Правительство Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Тихонова Мария Валерьевна

Генерация кода в режиме ”метамоделирования на лету” в системе QReal

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст.преп. кафедры СП Литвинов Ю.В.

Рецензент:
аспирант кафедры СП Подкопаев А.В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Maria Tikhonova

Code generation for “metamodelling on the fly” in QReal system

Bachelor’s Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
Senior Lecturer Litvinov Y.V.

Reviewer:
Postgraduate Student Podkopaev A.V.

Saint-Petersburg
2015

Оглавление

Введение

В настоящее время всё большее внимание уделяется средствам, с помощью которых можно ускорить процесс разработки программного обеспечения. Одним из них является визуальное моделирование [6] — способ задавать программу в терминах графических объектов, вместо того, чтобы использовать абстракции программирования. Использование языков общего назначения (например, UML) для разработки программного обеспечения приводит к тому, что диаграммы, описанные с помощью этих языков, выглядят сложными и теряют наглядность. Часто проще и удобнее создать специальный язык для решения какой-то задачи или группы задач и описать решение с помощью этого языка, чем специфицировать решение на языке общего назначения. Такой подход к созданию программного обеспечения называется предметно-ориентированным моделированием, а визуальные языки для решения конкретных задач — предметно-ориентированными языками. Создавать предметно-ориентированные языки "с нуля" для каждой конкретной задачи очень долго и сложно, поэтому используют DSM-платформы [2]. Примерами DSM-платформ являются MetaEdit+ [9], Eclipse Modeling Framework [8], Microsoft Visualization and Modeling SDK [3]. В данной работе будет использоваться DSM-платформа QReal [10], которая разрабатывается на кафедре системного программирования Санкт-Петербургского государственного университета.

Одной из основных составляющих частей DSM-платформы является *метаязык* — специальный язык, который предназначается для описания новых предметно-ориентированных языков. Модель, созданная с помощью метаязыка, должна содержать описание всех сущностей и связей нового языка, а также правила построения из них целевых визуальных моделей. Модель, описанная с помощью метаязыка, называется *метамоделью*.

Для быстрого прототипирования визуального языка на кафедре системного программирования был разработан метод "метамоделирования на лету", реализованный в системе QReal [4, 5] . Этот метод позволяет вносить изменения в визуальный язык прямо в процессе его разработки. При таком подходе уровень метамодели скрыт от пользователя.

Одной из важнейших возможностей DSM-платформы является также возможность генерации кода по моделям. Для этого нужно некоторое средство задания правил генерации по визуальному языку. В системе QReal был реализован специальный визуальный язык для задания правил генерации, однако же, это решение работает только для языков, описанных с помощью метамоделей. Хотелось бы генерировать код и в процессе "метамоделирования на лету". Поскольку данный режим позволяет быстро вносить изменения в язык, добавление возможности получать итоговый код для "метамоделирования на лету" позволяет ускорить процесс разработки. По-

лучается законченное решение, когда на глазах у пользователя создаётся и язык, и генератор, позволяющий получать из диаграмм на этом языке реальную пользу.

Таким образом, целью данной дипломной работы является разработка и реализация средства задания правил генерации для “метамоделирования на лету” в DSM-платформе QReal.

1. Постановка задачи

В рамках данного диплома было необходимо решить следующие задачи.

1. Проанализировать существующие решения в различных DSM-платформах, таких, как MetaEdit+, Eclipse Modeling Framework, Microsoft Visualization and Modeling SDK, а также существующие решения в DSM-платформе QReal, и предложить способы переиспользования этих результатов для поставленной задачи.

2. Разработать метод задания правил генерации для “метамоделирования на лету”, проанализировать, для каких случаев он может быть применен.

3. Разработать программное средство задания правил генерации для “метамоделирования на лету” и провести его апробацию.

2. Обзор

2.1. Обзор средств задания правил генерации в существующих системах

2.1.1. Платформа MetaEdit+

Данная DSM-платформа разрабатывается финской компанией MetaCase [9]. Система MetaEdit+ предназначена для создания DSM-решений и удобной работы с ними. DSM-платформа MetaEdit+ состоит из следующих подсистем:

- MetaEdit+ Workbench — средство создания визуальных языков и генераторов;
- MetaEdit+ — полнофункциональная среда, предназначенная для разработки систем с поддержкой возможности использования языков, генераторов кода и документации, созданных с помощью MetaEdit+ Workbench.

У системы MetaEdit+ есть много различных возможностей, среди которых стоит отметить:

- возможность работать с несколькими визуальными языками сразу при разработке сложной системы;
- многоплатформенность;
- возможность работать с диаграммами, созданными в других DSM-системах (в частности, Eclipse и Microsoft Modeling SDK);
- возможность генерировать код по диаграмме с помощью встроенных редакторов;
- возможность задавать собственные генераторы.

Для генерации в MetaEdit+ есть набор встроенных генераторов, которые позволяют генерировать по модели код на различных языках (C++, Java, Delphi). Также разработчики могут задавать свои генераторы для собственных языков моделирования. Для этого в среде есть средство задания генераторов, в котором используется специальный текстовый язык. Фиксированные строчки, которые будут присутствовать в сгенерированном коде без изменений, в этом языке выделяются одинарными кавычками. Имена и свойства объектов, которые берутся из модели, не выделяются никак. Также не выделяются никак строчки самого генератора.

На рис. 1 представлено средство для задания правил генерации в системе MetaEdit+. На рис. 2 представлен код, сгенерированный по этим правилам.

Правила генерации задаются в отдельном окне; в нём есть дерево элементов визуального языка (сущностей, отношений, ролей) и шаблонных структур текстового

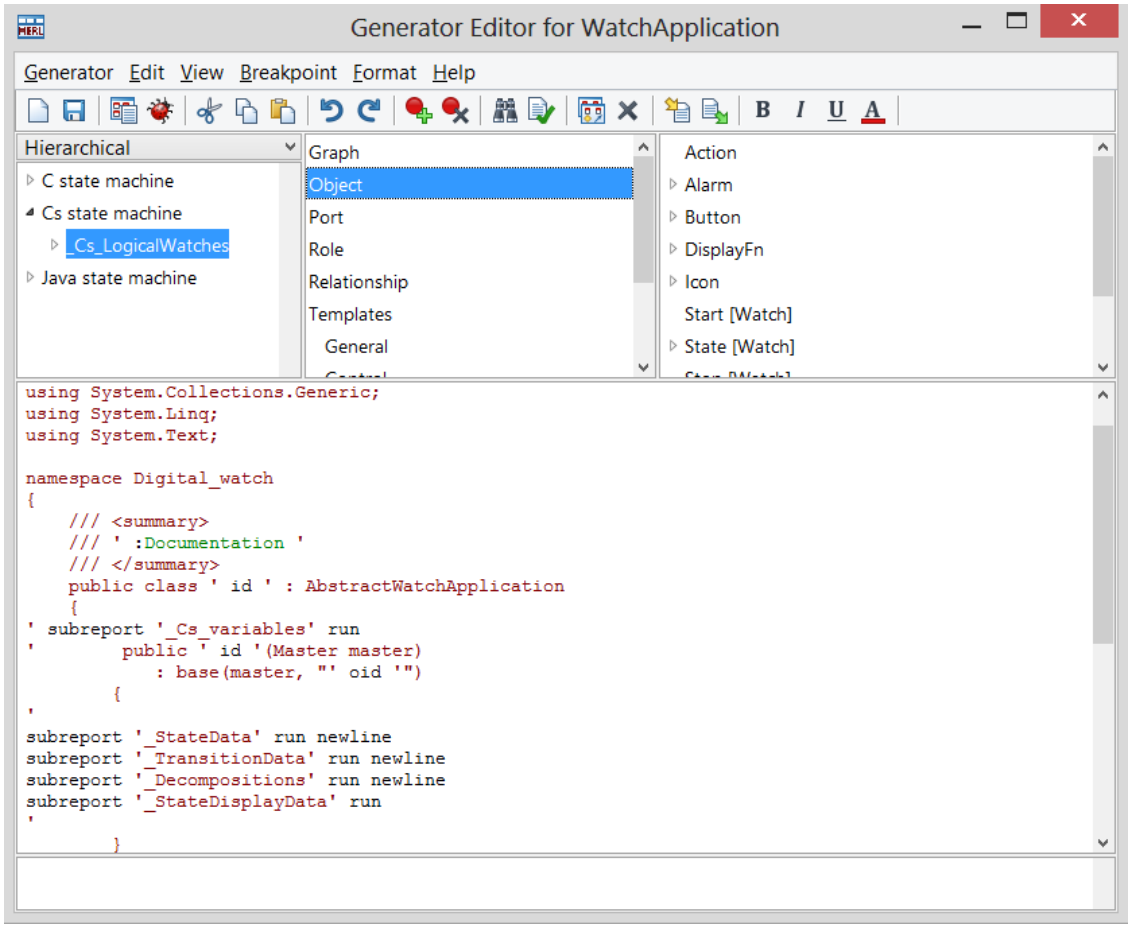

Рис. 1: Средство для задания правил генерации в системе MetaEdit+

языка генератора. Дерево организовано следующим образом: в одном столбце можно выбрать некоторый общий раздел (граф, объекты, отношения, шаблоны), а в следующем столбце можно выбрать уже какой-либо конкретный представитель раздела, например, сущность Alarm в диаграмме, описывающей часть приложения “Часы”, или её свойство Name, или шаблон foreach. Причём, если выбрать шаблон, то в текстовом окне появится автоматически дополненный шаблон, как представлено на рис. 3:

Шаблоны организованы в пять подтипов: General, Control, External I/O, Strings & Numbers, Representations. Конструкции, содержащиеся в подтипе General, позволяют получить общую информацию о текущем элементе (его имя, тип и т.д.); конструкции из подтипа Control позволяют совершать обход элементов и их сортировку; раздел External I/O позволяет перенаправлять вывод в файл, Strings & Numbers — присваивать локальные переменные и работать со строками, Representations — получать информацию о графических характеристиках объекта.

Таким образом, с помощью средства задания правил генерации в MetaEdit+ можно задавать сколь угодно сложные генераторы, хотя код на этом языке достаточно сложен для понимания. Созданные и изменённые генераторы можно также отлаживать в специальном редакторе.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Digital_watch
{
    /// <summary>
    /// Application for an alarm at a certain time of day. AlarmClock allows the user to set the alarm time by
editing hours and minutes and the alarm rings when the alarm time is reached.
    /// </summary>
    public class AlarmClock : AbstractWatchApplication
    {
     const int a22_4160 = 1;
     const int a22_4217 = 2;
     const int a22_4272 = 3;
     const int a22_4405 = 4;
     const int a22_4978 = 5;
     const int a22_5693 = 6;
     const int a22_5913 = 7;
```

Рис. 2: Окно со фрагментом кода, полученного с помощью генерации

```
foreach type
{
}
```

Рис. 3: Автодополнение шаблона в системе MetaEdit+

2.1.2. Платформа Microsoft Visualization and Modeling SDK (VMSDK)

Данная платформа [3] является средством для визуального моделирования, которое позволяет разрабатывать предметно-ориентированные решения в Microsoft Visual Studio. В платформе VMSDK есть возможность задавать визуальные языки, графические редакторы для них и дополнительные инструменты для работы с ними.

В системе VMSDK есть специальные генераторы кода, которые по заданным правилам генерируют итоговую программу. Для задания правил генерации используется специальный текстовый язык T4. Всё, что мы пишем в шаблон, затем транслируется в некоторый промежуточный класс, в котором есть метод Generate(), и для получения итогового файла с кодом вызывается этот метод.

Задавать логику генератора можно с помощью двух языков: C# или Visual Basic. В шаблоне нужно указать, какой язык будет использоваться для задания правил генерации.

Ниже перечислены основные характеристики средства задания правил генерации в данной системе:

- обычный текст, ничем не выделенный, транслируется без изменений в текстовую последовательность, которая появится в итоговом коде;
- текст, выделенный символами <# и #>, является кодом самого генератора и вставляется непосредственно в код метода Generate();
- строки вида <#= expr #> позволяют брать некоторые характеристики из модели, например, код, представленный ниже, будет генерировать enum по всем состояниям конечного автомата, как показано на рис. 4;

```
// States of the tool
enum States
{
<#
foreach(State state in StateMachine.States)
{
#>
    <#= state.Name #>,
```

Рис. 4: Код для генерации перечисления

- участки кода, выделенные символами <#+ #>, добавляются в промежуточный класс в некоторое место вне метода Generate(); таким образом, с их помощью можно выделить отдельный метод генерации, который будет вызываться в методе Generate().

2.1.3. Средство Eclipse:Actifsource

Данный продукт [7] предназначен для задания предметно-ориентированных визуальных языков и является подключаемым модулем к среде Eclipse. Actifsource включает в себя средства для создания метамоделей, а также средства для задания генераторов, тестирования и рефакторинга метамоделей.

Средство для задания правил генерации генерирует итоговый код по некоторому шаблону, написанному пользователем в специальном редакторе. Текст шаблона состоит из некоторых управляющих конструкций и фрагментов итогового кода. В отличие от Microsoft Visualization And Modeling SDK, Actifsource умеет анализировать входной текст без специальных символов, выделяющих управляющие конструкции. Это делает код более наглядным и простым для понимания.

Actifsource поддерживает наиболее распространённые языки (для них реализована подсветка синтаксиса, комментарии). Сейчас в списке поддерживаемых языков более 30 языков. Если нужного языка нет в списке, то можно определить свой собственный язык. Язык также можно переключить в любое время.

Ниже перечислены основные принципы языка для задания правил генерации:

- цветом выделяются конструкции итогового языка (class, method);

- новый шаблон может быть создан для какого-либо типа из метамодели; если шаблон создан для типа, то шаблон применяется ко всем экземплярам данного типа; результатом генерации является один файл для одного экземпляра типа; если же шаблон не базируется ни на каком типе, то этот шаблон применяется только один раз, и результатом генерации является один файл;
- чтобы обратиться к некоторой характеристике модели и вставить её в итоговый код, нужно вызвать меню Content Assist и выбрать нужную характеристику (некоторый класс из исходной модели); чтобы обратиться к её полям, нужно нажать "." и выбрать нужное поле (например, поле name у класса Service);
- характеристики, взятые из модели, будут в коде генератора подчёркнуты.

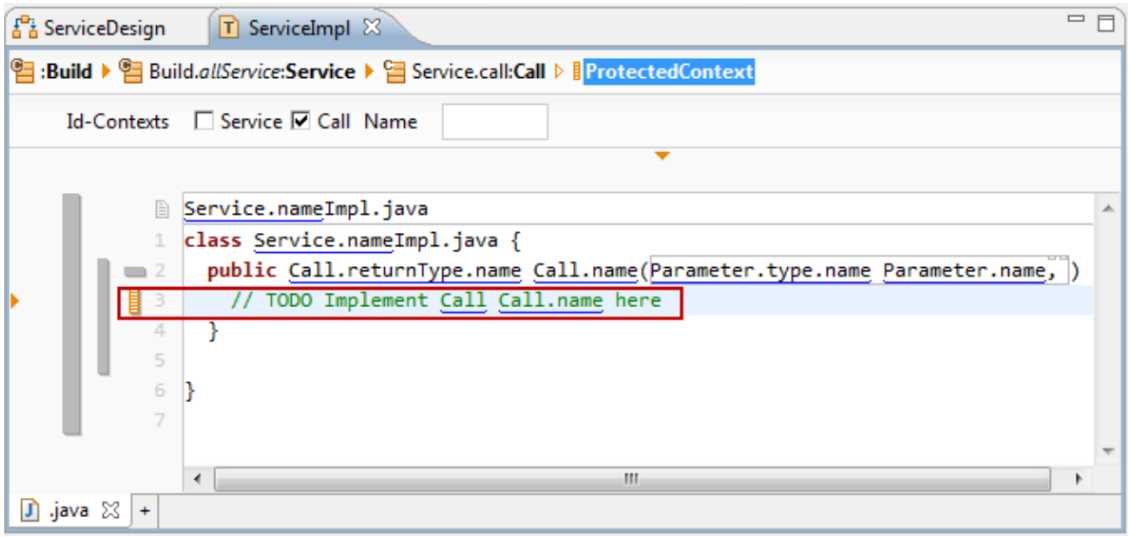

Рис. 5: Шаблон генератора

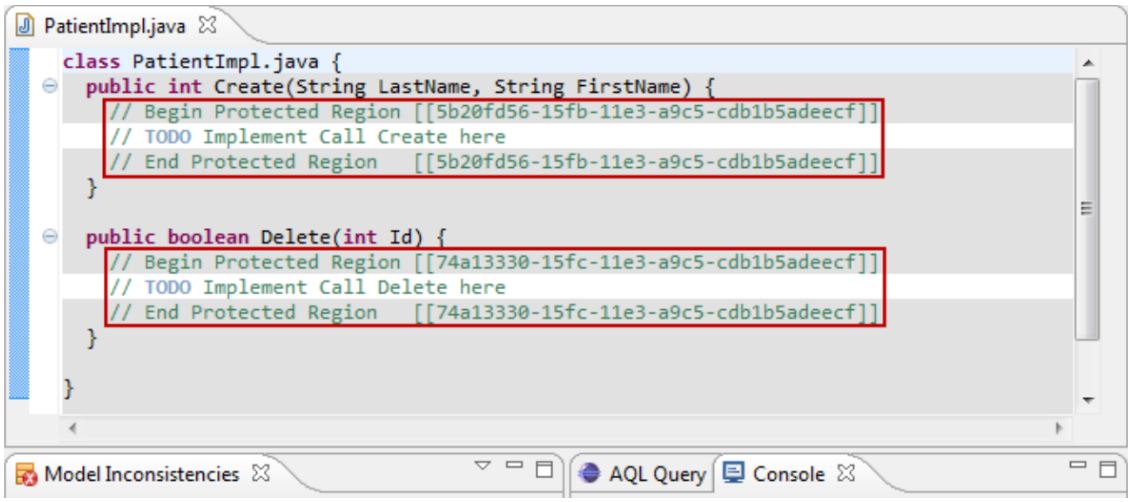

Рис. 6: Сгенерированный код

На представленных ниже изображениях (рис. 5, рис. 6) показан пример шаблона генератора и сгенерированный по нему код.

Такой подход делает невозможной полную генерацию кода, но с помощью него можно успешно производить прототипирование.

2.1.4. Язык Razor

Данный язык [1] является языком разметки и позволяет встраивать код, написанный на некотором языке программирования, в веб-страницы. Язык был разработан в 2010 году, а в 2011 году была сделана его поддержка для Microsoft Visual Studio. Razor основан на технологии ASP.NET и предназначен для создания веб-приложений; он поддерживает языки C# и Visual Basic. Страницы, созданные с помощью языка Razor, могут быть описаны как HTML-страницы с двумя типами кода: HTML-код и Razor-код. Данный язык не имеет отношения к DSM-платформам, но рассматривается в обзоре, потому что задача, которую он решает, близка к задаче генераторов в DSM-платформах: сгенерировать некоторый итоговый код по модели и правилу генерации.

Когда сервер осуществляет чтение данной страницы, он сначала исполняет код на Razor, а затем отправляет HTML-страницу браузеру. Код, исполняемый на сервере, может выполнять задачи, которые нельзя выполнить в браузере (к примеру, доступ к базе данных сервера).

Основные правила языка Razor представлены ниже.

1. Блоки кода, написанные на Razor, заключены в специальные символы @....
2. Встроенные выражения (переменные и функции) начинаются со специального символа @.
3. Блоки кода заканчиваются точкой с запятой.
4. Переменные описываются с помощью ключевого слова var.
5. Строки заключаются в двойные кавычки.
6. Код, написанный на C#, чувствителен к регистру; код, написанный на Visual Basic, к регистру не чувствителен.
7. Блоки не нужно никак специально выделять, Razor достаточно умён, чтобы найти конец блока самостоятельно.

На представленных ниже изображениях (рис. 7, рис. 8) показан код на языке Razor и C# и сгенерированный по нему текст.

```
<html>
<body>
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
var greeting = "Welcome to our site!";
var weekDay = DateTime.Now.DayOfWeek;
var greetingMessage = greeting + " Here in Huston it is: " + weekDay;
}

<p>The greeting is: @greetingMessage</p>
</body>
</html>
```

Рис. 7: Фрагмент кода на языке Razor

The value of myMessage is: Hello World

The greeting is: Welcome to our site! Here in Huston it is: Sunday

Рис. 8: Сгенерированный текст

2.1.5. Средства задания правил генерации в QReal QReal:Geny

Для системы QReal была разработана система задания правил генерации: текстовый язык Geny, редактор для него и интерпретатор. Это решение является некоторым усреднением решений, используемых в других metaCASE-системах: решение содержит прототип итоговой программы (такой способ задания правил генерации используется в системах VMSDK и Actifsource) и позволяет использовать управляющие конструкции (такой способ задания правил генерации используется в системе MetaEdit+).

В основу системы задания правил генерации Geny легли следующие 6 принципов.

1. Принцип “прямой передачи”. Этот принцип означает, что любой текст, который не обрамлён управляющими конструкциями, будет передан без изменений в результирующий поток. Данный принцип позволяет повысить читаемость кода, не загромождая код большим количеством специальных символов.

2. Принцип “текущего объекта”. Данный принцип означает, что при обращении к текущему объекту не требуется указывать его имя (текущий объект — это текущий узел графа, построенного по диаграмме). Этот принцип позволяет уменьшить количество кода в правиле.

3. Система заданий. Каждый файл на языке Geny является именованным заданием и может содержать внутри вложенные задания. Этот принцип обеспечивает модульность программ и позволяет задавать более сложные генераторы.

4. Управляющие строки. Это строки, которые начинаются со специальной конструкции #! и служат для выполнения команд:

   - управляющая конструкция toFile fileName перенаправляет вывод в файл;
   - конструкция foreach используется для обхода объектов из некоторого списка;
   - конструкция saveObj objName сохраняет объект с переданным именем;
   - конструкция switch/case обеспечивает механизм условных переходов, сравнивая характеристики текущего объекта с определёнными значениями;
   - конструкция if %propertyName% operation %propertyValue% сравнивает характеристики текущего объекта с некоторым значением (в качестве operation могут выступать операторы ==, !=, contains);
   - конструкция {,} отделяет код от других конструкций.

5. Управляющая конструкция внутри контекста. Этот принцип даёт возможность обращаться непосредственно к значениям характеристик объекта или к результату выполнения задания. Чтобы обратиться к какой-либо характеристике текущего объекта, нужно заключить её в специальные символы @@ (например, чтобы получить значение имени объекта, нужно написать @@name@@).

6. Система меток объектов. Данный принцип позволяет сохранить объект по имени с помощью операции saveObj, чтобы в дальнейшем при работе с другими объектами иметь возможность обратиться к этому объекту.

Ниже (рис. 9) представлен пример задания правил генерации на языке Geny. В этом примере используется специальный редактор языка Geny, который позволяет сделать написание кода более наглядным. В данном примере используется специальная подсветка синтаксиса, позволяющая избежать использования большого количества специальных символов. В частности, следующие управляющие конструкции выделены с помощью цвета, а не символов:

- управляющие конструкции выделены синим цветом, а не специальными символами #!;
- имена полей выделяются с помощью жирного шрифта, без использования специальных символов @@ @@;

- блоки кода распознаются автоматически, что позволяет не выделять их специальными символами #! и #!.

Однако недостатком данного решения является то, что оно не было до конца реализовано и встроено в систему QReal, вследствие чего переиспользование его в данной работе затруднительно.

2.1.6. Визуальное средство задания правил генерации

В системе QReal также было реализовано визуальное средство задания правил генерации — специальный визуальный язык, позволяющий описывать формально генератор для структурных языков, т.е. тех языков, которые не совершают действия над потоками данных или управления.

Язык позволяет задавать следующие конструкции:

- тип элемента языка, для которого задаётся генератор, и текстовый код, который будет генерироваться при обходе элемента этого типа;
- правила генерации для всех элементов, вложенных в некоторый элемент;
- текстовые шаблоны, по которым будет сгенерирован файл или метка с некоторым именем; под текстовым шаблоном подразумевается некоторый код, который написан на целевом языке программирования и программа на котором будет получена в результате работы генератора; такой шаблон может содержать некоторые метки, выделенные специальными символами:
  - для метки, которая выделена специальными символами @@ @@, в итоговом коде будет вместо неё вставлен некоторый кусок кода;
  - для метки, выделенной специальными символами ## ##, в итоговом коде будет подставлено значение некоторого поля из исходной модели;
- конвертеры, которые позволяют преобразовывать строковые представления

Визуальное средство также не было переиспользовано в проектируемом решении, поскольку оно также не было встроено в систему QReal и так как было решено реализовать именно текстовый язык для задания правил генерации. Добавление возможности задавать правила генерации с помощью визуального языка является возможной дальнейшей перспективой развития этой работы.

2.1.7. Сравнительный анализ средств задания правил генерации

В таблицах 1, 2 представлен сравнительный анализ вышеперечисленных средств задания правил генерации.

	MetaEdit+	Microsoft VM SDK
Выделение текста, без изменений попадающего в итоговый код	Выделяется символами ' '	Не выделяется никак
Выделение текста кода генератора	Не выделяется никак	Код генератора выделяется символами <# и #> Код вида <#= expr #> позволяет брать некоторые характеристики из модели Код вида <#+ #> выделяется в отдельный метод в генераторе
Возможность генерировать код любой сложности	Есть	Есть
Возможность генерировать код на любом языке программирования	Есть	Нет, код может быть сгенерирован только на языках C# и Visual Basic

Таблица 1: Сравнение средств задания правил генерации MetaEdit+ и VM SDK

	Eclipse Actifsource	Razor	QReal:Geny	Визуальное средство в QReal
Выделение текста, без изменений попадающего в итоговый код	Не выделяется	HTML-код	Не выделяется никак	-
Выделение текста кода генератора	Не выделяется	Выделяется специальными символами @	Начинается с символов #!	-
Возможность генерировать код любой сложности	Нет, можно только прототипировать итоговый код	Есть	Есть	Есть
Возможность генерировать код на любом языке программирования	Есть	-	Есть	Есть

Таблица 2: Сравнение средств задания правил генерации Actifsource, Razor, QReal:Geny и визуального средства

2.2. Требования к средству для задания правил генерации

В результате проведённого обзора к средству задания правил генерации в режиме “метамоделирования на лету” были сформулированы следующие требования.

1. Правило генерации задаётся для элемента метамодели.
2. Язык для задания правил генерации должен содержать конструкции, позволяющие обходить модель (все элементы некоторого типа, все исходящие связи и т.д.).
3. Язык для задания правил генерации должен быть простым.
4. Средство для задания правил генерации должно позволять генерировать код на любом языке программирования.

Для средства задания правил генерации в режиме “метамоделирования на лету” в качестве основного образца было выбраны средства задания правил генерации в системе MetaEdit+ и в системе Eclipse, поскольку метод, используемый в данных системах, позволяет генерировать код на любом языке программирования, в то же время в этих системах не используются сложные конструкции для выделения кода генератора и кода итоговой программы.

2.3. Обзор используемых решений

2.3.1. Meta-CASE система QReal

Система QReal — это DSM-платформа, которая разрабатывается на кафедре системного программирования математико-механического факультета СПбГУ на протяжении нескольких лет.

Данная система позволяет создавать модели на каком-либо визуальном языке, а также создавать визуальные языки, описывая их метамодель на специальном метаязыке. По описанию метамодели на метаязыке генерируется редактор, который затем подгружается в среду. С созданными языками можно работать точно так же, как и с уже имеющимися.

В системе имеется абстрактное ядро, которое реализует общую функциональность для всех языков, и плагины — подключаемые модули, которые реализуют конкретные языки или некоторую функциональность, характерную для конкретного DSM-решения. Ядро ничего не знает про конкретные языки и работает со всеми DSM-решениями одинаково, получая знания о языке и действиях, которые с ним можно совершать, из подключаемых модулей.

Например, с помощью QReal были реализованы такие технологии, как QReal:Robots (TRIKStudio) [12], редактор некоторых диаграмм UML, редакторы баз данных, редактор языка Дракон [11], редактор BPMN [13] и другие. Среди дополнительной функциональности, которую реализуют подключаемые модули, есть следующие полезные инструменты: инструмент, позволяющий задавать ограничения на создаваемый язык средство, позволяющее задавать и применять рефакторинги к диаграммам на визуальном языке инструмент, позволяющий проводить визуальную отладку программы

Также для некоторых языков есть генераторы, позволяющие получить по диаграмме на данном визуальном языке её представление на некотором текстовом языке.

Для реализации лексического и синтаксического анализаторов текстовых языков для средства задания правил генерации в режиме “метамоделирования на лету” была использована библиотека для разработки синтаксических анализаторов текстовых языков, реализованная в системе QReal. Библиотека содержит в себе средства для создания лексических анализаторов с использованием регулярных выражений. Класс Lexer позволяет задавать ключевые слова и лексемы языка, а Parser позволяет по описанию грамматики получить абстрактное синтаксическое дерево.

2.3.2. “Метамоделирование на лету”

В рамках проекта QReal была разработана методология “метамоделирования на лету” [4, 5] и реализована соответствующая технология, позволяющая вносить изменения в визуальный язык непосредственно во время работы с ним. Метаредактор при таком подходе не нужен, а уровень метамодели языка скрыт от пользователя. Такой подход позволяет сократить время на разработку DSM-решения.

Для того, чтобы реализовать технологию “метамоделирования на лету”, в системе QReal была реализована поддержка интерпретации метамоделей. При интерпретации метамодели редактор использует внутреннее представление метамодели для получения информации о языке.

Режим “метамоделирования на лету” позволяет делать следующие операции с элементами разрабатываемого языка.

1. Добавить элемент на палитру.
   При этом можно задать его имя и тип (узел или связь). Элемент добавится в палитру, и у него появится некоторое графическое представление, которое можно будет изменить в редакторе форм. Если обновить графическое представление элемента, все его экземпляры на диаграмме должны будут обновить свой вид.

2. Удалить элемент с палитры.
   Элемент можно удалить, если на диаграмме нет его экземпляров.

3. Редактировать свойства элемента.
   Редактировать тип, имя и значение по умолчанию у существующих свойств.

4. Добавлять новые свойства, задавая их имя, тип и значение по умолчанию.

5. Удалять существующие свойства.
   Если на диаграмме есть экземпляры элемента, свойства которого мы пытаемся удалить, то свойства удаляются у всех экземпляров.

Метамодель, которая была получена в режиме “метамоделирования на лету”, можно также открыть и редактировать в метаредакторе.

3. Язык для задания правил генерации

Средство для задания правил генерации в режиме “метамоделирования на лету” позволяет задавать правила для каждого элемента разрабатываемого языка. При этом можно использовать ключевые конструкции языка для задания правил генерации, а также обращаться к свойствам элемента, для которого задаётся правило. Текст, который без изменений попадёт в итоговый код, выделяется кавычками (‘’). Текст самого генератора не выделяется никак. Обращение к текущему элементу происходит по имени этого элемента в языке (например, если мы задаём правило для элемента State, то обратиться к его свойству Name можно так: State->Name).

Ниже (3, 4) представлены управляющие конструкции языка для задания правил генерации.

Конструкция	**Описание**
foreach (identifier in list)	Конструкция, которая позволяет обходить все элементы, принадлежащие данному списку list.
newline	Конструкция, позволяющая осуществить перевод строки.
tab	Конструкция, позволяющая добавить табуляцию.
callGeneratorFor(element, generatorName)	Конструкция, позволяющая вызвать в данном фрагменте кода генератор для какого-либо другого элемента и вставить получившийся результат в этот фрагмент; второй параметр необязателен и позволяет вызвать только генератор с данным именем. Если второй параметр не задан, то правило применяется целиком.
Generator generatorName	Конструкция, позволяющая создать отдельный генератор внутри элемента, чтобы потом вставить получившийся результат в нужный фрагмент кода.
.transitionEnd	Конструкция, позволяющая обратиться к элементу, связанному с концом данной связи.
.transitionStart	Конструкция, позволяющая обратиться к элементу, связанному с началом данной связи.
this	Конструкция, позволяющая обратиться к текущему элементу.
generateToFile(element, fileName, generatorName)	Конструкция, позволяющая перенаправить вывод в файл с именем fileName; два оставшихся параметра имеют тот же смысл, что и параметры конструкции callGeneratorFor.
if (condition) program1 else program2	Конструкция, позволяющая в зависимости от истинности условия (condition) выполнять либо первую ветку (program1), либо вторую (program2); ветка else является необязательной; в качестве условия может выступать булевское значение некоторого идентификатора или сравнение строк на равенство (==) или неравенство (!=).

Таблица 3: Управляющие конструкции для работы с элементами

Конструкция	**Описание**
.outcomingLinks	Конструкция, позволяющая получить исходящие из элемента связи.
.incomingLinks	Конструкция, позволяющая получить входящие в элемент связи.
.links	Конструкция, позволяющая получить все связи, имеющие один конец, связанный с этим элементом.

Таблица 4: Управляющие конструкции для работы со связями

4. Архитектура генератора

Лексический и синтаксический анализаторы и генератор итогового кода были реализованы как подключаемый модуль (также называемый плагин) к системе QReal. Также для реализации данного модуля потребовалось реализовать механизм работы с подключаемыми модулями для интерпретатора метамоделей. Теперь в системе QReal можно писать плагины, которые имеют доступ одновременно к модели и метамодели.

На рис. 10 показана архитектура предлагаемого решения; синим цветом обозначены добавленные классы и модули. Плагины для интерпретатора наследуются от интерфейса InterpretedPluginInterface. Его метод init принимает репозиторий с метамоделью языка и моделью и инициализирует плагин. Плагины возвращают в MainWindow список действий, которые они выполняют, а также список дополнительных пунктов контекстного меню для элементов языка. Плагины интерпретатора инициализируются при открытии или создании интерпретируемой диаграммы. В этот момент происходит поиск динамически загружаемых библиотек, предоставляющих интерфейс плагина интерпретатора.

Далее рассмотрим архитектуру подключаемого модуля, о котором идёт речь в дипломе. Он состоит из следующих частей.

1. Компонента ast содержит в себе классы абстрактного синтаксического дерева, которые возвращает синтаксический анализатор правил.

2. Класс TokenTypes содержит в себе список всех лексем языка, класс Lexer их определяет, а в классе Parser определены правила, по которым получается абстрактное синтаксическое дерево.

3. Компонента generator отвечает за генерацию итогового кода по абстрактному синтаксическому дереву, модели и метамодели языка. Компонента состоит из нескольких частей.

   (a) Каждому классу из компоненты ast соответствует класс в генераторе, который принимает узел соответствующего типа и возвращает строку сгенерированного кода (и, если требуется, перенаправляет вывод в некоторый файл).

   (b) Кроме того, имеется таблица идентификаторов, которая хранит в себе информацию об имени идентификатора, его типе и списке значений идентификаторов элементов в модели, которые он может принимать.

5. Реализация

Общая схема генерации итогового кода по диаграмме выглядит следующим образом. Для каждого элемента метамодели в её контекстное меню добавляется пункт “Добавить правило генерации”. Если выбрать данный пункт, то будет показано окно для задания правила генерации для данного элемента. В нём есть список шаблонов языка, список имён свойств данного элемента и текстовое поле для задания правила. В метамодель создаваемого языка для каждого элемента добавляется свойство generationRule, которое содержит в себе текст правила для данного элемента. Разбор правил начинается с корневого элемента (элемента, который мы при создании пометили как корневой элемент языка). Правило сначала разбирается на лексемы, а затем по нему строится абстрактное синтаксическое дерево. Если в правиле встречается строка callGeneratorFor (Element, GeneratorName), где Element — имя элемента метамодели, а GeneratorName — имя генератора, то в метамодели ищется правило генерации для элемента с именем Element, затем оно разбирается на лексемы и по нему тоже строится абстрактное синтаксическое дерево.

На рис. 11 показан внешний вид окна для задания правила генерации.

Имя элемента языка	**Свойства**
State (состояние)	Name (имя состояния)
StartState (начальное состояние)	Name (имя состояния)
EndState (конечное состояние)	Name (имя состояния)
Transition (переход)	Symbol (символ, по которому происходит переход в следующее состояние)

Таблица 5: Элементы языка для описания конечных автоматов

6. Апробация

Была проведена апробация предложенного средства для задания правил генерации на двух примерах, представленных ниже.

6.1. Апробация на примере с конечным автоматом

В качестве первого примера был выбран язык для описания конечных автоматов. Язык был реализован в режиме “метамоделирования на лету.” Элементы языка и их свойства перечислены в табл. 5.

С помощью этого языка была создана простая модель, представленная на рис. 12.

По данной модели нужно было сгенерировать итоговый код, который должен был содержать в себе, во-первых, перечисление (Enum) всех имён состояний, имеющихся на диаграмме (IntegerPartDigitState, PointState, FractionalPartDigitState, StartState, EndState), а во-вторых, логику переходов. Для каждого элемента логика перехода такова: если текущий символ принадлежит множеству символов, допустимых для какого-либо перехода из данного состояния, то сменить текущее состояние на то состояние, по которому этот переход был осуществлён; если же по текущему символу невозможен никакой переход, то сменить состояние на конечное.

Правила генерации и сгенерированный по ним код представлены в главе Приложение.

6.2. Апробация на примере с UML-диаграммой классов

В качестве второго примера был выбран язык для описания UML диаграмм.

Элементы языка для описания UML диаграмм и их свойства перечислены в табл.6.

С помощью этого языка была создана диаграмма, представленная на рис. 13. Поскольку режим “метамоделирования на лету” не позволяет задавать вложенные структуры, свойства были реализованы в нотации Entity-Relationship.

По данной диаграмме нужно было сгенерировать файлы для каждого класса, ко-

Имя элемента языка	**Свойства**
UmlClass (класс UML)	Name (имя класса)
Attribute (свойство)	Name (имя свойства) Type (тип свойства)
Generalization (наследование)	-
HasAttribute (элемент имеет свойство)	-

Таблица 6: Элементы языка для задания UML-диаграмм

торые бы содержали в себе конструктор класса, приватные поля, соответствующие атрибутам класса, и методы, позволяющие выставить значение атрибута (setters) или получить значение атрибута (getters).

Правила генерации и сгенерированный по ним код представлены в главе Приложение.

Заключение

В рамках данной дипломной работы были получены следующие результаты.

1. Разработан язык для задания правил генерации программного кода по визуальным моделям в режиме “метамоделирования на лету”.

2. Спроектировано средство задания правил генерации в режиме “метамоделирования на лету”.

3. Выполнена реализация лексического, синтаксического анализаторов и генератора для системы QReal.

4. Выполнена апробация средства задания правил генерации для двух примеров: для конечного автомата и для UML-диаграмм классов.

Список литературы

[1] Introduction to ASP.NET Web Programming Using the Razor Syntax (C). — URL: http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-(c) (дата обращения: 25.05.2015).

[2] Steven Kelly Juha-Pekka Tolvanen. Domain-Specific Modeling: Enabling Full Code Generation. — Wiley-IEEE Computer Society Pr., 2008.

[3] Visualization and Modeling SDK – Domain-Specific Languages. — URL: http://msdn.microsoft.com/ru-ru/library/bb126259(v=vs.100).aspx (дата обращения: 22.04.2015).

[4] А.И. Птахина. Разработка метамоделирования “на лету” в системе QReal // Список-2013: Материалы всероссийской научной конференции по проблемам информатики. 2013г ., Санкт-Петербург. — 2012.

[5] А.И. Птахина. Эволюция языков при метамоделировании «на лету» в DSM-платформе QReal // Список-2014: Материалы всероссийской научной конференции по проблемам информатики. 23–25 апреля 2014 г. Санкт-Петербург. — 2014.

[6] В. Кознов Д. Основы визуального моделирования. — Бином. Лаборатория Знаний, Интернет-Университет Информационных Технологий, 2008.

[7] Домашняя страница Actifsource. — URL: http://www.actifsource.com/ (дата обращения: 22.04.2015).

[8] Домашняя страница Eclipse Modeling Project. — URL: https://www.eclipse.org/modeling/emf/ (online; accessed: 26.05.2015).

[9] Домашняя страница MetaEdit+. — URL: http://www.metacase.com/ (дата обращения: 22.04.2015).

[10] Домашняя страница QReal. — URL: http://qreal.ru/ (дата обращения: 05.05.2015).

[11] Колантаевская А.С. Исследование удобства процесса моделирования на базе DSM-платформы QReal // дипл. работа. — 2013. — URL: http://se.math.spbu.ru/SE/diploma/2013/b/KolantaevskayaAnna_text.pdf (online; accessed: 2015-05-26).

[12] Литвинов Ю.В. Кириленко Я.А. TRIK Studio: среда обучения программированию с применением роботов // V Всероссийская конференция «Современное технологическое обучение: от компьютера к роботу» (сборник тезисов), СПб., ЗАО «Полиграфическое предприятие № 3», 2015. — 2015.

[13] Т.Ю. Агапова. Обзор современных систем управления бизнес-процессами // Список-2013: Материалы всероссийской научной конференции по проблемам информатики. 2013г ., Санкт-Петербург. — 2012.

Приложение

Генерация для конечного автомата

Ниже представлено правило для генерации перечисления.

```
'enum State {'
foreach (current in State) {
        current->name
    ','
        newline
}

foreach (current in StartState) {
    current->name
}
','
newline

foreach (current in EndState) {
    current->name
}
newline

'}'
```

По нему и по модели автомата будет сгенерировано следующее перечисление.

```
enum State {
        IntegerPartDigitState,
        PointState,
        FractionalPartDigitState,
        StartState,
        EndState
}
```

Ниже представлено правило для генерации переходов.

```
'case ' this->Name ':' newline
        foreach (transition in this::outcomingLinks(Transition)) {
                'if (symbol == ' transition->symbol ')' newline
                    'currentState = '
                    transition.transitionEnd->Name
```

```
                ';'
                newline
                    'break;'
            newline
            }
            'else' newline
                    'currentState = '
                    foreach (endState in EndState) {
                            endState->Name
                    }
                    ';' newline
                    'break;' newline
```

По нему и по модели будет сгенерирован следующий код (для краткости представлен код только для состояния PointState; на месте многоточия — перечисление всех символов, по которым можно попасть в состояние FractionalPartDigitState).

```
case PointState:
        if (symbol == ''0)
                currentState = FractionalPartDigitState;
                break;
if (symbol == ''1)
                currentState = FractionalPartDigitState;
                break;
...
        else
                currentState = EndState;
                break;
```

Генерация для UML-диаграммы

Ниже представлено правило генерации для класса.

```
'class ' this->Name

if (this->HasAncestor) {
        ' : public '
        foreach (transition in
            this::incomingLinks(Generalization)) {
                transition.transitionStart->Name
        }
```

```
}

newline
'{'
newline
'public:'
        newline
        tab
        this->Name '('
        foreach_excludeLast (transition in
        this::outcomingLinks(HasAttribute), exclude ',') {
                transition.transitionEnd->Type
                ' '
                transition.transitionEnd->Name
                ','
        }
');'

newline

foreach (transition in this::outcomingLinks(HasAttribute)) {
        callGeneratorFor(transition.transitionEnd
            , generatorOfGetters);
}

newline

foreach (transition in this::outcomingLinks(HasAttribute)) {
        callGeneratorFor(transition.transitionEnd
            , generatorOfSetters);
}

newline

'private:'
foreach (transition in this::outcomingLinks(HasAttribute)) {
        callGeneratorFor(transition.transitionEnd
            , generatorOfPrivateFields);
}
```

```
'}'
newline
```

Для элемента "свойство" правило выглядит следующим образом.

```
Generator generatorOfGetters {
                tab this->Type
                ' get'
                this->Name
                '() {'
                newline
                tab tab 'return m' this->Name
                ';'
                newline
                tab '};'
                newline
}

Generator generatorOfSetters {
        tab 'void set' this->Name '(' this->Type ' new'
            this->Name ') {'
        newline
        tab tab 'm' this->Name ' = new' this->Name
        ';'
        newline
        tab '};'
        newline
}

Generator generatorOfPrivateFields {
        tab this->Type ' m' this->Name ';'
        newline
}
```

Ниже представлен сгенерированный код для класса Person.

```
class Person
{
public:
        Person(String  Name, Integer  Age);
        String getName() {
                return mName;
```

```
        };
        Integer getAge() {
                return mAge;
        };

        void setName(String newName) {
                mName = newName;
        };
        void setAge(Integer newAge) {
                mAge = newAge;
        };

private:
    String mName;
        Integer mAge;
}
```

```
1  Task JavaClass
2  / Produce Java class from repo
3
4  foreach Class in elementsByType(Class)
5                  toFile name.java
6  class name {
7                              foreach MethodsContainer in children
8                                          foreach Method in children
9                  methodVisibility methodReturnType
10                 methodName( @@!task MethodParameters@@ ) {
11                 }
12                                         }
13                             }
14
15                             foreach FieldsContainer in children
16                                         foreach Field in children
17                 fieldVisibility fieldType fieldName;
18                                         }
19                             }
20                 }
21 }
22 }
```

Рис. 9: Пример задания правил генерации на языке Geny

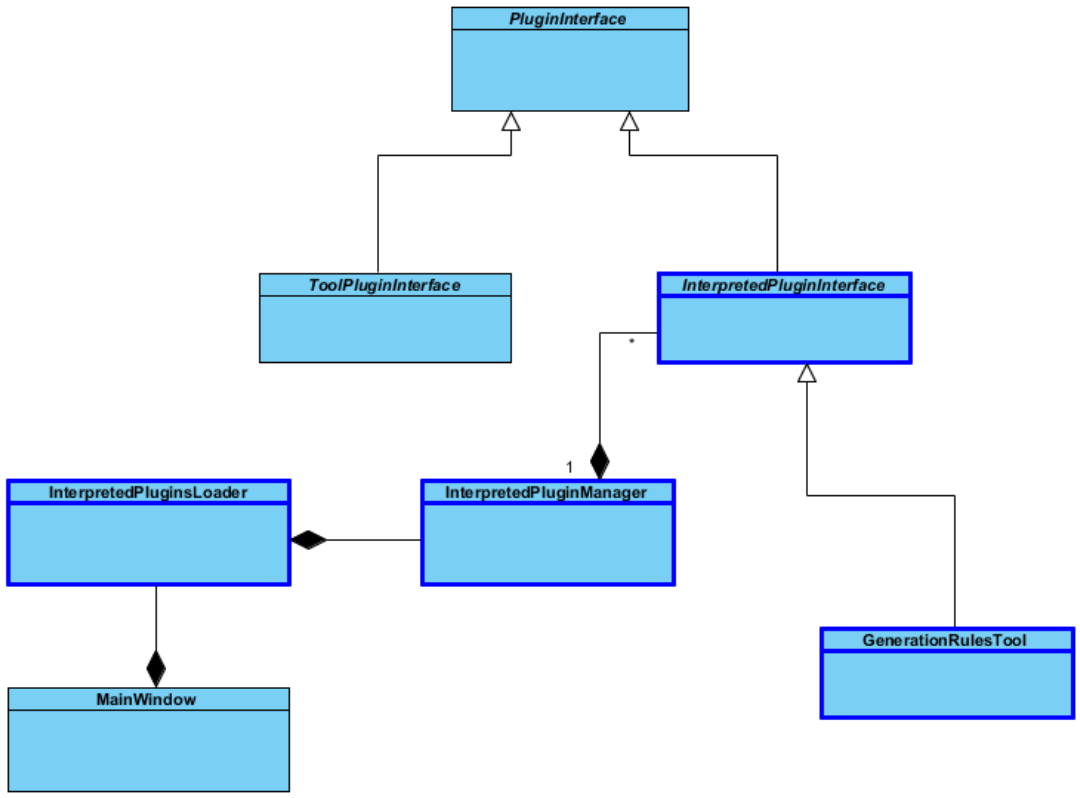

Рис. 10: Архитектура механизма для работы с подключаемыми модулями для интерпретатора метамоделей

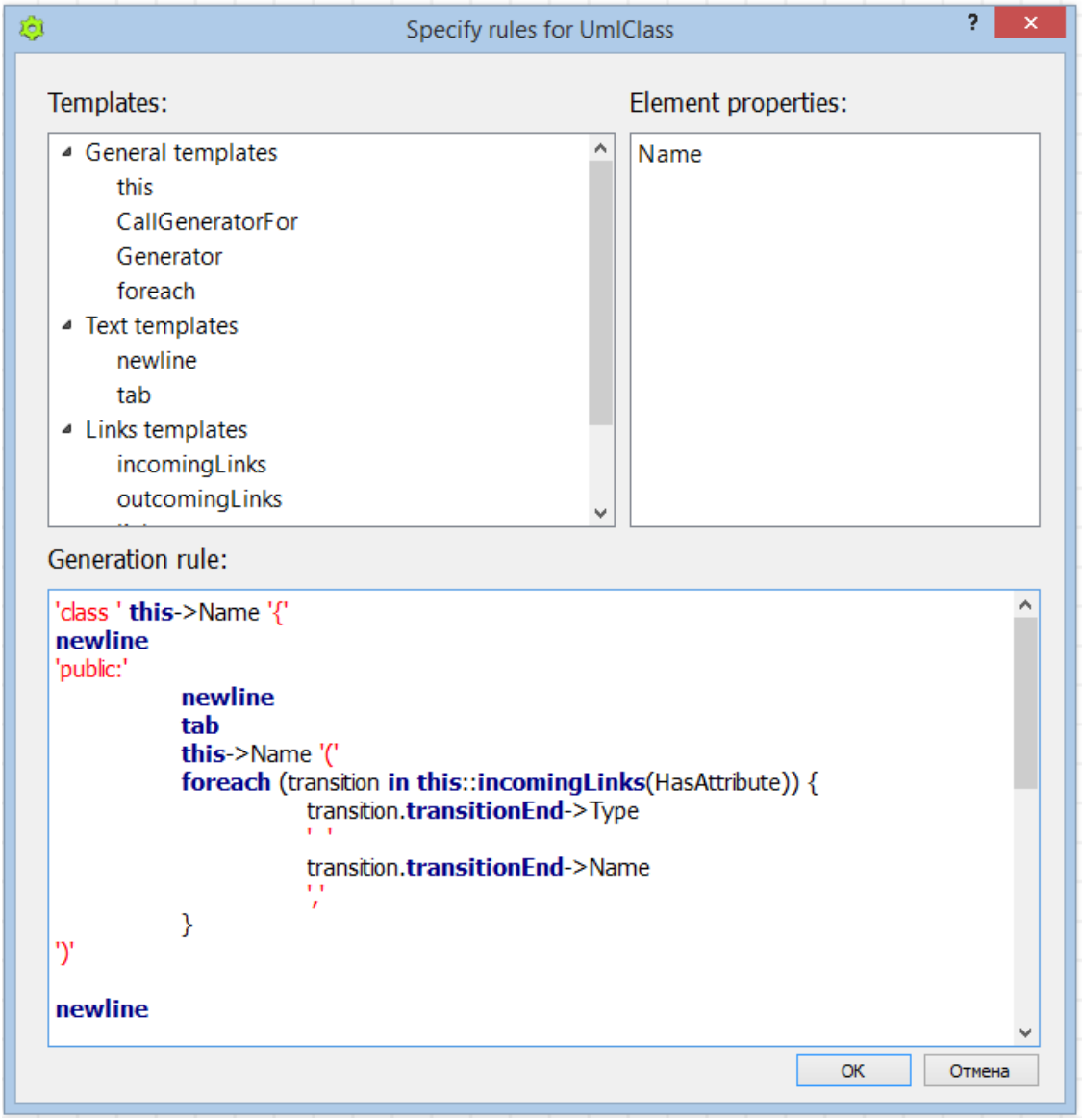

Рис. 11: Внешний вид окна для задания правил генерации

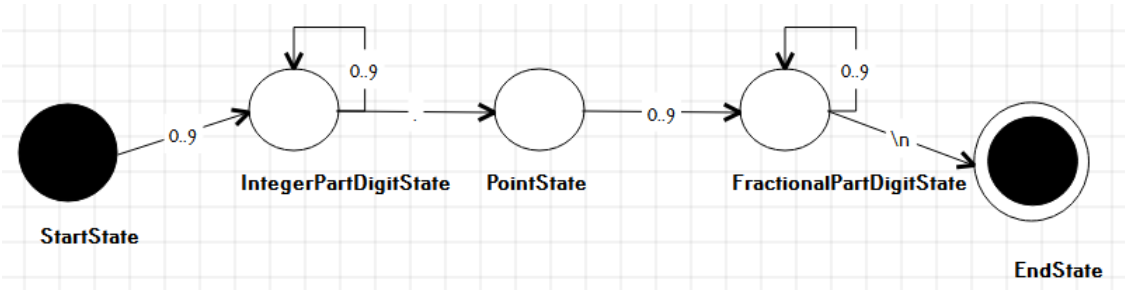

Рис. 12: Модель конечного автомата для разбора числа с плавающей точкой

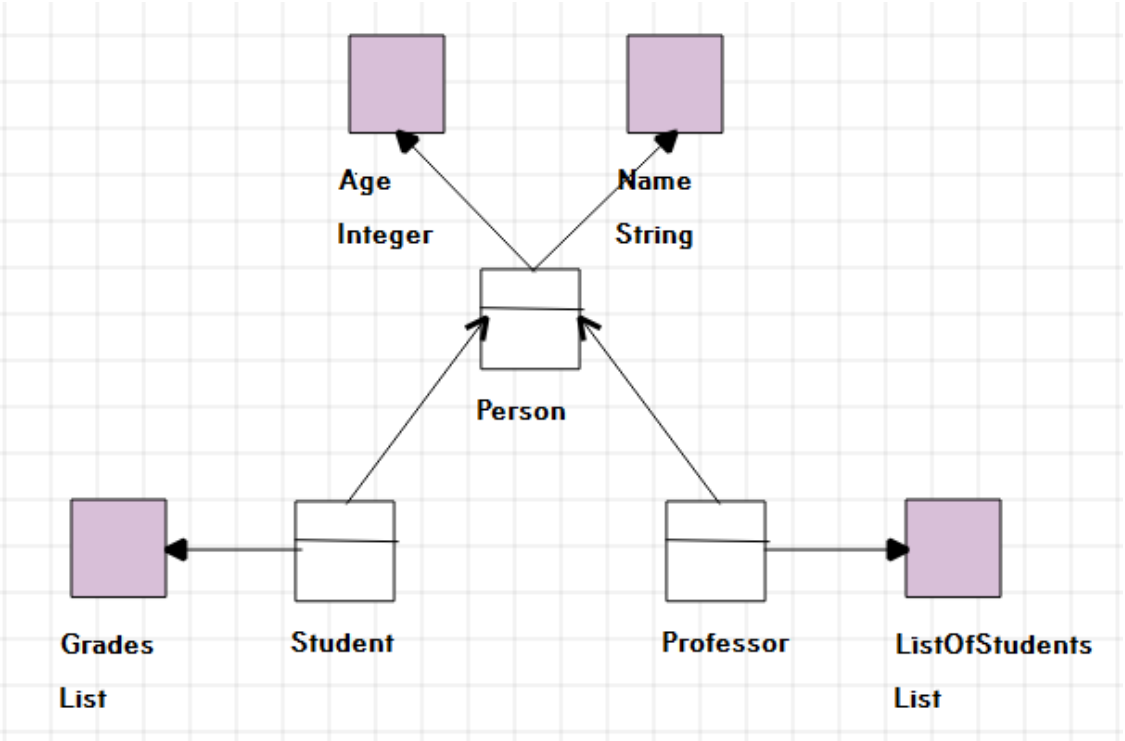

Рис. 13: Диаграмма на языке для описания UML-диаграмм