /**
89       * @unimod.action.descr set state context
90       */
91      public void z9(StateMachineContext context) {
92          clearStack();
93          contextType = ContextType.STATE;
94      }
95
96      /**
97       * @unimod.action.descr mark id token finished
98       */
99      public void z10(StateMachineContext context) {
100         idTokenFinished = true;
101     }
102
103     public ContextType getContextType() {
104         return contextType;
105     }
106
107     public boolean isIdTokenFinished() {
108         return idTokenFinished;
109     }
110
111     /**
112      * Returns the top element of context stack without removing it from stack.
113      *
114      * @return the top element of context stack.
115      */
116     public String peekTopElement() {
117         return s.peek();
118     }
119
120     /**
```

```
121         * Returns full Java path created from the sequence of id's in context stack.
122         *
123         * @return full Java path.
124         */
125        public String peekFullJavaPath() {
126            StringBuilder sb = new StringBuilder();
127
128            for (Iterator<String> iter = s.iterator(); iter.hasNext();) {
129                sb.append(iter.next());
130                if (iter.hasNext()) {
131                    sb.append('.');
132                }
133            }
134
135            return sb.toString();
136        }
137
138        /**
139         * Returns depth of the stack.
140         *
141         * @return depth of the stack.
142         */
143        public int getStackDepth() {
144            return s.size();
145        }
146
147        private void clearStack() {
148            s.clear();
149            idTokenFinished = false;
150        }
151    }
152
```

6.2.4. StreamErrorListener.java

```
1   package com.unimod.fsml.model;
2
3   import org.apache.commons.logging.Log;
4   import org.apache.commons.logging.LogFactory;
5
6   import com.evelopers.unimod.runtime.ControlledObject;
7   import com.evelopers.unimod.runtime.context.StateMachineContext;
8
9   /**
10   * @author Ivan Lagunov
11   */
12  @SuppressWarnings("unused")
13  public class StreamErrorListener implements ControlledObject {
14      private static final Log logger = LogFactory.getLog(StreamErrorListener.class);
15
16      /**
17      * @unimod.action.descr token was skipped
18      */
19      public void z1(StateMachineContext context) {
20          logger.debug("Token was skipped");
21      }
22
23      /**
24      *@unimod.action.descr tokens were added to stream
25      */
26      public void z2(StateMachineContext context) {
27          logger.debug("Tokens were added to stream");
28      }
29  }
30
```