

Санкт-Петербургский государственный университет

На правах рукописи

УДК 004.4'22

Литвинов Юрий Викторович

**Разработка визуальных предметно-ориентированных  
языков**

Специальность 05.13.11 —  
математическое и программное обеспечение вычислительных машин,  
комплексов, систем и сетей

Диссертация на соискание учёной степени  
кандидата технических наук

Научный руководитель:  
д. ф.-м.н., профессор  
А.Н. Терехов

Санкт-Петербург – 2015

# Содержание

<b>Введение</b>	<b>5</b>
<b>1 Визуальные языки и их свойства</b>	<b>15</b>
1.1 Визуальное моделирование . . . . .	15
1.2 Структура визуальных языков . . . . .	18
1.2.1 Синтаксис, семантика и прагматика . . . . .	19
1.2.2 Уровни моделирования . . . . .	22
1.3 Предметно-ориентированное моделирование . . . . .	25
1.3.1 Понятие предметно-ориентированного моделирования . . .	25
1.3.2 Инструментальные средства предметно-ориентированного моделирования . . . . .	28
1.4 Свойства визуальных языков . . . . .	30
<b>2 Существующие подходы к созданию DSM-решений</b>	<b>35</b>
2.1 Фокус и структура обзора . . . . .	35
2.2 Существующие методики и приёмы разработки предметно- ориентированных языков . . . . .	37
2.2.1 Модели жизненного цикла языка . . . . .	37
2.2.2 Паттерны и рекомендации по разработке предметно- ориентированных языков . . . . .	42
2.2.3 Способы внутренней организации визуальных языков . . .	45
2.3 Создание визуальных языков в существующих DSM-платформах	48
2.3.1 Платформа MetaEdit+ . . . . .	48
2.3.2 Eclipse Modeling Project . . . . .	50
2.3.3 Платформа Generic Modeling Environment . . . . .	54
2.3.4 Платформа PSL/PSA . . . . .	56
2.3.5 Платформа AToM3 . . . . .	57

2.3.6	Платформа Microsoft Modeling SDK . . . . .	58
2.3.7	Платформа Pounamu . . . . .	60
2.3.8	Платформа DOME . . . . .	62
2.3.9	Платформа MetaLanguage . . . . .	63
2.3.10	Сравнение рассмотренных платформ . . . . .	64
2.4	Выводы . . . . .	65
<b>3</b>	<b>Методология создания DSM-решения</b>	<b>69</b>
3.1	Фазы жизненного цикла визуального предметно-ориентированного языка . . . . .	69
3.1.1	Анализ применимости . . . . .	71
3.1.2	Анализ предметной области . . . . .	72
3.1.3	Проектирование и реализация . . . . .	74
3.1.4	Развёртывание . . . . .	76
3.1.5	Эволюция языка . . . . .	77
3.1.6	Вывод из эксплуатации . . . . .	79
3.2	„Классическая“ методология . . . . .	79
3.3	Метамоделирование на лету . . . . .	87
<b>4</b>	<b>Поддержка создания предметно-ориентированных решений в системе QReal</b>	<b>93</b>
4.1	Возможности ядра системы QReal . . . . .	93
4.2	Метаредактор . . . . .	96
4.2.1	Визуальный метаязык . . . . .	97
4.2.2	Особенности языка . . . . .	97
4.2.3	Генерация редакторов . . . . .	99
4.3	Редактор формы фигур . . . . .	101
4.4	Редактор ограничений . . . . .	103
4.5	Редактор правил рефакторинга . . . . .	106
4.6	Средства поддержки технологии „метамоделирования на лету“ . .	110
	<b>Заключение</b>	<b>114</b>
	<b>Список сокращений и условных обозначений</b>	<b>116</b>

<b>Литература</b>	<b>117</b>
<b>А Применение предложенных в работе подходов</b>	<b>127</b>
A.1 Среда программирования роботов QReal:Robots . . . . .	127
A.1.1 Постановка задачи . . . . .	128
A.1.2 Существующие среды визуального программирования ро- ботов . . . . .	131
A.1.3 Требования к DSM-решению . . . . .	134
A.1.4 Визуальный язык QReal:Robots . . . . .	135
A.1.5 Инструментальные средства QReal:Robots . . . . .	141
A.1.6 Опыт применения QReal . . . . .	148
A.1.7 Результаты проекта QReal:Robots . . . . .	153
A.2 Среда программирования распределённых мобильных приложе- ний QReal:Ubiq . . . . .	155
A.2.1 Постановка задачи . . . . .	155
A.2.2 Визуальный язык QReal:Ubiq . . . . .	157
A.2.3 Обсуждение . . . . .	162
A.2.4 Дальнейшее развитие QReal:Ubiq . . . . .	163
A.2.5 Результаты проекта QReal:Ubiq . . . . .	166
A.3 Среда разработки аппаратуры QReal:HaSCoL . . . . .	167
A.3.1 Постановка задачи . . . . .	168
A.3.2 Существующие средства визуального описания аппарат- ных систем . . . . .	171
A.3.3 Предлагаемый набор визуальных языков . . . . .	174
A.3.4 Результаты проекта QReal:HaSCoL . . . . .	183
<b>В Описание метаязыка QReal</b>	<b>185</b>

## Введение

Разработка сложных программных систем, несмотря на несколько десятилетний развития программной инженерии, до сих пор остаётся непростой задачей. Связано это отчасти с тем, что программное обеспечение невидимо и нематериально, его трудно себе представить. Бороться с этой проблемой помогает визуальное моделирование — подход, при использовании которого программа представляется в виде набора графических моделей, каждая из которых описывает её с разных точек зрения. Если при проектировании объектов реального мира используются чертежи, которые могут описывать, например, схему несущих конструкций здания, схему размещения электропроводки, и т.д., то при разработке сложных программных систем могут использоваться модели, описывающие разбиение системы на компоненты, протоколы взаимодействия объектов системы, и т.д. В отличие от чертежей, которые соотносятся с тем, как будет выглядеть в реальном мире проектируемый объект после того, как будет создан, вид визуальных моделей — лишь предмет договорённости между разработчиками. Каждый человек может представлять себе программу по-разному, что создаёт дополнительные трудности при применении визуального подхода. Тем не менее, благодаря наличию стандартных широко распространённых графических языков, визуальное моделирование повышает продуктивность труда и качество результирующего продукта при разработке. Существует довольно большое количество исследований, подтверждающих это экспериментально, см., например, [1, 2].

Сейчас визуальные модели используются в основном при анализе и проектировании, а также как средство документирования и передачи информации между разработчиками. Однако же, в отличие от, например, зданий, которые по набору чертежей ещё надо построить, программы целиком или их фрагменты возможно автоматически генерировать по набору визуальных моделей. Это

позволяет непосредственно использовать результаты анализа и проектирования и в значительной степени автоматизировать труд программистов, давая им возможность работать не с кодом программы на текстовом языке, а с гораздо более наглядными визуальными моделями.

Использование визуальных языков общего назначения, таких как **UML**<sup>1</sup>, делает задачу разработки программного обеспечения только с помощью графических языков сложной, в силу наличия семантического разрыва между кодом и моделями [3,4]. Такие языки работают в тех же терминах, в которых пишется исходный код на традиционных текстовых языках (классы, объекты, компоненты и т.д.), поэтому чтобы полностью специфицировать поведение системы и сделать возможной автоматическую генерацию, модель должна содержать в себе столько же информации, что и исходный код программы, что противоречит самому понятию модели как некоего упрощения моделируемого объекта. На самом деле, визуальная модель в этом случае даже менее удобна, чем код программы — визуальные символы занимают на экране больше места, чем текст. Если же визуальная модель будет изображать только важные аспекты функционирования системы, опуская излишние подробности, то её можно будет сохранить обозримой и полезной для человека, но это сделает её бесполезной для исполнителя (например, для интерпретатора, или генератора исходного кода). Именно так, в основном, используется **UML** сейчас — как средство для анализа и дизайна системы, а сама система специфицируется ручным кодированием на текстовых языках. Большинство инструментов для рисования UML-диаграмм позволяют сгенерировать заглушки, куда предполагается дописывать код вручную, но существенного выигрыша для разработчиков это не даёт. Подтверждение этим фактам можно найти в относительно недавних исследованиях [5,6].

Существует принципиально другой подход к использованию визуального моделирования, называемый предметно-ориентированным моделированием (**DSM**<sup>2</sup>, [7]). Он основан на том наблюдении, что иногда создать новый язык для какой-то узкой предметной области или даже для конкретной задачи и решить задачу на нём оказывается быстрее и эффективнее, чем решать эту задачу на языке общего назначения. В таком случае наличие у средств поддержки

---

<sup>1</sup>Unified Modeling Language, URL: <http://uml.org/> (дата обращения: 22.02.2014г.)

<sup>2</sup>Domain Specific Modeling

создаваемого языка знаний о предметной области позволяет добиться полной автоматической генерации программ про визуальным моделям.

Существует много широко известных текстовых предметно-ориентированных языков, например, язык для работы с данными SQL<sup>3</sup>, язык для работы с текстами awk<sup>4</sup>, средства описания контекстно-свободных грамматик для генераторов синтаксических анализаторов. В каждом из этих примеров программа на предметно-ориентированном языке работает в терминах той предметной области, для которой этот язык создан, что даёт возможность не задумываясь о деталях реализации решать требуемую задачу. Например, современные генераторы синтаксических анализаторов позволяют задавать грамматику в виде, очень похожем на формы Бэкуса-Наура, при этом программист может не думать о том, как будет реализован синтаксический анализатор для этой грамматики: каким-либо из автоматных методов или рекурсивным спуском. Как правило, допускаются и некоторые неоднозначности грамматики, и грамматики, плохие с точки зрения метода разбора, который реализует синтаксический анализатор (например, леворекурсивные грамматики для метода рекурсивного спуска). Всё это позволяет реализовывать синтаксические анализаторы даже людям, весьма поверхностно представляющим себе алгоритмы синтаксического анализа. Это общее свойство предметно-ориентированных языков — знания о предметной области „спрятаны“ в инструментальные средства, что позволяет существенно расширить круг пользователей языка, вплоть до того, что на нём смогут программировать люди, далёкие от программирования. Исследования [8–10] показывают, что продуктивность труда программистов при использовании предметно-ориентированных языков вырастает в 3-10 раз по сравнению с использованием языков общего назначения, поэтому такой подход представляется весьма перспективным.

Разумеется, создавать новый предметно-ориентированный визуальный язык и инструментальные средства его поддержки „с нуля“ для каждой узкой пред-

---

<sup>3</sup>Structured Query Language, URL:

[http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=53681](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681) (дата обращения 25.01.2015)

<sup>4</sup>Спецификация AWK, URL: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html> (дата обращения 22.02.2014г.)

метной области или конкретной задачи было бы неоправданно трудозатратно. Поэтому существуют специальные средства для автоматизации этой задачи, называемые „**DSM-платформа**“, или „**metaCASE-средство**“. Такие средства позволяют задать синтаксис визуального языка, используя какой-либо формализм (как правило, это метамодели), и автоматически сгенерировать редактор этого языка и другие средства инструментальной поддержки<sup>5</sup>. Это позволяет реализовывать технологии программирования, использующие новые предметно-ориентированные языки, за время порядка дней, что делает предметно-ориентированное моделирование оправданным даже для небольших проектов. Существуют зрелые исследовательские и промышленные DSM-платформы, такие как Eclipse Modeling Project<sup>6</sup>, MetaEdit+<sup>7</sup> и другие. Однако же, несмотря на значительные преимущества предметно-ориентированного моделирования, применяется оно довольно редко. Связано это, в частности, с недостатками существующих платформ и отсутствием развитой методологической базы для их применения. Во многих случаях для создания предметно-ориентированного решения требуется привлекать экспертов в создании языков, которыми зачастую оказываются авторы выбранной для реализации этого решения DSM-платформы, поэтому позволить себе это могут лишь крупные компании. Такая ситуация указывает на необходимость продолжения исследований в этой области, что и стало предметом данной работы.

**Целью** диссертационной работы является исследование предметно-ориентированных визуальных языков, их свойств, процесса их разработки, создание методологии разработки предметно-ориентированных решений, достаточно простой в применении, чтобы свой предметно-ориентированный язык и инструментальные средства для него мог создавать даже человек, не имеющий опыта и специальной подготовки в создании визуальных языков. Также требуется создать технологию, эту методологию реализующую, которая позволяла бы неспециалистам создавать свои предметно-ориентированные решения в короткие сроки без специальных знаний и специальной подготовки.

---

<sup>5</sup>Далее мы будем именовать визуальный редактор и инструментальные средства для работы с предметно-ориентированным языком „**Предметно-ориентированное решение**“)

<sup>6</sup>Домашняя страница Eclipse Modeling Project, URL: <http://www.eclipse.org/modeling/>

<sup>7</sup>Домашняя страница MetaEdit+, URL: <http://www.metacase.com/products.html>



Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. провести обзор подходов к реализации визуальных языков и средств, их реализующих;
2. разработать в рамках DSM-платформы QReal простую в использовании технологию создания предметно-ориентированных языков;
3. реализовать разработанную технологию в виде промышленного продукта;
4. провести апробацию технологии путём создания нескольких DSM-решений с её помощью.

**Объектом исследования** являются визуальные языки, методы их создания и технологии для разработки инструментальных средств визуальных языков.

В качестве **методов исследования** используются: теория формальных языков, теория графов, методы объектно-ориентированного программирования.

**Научная новизна** данной работы заключается в следующем:

1. Разработана новая методика и набор инструментальных средств для создания предметно-ориентированных языков с помощью графического языка метамоделирования и сопутствующих визуальных языков. Методика предполагает применение предметно-ориентированного подхода „самого к себе“, то есть предметно-ориентированные языки используются для описания всей функциональности разрабатываемых инструментальных средств для нового языка: редактора диаграмм, генераторов кода на текстовых языках по диаграммам, интерпретаторов, средства проверки ограничений на диаграммы, средства поддержки рефакторингов.
2. Предложен новый способ создания предметно-ориентированного языка: „метамоделирование на лету“. Способ предполагает изменение и дополнение визуального языка прямо в процессе создания диаграммы на нём, без использования отдельного метаредактора. В процессе разработки языка

при таком подходе не требуется оперировать с понятиями „метамодель“ и „метаредактор“, что снижает требования к квалификации пользователей.

3. С использованием предложенных методик разработаны новые предметно-ориентированные языки и средства инструментальной поддержки для них: язык программирования роботов и среда QReal:Robots (также известная как TRIKStudio), средство программирования приложений для мобильных телефонов QReal:Ubiq, средство разработки аппаратных систем QReal:HaSCoL.

**Практическая ценность** данной работы определяется использованием полученных результатов при разработке DSM-платформы QReal [11, 12], в ряде DSM-решений, созданных с её помощью, самым зрелым из которых стала среда программирования роботов QReal:Robots [13], предназначенная для обучения школьников основам информатики и кибернетики с использованием робототехнических конструкторов Lego Mindstorms NXT<sup>8</sup>.

Среда QReal разрабатывается в рамках деятельности научно-исследовательской группы по изучению визуального моделирования под руководством проф. А.Н. Терехова с 2007 года и базируется на более чем двадцатилетнем опыте коллектива кафедры системного программирования Санкт-Петербургского государственного университета в разработке графических языков [14–18]. Проект имеет открытый исходный код<sup>9</sup>, разрабатывается на языке C++ с использованием библиотеки Qt силами студентов и преподавателей кафедры, автор данной диссертации — один из руководителей проекта. QReal создаётся как средство визуального моделирования, поддерживающее ряд широкоизвестных визуальных языков (UML 2.0, BPMN<sup>10</sup>, блок-схемы), и одновременно как DSM-платформа, позволяющая быстро и без специальных знаний создавать свои собственные визуальные языки и DSM-решения на их основе. На данный момент среда существует в виде работающего прототипа. Проект поддержан грантом Санкт-Петербургского государственного университета 6.39.1054.2012. DSM-платформа QReal использовалась для реализации

---

<sup>8</sup>Домашняя страница робототехнического конструктора Lego Mindstorms NXT, URL: <http://www.lego.com/en-us/mindstorms> (дата обращения: 22.02.2014г.)

<sup>9</sup>Страница проекта и репозиторий с исходным кодом на GitHub, URL: <https://github.com/qreal/qreal>

<sup>10</sup>Business Process Model and Notation, URL: <http://www.bpmn.org/> (дата обращения: 22.02.2014г.)

ряда предметно-ориентированных решений, использовавшихся в проектах компании „ЛАНИТ-Терком“, связанных с разработкой информационных систем и систем компьютерного зрения.

Среда программирования роботов QReal:Robots (или TRIKStudio) — на данный момент наиболее зрелая предметно-ориентированная технология, созданная с помощью среды QReal. Условия, в которых она появилась, близки к идеальным для применения предметно-ориентированного подхода: достаточно узкая предметная область, необходимость в средствах для создания нетривиальных программ, при этом программы хорошо выражаются в терминах визуального языка. Задача заключается в следующем: в школах со времён академика Ершова для преподавания информатики используется понятие „исполнитель“ — некоторый объект, исполняющий команды, описанные в программе. В роли такого исполнителя до сих пор применяется „черепашка“ LOGO<sup>11</sup>, но она постепенно вытесняется реальными осязаемыми исполнителями — роботами, собираемыми из робототехнических конструкторов, самый популярный из которых на данный момент — Lego Mindstorms NXT. Программировать такие роботы труднее, чем „черепашку“, поскольку из набора можно собрать какую угодно конструкцию, и программировать приходится в терминах мощности и оборотов моторов, а не в командах вида „вперёд на 20 шагов“, „влево на 90 градусов“, как в „черепашке“. Поэтому (и с учётом того, что обучение информатике на этих конструкторах начинается с пятого класса) требуется представлять программу возможно более наглядно, и графические языки подходят для этой цели гораздо лучше, чем текстовые.

Первый прототип среды программирования был разработан автором данной диссертации с использованием системы QReal примерно за неделю, и включал в себя визуальный язык из примерно 20 сущностей, редактор к нему и интерпретатор, позволяющий исполнить программу на компьютере, посылая команды роботу по интерфейсу Bluetooth<sup>12</sup>. На данный момент система развилась в полноценный программный продукт, используемый во многих кружках по робототехнике в России и ближнем зарубежье.

<sup>11</sup>См., например, MyRobot, Язык программирования Лого, URL: <http://myrobot.ru/logo/aboutlogo.php>

<sup>12</sup>Спецификация беспроводных сетей ближней связи, URL: <https://www.bluetooth.org/en-us/specification/adopted-specifications> (дата обращения: 22.02.2014г.)

**Апробация работы** заключается в следующем.

- Некоторые результаты данной работы были доложены на второй научно-технической конференции молодых специалистов „Старт в будущее“ (Санкт-Петербург, 2011) [19]. Доклад был отмечен наградой.
- Результаты, связанные с применением разработанной технологии при создании среды QReal:Robots были доложены на VII Международной научно-практической конференции „Современные информационные технологии и ИТ-образование“ (Москва, 2012) [20].
- Результаты, связанные с применением разработанной технологии для разработки предметно-ориентированного языка для платформы Ubiq были доложены на международной конференции „10th Conference of Open Innovations Association FRUCT“ (Tampere, 2011) [21].
- Результаты диссертации использовались при проектировании и реализации DSM-платформы QReal и ряда DSM-решений, созданных с её помощью, включая среду программирования роботов QReal:Robots. Среда QReal использовалась для создания нескольких предметно-ориентированных решений в ЗАО „ЛАНИТ-Терком“. Среда QReal:Robots демонстрировалась на Открытых состязаниях Санкт-Петербурга по робототехнике в 2012 году и на робототехническом фестивале „Робофест 2012“ в Москве. Также она применялась для обучения школьников робототехнике в летнем робототехническом лагере в г. Сиверском в 2012 году. На данный момент эта среда переименована в TRIKStudio и используется как основное средство программирования кибернетического конструктора ТРИК<sup>13</sup>, используется в нескольких робототехнических кружках в России и на мастер-классах по робототехнике, проводимых компанией „Кибернетические технологии“.
- По теме диссертации опубликовано 3 научные работы (одна из них — в сборнике из списка ВАК, две другие — в сборнике, входящем в РИНЦ) и 7 тезисов докладов на конференциях (под авторством или в соавторстве с автором диссертации).

---

<sup>13</sup> Домашняя страница конструктора, URL: <http://trikset.com/> (дата обращения: 20.08.2014)

**Основные результаты** данной работы таковы.

1. Проведён анализ различных подходов к созданию инструментов для работы с визуальными языками и различных технологических средств, их реализующих. Выявлено отсутствие в существующих системах инструментальной поддержки самых ранних этапов разработки языка и недостаточная автоматизация последующих этапов.
2. Разработана методика и набор инструментальных средств для разработки предметно-ориентированных языков с помощью графического языка метамоделирования и сопутствующих визуальных языков.
3. Предложена новая методика метамоделирования, предполагающая расширение и уточнение предметно-ориентированного языка прямо в процессе его использования („метамоделирование на лету“).
4. Предложенные методики и технологии реализованы в виде промышленного продукта QReal.
5. Проведена апробация разработанных средств при создании системы QReal:Robots и нескольких других предметно-ориентированных решений.

Ниже приведён краткий план последующих глав диссертации.

В **Главе 1** приводятся основные понятия, используемые в предметно-ориентированном визуальном моделировании, обсуждается структура визуального языка, уровни абстракции, вводятся некоторые свойства визуальных языков, важные для дальнейшего изложения.

**Глава 2** содержит обзор существующих подходов к созданию DSM-решений: обсуждаются возможности, достоинства и недостатки существующих DSM-платформ, включая зрелые системы и академические разработки, анализируются существующие методологии создания, внедрения и сопровождения визуальных языков и DSM-решений, делаются выводы, касающиеся текущего состояния исследований в этой области.

**Глава 3** содержит описание предлагаемого подхода к разработке DSM-решений: приводятся этапы жизненного цикла DSM-решения, обсуждается возможная степень автоматизации каждого этапа, формулируются требования на

средства автоматизации, приводится описание предлагаемой технологии, включающей технику метамоделирования „на лету“.

В **Главе 4** анализируются результаты реализации инструментальных средств поддержки предлагаемой технологии в проекте QReal. Описываются возможности системы QReal, связанные с поддержкой техник метамоделирования, принятые архитектурные решения, приводятся соображения по дальнейшему развитию инструментальных средств.

**Приложение А** содержит примеры применения результатов, описанных в данной диссертации, для разработки DSM-решений. Описывается среда QReal:Robots, то, какие преимущества были получены от использования DSM-платформы QReal при её разработке, то, чем QReal помочь не смог, и почему. Также приводится описание среды разработки сервисов для мобильных телефонов QReal:Ubiq и среды разработки аппаратуры QReal:HaSCoL, описываются их визуальные языки, достоинства и недостатки использованных при их создании подходов.

**Приложение В** содержит описание визуального метаязыка системы QReal.

## Глава 1

# Визуальные языки и их свойства

В этой главе приводится контекст работы и основные понятия, используемые в дальнейшем при изложении результатов диссертации.

## 1.1 Визуальное моделирование

Идея использовать чертежи для разработки сложных систем родилась из аналогии с инженерными дисциплинами — так же как, например, при строительстве дома проект разрабатывается архитектором, а потом реализуется строителями, хотелось бы иметь возможность подготовить проект программной системы, а затем, когда все архитектурные решения уже приняты, просто его реализовать. Естественно по аналогии с инженерными дисциплинами использовать для описания архитектуры системы графические чертежи.

Однако разработка программных систем имеет ряд особенностей, делающих прямое заимствование опыта из инженерных дисциплин невозможным. Программное обеспечение незримо, поэтому изображать его можно по-разному, каждый человек может по-своему представлять себе программу. Это существенно отличает проектирование ПО<sup>1</sup> от проектирования объектов реального мира — архитектор рисует чертёж здания, который напоминает вид этого здания, когда оно будет достроено. „Чертёж“ же ПО — это всегда лишь некоторая договорённость между разработчиками о том, что и как будет изображено на „чертеже“. Такая договорённость называется *метафорой визуализации* [4, 22]. Например, при использовании языка UML для визуализации архитектуры системы мы договариваемся, что классы изображаются в виде прямоугольников,

---

<sup>1</sup>Программное обеспечение

а случаи использования — в виде овалов. Поскольку язык **UML** является де-факто стандартом индустрии и очень широко распространён, то нарисованная одним разработчиком схема будет скорее всего правильно понята другими разработчиками.

Поскольку программа не имеет какого-то внешнего вида и для её визуализации используются метафоры, то оказывается невозможным нарисовать „программу целиком“, каждая визуальная модель описывает какой-то свой аспект системы. При этом, даже если изображаемый аспект системы фиксирован, изображать его можно по-разному, используя разные уровни детализации или отображая на диаграммах разную информацию. Например, если мы хотим изобразить архитектуру системы с помощью языка **UML**, мы в зависимости от ситуации можем сделать это с помощью диаграмм компонентов или диаграмм классов, причём, если мы рисуем диаграммы для заказчика или начальства, они должны содержать меньше технических подробностей, чем диаграммы для программистов. Таким образом, каждая диаграмма рисуется с какой-то определённой целью для какой-то определённой категории людей, при этом изображая какой-то определённый аспект системы. Поэтому выделяют понятие „*точка зрения моделирования*“ [4], в которое входят все перечисленные выше соображения о назначении диаграммы. Понятие точки зрения моделирования применимо не только к конкретной диаграмме, но и к языку моделирования в целом, поэтому весьма важно для дальнейшего изложения. Каждый язык предназначен для рисования диаграмм, описывающих систему с точки зрения, свойственной этому языку. Поэтому это понятие можно с одной стороны использовать как базис классификации языков, с другой стороны, как мотивировку для создания новых языков. Следует отметить, что язык **UML** здесь рассматривается как набор взаимосвязанных языков, а не как один язык.

Следующее вводимое здесь понятие тесно связано с точками зрения моделирования — *семантический разрыв*. Визуальная модель не может содержать в себе всю информацию о системе, потому что модель — это всегда упрощение моделируемого объекта. Таким образом, модель системы, как правило, не содержит в себе всей информации, необходимой для генерации или интерпретации программы, описываемой этой моделью. То есть, среди точек зрения моделирования не бывает точки зрения исполнителя. Тот факт, что любая модель си-



стемы принципиально содержит недостаточно информации для исполнителя, и называется семантическим разрывом — разрывом между семантикой модели и семантикой исполняемой программы. Семантический разрыв делает невозможной полную генерацию кода программы по визуальной модели, либо заставляет вносить в модель столько информации, что она становится столь же сложна, что и моделируемая программа. Поэтому очень многие диаграммы на языке **UML** рисуются только как иллюстрации к технической документации, и основной объём работы по реализации системы вне зависимости от наличия визуальных моделей приходится выполнять программистам. Такая ситуация нежелательна, поскольку хотелось бы более продуктивно переиспользовать труд проектировщиков. Поэтому семантический разрыв пытаются преодолеть.

Существует класс инструментов, поддерживающих визуальное моделирование. Такие инструменты по традиции называют *CASE-системами* (или *CASE-пакетами*), хотя этот термин весьма неточен. Исторически термин „CASE“ (Computer-Aided Software Engineering) обозначал применение методов и технологических средств для разработки программного обеспечения с помощью компьютера, то есть обычные текстовые среды разработки с точки зрения такого определения — тоже **CASE**-инструменты. В таком значении этот термин давно не используется, сейчас **CASE** относится прежде всего к средствам разработки программного обеспечения, использующим визуальные модели. **CASE**-системы могут покрывать как весь цикл разработки программного обеспечения, так и отдельные его этапы. Для обозначения первой категории **CASE**-систем используется термин *I-CASE* (*Integrated CASE*), такие системы имеют тенденцию включать в себя всё необходимое для разработки **ПО**, от средств анализа требований до средств автоматизации тестирования и средств управления проектом, при этом интегрируются с компиляторами, отладчиками, профилировщиками и другими требуемыми для разработки инструментами. Средства из второй категории традиционно подразделяют на средства поддержки первых этапов водопадной модели жизненного цикла **ПО** (анализа и проектирования), называемые *Upper CASE*, и нижних этапов этой модели (реализации и тестирования), называемые *Lower CASE*. Исторически первые **CASE**-системы (например, PSL/PSA [23]) относились к категории Upper CASE, поскольку автоматизировали исключительно анализ требований, затем (в 80-х и начале 90-х годов 20-го

века) широкое распространение получили I-CASE-средства. Связано это с тем, что в те времена основной объём разрабатываемых программных продуктов приходился на программное обеспечение для мэйнфреймов, автоматизирующее бизнес-процессы крупных компаний. Там CASE-средства играли роль интегрированных средств разработки, в которых писалось всё программное обеспечение целиком, и они интегрировались со всеми остальными необходимыми средствами разработки, доступными для нужной платформы. Наиболее популярные современные CASE-средства, как правило, ориентированы на автоматизацию этапов анализа и проектирования, таким образом, относятся к Upper CASE по данной классификации.

Современные CASE-системы, как правило, состоят из большого числа компонентов. Наиболее важный из них — репозиторий, база данных, хранящая в себе всю информацию о разрабатываемой системе, с репозиторием работают все остальные компоненты. Вводится информация в репозиторий посредством редакторов, которые могут быть визуальными редакторами диаграмм, текстовыми редакторами, редакторами таблиц и т.д. Впоследствии репозиторий могут использовать генераторы, которые по модели системы генерируют код на текстовых языках (как правило, фрагменты программы, например, объявления классов без реализаций) или документацию, валидаторы, проверяющие корректность, целостность и полноту содержимого репозитория, интерпретаторы и отладчики, использующие хранящуюся в репозитории модель системы для непосредственного исполнения. Кроме того, в состав CASE-систем могут входить вспомогательные средства, такие как текстовый редактор (для работы со сгенерированным кодом или документацией), редактор экранных форм, средства управления проектом, средства интеграции с системами контроля версий, средства импорта и экспорта моделей и т.д.

## 1.2 Структура визуальных языков

Визуальные языки могут применяться и без какой-либо инструментальной поддержки, например, для рисования набросков архитектуры системы, моделей предметной области, требований и т.д. Используемые при этом языки не нуждаются ни в какой формализации и никак не определяются, иногда диа-

граммы на таких языках могут сопровождаться легендой, поясняющей значение символов. Такие языки встречаются очень часто — считается, что наиболее полезными диаграммами оказываются нарисованные „на салфетке“ наброски в ходе первого продумывания структуры будущей системы. Более того, они очень часто встречаются и в далёких от разработки программного обеспечения областях: например, схему метро можно рассматривать как диаграмму на некотором неформальном визуальном языке, которая при этом снабжена легендой, поясняющей значение типов символов на этой „диаграмме“ (забегая несколько вперёд, можно сказать, что эта легенда является аналогом метамодели визуального языка). Таким образом, от таких языков требуется только, чтобы диаграммы на них были понятны тем, для кого они предназначены.

Необходимость в формальном описании визуального языка возникает, когда для него требуется инструментальная поддержка. Для того, чтобы создать редактор диаграмм, требуется знать допустимый набор символов языка и правила, по которым эти символы могут комбинироваться в диаграммы. То же касается генераторов кода, валидаторов, интерпретаторов и всех остальных компонентов **CASE**-системы, которым важна семантика моделей, причём каждому инструменту может быть необходима своя часть формализации языка: генераторам кода важны проекции из символов визуального языка в строки текста, валидаторам важны формальные ограничения на модели, интерпретаторам — семантика языка. В связи с тем, что инструментальные средства для визуальных языков разрабатываются и используются весьма активно, вопросы формализации описания языка довольно хорошо проработаны, далее приводятся основные понятия, при этом используемые.

### 1.2.1 Синтаксис, семантика и прагматика

Любые языки, не только визуальные, но даже естественные языки, состоят из трёх компонентов — синтаксиса, семантики и прагматики [4]. Синтаксис описывает правила построения текстов на языке из знаков языка, семантика описывает значение текстов на языке (то есть его связь с предметной областью), прагматика описывает способы использования языка его пользователем.

*Синтаксис* визуального языка описывает правила, по которым из элементов языка составляются модели. Хочется заметить, что синтаксис определяет не множество корректных моделей, а структуру модели и её внешний вид, корректная с точки зрения синтаксиса языка модель может быть бессмысленной с точки зрения его семантики. Тем не менее, синтаксис языка часто является „первым барьером“ для ошибок, и усложняя синтаксис языка можно добиться уменьшения количества в том числе и семантических ошибок, допускаемых пользователями этого языка. Для визуальных языков синтаксис делится на три составляющие: абстрактный, конкретный и служебный синтаксисы.

*Абстрактный синтаксис* языка определяет логическую структуру моделей на этом языке. Как правило, абстрактный синтаксис содержит перечисление всех элементов языка, их свойства, правила соединения элементов в диаграммы. Например, для языка **UML** абстрактный синтаксис определяет, что существует элемент „класс“, у него есть свойства „имя“, „абстрактность“ и т.д., он имеет список операций, свойств, надклассов и т.д.

*Конкретный синтаксис*, также называемый *нотация*, определяет правила отображения элементов языка. Примером в случае языка **UML** может служить конкретный синтаксис элемента „класс“, который изображается прямоугольником с тремя секциями — название класса, поля и методы, при этом две последние не обязательны. Один и тот же элемент абстрактного синтаксиса может иметь несколько различных нотаций. Различия могут быть весьма существенными, например, в языке **UML** свойства класса могут быть изображены как текстовые поля внутри фигуры класса (например, `+name : String = ‘Temp’`), либо как ассоциации, связывающие класс с классами, которые являются типами полей. С точки зрения абстрактного синтаксиса два этих варианта неразличимы, но диаграммы, нарисованные в двух этих вариантах, могут радикально отличаться визуально.

*Служебный синтаксис*, или *синтаксис сериализации* определяет способ хранения диаграмм на диске. Обычно используется какой-либо текстовый язык на основе **XML**<sup>2</sup> для записи свойств элементов модели. Существует стандарт

---

<sup>2</sup>Extensible Markup Language. Стандарт 1.0, URL: <http://www.w3.org/TR/REC-xml/> (дата обращения 21.08.2014г.).

ный язык XMI (Xml Metadata Interchange), стандартизованный группой OMG<sup>3</sup>, который позволяет стандартным (и следовательно, переносимым) способом сериализовать визуальные модели на UML и подобных языках. Этот стандарт позволяет вместе с диаграммой на визуальном языке хранить и описание этого языка, что даёт возможность обмениваться моделями между инструментами, поддерживающими XMI, даже если они не поддерживают тот язык, на котором нарисована диаграмма. XMI не стандартизует аспекты, связанные с конкретным синтаксисом языка, таким образом, модель может передаваться между инструментами, а диаграммы по этой модели в каждом инструменте, скорее всего, придётся строить заново.

Следует отметить, что такое разделение синтаксиса на виды применимо и к текстовым языкам. Абстрактный синтаксис характеризует структуру программы, и определяет абстрактное синтаксическое дерево, которое строит компилятор, без привязки к конкретным лексемам. Конкретный синтаксис — это запись лексем языка, например, лексема начала блока операторов в языке может выглядеть как „{“, а может как „begin“. Абстрактное синтаксическое дерево в том и в другом случае выглядит одинаково, но вид программы будет разный. Особый служебный синтаксис текстовым языкам, как правило, не нужен, программы на них хранятся так же, как и изображаются, обычным текстом.

*Семантика* языка определяет значение его конструкций, то есть, в случае визуальных языков программирования, то, как будет интерпретироваться визуальная модель или какой код будет по ней сгенерирован. Способов задания семантики много, наиболее часто встречаются *денотационная* семантика (задание семантики языковых конструкций через отображение их в некоторые объекты с уже определённой семантикой) и *операционная* семантика (определение формальных преобразований над конструкциями языка, которые и определяют процесс вычислений). На практике используются разные способы задания семантики визуальных языков, их рассмотрение выходит за рамки данной работы.

---

<sup>3</sup>Object Management Group, URL: <http://www.omg.org/> (дата обращения: 14.02.2014г.), стандарт XMI можно найти здесь, URL: <http://www.omg.org/spec/XMI/> (дата обращения: 21.08.2014г.), также XMI стандартизован ISO как стандарт ISO/IEC 19503:2005

*Прагматика* описывает взаимодействие языка и его пользователя. Это понятие никак не формализуется, и обычно прагматика языка определяется просто исходя из здравого смысла его автора. Тем не менее, существуют и довольно обширные научные исследования в области удобства использования визуальных языков, например, [24].

### 1.2.2 Уровни моделирования

Абстрактный синтаксис языка — наиболее важный подвид синтаксиса с точки зрения различных инструментов, поэтому он должен быть описан максимально формально. Для его описания обычно используется метамоделирование — техника, при которой синтаксис визуального языка описывается с помощью некоторого другого, как правило, визуального языка, называемого *метаязыком*. Модель на метаязыке, специфицирующая синтаксис языка, называется *метамоделью* этого языка. Связь между моделями и метамоделями проиллюстрирована на рисунке 1.1.

*Предметная область* — это то, что мы моделируем, те объекты или явления реального мира, с которыми будет работать создаваемое программное обеспечение. Предметная область просто существует и сама по себе никак не описывается, любое её описание её упрощает, а значит, будет её моделью. В нашем примере мы хотим создать программу — каталог фильмов, предметная область для такой программы будет включать в себя фильмы и всё, что с ними связано — актёров, режиссёра, кассовые сборы, где и когда фильм снимался и т.д. и т.п. Большая часть этой информации для наших целей не нужна.

*Модель* — это некое упрощение предметной области, нужное для выполнения там некоторых полезных действий. Предметная область, как правило, обладает бесконечным многообразием, и работать со всеми её свойствами невозможно. В нашем примере про каталог фильмов нас не интересуют, например, все кино-театры, где показывали фильм, нам интересен только тот ограниченный объём информации, с которым будет работать создаваемая программа. Для создания модели уже используется некий язык. Если это неформальное описание предметной области, в качестве языка может выступать естественный язык, если это некая диаграмма, она будет рисоваться на некоем визуальном языке. Для


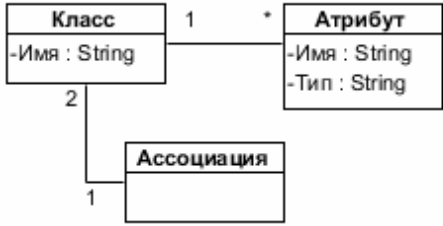
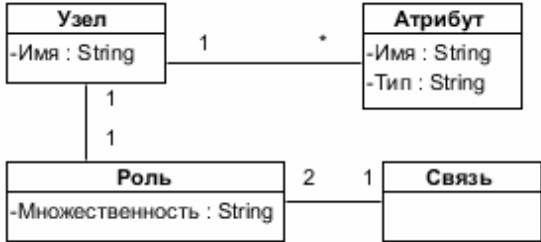
Уровни моделирования	Языковые средства	Пример
Предметная область	Нет	Каталог фильмов
Модель	Визуальный язык	Диаграмма классов 
Метамодель	Метаязык	Метамодель диаграммы классов 
Метаметамодель	Метаязык	Метамодель метаязыка 

Рисунок 1.1: Метауровни визуальных языков.

нашего примера был использован некий сильно упрощённый вариант языка диаграмм классов **UML**.

*Метамодель* — это описание визуального языка, то есть множества всех синтаксически корректных диаграмм на этом языке, с помощью некоего другого визуального языка. Этот язык называется метаязыком, и описывает множество всех элементов моделируемого языка и возможные связи между ними. Каждый элемент диаграммы на визуальном языке является экземпляром некоторой сущности, описанной в метамодели на метаязыке. В нашем примере для моделирования предметной области был использован очень простой язык с сущ-



ностями „класс“ и „атрибут“, и одной связью, которая может только связывать два класса и больше никакой информации не содержит. Атрибут „Название“ на диаграмме классов является экземпляром сущности „Атрибут“ в метамодели.

*Метаметамодель* — это метамодель метаязыка, то есть визуальная модель, которая задаёт множество допустимых метамodelей. Имея такую формализацию метаязыка, можно создавать инструменты, которые позволяют разрабатывать метамodelи для произвольных визуальных языков. Интересно, что термин „метаметаязык“ как язык, который используется для создания метаметамodelи, не требуется — метаязык сам по себе является визуальным языком и поэтому, если он достаточно выразителен, может быть использован для создания своей собственной метамodelи. Такой подход, в частности, применён в стандарте **UML** — сначала вводится метаязык в терминах самого себя (при этом авторам стандарта пришлось приложить некоторые усилия, чтобы избежать круговых зависимостей в определении), затем с его помощью описывается метамodelь **UML**. В нашем примере метаязык очень прост, состоит из сущностей, называемых в метаметамodelи „узлами“, и связей. Связи здесь более сложные, чем в языке, потому что мы хотим использовать множественность для задания дополнительных ограничений на модели. Например, у класса может быть сколько угодно атрибутов, но у атрибута может быть только один класс.

Проводя аналогию с текстовыми языками, модель можно сопоставить программе на текстовом языке, метамodelь — описанию грамматики языка (например, с помощью формы Бэкуса-Наура), метаметамodelь — описанию синтаксиса формализма, используемого для описания грамматики (например, описание синтаксиса форм Бэкуса-Наура, которое тоже можно выполнить с помощью форм Бэкуса-Наура). Заметим, что метаязыков может быть много, ведь синтаксис текстовых языков может быть описан с помощью различных формальных систем, например, с помощью различных языков описания синтаксиса генераторов синтаксических анализаторов, используемых для разработки компилятора этого языка. С визуальными языками ситуация аналогична.



## 1.3 Предметно-ориентированное моделирование

### 1.3.1 Понятие предметно-ориентированного моделирования

Формализмы для задания синтаксиса визуальных языков позволяют эффективно создавать новые визуальные языки. Это оказывается полезно не только международным группам, занимающимся стандартизацией широко распространённых языков визуального моделирования, но и небольшим группам разработчиков, создающим визуальные языки для своих проектов. Оказалось, что иногда создать специальный язык и решить на нём поставленную задачу можно проще и эффективнее, чем решать задачу на языке общего назначения. Особенно это верно в том случае, если имеется набор похожих задач, которые можно решать с помощью одного и того же языка. Такой подход получил название *предметно-ориентированное моделирование* (*Domain-Specific Modeling, DSM*). Основным принцип данного подхода состоит в том, что выбирается достаточно узкая предметная область или даже одна задача, конкретно для неё создаётся свой язык (*предметно-ориентированный язык*, или *Domain-Specific Language, DSL*), и дальше задачи решаются уже на нём. Поскольку предметная область узка, то средства инструментальной поддержки созданного языка могут использовать знания о предметной области, что позволяет, в частности, добиться полной кодогенерации. Кроме того, разработчики, использующие язык, работают в терминах, максимально приближенных к предметной области, так что программы получаются простыми и понятными даже людям, далёким от программирования.

Получаемые благодаря применению этого принципа преимущества по сравнению с использованием языков общего назначения таковы.

- Программирование ведётся в терминах предметной области, вся рутинная работа выполняется инструментальными средствами, что значительно повышает эффективность труда разработчиков.
- Узость выбранной предметной области позволяет обеспечить полную генерацию кода по описаниям на предметно-ориентированном языке. Генерация

ратор обладает знаниями о предметной области и использует их, чтобы преодолеть семантический разрыв — сама программа остаётся простой, генератор тоже может быть устроен довольно просто, но всё вместе позволяет сгенерировать любую нужную программу.

- Наличие полной кодогенерации позволяет разработчикам вообще не работать с кодом. Основным артефактом, с которым ведётся работа, являются модели на предметно-ориентированном языке, разработчики могут даже не догадываться о том, что их программы генерируются в код на текстовом языке. Иногда предметно-ориентированное решение устроено так, что генерация в текстовый язык и не требуется (например, модели интерпретируются).
- Полная кодогенерация позволяет избежать ошибок кодирования, единственным источником синтаксических ошибок в сгенерированном коде могут быть ошибки в генераторе. При этом исправления и улучшения генератора будут автоматически применимы ко всем программам, его использующим. Кроме того, поскольку инструментальные средства „знают“ про предметную область, возможно создание верификаторов, проверяющих и семантические ошибки в моделях прямо в процессе моделирования.

Естественным образом получается разделение труда между специалистами, создающими инструментальные средства, и специалистами, их использующими. Первая группа должна состоять из весьма квалифицированных специалистов, но весьма небольшого их числа, поскольку создать инструментальные средства требуется лишь единожды. Вторая группа может состоять из людей, даже вообще не владеющих программированием, и их обычно оказывается гораздо больше, чем разработчиков инструментальных средств. Поскольку предметно-ориентированный язык работает в терминах, близких к предметной области, программирование на нём могут осуществлять даже конечные пользователи.

Всё это, по ряду оценок (см., например, [8–10]) увеличивает эффективность труда разработчиков от трёх до десяти раз, поэтому предметно-ориентированное моделирование — интересная область для исследований.

Существуют широко распространённые текстовые предметно-ориентированные языки, успех которых подтверждает сделанные в этом

разделе заявления. Самый известный пример — язык запросов к базам данных SQL. Предметная область этого языка ограничивается работой с СУБД, она достаточно узка, чтобы пользователь мог работать в терминах, очень напоминающих естественный язык, и при этом все необходимые действия, которые было бы весьма сложно описывать на языке общего назначения, выполняются самой СУБД. Пользователи языка SQL вполне могут не уметь программировать на текстовых языках общего назначения, таких как Java и C#.

Второй пример семейства предметно-ориентированных языков — языки описания синтаксически управляемых трансляций для различных генераторов синтаксических анализаторов (таких как уасс<sup>4</sup> и его многочисленные варианты, ANTLR<sup>5</sup>). Программа там пишется в виде, очень похожем на описание контекстно-свободной грамматики в форме Бэкуса-Наура, по ней генерируется либо магазинный преобразователь, транслирующий созданный язык, либо преобразователь, работающий по алгоритму рекурсивного спуска. Генераторы синтаксических анализаторов знают достаточно о предметной области, чтобы по простому описанию грамматики языка сгенерировать довольно сложный синтаксический анализатор, тогда как ручное его кодирование было бы очень сложной задачей (автомат, разбирающий какой-либо типичный используемый в промышленности язык программирования, может иметь сотни состояний). Наличие таких инструментов делает возможным быстрое создание предметно-ориентированных текстовых языков, поэтому генераторы синтаксических анализаторов для текстовых языков аналогичны метаредакторам для языков графических.

Приведём пример графического предметно-ориентированного языка из [7]. Программа на этом языке представлена на рисунке 1.2.

Язык предназначается для создания приложений для мобильных телефонов. Элементы языка — это экранные формы, отображаемые на телефоне, связи, показывающие порядок перехода между формами, и различные действия, например, посылка SMS-сообщения или открытие веб-страницы. На рисунке

---

<sup>4</sup>Yet Another Compiler Compiler, широко распространённый и несколько раз перереализованный генератор синтаксических анализаторов, см., например, URL: <http://dinosaur.compilertools.net/> (дата обращения: 23.02.2014)

<sup>5</sup>ANother Tool for Language Recognition, URL: <http://www.antlr.org/> (дата обращения: 23.02.2014)

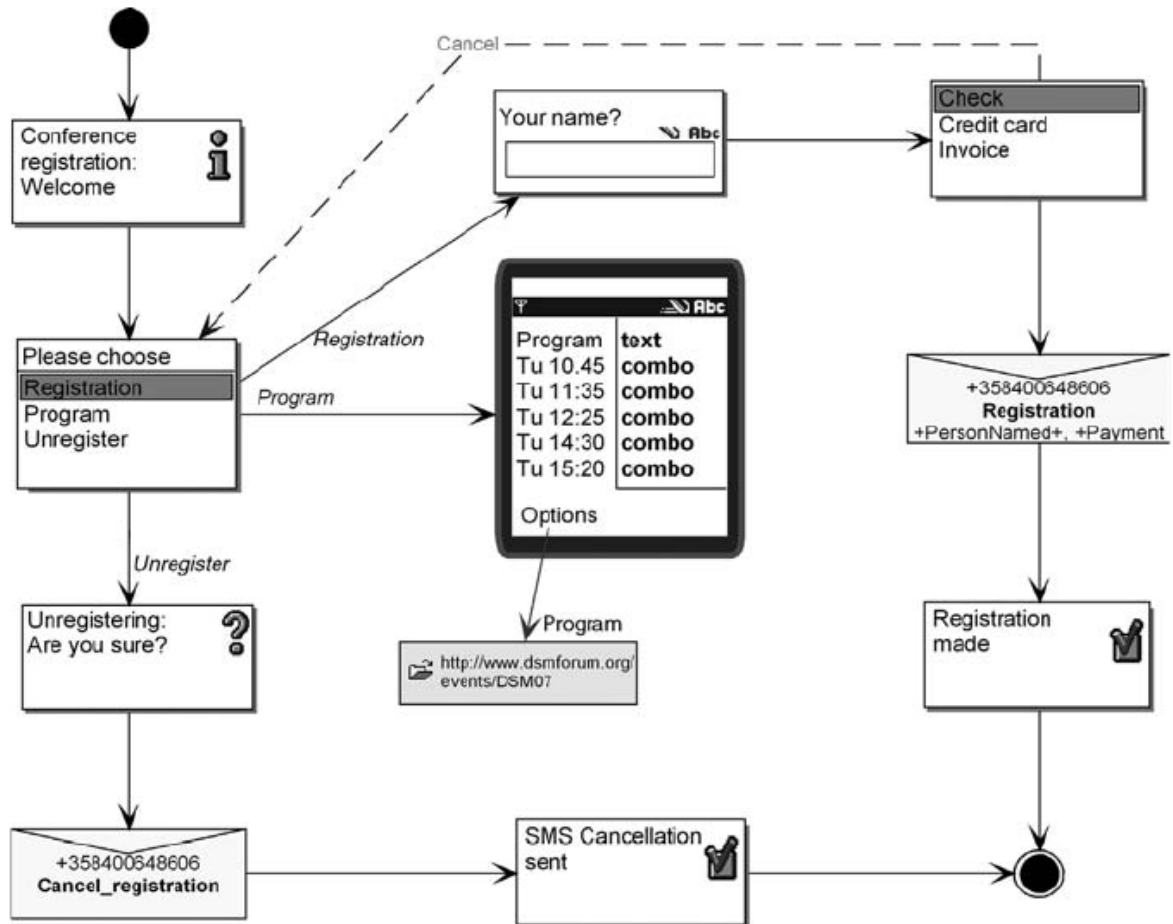


Рисунок 1.2: Пример визуального предметно-ориентированного языка  
(рисунок из [7])

представлен пример программы на этом языке — программа для регистрации на конференции. Как видно, язык достаточно прост и интуитивно понятен, настолько, что не требует подробного описания, чтобы читать диаграммы на нём. Вместе с тем, на такой диаграмме содержится вся необходимая информация для генерации мобильного приложения.

### 1.3.2 Инструментальные средства предметно-ориентированного моделирования

Визуальный предметно-ориентированный язык или набор языков, и средства инструментальной поддержки, такие как редакторы диаграмм, генераторы кода, верификаторы, репозиторий, и т.д., всё, что составляет целостную технологию разработки программного обеспечения на ос-

нове предметно-ориентированного подхода, будем называть *предметно-ориентированным решением*, или *DSM-решением*. Основные составляющие предметно-ориентированного решения таковы.

1. Предметно-ориентированный язык, отражающий специфику предметной области, и редактор для него.
2. Генераторы или интерпретаторы, извлекающие информацию из модели и формирующие по ней код, документацию, отчёты и т.д.
3. Предметно-ориентированная библиотека поддержки времени выполнения (domain-specific framework), в которую выносятся код, общий для всех программ, создаваемых с помощью DSM-решения. Она служит промежуточным звеном между сгенерированным кодом и целевой платформой, упрощая генератор и убирая дублирование кода в программах. Иногда бывает, что такая библиотека не нужна, генератор генерирует код прямо в целевую платформу, иногда бывает наоборот, библиотека представляет собой по сути движок, интерпретирующий сгенерированные файлы или даже просто конфигурируемый генератором (пример такого подхода см., например, в [25]). Возможны и промежуточные варианты.

Разумеется, создавать предметно-ориентированные решения вручную было бы слишком затратно. Один только хороший визуальный редактор вручную разрабатывается несколько десятков человеколет<sup>6</sup>, что делает невозможным исполнение главного принципа предметно-ориентированного моделирования — создание языка и всех нужных инструментальных средств к нему практически под каждую конкретную задачу. Поэтому существуют средства, предназначенные для автоматизации разработки DSM-решений, их мы будем называть *DSM-платформами*. Типичная DSM-платформа позволяет специфицировать метамодель языка в текстовом или графическом виде, задать внешний вид элементов языка (конкретный синтаксис) и сгенерировать визуальный редактор по этим спецификациям. Создание языка обычно требует усилий объёмом порядка

---

<sup>6</sup>см, например, оценки подобных проектов на ресурсе Ohloh: URL: <http://www.ohloh.net/p/dia>, <http://www.ohloh.net/p/umbrello> (дата обращения: 14.02.2014)

единиц человекоднев или даже человекочасов, что делает возможным применение предметно-ориентированного подхода даже в сравнительно небольших проектах. Многие DSM-платформы помимо визуального редактора позволяют сгенерировать генераторы, верификаторы и другие инструменты, для этого помимо метамодели для языка могут быть описаны правила генерации, набор ограничений и т.д.

## 1.4 Свойства визуальных языков

Визуальные языки обладают рядом свойств, некоторые из которых могут быть базисом для классификации языков, некоторые важны для реализации части функциональности **CASE**-пакетов и **DSM**-решений, поэтому вводятся в этом разделе. Существуют работы, специально посвящённые классификации языков, например, в [26], здесь же свойства визуальных языков будут рассмотрены из соображений важности для этой диссертации.

Наиболее важное с точки зрения реализации инструментов свойство языков — используемая в них модель представления информации. Предметно-ориентированные языки делятся на две крупные категории — *текстовые* и *графические* (или *визуальные*). В данной работе далее будут рассматриваться только визуальные языки. Про текстовые языки следует отметить, что не всегда в качестве представления для языков, попадающих в эту категорию, используется обычный текст. Возможно использование таблиц, возможно древовидное представление структуры программы, и несмотря на то, что программа выглядит как текст, для её редактирования требуется специальный редактор (см., например, среду JetBrains MPS [27]). Бывает так, что для задания текстовых языков используются визуальные метамодели, как, например, в [28], такие случаи также выходят за границы этого исследования.

Графические языки делятся дополнительно на *графовые* и *неграфовые*. Диаграмма на графовом языке представляет собой помеченный мультиграф, либо сводится к нему, то есть состоит из множества вершин и множества рёбер, которые их соединяют (см., например, [29] как пример подобной формализации). Каждая вершина или ребро может обладать различными свойствами, в том числе иметь тип, и, соответственно своему типу, отображаться по-разному.

Соответственно, редакторы таких языков работают в терминах вершин графа (называемых также *узлами* или *блоками*) и рёбер (называемых также *связями*). Неграфовые языки — языки, диаграммы на которых не обладают таким свойством. Наиболее распространены графовые языки, поскольку редакторы и генераторы для них обычно проще и, поскольку логика работы с графовыми моделями не сильно меняется от языка к языку, их реализация может быть переиспользована для других языков (что особенно важно при разработке DSM-платформ). Для каждого неграфового языка приходится обычно писать свой редактор вручную, поскольку такие языки могут быть довольно специфичны. Примерами графовых языков могут служить диаграммы классов и диаграммы активностей **UML**, примерами неграфовых языков — диаграммы последовательностей **UML** (и очень похожий на них язык **MSC** [30]), временные диаграммы **UML**, диаграммы Насси-Шнейдермана. Данная диссертация посвящена разработке визуальных языков с помощью DSM-платформ, на которых графовые языки создавать гораздо удобнее, поэтому неграфовые языки, несмотря на то, что довольно распространены и полезны на практике, выходят за её рамки. Многие из результатов, описанных в данной работе, могут быть без изменений распространены на неграфовые языки, но основной фокус в процессе исследования был сделан на графовых языках, и апробация результатов проводилась на них.

Существуют и широко распространены языки, комбинирующие свойства текстовых и графических — визуальные языки, которые внутри визуальных символов позволяют писать код на текстовом языке, эти языки мы будем называть *текстографическими* или *гибридными*. Такой подход с одной стороны совмещает наглядность визуальных языков и лаконичность текстовых языков, с другой стороны требует от пользователя знания и визуального языка, и текстовой части, при этом текстовая и визуальная части синтаксиса плохо интегрируются (возникает ряд проблем, связанных с визуализацией текстовой информации, наличие текстовых вставок может радикально снижать наглядность). Обычно в качестве текстовых вставок используется целевой язык генератора кода по диаграммам, примером такого подхода может служить разработанная на кафедре технология RTST [15, 16]. С точки зрения данной работы текст мо-



жет рассматриваться как свойство элемента языка, таким образом, такие языки могут успешно разрабатываться с использованием предлагаемой технологии.

Следующее важное свойство языков — используемая в них модель вычислений. Языки делятся по этому признаку опять-таки на две крупные категории — *статические* и *динамические*. Статические языки служат для задания структуры разрабатываемой системы, а динамические языки служат для задания поведения системы. Примером статического языка может служить диаграмма классов **UML** или **ER**<sup>7</sup>-диаграмма, описывающая структуру базы данных, примером динамического языка — диаграмма активностей или диаграмма последовательностей **UML**. Динамические языки, в свою очередь, характеризуются формализмом, используемым для определения их семантики, поскольку для них применимо собственно понятие „вычисление“. Наиболее типично применение формализма, являющегося вариантом сетей Петри (граф, по которому могут перемещаться токены исполнения, подробный обзор сетей Петри можно найти в [31]), либо диаграмм состояний (наиболее известный вариант которых — диаграммы Харела [32], используемые в языке **UML**).

Языки, использующие формализмы, сводящиеся к сетям Петри, имеют две важные с практической точки зрения подкатегории. Первая подкатегория языков в качестве токенов использует данные, которые обрабатываются программой. Каждый узел языка исполняется, когда имеет на всех своих входах данные, нужные ему для работы, и результатом его исполнения является набор данных, которые рассылаются на выходы блока. Блоки, одновременно имеющие все данные, могут исполняться параллельно. Такие языки мы будем называть *языками с процессом вычислений, ориентированным на поток данных*. Такие языки широко распространены среди инженеров и математиков, примерами сред, реализующих такой подход, являются среда математических расчётов Matlab/Simulink<sup>8</sup> и среда моделирования физических приборов LabVIEW<sup>9</sup>. Вторая подкатегория языков в качестве токенов использует нетипизированные токены без дополнительных свойств, характеризующие только передачу управления между узлами. Данные при таком подходе не покрываются формализ-

---

<sup>7</sup>Entity-Relationship

<sup>8</sup>Домашняя страница Simulink, URL: <http://www.mathworks.com/products/simulink/index.html> (дата обращения: 03.09.2014г.)

<sup>9</sup>Домашняя страница LabVIEW, URL: <http://www.ni.com/labview/> (дата обращения: 03.09.2014г.)



мом, описывающим семантику языка, и если всё-таки требуются для работы узла, то его поведение может быть не определено, если нужные данные недоступны. Параллельное исполнение в таких языках моделируется явно, порождением нескольких токенов. Такие языки мы будем называть *языками с процессом вычислений, ориентированным на поток управления*. Эти языки широко распространены в программной инженерии, примеры — диаграмма активностей **UML** и обычные блок-схемы.

Далее введём свойство, специфичное для данной работы: существенность для семантики языка того, как именно выглядят диаграммы. Для пояснения этого потребуются ещё два понятия — логическая и графическая модели. *Логическая модель* — часть модели системы, которая существенна для функционирования системы, графическая модель — часть модели, которая имеет значение для представления модели пользователю в процессе разработки системы. То есть, логическая модель — это то, что важно для генератора, графическая модель — то, что видит пользователь. Соответственно, свойства сущностей языка, которые относятся к логической модели (важны для генератора) назовём *логическими свойствами*, свойства, относящиеся к графической модели — *графическими свойствами*. Назовём *существенно графическими языками* языки, в которых графическая модель оказывает существенное влияние на функционирование системы, то есть внешний вид диаграмм, геометрическое расположение на них элементов и т.д. оказывают влияние на их семантику (например, на результат работы генератора). *Языками с выделенной логической моделью* будем называть языки, для которых это не так. Для таких языков диаграммы как правило являются лишь средством редактирования логической модели, и логическая модель может существовать вообще независимо от представления. Пример существенно графического языка — временная диаграмма **UML**, там временные интервалы определяются длиной линий. Пример языка с выделенной логической моделью — язык диаграммы классов **UML**, модель классов системы может быть произвольно разбита на диаграммы, классы на диаграмме могут располагаться произвольным образом, важны только их свойства и то, с какими элементами они связаны (при этом, расположение связей и даже то, изображены они на диаграмме или нет, не имеет значения). Графовые языки обычно имеют выделенную логическую модель, неграфовые языки име-

ют тенденцию быть существенно графическими, однако бывают исключения. Например, автором данной работы был реализован язык описания конечного автомата, распознающего жесты рукой для системы компьютерного стереозрения. Там состояния было удобно связывать с физическим расположением руки в пространстве, и от геометрического расположения на диаграмме элементов языка зависела генерация правил перехода. Язык был графовым (узлами были состояния автомата, связями — переходы), но при этом существенно графическим.

Отметим, что языки с выделенной логической моделью позволяют использовать один и тот же элемент логической модели на разных диаграммах или в разных местах одной диаграммы. Если редактор для этого языка позволяет, можно редактировать логические свойства элемента у любого из его графических „образов“, и все остальные „образы“ получают эти изменения автоматически.

## Глава 2

# Существующие подходы к созданию DSM-решений

В этой главе рассматриваются существующие результаты, схожие с полученными в рамках данной диссертации, приводится позиционирование данной работы относительно этих результатов.

## 2.1 Фокус и структура обзора

Поскольку основным предметом исследований в данной работе является методология создания визуальных предметно-ориентированных языков, то обзор будет состоять из, во-первых, рассмотрения существующих теоретических работ, связанных с методологией, и во-вторых, обзора существующих инструментальных средств для создания визуальных языков.

Как ни странно, вопросам методологии создания визуальных языков уделялось внимание значительно меньшее, чем текстовых. В случае с текстовыми предметно-ориентированными языками существуют подробные и обстоятельные работы, описывающие как жизненный цикл языка, так и основные деятельности, которые должны быть выполнены на каждом этапе жизненного цикла (такие, как [33] и обзор [34]), в случае с визуальными языками, как правило, имеется лишь набор слабо структурированных советов, рекомендаций и наблюдений, полученных из практики (наиболее подробное изложение можно найти в [7], также хорошим примером таких работ может послужить [35] или [36]). Поэтому в обзор войдут и статьи, относящиеся к текстовым языкам — вопросы реализации предметно-ориентированных средств разработки для текстовых языков для данной работы нерелевантны, но общая модель жизненного цик-

ла языка, деятельности по анализу предметной области, некоторые аспекты проектирования языка и им подобные могут быть переиспользованы при разработке визуальных языков практически без изменений. Разумеется, нам будут интересны и статьи, содержащие опыт разработки визуальных языков, пусть и слабоструктурированный, поскольку они могут послужить базисом для создания методологии разработки. По этим же причинам в обзор будут включены статьи, в которых обсуждаются аспекты организации языковой инфраструктуры (например, [37]) — различные точки зрения на внутреннюю структуру и устройство визуальных языков могут приводить к необходимости выполнения разных действий при их создании.

Также значительное внимание в обзоре будет уделено существующим инструментальным средствам, аналогам системы QReal. Поскольку данная работа посвящена процессу создания визуальных языков, особенности инструментальных средств, к этому процессу непосредственно не относящиеся, рассматриваться не будут. Большинство имеющихся описаний инструментальных средств не фокусируются на какой-либо методологии создания визуального языка (представляя средства, но не указания по их использованию). В тех редких случаях, когда это не так, и инструментальное средство имеет за собой чёткую методологию его применения (как, например, в работе [38]), такое описание будет рассмотрено в первой части обзора, вместе с работами, излагающими общие методологические указания.

Существующих инструментальных средств создания предметно-ориентированных визуальных языков много, однако опыт в этой области программной инженерии практически не переиспользуется (что отмечают и авторы [7]). Например, в обзоре будет упомянута система, публикация по которой датируется 1977 годом, имеющая все признаки типичной DSM-платформы [23], тогда как сам термин „предметно-ориентированное моделирование“ (Domain-Specific Modeling) начал упоминаться в литературе начиная примерно с 1995 года, причём до сих пор данный подход во многих работах преподносится как новый, активно развивающийся и перспективный. Поэтому средства будут рассматриваться не в хронологическом порядке, а в порядке частоты их упоминаний в литературе, от наиболее часто упоминаемых к менее известным, насколько может судить автор данной работы. Автору известно порядка 40

подобных средств, но, в силу ограниченности объёма диссертации, будут рассмотрены только наиболее известные, а также особенно интересные в контексте данной работы.

В данном обзоре не будут рассмотрены работы, доказывающие экономическую целесообразность применимости предметно-ориентированного подхода и визуального моделирования вообще, ссылки на них читатель может найти во введении к диссертации. Кроме того, не будут рассмотрены работы, в которых вводятся различные определения, относящиеся к визуальным предметно-ориентированным языкам, классификации языков и т.д., имеющие лишь вспомогательное значение для диссертации, ссылки на такие работы содержатся в главе 1.

## **2.2 Существующие методики и приёмы разработки предметно-ориентированных языков**

### **2.2.1 Модели жизненного цикла языка**

Начать следует с общих вопросов жизненного цикла предметно-ориентированных языков. В данном случае не имеет особого значения, визуальные это языки или текстовые, поскольку различия между ними играют существенную роль лишь при реализации. Наиболее обстоятельное, насколько нам известно, обсуждение данных вопросов приводится в статье [33]. Авторы выделяют следующие фазы существования предметно-ориентированного языка: принятие решения (о необходимости создания DSL), анализ (предметной области), проектирование (переиспользование языка общего назначения для создания DSL и спецификация DSL), реализация и развёртывание. Рекомендации по деятельности коллектива разработчиков на каждой фазе даны в виде паттернов с примерами применения в реальных проектах.

В фазе принятия решения наиболее интересными для нас показателями применимости DSM-подхода могут служить:

- наличие сложного интерфейса прикладных программ (API), который можно было бы представить сущностями созданного языка;
- наличие необходимости (или возможности) предметно-ориентированного анализа, верификации, оптимизации, параллелизации и трансформации (авторами используется акроним AVOPT, образованный первыми буквами английских эквивалентов перечисленных терминов), которые требуют знания особенностей предметной области и поэтому неосуществимы на языке общего назначения (авторы используют термин GPL, General Purpose Language);
- наличие семейства программных продуктов, имеющего довольно много общих частей и некоторые различия, которые могут быть выражены посредством DSL;
- представление сложных структур данных и работа с ними.

На фазе анализа в основном применяются неформальные методы, хотя авторы упоминают несколько существующих формальных методологий (перечислены методологии DARE, DSSA, FAST, FODA, ODE, ODM). Авторы отмечают отсутствие поддержки анализа предметной области в имеющихся инструментальных средствах, и из дальнейшего обзора станет ясно, что с визуальными предметно-ориентированными языками ситуация обстоит аналогичным образом. Существуют отдельные инструменты, поддерживающие формальный анализ предметной области, но ни один из них не интегрирован со средствами разработки языков, хотя анализ предметной области является входом для наиболее важной при создании предметно-ориентированного языка деятельности — выделения основных сущностей языка.

Паттерны фаз проектирования и реализации, описанные в статье, оказываются если не полностью неприменимы к визуальным языкам, то достаточно далеки от применяемых в случае визуальных языков подходов, чтобы было необходимо рассматривать другие источники. Действительно, авторы выделяют переиспользование языка общего назначения при создании DSL как один из возможных подходов к проектированию, и для визуальных языков это тоже может быть применимо, например, использование механизмов расширения язы-

ка UML, но это отдельная тема, которая заслуживает отдельного обсуждения. Подходы, типичные для этих фаз, будут обсуждаться далее при рассмотрении более подходящих для этого источников.

В статье совершенно не рассматриваются аспекты существования языка после его развёртывания, что довольно странно, поскольку редко бывает, чтобы язык был сразу же создан таким, каким его было бы удобно использовать. Эволюция предметно-ориентированного языка требует особого рассмотрения, поскольку должна осуществляться совместно с созданными на этом языке моделями (или программами), и её нельзя исключать из модели жизненного цикла. Интересные в контексте данной работы выводы, сделанные в статье — существующие инструментальные средства фокусируются в основном на вопросах реализации и либо не предоставляют вообще средств автоматизации других фаз жизненного цикла, либо эта поддержка весьма ограничена.

Работа [34] интересна сама по себе как подробная библиография, посвящённая текстовым предметно-ориентированным языкам, покрывающая вопросы терминологии, целесообразности использования DSL, методологии создания, реализации DSL, и содержащая ссылки на массу примеров. Для нас существенны вопросы методологии. Авторы делят фазы жизненного цикла языка на три группы — анализ, реализация и использование. Этап анализа, как и в предыдущей работе, предполагает применение неформального подхода или одной из формальных методик (здесь их применение называется инженерией предметной области, или domain engineering), кроме того, в качестве источника информации для анализа указываются унаследованные системы (legacy systems) либо существующие библиотеки и каркасы приложений. Реализация рассмотрена исключительно для текстовых языков, информации по последнему этапу жизненного цикла — использованию языка — не представлено.

Работы [39] и [40] рассматривают процесс разработки языка с менее технической точки зрения. Предлагается в качестве модели для разработки DSM-решения в небольших и средних компаниях использовать методологию Microsoft Solution Framework (MSF), несколько адаптированную под специфику DSM-подхода. Автор выделяет фазы выработки концепции (envisioning), планирования, разработки, стабилизации, внедрения и эксплуатации и сопровождения. При этом фаза планирования включает в себя выбор технологии,

разработку первой версии метамодели, после чего фаза разработки состоит из нескольких циклов, состоящих из уточнения требований, модификации метамодели, генерации редакторов и ручного кодирования, демонстрации и сбора обратной связи. Стабилизация включает в себя интеграцию, пилотное внедрение и доработку, после чего, после этапа внедрения, DSM-решение переходит в фазу сопровождения. В статье мало внимания уделяется реализационным аспектам, поэтому предлагаемый способ организации процесса разработки может быть использован совместно с другими подходами.

Интересная методология разработки предметно-ориентированных визуальных языков предлагается в работе [38]. Поскольку необходимыми знаниями о предметной области обладают только эксперты, как правило, не обладающие навыками создания языков, авторы предлагают коллаборативный подход к разработке — эксперт предметной области и эксперт по созданию языков работают вместе. Модель жизненного цикла в этом случае — последовательность итераций, каждая из которых состоит из фаз использования, „поломки“ (breakdown), обсуждения (negotiation), модификации и тестирования. Сначала пользователь пытается что-то сделать на текущей версии языка, доходит до того места, где у него что-то не получается или кажется неудобным, при этом разработчик языка сидит рядом и наблюдает за действиями пользователя. Потом разработчик и пользователь обсуждают необходимые изменения в языке, разработчик прямо на глазах у пользователя вносит изменения, при этом имея перед глазами созданную пользователем модель, тестирует изменения и передаёт управление снова пользователю, начиная следующую итерацию. Такой подход требует специальной поддержки со стороны инструментария, ведь если на внесение изменений в язык требуется слишком много времени, пользователь не сможет ждать. Описываемый в статье инструментарий позволяет удобно переключаться между режимами использования и разработки языка, при этом, поскольку используемые языковые концепции довольно просты, вносить изменения удаётся „на лету“. Для задания семантики языка требуется программирование на языке, похожем на LISP, поэтому сами пользователи создавать для себя язык всё-таки не могут.

Некоторые указания по методологии разработки представлены также в книге [7]. Предлагаемая методология состоит из четырёх фаз: предварительная



проработка, разработка, внедрение и сопровождение. При этом на фазе предварительной проработки предлагается начать с *proof of concept* — реализации небольшой части языка для наиболее изученного фрагмента предметной области, прежде всего, для того, чтобы продемонстрировать пользу от DSM-подхода. Эта фаза должна длиться всего несколько дней, после чего принимается решение о продолжении или завершении проекта. Далее следует приступить к разработке пилотного проекта, где впервые создаётся достаточно полная версия языка, и впервые её начинают использовать те, для кого этот язык создаётся. После пилотного проекта начинается фаза полномасштабного внедрения, когда уже несколько проектов внутри организации начинают использовать созданный язык. После этого начинается фаза сопровождения, на которой в язык и инструментальные средства вносятся изменения, при этом важно не нарушить совместимости с уже созданными моделями. После этого наступает момент, когда на созданном DSM-решении уже не создаются новые программы, это, как правило, означает, что предметная область изменилась настолько сильно, что не покрывается созданным решением. В этом случае решение выводится из эксплуатации.

При этом выделяются роли участников процесса создания и использования предметно-ориентированного решения. Роли включают в себя экспертов предметной области, пользователей предметно-ориентированного решения, разработчиков языка, эргономистов, разработчиков генераторов, разработчиков предметно-ориентированной библиотеки, разработчиков инструментария. Наличие разных ролей не обязательно означает, что их будут играть разные люди, но каждая роль имеет свою зону ответственности и связанные с ней задачи. Также приводятся соображения по организации команды — создавать DSM-решение должна небольшая группа высококвалифицированных специалистов.

В книге также приводятся рекомендации по деятельности на каждом этапе разработки, однако они слабо структурированы и могут восприниматься скорее как советы, чем предписания, поэтому нельзя сказать, что авторы предлагают целостную методологию разработки.

## 2.2.2 Паттерны и рекомендации по разработке предметно-ориентированных языков

Наиболее подробный свод рекомендаций по разработке предметно-ориентированных языков приведён в уже упоминавшейся книге [7] и базируется на многолетнем опыте авторов по разработке визуальных языков с использованием DSM-платформы MetaEdit+, а также разработки самой этой платформы. Рассматриваются преимущества от использования DSM, общие указания о том, когда следует применять DSM-подход (поскольку авторы редко ссылаются на другие работы, и сами серьёзных исследований преимуществ и применимости DSM-подхода не проводили, книга производит несколько рекламное впечатление, но тем не менее, многие указания из неё могут быть полезны на практике и подтверждаются в работах других авторов). В качестве основных составляющих DSM-решения авторы рассматривают визуальный язык, генератор кода, предметно-ориентированный каркас (или библиотеку, или окружение). Для задания абстрактного синтаксиса визуального языка предлагается использовать метамоделирование, конкретный синтаксис задаётся в графическом виде, либо в табличном виде, авторы обращают особое внимание на то, что не стоит пытаться всё изобразить графически. Обращается внимание на необходимость задания ограничений на модели. В качестве типичных подходов к реализации генераторов авторы выделяют генераторы на языках общего назначения, получающие доступ к модели через программный интерфейс репозитория, генераторы, основанные на паттерне „Visitor“ [41], шаблонные генераторы (использующие шаблонные файлы со вставками управляющих конструкций на специальном языке, которые подставляют в шаблонный файл информацию из модели), crawler-ы (генераторы, входные файлы для которых представляют собой набор управляющих конструкций со вставками фрагментов на целевом языке, в некотором смысле обратная по отношению к шаблонным генераторам ситуация), и генераторы генераторов. Предметно-ориентированный каркас, в тех случаях, когда он нужен, пишется вручную высококвалифицированными разработчиками, и служит, как правило, для упрощения генераторов.

Книга также содержит некоторые рекомендации, относящиеся к начальным фазам создания DSM-решения, например, приводится опросник, кото-

рый призван помочь принять решение о целесообразности начала разработки. Имеются также указания по выбору концепций предметной области, которые должны стать основными концепциями языка. Авторы не упоминают формальных методологий анализа предметной области. В качестве основных источников концепций выделяются архитектура существующих приложений в данной предметной области, сами существующие приложения, спецификации, типичные паттерны проектирования, целевое окружение и существующий код. Также предлагается обращать внимание на физическую структуру продукта (если предметно-ориентированный язык предназначен для программирования программно-аппаратного комплекса, полезным источником концепций могут стать его физические элементы, например, кнопки у часов), внешний вид приложения (если визуальный язык задаёт интерфейс, то лучше, чтобы модель на этом языке была близка по виду к разрабатываемому интерфейсу), различия между продуктами в линейке программных продуктов, мнение эксперта, требуемый вывод генератора.

Книга содержит массу подробных рекомендаций, касающихся всех аспектов разработки и внедрения предметно-ориентированного языка, вплоть до того, как преодолеть отторжение нового языка коллективом. Приводится и подробно анализируется пять примеров созданных при участии авторов DSM-решений, при этом решения были отобраны так, чтобы иллюстрировать разные подходы к разработке и разные типы визуальных языков. Книга может быть чрезвычайно полезна разработчикам DSM-решений и DSM-платформ, однако, не отменяет актуальности данной диссертации, поскольку авторы сфокусированы на использовании своего продукта (платформы MetaEdit+) и не рассматривают других подходов. К тому же, предлагаемая авторами методология довольно тяжеловесна и требует зрелой инструментальной и организационной поддержки.

Некоторый набор советов, частично пересекающийся с книгой, рассмотренной выше, приводится в статье [35]. Рекомендации разбиты на три раздела: разработка предметно-ориентированного языка, интерпретация или генерация, процесс и организация. Рекомендации сформулированы в достаточно общем виде, чтобы быть применимыми и к визуальным, и к текстовым языкам, автор обращает внимание, что не всегда визуальные нотации предпочтительны, в силу сложности реализации редакторов для них. Касательно организации процесса

автор также настаивает на итеративном процессе разработки, начинающемся с построения *proof of concept*, кроме того, автор рекомендует разрабатывать язык непосредственно в процессе анализа предметной области. Для этого требуется достаточно легковесный инструментарий, который позволял бы быстро увидеть результаты. Автор рекомендует не уделять серьёзного внимания опубликованным примерам реализации языков, поскольку предметно-ориентированные языки концептуально предназначены для узкой предметной области и настолько сильно отличаются друг от друга, что опыт создания одного языка может оказаться неприменимым для другой предметной области.

Статья также интересна списком открытых проблем: совмещение текстового и графического синтаксиса в языке, модульность и композиция языков, рефакторинг метамodelей, одновременный рефакторинг моделей и кода, автоматическая миграция моделей при изменении метамodelи, отладка на уровне моделей, работа с большими моделями.

Ещё одна работа, содержащая общие рекомендации — [36], от авторов системы *MetaEdit+*. Данная работа сфокусирована на этапе проектирования языка, а именно, поиске абстракций предметной области. Изложение ведётся по результатам исследования 23 примеров DSM-решений, созданных с помощью *MetaEdit+*. Было выделено четыре основных способа выделения сущностей: соображения эксперта предметной области, требуемый вывод генератора, внешний вид создаваемой системы, различия между продуктами в линейке программных продуктов. При этом первый способ (эксперт) считается в этом исследовании „не совсем честным“, поскольку эксперт сам должен был каким-либо образом провести требуемый анализ. Способы упорядочены по возрастанию сложности их применения — если есть эксперт, он представит готовые абстракции, которые достаточно реализовать в языке, а анализ различий продуктов, напротив, сложен (как правило, требует формальных методов, но ссылки на такие методы в статье не приведены).

### 2.2.3 Способы внутренней организации визуальных языков

Существуют работы, рассматривающие теоретические аспекты организации визуальных языков, которые могут быть важны для формирования методологии, поскольку такие аспекты так или иначе будут отражены в средствах задания языка (например, метаредакторах) и будут направлять деятельность проектировщика языка. Особенно это важно в случае больших языков или семейств языков со сложной метамodelью, таких как UML, и поэтому первый пример, который необходимо рассмотреть в этом разделе — языковая инфраструктура языка UML, пожалуй, самого известного на данный момент визуального языка программирования.

Стандарт UML версии 2.4.1 (текущий принятый на момент написания этого текста) разбит на две части: UML Infrastructure [42] и UML Superstructure [43]. Язык задаётся посредством метамодели, метамодель организована в соответствии с принципами модульности, расширяемости и переиспользуемости. За счёт этого на базе метамодели UML оказалось возможным определить ещё несколько стандартов, например, язык проектирования крупных систем SysML<sup>1</sup> (не только программных или аппаратных), язык моделирования метаданных для различных видов хранилищ данных CWM<sup>2</sup>. Метаязык, на котором описывается метамодель UML, описан в терминах самого метаязыка и является частью стандарта UML (авторам стандарта пришлось приложить некоторые усилия, чтобы избежать циклических ссылок). Используется четырёхуровневая иерархия метаописаний, где на нулевом уровне находятся данные времени выполнения, на первом уровне — модель, созданная пользователем, на втором уровне — метамодель UML, на третьем уровне — метаметамодель метаязыка, на котором задан UML (MOF<sup>3</sup>). Пример уровней метаописаний приведён на рисунке 2.1, взятом из [42].

<sup>1</sup>Домашняя страница стандарта на сайте OMG, URL: <http://www.omg.sysml.org/> (дата обращения 12.05.2013)

<sup>2</sup>Common Warehouse Metamodel, домашняя страница, URL: <http://www.omg.org/spec/CWM/> (дата обращения 12.05.2013)

<sup>3</sup>Meta Object Facility, домашняя страница, URL: <http://www.omg.org/mof/> (дата обращения 12.05.2013)

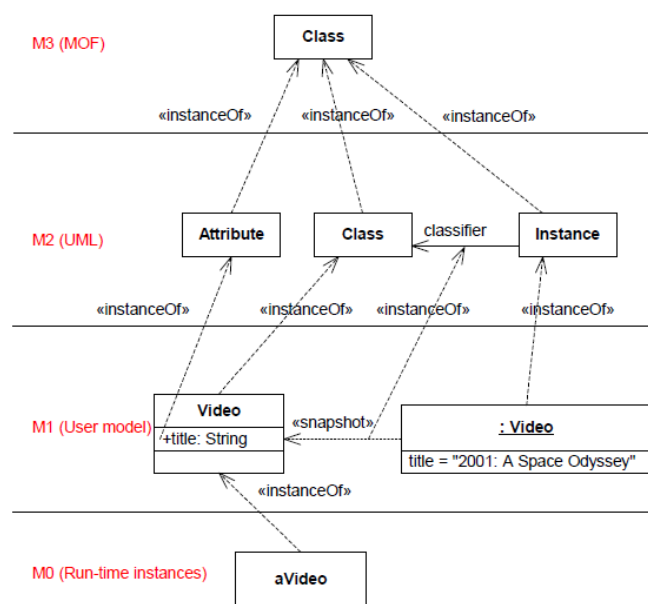


Рисунок 2.1: Иерархия метаописаний UML, из [42].

Интересно то, что некоторые сущности языка UML существуют одновременно на нескольких уровнях метаописаний, например, классификаторы (абстракция способа классификации экземпляров) являются, с одной стороны, базой для определения понятия „класс“ диаграммы классов (на уровне метамодели UML), с другой стороны, они же служат базой для определения классов и отношений в самом метаязыке (в пакете Core::Constructs). Кроме того, в стандарте UML определена только та часть метаязыка MOF, которая нужна для описания метамодели UML, сам MOF определяется отдельным стандартом, так что формально неверно то, что UML определён с помощью MOF. Всё это делает метамодель UML весьма запутанной, сложной для изучения и для переиспользования при создании предметно-ориентированных языков.

Поэтому язык UML включает в свой стандарт механизм легковесного расширения, называемый профилями. Профили позволяют расширить существующие метаклассы из метамодели UML, чтобы адаптировать язык под нужды конкретной предметной области. Достигается это путём определения стереотипов — „ограниченных“ метаклассов, которые могут быть применены к уже существующим метаклассам для того, чтобы дополнить их новыми свойствами или поменять их изображение. Например, метакласс „класс“ может быть расширен стереотипом „часы“, имеющим свойства „количество кнопок“, „тип операцион-

ной системы“ и специальную иконку для отображения. Генераторы, умеющие работать с определённым профилем, могут, во-первых, различать метаклассы с разными стереотипами, во-вторых, использовать их дополнительные атрибуты. Благодаря этому можно довольно простым и стандартным образом описывать предметно-ориентированные языки на базе UML. Однако же, профили позволяют только расширить язык UML, и сложность языка при использовании профилей всё равно сохранится. Вместе с тем, профили не позволяют определять новые сущности в метамодели. Профили, тем не менее, активно используются для генерации полного исходного кода, в том числе, в рамках подхода MDA [44].

В работе Аткинсона и Кюне [37] приводится критика подхода, применённого в разработке UML, основанная на том, что четырёхуровневая архитектура метауровней не делает разницы между отношениями „является лингвистическим экземпляром“ и „является онтологическим экземпляром“. Например, объект является экземпляром класса онтологически (то есть по смыслу), но формально диаграммы объектов и диаграммы классов не связаны. UML использует одно и то же отношение `instanceOf` как для обозначения того, что некоторая модель является экземпляром метамодели (лингвистическая связь), так и сугубо на уровне модели, для обозначения того, что объект является экземпляром класса. Авторы предлагают явно разделить эти два варианта отношения `instanceOf`, и ввести кроме лингвистической понятие онтологической метамодели. Таким образом, фактически, получится обобщение механизма профилей, позволяющее более адекватно выражать отношения предметной области. В качестве примера авторы приводят зоологическую классификацию животного мира: класс „Колли“ является наследником класса „Собака“, при этом онтологическим экземпляром класса „Порода“. Класс „Собака“ является онтологическим экземпляром класса „Вид“. И „Порода“, и „Вид“ — онтологические экземпляры класса „Биологический ранг“.

Идеи Кюне и Аткинсона были реализованы в системе MetaDepth [45, 46]. Платформа позволяет описывать только текстовые предметно-ориентированные языки, что не позволяет нам рассматривать её в обзоре вместе с другими DSM-платформами, но упоминания здесь она всё же заслуживает. Система позиционируется как платформа для „глубокого метамоделирования“ и предназначена прежде всего для создания метамodelей



предметно-ориентированных языков с использованием как лингвистического, так и онтологического инстанцирования. Описываемый язык может иметь произвольное количество метауровней (не просто „метаязык-метамодель-модель“, как в UML, а серия моделей, каждая из которых может быть моделью, описанной на языке более высокого метауровня, и одновременно метамоделью для более низкого метауровня). Такой подход позволяет естественно выражать отношения „тип-экземпляр“, возникающие, например, между диаграммами классов и диаграммами объектов в UML. Метамодель задаётся в текстовой форме, имеются средства для описания конкретного синтаксиса языков, ограничений и трансформаций моделей, все они учитывают то, что иерархия метамodelей „глубокая“. Авторы приводят примеры нескольких предметно-ориентированных языков, построенных по такому принципу, имеется работающий прототип системы в свободном доступе.

## 2.3 Создание визуальных языков в существующих DSM-платформах

### 2.3.1 Платформа MetaEdit+

Наиболее зрелой и известной на данный момент DSM-платформой, на наш взгляд, является среда MetaEdit+, разрабатываемая финской компанией MetaCase, на базе научного проекта University of Jyväskylä. Среда MetaEdit, из которой развилась среда MetaEdit+, создавалась с 1988 года, впоследствии была существенно переработана, и сейчас она (точнее, MetaEdit+) — успешный коммерческий проект, имеющий десятки внедрений и активно публикующий по своим результатам [7, 36, 47, 48]. Впрочем, исходный код системы закрыт, а лицензия довольно дорогая, что делает затруднительным переиспользование результатов проекта для дальнейших исследований другими научными группами.

Система состоит из двух компонент: MetaEdit+ Workbench и MetaEdit+ Modeler. Первая предназначена для создания предметно-ориентированных визуальных языков, вторая — для их использования. В качестве метаязыка система использует собственный язык GOPRR. Его название образовано первыми



буквами основных понятий, в нём используемых: граф (graph), объект (object), свойство (property), порт (port), отношение (relation), роль (role). Граф служит для представления модели (или диаграммы), в графе могут содержаться объекты, отношения и роли. Объект — это сущность предметной области (вершина графа), отношение связывает два или несколько объектов, при этом отношения соединяются с объектами с помощью ролей. Например, отношение наследования может иметь роли „предок“ и „потомок“, роль „предок“ рисуется в виде линии с треугольником на конце, роль „потомок“ в виде простой линии, а само отношение (это может показаться неинтуитивным) не визуализируется. Роли могут соединяться с самими объектами, а могут соединяться с портами — специальными областями внутри объектов, которые различаются генераторами и нужны, если одна и та же роль может иметь разные значения в зависимости от того, к какой части объекта подключена. Так, например, удобно моделировать различные приборы: объект „Усилитель“ может иметь аналоговый вход, цифровой вход и аналоговый выход. Свойства — это некоторые характеристики (вида „имя“-„значение“), которыми могут обладать все остальные понятия. Кроме этого, в метаязыке присутствуют понятия Binding (связывает отношения, роли и объекты), Object set (множество объектов, играющее одну роль, например, в Binding-e), Inheritance (отношение наследования между метатипами), Decomposition (возможность задать раскрытие какого-либо объекта в подграф, например, класс, раскрывающийся в диаграмму автоматов) и Explosion (возможность связывать объекты, отношения или роли с другими графами).

Предлагаемая этой DSM-платформой методика создания языка такова: сначала определяются основные концепции языка (сущности и связи), затем задаются ограничения, определяются графические символы для концепций, затем определяются правила генерации. Ограничения определяются как набор правил в метамодели, правила генерации описываются на специальном текстовом предметно-ориентированном языке с визуальным представлением иерархии вызовов правил. Процесс итеративен, все изменения, вносимые в метамодель, тут же применяются к модели. Если вносимое в метамодель изменение конфликтует с существующими моделями, выдаётся предупреждение. Существует визуальный язык для задания метамодели, но его использование не обязательно, причём в документации и обучающих материалах визуальный язык, как прави-

ло, не используется — метамодель редактируется с помощью диалоговых окон. Метамодель можно до определённой степени менять прямо из редактора модели: задавать сущностям новые свойства, менять существующие, редактировать их внешний вид с помощью графического редактора. Однако добавить новую сущность в язык или удалить существующую можно только через редактор метамодели. Поскольку изменения в метамодели тут же отражаются в редакторе модели, это позволяет организовать работу над языком короткими итерациями с непосредственным участием эксперта, как в случае с AgentSheets, но пользовательский интерфейс MetaEdit+ довольно сложен, что требует наличия подготовленного специалиста по созданию языков. Инструмент непосредственно поддерживает только создание и изменение языка, в работах авторов имеются методологические рекомендации по действиям в других фазах жизненного цикла, но непосредственно в инструментарии они не поддерживаются.

### 2.3.2 Eclipse Modeling Project

Ещё одна DSM-платформа, ставшая стандартом де-факто для исследований в области предметно-ориентированного визуального моделирования — Eclipse Modeling Project (EMP)<sup>4</sup>. Фактически, это объединение большого количества различных библиотек и инструментов, часть которых развивается независимо и предназначается не только для визуального моделирования. Проект строится на базе платформы Eclipse<sup>5</sup>, имеет открытый исходный код и большое сообщество разработчиков. На базе этого проекта существует несколько коммерческих инструментов, например, DSM-платформа Borland Together<sup>6</sup>, CASE-система Rational Software Architect<sup>7</sup>, DSM-платформа Obeo Designer<sup>8</sup>, а также несколько исследовательских проектов в области предметно-ориентированного моделирования, например, MOFLON [49]. При этом проект активно развивается, появляются новые библиотеки и средства создания предметно-ориентированных

---

<sup>4</sup>Домашняя страница Eclipse Modeling Project, URL: <http://www.eclipse.org/modeling/> (дата обращения 03.09.2014)

<sup>5</sup>Домашняя страница Eclipse, URL: <https://www.eclipse.org/> (дата обращения 03.09.2014)

<sup>6</sup>Сайт Borland, Домашняя страница Borland Together, URL: <http://www.borland.com/products/together/> (дата обращения 02.06.2013)

<sup>7</sup>Сайт IBM, Информация о продукте Rational Software Architect, URL: <http://www-03.ibm.com/software/products/us/en/ratisoftarch> (дата обращения 02.06.2013)

<sup>8</sup>Сайт Obeo Designer, URL: <http://www.obeodesigner.com/> (дата обращения 14.06.2014)

языков (например, библиотека для создания графических редакторов Graphiti<sup>9</sup> и генератор редакторов по текстовому описанию Spray<sup>10</sup>, построенный поверх этой библиотеки). Это создаёт большие сложности в использовании и изучении этого проекта, поскольку имеющаяся документация фрагментарна и сильно отстаёт от ведущейся разработки, публикации, как правило, посвящены отдельным подпроектам, и имеется лишь немного материала, посвящённого проекту в целом (например, [50], и статья на русском языке [51]).

В качестве метаязыка в ЕМР используется язык Ecore, который представляет собой близкую к стандарту реализацию MOF (точнее, его варианта EMOF), метаязыка UML. Элемент создаваемого языка представляется в метамодели объектом метакласса EClass, у него могут быть атрибуты, задаваемые объектами метакласса EAttribute, операции, задаваемые объектами EOperation, и ассоциации, задаваемые объектами EReference. Атрибуты имеют тип, моделируемый с помощью метакласса EDataType. Кроме того, имеется возможность добавлять аннотации элементам (которые могут быть использованы при генерации) с помощью метакласса EAnnotation.

Процесс разработки предметно-ориентированного языка, предполагаемый при использовании ЕМР, следующий: сначала создаётся проект плагина в Eclipse, затем в этом проекте определяется модель абстрактного синтаксиса (метамодель), модель конкретного синтаксиса, преобразования моделей в модели и моделей в текст (если требуется), текстовый конкретный синтаксис (если требуется), по результатам генерируются редакторы, генераторы и прочие инструменты, тестируются, процесс повторяется итеративно. По окончании разработки плагин-редактор дополняется различными мастерами, графикой и т.д., формируется инсталляционный пакет.

Абстрактный синтаксис языка может быть задан с помощью графической метамодели, либо импортирован из XSD-схемы, диаграммы классов UML, или даже исходного кода на Java. Это оказывается довольно удобным, поскольку позволяет переиспользовать существующие наработки: например, для языков, базирующихся на XML, XSD-схемы часто доступны как часть стандарта. Одна-

---

<sup>9</sup> Домашняя страница проекта Graphiti на сайте Eclipse, URL: <http://www.eclipse.org/graphiti/> (дата обращения 02.06.2013)

<sup>10</sup> Домашняя страница проекта Spray на Google Code, URL: <https://code.google.com/a/eclipselabs.org/p/spray/> (дата обращения 02.06.2013)

ко по метамодели средствами EMF могут быть сгенерированы только заглушки для классов предметной области, механизм оповещений об изменениях в модели, и простой редактор, представляющий модель в виде дерева. Конкретный синтаксис задаётся с помощью библиотеки GMF, которая сама создавалась как отдельная DSM-платформа, и использует предметно-ориентированный подход к разработке. Для задания формы фигур надо создать модель определения графики (Graphical definition model). Для описания палитры элементов и панелей инструментов, используемых в создаваемом визуальном редакторе, необходимо создать модель инструментов (Tooling model). После этого надо связать эти две модели с моделью абстрактного синтаксиса, созданной с помощью EMF, с помощью модели соответствия (Mapping model) и определить модель генерации (Generator model), где определить все оставшиеся необходимые для генерации редактора свойства. Редактирование всех моделей производится с помощью специально предназначенных для этого мастеров.

Определение текстового синтаксиса для создаваемых языков возможно с помощью компонент Xtext и TCS, первая из которых позволяет задавать синтаксис, используя EBNF<sup>11</sup>, вторая — путём связывания элементов конкретного синтаксиса с элементами метамодели. Определение правил преобразования модели в модель или модели в текст возможно с помощью языков QVT и ATL. Такие преобразования могут использоваться для определения рефакторингов над моделями, миграции моделей (то есть для преобразования моделей, созданных в соответствии со старой версией метамодели, в модели, соответствующие новой метамодели), объединения двух или нескольких моделей в одну модель или генерации кода на текстовых языках. Для многих решаемых задач существуют отдельные проекты, например, средства обеспечения миграции моделей Edapt и COPE. Все преобразования описываются на текстовых языках. Есть и специальные средства для задания правил генерации (компоненты Xpand/Xtend<sup>12</sup>), которые также в текстовом виде позволяют описывать правила генерации в виде шаблонов кода на целевом языке, дополненных императивным языком, задающим правила обхода модели и генерации параметризуемых из модели

---

<sup>11</sup>Extended Backus-Naur Form, Расширенная форма Бэкуса-Наура, URL: <http://www.cl.cam.ac.uk/mgk25/iso-14977.pdf> (дата обращения 03.09.2014)

<sup>12</sup>Страница Xpand на Eclipse Wiki, URL: <http://wiki.eclipse.org/Xpand> (дата обращения 12.06.2013)

фрагментов. Ограничения на модели также задаются в текстовом виде на языке OCL. Общая схема создания предметно-ориентированного решения в Eclipse Modeling Project представлена на рисунке 2.2.

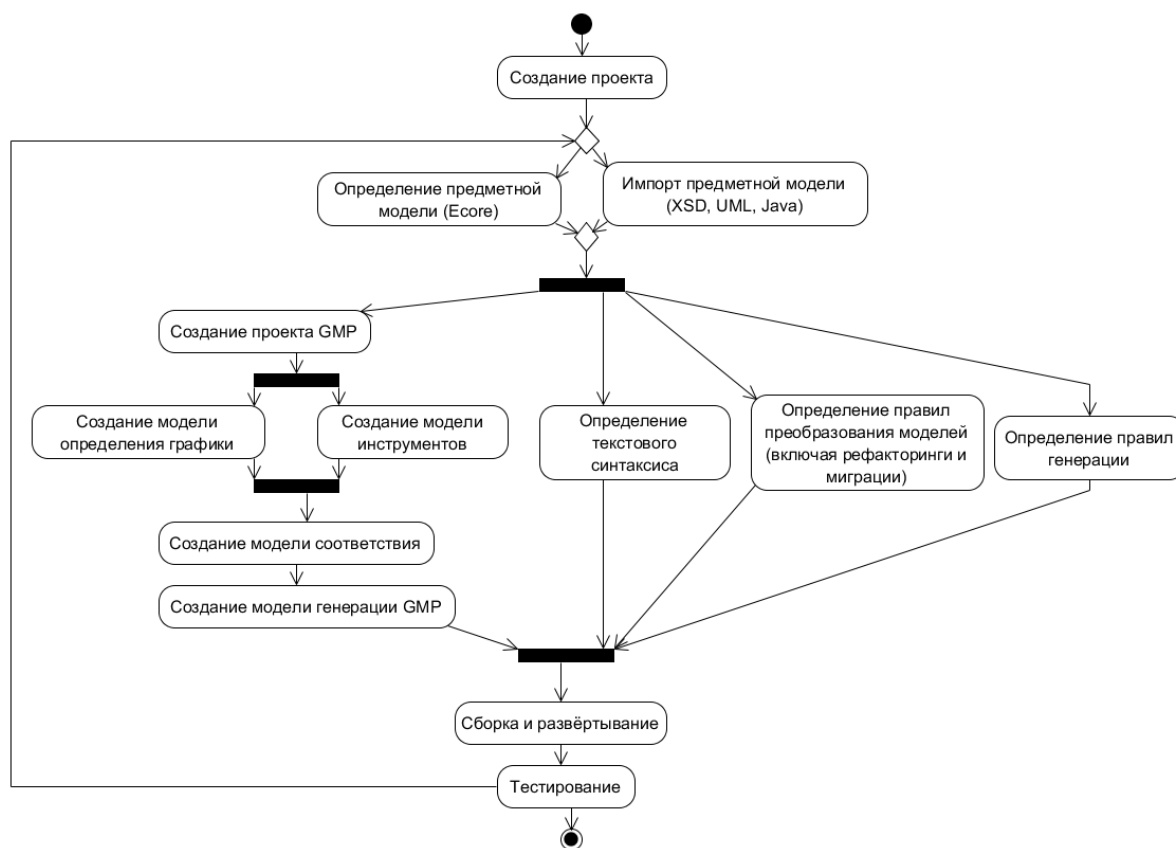


Рисунок 2.2: Процесс создания DSM-решения на EMP.

Как видно, процесс создания DSM-решений на EMP сложен сам по себе и дополнительно усложняется слабой связностью наличествующих компонентов. Существует даже отдельный проект по объединению всего существующего в единую технологию (Modeling Amalgamation Project<sup>13</sup>), однако новые компоненты продолжают появляться, технология активно развивается, и использование её всё ещё требует больших усилий на исследование и выбор доступных средств (далеко не все существующие в рамках EMP проекты достаточно зрелые для реального использования). Кроме того, наличествующие средства, помимо усилий на создание большого количества разных моделей и текстовых спецификаций, по которым будет сгенерировано DSM-решение, зачастую требуют и ручного кодирования на языке Java. Таким образом, EMP используется в основном как

<sup>13</sup> Домашняя страница Modeling Amalgamation Project на сайте Eclipse, URL: <http://www.eclipse.org/modeling/amalgam/> (дата обращения 12.06.2013)

база для разработки более доступных для конечного пользователя инструментов.

### 2.3.3 Платформа Generic Modeling Environment

Платформа Generic Modeling Environment (GME)<sup>14</sup> [52] является одной из самых известных академических разработок в области создания DSM-решений. Эта платформа имеет расширяемую модульную архитектуру, благодаря чему на её основе было создано несколько исследовательских и промышленных инструментов (например, средство симуляции встроенных систем MILAN<sup>15</sup>, средство для аспектно-ориентированного задания ограничений на модели C-SAW<sup>16</sup>). Проект имеет открытый исходный код, разрабатывается в основном на языке C++, до сих пор развивается.

Визуальные языки в GME задаются с помощью визуальных метамodelей (авторы GME рассматривают задачу разработки визуального языка как просто ещё одну задачу, к которой может быть применён предметно-ориентированный подход). Метаязык ориентирован на иерархическую декомпозицию моделей: любой составной элемент языка является моделью сам по себе, и задаётся метаклассом *Model*. Элементы, которые не могут иметь составных частей, называются атомами (метакласс *Atom*). Модель может состоять из подмоделей и атомов, при этом каждый атом внутри модели имеет роль (метакласс *Role*): роли задают „гнезда“ внутри модели, куда пользователь может вставлять различные подмодели и атомы. Управление отображением частей модели осуществляется с помощью метакласса *Aspect* — каждая модель имеет предопределённый набор аспектов, каждая часть может быть видима или скрыта в каком-либо аспекте. Каждая сущность языка имеет набор „основных“ аспектов, в которых она может быть создана или удалена. Связи между объектами задаются с помощью метакласса *Connection*. Связываемые элементы модели должны иметь одного родителя в иерархии вложенности и быть одновременно видимыми в одном

---

<sup>14</sup> Домашняя страница GME, URL: <http://www.isis.vanderbilt.edu/Projects/gme/> (дата обращения 12.06.2013)

<sup>15</sup> Домашняя страница MILAN, URL: <http://w3.isis.vanderbilt.edu/projects/milan/> (дата обращения 12.06.2013)

<sup>16</sup> Домашняя страница C-SAW, URL: <http://www.gray-area.org/Research/C-SAW/> (дата обращения: 12.06.2013)

аспекте. Если требуется связать элементы, относящиеся к разным моделям, используются невизуальные ссылки, описываемые в метамодели метаклассом *Reference*. Ссылка выглядит как обычный объект и может использоваться в какой-либо роли в модели, но на самом деле она представляет объект, находящийся где-то ещё, по аналогии со ссылками или указателями в текстовых языках программирования. Возможно также определение N-арных отношений, задаваемых в метамодели метаклассом *Set*. Отношения, атомы, ссылки, связи, модели могут иметь атрибуты, задаваемые метаклассом *Attribute*, и ограничения, задаваемые метаклассом *Constraint*. Ограничения задаются на стандартном языке OCL. Модели могут быть организованы в папки (описываемые на уровне метамодели метаклассом *Folder*), папки могут содержаться в проекте (метакласс *Project*).

Возможно переиспользовать существующие метамодели при разработке новых, для этого используются ссылки и операторы, определённые в метаязыке: эквивалентность, наследование реализации и наследование интерфейса. Оператор эквивалентности связывает два класса и делает их одним, объединяя все атрибуты, операции и связи двух классов. С помощью оператора эквивалентности можно создавать “точки склейки” между разными метамоделями. Различные операторы наследования введены для более точного управления наследованием отношений, в которых участвует наследуемый класс: в случае наследования реализации наследуются все атрибуты класса и те отношения вложенности, в которых наследуемый класс участвует как контейнер; в случае наследования интерфейса, наоборот, атрибуты не наследуются, но наследуются все отношения, в которых участвует класс, кроме отношений вложенности, в которых наследуемый класс участвует как контейнер.

По созданной метамодели может быть автоматически сгенерирован предметно-ориентированный визуальный редактор, после чего его тут же можно начать использовать в среде. В качестве конкретного синтаксиса используется синтаксис диаграммы классов UML, который может быть расширен иконками и цветом. Специальных средств для задания правил генерации или преобразования моделей в среде нет, имеется развитый программный интерфейс, с помощью которого авторы DSM-решений могут реализовать свои инструменты. Именно так реализован генератор редакторов самого GME — отдельный



плагин, который работает с метамоделью через API, и генерирует файл с описанием парадигмы (собственно, предметно-ориентированного редактора), который потом может быть загружен в среду. Такой подход, хоть и позволяет обеспечить достаточно быстрое прототипирование визуального языка, требует знания API GME и существенных усилий в „ручном“ программировании, чтобы создать полноценное решение. Автоматическая миграция моделей при изменении метамodelей также не поддерживается, таким образом, эволюцию предметно-ориентированного решения приходится организовывать также вручную. Существует отдельный проект GReAT<sup>17</sup>, предоставляющий возможности для описания преобразования моделей в GME.

### 2.3.4 Платформа PSL/PSA

Платформа PSL/PSA (Problem Statement Language/Problem Statement Analyzer) [23] является самой старой из известных автору систем, которые позволяют создавать специализированные языки программирования, являясь в некотором смысле самой старой из известных DSM-платформ. В обзор она включена прежде всего потому, что представляет несомненный исторический интерес. Её разработка началась в 1968 году и продолжается до сих пор<sup>18</sup>. Система предназначена для сбора и анализа требований к программному обеспечению, и использовалась в некоторых крупных проектах. Система содержит в себе текстовый язык описания информационных систем (PSL), репозиторий, где хранятся описания и часть, отвечающую за анализ и генерацию отчётов по репозиторию (PSA).

В PSL система состоит из объектов, объекты имеют свойства и могут быть связаны между собой отношениями. Эта простая модель специализируется для определённого класса систем, например, для информационных систем, так, что в них можно использовать только предопределённые объекты и отношения (то есть, по сути, создаются метамодели специализированных языков). Описание системы состоит из довольно большого числа (порядка 8) различных видов

<sup>17</sup>Домашняя страница GReAT, URL: <http://www.isis.vanderbilt.edu/tools/GReAT> (дата обращения 12.06.2013)

<sup>18</sup>По данным <http://www.pslpsa.com/>, система перерабатывается на современные технологии, а по данным <http://requirementsanalytics.com/index.php/consulting-and-training/pslpsa> существует проект по возрождению этой системы как ПО с открытым исходным кодом



моделей, каждая из которых описывает свой аспект системы. Например, есть отдельное описание формата ввода-вывода системы, описывающее её взаимодействие с внешним миром. По этим моделям могут генерироваться различные отчёты в текстовой или в графической (псевдографической) форме (поэтому в каком-то смысле её можно отнести к визуальным средствам).

### 2.3.5 Платформа АТоМ3

Платформа АТоМ<sup>3</sup><sup>19</sup> — это ещё один пример академической разработки, предназначенный прежде всего для изучения преобразования моделей. Проект имеет открытые исходные коды, реализован на языке Python, отличается нестабильностью и неудобностью пользовательского интерфейса, тем не менее, широко известен в научном сообществе, имеется большое количество публикаций (например, [53]). Ныне не развивается, авторы использовали полученный опыт при разработке системы MetaDepth, описанной выше.

В качестве метаязыка в АТоМ<sup>3</sup> используется визуальный язык „сущность-связь“, позволяющий в виде сущностей с атрибутами задавать сущности предметной области и в виде связей описывать взаимосвязи между ними. Имеется также несложный векторный редактор, позволяющий описывать конкретный синтаксис элементов. По описанию метамодели генерируется код редактора на языке Python, после чего редактор может быть загружен в среду. Ограничения на модели также описываются на языке Python, вручную.

Каждая метамодель может быть снабжена набором трансформаций, описываемых с помощью графовых грамматик. Грамматика состоит из продукций, каждая из которых имеет левую часть (шаблон, который ищется в исходном графе), правую часть (то, на что надо заменить найденный шаблон), условие применения (логическое выражение на языке Python, продукция применяется, только если оно истинно) и действие, выполняемое после применения продукции (тоже код на языке Python). В продукции можно указать соответствие между элементами левой и правой части, что позволяет копировать или модифицировать свойства, кроме того можно создавать или удалять узлы. С помощью таких преобразований можно определять операционную семантику языка либо

---

<sup>19</sup> Домашняя страница АТоМ<sup>3</sup>, URL: <http://atom3.cs.mcgill.ca/> (дата обращения 12.06.2013)

давать возможность преобразовать модель в модель на другом языке. Таким же образом реализуется и кодогенерация: графовая грамматика задаёт продукции, не совершающие содержательных преобразований, а код на текстовом языке генерируется как побочные эффекты выполнения правил.

Таким образом, цикл разработки DSM-решения на АТоМ<sup>3</sup> состоит из определения метамодели языка, определения конкретного синтаксиса, определения правил преобразования моделей, генерации редактора и тестирования. Однако, среда недостаточно зрела для промышленного использования и подходит только для разработки прототипов.

### 2.3.6 Платформа Microsoft Modeling SDK

Технология Microsoft Modeling SDK [54] более известна в научном сообществе как Microsoft DSL Tools и с начала своего существования (2003 г.) успела несколько раз сменить название. Технология является подключаемым модулем к среде разработки Microsoft Visual Studio<sup>20</sup> и доступна для бесплатного скачивания с сайта Microsoft (при этом сама среда Visual Studio далеко не бесплатна для коммерческого использования). Среда позволяет прямо в Visual Studio в графическом виде задать метамодель создаваемого языка, описать внешний вид элементов, задать в текстовом виде правила генерации и ограничения и сгенерировать подключаемый модуль с редактором языка и генератором, который можно запустить в Visual Studio. Технология требует установленной Visual Studio и для создания, и для использования предметно-ориентированных решений. Тем не менее, зрелость технологии и хорошая интеграция её в Visual Studio делают её весьма привлекательной для промышленных разработчиков.

Метамодель задаётся на метаязыке, напоминающем диаграммы классов UML — поддерживаются отношения между сущностями, свойства сущностей, наследование как сущностей, так и связей. Интересной особенностью метаязыка является то, что метамодель рисуется всегда в виде дерева, упорядоченного по связям между сущностями (вложенность или наличие связи), и если граф метамодели не допускает представление в виде дерева (как, например, при наличии циклических отношений), одну сущность нужно разместить на диаграмме

---

<sup>20</sup>Домашняя страница Microsoft Visual Studio, URL: <http://www.visualstudio.com/> (дата обращения: 11.08.2014)

метамоделю несколько раз. Конкретный синтаксис задаётся для каждой сущности посредством выбора из доступных в визуальном редакторе Visual Studio примитивов и описывается на той же диаграмме, что и абстрактный синтаксис — диаграмма разделена на две части, и сущности в абстрактной метамоделе соединены линиями с описаниями их конкретного синтаксиса. При желании форму фигур можно сделать произвольной, но это требует ручного кодирования на языке C#.

Правила генерации кода на текстовом языке задаются с помощью шаблонов на языке T4, которые представляют собой текст на целевом языке со вставками конструкций, управляющих процессом генерации, на языке C# (похожий подход применяется в технологии Microsoft ASP.NET<sup>21</sup> для генерации HTML-страниц). В управляющих конструкциях есть доступ к элементам модели, кроме того, есть возможность описывать произвольный код генератора внутри шаблона. Ограничения на модели описываются целиком на языке C# с использованием довольно развитой инфраструктуры для проверки ограничений, они могут проверяться как непосредственно при рисовании диаграммы на предметно-ориентированном языке, так и при сохранении диаграммы (если диаграмма неконсистентна, будет выдано предупреждение, и генератор не запустится). С помощью этого же механизма могут быть реализованы дополнительные инструменты работы с моделями, например, поддержка рефакторингов.

В книге [54] приводится ряд рекомендаций по процессу создания языка. Первое, что надо сделать по мнению авторов — выявить изменяемые части предметной области, которые и будут задаваться визуальным языком. Для этого авторы предлагают использовать деревья функциональности (feature trees<sup>22</sup>), метод, очень похожий на метод FODA, один из формальных методов, упоминавшихся в 2.2.1. Для разработки метамоделю авторы предлагают использовать наброски диаграмм на создаваемом языке для рассмотрения конкретных сценариев его применения и выделения наиболее изменчивых частей. Данные методы предлагается использовать неформально, инструментальной поддержки для них не предусмотрено. Для определения правил генерации предлагается

<sup>21</sup> Домашняя страница ASP.NET, URL: <http://www.asp.net/> (дата обращения 11.08.2014)

<sup>22</sup> см. K. Czarnecki, S. Helsen, U. Eisenecker, "Staged Configuration Using Feature Models" Software Product Lines: Third International Conference, Boston, MA, 2004

инкрементальный подход — сначала код модельного приложения копируется в шаблон без изменений, потом некоторые его части замещаются правилами генерации по модели, на каждом этапе генерируется предметно-ориентированное решение и проверяется его работоспособность.

В целом, технология производит впечатление достаточно зрелой, но требующей большого объёма ручного кодирования и знания составляющих её библиотек.

### 2.3.7 Платформа Roundapi

Платформа Roundapi [55] создавалась как DSM-платформа для разработки многопользовательских визуальных сред разработки, способных использовать одновременно несколько нотаций для одного языка. В основу архитектуры системы были заложены принципы простоты использования (то есть простоты задания визуального языка и определения инструментальной поддержки для него) и простоты расширения и модификации. Платформа имеет довольно гибкую архитектуру, включающую в себя API в виде веб-сервисов и сериализацию создаваемых моделей в виде XML-файла, что позволяет удобно создавать инструменты, расширяющие её функциональность. В частности, благодаря этому была реализована функциональность просмотрщиков моделей в браузере и на мобильном телефоне.

Roundapi обладает довольно развитыми метасредствами, включающими в себя визуальный метаредактор, визуальный редактор формы фигур, визуальный редактор обработчиков событий. В качестве метаязыка используется язык „сущность-связь“, с интересной особенностью: ассоциации в метамодели соединяют элементы, экземпляры которых будут соединять в модели экземпляры этих ассоциаций, например, отношение “реализует” между классом и объектом является отношением между метаклассами „класс“ и „объект“ в метамодели. Авторы намеренно сделали метаязык очень простым, чтобы он был доступен возможно более широкому кругу пользователей. На метаязыке описывается только абстрактный синтаксис языка, для задания конкретного синтаксиса существует редактор формы фигур, при этом связь между формами, созданными с помощью этого редактора, и элементами метамодели осуществляется с помощью

модели соответствия, которая позволяет задавать несколько форм для одного элемента, указывать, как поля на форме соответствуют свойствам элемента в метамодели и т.д. Для того, чтобы задать, какую из форм элемента использовать в конкретном контексте, используется понятие „вид“ (близкое к понятию „диаграмма“ языка UML), с помощью диалогового редактора можно указать, какие формы элементов используются в каком виде.

Заслуживает особого внимания способ реализации сложной логики средств инструментальной поддержки (например, проверки ограничений, генерации, задания сложных правил отображения между элементами метамодели и формами элементов, автоматического размещения элементов на диаграммах). Для этого используется визуальный язык задания правил обработки событий, напоминающий язык SDL или блок-схемы. В DSM-решении могут происходить различные события (например, создание или перемещение элемента, нажатие кнопки на панели инструментов), обработчик может подписываться на событие и выполнять некоторые действия, состоящие из довольно высокоуровневых команд системы. Имеется визуальный отладчик обработчиков. Имеется возможность определять свои блоки для диаграмм обработчика программированием на языке Java. При этом DSM-платформа обеспечивает динамическую компиляцию и загрузку написанного кода прямо в процессе работы.

Предлагаемая авторами методология разработки предполагает быстрое итеративное создание языка, с немедленным тестированием вносимых изменений, без длительной перекомпиляции и перезапуска платформы. Сначала описывается конкретный и абстрактный синтаксисы, модель соответствия, затем определяется логика инструментальных средств с помощью редактора обработчиков событий. При необходимости код на Java дописывается вручную. Возможно дальнейшее расширение инструментальной поддержки ручным кодированием с использованием API веб-сервисов или разбором XML-файлов с сохранёнными моделями. Таким образом, методология предполагает ручное программирование на поздних этапах разработки. Кроме того, даже ограничения или рефакторинги в Roundup задаются с помощью языка обработки событий (по сути, низкоуровневого языка общего назначения).

### 2.3.8 Платформа DOME

Платформа DOME [56, 57] (DOmain Modeling Environment) создавалась с целью упростить прототипирование и создание графических модельно-ориентированных сред разработки. Это одна из первых DSM-платформ с поддержкой графического описания метамодели визуального языка, разрабатываемая компанией Honeywell с 1992 года. Платформа имеет возможность как генерировать по метамодели редактор визуального языка на языке Smalltalk, так и интерпретировать метамодель напрямую. Для задания ограничений, правил генерации, трансформации моделей и настройки поведения элементов используется специальный текстовый язык Alter (диалект Scheme<sup>23</sup>) либо визуальный язык Projector. Есть и упрощённый генератор документации, позволяющий задавать шаблон документации в виде .doc-файла, где размечены места, заполняемые генератором по модели.

Метаязык, используемый в DOME, называется DTSL (DOME Tool Specification Language) и реализован в метаредакторе в самой системе, называемом ProtoDome. Язык представляет собой вариант диаграмм „сущность-связь“, позволяет задавать узлы языка, связи между ними, ограничения на связи, правила структурной декомпозиции диаграмм (какой узел может содержать какую диаграмму в качестве подэлемента). Большое внимание в метаязыке уделяется графической нотации создаваемого языка — большая часть редактируемых свойств относится именно к конкретному синтаксису. Связано это с тем, что DOME не имеет встроенного редактора конкретного синтаксиса, поэтому настройка внешнего вида элементов выполняется через свойства элементов в метаязыке либо ручным кодированием на Alter. Такой подход, тем не менее, достаточно удобен, поскольку DOME предоставляет много возможностей по настройке внешнего вида элементов, а наличие прямо в метаязыке таких сущностей, как, например, „Палитра“ или „Список свойств“ позволяет быстро и эффективно создавать редакторы, не прибегая к ручному кодированию или использованию вспомогательных инструментов. Визуальный метаязык хорошо интегрирован с текстовым языком Alter, каждый элемент метаязыка имеет на-

---

<sup>23</sup>Стандарт Scheme R6RS, URL:<http://www.r6rs.org/> (дата обращения 14.06.2014)

бор свойств — программ на Alter, которые вызываются средой при наступлении события, соответствующего свойству.

Предлагаемый подход к разработке языка состоит в описании метамодели языка в DOME и последующей её интерпретации, создании модели с её помощью и внесении изменений в метамодель „на лету“, при этом среда автоматически обновляет все имеющиеся в репозитории модели и открытые редакторы. Сами авторы отмечают в [57], что такие действия могут приводить к неконсистентности моделей, например, при изменении типа свойства, причём это не отслеживается средой. Кроме того, среда не сохраняет в модели значения свойств по умолчанию, так что изменение их в метамодели может привести к неожиданным для пользователей изменениям в модели. При этом для изменения метамодели требуется наличие метаредактора. После того, как метамодель будет завершена, к ней возможно написать генератор, либо на Alter или Projector, либо используя упрощённый генератор MetaScribe. Также есть возможность задать трансформации моделей, дополнительные ограничения, определить свои инструменты, которые будут добавлены на палитру, всё это делается на Alter или Projector, то есть языках общего назначения.

### 2.3.9 Платформа MetaLanguage

Платформа MetaLanguage [25, 29, 58, 59] разрабатывается в Высшей Школе Экономики в Перми. Система нацелена на разработку предметно-ориентированных визуальных языков, которые могут изменять свой синтаксис в процессе работы. Также важной отличительной особенностью системы MetaLanguage является то, что модели и метамодели рассматриваются единообразно, что даёт возможность создавать специализированные метаязыки для более удобного создания или модификации визуальных языков для конкретной предметной области. Система имеет репозиторий, реализованный в виде реляционной базы данных, графический редактор, валидатор, поддерживающий проверку описанных в метамодели ограничений, браузер моделей и средство трансформации моделей. Предметно-ориентированные решения, созданные с использованием системы, работают с репозиторием, интерпретируя модели и



метамоделей, которые в нём находятся. Исходных кодов системы или версии, доступной для скачивания, в свободном доступе, по всей видимости, нет.

В качестве основного метаязыка в системе MetaLanguage используется язык „сущность-связь“. Основные понятия языка — сущность, связь, свойство (свойства могут быть как у сущностей, так и у связей), ограничение. Авторы системы подчёркивают, что в роли метаязыка может выступать произвольный язык, созданный в системе, возможна потенциально бесконечная иерархия языков, в которой каждый язык — метаязык для последующего языка.

Процесс создания языка, предлагаемый авторами MetaLanguage, такой — проводится анализ предметной области, выделяются основные сущности и связи между ними, строится метамоделю языка. Метамоделю интерпретируется редактором, который позволяет создавать диаграммы — экземпляры созданной метамоделю. При необходимости в метамоделю могут быть внесены изменения, все модели, созданные с её помощью, будут автоматически обновлены, поскольку находятся в том же репозитории.

### 2.3.10 Сравнение рассмотренных платформ

Результаты анализа возможностей существующих DSM-платформ по созданию визуальных языков приведены в таблицах 2.1 и 2.2. Рассмотрены используемый в системе метаязык, наличие визуального метаредактора (или других способов описания метамоделю), наличие возможности описания конкретного синтаксиса языка, задания ограничений, средств описания правил генерации, трансформации моделей, задания семантики языков и возможности интерпретации диаграмм в DSM-решениях, создаваемых на базе рассматриваемой платформы, наличие поддержки совместной эволюции моделей и метамоделю. Рассмотренные возможности условно разделены на основные (необходимые для создания редакторов визуальных языков) и дополнительные (обеспечивающие важную функциональность помимо редакторов). Последней строкой в каждой таблице приведена система QReal, на базе которой велись исследования, представленные в данной диссертации. Возможности QReal будут подробно описаны в главе 4.



Хочется обратить внимание, что подавляющее большинство рассмотренных систем позволяют реализовать всю необходимую функциональность на языках программирования общего назначения — либо предоставляя программный интерфейс, либо свои исходные коды. Поэтому в таблице указано „вручную“ только в том случае, если авторы явно указывали программирование на текстовом языке как рекомендуемый способ реализации функциональности. Слово „нет“ в таблице стоит в том случае, если упоминаний о соответствующей функциональности не удалось найти ни в рассмотренных работах авторов технологии, ни в демонстрационной версии самой технологии, если она была доступна. При этом вполне возможно, что функциональность была описана в малоизвестных источниках и просто не была найдена при написании обзора. „Неизвестно“ стоит в случае, если наличие функциональности упоминается в открытых источниках, но не приводятся подробностей.

## 2.4 Выводы

Выполненный обзор научных публикаций по методологиям и технологиям создания предметно-ориентированных языков выявил недостаток внимания со стороны исследователей к технологиям, связанным с визуальными языками. Существующие работы либо подробно описывают методологию создания предметно-ориентированных языков, но ориентированы только на текстовые языки, либо описывают инструменты для создания предметно-ориентированных визуальных языков, но практически без методологической поддержки, причём очень часто внимание уделяется только самой реализации визуального языка. Таким образом, если автор DSM-решения уже „знает, что писать“ (то есть имеет в голове ясное представление о предметной области и даже метамодель создаваемого языка), то к его услугам множество существующих инструментов. Но если требуется вести разработку предметно-ориентированного решения „с нуля“ (начиная с оценки осуществимости и анализа предметной области), очень многое придётся делать без какой-либо инструментальной поддержки и, как правило, без руководства к действию. Кроме того, ни одна из существующих систем не автоматизирует все важные аспекты разработки DSM-решения. Поэтому сейчас создание DSM-решений требует немалой

квалификации и опыта даже при использовании промышленных средств, таких как MetaEdit+.

Таким образом, явно имеется пробел в существующих исследованиях, который данное исследование призвано помочь заполнить. Наиболее близкой по содержанию к данной диссертации является рассмотренная в обзоре работа [38], где предлагается методология коллаборативной разработки визуального языка и технология её поддержки. Однако эта методология применима лишь для довольно узкого класса языков и в относительно несложных случаях. Другие существующие технологические средства сконцентрированы на решении своих узких задач и не предназначены для поддержки полного цикла разработки языка. Многие из них реализуют только одну конкретную составляющую предметно-ориентированного решения, что, если вспомнить о практически полном отсутствии возможностей для интеграции между инструментами различных производителей, делает возможным только прототипирование DSM-решений.

Таблица 2.1: Основные возможности существующих DSM-платформ

Название	Метаязык	Метаредактор	Конкретный синтаксис	Ограничения
MetaEdit+	GOPRR	Визуальный или диалоговые окна	Визуально	Только средствами метаязыка
Eclipse Modeling Project	Ecore (аналог MOF)	Визуальный, текстовый, импорт метамодели	Визуально или вручную поверх библиотек	Текстовые (OCL)
Generic Modeling Environment	Свой, довольно развитый	Визуальный	Настройкой существующих фигур	Текстовые (OCL)
PSL/PSA	Сущность-связь	Текстовый	Нет	Нет
AToM3	Сущность-связь	Визуальный	Визуально	Вручную, на Python
Microsoft Modeling SDK	Свой (диаграммы классов)	Визуальный	Настройкой существующих фигур или кодированием на C#	Вручную поверх библиотеки
Pouamu	Сущность-связь	Визуальный	Визуально	Визуальным языком общего назначения или Java
DOMÉ	Свой (DSTL), довольно развитый	Визуальный	Настройкой существующих фигур или кодированием на Alter	Вручную, на Alter
MetaLanguage	Сущность-связь	Визуальный	Неизвестно	Да
QReal	Свой (диаграммы классов)	Текстовый, визуальный или „метамоделирование на лету“	Визуально	Визуальные

Таблица 2.2: Дополнительные возможности существующих DSM-платформ

Название	Генераторы	Трансформации	Семантика интерпретации	Эволюция
MetaEdit+	Шаблонные, визуальная иерархия	Вручную, через API	Нет	Да, в процессе редактирования моделей
Eclipse Modeling Project	Шаблонные или вручную поверх библиотек	Текстовые DSL (ATL, QVT)	Текстовая (QVT)	Да (Edapt, COPE)
Generic Modeling Environment	Вручную, через API	Вручную (или GReAT)	Нет	Нет
PSL/PSA	Нет	Нет	Нет	Нет
AToM3	Трансформацией моделей	Визуально, графовые грамматики	Визуально, трансформацией моделей и кодом на Python	Нет
Microsoft Modeling SDK	Шаблонные	Вручную	Нет	Вручную
Poupanu	Визуальным языком общего назначения или Java	Визуальным языком общего назначения или Java	Нет	Нет
DOME	Вручную, на Alter	Вручную, на Alter	Нет	Да, в процессе редактирования моделей (с ограничениями)
MetaLanguage	Неизвестно	Да	Неизвестно	Да, в процессе редактирования моделей
QReal	Вручную, на C++ или визуальные	Визуальные — графовые грамматики	Визуальная (Dynamic Meta Modeling)	В режиме „метамоделирования на лету“, для общего случая — в разработке

## Глава 3

# Методология создания DSM-решения

В данной главе рассматриваются основные этапы жизненного цикла визуального предметно-ориентированного языка и рассматриваются возможности по автоматизации каждого этапа. Описывается „классическая“ методология создания визуального языка с использованием метаредактора и методология „моделирования на лету“. Формулируются требования к средствам инструментальной поддержки, реализация этих требований в системе QReal будет описана в главе 4.

## 3.1 Фазы жизненного цикла визуального предметно-ориентированного языка

Визуальные предметно-ориентированные языки, как и все искусственные языки, создаются, развиваются, устаревают и выводятся из эксплуатации. В случае с предметно-ориентированными языками эти процессы проходят относительно быстро и часто повторяются, нередко случается так, что язык создаётся и развивается вместе с решением той задачи, для которой предназначен. Поэтому для предметно-ориентированных языков понимать их жизненный цикл особенно важно, чтобы иметь возможность организовать процесс их массового создания и поддержки. Исходя из проведённого анализа методологий и собственного опыта, будем выделять следующие этапы жизненного цикла предметно-ориентированных языков.

1. Анализ применимости.
2. Анализ предметной области.

3. Проектирование и реализация.
4. Развёртывание.
5. Эволюция языка.
6. Вывод из эксплуатации.

Данные этапы не обязательно проходятся последовательно, более типична ситуация, когда разработка языка ведётся итеративно. Кроме того, возможно даже перекрытие деятельности, ведущихся в разных фазах, так что между фазами часто нет чётких границ. Представленные фазы жизненного цикла общие для всех предметно-ориентированных языков (в том числе и текстовых), конкретная последовательность фаз, через которую проходит проект, определяется моделью жизненного цикла, которая будет введена позже, при описании методологии. Кроме того, видно, что жизненный цикл предметно-ориентированного языка рассматривается здесь в тесной связи с жизненным циклом использующего его предметно-ориентированного решения — существуют языки, живущие независимо от реализующих их инструментов (в том числе и предметно-ориентированные языки, такие как SQL), но в данной работе такие языки не рассматриваются. Реализация предметно-ориентированного решения для уже разработанного языка, как и разработка языка без разработки инструментальных средств его поддержки, являются частными случаями более общей задачи разработки DSM-решения целиком и достаточно нетипичны, чтобы их можно было отдельно не рассматривать.

Некоторые исследователи разделяют фазы проектирования и реализации, а также выделяют в отдельную фазу стабилизацию решения. Несмотря на то, что по нашему опыту решение может находиться в фазе стабилизации годами, мы будем считать её частью фазы проектирования и реализации, поскольку трудно разделить деятельности, ведущиеся на этих этапах. Сами проектирование и реализацию также, как представляется, не следует разделять, поскольку при использовании DSM-платформы довольно большая часть реализации может быть создана автоматически по результатам проектирования (как правило, метамоделю).

Ниже приводится описание основных деятельности, проводимых на каждом этапе, и соображения по поводу возможности их автоматизации.

### **3.1.1 Анализ применимости**

На этом этапе имеется задача, к которой может быть применён предметно-ориентированный подход, однако неясно, насколько оправдано его применение. Задачей фазы анализа применимости является оценка всех факторов, которые могут оказать влияние на успешность применения предметно-ориентированного решения, и вынесение решения о целесообразности начала его разработки. Среди таких факторов наиболее важными являются следующие.

Наличие достаточно узкой предметной области и наличие повторяющихся задач в ней, либо же задачи с большим количеством необходимых для её решения повторяющихся действий.

Наличие экспертизы в предметной области. Перед тем, как создавать своё DSM-решение, как правило, желательно реализовать несколько проектов в предметной области вручную. В любом случае, в разработке должен участвовать человек, обладающий достаточно полным представлением о предметной области.

Наличие команды высококвалифицированных программистов, которые будут заниматься разработкой самого предметно-ориентированного решения. Ошибки и недоделки в инструментальных средствах решения будут проявляться во всех проектах, выполненных с помощью этого решения, поэтому для разработки требуется привлекать по возможности лучших профессионалов.

Наличие необходимых ресурсов и поддержки со стороны руководства. Создание предметно-ориентированного решения сопряжено с довольно высоким уровнем риска и является в некотором смысле долгосрочной инвестицией — преимущества от его внедрения проявятся только через некоторое (иногда довольно значительное) время. Кроме того, внедрение предметно-ориентированного решения может привести к изменениям в структуре и в бизнес-процессах компании, руководство должно быть к этому готово.

Эта фаза наиболее сложна в автоматизации. Оценка должна быть выполнена экспертом на основании собственного опыта и слабоформализуемых знаний.

Могут помочь рекомендации и анкета, приведённые в [7], эта (или подобная) анкета также может быть реализована в каком-либо инструменте, но каждый проект имеет слишком много особенностей, чтобы их можно было учесть и предоставить авторам DSM-решения содержательную помощь. Кроме того, обычно если авторы уже начали пользоваться каким-либо инструментом, решение о начале проекта уже принято. Таким образом, неудивительно, что фаза анализа применимости не автоматизирована ни в одном из известных нам инструментов.

### 3.1.2 Анализ предметной области

На данном этапе используется опыт экспертов предметной области, чтобы выделить ключевые сущности создаваемого языка и отношения между ними. Существует ряд формальных методологий анализа, таких как FAST, FODA, и другие. Однако такие методологии используются довольно редко (поскольку, как правило, ресурсоёмки, и подходят только для больших проектов), чаще используются более неформальные подходы. Некоторые указания по этому поводу можно найти в [33], [35], [36] и опять же в [7]. Следует отметить, что в данном случае методы, применимые для текстовых языков, могут быть перенесены на визуальные языки без изменений, поскольку подходы к анализу предметной области мало зависят от последующей реализации языков. Результатом этой фазы должна стать концептуальная модель предметной области, включающая в себя основные сущности (которые потом станут сущностями создаваемого визуального языка), атрибуты этих сущностей, связи между сущностями. Кроме того, может быть полезно составить словарь предметной области и описание общих и отличающихся частей создаваемого продукта (или линейки программных продуктов), для разработки которого создаётся предметно-ориентированный язык.

Кратко перечислим методы анализа предметной области здесь.

1. Предметно-ориентированные визуальные языки часто строятся над существующей библиотекой. В таком случае концепции можно брать напрямую из существующего кода.
2. Эксперты предметной области могут уже иметь средства выражения знаний предметной области, имеют свой профессиональный язык, но не имеют инструментов для формализации своих знаний. Часто употребляемые



экспертом в разговоре существительные могут стать сущностями, реже употребляемые существительные — атрибутами сущностей, глаголы — операциями или связями.

3. Если язык призван визуализировать аппаратную систему, то хорошие кандидаты для сущностей — физические компоненты.
4. Хорошим источником сущностей может служить пользовательский интерфейс системы.
5. Бывает полезно проанализировать пространство изменчивости предметной области (variability space), тогда язык может визуализировать изменяющиеся её части, а постоянные части будут реализованы в виде предметно-ориентированной библиотеки или фиксированы в генераторе. Такой подход наиболее уместен при разработке линейки программных продуктов.

Инструментальная поддержка фазы анализа может быть весьма развитой, но, как отмечается авторами [33], существующие инструменты такой поддержкой не обладают, оставляя её специализированным инструментам инженерии знаний. Различные методологии анализа предметной области требуют различных инструментов (например, для FODA нужен визуальный язык описания требований, для анализа знаний экспертов достаточно обычной ER-диаграммы). Наличие таких инструментов в DSM-платформе может существенно облегчить и упорядочить сбор и анализ сведений о предметной области, тем более потому, что это делается всё равно, но неформально, или формальными средствами, для этого не предназначенными. По результатам анализа возможно автоматически генерировать прототип метамодели языка, можно организовать трассировку между метамodelью и моделью предметной области, давая возможность отслеживать, как изменения в наших знаниях о предметной области отразятся на метамодели.

Проект QReal ориентирован на „легковесный“ процесс создания предметно-ориентированных языков и предназначен для небольших и средних проектов, поэтому инструментальная поддержка формального анализа предметной области, хотя и является важной и интересной темой, выходит за рамки данной

работы. В QReal предлагается новый подход к анализу предметной области, совмещённому с созданием прототипа метамодели и одновременно с тестированием языка, подробнее об этом в разделе 3.3. Возможно, другие виды анализа будут реализованы в QReal в дальнейшем.

### 3.1.3 Проектирование и реализация

Проектирование и реализация визуального языка — наиболее хорошо изученная и поддерживаемая в существующих инструментах фаза его жизненного цикла. Помимо описания самого языка, обычно требуется реализовать визуальный редактор для него, генератор, предметно-ориентированную библиотеку, другие инструментальные средства (подробнее о составляющих DSM-решения и соображениях по их реализации см. в [7]). На этой фазе опыт разработки текстовых языков оказывается неприменим, поскольку текстовые языки обычно задаются с помощью грамматик, а визуальные языки — с помощью метамodelей.

Типичная деятельность по проектированию и реализации визуального языка включает в себя формализацию абстрактного синтаксиса создаваемого языка (в подавляющем большинстве случаев с помощью метамодели), задание конкретного синтаксиса, автоматическую генерацию по этим описаниям редактора языка, описание в том или ином виде семантики языка. Как правило, задаётся денотационная семантика путём создания генератора из моделей на визуальном языке в какой-либо текстовый язык, но возможно и задание правил интерпретации языка (в том числе, в виде операционной семантики как набора правил преобразования моделей). При использовании генеративного подхода типичные деятельности включают в себя написание модельного приложения, в которое будет производиться генерация из визуального языка „вручную“, разработка генератора, разработка предметно-ориентированной библиотеки, с которой будет работать сгенерированный код. По техническим вопросам реализации существует большое количество литературы, поэтому не будем останавливаться на них подробно.

Отдельный интересный вопрос, который зачастую обходится в работах по реализации предметно-ориентированных визуальных языков, и часто упомина-

ется в работах по реализации языков текстовых — переиспользование синтаксиса существующих языков. Как и в случае с текстовыми языками, визуальный язык вовсе не обязательно создавать с нуля, можно использовать уже существующие языки и использовать их как базу для создаваемого, путём создания „лёгких“ расширений (например, с помощью механизма профилей UML), либо путём переиспользования и расширения частей существующих метамodelей. Существующие технологии имеют тенденцию поощрять создание DSL с нуля, это не всегда оптимально. А чтобы поддержать переиспользование метамodelей, в DSM-платформе должны наличествовать специальные средства, такие как просмотрщики метамodelей, возможность декомпозиции метамodelи и импорта.

Требования к инструментальной поддержке фазы реализации подробно описаны в литературе (например, в [7]). Тут укажем лишь те требования, которые были положены в основу проекта QReal.

1. Наличие визуального языка описания метамodelи и визуального редактора конкретного синтаксиса языка (редактора формы фигур).
2. Наличие возможности визуально задавать ограничения на модели, семантику интерпретации языка и правила рефакторингов.
3. Возможность автоматически сгенерировать редактор визуального языка по метамodelи и описанию конкретного синтаксиса.
4. Возможность автоматически сгенерировать средства проверки ограничений, интерпретатор языка, средства применения рефакторингов.
5. Возможно меньшее время одного цикла „редактирование метамodelи — тестирование языка“.
6. Принцип „не надо знать то, чем не пользуешься“ — с помощью базовых конструкций метаредактора и редактора формы фигур должно быть возможно создать полностью работоспособный редактор и добавлять в него возможности, такие как ограничения или интерпретацию моделей, только по мере необходимости.

То, какое влияние эти требования оказывают на предлагаемую в данной работе методологию разработки, описано в разделах 3.2 и 3.3. То, как эти требования реализованы в DSM-платформе QReal, описано в главе 4.

### 3.1.4 Развёртывание

Фаза развёртывания предполагает наличие инструментальных средств поддержки языка, которые необходимо, во-первых, установить на рабочих местах пользователей, а во-вторых, обучить пользователей работе с ними. Подготовка инсталляционного пакета особых сложностей не представляет (при этом наличие в DSM-платформе поддержки создания инсталлятора созданного DSM-решения было бы весьма желательно), основное внимание на фазе развёртывания уделяется работе с пользователями. Несмотря на то, что визуальные предметно-ориентированные языки специально создаются для удобства использования конкретной группой пользователей, существуют тенденции по их отторжению. Связано это с тем, что, во-первых, люди привыкли пользоваться текстовыми языками программирования (в тех случаях, когда это не так, например, при внедрении технологии QReal:Robots, описанной в разделе A.1, процесс внедрения проходит легче), во-вторых, предметно-ориентированные языки практически всегда будут новыми для пользователей. Тратить значительное время на изучение предметно-ориентированной технологии пользователи не хотят по понятным причинам — будучи применимыми только в очень узкой сфере, эти знания не повысят их стоимость на рынке труда. Кроме того, DSM-решение, как и любой новый продукт, имеет тенденцию содержать ошибки, что будет ещё одной причиной отторжения. Поэтому DSM-решение должно быть максимально удобно для пользователя, по возможности не требовать специального обучения, быть стабильным, и команда его разработчиков должна быть готова быстро вносить в него изменения по запросам пользователей.

От инструментальной поддержки на этой фазе требуется наличие автоматизации создания документации. Практика показывает, что очень полезными оказываются всплывающие подсказки к элементам визуального языка (настолько, что в проекте QReal подсказки задаются в определении элементов в метамодели). Документация тоже необходима, и иметь средства, связывающие ме-

тамодель и документацию, может быть полезно, но не следует рассчитывать, что пользователи документацию обязательно прочитают. От DSM-платформы также требуется возможность тонкой настройки пользовательского интерфейса, что может быть тяжело организовать, поскольку DSM-платформа должна позволять создавать совершенно разные DSM-решения, различия в пользовательском интерфейсе между которыми трудно формализовать и обобщить.

### 3.1.5 Эволюция языка

В процессе эксплуатации созданного DSM-решения у пользователей возникает множество замечаний и предложений. Связано это, с одной стороны, с тем, что разработчики DSM-решения очень редко являются специалистами в предметной области, для которой это решение предназначается, и проблемы в их понимании предметной области становятся очевидными, когда решением начинают пользоваться эксперты. С другой стороны, сама предметная область обычно изменчива: меняются бизнес-процессы, нормативные документы, понимание решаемых задач и т.д. Кроме того, по мере использования возникают новые пожелания к системе, как и в случае любого другого программного продукта. Всё это приводит к тому, что как DSM-решение в целом, так и визуальный язык, который лежит в его основе, не может оставаться фиксированным и непрерывно эволюционирует. Эта важная фаза жизненного цикла часто попросту игнорируется исследователями и авторами инструментов.

Инструментальная поддержка фазы эволюции должна присутствовать в любой DSM-платформе, которую предполагается использовать в промышленных проектах. Прежде всего, пользователи не должны терять результаты своей работы: при изменении визуального языка и его метамодели, модели, созданные в соответствии со старыми метамоделями, должны продолжать открываться в редакторе. Существует несколько подходов к тому, как это обеспечить, самый простой — снабдить модели информацией о версии метамодели, с помощью которой они были созданы, и вручную создавать конвертеры, преобразующие модели в более новые версии. При этом могут быть полезны возможности DSM-платформы, такие как средства задания преобразования моделей (так, например, делается в Eclipse Modeling Project, правила миграции моделей описывают-

ся как трансформации „модель-в-модель“). Такой подход чрезвычайно распространён и в текстовых средах разработки, современные IDE часто предлагают сконвертировать проект при открытии его в более новой версии. Более сложный подход — постараться организовать миграцию моделей автоматически, по информации о различиях между версиями метамodelей. Это не всегда возможно без вмешательства пользователя — например, тип какого-либо свойства поменялся со строкового на числовой, и если в существующей модели значение этого свойства не приводится к числу, пользователь должен указать новое значение для свойства самостоятельно. В таких случаях обычно новая версия редактора в состоянии работать с моделями, созданными со старой версией метамodelей, но изменения каким-то образом указываются пользователю. Например, в системе MetaEdit+ при удалении элемента из метамodelи он продолжает быть видимым и редактируемым в старых моделях, но новые элементы такого типа создавать нельзя, а старые элементы отмечаются как устаревшие, и существует возможность получить список устаревших элементов модели.

Кроме того, необходима поддержка эволюции языка не только для пользователей, но и для авторов DSM-решения. Визуальный язык следует рассматривать как часть проекта, для которого он создаётся, и описание языка должно существовать на равных правах с остальным кодом и моделями, на этом языке созданными. Например, метамodelь языка должна храниться в системе контроля версий, и для DSM-платформы желательна интеграция с такими системами (хотя и не обязательна, в силу наличия зрелых и удобных в работе отдельных инструментов). Необходимо иметь возможность сравнивать различные версии метамodelей.

Желательно также иметь возможность анализировать использование созданного языка в „рабочих“ условиях. DSM-решение может собирать статистику использования элементов языка, статистику возникающих ошибок пользователя, частоту изменения свойств и настроек по умолчанию. Это поможет авторам выявить неиспользуемые или неправильно используемые элементы, актуальность предлагаемых умолчаний, и внести в описание языка соответствующие изменения, чтобы повысить удобство его использования.

### 3.1.6 Вывод из эксплуатации

Вывод из эксплуатации — фаза, присущая всем программным продуктам. DSM-решение может стать неактуальным, если предметная область изменилась настолько, что вышла за рамки решения, проект, в котором использовалось DSM-решение, закрыт, либо DSM-решение устарело. В зависимости от ситуации могут потребоваться разные действия, такие как начало нового проекта по разработке нового DSM-решения, реинжиниринг старого решения, полный вывод решения из эксплуатации. Инструментальная поддержка этой фазы со стороны DSM-платформы обычно не требуется, однако забывать об этой фазе не следует.

## 3.2 „Классическая“ методология

На основе представленных фаз жизненного цикла визуального языка можно предложить модели жизненного цикла DSM-решения и методологии на их основе. Первая методология, поддерживаемая DSM-платформой QReal и предлагаемая в данной работе, — методология, основанная на применении визуального метаредактора. Такой подход к разработке визуальных языков реализуется в большинстве существующих DSM-платформ, поэтому его можно назвать „классическим“ подходом, и здесь автор не претендует на научную новизну. Вклад данной работы состоит в описании этой методологии в той форме, в которой её рекомендуется применять, и представлении реализации её инструментальной поддержки, описанной в главе 4.

Поскольку в рамках проекта QReal никогда не создавались по-настоящему большие визуальные языки и DSM-решения, методология предназначена и апробирована для небольших и средних DSM-проектов. Что здесь понимается под „небольшими и средними проектами“: в [36] упоминаются языки, состоящие более чем из 500 сущностей, авторы сравнивают это с UML, где по их данным 286 сущностей в метамодели. Сейчас самый большой язык, созданный с помощью QReal, имеет 43 сущности в метамодели, разработка соответствующего DSM-решения велась в течение двух лет командой в среднем из 3-4 человек, это

DSM-решение описано в разделе A.1 данной работы. Будем называть средними проектами проекты примерно такого масштаба.

Большинство авторов обращают особое внимание на итеративность процесса разработки визуального языка. Итерации необходимы, чтобы последовательно уточнять понимание предметной области и получать обратную связь от пользователей тогда, когда вносить изменение в проект ещё не поздно. Кроме того, следуя [7], будем выделять отдельно первую итерацию — доказательство осуществимости (proof of concept). Она особенно важна, поскольку часто к самому DSM-подходу, как и ко всему новому, относятся с некоторым подозрением. Доказательство осуществимости заключается в выделении какого-то небольшого фрагмента предметной области и быстрой реализации инструментария только для него — с языком, редактором, генератором и т.д., чтобы результатом этой итерации уже можно было пользоваться и было видно, что он уже улучшает производительность труда. В отличие от [7], мы не будем выделять отдельно пилотный проект. Опыт разработки решений в рамках проекта QReal показывает, что у небольших проектов нельзя выделить время, когда решение готово и может быть использовано для пилотного внедрения. Внедрение обычно начинается прямо с готовности первого прототипа, пользователей желательно вовлекать прямо в процесс разработки.

В остальном, модель жизненного цикла, используемого в „классической“ методологии, похожа на предлагаемую в [40], её общая схема представлена на рисунке 3.1.

Деятельность на фазе анализа применимости была описана в разделе 3.1.1 данной работы, её результатом является решение о том, имеет ли смысл применять предметно-ориентированный подход для данной ситуации. Этот этап не имеет инструментальной поддержки и требует опыта экспертов в предметно-ориентированном подходе.

Анализ предметной области в предлагаемой методологии проводится неформально, согласно рекомендациям из 3.1.2 данной работы. Специальной инструментальной поддержки этого этапа в данном случае не предполагается, впрочем, для небольших проектов, она и не требуется. Этап анализа предметной области может перекрываться с этапом анализа применимости, поскольку для принятия решения о применении DSM-подхода могут требоваться знания о





Рисунок 3.1: „Классическая“ методология разработки визуального предметно-ориентированного языка.

предметной области. Этот этап также может перекрываться с первой итерацией разработки, поскольку для записи знаний о предметной области может использоваться метаредактор (это рекомендуемая практика, набросок метамодели полезен как для понимания взаимосвязи основных концепций предметной области, так и как то, из чего может быть получен первый прототип редактора языка).

Следующий этап, доказательство осуществимости (proof of concept) является первым этапом разработки и имеет внутреннюю структуру такую же, как одна из итераций следующего этапа, итерации разработки и внедрения. Отличие его от последующих итераций в том, что он не имеет своей целью реализацию всего, что необходимо, и относительно короток по времени. На этом этапе выбирается какой-то узкий срез предметной области и для него строится законченное DSM-решение, с редактором, генератором, предметно-ориентированной библиотекой и т.д. Задача этого этапа — убедить будущих пользователей и руководство в применимости DSM-подхода, получить раннюю обратную связь и самим убедиться в осуществимости проекта. Результатом должен являться первый ограниченный в функциональности прототип, демонстрирующий, тем

не менее, все составные части будущего продукта. На основании результатов этого этапа может быть принято решение отказаться от продолжения проекта.

Разработка и внедрение состоит из нескольких итераций, порядок действий для каждой из которых представлен на рисунке 3.2. Начинается разработка с определения метамодели языка с помощью визуального метаязыка, либо внесения изменений в уже существующую метамодель. Как только абстрактный синтаксис задан, следующие действия можно выполнять параллельно.

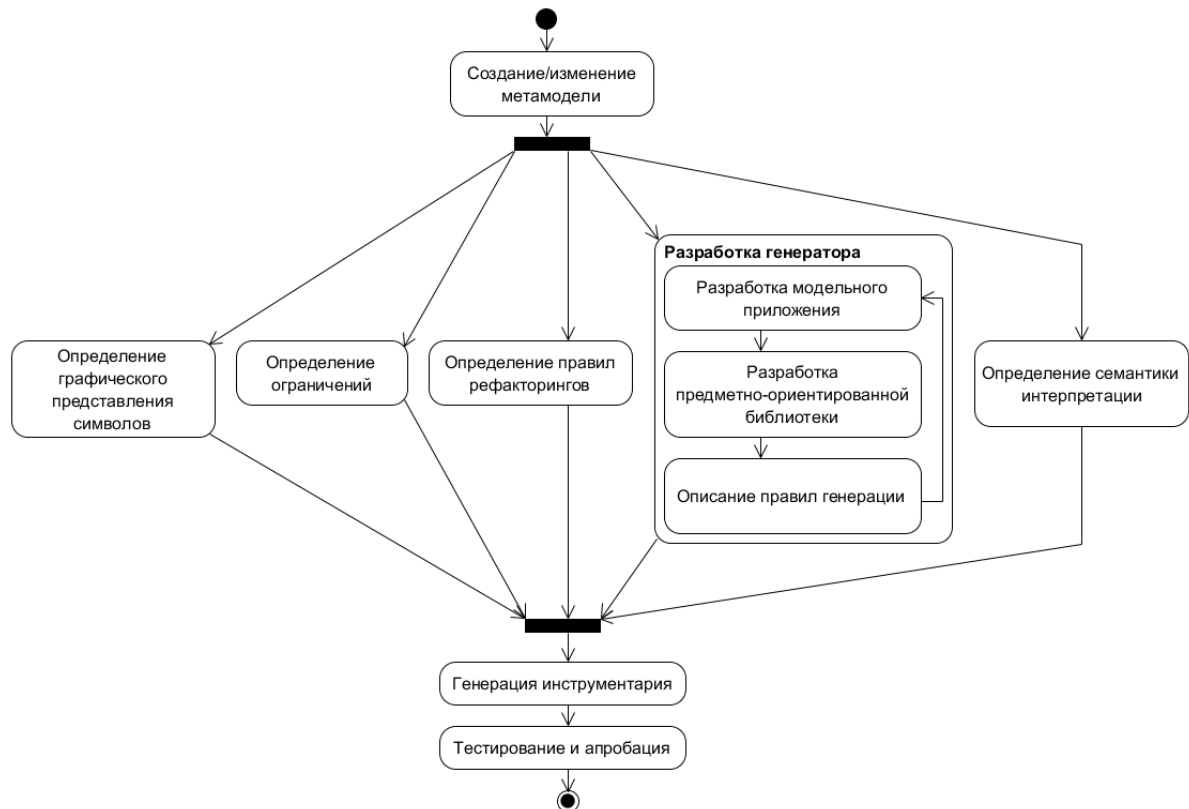


Рисунок 3.2: Итерация проектирования и разработки языка.

1. Определение графического представления символов языка. Это должно делаться либо в графическом редакторе формы фигур, либо с помощью декларативного текстового языка описания формы, DSM-платформа должна минимизировать ручное кодирование. Связано это с тем, что разработку предполагается вести короткими итерациями, и тратить время на кодирование и отладку весьма нежелательно.
2. Задание ограничений на модели. Здесь речь идёт не про ограничения на состояние создаваемой с помощью DSM-решения системы во время вы-

полнения, а про ограничения на модели, создаваемые при помощи DSM-решения, которые проверяются при рисовании моделей. Ограничения могут быть заданы различными способами, насколько это позволяет выбранная для реализации DSM-платформа. В проекте QReal ограничения задаются с помощью визуального языка, при этом модель ограничений отделена от метамодели и не обязательна для создания редактора. Это позволяет авторам языков не задумываться об ограничениях до тех пор, пока необходимость их использования не станет очевидной.

3. Определение правил рефакторингов моделей — довольно редко встречающаяся в DSM-платформах функциональность, но, поскольку автоматизация рефакторингов становится всё более распространённой в текстовых средах разработки, представляется, что скоро и инструменты визуального программирования без рефакторингов будут немыслимы. Простой пример рефакторинга — вынесение фрагмента диаграммы на языке описания поведения в подпрограмму. Рефакторинги специфичны для конкретных языков, поэтому их необходимо задавать на уровне метамодели, а не поддерживать вручную в DSM-решении. Как и в случае с ограничениями, используемый инструментарий должен позволять не задумываться над рефакторингами без нужды и генерировать инструменты без них. Кроме того, правила задания рефакторингов должны описываться на визуальном предметно-ориентированном языке, опять-таки, по той причине, что при быстром цикле итераций, предполагаемом данной методологией, на ручное кодирование тратить усилия нежелательно.
4. Разработка семантики языка может осуществляться в двух видах, либо путём описания правил интерпретации, либо путём создания генератора в какой-либо текстовый язык. Наиболее часто требуется только порождать по моделям код, тогда используется генератор, иногда бывает полезно делать и то, и другое (например, интерпретатор для интерактивной отладки на уровне модели и генератор для порождения результирующего кода). Иногда генерировать текстовое представление программы не требуется вовсе (например, в режиме удалённого управления роботом или

симуляции на двухмерной модели в примере из раздела A.1). Подходы к заданию этих двух видов описания семантики совершенно разные.

- (a) Семантика интерпретации, как правило, задаётся в виде правил преобразования графа модели, сродни рефакторингам. Интерпретатор применяет эти правила до тех пор, пока существует хотя бы одно правило, которое он может применить. Правила преобразования могут иметь побочные эффекты, с помощью которых может быть реализовано взаимодействие с внешними компонентами или устройствами, вычисление выражений на текстовом языке, встроенном в визуальный, и даже генерация. Семантика интерпретации должна задаваться на визуальном языке (наиболее удобный формализм для этого — графовые грамматики), побочные эффекты могут записываться на текстовом языке.
- (b) Разработка генератора в текстовый язык обычно несколько сложнее. Процесс разработки состоит из трёх шагов, как правило, выполняемых итеративно.
  - i. Разработка модельного приложения. Модельное приложение здесь — это пример того, что требуется сгенерировать. Писать генератор сразу же, без чёткого понимания, что именно должно быть сгенерировано, очень сложно, поэтому сначала требуется создать пример приложения вручную и добиться его работоспособности. Пример должен быть простым, но покрывать все разумные случаи, которые могут быть сгенерированы по диаграмме. При этом при написании примера необходимо помнить о том, что его потом придётся генерировать, так что структура программы должна быть удобной для генерации. Для разработки примера используются средства программирования целевой платформы. После того, как приложение отлажено, в DSM-решении рисуется модель, по которой оно могло бы быть сгенерировано.
  - ii. Когда модельное приложение готово, его части, которые не будут меняться от модели к модели, имеет смысл вынести в отдель-

ную библиотеку, написанную вручную. Такая библиотека называется библиотекой времени выполнения (runtime) и нужна для того, чтобы упростить написание генератора. Она может быть довольно сложной (вплоть до того, что представлять из себя полноценное приложение, которое лишь конфигурируется результатом генерации), её может вообще не быть (если генерируются достаточно общие приложения, которым достаточно возможностей целевой платформы). Подробнее соображения по разработке предметно-ориентированной библиотеки и балансе функциональности между ней и генератором см. в [7].

- iii. Разработка собственно генератора, как правило, заключается в „шаблонизации“ готового модельного приложения — места, параметризуемые информацией из модели, размечаются директивами, руководствуясь которыми генератор формирует целевой код. Конкретный способ задания правил генерации зависит от возможностей DSM-платформы, как правило, используется текстовый язык описания правил генерации либо генератор пишется вручную. В проекте QReal также используется текстовый предметно-ориентированный язык или рукописные генераторы, велись работы и по созданию визуального языка для этой цели. В процессе разработки генератора часто приходится править модельное приложение, чтобы сделать его структуру более пригодной для генерации, после чего корректировать библиотеку времени выполнения, поэтому процесс разработки генератора итеративен.

5. Следующий шаг, генерация инструментария, должен производиться с возможно меньшим участием человека. Кроме того, генерация должна быть достаточно быстрой, и результат генерации должно быть можно сразу же проверить в среде. Это важно, поскольку тогда результат внесения своих изменений разработчик языка может проверить, ещё помня о них, и не переключая своё внимание на рутинные действия.

6. Тестирование и апробация созданных инструментов проводятся сначала разработчиком, затем экспертами предметной области, для которых создаётся язык. Для начальных итераций обычно достаточно наличия одного эксперта, который бы попробовал воспользоваться созданным DSM-решением и высказал замечания, для более поздних итераций требуется расширять базу пользователей и давать им возможность решать реальные практические задачи. Чем раньше это будет сделано, тем быстрее будет получена обратная связь и тем быстрее можно будет внести коррективы в создаваемый инструментарий или выявить нужную пользователям функциональность, которая пока не была реализована. На этом этапе потребуется сборка инсталляционного пакета для DSM-решения, написание документации, исправление большого количества мелких ошибок, демонстрация инструмента пользователям, наблюдение за их действиями в инструменте, сбор обратной связи. По результатам апробации вносятся необходимые коррективы в концептуальную модель и в метамодель языка, после чего цикл разработки повторяется заново.

Следующий этап, поддержка и сопровождение, включает в себя прежде всего исправление ошибок в инструментарии и расширение или корректировку визуального языка. К этому моменту основная разработка должна быть завершена, поэтому сопровождение осуществляется меньшей командой разработчиков. На этом этапе важным фактором является версионирование метамодели вместе с другими исходными кодами DSM-решения, чтобы не допустить ситуации, когда изменения в метамодели не согласованы с изменениями в инструментах, которые её используют. Кроме того, необходимо следить за совместимостью существующих пользовательских моделей с новыми версиями метамодели языка — пользователи не должны потерять уже проделанную работу. Поддержка и сопровождение обычно состоит из коротких циклов внесения изменений, подготовки, если требуется, средств миграции моделей на новую версию метамодели, подготовку инсталляционного пакета, тестирование и развёртывание новой версии инструментария у пользователей. Даже небольшие DSM-решения могут быть используемыми годами, поэтому поддержка и сопровождение могут быть очень продолжительны по времени.

Вывод из эксплуатации — заключительная фаза жизненного цикла DSM-решения. Устаревшее DSM-решение может быть заменено более новым, может быть произведён реинжиниринг, оно может быть просто снято с сопровождения. При этом в любом случае пользователей следует заранее оповестить о прекращении поддержки и запланировать дальнейшие действия.

### 3.3 Метамоделирование на лету

„Метамоделирование на лету“ — новая методология, разработанная в рамках данной диссертационной работы и призванная оптимизировать начальные этапы „классической“ методологии с целью максимально возможно сократить время от принятия решения о реализации DSM-решения до получения первого работающего прототипа редактора визуального языка, и максимально вовлечь конечного пользователя в процесс разработки. Основной принцип данной методологии состоит в том, что создание визуального языка проходит непосредственно в процессе рисования диаграммы, без использования метаредактора. Для этого требуется специальная поддержка со стороны DSM-платформы, реализация которой в QReal будет описана в главе 4:

1. интерпретация метамоделей, когда редактор не генерируется по метамоделю языка, а имеется настраиваемый редактор, который использует внутреннее представление метамоделей в памяти для получения информации о языке;
2. пользовательский интерфейс над интерпретатором метамоделей, дающий возможность добавлять элементы, удалять элементы, добавлять, редактировать и удалять свойства элементов, менять графическое представление элементов прямо во время работы.

Основной этап предлагаемой методологии — прототипирование языка — начинается сразу после этапа анализа применимости, и идейно близка к методологии, изложенной в работе [38]. Разработчик языка и будущий пользователь работают за одним рабочим местом. При запуске DSM-платформы они видят канву для рисования и пустую палитру. Пользователь объясняет, что он

примерно хотел бы нарисовать, разработчик языка добавляет на палитру новые элементы, определяет для них графическое представление, пользователь рисует. Пользователь может сказать, что такой-то элемент должен содержать такую-то дополнительную информацию, тогда разработчик добавляет элементу новое свойство, задаёт его тип и значение по умолчанию, и пользователь продолжает рисовать диаграмму, используя новое свойство. Через некоторое время пользователь может сам добавлять и редактировать типы элементов, и работа полностью передаётся ему, разработчик языка лишь следит за процессом и консультирует при необходимости пользователя. Работа заканчивается, когда модельное приложение полностью нарисовано, после чего текущая интерпретируемая метамодель сохраняется в виде, пригодном для дальнейшего редактирования в метаредакторе. После этой фазы идут итерации „классической“ методологии по доработке созданного прототипа, дополнению его ограничениями, рефакторингами, интерпретатором и генератором, подготовки инсталляционного пакета, развертывания и сбора обратной связи. На этих этапах пользователи, как и в „классической“ модели, непосредственно в разработке не участвуют, поскольку этапы гораздо более продолжительны во времени и прямо при пользователе выполнены быть не могут.

Модель жизненного цикла языка, использующая „метамоделирование на лету“, представлена на рисунке 3.3.

Здесь не проводится формального или даже неформального анализа предметной области в общепринятом понимании, знания о предметной области извлекаются из эксперта прямо в процессе прототипирования языка. Также отсутствует фаза доказательства осуществимости (proof of concept), поскольку редактор создаётся прямо на глазах пользователя, и пользователь может сразу сказать, то ли это, что он хочет. Разумеется, генератор, предметно-ориентированную библиотеку и т.д. придётся писать потом, так что на самом деле proof of concept просто смещён на первую итерацию разработки, но по окончании прототипирования по крайней мере у одного будущего пользователя DSM-решения есть уверенность в его полезности и применимости (причём, довольно сильная, потому что фактически он сам это решение создал).

Все остальные этапы жизненного цикла и деятельности на этих этапах совпадают с аналогичными этапами в „классической“ методологии (посколь-





Рисунок 3.3: Методология „метамоделирования на лету“.

ку результатом прототипирования будет обычная метамодель). Для небольших проектов эта методология наиболее применима, поскольку в таком случае всю остальную инструментальную поддержку оказывается возможно реализовать за одну-две итерации и получить максимальную выгоду от высокой скорости начального прототипирования языка. При этом наиболее эффективна эта методология в том случае, если пользователем будущего языка станет сам его разработчик.

Инструментальные средства поддержки метамоделирования на лету не могут быть такими же выразительными, как средства, предоставляемые полноценным метаредактором. Связано это с тем, что, функциональность метамоделирования на лету должна быть максимально простой и легковесной, чтобы ею мог пользоваться эксперт предметной области, не разбирающийся в тонкостях создания визуальных языков. По сути эксперт работает с метамоделью языка, но не должен подозревать об этом и тем более не должен владеть связанной с метамоделированием терминологией. Метамодель оказывается скрыта от пользователя. Разумеется, полноценный синтаксис визуального языка таким образом не задать, простой пример — указание того, к каким узлам может быть подключена связь, было бы слишком сложно для такого режима. Поэтому в режиме метамоделирования на лету для метамодели предполагаются разум-

ные умолчания, дающие максимальную свободу действий пользователя — например, связям разрешается соединять любые узлы языка. Это допустимо, поскольку при первоначальном прототипировании пользователь рисует заведомо „правильные“ диаграммы, то есть то, что он хочет увидеть. Для полноценного решения такой подход недопустим, поскольку пользователи неизбежно будут ошибаться, и система должна не давать пользователю совершать ошибок настолько, насколько это возможно. Поэтому после быстрого прототипирования требуется правка метамодели языка в метаредакторе и внесение в неё дополнительных ограничений. Кроме того, сам язык после быстрого прототипирования будет задан не так, как это делалось бы вручную — без учёта инфраструктурных соображений, таких как группировка сущностей языка по пакетам или отношений наследования между элементами. Таким образом, чтобы получить сопровождаемую и переиспользуемую метамодель языка, требуется рефакторинг полученной после метамоделирования на лету метамодели.

Сформулируем требования к интерфейсу и функциональности режима метамоделирования на лету.

1. Вся функциональность метамоделирования на лету должна расширять существующую функциональность редактора диаграмм. С точки зрения интерфейса этот режим должен выглядеть как рисование диаграммы в обычном DSM-решении, с некоторыми дополнительными возможностями.
2. Должна присутствовать возможность добавить элемент на палитру, задав его имя. При этом для элемента задаётся некоторая форма по умолчанию (например, прямоугольник), элемент создаётся без свойств.
3. Должна быть возможность задать внешний вид элемента с использованием редактора формы фигур, при этом все уже существующие на диаграмме элементы должны обновить свой внешний вид.
4. Должна быть возможность удалить элемент из палитры, если его экземпляров нет на диаграмме.
5. Должна быть возможность создавать копию элемента в палитре. При этом его внешний вид и свойства копируются, после чего могут быть редактируемы независимо.

6. Должна быть возможность редактировать свойства элементов с палитры.
  - (a) Добавлять новые свойства, задавая им имя, тип и значение по умолчанию. Существующим на диаграмме элементам новые свойства добавляются автоматически, со значением по умолчанию, заданным для этого свойства. Явно значение по умолчанию при добавлении можно не указывать, в таком случае используется значение по умолчанию для типа свойства (например, свойства целочисленного типа инициализируются значением 0).
  - (b) Удалять существующие свойства. Если на диаграмме присутствуют элементы с удаляемыми свойствами, выдаётся предупреждение с перечислением элементов, имеющих эти свойства, и свойства просто удаляются с диаграммы, если пользователь согласен.
  - (c) Редактировать имя, тип и значение по умолчанию существующих свойств. При этом, если изменяется тип свойства элемента, экземпляры которого присутствуют на диаграмме, должно выполняться автоматическое преобразование, если это возможно. Если невозможно (например, строку, содержащую латинские буквы, преобразуют в число), пользователю выдаётся предупреждение с перечислением экземпляров элементов, имеющих такие свойства.
7. Практически никакие ограничения на рисуемые в этом режиме диаграммы не накладываются (за исключением ограничений, присущих самой среде). Любая связь может быть подключена к любому элементу, любой элемент может содержаться в любом другом элементе. Корректность значений свойств элементов проверяется с точки зрения типа свойства, дополнительные ограничения на значения свойств в этом режиме не накладываются.
8. Метамоделю, полученную в режиме метамоделирования на лету, должно быть можно открыть и редактировать в метаредакторе. Метамоделю должна быть „плоской“ в том смысле, что система не должна делать предположений о структуре и иерархической организации метамодели. Даже в том случае, когда применялось копирование элемента, копия и оригинал

не должны быть связаны. Структурирование модели проводится отдельным этапом, в метаредакторе, по окончании быстрого прототипирования.

Перечисленные требования до некоторой степени поддержаны в DSM-платформе MetaEdit+, там имеется возможность прямо в процессе моделирования менять внешний вид и свойства элемента. При этом для того, чтобы добавить или удалить элемент, необходимо воспользоваться редактором метамодели, так что это не может быть легко сделано самим экспертом без помощи человека, умеющего пользоваться этим инструментом, поэтому можно утверждать, что MetaEdit+ не может быть в полной мере использован как DSM-платформа, поддерживающая данную методологию. Ближе всего к данной методологии методология, предлагаемая инструментом Agentsheets [38], она тоже позволяет работать непосредственно с пользователем и очень быстро создавать визуальные языки (даже с поддержкой интерпретации). Однако, во-первых, технология предполагает программирование на текстовом языке, что делает невозможным её использование без помощи специалиста по созданию языка, во-вторых, она не предполагает доработки метамодели в полноценном метаредакторе, поэтому применима в более узком наборе случаев, чем типичные DSM-платформы, и плохо масштабируется на большие проекты.

## Глава 4

# Поддержка создания предметно-ориентированных решений в системе QReal

В данной главе описываются технологические средства для разработки предметно-ориентированных визуальных языков, реализованные под руководством автора в системе QReal. Описаны средства для разработки языков с использованием метаредактора (в соответствии с „классической“ методологией): сам метаредактор, редактор конкретного синтаксиса, редактор ограничений, редактор правил рефакторингов, средства генерации редакторов и инструментальной поддержки. Также описаны реализованные средства поддержки методологии „метамоделирования на лету“.

## 4.1 Возможности ядра системы QReal

Прежде чем переходить к описанию средств описания визуальных языков, необходимо кратко описать возможности системы, в рамках которой будут работать созданные по метамоделям редакторы и средства инструментальной поддержки. DSM-платформа QReal построена по модульному принципу: имеется абстрактное ядро, реализующее общую для всех языков функциональность и инфраструктуру редакторов, в него как подключаемые модули (плагины) загружаются редакторы, реализующие специфику конкретных языков, и инструменты, реализующие какую-либо ещё специфичную для конкретного DSM-решения функциональность. Ядро не содержит в себе никаких знаний про конкретные языки и работает со всеми DSM-решениями одинаковым образом. Всю

необходимую информацию о языке и действиях, которые можно выполнить над диаграммами, созданными с его помощью, ядро получает из плагинов, либо непосредственно из метамодели языка, если ядро работает в режиме интерпретации метамодели. Общая архитектура системы представлена на рисунке 4.1.

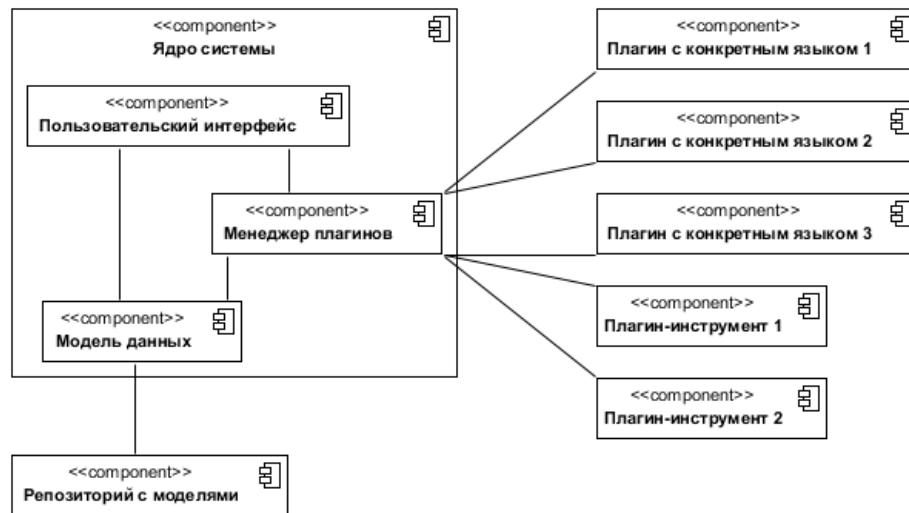


Рисунок 4.1: Общая архитектура системы QReal.

Плагины-инструменты могут настраивать внешний вид и состав интерфейса QReal под свои нужды, базовый интерфейс среды представлен на рисунке 4.2. Интерфейс содержит область для редактирования диаграммы, называемую сценой, редактор свойств, палитру элементов, обозреватели логической и графической моделей, „миникарту“, панель инструментов и меню.

Палитра элементов инициализируется описанными в плагине-редакторе элементами. Одновременно могут быть подключены несколько плагинов, тогда создаётся несколько палитр, между которыми можно переключаться с помощью выпадающего списка. В палитру попадают только те элементы, которые имеют графическое представление, элементы, которые его не имеют, считаются абстрактными и используются в качестве базовых для определения других элементов.

Редактор свойств отображает все свойства выделенного в данный момент элемента и даёт возможность их редактировать. Свойство характеризуется своим именем (задаваемым в метамодели), типом (одним из фиксированного набора элементарных типов, типом-перечислением или типом-ссылкой на другой элемент диаграммы) и значением по умолчанию. Для разных типов редактор

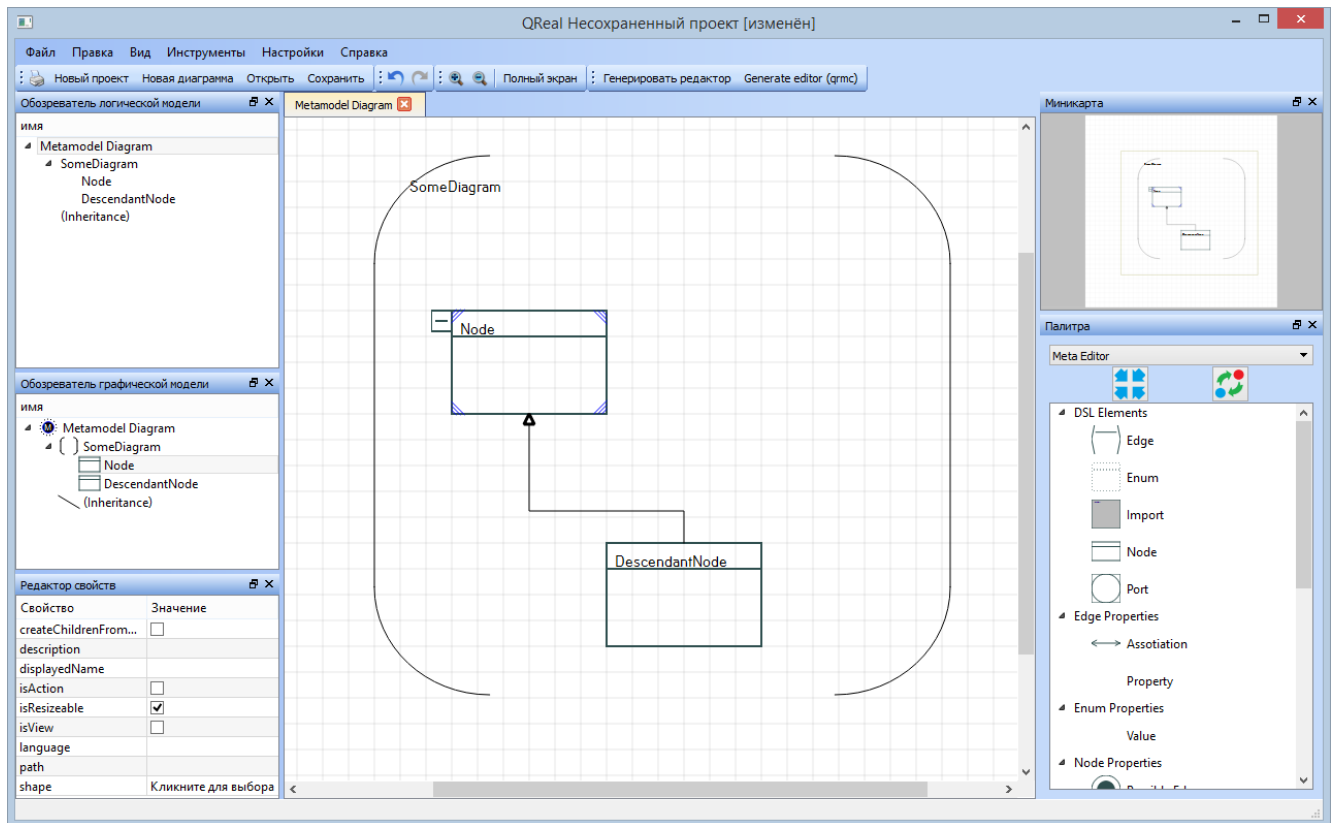


Рисунок 4.2: Пользовательский интерфейс системы QReal.

свойств предлагает разные средства редактирования, например, выпадающие списки для типов-перечислений, поля ввода для строк, „галочки“ для булевых свойств.

Панель инструментов содержит элементы, общие для всех редакторов (сохранение/загрузка, отмена операции и т.д.), и область, в которую добавляются действия из плагинов-инструментов. Действие может вызвать произвольный код из плагина, который может использовать достаточно богатый программный интерфейс ядра QReal, предоставляющий доступ к сохранённой в репозитории модели, сцене (например, возможность подсветки указанного элемента на сцене), возможность подписываться на системные события (например, закрытие вкладки с диаграммой, изменение системных настроек).

Любой редактор визуального языка обязан реализовывать определённый интерфейс, чтобы быть подключенным к ядру системы. Этот интерфейс определяет информацию, которую настраиваемый визуальный редактор, входящий в ядро системы, должен знать о синтаксисе визуального языка, чтобы давать возможность создавать диаграммы. С точки зрения ядра системы подключа-

емый визуальный редактор представляет собой набор диаграмм (различных визуальных языков, входящих в состав редактора, их может быть несколько), каждая диаграмма содержит в себе список элементов. Элемент может быть либо узлом, либо связью. Любой элемент обладает списком свойств (свойства с точки зрения абстрактного синтаксиса представляют собой тройки „имя - тип - значение по умолчанию“) и внешним видом. Внешний вид узла задаётся в векторном графическом формате, близком к SVG<sup>1</sup>, но с некоторыми расширениями (возможностью определить порты, к которым могут подключаться связи и областями для вывода значений свойств из репозитория). Внешний вид связи определяется стилем линии (сплошная, пунктирная и т.д.) и видом конца линии, выбираемым из нескольких предопределённых значений (открытая и закрытая стрелки, закрашенный и незакрашенный ромб и т.д.). Для узла также указывается, какие связи можно к нему подключить и какие узлы он может содержать внутри себя как составные элементы.

Редактор визуального языка можно создать вручную, реализовав описанный выше интерфейс на языке C++ и собрав подключаемую библиотеку. Однако такой подход весьма неэффективен, поэтому был использован лишь однажды, для нужд тестирования системы. Далее речь пойдёт об автоматизации создания визуальных редакторов. Здесь был дан лишь краткий обзор возможностей ядра системы, необходимый для дальнейшего изложения, более подробно см. статьи [11, 12, 60].

## 4.2 Метаредактор

В системе QReal для быстрого создания редакторов используется визуальный метаязык и соответствующий редактор, называемый метаредактором. По визуальной метамодели языка, созданной в метаредакторе, генерируется код на C++, реализующий интерфейс подключаемого модуля, который требуется ядру системы.

---

<sup>1</sup>Домашняя страница формата SVG на сайте консорциума W3, URL: <http://www.w3.org/Graphics/SVG/> (дата обращения 07.09.2014)



### 4.2.1 Визуальный метаязык

В основу метаязыка системы QReal заложены принципы, используемые, например, в метаязыке MOF<sup>2</sup>: элементы моделируемого языка (сущности и связи) моделируются сущностями метаязыка (Node и Edge), эти сущности могут быть связаны отношениями наследования и „является контейнером для“. И сущность, и связь могут иметь свойства, значения для которых можно задавать при редактировании модели в редакторе свойств QReal. Имеется ряд вспомогательных элементов метаязыка, определяющих поведение элементов языка при взаимодействии с ними пользователя, например, позволяет ли контейнер располагать внутри себя элементы произвольным образом, или сам располагает элементы подряд друг под другом. Подробнее об имеющихся элементах см. в приложении В.

### 4.2.2 Особенности языка

Метаязык QReal имеет следующие особенности, важные для создания CASE-инструментов промышленного уровня.

1. Принцип „Если функциональность не требуется, о ней можно не знать“. Метаредактор определяет ряд разумных умолчаний, так что для создания простого языка достаточно владеть всего несколькими элементами метаязыка. Остальные элементы можно добавлять в метамодель по мере необходимости.
2. Возможность определения нескольких языков в рамках одной метамодели. Это даёт возможность реализовать подход, при котором проектируемая система рассматривается с нескольких разных точек зрения, которые, тем не менее, взаимосвязаны и подчиняются единым правилам.
3. Возможность переиспользования фрагментов метамodelей. Это даёт возможность создавать семейства языков, базирующиеся на общей метамодели, а также иметь метамодель-библиотеку для хранения и переиспользова-

---

<sup>2</sup>Страница стандарта MOF на сайте OMG, URL: <http://www.omg.org/mof/> (дата обращения 07.09.2014)

ния основных конструкций. Особенно полезно это при создании диалектов существующих языков, таких как UML.

4. Возможность задания отношения вложенности между элементами. Эта возможность является скорее синтаксическим сахаром, поскольку отношение вложенности легко задаётся через связи (два элемента модели, соединённые отношением „вложен“, семантически эквивалентны одному элементу, вложенному в другой). Тем не менее, это отношение весьма полезно, поскольку работа с элементами-контейнерами и вложенностью требует специальной (и довольно серьёзной) поддержки со стороны редактора.
5. Типизированные порты дают возможность различать места на фигуре, к которым могут быть подключены связи, и по-разному обрабатывать связи в зависимости от того, к какому конкретно порту они подключены. Потенциально таким портам можно указывать количество подключаемых связей, задавая тем самым языки наподобие электрических схем, но эта возможность в QReal пока не реализована.
6. Отношение раскрытия (или эксплозия) появилось в метаязыке как обобщение понятия „провязка“, то есть неграфической связи между элементами. Отношение „провязка“ позволяло определить произвольную связь между элементами, в том числе и на разных диаграммах, которая могла использоваться, например, для связывания случая использования и диаграммы классов, его реализующей, или класса и реализующей его машины состояний. Кроме того, провязкой же могли связываться элементы, просто „знающие“ друг о друге, например, узел ветвления и узел объединения ветвей из диаграммы активностей UML. Семантика провязки определялась для каждого конкретного визуального языка. Раскрытие, в отличие от провязки, позволяет явно задать поведение редактора при работе со связанными этим отношением элементами. Также это отношение напрямую связано с пользовательской палитрой — палитрой, похожей на основную палитру языка, на которую выносятся все созданные пользователем подпрограммы, которые можно использовать как обычные узлы языка. Естественно, это имеет смысл только для языков, в которых

определено понятие „подпрограмма“, поэтому такое поведение задаётся в метаредакторе как свойство отношения раскрытия. Именно с помощью этого отношения реализованы подпрограммы в языке QReal:Robots.

### 4.2.3 Генерация редакторов

Созданную в метаредакторе метамодель можно использовать для того, чтобы получить редактор, несколькими способами: сгенерировать исходный код редактора непосредственно по метамодели, сгенерировать сначала XML-описание, а затем по XML-описанию исходный код редактора, либо открыть метамодель в интерпретаторе метамodelей и обойтись вовсе без генерации. Схематически разные способы представлены на рисунке 4.3.

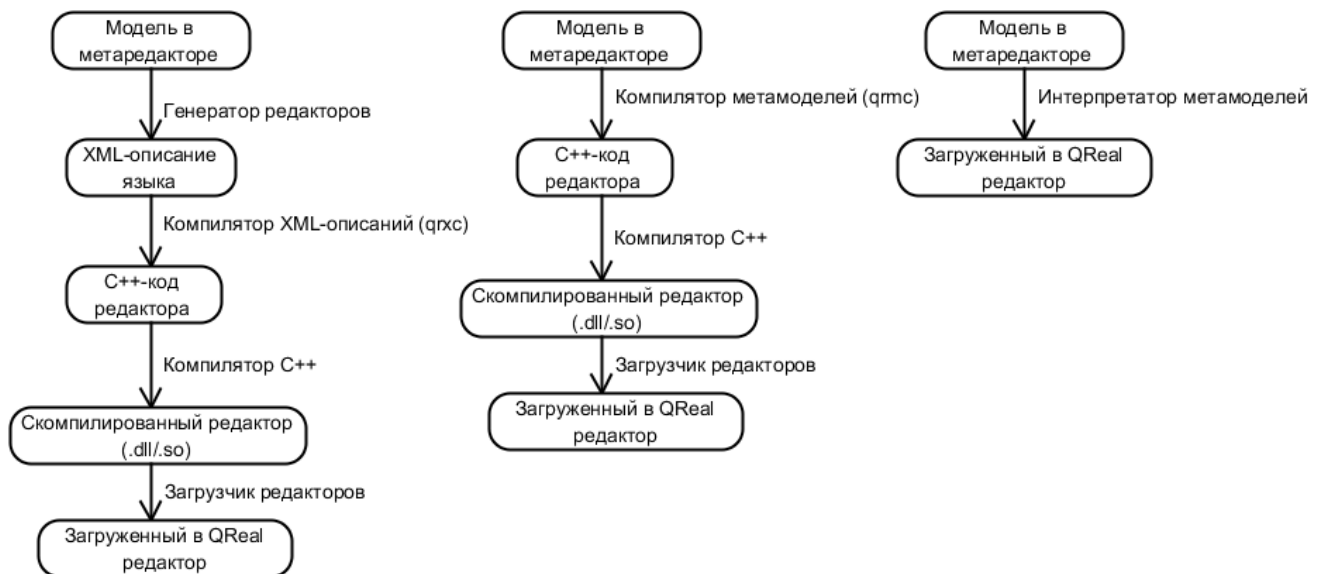


Рисунок 4.3: Получение редактора по метамодели.

Способ, использующий XML-описание метамодели, был реализован в QReal первым. Связано это с тем, что изначально все описания метамodelей задавались в XML-виде, метаредактор появился позднее. Поскольку уже существовала утилита, генерирующая исходный код редактора по XML-описанию, сделать генерацию из визуальной метамодели в XML и поддержать в среде автоматический запуск утилиты, генерирующей код на C++, а затем самого компилятора, оказалось проще. Однако такой подход концептуально сложен: для того, чтобы добавить новую функциональность в метаязык, требуется поддержать её в

самом визуальном метаязыке, в генераторе XML-описания, в утилите, которая генерирует код на C++, и в ядре системы. Поэтому было принято решение генерировать код на C++ по метамодели напрямую, и была разработана вторая утилита, принимающая на вход файл с сохранённой метамоделью. Тем не менее, система сборки QReal до сих пор использует подход с промежуточным XML-представлением, и большинство визуальных языков заданы именно в этом виде. Связано это с тем, что в QReal до сих пор не реализован механизм миграции моделей при изменении метамодели языка, так что при практически любом изменении в метаязыке уже созданные в нём метамодели окажется невозможно открыть в QReal. XML-представление, поскольку легко редактируется вручную, гораздо устойчивее к изменениям, так что нет риска оказаться в ситуации, когда редакторы невозможно собрать автоматически без ручной правки файлов с сохранениями.

Структура XML-файла практически полностью соответствует структуре метамодели языка (точнее, с исторической точки зрения, наоборот, визуальный метаязык появился как визуализация существующего XML-языка). Синтаксис языка был впервые предложен в дипломной работе А.А. Симоновой [61], однако претерпел с тех пор ряд значительных изменений. Новые возможности ядра QReal реализуются сначала в XML-языке, а уже затем переносятся в метаязык, и на данный момент XML-язык обладает возможностями, которые в метаязыке пока отсутствуют. Пример такой возможности — задание групп в палитре. Эта возможность позволяют собрать близкие по смыслу элементы языка и сворачивать или разворачивать в палитре группы элементов. Также есть возможность задать одновременное создание сразу нескольких элементов, например, чтобы при создании диаграммы поведения робота на неё сразу же добавлялся блок „начало“, обязательный для всех диаграмм.

Интерпретатор метамodelей позволяет избежать необходимости генерации кода на C++ и компиляции редактора. Метамодель языка, подготовленная в метаредакторе, загружается в интерпретатор, и он с её помощью эмулирует поведение сгенерированного плагина-редактора. При таком подходе скорость работы системы несколько меньше, чем в случае скомпилированного редактора, зато отсутствие необходимости пересобирать редактор после каждого изменения языка значительно снижает время цикла „разработка-тестирование“

для вносимых в язык изменений. Кроме того, интерпретатор делает возможным внесение в метамодель изменений прямо в процессе работы с редактором языка, поэтому применение такого подхода необходимо для реализации инструментальной поддержки режима „метамоделирования на лету“. Подробно реализация интерпретатора метамodelей описана в курсовой работе Птахиной Алины Ивановны [62]. На данный момент в QReal применяется и генеративный, и интерпретативный подходы к созданию редакторов, интерпретатор используется для быстрого прототипирования языка, генераторы — для финализации языка и создания инсталляционных пакетов, поставляемых пользователям. Для того, чтобы гарантировать, что результирующие редакторы ведут себя одинаково для одной метамodelи языка вне зависимости от способа получения редактора, была разработана и включена в процесс сборки тестирующая система, создающая тестовые редакторы для нескольких разных метамodelей всеми тремя способами и проверяющая, что все функции интерфейса редактора во всех трёх случаях возвращают одинаковый результат.

### 4.3 Редактор формы фигур

Редактор формы фигур предназначен для задания конкретного синтаксиса языка и представляет собой простой векторный редактор, запускаемый из метаредактора при редактировании свойства „Форма“ узла. Использовать существующие графические редакторы оказалось невозможным в связи с тем, что требуется задавать не только внешний вид элемента, но и поведение элемента при взаимодействии с ним пользователя — куда можно подключать связи, как должна изменяться форма элемента при его растягивании или сжатии, где выводить значения свойств из репозитория.

Пользовательский интерфейс редактора формы фигур представлен на рисунке 4.4.

Редактор позволяет использовать изображения, заранее подготовленные в обычных графических редакторах, в качестве элементов создаваемой формы фигуры, например, так реализованы фигуры блоков в QReal:Robots.

Изображения могут быть дополнены точечными и линейными портами. Порт — место на фигуре, к которому можно подключить связь, точечный порт

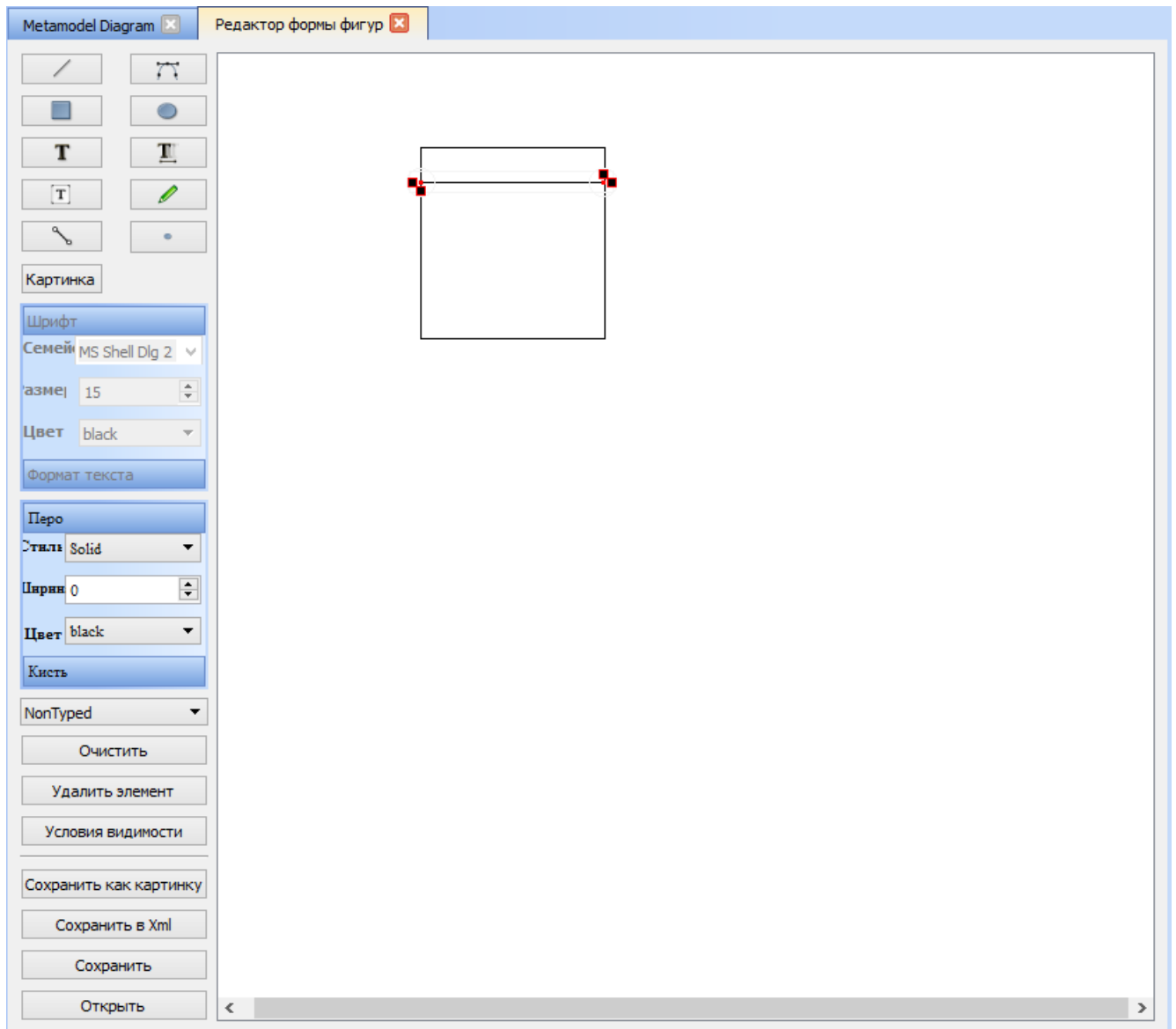


Рисунок 4.4: Редактор формы фигур.

отличается от линейного тем, что к линейному порту связь может подсоединяться в любом месте линии. Порт может иметь тип, один из заранее определённых в метамодели, связи может быть указано, к портам какого типа её разрешено подключать. Для того, чтобы задать тип, надо выделить порт и выбрать его тип из выпадающего списка. Значения в этом списке берутся из метамодели.

Имеется возможность задать отображение статического текста, динамического текста и текста как картинки. Динамический текст — поле на фигуре, текст для которого берётся из свойства фигуры в репозитории. В редакторе формы фигур указывается имя свойства, откуда надо брать значение. В процессе рисования диаграммы на созданном языке подставляется реальное текущее

значение свойства, кроме того, значение свойства можно редактировать прямо на сцене. Статический текст — это не редактируемая надпись на элементе, выполненная в том же стиле, что и динамический текст. В редакторе формы фигур в этом случае указывается сам текст, который должен отображаться. Текст как картинка — это статический текст, которому можно указать произвольный шрифт, стиль начертания, цвет и размер, но при этом его нельзя редактировать, выделять или перемещать, он ведёт себя просто как элемент изображения.

Для каждого элемента изображения можно определить условие, при котором он будет отображаться. Условие задаётся в виде логического выражения, в котором можно использовать значения свойств элемента.

Результат создания формы фигуры может быть сохранён в виде XML-строки как свойство элемента Node в метамодели либо как отдельный XML-файл, который можно переиспользовать при создании других фигур.

## 4.4 Редактор ограничений

Ограничения необходимы для того, чтобы, во-первых, минимизировать вероятность ошибок при разработке системы, во-вторых, чтобы иметь возможность обнаружить ошибку как можно раньше. Наличие средств задания ограничений позволяет существенно повысить удобство пользования визуальной технологией и повысить качество программ, создаваемых с её помощью, поэтому средства задания ограничений очень распространены в существующих DSM-платформах.

Ограничения бывают двух видов — ограничения на состояние разработанной системы во время её работы и ограничения на модель системы в процессе её разработки. Первый вид ограничений схож с операторами `assert` в текстовых языках программирования, второй — с синтаксическими проверками кода, выполняемыми компилятором или средой разработки. В случае DSM-платформы первый вид ограничений может быть реализован только для частных случаев, в рамках разрабатываемых с помощью DSM-платформы визуальных технологий. Второй же вид ограничений может описываться на уровне метамодели визуального языка, в этом случае ограничения могут автоматически проверяться

редактором диаграмм. Далее речь пойдёт исключительно про второй вид ограничений.

Задание ограничений на создаваемую модель в некотором виде происходит в самой метамодели, правила синтаксиса языка не позволят создать некорректную диаграмму, если редактор строго им следует (что в случае DSM-платформ практически всегда так, потому что редактор определяется метамodelью). Насколько сложные ограничения можно задать в метамодели языка, зависит от используемого метаязыка, например, в метаязык можно включить понятие типа для свойств, тогда редактор не позволит при создании диаграмм задавать свойствам значения, не соответствующие их типу. Можно включить в метамодель проверку на соответствие значений свойств регулярным выражениям или логическим условиям, можно продолжить обобщать такой подход: например, метамодель UML использует для задания ограничений полноценный текстовый язык OCL, позволяющий писать сколь угодно сложные запросы к репозиторию с моделью и накладывать сколь угодно сложные ограничения на её состояние.

В системе QReal в соответствии с принципом „не надо знать то, чем не пользуешься“ сложные ограничения задаются отдельно от метамодели, с помощью модели ограничений. Метамодель может быть создана без ограничений, они могут быть добавлены в язык потом, по мере осознания разработчиками языка их необходимости, и отдельно, без внесения изменений в метамодель. Кроме того, в соответствии с идеологией предметно-ориентированного визуального моделирования ограничения задаются с помощью специально созданного для этого предметно-ориентированного визуального языка. Данный язык, а также средства проверки ограничений, записанных с его помощью, были реализованы в рамках курсовой работы Дерипаска Анны Олеговны [63] под руководством автора данной работы, и подробно описаны в отчёте о курсовой работе и в статье [64], ниже приводится лишь краткое описание данных средств.

Модель ограничений создаётся после создания метамодели языка (или совместно с ней), также имеется возможность задавать ограничения, проверяемые для всех метамodelей (например, что связи должны быть обязательно подсоединены к узлам). Модель ограничений состоит из набора ограничений на узлы и связи. Для каждого узла или связи возможно указать ограничения на значение его свойств, для узла можно указать ограничения на входящие и исхо-



дящие связи, на содержащиеся в узле элементы или наоборот, на содержащий узел элемент, для связи — на узлы, с которых начинается или которыми заканчивается связь. Для каждого ограничения можно указать его важность — предупреждение или критическая ошибка. В случае нарушения ограничения-предупреждения элемент-нарушитель подсвечивается на диаграмме, в случае нарушения критического ограничения в дополнение к подсветке в окне ошибок появляется текстовое сообщение.

Пример простого ограничения, примененного ко всем типам связей всех метамоделей, приведён на рисунке 4.5. Ограничение требует, чтобы у любой связи существовал и начальный, и конечный узел (зелёный квадрат означает узел, из которого исходит связь, красный — узел, в который связь входит, true означает, что они должны существовать). Ограничение имеет важность „предупреждение“.

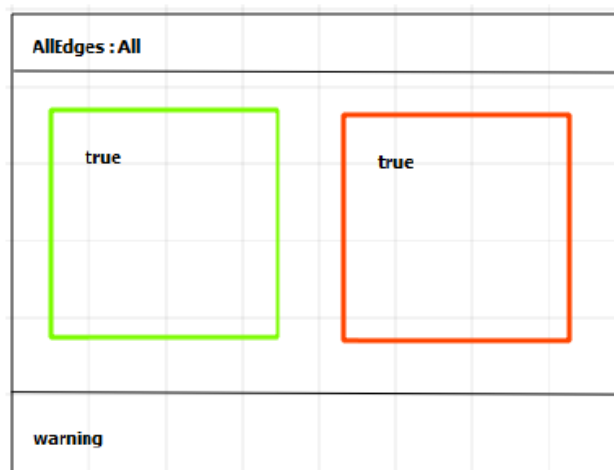


Рисунок 4.5: Ограничение, проверяющее наличие начального и конечного узла у связи.

Пример диаграммы с нарушением этого ограничения приведён на рисунке 4.6. Связь, не имеющая конечного узла, выделена средством проверки ограничений красным.

Механизм проверки ограничений реализован следующим образом. По модели ограничений генерируется код подключаемого модуля проверки ограничений на C++, который компилируется в динамическую библиотеку. При запуске QReal все модули проверки ограничений загружаются в подсистему проверки ограничений ядра QReal. При выполнении пользователем определённых

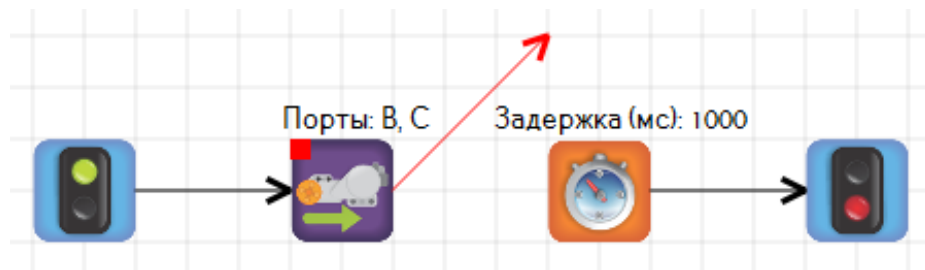


Рисунок 4.6: Пример отображения нарушения ограничения.

действий (таких как изменение имени элемента, изменение значения свойства элемента, создании или удалении элемента, подключение или отключение связи) вызывается обработчик в подсистеме проверки ограничений. Обработчик быстро проверяет наличие ограничения на тип изменённого элемента в списке подключённых модулей проверки, а также для всех связей и содержащих элемент элементов, и наоборот, элементов, содержащихся в данном. В случае, если в подключённых модулях хоть одно такое ограничение есть, для каждого ограничения вызывается его проверка, результаты собираются в список и, если есть невыполненные ограничения, информация об этом предоставляется пользователю в виде подсветки элементов на диаграмме и, возможно, в текстовом виде в окне ошибок. Таким образом, несмотря на то, что проверка ограничений вызывается при практически любом редактировании модели, за счёт фильтрации ограничений по типу проверяемого объекта это не приводит к существенной потере производительности.

## 4.5 Редактор правил рефакторинга

Возможность автоматически применять рефакторинги для диаграмм довольно редка среди CASE-систем и DSM-платформ, но стала вполне обычной для текстовых сред программирования, поэтому ожидается и от создаваемых визуальных сред, чтобы успешно конкурировать с текстовыми. Рефакторинг определяется как изменение внутренней структуры программы без изменения её видимого поведения с целью сделать её проще для понимания и упростить её сопровождение [65]. В случае с визуальными моделями рефакторинг может рассматриваться как изменение логической модели системы или как изменение

внешнего вида диаграммы, не затрагивающее логическую модель (например, перерасположение элементов). Второй вид рефакторинга исследован достаточно хорошо (см. Dot, KIELER и т.д.), и система QReal использует стороннюю компоненту GraphViz<sup>3</sup> для автоматической раскладки элементов на диаграммах. Первый вид рефакторингов изучен гораздо меньше.

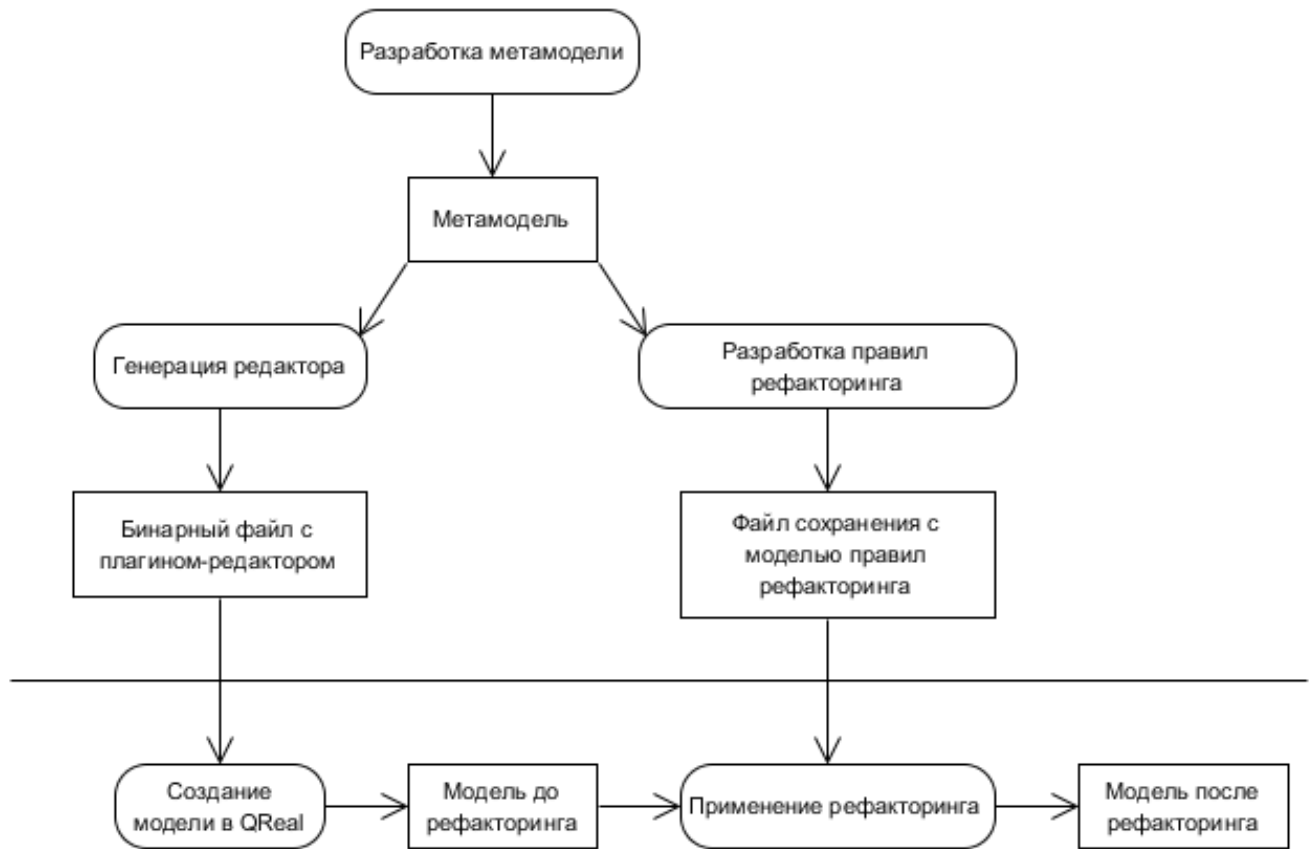
В общем случае схема рефакторинга визуальной модели такова [66]. Система состоит из нескольких визуальных моделей, характеризующих систему с разных точек зрения, по моделям автоматически генерируется часть кода, другая часть кода, возможно, пишется вручную. При рефакторинге необходимо, во-первых, внести необходимые изменения в редактируемую модель, во-вторых, синхронизировать изменения с другими моделями, в-третьих, выполнить регенерацию кода по моделям, затронутым изменениями, в-четвёртых, согласовать рукописный код. Рефакторинг вручную может оказаться весьма трудоёмким процессом, поэтому естественно желание его автоматизировать. Но для каждого визуального языка набор рефакторингов может быть свой, зависящий от семантики конкретного языка. Поэтому требуется задавать правила рефакторингов на уровне метамодели. В QReal для задания рефакторингов принят тот же подход, что и для задания ограничений — правила рефакторингов, если они требуются, определяются в отдельной модели после определения метамодели языка, для этого используется специализированный визуальный язык. Этот визуальный язык и средства, позволяющие применять заданные с его помощью рефакторинги, был реализован в рамках курсовой работы Кузенковой Анастасии Сергеевны под руководством автора данной работы. Ниже приводится краткое описание реализации системы рефакторингов в QReal, подробнее см. в отчёте по курсовой работе [67], а также в статьях [68, 69]. Общая схема разработки и применения правил рефакторинга представлена на рисунке 4.7.

Язык описания правил рефакторинга имеет интересную особенность — его элементами являются элементы языка, для которого определяется правило рефакторинга. Эти элементы автоматически подгружаются из метамодели разрабатываемого языка редактором правил рефакторинга и используются для определения шаблонов, которые будет сопоставлять в диаграмме средство выполнения рефакторинга при его применении. Сами правила задаются в виде

---

<sup>3</sup>Домашняя страница GraphViz, URL: <http://www.graphviz.org/> (дата обращения 07.09.2014)

## Разработка языка



## Использование языка

Рисунок 4.7: Схема разработки и применения правил рефакторинга.

графовых грамматик: правило делится на часть BEFORE, в которой задаётся шаблон, который ищется в диаграмме, и на часть AFTER, на которую замещается сопоставленный шаблон. При этом в правиле можно использовать специальные элементы, такие как „Выделенный фрагмент диаграммы“, каждому элементу можно указать его идентификатор. Элементы, имеющие одинаковый идентификатор в левой и правой части правила, считаются одним элементом и будут сохранены (возможно, с изменениями) при рефакторинге. Элементы, идентификатор которых не встречается в части BEFORE, будут созданы, элементы, идентификатор которых не встречается в части AFTER — удалены. Можно модифицировать существующее значение свойства элемента, для доступа к нему используется ключевое слово EXISTS. Пример правила рефакторинга для метаязыка, который к имени каждого узла приписывает префикс „robot“, представлен на рисунке 4.8. На рисунке в части BEFORE задаётся шаблон, в ча-

сти AFTER — то, на что нужно заменить шаблон после сопоставления, стрелка между частями правила не несёт семантической нагрузки и нужна лишь чтобы сделать правило более читаемым. Цифра „1“ справа от узлов в правиле — идентификатор узлов, по которому производится их сопоставление.

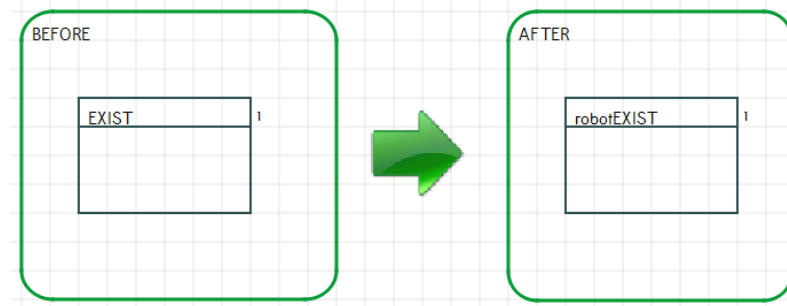


Рисунок 4.8: Пример правила рефакторинга, дописывающего префикс „robot“ к имени узла.

После того, как правило создано, оно сохраняется (в виде обычного файла сохранения QReal), и при создании диаграммы на языке, для которого оно было определено, может быть применено. Для этого пользователю доступно диалоговое окно, в котором он может выбрать применяемый рефакторинг, найти следующее соответствие шаблона рефакторинга в диаграмме и применить рефакторинг к найденному соответствию. Внешний вид этого диалогового окна представлен на рисунке 4.9.

С технической точки зрения средство применения рефакторингов использует тот же механизм работы с графовыми грамматиками, что и используемый для задания семантики интерпретации визуальных языков. Этот механизм, а также существующие его аналоги, подробно описан в статьях [70–73]. Физически средство применения рефакторингов реализовано как подключаемый модуль к QReal, поскольку, в отличие от подсистемы проверки ограничений, не требует тесной интеграции с ядром системы.

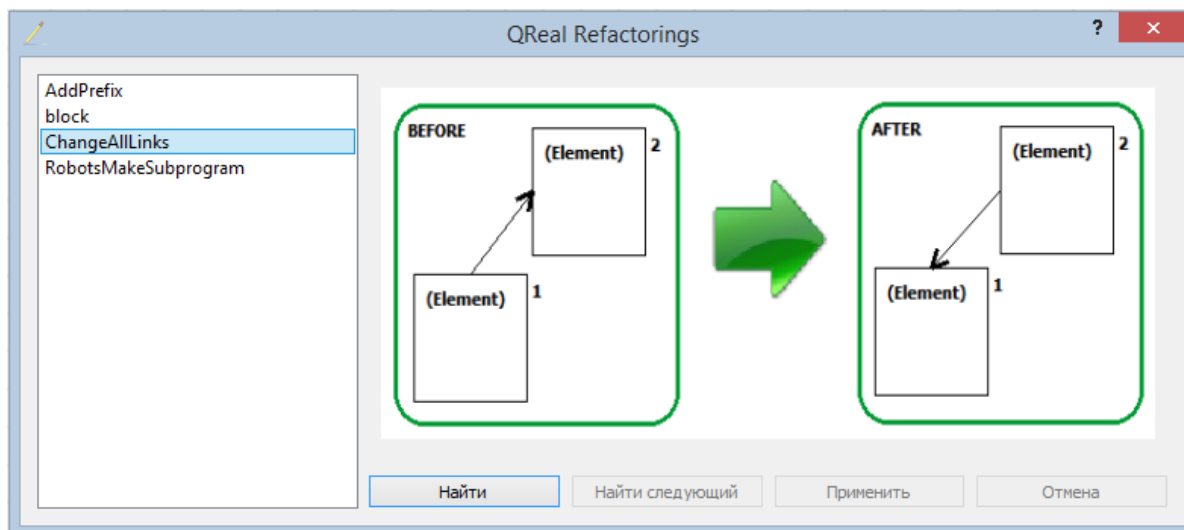


Рисунок 4.9: Диалоговое окно применения рефакторинга, с рефакторингом „обращение всех связей“.

## 4.6 Средства поддержки технологии „метамоделирования на лету“

Технология „метамоделирования на лету“ позволяет разрабатывать предметно-ориентированные языки, не используя метаредактор, прямо в процессе рисования диаграммы на разрабатываемом языке. Методологические аспекты технологии, включая описание процесса создания языка в соответствии с ней, описаны в разделе 3.3 настоящей работы, здесь будет кратко описана реализация инструментальных средств поддержки этой технологии в системе QReal, выполненная под руководством автора Птахиной Алиной Ивановной в рамках курсовой работы. Подробное описание реализации можно найти в тексте курсовой работы [74] и в статье [75].

Метамоделирование на лету — это особый режим работы QReal, не доступный при работе с такими технологиями, как QReal:Robots. Таким образом, пользователи предметно-ориентированных решений, созданных на основе QReal, имеют возможность даже не знать о существовании такой функциональности. Если же QReal запущен в режиме DSM-платформы, на стартовом экране будет предложено выбрать между метаредактором, или созданием или открытием интерпретируемой метамодел, как показано на рисунке 4.10.

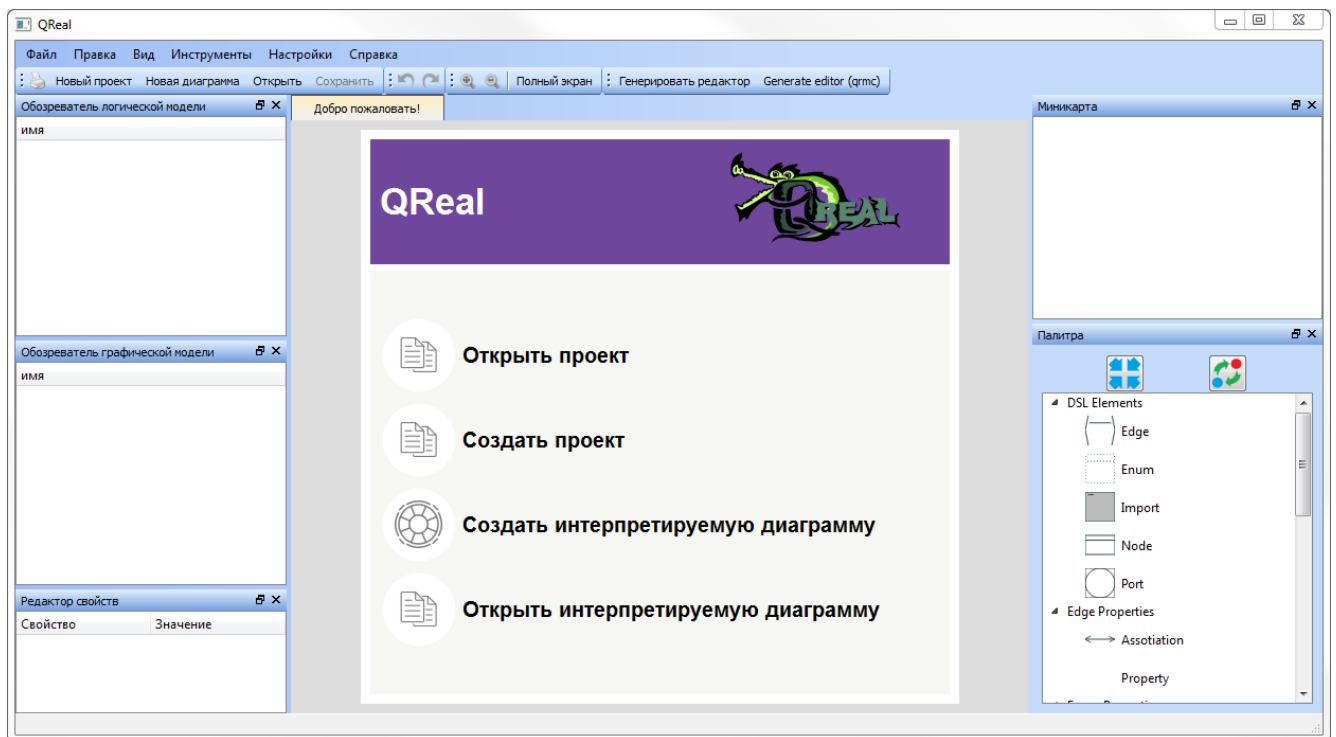


Рисунок 4.10: Стартовый экран QReal.

При выборе пункта меню „Открыть интерпретируемую диаграмму“ будет предложено выбрать файл сохранения с метамоделью языка, которая будет загружена в интерпретатор, после чего будет создана пустая модель с этой метамоделью. При желании можно после этого загрузить файл с сохранённой моделью, созданной с помощью языка, метамодель которого загружена. При выборе пункта меню „Создать интерпретируемую диаграмму“ будет предложено ввести имя создаваемого языка, после чего будет создана пустая метамодель (в среде при этом будет отображаться пустая палитра элементов) и пустая модель.

Элемент можно добавить в язык, кликнув правой кнопкой мыши на палитру и выбрав в контекстном меню пункт „Добавить элемент“. После этого надо выбрать метатип создаваемого элемента — сущность или связь, и ввести его имя. Если создаваемый элемент — сущность, его можно пометить как корневой элемент диаграммы, в этом случае он будет создаваться автоматически при создании диаграммы, и только он может быть в корне иерархии включения элементов (то есть это тот элемент, в который как в сыновья добавляются все остальные, это обычно и есть узел „диаграмма“ для языка). Вновь созданный элемент будет иметь форму по умолчанию (прямоугольник для сущностей и

обычную линию для связей) и не будет иметь свойств. Элемент уже можно начать использовать, например, если это корневой элемент нового языка, перетащить в „Обозреватель графической модели“, что приведёт к созданию новой диаграммы, куда можно добавлять этот же или другие элементы. Форму элемента можно поменять, кликнув по элементу на палитре или на диаграмме правой кнопкой и выбрав пункт „Изменить внешний вид“ в контекстном меню. Схематически данный процесс показан на рисунке 4.11.

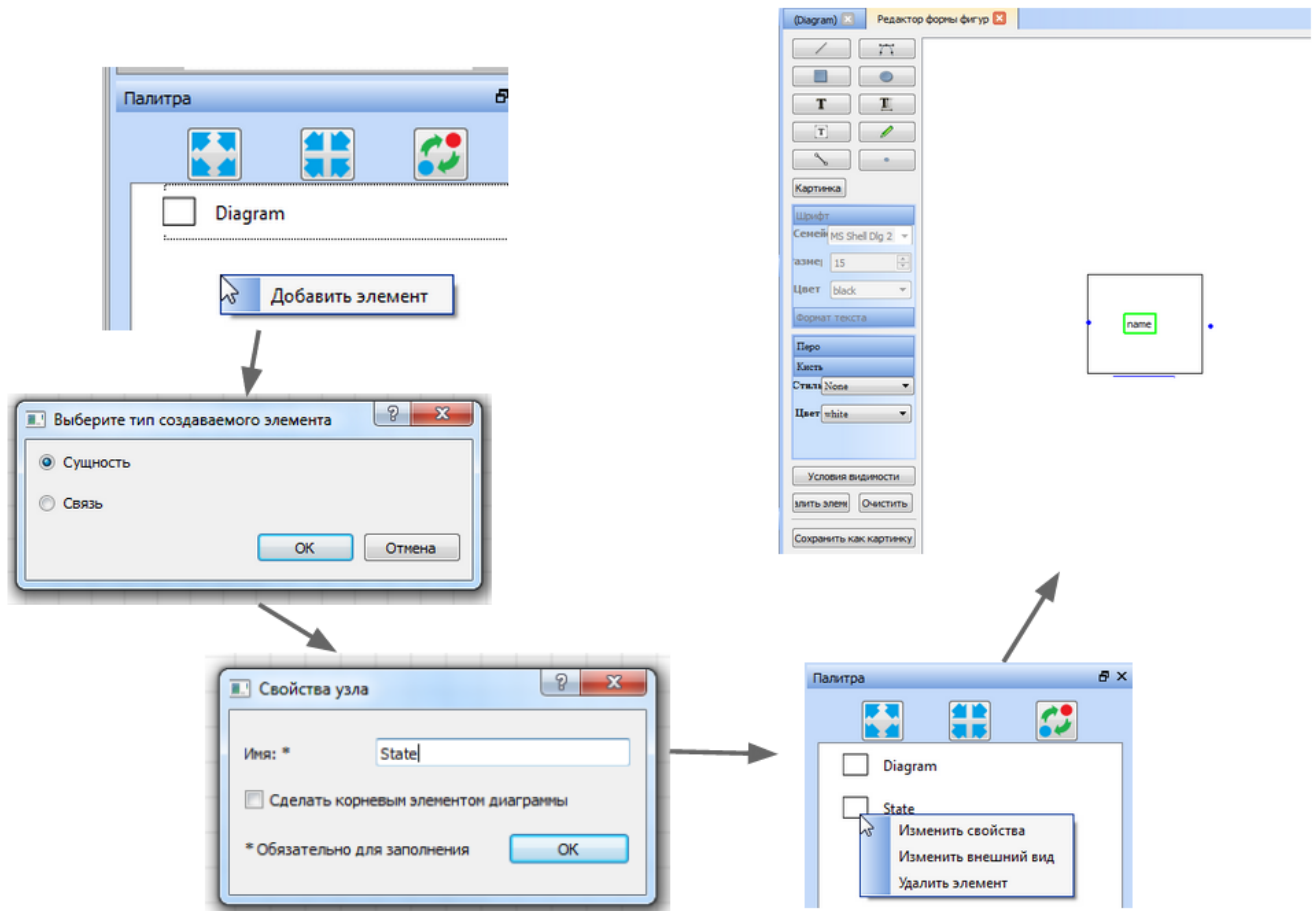


Рисунок 4.11: Создание нового элемента языка.

Свойства элемента также можно добавить, удалить или отредактировать через контекстное меню. При этом появится диалоговое окно со списком свойств. Свойства характеризуются своим именем, типом и значением по умолчанию. При создании нового свойства эти характеристики надо будет ввести, для существующих свойств их можно редактировать. При этом есть некоторые ограничения, связанные с поддержанием консистентности модели. Например, элементам, экземпляры которых уже существуют на диаграмме, нельзя добавить



новое свойство — уже существующие экземпляры добавленного свойства иметь не будут, что может привести к ошибкам в инструментальных средствах, которые рассчитаны на то, что все обозначенные в метамодели свойства у экземпляров элементов есть. В таком случае система запрещает добавление свойства и требует, чтобы до добавления свойства все экземпляры были удалены. При этом удаление свойства элемента безопасно — существующие экземпляры будут иметь удалённое свойство, но оно никогда не будет запрошено, поскольку отсутствует в метамодели. Эти проблемы напрямую связаны с проблемой совместной эволюции моделей и метамодели языка, поскольку они будут проявляться и при открытии файла сохранения с моделью, выполненной в старой версии языка. На данный момент эти проблемы решаются в рамках отдельного исследования. Процесс редактирования свойства элемента схематически представлен на рисунке 4.12

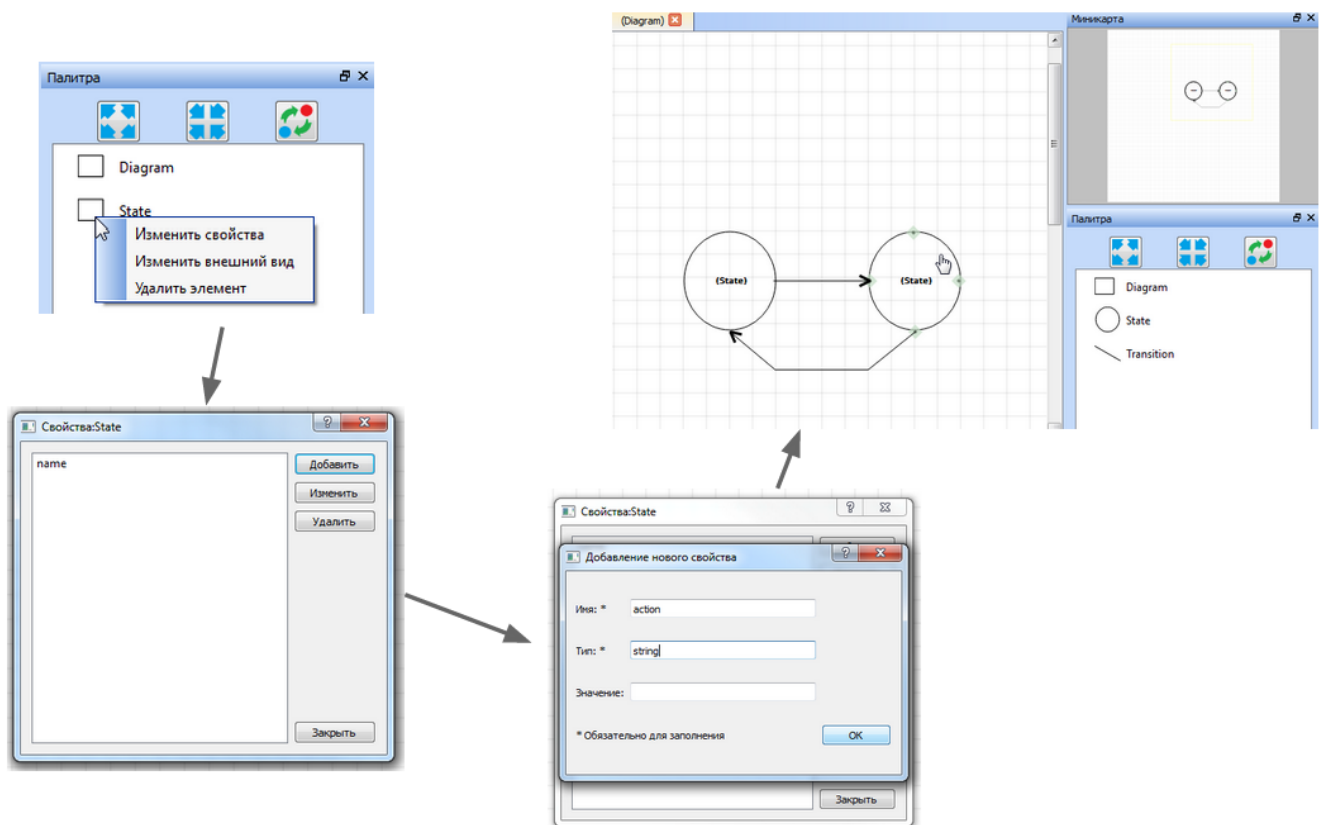


Рисунок 4.12: Редактирование свойств элемента.

## Заключение

Основные результаты, полученные в данной диссертации, таковы.

1. Проведён анализ различных подходов к реализации визуальных языков и различных технологических средств, их реализующих.
2. Разработана методика и набор инструментальных средств для создания предметно-ориентированных языков с помощью графического языка метамоделирования и сопутствующих визуальных языков.
3. Предложен новый способ метамоделирования: „метамоделирование на лету“.
4. Предложенные методики и технологии реализованы в виде промышленного продукта QReal.
5. Проведена апробация при создании редактора, генератора, средств проверки ограничений среды QReal:Robots и других предметно-ориентированных решений.

Реализация предложенных в данной диссертации методик осуществлялась коллективом студентов и аспирантов кафедры системного программирования СПбГУ в рамках проекта QReal. Хочется особо отметить вклад студентов, работавших над данным проектом под руководством автора диссертации: Абрамова Ивана Александровича, Дерипаска Анны Олеговны, Гудошниковой Анны Андреевны, Жуковой Беллы Юрьевны, Заболотных Елены Петровны, Занько Софьи Владимировны, Иванова Всеволода Юрьевича, Кузенковой Анастасии Сергеевны, Кузнецовой Марьи Юрьевны, Курбанова Рауфа Эльшад оглы, Назаренко Владимира Владимировича, Нефёдова Ефима Андреевича, Никольского Кирилла Андреевича, Осечкиной Марии Сергеевны, Птахиной Алины Ивановны, Пышновой Александры Витальевны, Савина Никиты Сергеевича,

Соковиковой Натальи Алексеевны, Такун Евгении Игоревны, Тихоновой Марии Валерьевны, всех студентов, работавших под руководством Брыксина Тимофея Александровича, а также вклад Дмитрия Мордвинова и Ирины Брюхановой. Без них данная диссертация вряд ли была бы возможна.

## Список сокращений и условных обозначений

**UML** Unified Modeling Language, URL: <http://uml.org/> (дата обращения: 22.02.2014г.)

**DSM** Domain Specific Modeling

**DSL** Domain Specific Language

**ER** Entity-Relationship

**DSM-платформа** Инструментарий для создания предметно-ориентированных средств разработки (DSM-решений)

**metaCASE-средство** Здесь используется как синоним термина „DSM-платформа“, получил широкое распространение после выхода в свет статьи [76], но в данной диссертации этот термин почти не используется, в силу расплывчатости понятия „CASE“.

**Предметно-ориентированное решение** Предметно-ориентированный язык и технологические средства для работы с ним (например, редактор, генераторы, верификаторы и т.д.)

**BPMN** Business Process Model and Notation, URL: <http://www.bpmn.org/> (дата обращения: 22.02.2014г.)

**Bluetooth** Спецификация беспроводных сетей ближней связи, URL: <https://www.bluetooth.org/en-us/specification/adopted-specifications> (дата обращения: 22.02.2014г.)

**USB** Universal Serial Bus, „универсальная последовательная шина“, интерфейс передачи данных для периферийных устройств URL: <http://www.usb.org/> (дата обращения: 22.02.2014г.)

**ПО** Программное обеспечение

**CASE** Computer-Aided Software Engineering, термин, которым исторически обозначают среды визуального моделирования и программирования.

**XML** Extensible Markup Language. Стандарт 1.0, URL:  
<http://www.w3.org/TR/REC-xml/> (дата обращения 21.08.2014г.).

## Литература

1. Heijstek Werner, Chaudron Michel RV. Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process // Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on / IEEE. 2009. С. 113–120.
2. Baker Paul, Loh Shiou, Weil Frank. Model-driven engineering in a large industrial context—Motorola case study // Model Driven Engineering Languages and Systems. Springer, 2005. С. 476–491.
3. Selic Bran. The pragmatics of model-driven development // Software, IEEE. 2003. Т. 20, № 5. С. 19–25.
4. Кознов Д. В. Основы визуального моделирования. Бином. Лаборатория знаний, 2008.
5. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases / Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu [и др.] // Empirical Software Engineering. 2013. Т. 18, № 1. С. 89–116.
6. Chen Minder. The Methodology-fit of UML: An Empirical Study of UML Adaptation // COMMUNICATIONS OF THE ICISA. 2012. С. 1.
7. Kelly Steven, Tolvanen Juha-Pekka. Domain-specific modeling: enabling full code generation. Wiley-IEEE Computer Society Press, 2008.
8. A software engineering experiment in software component generation / Richard B Kieburtz, Laura McKinney, Jeffrey M Bell [и др.] // Proceedings of the 18th international conference on Software engineering / IEEE Computer Society. 1996. С. 542–552.

9. Kelly Steven, Tolvanen Juha-Pekka. Visual domain-specific modeling: Benefits and experiences of using metaCASE tools // International Workshop on Model Engineering, at ECOOP. 2000.
10. Gray Jeff, Karsai Gábor. An examination of DSLs for concisely representing model traversals and transformations // System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on / IEEE. 2003. С. 10–pp.
11. Архитектура среды визуального моделирования QReal / А.Н. Терехов, Т.А. Брыксин, Ю.В. Литвинов [и др.] // Системное программирование. 2009. № 4. С. 171–196.
12. Средства быстрой разработки предметно-ориентированных решений в metaCASE-средстве QReal / А.С. Кузенкова, А.О. Дерипаска, К.С. Таран [и др.] // Научно-технические ведомости СПбГПУ, Информатика, телекоммуникации, управление. 2011. № 4 (128). С. 142–145.
13. Брыксин Т.А., Литвинов Ю.В. Среда визуального программирования роботов QReal:Robots // Материалы международной конференции "Информационные технологии в образовании и науке". Самарский филиал МГПУ, МГПУ, 2011. С. 332–334.
14. Иванов А.Н., Стригун С.С. Технологическое решение REAL-IT: автоматизированная разработка пользовательского интерфейса информационных систем // Системное программирование. 2005. Т. 1.
15. Парфенов В.В., Терехов А.Н. RTST-технология программирования встроенных систем реального времени // Системная информатика. 1997. № 5. С. 228–256.
16. Терехов А.Н. RTST-технология программирования встроенных систем реального времени // Записки семинара кафедры системного программирования "CASE-средства RTST++". 1998. № 1. С. 3–17.
17. Объектно-ориентированное расширение технологии RTST / А Иванов, Дм Кознов, А Лебедев [и др.] // Записки семинара кафедры системного программирования "CASE-средства RTST++". 1998. № 1. С. 17–36.

18. Иванов А, Дм. Кознов, Т. Мурашева. Поведенческая модель RTST++ // Записки семинара кафедры системного программирования “CASE-средства RTST++”. № 1. С. 37–49.
19. Кузенкова А.С., Брыксин Т.А., Литвинов Ю.В. Мета моделирование: современный подход к созданию средств визуального проектирования // Материалы второй научно-технической конференции молодых специалистов «Старт в будущее», посвященной 50-летию полета Ю.А. Гагарина в космос. ОАО "КБСМ 2011. С. 228–231.
20. Литвинов Ю.В. Визуальные средства программирования роботов и их использование в школах // Современные информационные технологии и ИТ-образование, сборник избранных трудов VII Международной научно - практической конференции. ИНТУИТ.РУ, 2012. С. 858–868.
21. Ubiq Mobile + QReal: a Technology for Development of Distributed Mobile Services / Timofey Bryksin, Yuri Litvinov, Valentin Onossovski [и др.] // 10th Conference of Open Innovations Association FRUCT and the 2nd Finnish-Russian Mobile Linux Summit: Proceedings. State University of Aerospace Instrumentation (SUAI), 2011. С. 27–35.
22. Averbukh Vladimir L. Visualization metaphors // Programming and Computer Software. 2001. Т. 27, № 5. С. 227–237.
23. Teichroew Daniel, Hershey III Ernest A. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems // Software Engineering, IEEE Transactions on. 1977. № 1. С. 41–48.
24. Паронджанов Владимир. Как улучшить работу ума // М.: "Дело. 2001.
25. Лядова Л.Н., Сухов А.О. Визуальные языки и языковые инструментари: методы и средства реализации // Труды Конгресса по интеллектуальным системам и информационным технологиям «AIS-IT'10». 2010. Т. 1. С. 374–382.



26. A classification framework to support the design of visual languages / Gennaro Costagliola, Andrea Delucia, Sergio Orefice [и др.] // Journal of Visual Languages & Computing. 2002. Т. 13, № 6. С. 573–600.
27. Dmitriev Sergey. Language oriented programming: The next programming paradigm // JetBrains onBoard. 2004. Т. 1, № 2.
28. Karlsch Martin. A model-driven framework for domain specific languages // Unpublished master's thesis, Hasso-Plattner-Institute of Software Systems Engineering, Potsdam, Germany. 2007.
29. Сухов А.О. Разработка инструментальных средств создания визуальных предметно-ориентированных языков. Ph.D. thesis: Институт системного программирования Российской академии наук. 2013.
30. Mauw Sjouke. The formalization of message sequence charts // Computer Networks and ISDN Systems. 1996. Т. 28, № 12. С. 1643–1657.
31. Murata Tadao. Petri nets: Properties, analysis and applications // Proceedings of the IEEE. 1989. Т. 77, № 4. С. 541–580.
32. Harel David. Statecharts: A visual formalism for complex systems // Science of computer programming. 1987. Т. 8, № 3. С. 231–274.
33. Mernik Marjan, Heering Jan, Sloane Anthony M. When and how to develop domain-specific languages // ACM computing surveys (CSUR). 2005. Т. 37, № 4. С. 316–344.
34. Van Deursen Arie, Klint Paul, Visser Joost. Domain-specific languages: an annotated bibliography // ACM Sigplan Notices. 2000. Т. 35, № 6. С. 26–36.
35. Voelter Markus. Best practices for DSLs and model-driven development // Journal of Object Technology. 2009. Т. 8, № 6. С. 79–102.
36. Luoma Janne, Kelly Steven, Tolvanen Juha-Pekka. Defining domain-specific modeling languages: Collected experiences // 4 th Workshop on Domain-Specific Modeling. 2004.

37. Atkinson Colin, Kuhne Thomas. Model-driven development: a metamodeling foundation // Software, IEEE. 2003. Т. 20, № 5. С. 36–41.
38. Repenning Alexander, Sumner Tamara. Agentsheets: A medium for creating domain-oriented visual languages // Computer. 1995. Т. 28, № 3. С. 17–25.
39. Koznov Dmitriy. Process Model of DSM Solution Development and Evolution for Small and Medium-Sized Software Companies // Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International / IEEE. 2011. С. 85–92.
40. Кознов Дмитрий Владимирович. Разработка и сопровождение DSM-решений на основе MSF // Системное программирование. 2008. Т. 3, № 1. С. 80–96.
41. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон [и др.]. 2001.
42. OMG. Unified Modeling Language (OMG UML), Infrastructure, V2.4.1. 2011. URL: <http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF>.
43. OMG. Unified Modeling Language (OMG UML), Superstructure, V2.4.1. 2011. URL: <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>.
44. Patterns: Model-Driven Development Using IBM Rational Software Architect / Peter Swithinbank, Mandy Chessell, Tracy Gardner [и др.]. IBM, International Technical Support Organization, 2005.
45. De Lara Juan, Guerra Esther. Deep meta-modelling with metadepth // Objects, Models, Components, Patterns. Springer, 2010. С. 1–20.
46. de Lara Juan, Guerra Esther. Domain-specific textual meta-modelling languages for model driven engineering // Modelling Foundations and Applications. Springer, 2012. С. 259–274.
47. Tolvanen Juha-Pekka, Pohjonen Risto, Kelly Steven. Advanced tooling for domain-specific modeling: MetaEdit+ // Sprinkle, J., Gray, J., Rossi, M.,

- Tolvanen, JP (eds.) The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland. 2007.
48. Tolvanen Juha-Pekka, Kelly Steven. MetaEdit+: defining and using integrated domain-specific modeling languages // Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications / ACM. 2009. С. 819–820.
  49. Metamodel-based tool integration with MOFLON / Carsten Amelunxen, Felix Klar, Alexander Königs [и др.] // Proceedings of the 30th international conference on Software engineering / ACM. 2008. С. 807–810.
  50. Gronback Richard C. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional, 2009.
  51. Сорокин А.В., Кознов Д.В. Обзор Eclipse Modeling Project // Ред. коллегия: А.Н. Терехов, Д.Ю. Булычев, Д.В. Кознов, А.Н. Иванов Отв. редактор: Д.В. Кознов. 2010. С. 6.
  52. The generic modeling environment / Akos Ledecz, Miklos Maroti, Arpad Bakay [и др.] // Workshop on Intelligent Signal Processing, Budapest, Hungary. Т. 17. 2001.
  53. Vangheluwe Hans, de Lara Juan. {Domain-Specific Visual Modelling in AToM<sup>3</sup>} // Proceedings of the 4th oopsla workshop on domain-specific modeling. Vancouver, Canada: University of Jyväskylä. 2004.
  54. Domain-specific development with visual studio dsl tools / Steve Cook, Gareth Jones, Stuart Kent [и др.]. Pearson Education, 2007.
  55. Pounamu: A meta-tool for exploratory domain-specific visual language tool development / Nianping Zhu, John Grundy, John Hosking [и др.] // Journal of Systems and Software. 2007. Т. 80, № 8. С. 1390–1407.
  56. Engstrom Eric, Krueger Jonathan. Building and rapidly evolving domain-specific tools with DOME // Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on / IEEE. 2000. С. 83–88.

57. Inc. Honeywell. DOME Guide, version 5.2. 1999. URL: <http://www.cse.msu.edu/SENS/Software/DOME/DoMEGuide.pdf>.
58. An Approach to Development of Visual Modeling Toolkits / Alexander O Sukhov, Lyudmila N Lyadova, ВВ Ланин [и др.] // International Journal. 2012. Т. 6, № 2. С. 184–199.
59. Лядова Л.Н., Сухов А.О. Языковой инструментарий системы MetaLanguage // Математика программных систем. Межвузовский сборник научных статей. 2008. С. 40–51.
60. QReal DSM platform-An Environment for Creation of Specific Visual IDEs. / Anastasiia Kuzenkova, Anna Deripaska, Timofey Bryksin [и др.] // ENASE. 2013. С. 205–211.
61. Симонова А.А. Подход к разработке CASE-пакетов. 2007. URL: [http://se.math.spbu.ru/SE/diploma/2007/Simonova\\_dip.doc](http://se.math.spbu.ru/SE/diploma/2007/Simonova_dip.doc).
62. Птахина А.И. Интерпретация метамodelей в metaCASE-системе QReal. 2012. URL: [http://se.math.spbu.ru/SE/YearlyProjects/2012/345/345\\_Ptakhina\\_report.pdf](http://se.math.spbu.ru/SE/YearlyProjects/2012/345/345_Ptakhina_report.pdf).
63. Дерипаска А.О. Визуальный язык задания ограничений на модели в QReal. 2012. URL: [http://se.math.spbu.ru/SE/YearlyProjects/2012/345/345\\_Deripaska\\_report.pdf](http://se.math.spbu.ru/SE/YearlyProjects/2012/345/345_Deripaska_report.pdf).
64. Дерипаска Анна Олеговна. Языки задания ограничений // Компьютерные инструменты в образовании. 2013. Т. 1. С. 13–26.
65. Фаулер М, Бек К, Брант Д. Рефакторинг. Улучшение существующего кода. Символ-Плюс СПб., 2003.
66. Mens Tom, Taentzer Gabriele, Müller Dirk. Challenges in model refactoring // Proc. 1st Workshop on Refactoring Tools, University of Berlin. Т. 98. 2007.
67. Кузенкова А.С. Поддержка механизма рефакторингов в metaCASE-системе QReal. 2012. URL: [http://se.math.spbu.ru/SE/YearlyProjects/2012/345/345\\_Kuzenkova\\_report.pdf](http://se.math.spbu.ru/SE/YearlyProjects/2012/345/345_Kuzenkova_report.pdf).

68. Кузенкова А.С., Литвинов Ю.В. Поддержка механизма рефакторингов в DSM-платформе QReal // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада "Технологии Microsoft в теории и практике программирования". Изд-во СПбГПУ, 2013. С. 71–72.
69. Кузенкова А.С. Поддержка механизма рефакторингов в metaCASE-системе QReal // Список-2012: Материалы всероссийской научной конференции по проблемам информатики. 25-27 апр. 2012г., Санкт-Петербург. Изд-во ВВМ, 2012. С. 24–33.
70. Поляков В.А., Брыксин Т.А. Средство разработки визуальных интерпретаторов и отладчиков диаграмм в проекте QReal // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада "Технологии Microsoft в теории и практике программирования". Изд-во СПбГПУ, 2013. С. 80–81.
71. Поляков Владимир Александрович, Брыксин Тимофей Александрович. Подходы к заданию семантики интерпретации диаграмм, основанные на технологии преобразования графов // Компьютерные инструменты в образовании. 2013. Т. 2. С. 3–17.
72. Поляков В.А., Брыксин Т.А. Подходы к заданию семантики интерпретации диаграмм в рамках DSM-подхода // Системное программирование. 2012. № 7. С. 156–183.
73. Поляков В.А. Разработка визуального интерпретатора моделей в системе QReal // Список-2012: Материалы всероссийской научной конференции по проблемам информатики. 25-27 апр. 2012г., Санкт-Петербург. Изд-во ВВМ, 2012. С. 56–61.
74. Птахина А.И. Разработка метамоделирования “на лету” в metaCASE-системе QReal. 2013. URL: <http://se.math.spbu.ru/SE/YearlyProjects/2013/445/445-Ptakhina-report.pdf>.

75. Птахина А.И. Разработка метамоделирования “на лету” в системе QReal // Список-2013: Материалы всероссийской научной конференции по проблемам информатики. Изд-во ВВМ, 2013. С. 28–36.
76. Alderson Albert. Meta-CASE technology // Software Development Environments and CASE Technology. Springer, 1991. С. 81–91.
77. Тихонова М.В. Среда программирования QReal:Robots // Список-2012: Материалы всероссийской научной конференции по проблемам информатики. 25-27 апр. 2012г, Санкт-Петербург. Изд-во ВВМ, 2012. С. 70–75.
78. Терехов А.Н., Брыксин Т.А., Литвинов Ю.В. Среда визуального программирования роботов QReal:Robots // III Всероссийская конференция "Современное технологическое обучение: от компьютера к роботу" (сборник тезисов). 2013. С. 2–5.
79. Portsmore Merredith. ROBOLAB: Intuitive Robotic Programming Software to Support Life Long Learning // APPLE Learning Technology Review. 1999.
80. Syme Don, Granicz Adam, Cisternino Antonio. Expert F# 3.0. Springer, 2012.
81. Onossovski Valentin, Terekhov Andrey N. Ubiq Mobile — a new universal platform for mobile online services // Proceedings of 6th seminar of FRUCT Program. 2009.
82. Дерипаска А.О. Визуальный язык для платформы Ubiq Mobile в среде QReal. 2013. URL: <http://se.math.spbu.ru/SE/YearlyProjects/2013/445/445-Deripaska-report.pdf>.
83. Такун Е.И. Реализация режима быстрого прототипирования в CASE-системе QReal. 2011. URL: <http://se.math.spbu.ru/SE/diploma/2011/Takun>

## Приложение А

### Применение предложенных в работе подходов

В этом приложении приводится описание применения предлагаемой в диссертации технологии для разработки различных предметно-ориентированных решений. В силу ограниченности объёма работы были выбраны три наиболее зрелых решения: QReal:Robots, QReal:Ubiq, QReal:Hascol. Все эти решения имеют достаточно сильно различающиеся предметные области и разнятся по применённым в них подходам к созданию предметно-ориентированных языков. Соответственно, результирующие предметно-ориентированные языки достаточно сильно отличаются друг от друга, что является аргументом в пользу широкой применимости предлагаемой технологии.

#### А.1 Среда программирования роботов

##### QReal:Robots

Наиболее зрелый на данный момент пример применения технологии QReal — среда для обучения программированию в школах QReal:Robots. Этот проект создавался в практически идеальных для применения предметно-ориентированного подхода условиях: имелась достаточно узкая предметная область, в которой уже довольно активно применялось визуальное программирование, имелась потребность в средствах программирования в рамках этой предметной области для написания нетривиальных программ, имелась DSM-платформа QReal, на которой было возможно в короткие сроки реализовать такое решение. В этой работе результаты проекта QReal:Robots изложены достаточно кратко, более подробно см. статьи [13,20,77,78] Далее описание результатов приводится в такой последовательности — приводится введение в пред-

метную область и мотивация к созданию DSM-решения, анализ и формулировка требований, изложение возможностей визуального языка и инструментальных средств для него с точки зрения пользователя и „ручной“ реализации, далее рассказывается, как при реализации DSM-решения применялись возможности платформы QReal, в чём она смогла помочь, а в чём нет, и почему.

### А.1.1 Постановка задачи

Сейчас в школах для начального преподавания информатики активно внедряются робототехнические конструкторы, наиболее популярный из которых на данный момент — конструктор LEGO Mindstorms NXT 2.0<sup>1</sup>. Идея использовать роботы для преподавания не случайна, понятие „исполнитель“ традиционно используется в школьном преподавании в России со времён академика Ершова, а робот — наиболее наглядный вид исполнителя. Исполнитель — это некая сущность, способная выполнять команды, указанные в программе, в некоторой среде. До сих пор самым популярным исполнителем в школах остаётся „черепашка“ LOGO<sup>2</sup>, предложенная американским педагогом Сеймуром Пейпертом ещё в 1967 году. „Черепашка“ может перемещаться по экрану, оставляя за собой след, и подчиняется командам вида „поднять перо“, „опустить перо“, „20 шагов вперёд“, „на 90 градусов налево“. В текстовых программах может быть не очевидно, где исполнение алгоритма пошло не так, а иногда непонятно даже, правильно работает программа или нет. В случае с „черепашкой“ отклонение выполнения программы от задумки автора будет видимо сразу же, „черепашка“ нарисует что-то некорректное. При этом сразу видно, где допущена ошибка, можно легко найти строку программы, на которой „черепашка“ отклонилась от задуманной траектории.

Исполнитель, движущийся по экрану, всё же оказывается недостаточно наглядным. Сеймур Пейперт в своих ранних экспериментах использовал механическую черепашку. Сейчас развитие электроники сделало возможным создавать относительно недорогие устройства, исполняющие программу с помощью

<sup>1</sup>Домашняя страница конструктора Mindstorms, URL: <http://mindstorms.lego.com/> (дата обращения 07.09.2014)

<sup>2</sup>MyRobot, Язык программирования Лого, URL: <http://myrobot.ru/logo/aboutlogo.php> (дата обращения 07.09.2014)



дистанционного управления с компьютера или позволяющие загружать программу для автономного исполнения. Для преподавания в школах из таких устройств наиболее интересны робототехнические конструкторы, потому что они требуют сборки робота перед его программированием, что интереснее для детей и развивает их конструкторские навыки. Самый известный такой конструктор — Mindstorms NXT. Он позволяет из трёх моторов, трёх видов датчиков (датчика касания, ультразвукового датчика расстояния, датчика освещённости), блока управления и пластмассовых деталей собирать довольно сложные конструкции, от движущихся тележек до стационарных устройств, например, для росписи ёлочных украшений или игры на барабанах.

Программировать такие роботы сложнее, чем „черепашку“, поскольку из конструктора можно собрать самые разные модели роботов, и программировать приходится в терминах, например, вида „мотор А включить на 50% мощности“, „ждать срабатывания датчика касания“, а не „вперёд на 20 шагов“, как в „черепашке“. Это интереснее и полезнее для детей, но сложнее. Дополнительную сложность добавляет и то, что робот может взаимодействовать с окружением с помощью датчиков, в отличие от „черепашки“. Это позволяет преподавать не только основы информатики, но и основы кибернетики, демонстрируя принципы построения программ, работающих во внешней среде, которую они не могут контролировать. Типичный пример задачи, которую решают начинающие программисты с помощью робота — движение по линии, когда робот с одним или двумя датчиками освещённости должен проехать по чёрной линии, нарисованной на полу. На „черепашке“ обучать решению подобных задач невозможно.

Сложность программирования робототехнических конструкторов преодолевается использованием визуальных языков программирования. Они гораздо понятнее и нагляднее для школьников, чем текстовые языки. Программы на таких языках состоят из элементарных блоков, например, „включить мотор“, „ждать срабатывания датчика“, которые достаточно перетащить с палитры на диаграмму и соединить связями. При наличии удобного редактора для такого языка пользоваться им могут даже дошкольники, не умеющие ещё читать. Как правило, с помощью визуальных языков школьники программируют примерно до седьмого класса, после чего постепенно переходят на текстовые языки. В комплекте с конструктором поставляется среда визуального программирования.

ния NXT-G, существуют отдельно распространяемые среды, самой популярной из которых в школах является Robolab [79]. Визуальное программирование в сообществе людей, занимающихся программированием таких роботов, весьма популярно.

Существующие среды программирования роботов имеют ряд недостатков, затрудняющих их использование в школах, например, отсутствие русификации, невозможность отладки программы. Это делает актуальной задачу создания новой такой среды, учитывающей все достоинства и недостатки существующих, а также уже накопленный опыт преподавания. Поэтому возникла ситуация, чрезвычайно интересная для апробации DSM—платформы. В выбранной предметной области уже сложились традиции использования визуальных языков, поэтому можно рассчитывать на содержательное сравнение с существующими средами. Кроме того, задачи, решаемые в данной предметной области, достаточно нетривиальны, чтобы полученный опыт разработки визуального языка можно было перенести на более сложные случаи, возникающие при промышленном программировании. Все типичные для императивного программирования конструкции, такие как ветвления и циклы, используются при программировании роботов, поэтому полученные результаты окажутся применимыми и для других случаев, где требуется императивный подход. Вместе с тем, предметная область достаточно узка, чтобы можно было получить заметные выгоды от использования предметно-ориентированного языка: на языке общего назначения программирование велось бы путём вызова функций API<sup>3</sup> робота, для чего требовалось бы писать довольно много вспомогательного кода, например, объявления функций. Кроме того, при использовании визуального языка невозможно совершить некоторые ошибки, типичные для текстовых языков, например, невозможно ошибиться в написании имени вызываемой функции, если достаточно просто перетащить соответствующий ей блок из палитры.

---

<sup>3</sup>Application Program Interface, интерфейс прикладных программ

## А.1.2 Существующие среды визуального программирования роботов

Как уже отмечалось, визуальное программирование весьма популярно среди людей, работающих с конструкторами Mindstorms. Анализ существующих сред является естественным источником для формулировки требований к проектируемому DSM-решению, наряду с интервьюированием экспертов в предметной области и потенциальных пользователей. Результаты анализа существующих сред представлены ниже.

Среда NXT-G поставляется вместе с конструктором Mindstorms NXT. Среда базируется на системе визуального программирования LabView, используемой для моделирования различных экспериментов. Среда LabView в качестве языка программирования использует визуальный язык G, язык с процессом вычислений, ориентированным на данные — в нём связи между блоками обозначают не последовательность выполнения операторов, а зависимости между блоками по данным. Основная проблема среды NXT-G заключается в отсутствии полноценной поддержки математических выражений. В языке присутствуют блоки для арифметических действий, констант, переменных, и чтобы построить математическое выражение, их надо соединять связями. Таким образом, на диаграмме приходится рисовать фактически дерево разбора арифметического выражения, что делает программирование даже несложных задач, требующих математики, чрезвычайно утомительным. В школьной программе такие задачи возникают часто, например, движение по линии или езда вдоль стены с помощью датчика расстояния требует вычисления производной, поэтому NXT-G для преподавания в школах практически не используется. Ещё одной особенностью этой среды, оказавшейся недостатком при преподавании в школах, стало то, что не все свойства блоков видны на диаграмме, требуется кликнуть на блок и открыть его свойства в редакторе свойств. Это делает невозможным показ всей программы на проекторе, что затрудняет воспроизведение программы учениками. У NXT-G нет официальной русификации, отсутствуют встроенные средства отладки и генерации текстовой формы языка. Возможно добавление сторонних блоков средствами LabView, сам NXT-G позволяет выделять фрагменты диаграмм в подпрограммы и использовать их с помощью специального блока.

Среда Robolab также создавалась на основе среды LabView, и, в отличие от NXT-G, создавалась специально для преподавания. Примером специфичного для преподавания решения, принятого в среде Robolab, может послужить разделение возможностей среды на уровни. При запуске среды предлагается выбрать уровень, на котором будет вестись работа, на первых уровнях пользователь может только заполнять пустые места в уже готовом шаблоне программы, используя очень ограниченный набор блоков. На более высоких уровнях предоставляется возможность самостоятельно задавать связи между блоками, доступно больше различных видов блоков (например, на начальном уровне есть блок „ждать“, на более высоком — набор блоков „ждать 1 сек“, „ждать 2 сек“, „ждать 5 сек“ и т.д., на ещё более высоком — „ждать N сек“, где N является параметром блока и может являться результатом вычисления. Такое разделение позволяет начать работу со средой детям дошкольного возраста, которые не умеют ещё даже читать (картинки, используемые в блоках, достаточно понятны), при этом дети могут исследовать среду и разбираться в функциональности более высоких уровней постепенно, практически без помощи преподавателей.

Математические выражения Robolab поддерживает гораздо лучше, чем NXT-G, позволяя писать произвольные выражения в виде текста, используя тригонометрические функции и переменные, обращаться к значениям, возвращаемым датчиками, прямо из выражений. Циклы в Robolab реализованы довольно необычно — есть блоки „начало цикла“ и „конец цикла“, связь между концом и началом никак не визуализируется. Есть довольно развитая поддержка параллельных задач, есть все необходимые конструкции императивного программирования, включая подпрограммы. Русифицирована среда лишь частично, не имеет встроенных средств отладки, не может порождать код программы в текстовом виде, имеет довольно несовременный пользовательский интерфейс, при этом не бесплатна, и стоит довольно дорого для российских школ. При этом, несмотря на все перечисленные недостатки, эта среда вполне подходит для иллюстрации материала информатики и кибернетики примерно до 7-го класса школы, и сейчас является самой распространённой в школах России.

Среда Microsoft Robotics Developer Studio (MRDS), в отличие от рассмотренных ранее сред, не создавалась исключительно для конструктора Mindstorms NXT. Среда предназначена для создания сложных многопоточных приложений

с реактивной моделью поведения, которые весьма типичны для задач робототехники — современная робототехническая система представляет собой целый комплекс вычислительных средств, часть из которых может быть установлена на роботе, а часть — на компьютере, связанном с роботом скоростной сетью. Кроме того, необходимость в высокопроизводительных распределённых программных системах возникает не только в робототехнике, поэтому среда MRDS применяется и в других областях, например, с её помощью разрабатывались серверные части крупных веб-приложений. Программа в MRDS представляется в виде набора веб-сервисов, исполняемых независимо и обменивающихся данными друг с другом. Процесс программирования сводится к связыванию входов и выходов предопределённых сервисов, что осуществляется на визуальном языке VPL (Visual Programming Language). Этот язык, таким образом, попадает в категорию языков с процессом управления, ориентированным на данные, чем очень похож на визуальные языки систем, рассмотренных выше. Среда имеет развитую трехмерную среду симуляции, позволяя исполнять созданные программы не только на реальном роботе, но и на трёхмерной модели в окружении с подробной симуляцией физики (например, в стандартной поставке имеется подробная трёхмерная модель квартиры, по которой может перемещаться робот). Среда поддерживает генерацию кода на языке C#, отладку, распространяется бесплатно.

Несмотря на всё это, среда MRDS очень редко используется в школах для преподавания информатики. Связано это прежде всего с используемой там моделью вычислений. Представлять программу в виде взаимодействующих независимо исполняющихся веб-сервисов удобно профессиональным программистам, но школьникам, впервые пробуящим программировать, потребуется знать и уметь слишком многое, чтобы писать содержательные программы. К тому же, такой подход к программированию слишком специфичен и не может быть использован в качестве иллюстративного материала для „обычного“ императивного программирования, знание которого будет необходимо школьникам в дальнейшем. Кроме того, генерируемый средой код не может быть исполнен непосредственно на роботе Mindstorms NXT — он имеет слишком мало ресурсов для этого, робот может управляться только дистанционно по интерфейсу Bluetooth. Среда MRDS ориентирована на более дорогие устройства, в частно-

сти, „стандартная модель“ робота, рекомендуемая для использования в MRDS включает в себя ноутбук в качестве блока управления, который один дороже всего конструктора Mindstorms NXT. Использование модели робота позволяет решить эту проблему только частично — модель, даже с детальной реализацией физических эффектов, не позволяет воспроизвести всё многообразие реального мира, из-за которого, собственно, и возникают проблемы, которые решаются в кибернетике.

### **А.1.3 Требования к DSM-решению**

По результатам анализа существующих сред и общения с экспертами предметной области (преподавателями и методистами, использующими робототехнические конструкторы в своей работе) были сформированы следующие требования к проектируемому DSM-решению.

1. Среда должна позволять создавать довольно сложные программы, поскольку предполагается, что она будет служить для иллюстрации нетривиальных вопросов информатики и кибернетики, при этом её язык должен оставаться близким к „традиционным“ императивным языкам программирования. Среда должна удобно поддерживать нетривиальные математические выражения, переменные, ветвления, циклы, параллельные задачи.
2. Среда должна быть проста и удобна в работе, поскольку неудобный пользовательский интерфейс создавал бы дополнительную когнитивную нагрузку при изучении и без того сложного материала. Среда должна быть интуитивно понятна, работа со средой должна быть возможна при минимальном участии преподавателя.
3. Требуется наличие встроенных средств отладки, поскольку школьникам важно не столько сделать работающую программу, сколько разобраться в том, как она работает. Ошибки могут иметь большую педагогическую ценность, и среда должна иметь возможность демонстрировать ошибки (и ход выполнения программы вообще) школьникам. Весьма желательна возможность исполнения программы на модели робота на компьютере.

4. Требуется наличие возможности порождения текстового представления программы. Школьники старших классов, серьёзно занимающиеся программированием, должны видеть связь между диаграммами и кодом на текстовых языках, иметь возможность редактировать текстовую программу в той же среде, в которой они привыкли работать.
5. Среда должна быть русскоязычной, поскольку основная группа её пользователей ещё не владеет в должной степени иностранными языками.

Как видно из приведённого выше обзора, ни одна из существующих сред программирования роботов указанным требованиям не удовлетворяет. Наиболее активно в школах используется среда Robolab, но, поскольку он недёшев, среди школьных учителей есть желание заменить её на более современный (и по возможности бесплатный) аналог. Поэтому существует реальная потребность в создании новой такой среды, что и было сделано на базе метатехнологии QReal.

#### **A.1.4 Визуальный язык QReal:Robots**

В качестве модели вычислений для визуального языка в силу наличия требования близости к „традиционным“ языкам программирования была выбрана модель вычислений, ориентированная на поток исполнения. Блок языка представляет собой элементарную команду роботу, связи между блоками указывают последовательность, в которой выполняются блоки, это делает программы больше похожими на блок-схемы, чем программы на существующих средах программирования роботов. Пример программы приведён на рисунке A.1. Двухмоторная тележка с датчиком касания под управлением этой программы будет двигаться до срабатывания датчика касания, после чего подаст звуковой сигнал и будет некоторое время двигаться в обратном направлении, после чего повторит действия сначала.

Все блоки языка разбиты на смысловые группы, которые могут быть отдельно свёрнуты или развёрнуты в палитре. Описание для каждого блока приведено ниже.

1. Группа „Алгоритмы“ предназначена для блоков, определяющих последовательность выполнения команд программы.

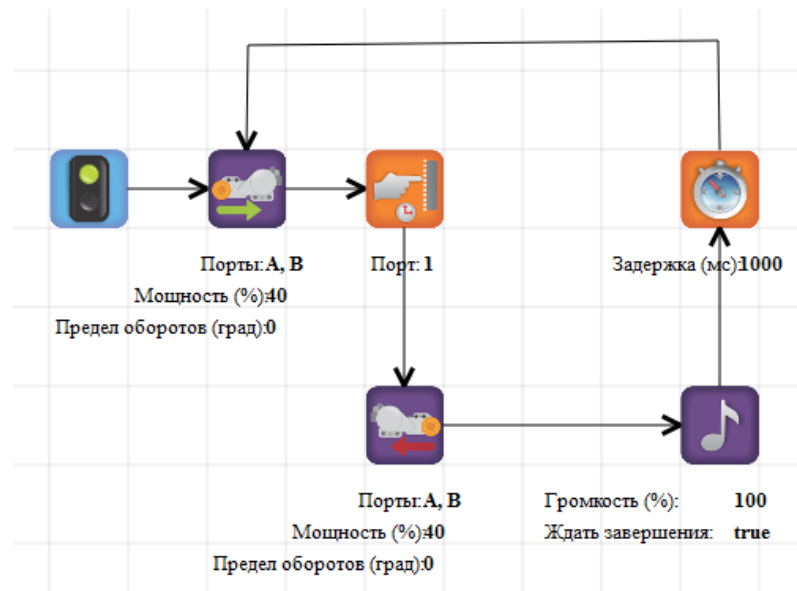


Рисунок А.1: Пример программы QReal:Robots.

- (а) „Диаграмма поведения робота“ — блок, на который добавляются все остальные блоки программы. Обычно непосредственно в рабочей области не рисуется, но при создании новой диаграммы перетаскивается из палитры в обозреватель модели.
- (б) „Линия соединения“ — единственная связь, присутствующая в языке, задаёт последовательность исполнения блоков. Имеет метку, которая позволяет определять, когда управление надо передать по линии соединения в случае условного оператора или цикла — например, метка „больше 0“ говорит, что управление по данной связи будет передано только тогда, когда выражение в условном операторе, подключённом к этой связи, будет иметь значение, большее 0. По умолчанию связи не помечены ничем.
- (с) „Условие“ — условный оператор. Параметризуется арифметическим выражением и должен иметь две исходящие связи, одна из которых выбирается для передачи управления в зависимости от значения выражения и метки связи.
- (d) „Цикл“ — оператор цикла, параметризуется арифметическим выражением, значение которого будет количеством итераций, которые надо выполнить, и должен иметь две исходящие связи. Пока блок



посещён меньшее количество раз, чем желаемое количество итераций, управление передаётся по связи, помеченной как „итерация“, как только желаемое количество итераций достигнуто, счётчик сбрасывается, и управление передаётся по непомеченной связи.

2. Группа „Действия“ содержит блоки, реализующие элементарные команды роботу.

- (a) „Гудок“ — проигрывает звук фиксированной частоты и фиксированной длительности с заданной громкостью.
- (b) „Играть звук“ аналогичен блоку „Гудок“, но позволяет настраивать частоту и длительность звука.
- (c) „Моторы вперёд“ — включает моторы по заданным портам с заданной мощностью (в процентах от максимальной).
- (d) „Моторы назад“ — аналогичен блоку „Моторы вперёд“, но включает моторы в противоположном направлении.
- (e) „Моторы стоп“ — отключить моторы на заданных портах.
- (f) „Параллельные задачи“ — разветвляет исполнение программы на два или более параллельно исполняемых потока.
- (g) „Функция“ — позволяет вычислить произвольное выражение, записанное в текстовой форме как параметр блока. Выражения в QReal:Robots могут использоваться везде, где могут использоваться числовые значения, блок „Функция“ введён для удобства как выделенное место для вычислений.

3. Группа „Инициализация“ содержит блоки, обозначающие начало и конец программы, и позволяющие задать начальное состояние различных подсистем робота.

- (a) „Блок инициализации“ обозначает место, с которого начинается исполнение программы, и позволяет задать, какие датчики подключены к портам управляющего блока робота.
- (b) „Конец“ — завершает работу программы и отключает моторы и датчики робота.

- (с) „Начало“ — аналогичен блоку инициализации, но не позволяет задать конфигурацию датчиков. В случае его использования конфигурация датчиков берётся из настроек, либо вычисляется по использованию блоков работы с датчиками в программе. Используется в программах, не использующих датчики, либо использующих один—два датчика, не требующих сложного конфигурирования.
  - (d) „Сбросить показания энкодера“ — обнуляет показания датчиков оборотов моторов по выбранным портам.
4. Группа „Ожидания“ содержит блоки, останавливающие выполнение программы (или одного из параллельных потоков) до наступления некоторого события.
- (a) „Ждать интенсивность цвета“ продолжает выполнение, когда датчик цвета по данному порту вернёт требуемое значение.
  - (b) „Ждать свет“ аналогичен блоку „Ждать интенсивность цвета“, но работает с датчиком света. Поясим, что набор Mindstorms NXT распространяется с датчиком цвета, который способен различать шесть цветов (красный, зелёный, синий, жёлтый, чёрный, белый), и измерять интенсивность света в одном из трёх режимов подсветки (красным, зелёным или синим цветом), или без подсветки (в пассивном режиме). Кроме этого, набор может использовать датчик света, распространявшийся в более ранних версиях конструктора. Этот датчик может только измерять интенсивность света с красной подсветкой или без подсветки, распознавать цвета не может. Кроме того, есть ещё датчик цвета стороннего производителя <sup>4</sup>, аналогичный по характеристикам датчику цвета из Mindstorms NXT, но способный распознавать больше цветов, QReal:Robots его на данный момент не поддерживает.
  - (с) „Ждать сенсор касания“ продолжает выполнение, когда срабатывает датчик касания.

---

<sup>4</sup>компании HiTechnic, документация на датчик находится по URL <http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=NCO1038> (дата обращения: 17.02.2014)

- (d) „Ждать сонар“ продолжает выполнение, когда ультразвуковой датчик расстояния возвращает значение в требуемых границах.
  - (e) „Ждать цвет“ продолжает выполнение, когда сенсор цвета возвращает заданный цвет. Заметим, что в режиме распознавания цвета датчик цвета не может измерять интенсивность, поэтому данный блок не может быть использован вместе с блоком „Ждать интенсивность цвета“ для датчика на том же порту.
  - (f) „Ждать энкодер“ продолжает выполнение, когда датчик оборотов мотора на заданном порту вернёт заданное значение.
  - (g) „Таймер“ продолжает выполнение программы по истечении заданного временного интервала (в миллисекундах).
5. Группа „Сегвей“ содержит блоки, предназначенные специально для балансировки робота-сегвея с использованием средств операционной системы `nxtOSEK` и библиотеки `NXTway-GS`<sup>5</sup>. Они были добавлены в палитру как демонстрация возможностей по использованию специфических функций операционной системы и сторонних библиотек, написанных на текстовых языках, а также как зрелищная демонстрация возможностей среды (балансирующий на двух колёсах робот привлекал внимание даже серьёзно увлекающихся робототехникой школьников, многие из которых затем интересовались средой, в которой он был запрограммирован).
- (a) „Балансировка“ — вызов функции `balance_control` библиотеки `NXTway-GS`. Блок принимает текущие показания датчиков оборотов моторов и другие параметры, связанные с текущим положением робота, получает данные с гироскопа, и записывает в переданные переменные мощности моторов, которые надо выставить, чтобы робот сохранял вертикальное положение.
  - (b) „Инициализация балансировки“ — блок, вызывающий функцию `balance_init` библиотеки `NXTway-GS`. Она должна вызываться в начале работы программы, когда сегвей зафиксирован в вертикальном

---

<sup>5</sup> Домашняя страница библиотеки `NXTway-GS`, URL: [http://lejos-osek.sourceforge.net/nxtway\\_gs.htm](http://lejos-osek.sourceforge.net/nxtway_gs.htm) (дата обращения: 17.02.2014)

положении, и инициализирует внутренние переменные программы балансировки, в частности, показания гироскопа в вертикальном положении.

Язык позволяет везде, где могут быть использованы численные значения, использовать и математические выражения. Выражения могут состоять из чисел, арифметических операций, переменных, специальных переменных, содержащих текущие значения сенсоров (называемых сенсорными переменными), тригонометрических функций. Все выражения (и все переменные) всегда имеют вещественный тип, переменные не объявляются, а начинают существовать в месте первого использования. Зависимости между блоками по данным никак не визуализируются, если один блок использует переменную, которой присваивается значение в другом блоке, то корректная последовательность исполнения этих блоков — ответственность программиста.

Пример программы, использующей математические выражения, приведён на рисунке [А.2](#). Эта программа решает задачу движения робота по чёрной линии, нарисованной на полу — самую популярную задачу на различных соревнованиях по робототехнике. Приведённое решение использует робот с двумя датчиками цвета и двумя моторами, независимо приводящими в движение колёса по бокам робота. Вычисляется разность между показаниями сенсоров, которые должны находиться по разные стороны от линии. Если, например, левый сенсор видит белую область, а правый — чёрную, робот съезжает с линии влево, и ему надо повернуть вправо, чтобы остаться на линии. Умноженная на некий подбираемый эмпирически коэффициент эта разность становится управляющим воздействием на моторы, которое прибавляется к мощности одного мотора и вычитается из мощности другого, тем самым обеспечивая доворот робота.

Интуитивная понятность программы обеспечивается приёмом, часто используемым при проектировании предметно-ориентированных языков: для изображения блоков используются иконки с изображениями, понятными людям, видевшим робот. На блоках управления моторами нарисованы моторы Mindstorms NXT, блоки работы с датчиками выполнены более абстрактно, но тоже интуитивно понятны. Это позволяет успешно пользоваться средой даже людям, которые впервые её видят, но знакомы с конструктором. Кроме того, оказалось важно, что связи рисуются со стрелками на концах, в других сре-

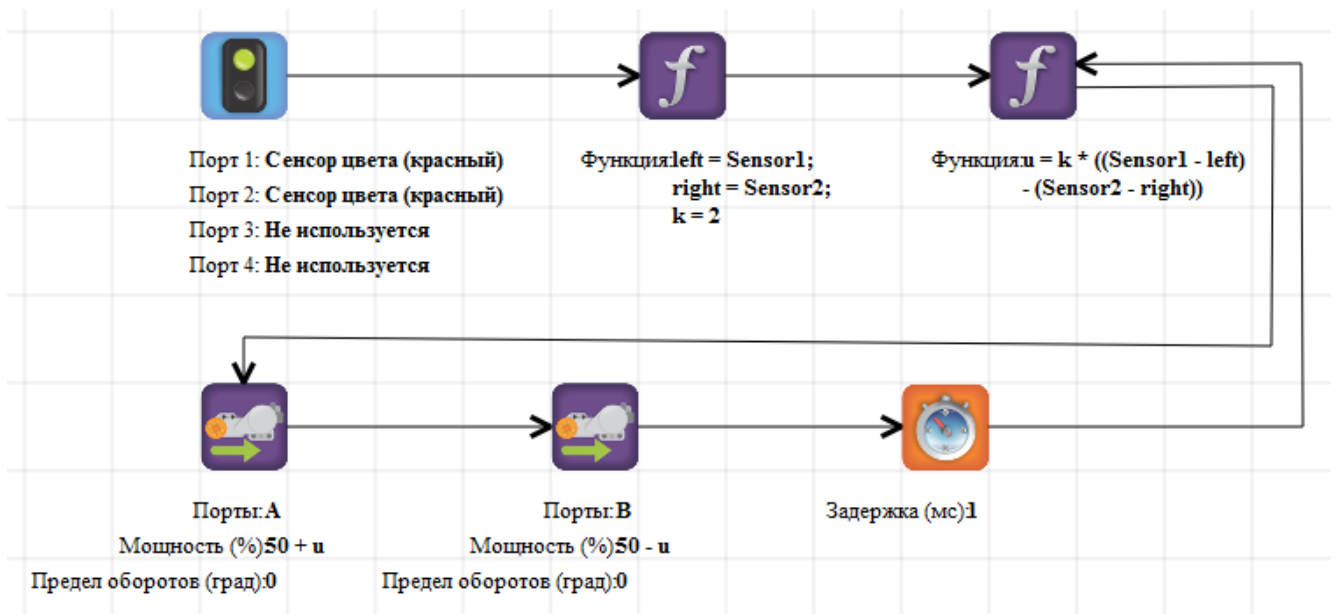


Рисунок А.2: Движение по линии.

дах связи стрелок не имели, и пользователи обратили на это внимание — со стрелками им показалось гораздо понятнее. Это оказалось довольно неожиданным, поскольку направления связей всегда очевидны, и стрелки создавались исключительно как декоративные элементы — для пользователей предметно-ориентированного языка может оказаться важным то, что кажется совершенно неважным авторам языка.

### А.1.5 Инструментальные средства QReal:Robots

С помощью технологии QReal оказалось легко создать визуальный язык и редактор для него, однако инструментальные средства, такие как симулятор робота (далее называемый „двухмерная модель“), интерпретатор диаграмм, управляющий роботом удалённо, генератор кода для загрузки на робота пришлось создавать вручную. Ниже представлен краткий обзор созданных вручную инструментальных средств.

Наиболее объёмным по коду и сложным в реализации является интерпретатор диаграмм. Интерпретатор принимает на вход репозиторий с диаграммой поведения робота и исполняет её, в зависимости от режима, либо на настоящем роботе посылкой ему команд по интерфейсу Bluetooth или USB, либо

на двухмерной модели робота. Интерпретатор создавался с учётом следующих требований.

1. Возможность исполнять программу, управляя роботом по Bluetooth или USB, в зависимости от выбора пользователя.
2. Возможность исполнять программу на двухмерной модели робота. Двухмерная модель должна эмулировать окружение, способное взаимодействовать со всеми видами сенсоров — должна быть возможность задать положение стен, фиксируемых датчиками касания и расстояния, и непроходимых для робота, и разноцветных линий на полу, видимых для датчиков света и цвета.
3. Архитектура интерпретатора должна позволять быстро (в течение единиц часов) добавлять поддержку новых блоков визуального языка, и новых видов оборудования. Должна быть возможность относительно просто и без серьёзных изменений в архитектуре поддерживать новые виды устройств, включая другие робототехнические конструкторы, отличные от NXT, но имеющие схожую архитектуру.

Архитектура интерпретатора представлена на рисунке [A.3](#).

Главный класс, который предоставляет свои функции графическому интерфейсу пользователя и может быть вызван из ядра системы — Interpreter. Он содержит в себе логическую модель робота, таблицу блоков и список активных потоков. Логическая модель робота (класс RobotModel) играет роль программного интерфейса к устройству, и состоит из объектов (наследников RobotPart), каждый из которых реализует программный интерфейс к какому-либо элементу робота — одному из датчиков, мотору, блоку управления. Эти объекты предоставляют высокоуровневые команды, которыми можно пользоваться при реализации поведения блока визуального языка, например, включение мотора, проигрывание звука. Таблица блоков — это отображение блоков на диаграмме на объекты, реализующие их поведение. Для каждого блока языка существует свой класс (наследник Block), который его реализует, способный принять управление, выполнить какое-то действие (возможно, асинхронно), и вернуть интерпретатору следующий блок, который надо выполнить. Объекты

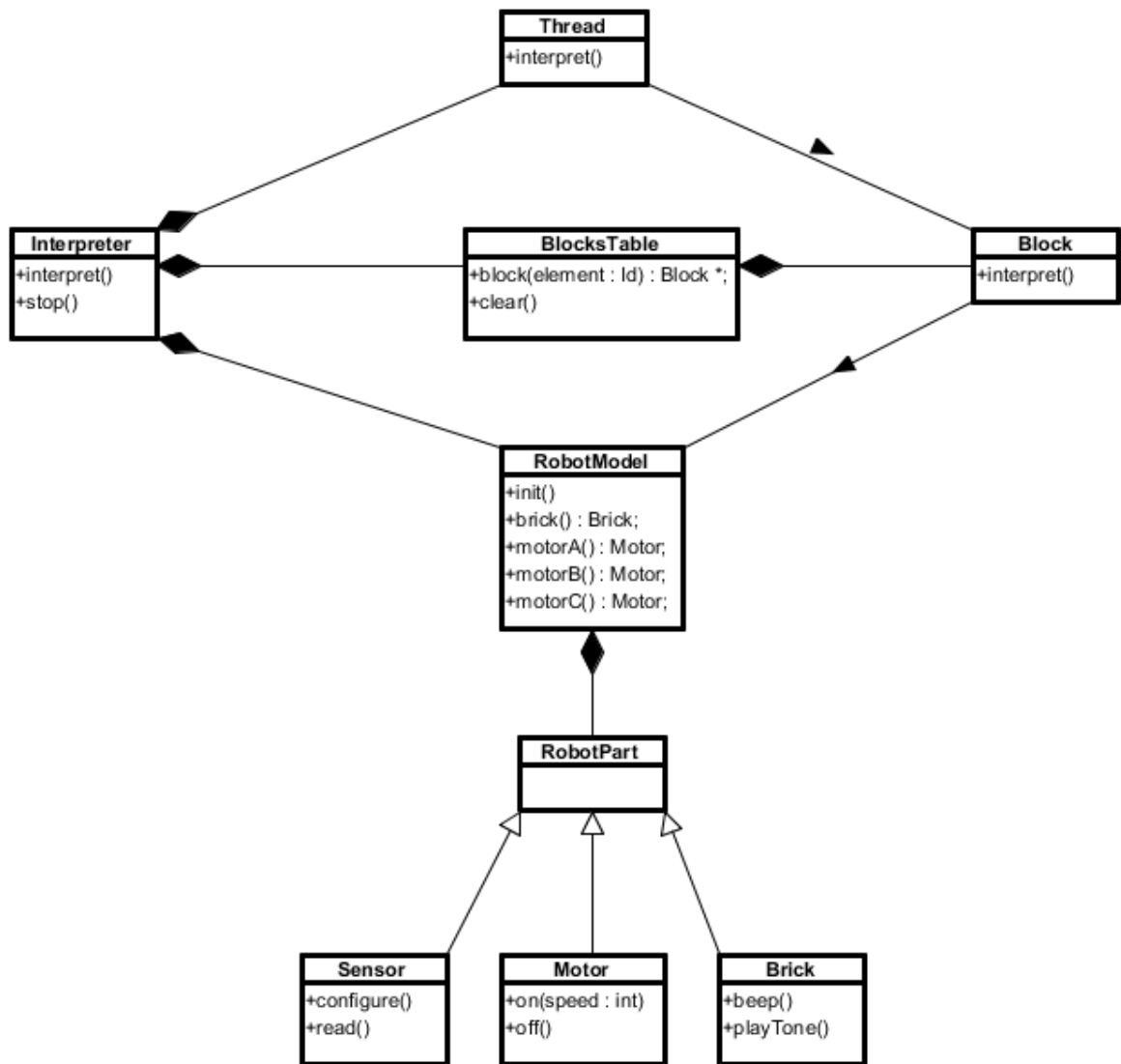


Рисунок А.3: Архитектура интерпретатора.

этих классов создаются для каждого блока при начале работы интерпретатора и помещаются в таблицу, чтобы, когда управление дойдёт до какого-либо блока на диаграмме, найти по идентификатору блока соответствующий ему объект в таблице.

Логическая модель робота и её составные части могут быть реализованы по-разному, давая возможность прозрачно для реализации блоков исполнять программу на реальном роботе, двухмерной модели, или, потенциально, другом устройстве. Для реализации этого использован паттерн „Мост“, дающий возможность для иерархии классов, видимых реализациям блоков, использовать заменяемые реализации. Модель робота содержит ссылку на реализацию модели робота, которая может быть либо логической моделью реального робота,

либо логической моделью двухмерной модели, либо пустой моделью (модель, которая не посылает команды никуда, а просто эмулирует некоторое фиксированное поведение, полезна для начальной отладки программы). Каждый класс, представляющий датчик, имеет ссылку на реализацию, которая тоже может быть одного из трёх видов, аналогично самой модели. Конкретная модель может создавать модели датчиков своего типа, поэтому достаточно инициализировать интерпретатор требуемым типом модели, вся дальнейшая необходимая инициализация выполнится автоматически.

В случае с двухмерной моделью классы-реализации элементов модели перенаправляют запросы в объект класса `D2RobotModel`. Этот класс занимается собственно симуляцией робота, обчитывая его перемещение, реакции на команды от классов-реализаций. Внешнее окружение моделирует отдельный класс `WorldModel`, и `D2RobotModel` может запрашивать показания датчика в заданной позиции и с заданным направлением у этого класса. Отображением результатов симуляции занимается класс `D2ModelWidget`. Архитектура этой части системы представлена на рисунке A.4.

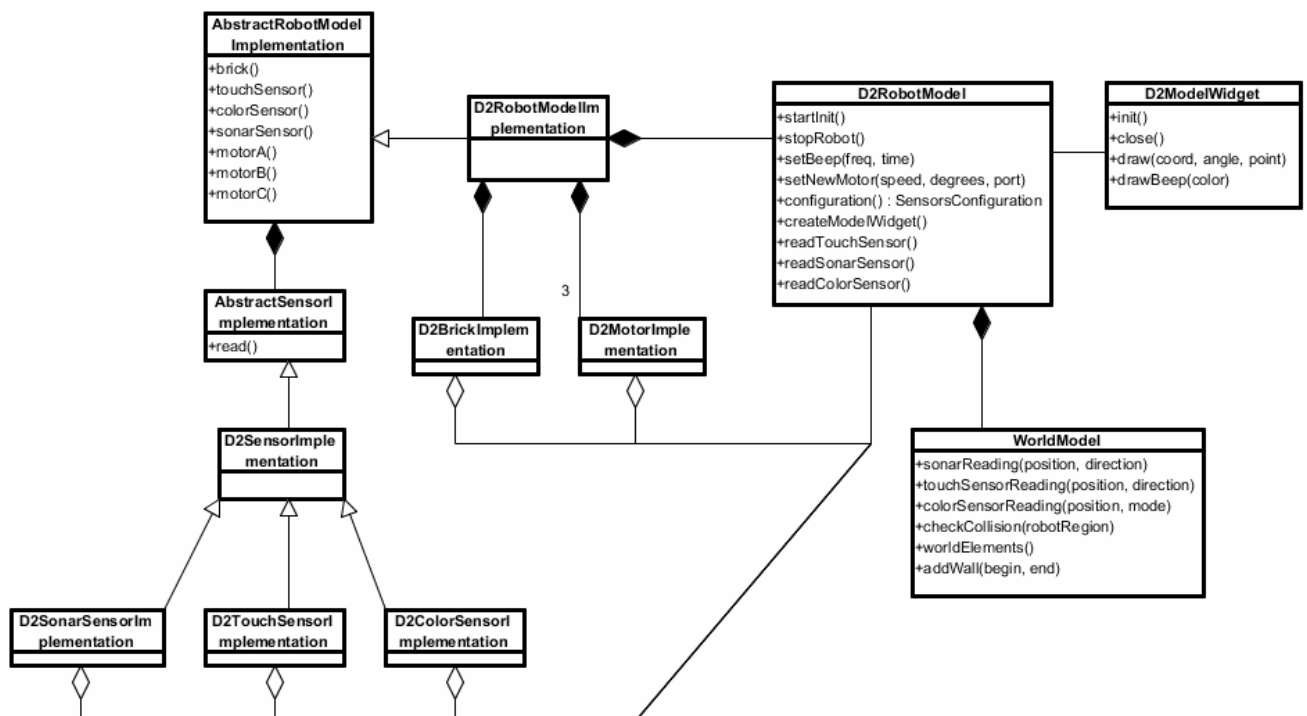


Рисунок A.4: Архитектура двухмерной модели.



Управление роботом по интерфейсам Bluetooth и USB использует одну логическую модель робота, класс `RealRobotModel`. Это оказалось возможно, потому что оба эти интерфейса используют одинаковый набор команд в примерно одинаковом формате, разнятся лишь заголовки пакетов с командами, передаваемых на робот. Поэтому оказалось возможным вынести транспортный уровень в отдельный набор классов, где инкапсулирована низкоуровневая работа с каналами передачи, и логическая модель просто параметризуется нужным транспортным уровнем. Транспорт для USB использует драйвер робота `Fantom` из поставки конструктора, транспорт для Bluetooth работает непосредственно с системными средствами передачи (для работы с виртуальным COM-портом Bluetooth-канала используется сторонняя библиотека `QextSerialPort`).

Окно двухмерной модели представлено на рисунке [A.5](#). Двухмерная модель моделирует одну предопределённую конфигурацию робота — трёхколёсную тележку с двумя ведущими колёсами, используемую во многих задачах, связанных с движением (например, в робофутболе), однако можно задавать позицию и направление датчиков. Цветные линии на полу, видимые датчикам света и цвета, можно рисовать инструментами „Линия“, „Карандаш“, „Эллипс“, стены — инструментом „Стена“. Все элементы после размещения в рабочей области редактируемы, так что двухмерная модель близка по функциональности к простому векторному редактору. Имеется возможность сохранить нарисованную модель окружения в xml-файл.

Генератор кода на текстовом языке реализован в виде отдельного подключаемого модуля и может работать независимо от интерпретатора. Он генерирует код на языке C с использованием программного интерфейса операционной системы `nxtOSEK`, на данный момент считающейся самой быстрой операционной системой для роботов `Mindstorms NXT`. Для того, чтобы сгенерированный код можно было загружать на робот, требуется, чтобы на роботе была установлена `nxtOSEK`, а на компьютере с `QReal:Robots` — кросскомпилятор и комплект средств разработки для этой ОС. Всё необходимое для кросскомпиляции поставляется в виде отдельного инсталляционного пакета, который требует установленного `QReal:Robots` для установки. Решение не включать средства разработки в основной инсталляционный пакет `QReal:Robots` было принято из-за их большого объёма, что могло создать трудности при загрузке инсталляционного

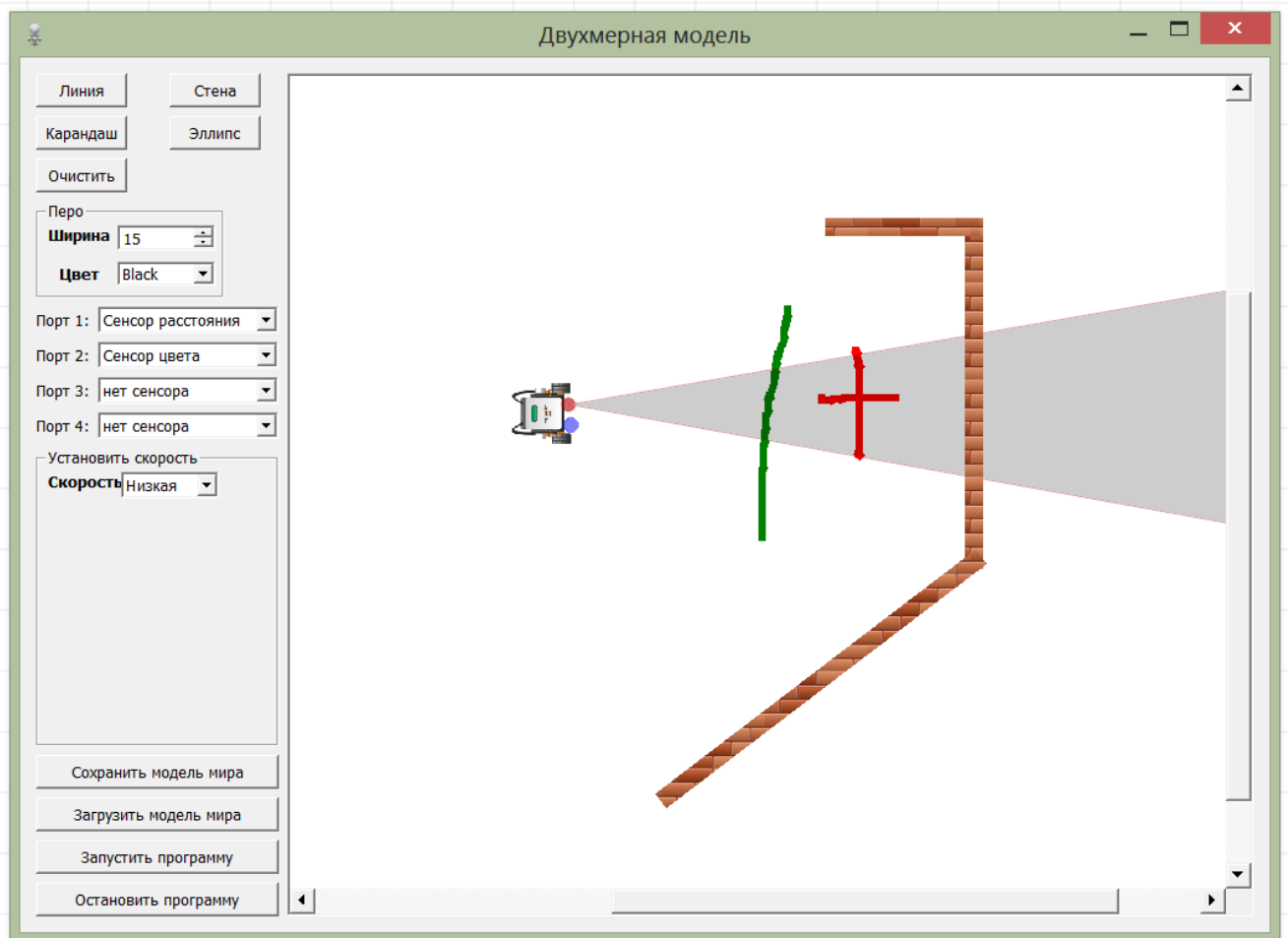


Рисунок А.5: Окно двухмерной модели.

пакета пользователям, которым генерация текстовых программ не требуется. Генератор порождает .с-файл с кодом программы и .oil-файл с настройками её запуска, после чего запускается кросскомпилятор, собирающий бинарный образ программы, который потом загружается на робот по USB посредством утилиты из поставки конструктора. Для пользователя этот процесс прозрачен, ему достаточно нажать на кнопку „Загрузить“ и дождаться сообщения об успешном завершении процесса загрузки. Если пользователю не требуется загружать программу на робот, он может просто сгенерировать код, просмотреть и отредактировать его во встроенном в QReal:Robots текстовом редакторе с подсветкой синтаксиса.

Генератор организован по шаблонной схеме: имеется файл с шаблоном порождаемого кода, в котором отмечены места, параметризуемые информацией из модели. Такие места, в свою очередь, могут сами заполняться текстами,

сформированными с помощью шаблонов, и т.д., что позволяет генерировать сложно структурированные повторяющиеся фрагменты кода. Пример шаблона верхнего уровня для генерации кода программы приведён ниже.

```
#include "kernel.h"
#include "ecrobot_interface.h"
@@BALANCER@@
@@VARIABLES@@

void ecrobot_device_initialize(void)
{
  @@INITHOOKS@@
}

void ecrobot_device_terminate(void)
{
  @@TERMINATEHOOKS@@
}

/* nxtOSEK hook to be invoked from an ISR in category 2 */
void user_1ms_isr_type2(void){ /* do nothing */ }

@@CODE@@
```

С помощью текста, заключённого в „@@“, отмечаются места для вставки параметризованного кода, так называемые заглушки (placeholders), генератор заменяет вхождения заглушек на сгенерированный для них по модели код. В приведённом выше примере заглушка @@BALANCER@@ заменяется на объявления функций и переменных для балансировки сегвея, если блоки работы с сегвеем используются на диаграмме, или пустой строкой, если таких блоков нет. @@VARIABLES@@ заменяется на объявления переменных — генератор в начале работы анализирует все арифметические выражения в программе, формирует таблицу переменных, и все объявления переменных генерирует на место этой заглушки. Заглушки @@INITHOOKS@@ и @@TERMINATEHOOKS@@ за-

меняются на код инициализации и деинициализации датчиков, зависящий от того, какие датчики использовались в программе.

Заглушка @@CODE@@ заменяется на код, сгенерированный по диаграмме. По каждому блоку визуального языка генерируется свой небольшой фрагмент программы, реализующий поведение этого блока. Некоторую сложность представляют конструкции, управляющие потоком исполнения — условные операторы и операторы цикла. В отличие от структурных текстовых языков, где операторы образуют иерархическую структуру, визуальный язык позволяет связывать любой блок с любым блоком, что позволяет рисовать неструктурные программы, где поток управления попадает внутрь „тела“ условного оператора или цикла извне. Это равносильно использованию оператора `goto` в текстовых языках, при этом визуальный язык QReal:Robots (как и многие другие подобные языки) не визуализирует структурные конструкции и нарушение структурности может произойти незаметно для программиста. Пример неструктурной программы приведён на рисунке A.6. Генератор применяет ряд эвристик для анализа потока исполнения программы и поиска структурных условных операторов и циклов. В случае, если структурный код не может быть порождён по данной диаграмме (как, например, представленной на рисунке), генератор выдаёт ошибку и не генерирует код. Такое решение было принято в связи со спецификой применения генератора — он должен порождать по возможности качественный и читаемый код, очевидным образом связанный с диаграммой, поскольку служит для обучения школьников программированию. Это исключает использование `goto` в сгенерированном коде, и делает нежелательным использование техник `goto elimination`, поскольку они предполагают раскопирование неструктурных участков кода. Интерпретация таких диаграмм, тем не менее, вполне возможна.

## A.1.6 Опыт применения QReal

### Метамодель языка QReal:Robots

Визуальный язык QReal:Robots создавался в метаредакторе QReal. Мета-модель одной из версий языка представлена на рисунке A.7. Как видно, вся метамодель языка уместается на одном экране. Некоторая содержательная ин-

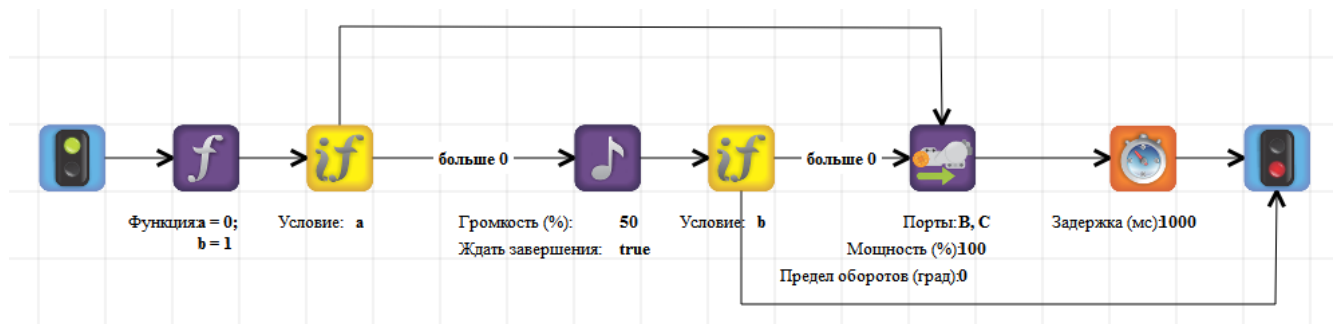


Рисунок А.6: Пример неструктурной программы.

формация на рисунке не показана, поскольку она находится в свойствах элементов и редактируется через редактор свойств (например, внешний вид элемента). Мета модель всё же слишком велика, чтобы на рисунке были видны все детали, поэтому ниже приводится словесное описание изображённого.

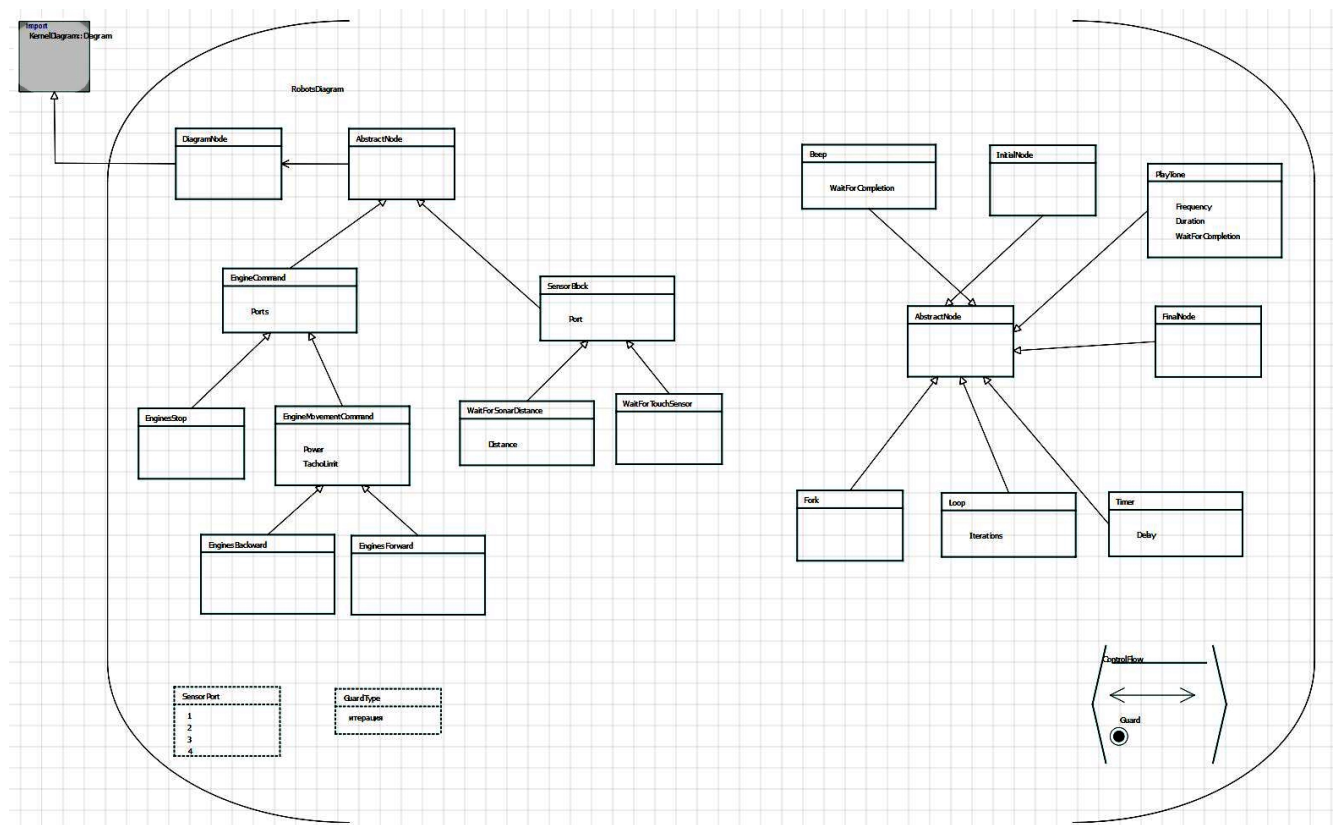


Рисунок А.7: Мета модель языка QReal:Robots.

Корневым элементом любой диаграммы визуального языка является узел „Диаграмма“ (сущность `DiagramNode` в метамодели). Она наследуется от узла „Диаграмма“, объявленного в импортированной метамодели `KernelDiagram`,

получая от неё все свойства, там объявленные. Корнем иерархии наследования узлов языка является узел `AbstractNode`, который связан отношением `Contains` с `DiagramNode`, давая тем самым возможность всем своим наследникам располагаться на диаграмме. `AbstractNode` на рисунке [A.7](#) имеется в двух экземплярах, поскольку от него наследуются все узлы языка, и отношения наследования, соединённые с одним `AbstractNode`, сделали бы диаграмму трудночитаемой — это пример применения принципа разделения логической и графической моделей. Для узла `AbstractNode` не задано графическое представление, поэтому он не будет отображаться в палитре, и не может быть использован на диаграммах. От `AbstractNode` наследуются несколько конкретных узлов, видимых в палитре, таких как `InitialNode`, `FinalNode` (блоки „Начало“ и „Конец“ соответственно), и два абстрактных узла — `EngineCommand` (блок управления двигателями) и `SensorBlock` (блок работы с датчиками). Эти узлы абстрактные, поскольку в них определяются свойства, общие для всех их потомков (например, порты, к которым применяются команды двигателей), сами они никакой смысловой нагрузки не несут и не видны в палитре. От `EngineCommand` наследуются абстрактный блок `EngineMovementCommand`, имеющий свойство `Power` (мощность двигателя в процентах от максимальной), и конкретный блок `EngineStop`, который не требует параметра мощности, и поэтому не наследуется от `EngineMovementCommand`. Наследники `EngineMovementCommand` — блоки `EnginesBackward` и `EnginesForward`, которые никаких своих дополнительных свойств не имеют. Кроме блоков, в метамодели описано одно отношение (`ControlFlow`, „Поток управления“), и два типа-перечисления: `SensorPort` (порты, к которым может быть подключен датчик), и `GuardType` (тип условия над связью „Поток управления“, используемого в блоках „Условие“ и „Цикл“).

## Обсуждение

Первые версии языка создавались в метаредакторе, первый прототип редактора был автоматически сгенерирован по метамодели в весьма короткие сроки — всего за два-три часа, из которых большая часть времени ушла на поиск иконок для отображения блоков. Возможность быстро получить редактор языка по метамодели и удобство редактирования метамодели в метаредакторе позволило не только существенно сократить время разработки языка, но и

дать возможность экспериментировать, меняя свойства и внешний вид элементов. Кроме того, метамодель оказалось довольно удобно сопровождать, причём не только автору языка — работа над QReal:Robots в дальнейшем во многом была передана студентам, которые быстро разбирались в метаредакторе и могли самостоятельно добавлять в язык новые блоки или менять существующие. Рассматривалась даже возможность позволить самим пользователям менять свойства блоков языка (например, учителю скрыть ненужные в данный момент свойства блоков перед занятием), однако пока она не была востребована. Тем не менее, последние версии языка используют не графическую метамодель, а её XML-представление — за время работы над проектом XML-вариант метаязыка расширился новыми возможностями, которые не были реализованы в графическом метаязыке, например, задание групп блоков в палитре. Несмотря на это, вносить изменения в язык всё же достаточно просто для людей, занимавшихся разработкой QReal (например, студентов).

Если технология сильно сократила время разработки визуального языка и редактора для него, то с созданием других средств инструментальной поддержки, таких как интерпретатора диаграмм или генератора кода, она смогла помочь в значительно меньшей степени. Это вполне ожидаемо, поскольку эти части содержат большое количество специфичных знаний предметной области, которые невозможно обобщить и вынести на метауровень, сделав частью технологии. Например, робот управляется удалённо так называемыми прямыми командами, которые можно посылать по интерфейсам Bluetooth или USB, и как формирование пакета с командой, так и логика работы с каналом передачи данных должны реализовываться вручную. Такого рода работы останутся трудоёмкими при любом уровне поддержки со стороны DSM-платформы, поскольку очень многое требуется изучить самому разработчику: систему прямых команд робота, API операционной системы робота для генерации кода в неё, работу с Bluetooth, USB, кросскомпиляцию, прошивку робота, загрузку программы на робот, и т.д. Тем не менее, DSM-платформа может взять на себя часть рутинной работы, и, предоставляя некий набор шаблонов и методологию, организовать и направить деятельность автора DSM-решения.

Наиболее интересной для автоматизации частью представляется генератор кода, поскольку содержит много похожего от блока к блоку кода. Генератор

содержит неспецифичные для конкретного языка части, такие как механизм анализа потока управления, и такую низкоуровневую функциональность, как код формирования выходного файла или обхода модели. Некоторые части допускают обобщение до наиболее общей задачи генерации кода (любой генератор, например, должен выводить результаты в некий файл или файлы). Некоторые, такие как анализ потока управления, могут быть применены для всех языков, имеющих в своей основе семантику сетей Петри. По результатам проекта QReal:Robots было предложено два возможных направления автоматизации создания генераторов: вынесение общей части функциональности в библиотеку (или объектно-ориентированный каркас) разработки генераторов, и создание инструмента, который бы позволял описывать правила генерации на специализированном языке. Первое направление было реализовано, библиотека поддержки разработки плагинов QReal теперь содержит код, общий для всех генераторов, куда вынесена функциональность работы с шаблонами, с выходными файлами, механизм отображения ошибок генерации.

Была также предпринята попытка переписать генератор QReal:Robots на языке задания правил генерации, описанном в главе 4, но она была признана неуспешной: оказалось, что знать ещё один текстовый язык (язык описания правил генерации) и работать со средством, порождающим из него генератор (в котором может быть довольно большое количество своих ошибок) даже менее удобно, чем писать код на языке общего назначения наподобие C++. Связано это ещё и с тем, что для C++ имеются хорошие среды разработки, тогда как на „самодельном“ текстовом языке приходилось писать в довольно неудобном редакторе. В результате этого эксперимента был сделан вывод, что язык описания правил генерации имеет смысл попробовать сделать графическим (несмотря на то, что он должен активно работать с текстом), и использовать в таком случае все возможности самой DSM-платформы. Либо же DSM-платформа помимо создания инструментальных средств для визуальных языков должна позволять создавать удобные инструментальные средства и для текстовых языков, чтобы предметно-ориентированные текстовые языки могли эффективно применяться как в DSM-решениях, так и для нужд самой DSM-платформы так же, как применяются сейчас визуальные языки. Оба эти направления приводятся здесь, как возможности для дальнейшего исследования, и выходят за рамки данной



работы — в QReal как на момент создания DSM-решения QReal:Robots, так и на момент написания этой диссертации для написания генераторов используется язык C++.

Вторая возможная цель автоматизации создания дополнительных инструментальных средств для языка — интерпретатор диаграмм. В QReal:Robots интерпретатор был написан целиком на C++, и, как и в случае с генератором, усилия на его разработку удалось бы значительно сэкономить, если бы существовала библиотека (или каркас) с общим для всех интерпретаторов кодом. Сам процесс интерпретации как последовательности передачи токенов исполнения и выполнения некий действий в узле, получившем токен, общий для всех языков с семантикой, основанной на сетях Петри, поэтому может быть реализован языконезависимым образом. Действия, выполняемые в узлах, сильно языкозависимы (не все, блоки ветвления, параллельного исполнения, начала и конца программы могут присутствовать во многих языках и иметь одинаковую семантику). Специфичные для конкретного языка действия можно реализовывать на языке общего назначения, либо же на каком-либо интерпретируемом языке общего назначения, наподобие Python или F# [80]. Использование интерпретируемых языков имеет весомое преимущество — действия можно модифицировать без пересборки интерпретатора, это могут делать в том числе и сами пользователи DSM-решения. Недостаток такого подхода очевиден: на всех машинах, где используется DSM-решение, требуется наличие интерпретатора выбранного языка. Поскольку на данный момент задача интерпретации достаточно сложных диаграмм возникала только в QReal:Robots, задача автоматизации создания интерпретаторов в рамках проекта QReal серьёзно не рассматривалась, и соображения выше приведены здесь как возможные направления дальнейшей работы.

### A.1.7 Результаты проекта QReal:Robots

Проект QReal:Robots, по мнению автора, полностью доказал состоятельность предметно-ориентированного подхода и работоспособность подходов, реализованных в проекте QReal и описываемых в данной работе. В результате проекта с помощью DSM-платформы была создана система визуального про-

граммирования, которая оказалась не хуже разработанных вручную, при этом время на разработку визуального редактора для этой системы удалось с помощью DSM-платформы сократить в несколько десятков раз по сравнению с разработкой редактора с такой же функциональностью вручную. Однако выигрыш во времени при разработке и доведении до отчуждаемого продукта всей системы в целом оказался не таким значительным, как при разработке визуального редактора — многие задачи оказались неавтоматизируемы в принципе. В ходе проекта были выявлены задачи, решение которых позволило бы уменьшить трудоёмкость создания подобных систем. В целом DSM-подход показал себя полезным на практике, и дальнейшие исследования в этой области помогли бы эффективно создавать специализированные среды визуального программирования и для гораздо более узких предметных областей.

Результат проекта — среда QReal:Robots — была представлена на „Открытых состязаниях Санкт-Петербурга по робототехнике“ 2012 года и на робототехническом фестивале „Робофест 2012“ в Москве. В качестве доказательства применимости этой среды к реальным задачам, решаемым школьниками с помощью сред-аналогов, команда студентов приняла участие в соревнованиях в движении робота по линии с программой, реализованной на QReal:Robots. Несмотря на то, что для студентов кафедры системного программирования участие в соревнованиях стало практически первым опытом решения задач кибернетики, им удалось показать достойные результаты, уступив фактически лишь специально созданным для этой задачи роботам (которые по техническим характеристикам имели преимущество перед роботами, собранными из конструктора Mindstorms NXT). На соревнованиях было проведено анкетирование школьников, которым было предложено решить некоторые простые задачи на QReal:Robots, результаты показали, что пользовательский интерфейс продукта достаточно хорош, чтобы вызвать у пользователей симпатию. С использованием QReal:Robots было также проведено занятие в детском робототехническом лагере в г. Сиверский летом 2012 года, где школьники успешно решили задачу движения робота по линии на двухмерной модели.

Кроме соревнований, среда QReal:Robots представлялась на стендовых докладах при соревнованиях, и на нескольких специализированных конференциях школьных учителей и методистов, где вызвала большой интерес среди учите-

лей. Некоторые из них впоследствии обращались к автору с предложением оказать посильную помощь в разработке и тестировании, некоторые использовали QReal:Robots в своих занятиях. Таким образом, QReal:Robots можно считать полноценным отчуждаемым продуктом, востребованным и имеющим реальных пользователей, в том числе и не владеющих программированием.

## **А.2 Среда программирования распределённых мобильных приложений QReal:Ubiq**

Так же, как и QReal:Robots, QReal:Ubiq появилась из практической задачи, с которой к нам обратились представители индустрии — программирование мобильных приложений в рамках архитектуры платформы Ubiq Mobile. В этом случае также имелась достаточно узкая предметная область, но приложения, которые требовалось разрабатывать, были гораздо ближе к типичным разрабатываемым промышленно приложениям, чем QReal:Robots, в частности, имели и нетривиальную логику, и пользовательский интерфейс. Поэтому данная задача была интересна как некоторая попытка заменить визуальным программированием программирование на текстовых языках для достаточно большого класса задач. Успешная реализация подобной технологии открыла бы возможность для создания целой серии технологий, направленных на различные платформы.

### **А.2.1 Постановка задачи**

Платформа Ubiq Mobile [81] предназначена для создания распределённых мобильных приложений со сложной серверной логикой, таких как бизнес-приложения, многопользовательские игры и т.д. Архитектурно платформа представляет собой сервер с выполняемыми на нём сервисами и набор мобильных клиентов, связанных с сервером по проприетарному протоколу. Клиенты могут работать на телефонах с очень маленькими возможностями, такими как устаревшие Java-телефоны, в этом случае используется тонкий клиент, который только отображает формируемое на сервере изображение пользовательского интерфейса и передаёт на сервер события нажатий на клавиши. Для более современных моделей возможна передача на клиент построенного на сервере

дерева визуальных элементов управления, которые отображаются на телефоне средствами его операционной системы. Возможно также написание толстого клиента, реализующего часть бизнес-логики на телефоне. В любом случае, на сервере для каждого подключённого клиента создаётся отдельный поток, в котором и работает большая часть клиентской логики программы. Этот поток может общаться с потоками, относящимися к серверной части сервиса, таким образом достигается возможность взаимодействовать с серверным бэкэндом и другими пользователями. Таким образом, сообщения между сервером и клиентом физически передаются между потоками, работающими на сервере, по каналу данных передаётся только результат обработки.

С точки зрения прикладного программиста процесс создания сервиса в Ubiq Mobile представляет собой программирование клиентского и серверных потоков как реализацию подклассов одного из классов платформы Ubiq Mobile. Требуется определить формат сообщений, которыми обмениваются клиент и сервер (в виде C#-классов) и описать методы, реагирующие на приём сообщения на клиенте и на сервере. Также потребуется описать пользовательский интерфейс приложения, также в виде набора объектов на C# (это делается в достаточно декларативном и удобном для текстового программирования стиле). Результат должен быть оформлен в виде C#-библиотеки, собран вместе с библиотеками Ubiq Mobile, выложен в папку с исполняемым файлом сервера Ubiq Mobile и добавлен в его конфигурационный файл. Очевидно, что все подобные библиотеки имеют похожую структуру, и для их создания требуется выполнение повторяющихся действий, поэтому имеется возможность для автоматизации.

Авторы платформы Ubiq Mobile выбрали DSM-платформу QReal для реализации технологии и набора визуальных языков, которые упростили бы разработку мобильных приложений под эту платформу. Целью совместной работы было создание на базе QReal технологии, позволявшей описывать в визуальном виде только изменяющуюся от приложения к приложению часть, и генерировать по визуальной модели проект для среды разработки Visual Studio, который был бы готов к сборке итоговой библиотеки, не требуя при этом ручных правок.

Разработка была разделена на несколько этапов. Целью первого этапа было создание прототипа технологии, чтобы оценить применимость DSM-подхода к этой задаче. Прототип должен был генерировать логику поведения клиентской

части приложения, позволяющего осуществлять видеонаблюдение: веб-камера, подключённая к компьютеру, передаёт с определённой частотой кадры на сервер Ubiq Mobile, к серверу могут подключаться мобильные клиенты, которым перенаправляются кадры с камеры. И камер, и клиентов может быть несколько в один момент времени, клиент может указать, с какой камеры ему необходимо получать кадры. Пользовательский интерфейс клиентского приложения на этом этапе должен был описываться вручную. Задачи второго этапа включали в себя реализацию полноценной технологии, включающей в себя визуальное задание пользовательского интерфейса приложения. Ниже приводится подробное описание только первого этапа, поскольку работы по второму этапу ещё не закончены и ведутся со значительно меньшим участием автора данной диссертации.

### **А.2.2 Визуальный язык QReal:Ubiq**

Фазы анализа применимости и анализа предметной области для данного DSM-решения состояли в консультациях с авторами технологии Ubiq Mobile, выступавшими здесь в роли экспертов предметной области и в анализе существующих исходных кодов сервисов, написанных под эту платформу. В данном случае для выделения ключевых сущностей предметной области и создания языка был выбран подход „от исходного кода“, поскольку у авторов платформы уже имелся достаточно большой набор работающих примеров и понимание того, что хотелось бы автоматизировать.

В результате анализа исходного кода было принято решение разработать три различных визуальных языка, отвечавших двум основным областям вариативности создаваемых сервисов: описание структуры сообщений, которыми обмениваются клиент и сервер, и описание логики обработки сообщений на клиенте и на сервере. Третий визуальный язык был нужен для описания связи между различными диаграммами первых двух языков, и диаграммы на нём служили чем-то вроде заголовка визуальной модели системы.

Визуальные языки QReal:Ubiq состоят из следующих сущностей.

1. Мастер-диаграмма (Ubiq Master Diagram) описывает главный класс серверного приложения и является по сути связью между остальными диа-

граммами системы. Эта диаграмма должна содержать ровно один элемент Master Node, в котором описано, откуда сервер получает сообщения и какие обработчики необходимо вызвать при их получении. Такая диаграмма должна быть одна в проекте. Виды узлов на этой диаграмме таковы.

- (a) Constant — константа, описанная внутри класса сервера. Имеет имя, тип и значение. Может находиться только внутри Master Node.
- (b) Field — поле класса сервера. Имеет имя, тип, значение по умолчанию, может быть отмечено как статическое. Может находиться только внутри Master Node.
- (c) Handler — описывает источник событий для сервера, имеет имя (которое может принимать два значения: либо OnTcpIpMessage, что означает, что сервер может обрабатывать сообщения, полученные по TCP-IP, либо OnMailBoxMessage, что означает, что сервер может обрабатывать сообщения от другого процесса). Может находиться только внутри Master Node. Каждый такой узел должен быть связан с помощью провязки с диаграммой активностей, описывающей поведение соответствующего обработчика.
- (d) Master Diagram — корневой узел диаграммы, все остальные узлы должны быть вложены в него.
- (e) Master Node — описание класса серверной части приложения. Может содержать в себе описания обработчиков событий, констант, полей, препроцессоров (узлы Handler, Constant, Field, Preprocessor), имеет свойства „имя“ (имя класса сервера), initCode (произвольный код на C#, генерирующийся в конструктор класса), onTcpIpCloseHandler (произвольный код на C#, выполняющийся при закрытии TCP-IP-соединения). Может быть связан провязкой с диаграммами активностей, описывающими реализацию вспомогательных функций, которые сгенерируются как методы этого класса.
- (f) MasterDiagramComment — комментарий.
- (g) Preprocessor — метод, вызываемый перед передачей сообщения обработчику, предназначенный для преобразования сообщения перед

передачей обработчику. Должен быть провязан с диаграммой активностей, реализующей этот метод.

2. Диаграмма структур данных (Ubiq Data Structures), на ней описывается класс Message, предназначенный для обмена сообщениями между клиентом и сервером или сервером и другими процессами, а также другие классы, служащие данными. Такая диаграмма должна быть одна в проекте. Узлы на этой диаграмме таковы.

- (a) Comment — комментарий.
- (b) Custom Class — произвольный класс, содержащий данные. Имеет свойство „имя“ (которое генерируется в имя класса), содержит в себе поля (элементы типа Field).
- (c) Data Structures Diagram — сама диаграмма, корневой элемент.
- (d) Enum Element — элемент перечисления. Имеет имя и значение. Может находиться в узлах Message Codes и Error Codes, будет сгенерирована как константа внутри класса Message.
- (e) Error Codes — описание кодов ошибок, используемых в классе Message. Должен быть один на диаграмме. Содержит в себе элементы Enum Element.
- (f) Field — произвольное поле класса Message или произвольного класса (может лежать в Message Class или Custom Class), генерируется в свойство соответствующего класса. Имеет имя (имя свойства), значение по умолчанию, тип, может быть сериализуемым или несериализуемым.
- (g) Message Class — описание класса Message, должен быть один на диаграмме. Содержит в себе только поля (элементы Field).
- (h) Message Codes — описание кодов сообщения, используемых в классе Message. Должен быть один на диаграмме. Содержит в себе элементы Enum Element.

3. Диаграмма активностей (Ubiq Activity Diagram). На таких диаграммах задаётся поведение обработчика сообщений, метода, препроцессора и т.д.

В модели их может быть несколько (по числу обработчиков). Содержание несколько разнится в зависимости от вида диаграммы — диаграмма активностей для обработчика, диаграмма активностей для препроцессора, вспомогательной функции. Диаграмма для обработчика обычно состоит из нескольких цепочек операторов, начинающихся с узла `HandlerStart`, препроцессор начинается с `Initial Node`, функция — с `Function Signature`. Узлы, используемые на всех видах диаграммы активностей, таковы.

- (a) `Action` — действие, имеет свойство „имя“, которое должно содержать код на `C#`. Он будет без изменений скопирован в результат генерации.
- (b) `Activity Diagram` — сама диаграмма.
- (c) `Activity Final Node` — завершающий узел, означает конец цепочки операторов. Полезен, например, для рисования пустой ветки `else`, в остальных случаях необязателен.
- (d) `Actual Parameter` — фактический параметр, передаваемый в функцию при вызове. Имеет имя, которое должно быть `C#`-кодом, подставляемым вместо параметра при вызове функции, может содержаться только в узле `Function Call`.
- (e) `Comment` — комментарий.
- (f) `Control Flow` — направленная связь, означающая передачу управления от оператора к оператору. Имеет свойство `guard`, используемое при генерации узлов `Decision Node`.
- (g) `Decision Node` — оператор `"if"`. Должен иметь ровно две исходящие связи, ровно у одной из которых свойство `guard` не пусто. Обе ветки должны сходиться либо на одном `Merge Node`, либо на одном `Activity Final Node`. Свойство `guard` становится условием оператора `if` в сгенерированном коде.
- (h) `Formal Parameter` — формальный параметр, описываемый в заголовке вспомогательной функции. Может содержаться только в узле `Function Signature`. Имеет имя и тип.



- (i) Function Call — вызов функции. Имеет имя, которое должно совпадать с именем диаграммы активностей, реализующей вызываемую функцию. Содержит набор параметров (узлов Actual Parameter), и максимум один узел Return Value, показывающий, куда положить результат вызова функции.
- (j) Function Signature — описание вспомогательной функции, с него начинается цепочка операторов диаграммы активностей для вспомогательной функции. Имеет имя (должно совпадать с именем диаграммы), и тип возвращаемого значения. Может содержать в себе формальные параметры (типа Formal Parameter).
- (k) Handler Start — начало обработчика сообщения. С него начинается цепочка операторов на диаграмме активностей обработчика. Имеет имя, которое должно совпадать с именем константы — типа сообщения, описанного на диаграмме структур данных.
- (l) Initial Node — начальный узел, с него начинается цепочка операторов диаграммы активностей препроцессора.
- (m) Merge Node — точка слияния двух веток выполнения оператора if.
- (n) Package — вспомогательный узел, предназначенный для организации диаграмм активностей в связанные блоки в логической модели. Семантической нагрузки не несёт.
- (o) Return — возврат значения из вспомогательной функции, генерируется в оператор return C#. Равносителен узлу Action с оператором return <возвращаемое значение>;
- (p) Return Value — указывает, куда положить возвращаемое функцией значение. Может содержаться только в узле Function Call. Имеет имя (код на C#, обычно имя переменной) и тип. Если тип не пуст, генерируется объявление переменной для хранения результата, если пуст, считается, что переменная уже объявлена.

Как видно из описания, язык имеет гибридную структуру, то есть для задания программы активно используются и визуальные, и текстовые символы. При этом визуальная и текстовая части языка частично взаимозаменяемы — язык

позволяет не рисовать диаграмму обработчика сообщения, а написать весь код вручную в единственном блоке. В качестве текстовой части используется язык C#, код на котором непосредственно генерируется в выходные файлы. Технология не имеет синтаксического анализатора языка C#, поэтому не может выполнять никаких действий над содержимым блоков, и даже синтаксические ошибки в коде будут обнаружены только после генерации. Язык задания логики обработчиков (диаграмма активностей) по сути представляет собой диаграмму активностей UML, модифицированную для того, чтобы отразить особенности Ubiq Mobile.

### А.2.3 Обсуждение

Главный недостаток получившегося набора языков вытекает из выбранного способа анализа предметной области: визуальный язык слишком привязан к целевому коду, так что без достаточно хорошего владения технологией Ubiq Mobile или специальной подготовки создавать программы оказывается довольно сложно. Фактически, надо представлять, в какую часть программы будет сгенерирован тот или иной код, который пишется в узлах Action и различных других узлах, где допускается ввод кода на C# вручную. Смысл некоторых свойств практически невозможно объяснить человеку, не знакомому детально с Ubiq Mobile (поэтому такие свойства опускались при описании языка). Кроме того, наличие в C# переменных и полей классов делает возможным модификацию локального или даже глобального состояния из кода в узле Action обработчика, что приводит к неявным и никак не визуализируемым зависимостям по данным, на диаграммах визуализируется только поток исполнения. Фактически оказалось, что очень сложно писать код на C# в элементах визуального языка, не смотря при этом в сгенерированный код. Кроме того, диаграммы активностей более громоздкие, чем код, который они призваны визуализировать, и поэтому в них оказывается сложнее разобраться. Субъективный вывод, сделанный автором данной работы в ходе разработки модельного примера — диаграммы активностей не оправдывают себя, проще и продуктивнее писать код обработчиков вручную.

Перечисленные недостатки являются проблемой для всех гибридных (текстографических) языков, в которых текстовая часть является кодом на целевом языке и не анализируется синтаксически самим DSM-решением. В любом случае, пользователи DSM-решения могут эксплуатировать знания о результатах генерации для создания эффективных по их мнению, но сложных в сопровождении программ, пример такого: объявление переменной в одном блоке и использование её в другом. При изменении потока управления результат генерации может оказаться некорректным. Эти проблемы частично можно решить синтаксическим анализом кода внутри элементов, но это может быть технически сложно в реализации. Кроме того, подобный подход требует наличия качественного интегрированного текстового редактора внутри DSM-решения, с подсветкой синтаксиса, автодополнением и прочей функциональностью, свойственной современным средам разработки.

Основное достоинство получившегося решения состоит в том, что от пользователя скрыта вся объектно-ориентированная составляющая программирования под Ubiq Mobile, все объявления классов генерируются автоматически, генерируются необходимые отношения наследования, поля и заголовки методов. Фактически всё, что пользователь рисует на диаграммах, укладывается в парадигму структурного программирования и в знания, входящие в программу средней школы. Как кажется, это весьма важный результат, потому как технология уже существенно снизила порог вхождения, и дальнейшее развитие в этом направлении могло бы путём последовательного исключения необходимых знаний сделать программирование под Ubiq Mobile доступным существенно более широкому кругу людей, чем ранее. А именно это и является одной из основных целей создания предметно-ориентированного решения в DSM-подходе.

#### **А.2.4 Дальнейшее развитие QReal:Ubiq**

Работы по второму этапу создания технологии программирования под платформу Ubiq Mobile велись в направлении создания полноценной самодостаточной технологии, включающей в себя в том числе возможность задания пользовательского интерфейса мобильного приложения, и исправления недостатков первой версии языка, описанных выше. Работа велась под руководством автора

данной диссертации в рамках курсовой работы Дерипаска Анны Олеговны [82], здесь будет приведено только краткое изложение основных результатов.

Сам по себе интересен анализ существующих подходов к созданию мобильных приложений и визуальному заданию логики работы программы, проведённый в курсовой работе. Большинство существующих решений для создания мобильных приложений либо не позволяет задавать сложную логику вообще, ограничиваясь правилами перехода между экранами, либо позволяет задавать её в текстовом виде. Интересной альтернативой является язык Scratch и ряд специализированных технологий на его основе: программа имеет текстовый вид, но собирается из набора визуальных блоков, каждый из которых соответствует текстовому оператору. Структура программы полностью соответствует программе на текстовом языке, поэтому выразительная сила такого подхода всё же ниже, чем при использовании „настоящих“ визуальных языков, зато требуется меньше знаний (мы не вспоминаем синтаксис языка, а выбираем готовые блоки из палитры) и меньше вероятность ошибиться (блоки неправильных типов невозможно соединить друг с другом).

В результате анализа существующих решений и опыта использования прототипа языка было принято решение полностью отказаться от текстового программирования на диаграммах и реализовать две схемы.

1. Язык с настраиваемыми элементами, реализующими крупный блок функциональности, например, расстановка кораблей на поле в игре „Морской бой“. Семантика таких элементов реализуется для каждой конкретной задачи на языке C#.
2. Вынести все элементарные конструкции на уровень визуального языка, существенно сузив при этом класс решаемых задач (иначе в таком случае получился бы визуальный язык общего назначения, что не соответствовало нашим целям).

Первая схема была реализована только в виде прототипа, поскольку, несмотря на то, что она существенно более гибкая, требует возможностей, которыми QReal на данный момент не обладает — чтобы описывать семантику настраиваемого элемента, надо уметь изменять генератор кода прямо во время создания модели, иначе обеспечить интеграцию сгенерированного и рукописного кода

очень сложно. Полноценно была реализована вторая схема, в качестве предметной области были выбраны игры с доской, такие как „Крестики-нолики“, „Морской бой“, „Шашки“ и т.д. Блоки языка включали в себя объявления и изменения значений переменных и списков, команды отрисовки, условные операторы, подпрограммы. Также был реализован язык, позволяющий описывать пользовательский интерфейс генерируемого приложения и правила переходов между экранами, пример такой диаграммы показан на рисунке A.8.

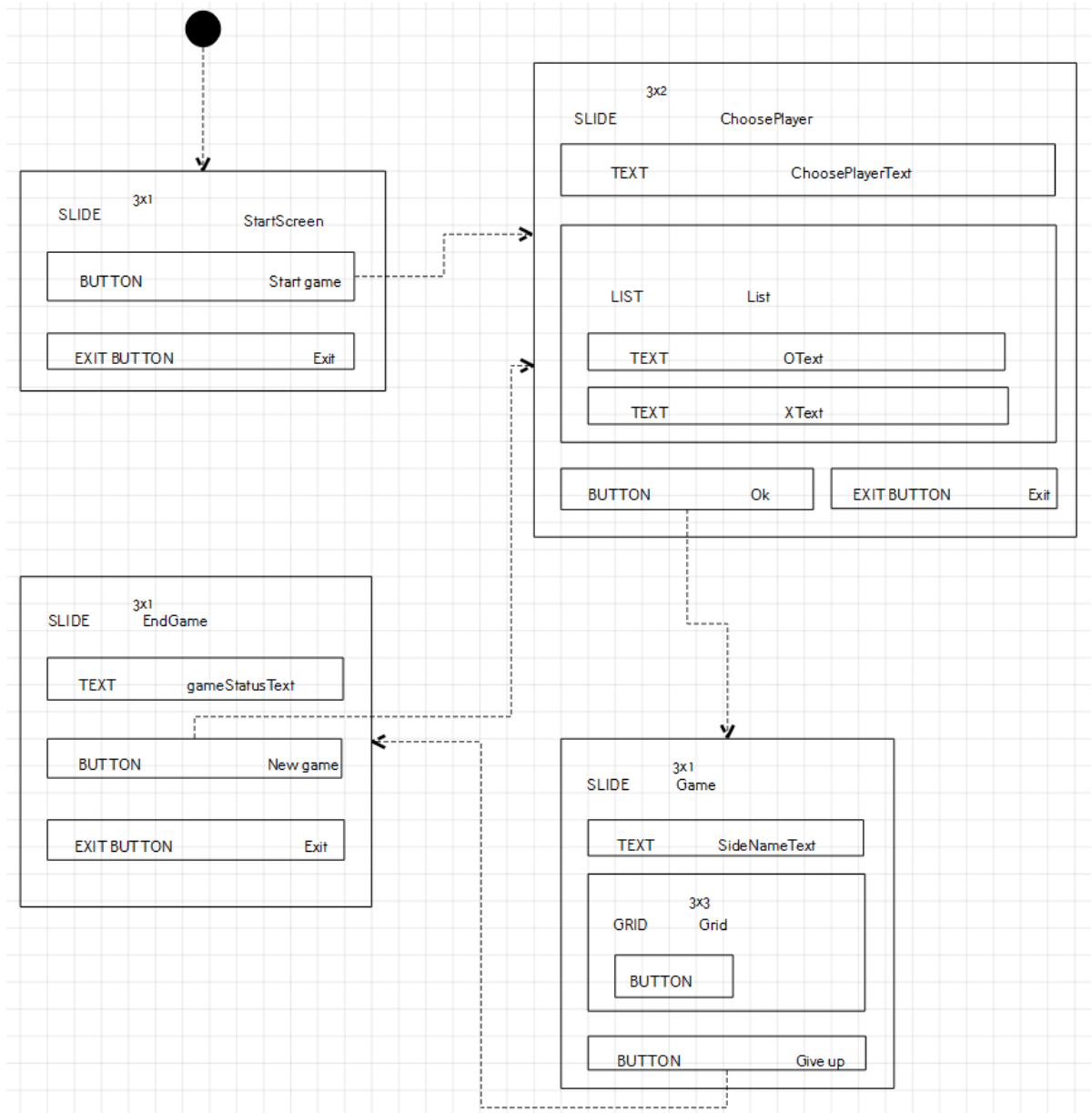


Рисунок A.8: Задание интерфейса приложения и переходов между экранами в QReal:Ubiq.

Результатом реализации модельного примера (в качестве которого была выбрана игра „Крестики-нолики“) стало то, что диаграммы переходов между формами оказались полезны сами по себе, поскольку хорошо визуализируют переходы, которые могут быть неочевидны в коде. Кроме того, по ним просто сгенерировать скелет приложения, а если сложная логика не требуется (например, приложение-визитка), то и приложение целиком. Однако задание сложной логики без кода на целевом языке привело к ещё более громоздким диаграммам, которые хоть и проще для понимания, чем диаграммы из прототипа, полученного на первом этапе работы, но всё же менее удобны, чем ручное кодирование.

### А.2.5 Результаты проекта QReal:Ubiq

Первый этап проекта разрабатывался для мастер-класса в рамках конференции FRUCT 2011, где был успешно продемонстрирован, а на самой конференции был сделан доклад [21]. На мастер-классе демонстрировалась разработка приложения для отображения данных с веб-камеры, аудитории был показан процесс создания и генерации логики клиентской части. Технология показалась весьма привлекательной авторам Ubiq Mobile, поэтому работа над ней была продолжена и после выступления.

С точки зрения исследования визуальных языков результаты проекта оказались менее однозначными: конечного ответа на вопрос „возможно ли эффективно использовать визуальные языки вместо текстовых при решении достаточно общих задач“ получено не было. С одной стороны, визуальная технология дала очевидный выигрыш в том, что уменьшила объём необходимых для программирования знаний, с другой стороны, для некоторых задач, возникающих при разработке мобильного приложения, использование текстовых языков пока эффективнее.

Работа над этим проектом поставила новые вопросы, требующие исследования.

1. Полноценная реализация редактора пользовательского интерфейса с визуализацией переходов между формами (или экранами) средствами DSM-платформы. Для этого различные элементы пользовательского интерфей-

са, такие как поля ввода, кнопки, выпадающие списки и т.д. должны быть полноправными частями формы элемента языка.

2. Поддержка работы с семействами визуальных языков, каждый из которых может уточнять более общий язык под конкретную задачу. Таким образом может быть реализована схема с настраиваемыми элементами из раздела A.2.4 — создать общий язык разработки мобильных приложений, от него породить язык задания игр с полем, от него — язык, содержащий специфические блоки для игры „Морской бой“. Это не снимает всех проблем, перечисленных в разделе A.2.4 касательно этой схемы, но представляется логичным развитием идей DSM-подхода.

## A.3 Среда разработки аппаратуры QReal:HaSCoL

Технология QReal:HaSCoL была исторически первой технологией, разработанной автором на платформе QReal, и именно опыт её разработки во многом определил направления дальнейших исследований и, в итоге, полученные в данной диссертации результаты. В частности, в ходе этой работы возникла потребность в визуальном метаредакторе. Данная технология не использует многих описанных в этой работе возможностей платформы QReal, поскольку на момент работы над ней эти возможности ещё не были реализованы, кроме того, сведения, приведённые в обзоре аналогичных подходов, могли устареть. Это не представляется критичным недостатком, поскольку основная цель дальнейшего изложения — продемонстрировать пример разработки предметно-ориентированного визуального языка, причём для данной работы более важны методологические аспекты, чем специфика предметной области или особенности реализации конкретной DSM-платформы. В этом плане предлагаемый пример интересен тем, что язык создавался и был строго формализован как расширение метамодели языка UML 2.0, поэтому данный опыт может быть использован для сравнения с более „легковесными“ подходами, применявшимися в примерах из разделов A.1 и A.2.

### А.3.1 Постановка задачи

Различные встроенные устройства играют большую роль в нашей жизни, однако задача их разработки всё ещё остаётся сложной и трудоёмкой. В 80-х годах двадцатого века появились языки Verilog и VHDL, которые и по сей день остаются фактическим стандартом в деле разработки аппаратного обеспечения. Эти языки позволяют описать устройство аналогично коду обычной программы, их синтаксис похож на синтаксис традиционных языков программирования. Они позволяют синтезировать описание разрабатываемого аппаратного обеспечения, пригодное для производства, или произвести эмуляцию.

Тем не менее, данные языки заставляют описывать систему в низкоуровневых терминах, поэтому актуальна задача разработки новых технологий с более высоким уровнем абстракции. Визуальные методы разработки в данной предметной области даже более применимы, чем в области разработки ПО, поскольку аппаратное обеспечение традиционно описывалось различными чертежами и схемами. Однако же, в силу высокой (и постоянно растущей) сложности аппаратного обеспечения, средство визуального моделирования не может быть просто редактором электронных схем, а должно само поддерживать высокоуровневые концепции. Ещё одним важным требованием, накладываемым на такое средство, является исполнимость модели. Средство должно иметь возможность синтезировать описание устройства в виде, пригодном для использования промышленным оборудованием при производстве, и генерировать эмулятор устройства, позволяющий вести отладку, тестирование и оценку производительности, не прибегая к созданию дорогостоящего прототипа. Все эти сложности затрудняют разработку новых визуальных технологий.

На кафедре системного программирования математико-механического факультета СПбГУ разрабатывается текстовый язык разработки аппаратных систем HaSCoL (Hardware-Software Codesign Language) гораздо более высокого уровня, чем VHDL или Verilog. Перед автором данной диссертации была поставлена задача разработать визуальную технологию, которая бы использовала язык HaSCoL как целевой язык для генерации и позволяла бы проектировать аппаратуру в высокоуровневых терминах этого языка. Это дало бы возможность использовать инструментальные средства HaSCoL для создания как спе-



цификации устройства для конфигурации FPGA или производства, так и для генерации программного эмулятора. С другой стороны, визуальная технология должна была повысить удобство программирования на языке HaSCoL, а поскольку этот язык новый, это могло бы позитивно сказаться на эффективности его внедрения. Кроме того, использование высокоуровневого языка в качестве целевого для генерации свело бы к минимуму затраты на анализ предметной области при разработке DSM-решения, поскольку большая часть этой работы уже была выполнена при разработке текстового языка.

Язык HaSCoL описывает систему в терминах исполняемых параллельно обработчиков сигналов. Обработчики объединены в процессы — сущности, инкапсулирующие в себе ресурсы (данные), обработчики сигналов и другие процессы, и имеющие входы и выходы. Обработчик представляет из себя последовательность однократных шагов, выполняющихся, если некоторые условия (получение сообщения, состояние ресурсов процесса) истинны. Обработчики могут начинать своё исполнение на каждом такте, на котором выполнены условия его старта, вне зависимости от того, выполняются они уже или нет. Пример описания процесса:

```

process DynamicArbiter =
begin
  in one(uint(2), uint(8));
  in two(uint(2), uint(8));
  out res(uint(8));
  group {
    -- что делать, если пришли оба сигнала:
    -- условия на принимаемые сигналы определяют готовность
    -- принятия данных из каждого входа в отдельности
    one(prio1, data1) and prio1 >= prio2,
    two(prio2, data2) and prio1 < prio2
    {
      send res (if prio1 >= prio2 then data1 else data2 fi)
    }
    -- если пришел только один сигнал --- случай попроще
    -- подчеркивание вместо имени параметра означает, что
    -- нас данный параметр не интересует и пользоваться им
    -- мы не будем
    one(_, ddata) {send res (ddata)}
    two(_, ddata) {send res (ddata)}
  }
end

```

Процесс имеет два входа (one, two) с двумя параметрами и выход res, возвращающий значение типа uint(8). Конструкцией group три обработчика объединены в группу, перед фигурными скобками записано условие, при истинности которого обработчик начнёт работу, внутри фигурных скобок — тело обработчика. В условиях могут участвовать проверки наличия данных на входах, логические выражения с участием входных данных и локальных данных процесса. Конструкция send в теле обработчика — посылка сигнала на указанный выход. Входы и выходы могут также иметь несколько портов, каждый порт может быть связан с очередью сообщений. Например,

```
in InputData (int(16))[A[1], B];
```

определяет вход с одним параметром и двумя портами, при этом размер очереди порта  $A - 1$ , размер очереди порта  $B - 0$ .

Для поддержки структурной декомпозиции процессов в язык был введён структурный уровень. Структурные конструкции позволяют отображать входы объемлющего процесса на порты входов вложенного, и выходы вложенного процесса на выходы объемлющего или входы вложенного, позволяя таким образом структурно декомпозировать процессы на наборы взаимодействующих вложенных подпроцессов.

Каждый процесс имеет свой тип, задаваемый явно или неявно. Тип процесса — перечень его входов и выходов с указанием типов их аргументов. Над типами процессов определено отношение структурного сабтайпинга — неформально говоря, процесс  $A$  имеет тип, являющийся подтипом типа процесса  $B$ , если  $A$  можно везде использовать вместо  $B$ . Тип процесса по умолчанию получается из спецификации процесса, тип можно указать и явно, это используется, например, для описания свойств параметров функторов.

Функтор — это процесс, параметризованный другим процессом. Например,

```
process Wrapper (P: Proc) =
begin
  process X = P;
  process Y = P;
  ...
end
```

Процесс `Wrapper` параметризован процессом  $P$  типа `Proc`. Применение функтора выглядит так:

```
process W = Wrapper(aProc);
```

### **А.3.2 Существующие средства визуального описания аппаратных систем**

Самый известный из существующих визуальных языков, UML 2, создавался с учётом необходимости описывать аппаратные средства. В версиях UML 1.x специальных средств для описания аппаратных систем не было, поэтому

приходилось создавать профили UML, например, UML-RT. В UML 2 этот пробел был заполнен, в язык было введено понятие „структурированный классификатор“, который может содержать внутри себя набор частей, соединённых между собой соединителями. Взаимодействие структурированных классификаторов с внешним миром и внутренними частями происходит исключительно через порты. Порт — это точка взаимодействия со строго определённым интерфейсом. Один структурированный классификатор может иметь несколько портов, а также имеет возможность определить, на какой из портов пришёл запрос. Запросы, приходящие на порты, могут быть обработаны непосредственно объектом-хозяином порта или переданы на порт какой-либо его части. Каждый порт имеет набор предоставляемых им интерфейсов и набор интерфейсов, требуемых от внешней среды. Пример нотации представлен на рисунке А.9.

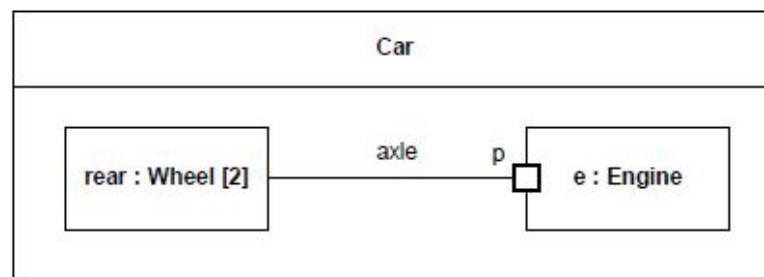


Рисунок А.9: Структурированный классификатор UML 2.

Для задания внутреннего поведения элементов системы в UML 2 обычно используется диаграмма конечных автоматов. Каждый структурированный классификатор может иметь связанный с ним конечный автомат, который описывает реакцию на события, приходящие на его порты.

Помимо „чистого“ UML 2 используются и специальные профили UML, предназначенные для использования с текстовыми языками (подход, весьма близкий к предлагаемому). Пример такого профиля — профиль UML для языка SystemC. SystemC — язык проектирования уровня системы, основанный на C++ и поддерживаемый группой крупных компаний. Система в SystemC состоит из модулей, каждый модуль может содержать переменные, порты для взаимодействия с окружением и процессы, реализующие функциональность модуля. Процессы исполняются параллельно и могут реагировать на события. Связь между модулями осуществляется с помощью портов, которые предоставляют

модулям доступ к каналам. Каналы бывают примитивными и иерархическими: иерархический канал — тоже модуль, имеет свои процессы и может иметь доступ к другим каналам.

Профиль UML для SystemC логически разделён на 3 части.

1. Структуры и коммуникации — определяет стереотипы для базовых строительных блоков SystemC. Эти стереотипы представляют модули, порты, интерфейсы и каналы, и используются на различных диаграммах UML, например, на диаграммах классов и диаграммах композитных структур.
2. Поведение и синхронизация — определяет стереотипы для спецификации поведения процессов SystemC. Эти стереотипы используются в диаграммах состояний UML.
3. Типы данных — представляет типы данных SystemC.

Пример нотации профиля UML для SystemC представлен на рисунке A.10.

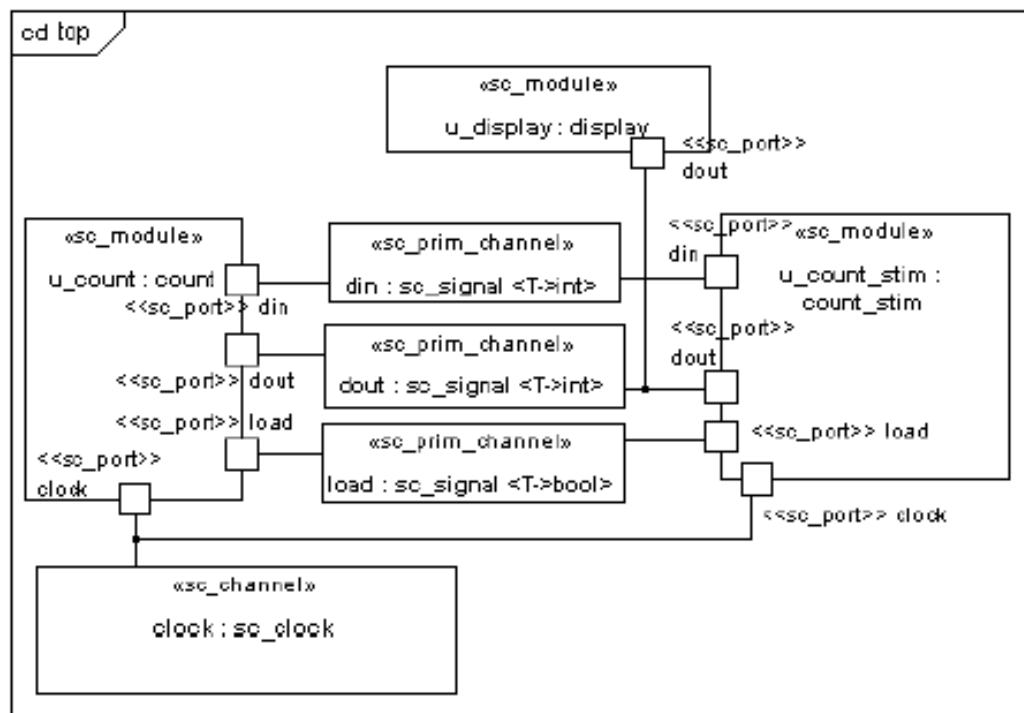


Рисунок A.10: Пример нотации профиля для SystemC.

Как видно из обзора, идея визуального моделирования аппаратного обеспечения не нова, при этом наибольшее внимание уделяется языку UML (по-видимому, как наиболее распространённому визуальному языку). При этом,

существующие языки пытаются целиком специфицировать систему графическими средствами, что зачастую приводит к весьма громоздким диаграммам.

### **А.3.3 Предлагаемый набор визуальных языков**

Основные принципы, в соответствии с которыми разрабатывалась нотация, таковы.

1. Модель должна быть исполняемой, то есть позволять синтезировать код на языке системы CoolKit без необходимости ручной корректировки результатов. Пригодная для промышленного использования технология должна позволять программисту получать готовую низкоуровневую спецификацию системы или эмулятор, действуя в одной среде разработки и, желательно, с одним представлением системы.
2. Графическими примитивами должны быть представлены только те элементы программы, которые наиболее выгодно представлять графически. Остальная необходимая для синтеза информация должна быть представлена на визуальной модели в текстовой форме. Такой подход позволяет сохранять диаграммы достаточно компактными, но при этом, как обсуждалось в разделе [А.2](#), делает программы существенно более сложными для понимания.
3. Нотация разрабатывалась для использования внутри среды разработки, поэтому некоторые её элементы могут быть неудобны для представления на бумаге.

Для визуализации языка CoolKit плохо подходит непосредственно UML и даже профиль для UML — целевой язык не является строго говоря объектно-ориентированным, к тому же обладает рядом особенностей, которые плохо или неудобно выражаются в терминах UML. Предлагаемый графический язык, хотя и не является профилем UML, сформулирован с активным использованием метамодели UML. Для формализации языка используется метамоделирование на метаязыке MOF — синтаксис графических конструкций языка описан с помощью диаграмм UML. Метамодель языка построена на метамодели ядра и

некоторых диаграмм UML, таким образом, язык лишь незначительно отличается от UML, и его синтаксис и семантика интуитивно понятны специалистам, занимающимся визуальным моделированием. Кроме того, переиспользование метамодели UML позволило сэкономить массу усилий при формализации языка. Одним из полученных результатов стало понимание того, что формализация предметно-ориентированных языков программирования может быть существенно упрощена благодаря переиспользованию некоей стандартной метамодели, например, UML.

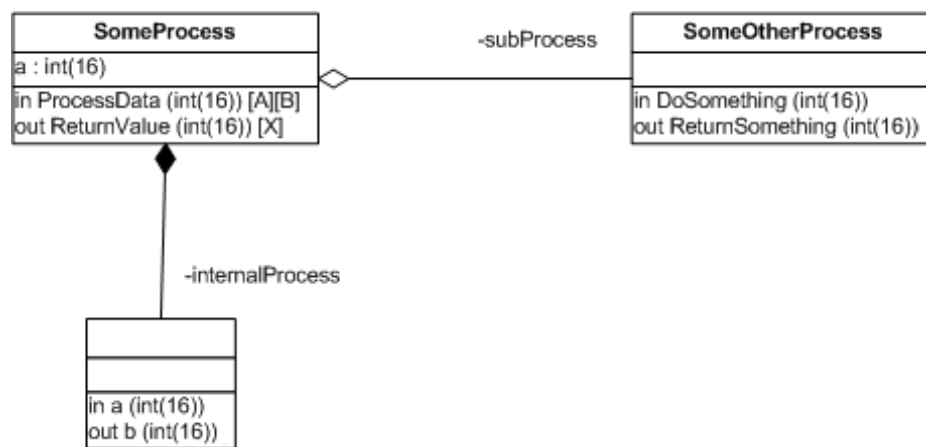
Для спецификации системы в нашем языке используется два вида диаграмм — диаграмма типов процессов и диаграмма отображения портов. Заметим, что мы избежали необходимости использовать диаграммы автоматов для описания поведения системы — эту роль выполняют текстовые блоки на целевом языке. Диаграмма типов процессов базируется на диаграмме классов UML, а диаграмма отображения портов — на диаграмме композитных структур.

### **Диаграмма типов процессов**

Диаграмма типов процессов используется для задания основных структурных свойств и отношений процессов, составляющих пакет или приложение. На диаграмме изображаются сами процессы, их входы и выходы, отношения вложенности, отношения генерализации, и функторы. Заметим, что на этой диаграмме не рисуются обработчики событий, и могут не рисоваться ресурсы процесса (список ресурсов процесса может быть неполным, из того, что ресурс не изображён на диаграмме, нельзя делать вывод, что он не описан в процессе). Некоторые вложенные процессы тоже могут быть опущены на этой диаграмме, если это не повлияет на корректность модели. Часть метамодели диаграммы представлена на рисунке [A.11](#).

Пример изображения вложенных процессов представлен на рисунке [A.12](#). Как видно из примера, нотация перечисляет ресурсы, входы и выходы процессов в таком виде, в каком они были в языке системы CoolKit. По-настоящему графически здесь изображаются только отношения использования одного процесса внутри другого или вложенности процессов (т.е. когда один процесс описан непосредственно внутри другого). Заметим, что вложенный процесс может не иметь имени — тогда оно просто не отображается на диаграмме. Поскольку

вложенные или используемые процессы являются деталями реализации объемлющего процесса, они могут не рисоваться на диаграмме.



Существенную выгоду от использования этого типа диаграммы можно получить при использовании в программе функторов. Пример нотации объявления функторов приведён на рисунке [A.13](#).

В частности, из-за особенности нотации, которую можно видеть на примере, было принято решение не использовать диаграммы классов UML. Нотация отражает возможность языка системы CoolKit описывать тип процесса — формальный параметр функтора прямо в месте описания формального парамет-



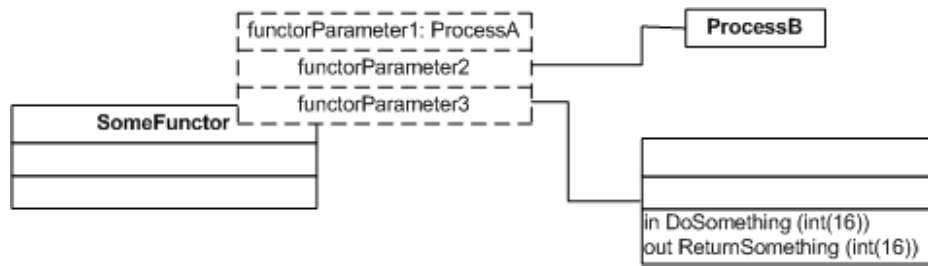


Рисунок А.13: Нотация функторов.

ра. Вынесение типов параметров в отдельные графические сущности позволяет сделать отношения между процессами — формальными или фактическими параметрами функторов гораздо более наглядными. Применение функтора изображается так, как показано на рисунке А.14.

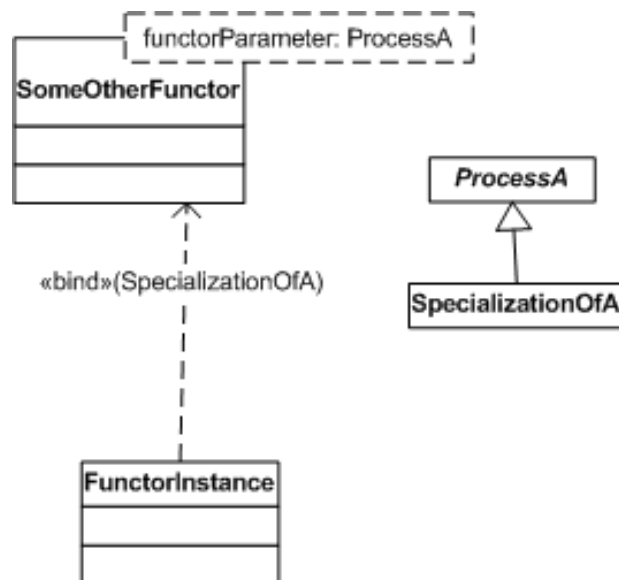


Рисунок А.14: Применение функтора.

Для того, чтобы проиллюстрировать применение диаграммы типов процессов, рассмотрим содержательный пример — задачу об арбитре динамических приоритетов 4 в 1. Задача формулируется следующим образом: на один из четырёх входов поступают данные, первый параметр пришедших данных — приоритет. Если в одном такте данные поступили на несколько входов, на выход выдаются данные с наибольшим приоритетом, остальные входы объявляются неготовыми. Если приходит только одно сообщение, оно отправляется на выход.

Следуя оригинальному решению поступим следующим образом — сначала напомним арбитр динамических приоритетов 2 в 1 (с двумя входами и одним выходом), затем создадим арбитр-функтор 4 в 1, использующий 3 арбитра 2 в 1, который и решит задачу (см. рисунок A.15).

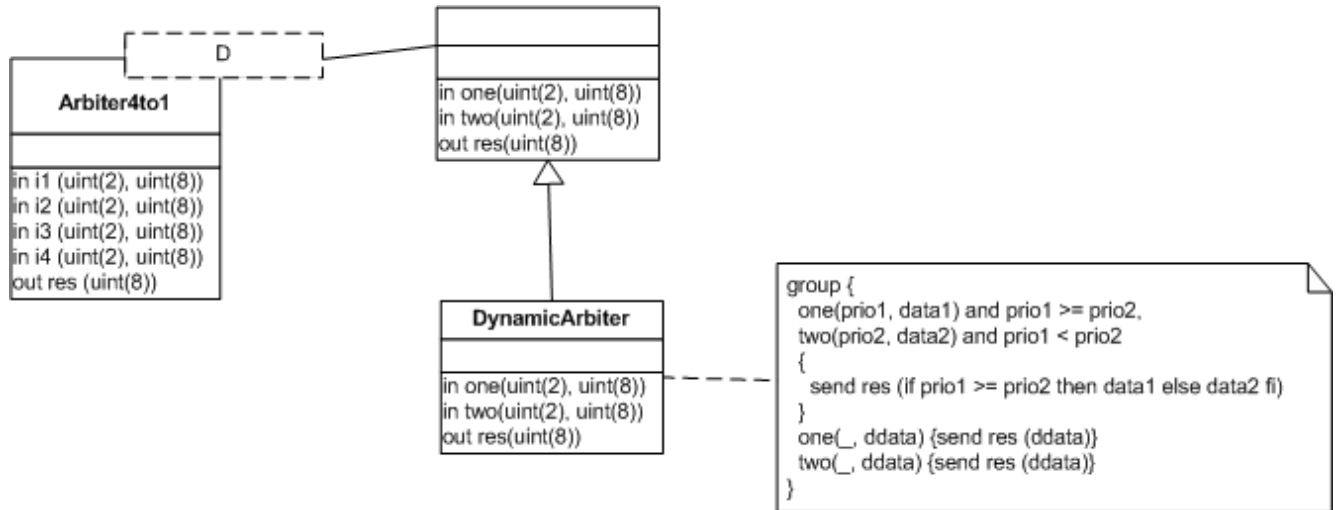


Рисунок A.15: Диаграмма типов процессов для задачи „Арбитр 4 в 1“.

В комментарии — код процесса DynamicArbiter на языке системы CoolKit, описывающий реализацию арбитра 2 к 1. Код из комментария при синтезе добавляется к сгенерированному описанию процесса, таким образом мы получаем полностью специфицированный арбитр 2 к 1, который может быть использован как фактический параметр функтора Arbiter4to1 (на это указывает отношение генерализации, связывающее DynamicArbiter и безымянный тип процесса — формальный параметр функтора). Про процесс Arbiter4to1 мы указали пока только то, что он является функтором (готов использовать любой процесс, поддерживающий интерфейс арбитра 2 к 1, который мы определили с помощью безымянного типа процесса), и указали, что у него есть 4 входа и один выход — то есть просто "нарисовали" условие задачи.

### Диаграмма отображения портов

Диаграмма отображения портов рисуется для отдельного процесса и используется для отображения связей между входными и выходными портами вложенных процессов того процесса, для которого рисовалась диаграмма. Таким образом, эта диаграмма фактически является графической нотацией для

структурного уровня языка системы CoolKit. Синтаксис с незначительными изменениями совпадает с синтаксисом диаграммы составных структур UML. На диаграмме изображаются процессы и их порты (все порты входов и выходов), связи между портами, внутри процессов могут находиться вложенные процессы. Важно различие между процессом верхнего уровня — процессом, относительно которого рисуется диаграмма, и вложенными процессами. Вложенные процессы на самом деле являются экземплярами процессов, имеют имя и тип. Процесс верхнего уровня имеет только тип. Пример нотации изображён на рисунке A.16. Вернёмся к нашему примеру с арбитром 4 к 1. Диаграмма отображения портов для этого примера выглядит так, как показано на рисунке A.17.

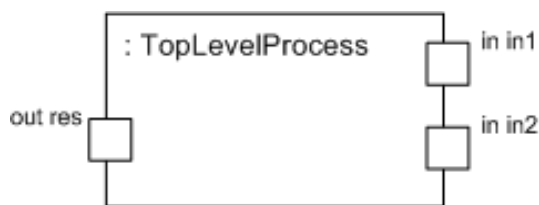


Рисунок A.16: Процесс на диаграмме отображения портов.

Здесь мы видим процесс верхнего уровня `Arbiter4to1` — без имени, но с типом и с формальным параметром функтора. `Arbiter4to1` имеет 4 входных порта и 1 выходной, их типы здесь уже можно не указывать, они были на диаграмме типов процессов, которую мы нарисовали ранее. Процесс имеет три вложенных процесса `A12`, `A34` и `A1234` типа `D`, который, как следует из диаграммы типов процессов, описывает процессы, имеющие два входа и один выход. Если мы считаем, что в качестве параметра функтора передаётся арбитр 2 к 1, изображённое на рисунке соединение портов решает задачу.

## Генерация

Генерация в язык системы CoolKit проходит довольно очевидным образом, поскольку графический и текстовый языки имеют одинаковую модель представления аппаратного устройства. Скелеты описаний процессов генерируются по диаграмме типов процессов, потом они дополняются вложенными процессами и конструкциями, описывающими связь портов по диаграмме отображения

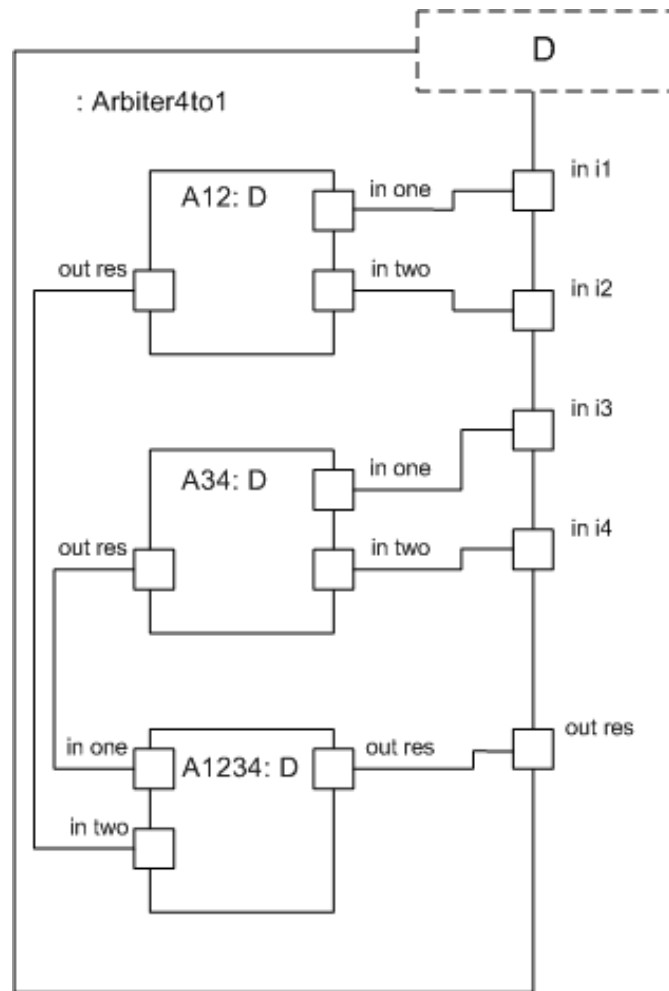


Рисунок А.17: Диаграмма отображения портов для задачи „Арбитр 4 в 1“.

портов, потом они дополняются реализацией, которая так или иначе представлена в текстовом виде на диаграмме — в виде комментария или в виде неотображаемого атрибута процесса. В результате получается полная программа на языке системы CoolKit, готовая к дальнейшей трансляции в VHDL и далее — в эмулятор или спецификацию устройства.

При генерации считается, что одноимённые однотипные сущности в одном пространстве имён представляют собой одну сущность. То есть, например, если на одной диаграмме описан процесс с одним входом, а на другой диаграмме описан процесс с тем же именем и другим входом, будет сгенерирован один процесс с двумя входами. Такой подход позволяет изображать на каждой диаграмме только существенные для неё части системы, хотя и может привести к некоторой сложности для понимания набора диаграмм в целом. Предполагается, что у средства визуального моделирования существует возможность удобным

способом предоставить пользователю информацию о невидимых на диаграмме элементах. Есть некоторые тонкости с тем, что разные формально сущности на разных диаграммах представляют собой одну логическую сущность, например, процесс на диаграмме типов процессов и процесс на диаграмме отображения портов. Такие сущности всё же должны считаться генерацией однотипными и объединяться в одну сущность. Например, двух ранее приведённых диаграмм, описывающих реализацию арбитра 4 к 1, достаточно, чтобы сгенерировать такой код на языке системы CoolKit:

```

process DynamicArbiter =
begin
  in one(uint(2), uint(8));
  in two(uint(2), uint(8));
  out res(uint(8));
  group {
    one(prio1, data1) and prio1 >= prio2,
    two(prio2, data2) and prio1 < prio2
    {
      send res (if prio1 >= prio2 then data1 else data2 fi)
    }
    one(_, ddata) {send res (ddata)}
    two(_, ddata) {send res (ddata)}
  }
end

process Arbiter4to1 (D : begin
  in one(uint(2), uint(8));
  in two(uint(2), uint(8));
  out res(uint(8));
  end
) =
begin
  in i1 (uint(2), uint(8));
  in i2 (uint(2), uint(8));
  in i3 (uint(2), uint(8));
  in i4 (uint(2), uint(8));
  out res (uint(8));
  process A12 = D with one = i1, two = i2, res = A1234.one;
  process A34 = D with one = i3, two = i4, res = A1234.two;
  process A1234 = D with res = res;
end

```

Для того, чтобы получить исполнимый код, нужно создать экземпляр процесса `Arbiter4to1`, использующий в качестве параметра процесс `DynamicArbiter`. Для этого достаточно нарисовать диаграмму, изображённую на рисунке A.18.

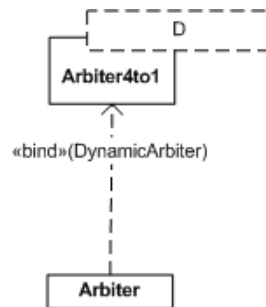


Рисунок A.18: Применение функтора „Арбитр 4 к 1“.

Эта диаграмма породит код

```
process Arbiter = Arbiter4to1(DynamicArbiter);
```

Этот код вместе с приведённым ранее кодом может быть синтезирован и исполнен на эмуляторе.

### A.3.4 Результаты проекта QReal:HaSCoL

В результате работы над технологией QReal:HaSCoL было создано два языка, с помощью которых описывалась структурная часть системы. Технология не была доведена до промышленного использования, полученные результаты нигде не публиковались, связано это с недостаточной зрелостью базовой технологии QReal на момент работы над QReal:HaSCoL. Однако проблемы, возникшие при разработке, во многом послужили мотивацией для реализации тех возможностей метатехнологии QReal, которые представляются в данной работе. Кроме того, визуальный язык QReal:HaSCoL далее использовался как пример для апробации различных новых возможностей QReal, например, „метамоделирования на лету“ [74, 83].

С точки зрения введённой в данной работе классификации созданные языки являются статическими текстографическими языками. В данном случае использование текстовой нотации не так осложняет понимание диаграмм, как в случае QReal:Ubiq (раздел A.2), поскольку текстовая и графическая части не

взаимозаменяемы: структурная часть системы целиком описывается графически, поведенческая — целиком текстово. Были предприняты попытки использовать диаграммы активностей языка UML 2.0 для задания и поведенческой части в графическом виде, но в результате получались громоздкие и трудночитаемые диаграммы, поэтому было принято решение поведенческую часть задавать в текстовом виде.

В отличие от всех последующих языков на базе платформы QReal, языки QReal:HaSCoL создавались на базе метамодели UML 2.0. Это было сделано для того, чтобы строго формализовать язык и сделать возможной его реализацию не только на базе QReal (впрочем, таких попыток не предпринималось), а также опробовать на реальном примере такой подход к разработке синтаксиса языка. Результаты показали, что переиспользование метамодели UML действительно позволяет сэкономить много усилий (потребовалось ввести всего две-три новые сущности на каждый язык из семейства), однако от разработчика требуется хорошо ориентироваться в метамодели UML, что само по себе весьма непросто. Стандарт UML достаточно объёмен, а метамодель весьма запутанна (особую сложность для читаемости метамодели представляет активное использование операций объединения пакетов `PackageMerge` и использование сущностей с одинаковыми именами в разных пакетах). Как кажется, затраты на изучение метамодели UML превышают выгоду, получаемую от переиспользования этой метамодели, так что разрабатывать новый предметно-ориентированный язык „с нуля“ с использованием метаязыка выбранной DSM-платформы более оправданно, чем перед созданием языка специально изучать метамодель UML. Наличие эффективных инструментальных средств, помогающих разобраться в больших метамоделях наподобие UML и автоматизировать переиспользование фрагментов метамодели, как кажется, смогло бы изменить ситуацию, однако на данный момент такие средства не распространены, и создание таких средств представляется интересным направлением дальнейших исследований.



## Приложение В

### Описание метаязыка QReal

Таблица В.1: Метаязык системы QReal


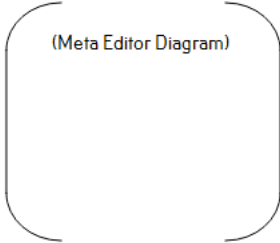
Название	Внешний вид	Описание
Metamodel Diagram		Корневой элемент метамодели. Может содержать в себе узлы Meta Editor Diagram (метамодели языков, входящих в плагин), свойства этого элемента содержат настройки сборки редактора, такие как относительный путь к исходным файлам QReal.
Meta Editor Diagram		Корневой узел метамодели одного визуального языка. Может содержать в себе узлы Node, Edge, Enum. Для каждого узла этого типа в метамодели редактора будет сгенерирована своя вкладка в палитре элементов.

Таблица В.1: Метаязык системы QReal (многостраничная)

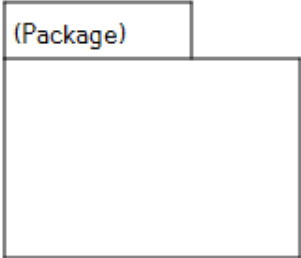
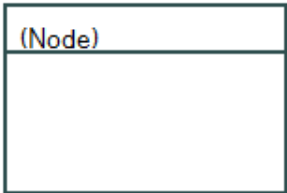
Название	Внешний вид	Описание
Package Diagram		<p>Пакет. Предназначен для группировки метамodelей и обеспечения переиспользования фрагментов метамodelей. Также может быть корневым элементом метамodelи. Может содержать в себе другие пакеты или узлы Metamodel Diagram. На данный момент функциональность этого узла поддерживается лишь частично и используется в основном для визуализации зависимостей по включению между метамodelями при импорте их из XML-описаний.</p>
Node		<p>Представляет узел создаваемого языка. Может содержать в себе узлы Property, Possible Edge, Properties as Container. Свойства включают в себя имя узла, отображаемое имя узла (с первым работают генераторы и другие инструменты, отображаемое имя показывается пользователю), форму узла (форма хранится в виде XML-строки, для её редактирования из метаредактора открывается редактор формы фигур), некоторые свойства, связанные с пользовательским интерфейсом (текст всплывающей подсказки для этого узла, может ли узел менять размер, можно ли создавать вложенные в него узлы с помощью контекстного меню).</p>

Таблица В.1: Метаязык системы QReal (многостраничная)

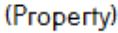

Название	Внешний вид	Описание
Property		Свойство. Не имеет какого-либо специального графического оформления, может находиться только внутри узлов Node и Edge. Для свойства возможно указание имени, отображаемого имени, типа, значения по умолчанию. Типом свойства может быть один из элементарных типов (целочисленный, булевый, строковый), тип-перечисление, определённый в этой же метамодели, или ссылочный тип, указывающий на объект типа, определённого в этой же метамодели.
Possible Edge		Возможная связь. Указывает, какие элементы могут быть соединены связью. Задаются имя узла, из которого может начинаться связь и имя узла, в котором связь может заканчиваться, также есть возможность указать, направленная связь или нет (то есть можно или нет поменять начальный и конечный узлы местами). Может находиться внутри узла Node.

Таблица В.1: Метаязык системы QReal (многостраничная)


Название	Внешний вид	Описание
Properties as Container		<p>Свойства контейнера. Этот элемент отвечает за задание чисто визуальных свойств взаимодействия узла с вложенными в него узлами, например, должен ли узел автоматически располагать вложенные узлы друг под другом в столбик, или пользователь может произвольно располагать вложенные узлы внутри узла-контейнера. Также можно задать расстояние между вложенными элементами и расстояние от элементов до границы контейнера, если контейнер сам отвечает за расположение вложенных узлов. Можно указать, должен ли контейнер автоматически сжиматься или расширяться в соответствии со своим содержимым.</p>

Таблица В.1: Метаязык системы QReal (многостраничная)

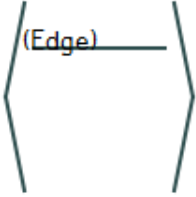
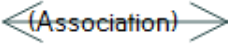
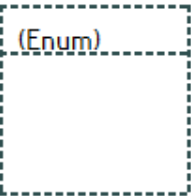

Название	Внешний вид	Описание
Edge		Связь. Задаётся имя, отображаемое имя связи, тип линии (сплошная, пунктирная и т.д.), текст, отображаемый на линии (либо фиксированный, либо свойство связи из репозитория). Кроме того, связь может иметь свойства, так же как узел. Также можно указать, из какого типа портов связь может исходить и в какой тип портов связь может входить (см. описание элемента Port). Если типы портов не указаны, связь может быть подключена к любому порту, если он относится к допустимому узлу (см. описание элемента Possible Edge).
Association		Элемент для задания свойств начала и конца связи, вкладывается в элемент Edge. Для конца связи можно указать имя конца и тип стрелки (открытая, закрытая, закрашенная стрелка, закрашенный и незакрашенный ромб, без стрелки).
Enum		Тип-перечисление. Позволяет указать фиксированный набор значений, который можно использовать как тип какого-либо свойства. Состоит из набора элементов Value.
Value		Одно значение типа-перечисления, вкладывается в элемент Enum. Содержит только имя значения.

Таблица В.1: Метаязык системы QReal (многостраничная)

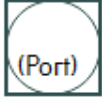


Название	Внешний вид	Описание
Port		Объявление типа порта, к которому могут быть подключены определённые связи. При редактировании формы узла имеется возможность для каждого порта, рисуемого на изображении узла, указать его тип, а для связи указать тип порта, к которому связь может быть подключена.
Import		Импорт элемента. Позволяет использовать элементы одной метамодели в другой метамодели. Имеется возможность переопределить имя элемента и его отображаемое имя.
Container		Связь, задающая допустимую вложенность между элементами. Направлена от элемента, который может вкладываться в другой, к тому элементу, в который он может вкладываться. По умолчанию элементы не вкладываются друг в друга. Связь учитывает отношение наследования, то есть любой потомок вкладываемого элемента может быть вложен в любой потомок элемента, в который исходный элемент вкладывается. Этим отношением могут быть связаны только узлы.

Таблица В.1: Метаязык системы QReal (многостраничная)



Название	Внешний вид	Описание
Explodes To		Эксплозия, или отношение раскрытия. Связывает элемент и тот элемент, в который первый элемент может раскрываться (например, узел, обозначающий подпрограмму, и диаграмму подпрограммы, в которую он раскрывается). Связь содержит ряд вспомогательных свойств, например, необходимо ли немедленно создать тот узел, на который ссылается исходный, или сделать ли связь переиспользуемой (в этом случае узел, соединённый этой связью, добавляется на пользовательскую палитру и может быть добавлен в других местах диаграммы или на другие диаграммы, с сохранением связи-раскрытия). Связь также учитывает отношение наследования.
Inheritance		Наследование. Обозначает, что один элемент является подвидом другого элемента и может быть использован везде вместо своего предка. Наследование означает, что узел-потомок имеет все свойства узла-предка, а также наследует все отношения вложенности и раскрытия, которыми был связан предок. Наследование может быть множественным.

Таблица В.1: Метаязык системы QReal (многостраничная)