САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

СРЕДСТВА ЗАДАНИЯ ИСПОЛНИМОЙ СЕМАНТИКИ ВИЗУАЛЬНЫХ ЯЗЫКОВ В СИСТЕМЕ QREAL

Поляков Владимир Александрович

Дипломная работа

Научный руководитель	..... / подпись /	ст. преп. Брыксин Т.А.
Рецензент	..... / подпись /	к.ф.-м.н., доцент Кознов Д.В.
“Допустить к защите” заведующий кафедрой,	..... / подпись /	д.ф.-м.н., проф. Терехов А.Н.

Санкт-Петербург
2013

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

TOOLS FOR DEFINING EXECUTABLE SEMANTICS OF VISUAL LANGUAGES IN QREAL IDE

by

Vladimir Polyakov

Specialist Thesis

Supervisor	. . . . .	Senior Lecturer T.A. Bryksin
Reviewer	. . . . .	Associate Professor D.V. Koznov
"Approved by" Head of Department	. . . . .	Professor A.N. Terekhov

Saint Petersburg
2013

Оглавление

Введение

В настоящее время большинство уже существующих и разрабатываемых технологий программирования основано на текстовых языках. Для таких текстовых языков есть много интегрированных сред разработки и других программных продуктов, значительно повышающих удобство и скорость создания новых программ. Каждый из этих продуктов предоставляет широкий набор различной функциональности, позволяющей разработчику программировать более эффективно. Одной из них является возможность пошаговой интерпретации и/или отладки написанного кода.

Помимо текстовых языков при разработке ПО в последнее время часто используются модельно-ориентированные средства разработки (model-based development). Этот подход предназначен для описания самой системы и ее предметной области в виде множества моделей, характеризующих её с разных сторон. При этом часто даже создаются новые специализированные предметно-ориентированные языки (DSL), хорошо подходящие под конкретные задачи, и эти задачи решаются уже с их помощью. При наличии такого нового специализированного языка скорость решения поставленных задач может существенно возрастать, модели становятся более наглядными и понятными, оказывается возможной эффективная автоматическая кодогенерация из них. Такой подход к разработке ПО называют предметно-ориентированным моделированием [4, 7, 29].

Существует множество различных визуальных сред разработки, обзор таких средств можно найти в [15, 26]. Один из способов сделать эти средства более удобными — заимствовать привычную и хорошо зарекомендовавшую себя функциональность, например, подсветку синтаксиса, автодополнение, автоматический рефакторинг и т.п., из аналогичных инструментов для текстовых языков. Распространение возможности пошаговой интерпретации и отладки на поведенческие визуальные модели является одним из шагов по достижению этой цели. Подобный инструмент позволяет разработчику искать и исправлять ошибки на ранних этапах разработки. Также визуальная интерпретация диаграмм способствует лучшему пониманию проектировщиком принципов работы разрабатываемой системы, что, в итоге, повышает её качество.

В некоторых UML-пакетах (например, в Borland Together[1]) возможность визуальной интерпретации и отладки уже присутствует. Однако данный подход было бы полезно распространить и на DSM-платформы, с помощью которых можно создавать и реализовывать новые DSL и соответствующие им DSM-пакеты. Это позволит пользователю таких систем быстро создавать интерпретаторы и отладчики для разрабатываемых им визуальных языков. Чтобы обеспечить возможность создания подобных интерпретаторов, необходимо задать формальную семантику для элементов соответствующего визуального языка.

Таким образом, цель данной дипломной работы состоит в создании средств задания исполнимой семантики визуальных языков в metaCASE-системе QReal [1] с возможностью последующей интерпретации моделей по указанной семантике.

[1] Borland Together, http://www.borland.com/products/Together

1 Постановка задачи

Целью данной работы является создание инструментальных средств задания семантики исполнения визуальных языков и интерпретации моделей в рамках этой семантики в проекте QReal. Для достижения цели были выделены следующие основные подзадачи:

1. Изучить существующие подходы к заданию семантики визуальных языков, сделать их подробный обзор, проанализировать, сравнить и оценить их применимость к системе QReal.
2. Выбрать один из существующих подходов или предложить собственное решение для системы QReal, основанное на положительных чертах данных методов.
3. Реализовать средства задания семантики выбранного класса визуальных языков, согласованные с получившимся подходом. Добавить возможность интерпретации модели в соответствии с её семантикой.
4. Выполнить тестирование реализованных средств, произвести апробацию реализованного подхода на различных реальных визуальных редакторах/языках, имеющихся в QReal.

2 Обзор литературы

Для того, чтобы обеспечить возможность исполнения или отладки поведенческих диаграмм необходимо задать формальную семантику для элементов соответствующего визуального языка. В результате исследования было выявлено несколько основных способов спецификации этой семантики, обзор которых будет дан ниже, также будут сделаны выводы относительно их применимости и возможности улучшения. Более подробный обзор данных методов задания семантики можно прочитать в статьях [9, 10].

2.1 Семантика языков

Любой новый язык является набором знаков, при помощи которых пользователь, следуя определенным правилам, может составить некий текст (осмысленную конструкцию). Традиционно любой язык раскладывается по трём измерениям: синтаксис, определяющий правила построения текстов из знаков, семантика, задающая проекции знаков на предметную область языка, и прагматика, описывающая способы использования языка пользователем.

Для языков визуального моделирования выделяют три вида синтаксиса: абстрактный, конкретный и служебный. Служебный синтаксис отвечает за представление визуальных спецификаций на данном языке в памяти. Абстрактный синтаксис определяет набор правил, при помощи которых можно объединять простые составляющие языка (знаки и конструкции) в сложные (тексты, т.е. визуальные модели). Обычно он задаётся в виде грамматики или метамодели. Конкретный синтаксис (графическая нотация или просто нотация) состоит из формализованного набора графических символов, которыми пользователь может записывать тексты, т.е. из правил изображения конструкций языка. Семантика позволяет же придавать конструкциям смысл, т.е. определяет их соответствие реальным ситуациям из предметной или из любой другой формально определённой области, называемой семантической областью. Например, семантика интерпретации используется для преобразования абстрактного представления этих конструкций в исполнимое.

В теории языков программирования выделяют четыре основных типа семантик: операционная, денотационная, аксиоматическая и трансляционная.

Денотационная семантика была описана в работе [42]. Обычно при таком подходе в качестве семантической области используются некоторые математические объекты или функции, называемые денотациями (denotation), а соответствие конструкций языка этим объектам задаётся рекурсивно, т.е. денотация для выражения должна составляться из денотаций подвыражений. Часто для создания этого соответствия используется λ-исчисление. Данный тип семантик является математически строгим и формальным, но в чистом виде порой не обладает достаточной степенью понятности для неспециалистов.

Операционная семантика[1] также является математически строгой и бывает двух типов: структурная операционная семантика (семантика малого шага, [38]) и естественная семантика (семантика большого шага, [28]). Структурная операционная семантика состоит из набора правил, при помощи которых исполнение конкретной программы на данном языке будет

[1] http://en.wikipedia.org/wiki/Operational_semantics

представлено в виде последовательности индивидуальных вычислительных шагов. После применения этих правил могут оставаться некоторые остаточные вычисления, учитывающиеся при дальнейшем представлении программы в виде набора вычислительных шагов. Естественная семантика такие остаточные вычисления запрещает и сразу определяет, каким образом по выражению будет получен конечный результат.

Аксиоматические семантики [25] представляют собой набор логических аксиом. Для них любые выражения, записанные на данном языке, рассматриваются в виде логических формул, значение которых будет формально выведено из исходного набора аксиом.

Ещё одним важным типом семантик являются трансляционные [45]. Они состоят из набора правил преобразования моделей, с помощью которых модель на исходном языке переводится в модель на другом языке, для которого уже задана исполнимая семантика. Например, часто это сводится к генерации кода на языке общего назначения, для которого уже существуют компиляторы и отладчики (Java, C++, Python и т.п.). Такой подход, с одной стороны, требует от разработчика знаний сразу в обеих областях (исходной и целевой), а с другой — является хорошо воспринимаемым, т.к. понятия абстрактной исходной предметной области переходят в конкретные конструкции исполнимого языка.

Трансляционные семантики в том числе могут применяться и просто для решения задачи трансформации моделей, т.е. для обеспечения преобразований моделей в рамках одного визуального языка (например, для осуществления рефакторинга моделей, который будет более подробно рассмотрен в разделе 6), перевода моделей с одного языка на другой (это нужно, например, для того чтобы обеспечить экспорт модели с одного языка в другой). При этом второй язык может быть текстовым, и тогда это будет обычная генерация кода. Существует несколько различных способов задания трансформации моделей, некоторые из которых основаны на графовых грамматиках [14]. Таким образом, проблема трансформации моделей является довольно актуальной по настоящее время. Один из способов её решения был реализован в средстве MetaLanguage [47].

Приведя основные общие типы задания семантики языков программирования, перейдём к рассмотрению ряда конкретных подходов, предложенных в различных работах и использующихся в программных средствах.

2.2 Двухуровневая схема отладки

Наиболее интуитивным с точки зрения реализации способом задания семантики исполнения языков является непосредственное создание интерпретатора или отладчика для конкретного предметного языка в виде отдельного модуля. Работает данный модуль по следующему принципу. По модели генерируется машинный код, который затем интерпретируется, а процесс исполнения отслеживается и определённым образом отображается пользователю. Также очевидна проекция стандартного функционала отладчиков на такую систему. Наличие машинного кода позволяет осуществлять пошаговую интерпретацию, ставить точки останова, следить за хранящимися данными.

Но бывают ситуации, когда применение данного подхода либо совершенно неэффективно, либо вообще невозможно, например, в случае языков со сложными проекциями типов данных и операторов. Такими языками в том числе являются и визуальные языки. Сложности могут

возникать как и при самой генерации машинного кода, так и при организации отслеживания хода исполнения. Поэтому Карташев в своей статье [3] предлагает реализовывать отладчики для таких языков по двухуровневой схеме, т.е. при помощи генерации кода на объектном языке и использования существующего отладчика этого языка.

2.2.1 Описание

Двухуровневая схема отладки основана на идее двухуровневой трансляции, которая состоит в том, что программа на исходном языке транслируется в программу на другом, целевом языке, для которого существуют компилятор и отладчик.

Если посмотреть на процесс отладки со стороны исходного языка, то он должен происходить в терминах этого исходного языка, т.е. положение потока исполнения в исходном коде, точки останова в определённых местах исходного кода и др. Поэтому, например, команда пошаговой отладки с остановкой на том же уровне стека вызовов (step over) должна учитывать изменение стека вызовов исходной программы, а не объектной. Сам же стек вызовов исходной программы лишь конструируется на основе стека вызовов объектной программы.

Более подробно данную проблему можно описать на примере следующим образом. По коду программы на исходном языке генерируется код на некотором объектном языке, и установка точек останова происходит в нём в терминах номеров строк кода. Каждой конструкции программы на исходном языке соответствует несколько строк кода на объектном, поэтому для того, чтобы установить точку останова в терминах исходного языка, необходимо знать, на какую из этих строк кода нужно ставить точку останова в объектном языке. Также это нужно учитывать и при простой пошаговой отладке, т.к. не всегда переход к следующему элементу потока исполнения в исходной программе будет соответствовать переходу на следующую строку кода в объектном.

Таким образом, создание отладчика исходного языка нужно производить на основе существующего отладчика объектного языка. Этот отладчик должен иметь интерфейс доступа к таким функциям, как запуск, приостановка и принудительное завершение программы, установка и снятие точек останова в терминах объектного языка, получение информации о стеках потоков вызовов в программе и др. В статье [3] приведено описание того, как данные стандартные функции могут быть реализованы в терминах исходного языка.

2.2.2 Анализ

Несмотря на понятную идею в плане реализации, данный подход требует от разработчиков новых предметных языков серьезных инженерных навыков для организации точного и полного соответствия конструкций исходного языка конструкциям целевого объектного языка и взаимодействия этих конструкций.

В связи с высокой технической сложностью задачи и тем, что данный подход ориентирован на создание единичного отладчика для определённого языка, обобщение подхода двухуровневой отладки на предметно-ориентированное моделирование выглядит нецелесообразным. Для нового языка процесс нужно будет повторять заново со всеми

техническими деталями, что не дает существенного упрощения, ускорения и автоматизации процесса создания отладчика по сравнению с ручным кодированием.

2.3 xUML

Появление унифицированного языка моделирования UML (Unified Modeling Language) [37] и его использование для объектно-ориентированного моделирования при разработке ПО способствовало сильному продвижению визуального проектирования среди разработчиков. Однако, сам по себе UML — это язык именно проектирования и документирования, а не программирования, семантика многих его элементов либо не задана явно, либо чересчур детально, но не формально описана в спецификации, благодаря чему могут случаться различные ошибки в их согласованности. Для того чтобы применять диаграммы UML для генерации кода, был создан язык, получивший название исполняемого UML (Executable UML, xUML [19, 33, 34]). Формально этот язык является профилем UML, т.е. подмножеством языка UML с четко определённой семантикой для каждого элемента. Это подмножество выбрано так, чтобы сделать на основе UML полноценный визуальный язык программирования общего назначения. Так же как и UML, xUML основан на объектно-ориентированном подходе, т.е. описание системы ведется в терминах классов, атрибутов и т.п.

2.3.1 Описание

Для задания семантики отдельных элементов в xUML используется явно определённая семантика действий (Precise Action Semantics, PAS[1], [34]). PAS — это фиксированный набор семантических операций, не имеющий конкретного синтаксиса и реализации. PAS обеспечивает независимость, например, от конкретной целевой платформы или языка, на которых будут исполняться модели. Благодаря тому, что элементы обладают фиксированной семантикой, появляется возможность однозначно генерировать исполняемый код по модели, т.е. можно говорить об интерпретации модели.

Программирование на xUML основано на трёх следующих компонентах.

- Диаграмма классов, характеризующая структурную модель наблюдаемой системы. В неё входит отображение объектов реального мира классами. Эти классы имеют атрибуты, а связи между классами представляются в виде отношений.
- Диаграмма состояний, определяющая набор действий, которые совершит активный объект, т.е. изображающая жизненный цикл объекта, представленный в виде конечного автомата.
- Язык действий (Action Language, AL), позволяющий задавать поведение объекта при проходе каждого состояния в диаграмме состояний. Он необходим для создания экземпляров классов, установки связей между ними, выполнения различных операций над атрибутами и т.п. AL является конкретной реализацией PAS, выбор которой лежит на пользователе, а не заложен в xUML. xUML явно определяет только PAS.

[1] Стандартизирована Object Management Group (OMG) в 2001 году

Таким образом, упрощенно данный подход основан на представлении поведения системы в виде набора диаграмм состояний для каждого активного объекта. Действия при проходе каждого состояния в таких диаграммах записываются при помощи AL.

Действие и PAS — это ключевые понятия в исполняемой части языка xUML. Действие — это конкретная операция, определённая на элементе модели, принадлежащая классу и характеризующая его поведение после инициализации. Достаточно выучить только один AL, т.к. остальные будут иметь тот же семантический набор операций. На данный момент существует довольно много готовых AL: OAL [36], SMALL [44], TALL [34] и т. д.

2.3.2 Анализ

Существует несколько программных средств (например, CASSANDRA/xUML[1], Cameo Simulation Toolkit[2], Telelogic Tau[3], UniMod[4] [2]), позволяющих интерпретировать диаграммы на xUML. Однако, этот язык основан на понятиях объектно-ориентированного подхода и на стандартных конструкциях текстовых языков программирования (ветвления, циклы и т.п.), поэтому на практике значительного повышения уровня абстракции он не даёт. Поведение системы нужно описывать в терминах диаграмм состояний, хотя в общих случаях оно может быть устроено так, что его выражение при помощи конечных автоматов будет очень громоздким и сложным.

2.4 EProvide

EProvide (Eclipse Plugin for Prototyping Visual Interpreters and Debuggers[5]) — это подключаемый модуль (плагин) для среды разработки Eclipse[6], позволяющий быстро задавать операционную семантику для предметно-ориентированных языков, основанный на технологиях моделирования Eclipse [13] (Eclipse Modeling Framework[7] (EMF), Graphical Modeling Framework[8] (GMF)) и встраивающийся в систему исполнения программ Eclipse. Подход, использующийся в этом плагине, описан в статьях [41, 48].

2.4.1 Описание

Обычно семантику языков моделирования записывают при помощи различных правил преобразования моделей, например, используя стандарт OMG Query/View/Transformation (QVT[9]). С их помощью можно переводить модели с одного языка на другой. Часто вторым языком является язык общего назначения, обладающий возможностью компиляции или интерпретации (например, Java, C++ или Python). Такой трансляционный подход в задании

[1] CASSANDRA/xUML, http://www.knowgravity.com/ger/value/cassandra.htm
[2] Cameo Simulation Toolkit, http://www.magicdraw.com/simulation
[3] Telelogic Tau, http://www-01.ibm.com/software/awdtools/tau/
[4] UniMod, http://unimod.sourceforge.net/index.html
[5] EProvide, eprovide.sourceforge.net
[6] Eclipse, www.eclipse.org
[7] EMF, http://www.eclipse.org/modeling/emf
[8] GMF, http://www.eclipse.org/modeling/gmp
[9] QVT, http://www.omg.org/spec/QVT

семантик уменьшает уровень абстракции, т.к. разработчику нужно разбираться сразу в двух предметных областях — исходной и целевой, в которую происходит переход. Обеспечение обратной связи (например, в случае ошибок) целевой модели и исходной также является довольно сложной задачей. Такой подход является процедурой циклической разработки (round-trip engineering[1], [43]).

Весь набор способов задания семантики для произвольных языков можно разделить на несколько классов. Одним из основных классов является операционная семантика[2] (ОС). ОС языка представляет любой конкретный объект, записанный при помощи этого языка, в виде набора вычислительных шагов. Формально, её можно представить в виде пары $<\Gamma, \rightarrow>$, где Γ — множество конфигураций, а $\rightarrow$ — бинарное отношение перехода на множестве конфигураций. ОС бывают двух типов: структурная операционная семантика (СОС), предложенная Плоткиным [38], и естественная семантика.

Главным понятием в СОС является состояние. Обычно это набор определённых переменных и их значений. Конфигурацией же является пара $<S, \sigma>$, где S — фрагмент программы, а σ — текущее состояние. Отношение перехода также является бинарным отношением на множестве конфигураций. Таким образом, переходы в СОС описываются согласно абстрактным синтаксическим конструкциям языка, появляется возможность задавать для языков абстрактные интерпретаторы, основанные на их синтаксисе. Также СОС позволяет использовать структурную индукцию, например, при доказательстве корректности интерпретаторов и отладчиков.

Ваксмут (Wachsmuth) предложил использовать СОС, записанную декларативным текстовым способом при помощи языка QVT Relations [35], являющегося частью стандарта OMG QVT. QVT Relations — это высокоуровневый декларативный язык, разработанный для создания как односторонних, так и двухсторонних преобразований моделей. Позже в работе [41] этот подход был совмещён с технологиями визуального моделирования, и был реализован подключаемый модуль EProvide, позволяющий прототипировать визуальные интерпретаторы и отладчики для предметных языков. Также были добавлены и другие языки для описания семантики: Java, Abstract State Machines (ASM, [18]), Prolog и Scheme[3].

Важно отметить, что при интерпретации или отладке происходит изменение состояния модели, которое, во-первых, нужно возвращать к исходному состоянию в конце интерпретации или при ручной остановке, а во-вторых, это состояние нужно уметь изменять в режиме отладки через графический интерфейс пользователя. Для этого необходимо вручную создать отдельную от исходной метамодель, состояние экземпляров элементов которой и будет изменяться. Также туда можно добавить дополнительные структуры, необходимые, например, для обеспечения функциональности точек останова.

[1] http://en.wikipedia.org/wiki/Round-trip_engineering
[2] http://en.wikipedia.org/wiki/Operational_semantics
[3] Scheme, http://www.r6rs.org

2.4.2 Анализ

Подход, предложенный Ваксмутом, является достаточно общим и позволяет работать с различными языками спецификаций, с помощью которых будет происходить непосредственное задание семантики, имеет гибкую позицию относительно реализации точек останова и прочих функциональностей стандартных отладчиков, полноценно реализован под платформу Eclipse. Но если смотреть со стороны удобства использования, то ручное написание семантики на каком-либо текстовом языке понижает достигнутый уровень абстракции. И для успешного применения этого подхода в средах, отличных от Eclipse, необходимо наличие интерпретатора данного языка, реализация которого может оказаться нетривиальной задачей.

От xUML EProvide отличается тем, что для задания семантики достаточно лишь описать метамодель языка и способ трансформации моделей, например, при помощи стандарта QVT Relations, и не нужно рисовать много UML диаграмм. В EProvide легко можно организовать пошаговое исполнение, совмещённое с поддержкой различного вида точек останова и слежения за значениями атрибутов элементов. Для xUML, как было отмечено в параграфе 2.3.1, существует множество поддерживающих его программных средств, что усложняет выбор среди них при необходимости решить определённую задачу.

2.5 Основанные на технологии преобразования графов

2.5.1 Dynamic Meta Modeling

Другим интересным способом задания семантики визуальных языков является Dynamic Meta Modeling (далее DMM). Этот подход подробно рассмотрен в кандидатской диссертации Яна Хаусмана (Jan Hausmann) из университета в Падерборне (Германия) [24]. Несмотря на то, что приведённое в работе исследование изложено с опорой на UML, DMM позволяет определять семантику для любых визуальных языков. DMM обладает высокой степенью формальности, ясен и адекватен при определении семантики интерпретации визуальных языков и основывается на концепции семантики действий (action semantics), совместившей в себе черты как денотационной, так и операционной семантики.

2.5.1.1 Описание

Общая схема DMM-подхода изображена на рис. 1. Данный рисунок построен, исходя из предположения, что абстрактный язык программирования должен состоять из трёх частей: описания синтаксиса (syntax definition), описания семантики (semantics definition, т.е. множества возможных действий) и семантического соответствия (semantic mapping), которое отвечает за придание определённым синтаксическим конструкциям некого семантического смысла.

Для всех визуальных языков характерно наличие двух уровней: уровня языка и уровня модели (в литературе также встречаются другие термины — уровень метамодели и уровень модели соответственно). Уровень языка содержит описание синтаксиса языка, правила отрисовки графических элементов и описание других его особенностей. На уровне модели происходит непосредственное создание конкретной спецификации (expression) на этом языке,

состоящей из элементов метамодели (model elements). На рис. 1 эти уровни помечены буквами “L” (language) и “M” (model). Наличие такого разделения подразумевает, что компоненты на уровне модели должны быть согласованы (связь conforms to) с компонентами на уровне языка, т.е. конкретная модель должна удовлетворять синтаксическим правилам языка, конкретная реализация поведения системы должна удовлетворять правилам определения семантики на уровне языка и т.п.

В качестве синтаксической модели на уровне языка и её представления на уровне модели в DMM может выступать любой визуальный язык, что обеспечивает универсальность данной технологии. Исполнение же конкретной модели в итоге будет представлено в виде системы переходов между состояниями с метками (Labeled state Transition System, LTS).

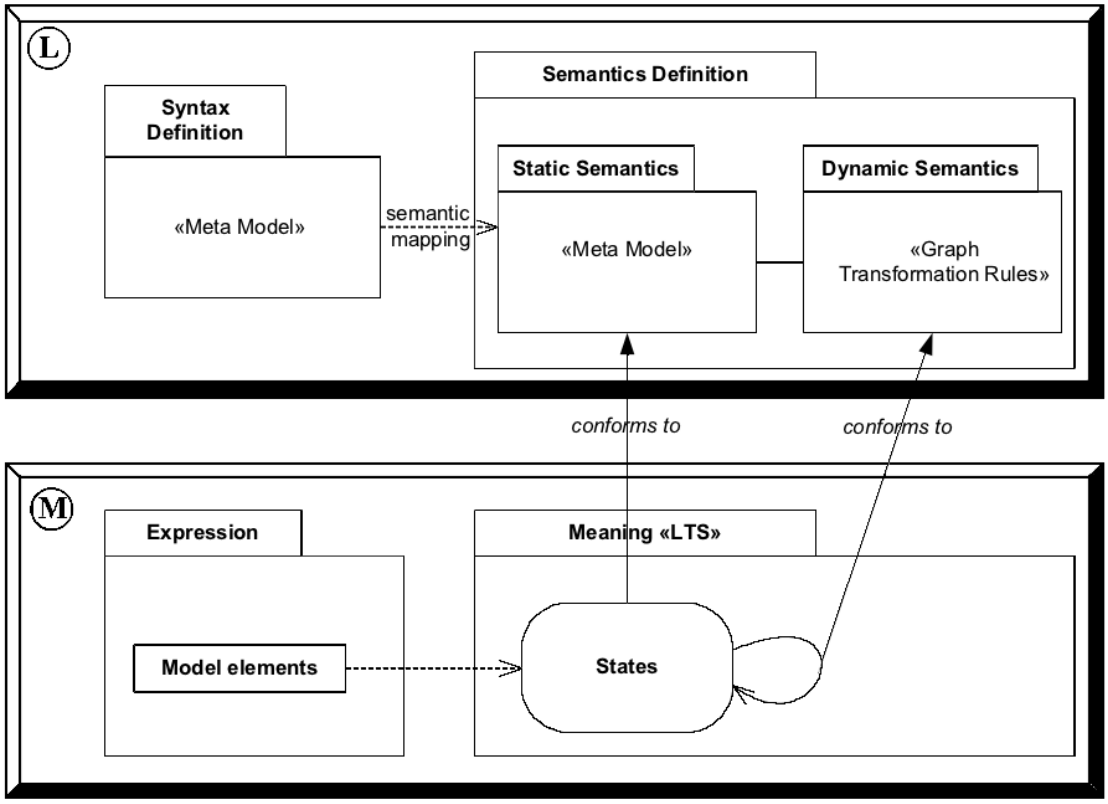

Рис. 1. Общая схема DMM-подхода (этот рис. и все последующие в этом разделе взяты из работы [24])

Для организации семантического соответствия используется концепция мета-отношений (Meta Relations). По своей структуре она значительно сложнее, чем просто соответствие одного синтаксического элемента одному семантическому, т.к. нужно учитывать вложенные соответствия, изображенные на рис. 2. Допустим в синтаксической метамодели языка (syntax definition) присутствует агрегирование, и элемент “Class” агрегирует элемент “Attribute”. В определении семантики (semantics definition) элемент “Class” соответствует элементу “Object”, а элементу “Attribute” — “Slot”, агрегирование которых устроено аналогично. Данная схема интуитивно предполагает, что каждый атрибут класса соответствует ровно одному слоту соответствующего объекта, а не слоту произвольного другого. Таким образом соответствие

“Class”/”Object” а некотором смысле задаёт ограничения на соответствие “Attribute”/”Slot”, которое является вложенным в него.

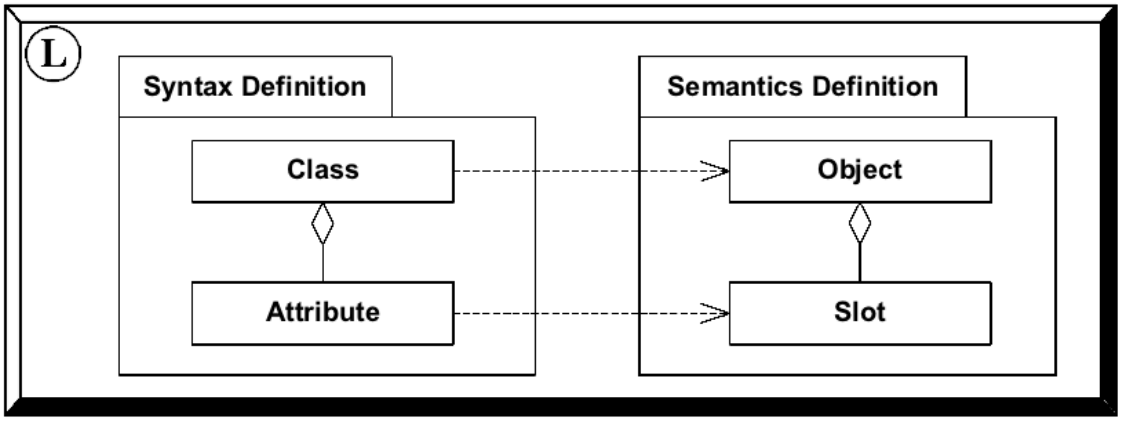

Рис. 2. Пример вложенного соответствия

Семантической областью, т.е. системой, в терминах которой будет определяться семантика и выполняться интерпретация моделей, в DMM является технология преобразования графов, рассматриваемая далее. Семантика в данном подходе делится на статическую (static semantics) и динамическую (dynamic semantics). Статическая семантика на уровне языка представляет собой метамодель для описания набора состояний системы и доступных правил динамической операционной семантики без конкретной реализации. Семантическое соответствие синтаксической области на статическую семантику языка является денотационной частью DMM. Динамическая семантика на уровне языка является набором правил преобразований графов (graph transformation rules) и составляет операционную часть DMM.

В результате, на уровне модели в качестве реализации подходов, описанных на уровне языка, имеем систему LTS (meaning “LTS”), согласованную с ними, т.е. состояния (states) такой системы соответствуют состояниям, описанным в статической семантике на уровне языка. Правила перехода между этими состояниями являются правилами преобразования графов, которые реализованы на уровне языка, согласованы со статической семантикой и удовлетворяют синтаксическим особенностям создания правил, о которых будет рассказано в следующем параграфе.

2.5.1.2 Преобразования графов

В DMM модель, созданная на визуальном языке, рассматривается в виде типизированного мультиграфа с атрибутами и метками на узлах и рёбрах, допускающего наследование, связанного с возможностью расширения типов узлов и рёбер.

В общем виде [40], правило преобразования графов содержит левую и правую часть. Суть этих частей в следующем: в исходном графе ищется подграф, совпадающий с левой частью, и заменяется на граф из правой части. Для удобства левую и правую часть часто совмещают в одну, добавляя к каждому новому элементу метку {new}, а к каждому удалённому — метку {destroyed}.

В DMM к этому описанию было добавлено несколько расширений: отрицающие применение условия (Negative Application Conditions, NAC) и универсальное замыкание

(Universal Quantification, UQ). При использовании NAC к правилу добавляется несколько перечеркнутых элементов, означающих, что правило применимо только в случае отсутствия данных элементов в графе (правильным образом соединённых с остальными элементами искомого подграфа). Наличие же в правиле универсальной структуры (Universally Quantified Structures, UQS) означает, что правило будет применено сразу ко всем подходящим местам в исходном графе.

Для того, чтобы такие правила было удобно использовать, у каждого из них задаётся имя, список параметров и, по необходимости, ссылка на базовый элемент, для которого оно должно применяться. Это позволяет внутри одних правил вызывать для конкретных элементов другие (invocations).

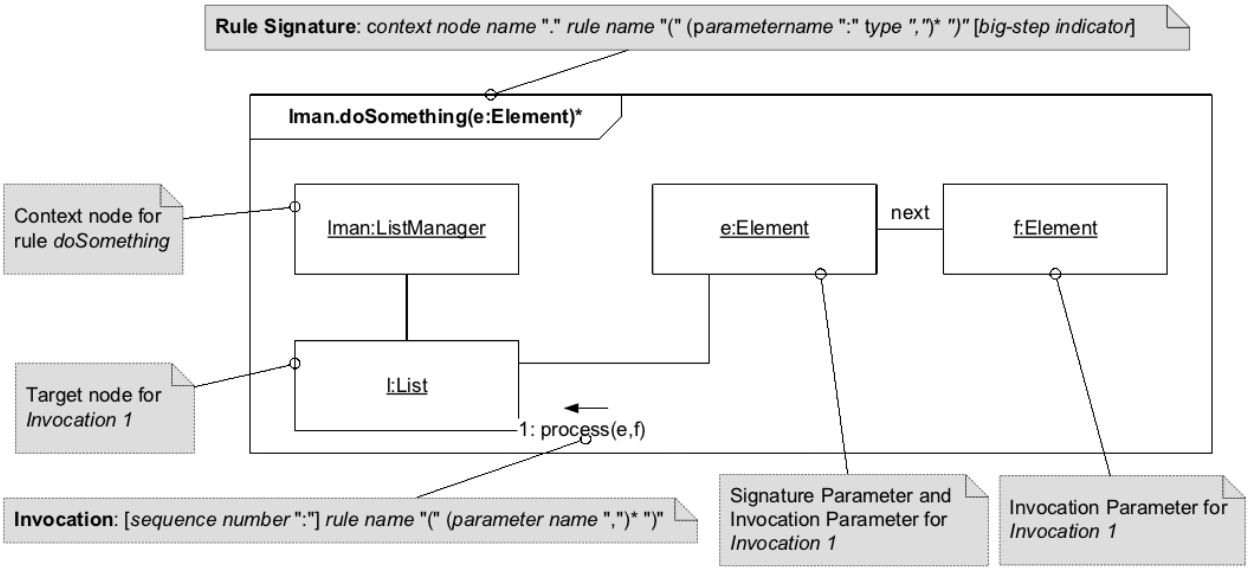

Рис. 3. Сигнатура правила преобразования графов

На рис. 3 приведена общая сигнатура как правила (rule signature), так и вызовов других правил (invocation). Сигнатура правила состоит из четырёх частей:

- название правила (rule name);
- индикатор (big-step indicatior, “*”) того, что правило является правилом большого шага, которые могут применяться к исходному графу в любой момент, в отличие от правил малого шага, которые могут вызываться только внутри других правил;
- имя базового элемента (context node name, на рис. 3 это элемент “lman” типа ListManager, изображённый внутри правила в левой его части), для которого можно вызывать данное правило внутри других;
- список элементов-параметров (“parametername:type”, на рис. 3 это элемент “e” (signature parameter) типа Element).

Сигнатура вызова одного правила внутри другого состоит из трёх частей:

- порядкового номера (sequence number), означающего порядок выполнения этих вызовов внутри данного правила;
- названия правила (rule name);

- списка элементов-параметров (“parametername:type”).

На рис. 3 приведён один такой вызов — вызов правила process (“1:process(e, f)”). Такой вызов в этом правиле один, поэтому он имеет порядковый номер 1. Элемент (target node), для которого он вызывается, — элемент l типа List, параметры правила (invocation parameters) находятся по соответствующему имени.

2.5.1.3 Анализ

DMM-подход является зрелой технологией для задания семантики визуальных языков. Он формален, точен, универсален и понятен, имеет высокую наглядность как при задании семантики, так и при интерпретации моделей.

DMM потенциально можно использовать и для визуализации процесса отладки диаграмм. Можно организовать настраиваемое пошаговое исполнение, т.е. сколько правил следует применить перед следующим обновлением диаграммы, интерфейс, позволяющий следить за значениями атрибутов различных элементов и изменять их прямо во время исполнения и т.п. В случаях, когда правило можно применить в разных местах или когда есть несколько правил, которые можно применить на данном шаге, можно предоставлять этот выбор пользователю. Также можно ввести понятие точек останова, причём разных видов. Например, точка останова на применение конкретного правила, на изменение конкретного элемента, на получение определённой конструкции в графе модели и т.п. Подробнее с этими идеями можно ознакомиться в работе [16].

Среди главных недостатков этого подхода можно выделить высокую алгоритмическую сложность, связанную с задачей поиска подграфа в графе, которая является NP-полной, а также отсутствие реализации.

Если сравнивать DMM с xUML, то можно выделить следующие существенные отличия: DMM может применяться к любым визуальным языкам, а xUML фиксирован; семантика в DMM задаётся, в основном, визуально, а в xUML важную часть составляет код на AL; DMM подразумевает, что за ходом интерпретации можно будет наблюдать визуально, следя за изменениями модели, а для xUML эта визуализация будет состоять из подсветки текущего исполняемого элемента модели.

DMM, так же как и EProvide, основан на преобразованиях моделей. Но в отличие от EProvide способ задания семантики в нём более естественен, т.к. является визуальным. К тому же он основан на хорошо исследованной области преобразования графов, что способствует, например, доказуемости поведения программ. DMM имеет пару особенностей, взятых из формальной теории преобразования графов, например, NAC и UQS, а EProvide зато поддерживает более обширный функционал для задания ограничений на применение своих правил (отношений) при помощи when- и where- блоков, а также позволяет осуществлять различные запросы на входной модели.

2.5.2 AToM^3

Среди открытых DSM-платформ, позволяющих создавать новые DSL и соответствующие им DSM-пакеты, помимо среды моделирования Eclipse Modeling Project [13] существует также ряд других (см., например, [15, 17, 27]). Среди них можно выделить кроссплатформенный

проект AToM^3 (A Tool for Multi-Formalism and Meta-Modelling, [31]), написанный целиком на языке Python. Кроме традиционного для таких платформ функционала AToM^3 предоставляет возможность трансляции моделей, записанных как на одном визуальном языке, так и на разных, оптимизации (по сути, замене конструкций более эффективными) и симуляции (интерпретации) моделей, а также генерации кода по моделям. Подход, реализацией которого и является AToM^3, описан в статье [32] Хуана де Лара (Juan de Lara) и Ханса Вангелуве (Hans Vangheluwe).

2.5.2.1 Описание

Трансляция и интерпретация моделей, а также генерация кода по моделям осуществляется в AToM^3 при помощи графовых грамматик, т.е. при помощи преобразований графов, более подробно описанных в DMM-подходе, рассмотренном выше. Создатели AToM^3 понимают, что задача поиска подграфа в графе в общем случае NP-полная и что это накладывает свои ограничения на производительность, однако использование малых графов в качестве левой части правил, а также наличие большого числа ограничений на типы и значения атрибутов значительно уменьшает глубину поиска. Среди преимуществ данного подхода авторы отмечают его формальность и высокоуровневость, визуальность и серьезную теоретическую основу.

В рамках интерпретации и отладки AToM^3 поддерживает пошаговое и непрерывное исполнение, возможность изменения модели “на лету”, т.е. прямо перед исполнением очередного шага. Механизм исполнения не рассчитан на выделение особой модели времени исполнения, как, например, в средстве EProvide, с которой будут происходить задаваемые правилами семантики преобразования, поэтому все изменения происходят с оригинальной моделью. Операцию возвращения модели в начальное состояние после окончания интерпретации нужно проделывать вручную после каждого исполнения.

Важным моментом внутренней реализации проекта AToM^3 является то, что каждое правило преобразования графов задаётся отдельно, при этом необходимо определять левую часть правила в одном окне, а правую — в другом. AToM^3 является средой с мультиформализмом, т.е. можно одновременно работать с несколькими визуальными языками сразу. Из-за этого для создания правил преобразования модели на одном языке в другой необходимо использовать обобщённые связи, которые могут соединять элементы различных языков.

Модель каждого правила сохраняется отдельно в виде скрипта на языке Python, по этой модели генерируется ещё один скрипт, являющийся кодом, который будет делать необходимые изменения, согласно правилу. Первый файл необходим для того, чтобы можно было удобно изменять логику работы этих правил, а второй — чтобы быстро понимать, что нужно сделать.

Другой отличительной от DMM-подхода особенностью является возможность при помощи метки “ANY” определять, что значение атрибута для применения правила может быть любым, при помощи метки “COPIED” задавать, скопировать ли атрибут элемента модели или изменить (тогда метка должна быть “SPECIFIED”). Также можно ставить более сложные ограничения на применение правила, связывающие атрибуты нескольких элементов левой части, при помощи OCL [49] или задавать их на языке Python. Кроме дополнительных условий можно на тех же языках указывать список действий, которые необходимо исполнить после применения правила.

2.5.2.2 Анализ

Подход к заданию семантики интерпретации, использованный в проекте AToM^3, фактически является частичной реализацией формального DMM-подхода с расширением в виде возможности ставить дополнительные условия на значения атрибутов элементов для применения правил, а также производить некоторый набор операций после непосредственного изменения модели. В связи с этим на него распространяется то же замечание про NP-полноту задачи поиска подграфа, что и на DMM-подход. Возможность конвертации моделей на одном языке в другой, а также изменение модели в оптимизационных целях являются отличительными составляющими данного подхода.

Формально подход, реализованный в AToM^3, в сравнении с xUML и EProvide почти полностью совпадает с DMM-подходом. Существенным улучшением DMM-подхода является возможность задавать ограничения на применение правил при помощи OCL или на языке Python и исполнять определённый код на языке Python после применения правила. Причём оба этих механизма имеют полный доступ ко всем элементам, содержащимся в правиле. Благодаря этому данный подход становится больше похож на реализацию подхода из EProvide с использованием графической нотации задания преобразований моделей.

Также данный подход от DMM отличает некомпактный способ задания правил преобразования. Нужно явно указывать левую и правую часть и задавать соответствие элементов индексами. Зато для атрибутов в левой части можно указывать значение “ANY”, когда не важно, чему он равен, в правой части — “COPIED”, если при применении правила значение нужно скопировать, и в обоих частях — “SPECIFIED”, если значение задаётся явно.

2.5.3 Анимированная симуляция диаграмм на базе GenGED

Описание данного подхода изложено в статьях [20, 21, 22] на различных примерах. В [22] рассматривается система, состоящая из набора ресторанов, обслуживающих очереди покупателей на машинах. В [20] приводится пример анимации работы лифта при помощи сетей Петри, а в [21] разобрана проблема обедающих философов. Главным в этом подходе является тот факт, что при интерпретации пользователь видит не изменение диаграммы на визуальном языке, а некоторую анимацию.

2.5.3.1 Описание

Представление предметной области задачи и самой задачи в виде визуального языка моделирования и с его помощью созданной диаграммы, соответственно, и последующая интерпретация диаграммы в терминах этого же визуального языка порой являются недостаточно очевидными для пользователей, не являющихся разработчиками этого языка. Переход от диаграмм к реальным анимационным картинкам, которые будут непрерывно изменяться при интерпретации, как видеофильм, должен сгладить этот семантический разрыв.

Как показано на рис. 4, основой для всего процесса анимации является некоторая поведенческая модель (UML behavioral model), записанная при помощи UML. Для исполнения она переводится в симуляционную систему (simulation system), на экран же пользователю выводится набор анимационных представлений (animation views). В симуляционной системе

модель переходит из состояния в состояние при помощи симуляционных шагов (simulation steps), а в анимационных представлениях используются анимационные шаги (animation steps).

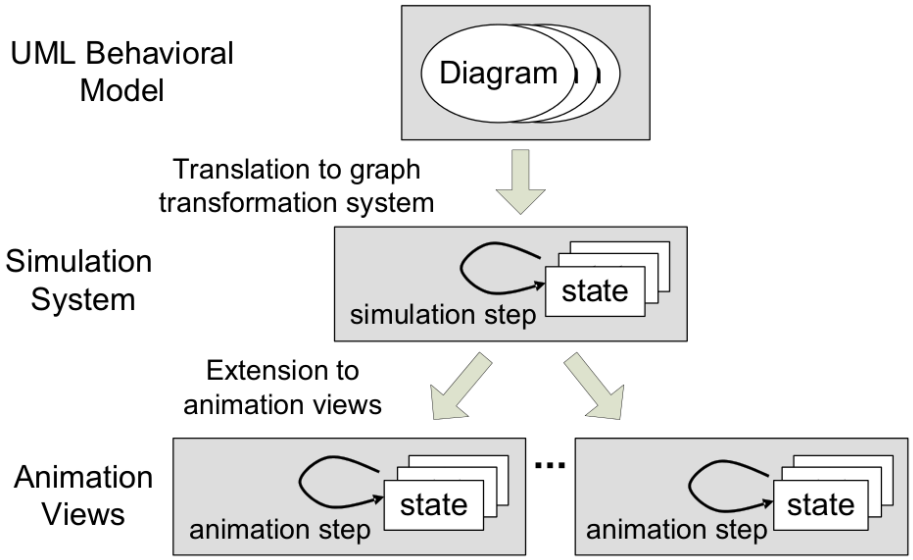

Рис. 4. Формальная схема анимационного представления исполнения (этот рисунок взят из работы [22])

Система, заданная при помощи набора UML диаграмм, представляется в виде объектной UML диаграммы, формализованной как граф, а сам процесс исполнения заключается в преобразовании этой диаграммы. Для того чтобы работать с анимационным представлением, не выходя за рамки объектных диаграмм, производится расширение набора способов визуализации вершин графа модели путём добавление новых классов в диаграмму классов, описывающую предметную область. Подробнее об этом расширении будет рассказано в следующем разделе на конкретном примере.

В рамках способа задания семантики в данном подходе используются правила преобразования графов с ограничениями на языке OCL как на уровне симуляции, так и на уровне анимации. Для обеспечения анимации сценариев предлагается использовать средство GenGED[1], позволяющее расширять системы преобразования графов такой функциональностью и экспортировать результат в формат SVG[2].

2.5.3.2 Анализ

В плане способа задания семантики данный подход полностью основывается на технологии преобразования графов, что даёт возможность создавать наглядные и понятные операционные семантики. С их помощью данный подход позволяет задавать реальное исполнение различных поведенческих UML-диаграмм (например, диаграмм кооперации и протокольных диаграмм состояний), при помощи которых описывается разрабатываемая система. Также показано её применение для создания нового уровня представления интерпретации системы в виде анимации. Использование данного подхода при задании

[1] GenGED, URL: http://user.cs.tu-berlin.de/~genged
[2] http://www.w3.org/TR/svg

семантики исполнения позволяет существенно сократить разрыв между предметной областью и её представлением в системе.

Как и все остальные подходы, использующие преобразования графов, из-за NP-полноты задачи поиска подграфа в графе, на больших графах возможно значительное уменьшение производительности, в отличие от подходов, использованных в xUML или в EProvide.

По фактическому содержанию данный подход является некоторым упрощением способа, использующегося в AToM^3, т.к. не позволяет задавать на языке Python дополнительные ограничения на применение правил и исполнять некоторый код после применения. Таким образом, сравнение его с остальными подходами аналогично уже приведённым сравнениям. Однако главным недостатком, в отличие от DMM или AToM^3, является его неуниверсальность, а именно тот факт, что исходную систему нужно описывать набором UML диаграмм и что исполняемая модель будет представлять собой объектную диаграмму. Существенным плюсом же является то, что процесс визуализации интерпретации происходит максимально понятно для пользователя, т.к. используются не простые диаграммы, а анимация.

3 Описание подхода

По итогам проведённого анализа и сравнения уже существующих подходов к заданию исполнимой семантики визуальных языков было сформулировано собственное решение, разработанное применительно к системе QReal. Семантической областью данного решения является технологии преобразования графов и трансформации моделей, которые были описаны в DMM-подходе. Как и в подходе, использующемся в AToM^3, на применение правил преобразования графов можно накладывать дополнительные условия, также присутствует возможность исполнения некоторого кода на интерпретируемом языке после применения правила. Всего в предложенном подходе можно выделить три основные составляющие.

3.1 Семантическая модель

С точки зрения реализации системы QReal, любая модель на некотором визуальном языке представляется в виде двух частей: логической модели и графической модели. Это происходит как при представлении, так и при хранении данных, т.е. некоторый элемент модели может иметь несколько графических представлений, которые отвечают за отображение элемента пользователю на соответствующей диаграмме, и только одно логическое, использующееся для хранения общей информации об элементе и его атрибутах. Также бывают элементы, не имеющие либо графического, либо логического представления совсем. Инвариантом остается факт, что конкретному графическому представлению некоторого элемента соответствует не более одного логического. Таким образом, фиксированную логическую модель пользователь легко может отобразить на экране различными способами. Работа с различными представлениями конкретной модели визуального языка осуществляется при помощи специального API графической и логической модели.

Элементы пользовательских визуальных языков часто соответствуют конкретным сущностям из предметной области, для которой этот язык и создаётся. Таким образом, при определении семантики визуального языка иногда может быть нужно указать семантику отдельных элементов явно, т.е. набор дополнительных специфических атрибутов и процедур на интерпретируемом текстовом языке над значениями атрибутов этого элемента, которые будут использоваться только при формализации общей семантики языка и при непосредственной интерпретации модели. Такой набор процедур для конкретного элемента будет составлять его возможное поведение. Примером семантического атрибута может быть, например, логический атрибут started у начального элемента InitialNode редактора блок-схем. По умолчанию он равен false, при начале интерпретации диаграммы выставляется в true, а после завершения — снова false. Используется для определения того, что уже начато исполнение данной модели.

В связи с этим к существующему в системе разделению на графическую и логическую модель нужно добавить новую семантическую модель, которая бы содержала эти новые атрибуты и процедуры. Доступ к ним осуществлялся бы через новое API семантической модели.

Определять семантику элементов исходного визуального языка нужно в специальном расширении уже существующего метаредактора, содержащего стандартный набор средств для добавления новых атрибутов к элементам и создания новых элементов. Расширение

заключается в возможности использования элементов исходного языка и добавления атрибутов к ним. Для работы с данным расширением нужно указать путь до метамодели изменяемого языка.

В итоге после добавления к элементам необходимой семантики генерируется метамодель нового визуального языка, содержащая все элементы исходного вместе с их семантикой. Эта метамодель автоматически компилируется и загружается в работающую систему. В дальнейшем пользователь работает только с этим полученным языком с добавленной семантикой элементов и создаёт модели, подлежащие интерпретации, именно на нём.

Семантика элементов является специфической служебной информацией, необходимой для работы внутренних составляющих интепретатора, поэтому при обычном использовании получившегося визуального языка пользователю она не отображается. Доступ к ней можно получить только в редакторе семантики визуального языка, о котором пойдёт речь ниже.

Таким образом, на основе метамодели исходного визуального языка при помощи метаредактора генерируется метамодель того же языка, расширенная семантикой элементов. Эта метамодель затем используется в редакторе семантики этого языка, в котором будут непосредственно задаваться правила семантики, а также на её основе в систему подгружается новый редактор целевого визуального языка, с которым пользователь и работает впоследствии. На данный момент семантику элементов нужно добавлять вручную в логическую модель визуального языка.

3.2 Редактор семантики визуального языка

3.2.1 Общее описание редактора

Редактор семантики визуального языка генерируется на основе полученной на предыдущем шаге метамодели исходного языка, расширенной семантикой элементов. Сборка и загрузка в работающую систему созданного редактора происходит автоматически. Получившийся редактор позволяет задавать семантику именно для выбранного визуального языка, в нём будут содержаться все элементы из указанной метамодели. Семантические атрибуты и поведение этих элементов будут доступны для редактирования.

Семантика визуального языка будет состоять из набора правил преобразования графов, пошаговое применение которых к графу исходной модели позволит визуализировать процесс интерпретации. На каждом шаге выбирается случайное правило или правило с наивысшим приоритетом (если таких несколько, то берётся произвольное), которое может быть применено к модели, и применяется в произвольном подходящем под правило месте. Принцип работы такого процесса напоминает выполнение алгоритмов Маркова на строках, а наличие приоритета позволяет упорядочить возможное применение правил.

Заданная семантика визуального языка может быть предназначена как для интерпретации и отладки, так и для генерации полноценного кода по моделям. Таким образом, в зависимости от вида семантики нужно указывать разное время задержки между шагами исполнения, т.е. применения правил к модели.

3.2.2 Элементы редактора семантики

Редактор семантики визуального языка позволяет определить упомянутый набор правил преобразования графов визуальным способом. Для этого, кроме элементов исходного визуального языка, в данном редакторе можно использовать следующие новые узлы: “Semantics Rule”, “Wildcard”, “Initialization”, “Control Flow Mark”, и связи: “Replacement”, “Control Flow Location”. Назначение каждого из них представлено ниже.

Левая и правая часть каждого правила совмещены, как было предложено в DMM-подходе, и помещаются внутрь элемента “Semantics Rule”, являющего контейнером для всех представленных элементов. Создание и удаление элементов происходит при помощи добавления к ним меток “@new@” и “@deleted@” соответственно. Замена одного элемента на другой с сохранением всех его связей задаётся при помощи специальной направленной связи “Replacement”.

Для того, чтобы выделить поток исполнения в модели, можно использовать специальный маркер “Control Flow Mark”, связь “Control Flow Location” с которым показывает, что поток находится на этом элементе. При интерпретации вместо обычной отрисовки этих элементов происходит просто подсветка места в диаграмме, на котором находится поток исполнения.

В элементе “Initialization” определяется тип семантики языка (интерпретация или генерация), а также задаётся в текстовой форме инициализация интерпретации, описание которой дано в параграфе 3.3. “Wildcard” — это унификатор узла, обозначающий произвольный элемент модели.

3.2.3 Особенности создания правил семантики

Во время поиска левой части правила при сравнении элементов правила и элементов в модели проверяется равенство не только типов этих элементов, но и значений их атрибутов. В редакторе можно задавать значение как обычных атрибутов элементов, так и семантических. Если атрибут какого-то элемента правила не проинициализирован, то значение этого атрибута у элемента в модели может быть любое. При добавлении нового элемента в правило все его атрибуты автоматически не проинициализированы. Если ни тип, ни атрибуты элемента в правила не важны, то можно использовать унификатор узла или связи. Сравнение любого элемента модели того же класса (узел или связь) с ними будет возвращать истину.

Также для каждого правила можно определить его приоритет и выбрать текстовый интерпретируемый язык, на котором будут записываться ограничение и реакция на применение соответствующего правила. Простейший вариант приоритета — целое число. В общем случае, приоритет — это значение некоторой функции, заданной на атрибутах элементов правила, т.е. приоритет зависит от конкретного места в исходной модели. Например, если у каждого элемента визуального языка есть целочисленный вес, а правила сначала должны применяться на подграфах с максимальным суммарным весом элементов, что зависит от того, какой именно подграф исходной модели совпал с левой частью правила. Данная функциональность может понадобиться при визуализации различных формальных алгоритмов на графах.

В идеале необходимо обеспечить поддержку всех описанных в DMM-подходе возможных особенностей правил преобразования графов от вызовов в правиле других правил до

отрицающих применение условий и универсального замыкания. Но на данный момент достаточно ограничиться созданием, заменой и удалением элементов, а также наличием унификаторов.

3.3 Инициализация интерпретации, ограничение и реакция на применение правила

Помимо графической составляющей, которую составляют правила преобразования графов, в семантике визуального языка также присутствует и текстовая часть в виде начальной инициализации интерпретации, ограничений и реакций на применение правил. Опишем каждую из них более подробно.

На каждое исполнение модели создаётся один экземпляр интерпретатора текстового языка, использующегося для задания поведения элементов, ограничений и реакций на применение правил и т.п. Таким образом, определённые в этом интепретаторе объекты (переменные, функции и т.д.) доступны в любой части текстовой составляющей семантики визуального языка. Задать начальное состояние интерпретатора (начальные значения общих переменных, общие функции и т.п.) перед непосредственным исполнением модели можно при помощи инициализации интерпретации. Это код на интерпретируемом языке, который записывается в специальном элементе “Initialization” и гарантированно будет исполнен до применения первого подходящего правила преобразования графов.

Ограничения на применение правил нужны для более детальной настройки возможности применения правила помимо условий на равенство конкретным значениям атрибутов элементов. Такие ограничения представляют собой функцию на интерпретируемом языке, возвращающую истину или ложь и имеющую доступ к значениям атрибутов элементов правила и к прочим объектам, определённым при инициализации или в ходе интерпретации. Если ограничение на подходящем под правило месте возвращает ложь, то оно не может быть здесь применено. Например, при работе с блок-схемами для создания правила перехода потока исполнения с условного элемента дальше по ветке, соответствующей истине, в ограничение на применение правила можно поместить равенство истине как условия, записанного в атрибуте соответствующего элемента, так и значение типа связи, которая из него выходит.

Для исполнения поведения элементов правила и организации их взаимодействия между собой, а также организации взаимодействия правил как таковых, было введено понятие “реакция на применение правила”. Это процедура на интерпретируемом текстовом языке, которая имеет доступ к значениям атрибутов элементов правила как на чтение, так и на запись. Как и ограничения, реакция может использовать уже определённые, например, при инициализации, объекты и любой другой функционал, доступный данному языку. Исполняется же этот код сразу после непосредственного создания новых и до удаления старых элементов согласно правилу.

4 Архитектура

Согласно предложенному подходу, были реализованы и встроены в систему QReal средства задания исполнимой семантики визуальных языков и дальнейшей интерпретации по ней. В данном разделе будут разобраны вопросы, связанные с архитектурой встраиваемого решения.

4.1 Визуальный интерпретатор как компонента системы QReal

Предложенный способ задания семантики визуальных языков и визуальной интерпретации моделей был реализован в рамках проекта QReal в виде отдельного подключаемого модуля, а части, отвечающие за поиск подграфов в модели и за добавление различных новых атрибутов и элементов в метамодель, была вынесена в общедоступное место, чтобы другие плагины при необходимости могли ими воспользоваться.

Для корректной работы реализованного визуального интерпретатора в запущенной системе должен присутствовать метаредактор, чтобы обеспечить возможность сборки сгенерированного по визуальному языку редактора семантики. Также данный плагин зависит от сторонней свободной библиотеки QScintilla[1], предоставляющей удобный текстовый редактор с автодополнением и подсветкой синтаксиса и использующейся при работе с текстовыми интерпретируемыми языками в реакции на применение правил и т.п. Наличие в операционной системе интерпретатора языка Python не обязательно, но желательно, т.к. с его помощью можно задавать сложные ограничения и реакции на применение правил.

В самой реализации данного плагина можно выделить четыре главных момента: генерация и загрузка редактора семантики указанного визуального языка, поиск подграфа в графе в рамках данной предметной области, изменение модели согласно правилу преобразования графов и интерпретация инициализационного кода, ограничений и реакций на применение правил. Особенности реализации данных составляющих будут описаны в следующем разделе. Рассмотрим сначала типичный алгоритм работы проектировщика визуального языка от задания семантики до самой интерпретации диаграмм.

4.2 Алгоритм работы проектировщика

Набор действий создателя визуального языка по добавлению к нему семантики и началу работы с интерпретатором моделей можно представить следующим образом. Проектировщик создаёт новый визуальный язык в метаредакторе, указав дополнительные атрибуты и поведение элементов, необходимые для исполнения, в логической модели. Потом через пользовательский интерфейс задаёт путь до метамодели получившегося языка. После этого открывает редактор семантики для данного языка, который был автоматически загружен в систему.

Далее в этом редакторе проектировщик рисует набор правил преобразования графов, т.е. описывает семантику разрабатываемого языка. Затем при помощи пользовательского интерфейса он загружает получившуюся семантику в интерпретатор, создаёт модель на исходном визуальном языке, используя соответствующий редактор и запускает её

[1] QScintilla, http://www.riverbankcomputing.com/software/qscintilla

интерпретацию, следя за ходом исполнения в самом редакторе, а также в специальном виджете, предназначенном для вывода информационных сообщений и сообщений об ошибках.

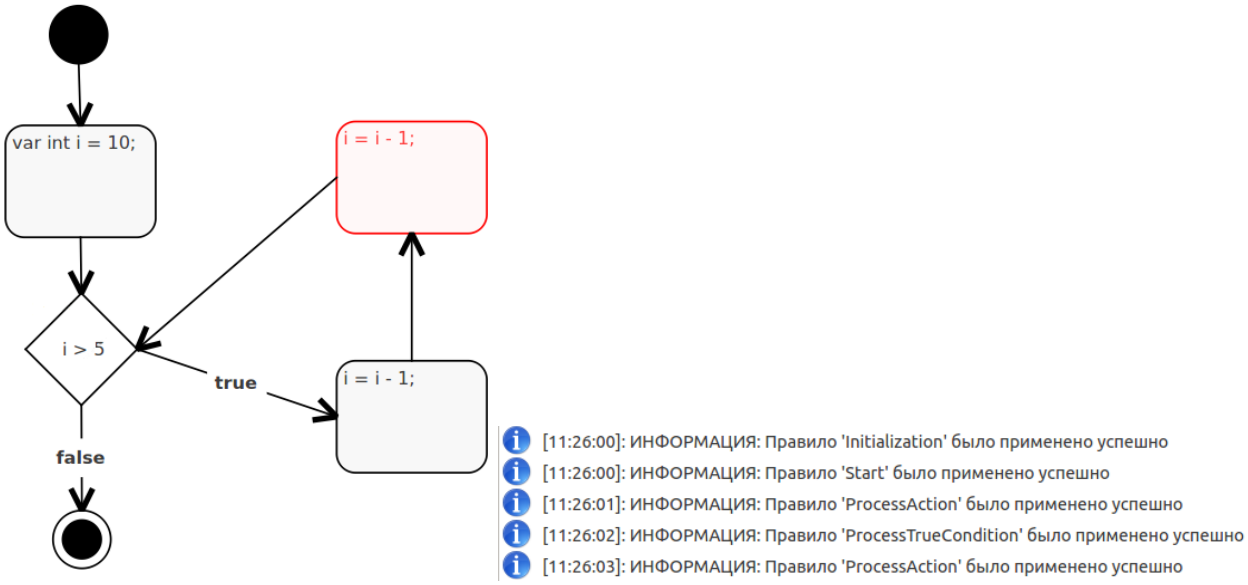

Рис. 5. Пример интерпретации диаграммы блок-схем

На рис. 5 изображена блок-схема простого цикла относительно значения переменной i. При интерпретации этой блок-схемы результаты применения различных правил выводятся на экран, также разработчик может наблюдать подсветку элемента с потоком исполнения. Теперь опишем последовательность действий, осуществляемых интерпретатором для того, чтобы применить очередное правило.

4.3 Алгоритм применения конкретного правила при интерпретации

В реализованном средстве задания семантики визуальных языков в качестве интерпретируемого языка, упомянутого в разделе 3, может использоваться Python. Опишем более подробно набор действий, которые выполняет интерпретатор при осуществлении очередного шага интерпретации с этим текстовым языком. Допустим очередное рассматриваемое интерпретатором правило гарантированно может быть где-то применено.

Сначала в исходной модели ищутся все соответствующие правилу подграфы и сохраняются в список. Для каждого из них проверяется ограничение на применение правила. При этом генерируется скрипт-ограничение в виде текстовой строки, которая затем побайтово записывается в процесс интерпретатора языка Python. Затем ожидается ответ от этого процесса в виде булевского значения. Отрицательный ответ означает, что подграф не прошёл проверку, что ведёт к удалению его из списка подходящих под правило подграфов.

Из списка, получившегося после проверки ограничений, выбирается первый подграф, и начинается применение правила к этому месту. При помощи API репозитория QReal согласно правилу создаются новые элементы и элементы на замену. После этого генерируется скрипт-реакция и сохраняется в виде отдельного файла на диск. В процесс интерпретатора языка Python записывается команда *execfile* с указанием пути до скрипта. Ожидается ответ от этого

процесса в виде множества изменённых значений атрибутов элементов. Затем этот ответ разбирается, и изменяются соответствующие атрибуты элементов. Потом удаляются и заменяются необходимые элементы, и запускается перерисовка диаграммы в редакторе. В самом конце происходит передвижение потока исполнения согласно правилу.

5 Особенности реализации

В реализации предложенного способа задания семантики визуальных языков в системе QReal можно выделить три основных направления: создание редактора семантики, непосредственное применение правила на графе, а также интерпретация текстовой составляющей семантики визуальных языков. В этом разделе будут разобраны основные проблемы, которые были решены при разработке соответствующих компонент.

5.1 Модуль работы с графами

В предложенном подходе модель на визуальном языке — это типизированный ориентированный мультиграф с атрибутами и метками на узлах и рёбрах. Данный модуль отвечает за поиск указанных подграфов в исходной модели, а также за корректное применение самих преобразований, т.е. за создание, удаление и замену элементов согласно правилу. Создание алгоритма поиска подграфа в графе и корректное размещение новых элементов в модели являются основными подзадачами, которые были решены при реализации данного модуля.

5.1.1 Поиск подграфа

5.1.1.1 Другие решения

GRaphs for Object-Oriented VErification, GROOVE[1] [39], является открытым программным средством, предназначенном для применения преобразований графов и автоматической верификации моделей относительно итоговой системы преобразований.

GROOVE является свободным проектом, распространяемым по лицензии Apache License v2.0 и реализованным на языке Java. В нём присутствует вся необходимая функциональность по работе с графами, необходимая для реализации предложенного подхода по заданию семантик визуальных языков, но его использование привнесёт в систему QReal дополнительную зависимость от сторонних программных продуктов, что крайне нежелательно.

Также при использовании GROOVE придётся создавать специальные довольно сложные модули, позволяющие переводить формат как интерпретируемой модели, так и самих правил преобразования графов из поддерживаемого в системе QReal в поддерживаемый GROOVE и наоборот. Подробнее о необходимых преобразованиях форматов моделей можно прочесть в работе [16]. Вдобавок может возникнуть проблема совместимости между кодом на Java в GROOVE и кодом на C++/Qt в QReal.

По аналогичным причинам невозможно и использование другого открытого средства для осуществления и визуализации преобразований графов, которое называется GenGED[2] [17]. Так же, как и GROOVE, GenGED реализован на языке Java, предназначен для свободного и некоммерческого использования. Главным его недостатком является отсутствие

[1] GROOVE, http://groove.sourceforge.net/groove-index.html

[2] GenGED, URL: http://user.cs.tu-berlin.de/~genged

кроссплатформенности (работает только под Linux и Solaris, потому что использует средство разрешения ограничений ParCon [23]).

5.1.1.2 Алгоритм поиска шаблона в модели в средстве MetaLanguage

В системе MetaLanguage присутствует возможность осуществлять трансформации моделей при помощи правил преобразования графов. Для этого был предложен специальный алгоритм, описанный в статье [46]. Он состоит из трёх этапов и исходит из предположения, что обычно в рассматриваемых моделях рёбер меньше, чем вершин, что даёт возможность сопоставить каждому ребру свою вершину.

Таким образом, алгоритм начинается с поиска всех вхождений в исходной модели произвольной связи из шаблона. Затем, начиная с каждой найденной связи, происходит рекурсивный поиск всех остальных связей из шаблона. В конце по найденным связям проверяются и добавляются в граф, соответствующий шаблону, вершины, инцидентные им.

В отличие от данного решения, алгоритм, реализованный в системе QReal, позволяет искать шаблон с произвольным числом связей внутри. Он начинается с вершины, накапливает и расширяет подграф шаблона, совпадающий с некоторым подграфом модели, добавляя в него по одной вершине за один шаг, т.е. на каждом шаге имеется корректный найденный подграф, а в алгоритме системы MetaLanguage проверка корректности вершин происходит в самом конце.

5.1.1.3 Реализованный алгоритм

В итоге для организации поиска заданного шаблона в модели был предложен следующий итеративно-рекурсивный алгоритм, накопление результата в котором будет происходить постепенно. Данный алгоритм находит все вхождения этого шаблона в модели, промежуточным результатом работы будет одно найденное вхождение.

Рассмотрим набор множеств и отображений, необходимых для работы этого алгоритма. Пусть исходная модель является графом $G=(V,E)$, где V — множество его вершин, E — множество рёбер, а искомый шаблон — граф $G'=(V',E')$. В дальнейшем будем придерживаться соглашения, что обозначения с штрихом относятся к вещам, связанным с шаблоном, а без — с исходной моделью. Тогда:

- Отображение $\Phi:G' \to G$ будет определяться постепенно и на определённых шагах будет содержать промежуточный результат работы алгоритма. Взятое на элементе G', оно выдаёт соответствующий ему элемент G.
- Подмножество вершин V', образующих подграф G', который на данном шаге совпадает с некоторым подграфом G, обозначим за H'.
- Подмножество вершин H', у которых есть связи с вершинами из V', не лежащими в H', обозначим за W'. Данное множество представлено в виде списка, поэтому имеет значение, в каком порядке туда будут добавляться элементы.
- Индекс вершины в W', рассматриваемой на текущем шаге, обозначим за i. В начале работы проинициализирован нулём.

Таким образом, на каждом шаге имеется некий подграф с множеством вершин, равным H', в G', который уже был найден в G. Отображение вершин и рёбер этого подграфа на элементы G уже содержится в Φ. Также присутствует список вершин W', который позволяет продолжать поиск G' в G.

Сразу отметим, что, т.к. граф модели является типизированным ориентированным графом с атрибутами, то при сравнении вершин учитывается равенство их типов и атрибутов, а при сравнении рёбер ещё и их направления. Теперь рассмотрим принцип работы алгоритма целиком.

На первом шаге выбираем произвольную вершину из V' и ищем все вершины в V, совпадающие с ней по критериям, которые были описаны выше. Запускаем цикл, проходящий по ним и при проходе каждого делающий следующие операции: обнуление i, очищение H' и W' от старых данных и их заполнение информацией про соответствующие первые вершины.

Второй шаг данного алгоритма представляет собой рекурсивную функцию, возвращающую булевское значение. В её теле происходит следующее. Берём вершину из списка W' по индексу i, находим для неё первое ребро e' из E', не ведущее в H', т.е. вторая вершина данного ребра не должна лежать в H'. Если такого ребра не нашлось, то увеличиваем индекс i на 1 и запускаем второй шаг снова, если i не равно мощности W'. Если же равно, то, значит, искомый граф найден целиком и в Φ будет содержаться промежуточный результат, т.е. одно найденное вхождение данного графа. Добавляем его в список найденных подграфов и возвращаем истину, как результат работы второго шага.

Обозначим вершину, выбранную из W', за x', а вершину, с которой e' соединено помимо x', за a'. Вершину $\Phi(x')$ обозначим за b. Ищем у b все ребра, совпадающие с e' и не ведущие в $\Phi(H')$. Обозначим полученное множество рёбер за E_1. Если таких рёбер не нашлось, то совершаем откат и запускаем второй шаг снова.

Запомним текущее состояние рассматриваемых множеств и отображений. После этого для каждого ребра из E_1 делаем следующее. Обозначим вершину, с которой оно соединено, за c. Ищем все рёбра в E', имеющие одним из своих концов a', а вторым — вершину из H', и проверяем, есть ли такие же в E из c. Если какого-то не нашлось, то переходим к следующему ребру из E_1. Если нашлись все нужные рёбра, то $\Phi(a') = c$, а также $\forall r' \in E'$, один из концов r' равен a', а второй $\in H' : \Phi(r') = r \in E$, где r является найденным выше соответствующим ребром. Запускаем второй шаг снова, если индекс i не равен мощности W'. Если второй шаг вернул истину, т.е. мы нашли нужный подграф, то откатываемся к запомненному состоянию и переходим к следующему ребру из E_1.

Откат происходит по следующему принципу. Убираем последнюю добавленную вершину в H', убираем последнюю вершину в списке W', уменьшив при необходимости индекс i на 1, убираем информацию об удаленной вершине в Φ (как про саму вершину, так и про все его рёбра в H').

Нахождение заданного шаблона в графе позволяет в дальнейшем легко производить операции создания новых элементов (как вершин, так и рёбер), их корректного соединения с уже существующими элементами модели, удаления или замены старых элементов.

5.1.2 Изменение модели согласно правилу

Второй важной подзадачей в модуле работы с графами является непосредственное изменение модели согласно правилу после нахождения места применения этого правила. Оно осуществляется при помощи стандартного API, предоставляемого QReal, но для корректного его исполнения нужно сделать пару дополнительных шагов.

При создании и замене элементов необходимо учитывать следующие моменты. Во-первых, нужно скопировать значение атрибутов соответствующих элементов из правила созданному элементу в модели. Также этот новый созданный элемент нужно добавить в соответствие элементов правила элементам модели, чтобы его можно было использовать при интерпретации реакции на применение правила. Во-вторых, при замене элемента нужно правильно переподсоединить все связи, одним из концов которых был заменяемый элемент.

Также возникает вопрос и при удалении элементов. Нужно ли удалять все связи элемента при его удалении с диаграммы? На данный момент такие связи автоматически не удаляются, в идеале же это можно сделать настраиваемым.

Самой главной сложностью корректного преобразования модели согласно правилу является автоматическая раскладка элементов на диаграмме после создания, замены и удаления элементов. Это происходит из-за того, что новые элементы не должны перекрывать старые, т.к. от этого теряется информативность и читаемость диаграммы.

Удаление и замену элементов можно производить практически всегда без перераскладки элементов (добавленный элемент имеет те же координаты, что и заменяемый). Создание же нового элемента требует расчёта его места на диаграмме. В качестве решения этой проблемы можно запускать средства автоматической раскладки после каждого такого создания. Подробнее о нём можно прочитать в следующем разделе. Сейчас новые элементы просто отрисовываются правее всех остальных на диаграмме.

5.2 Интерпретация текстовой составляющей задания семантики

Помимо графической составляющей семантика визуального языка содержит ещё и не менее важную текстовую часть, которая состоит из трёх частей: инициализация интерпретации, ограничение на применение правила и реакция на применение правила. Также к ней относится и поведение элементов исходного языка, но оно определяется не в редакторе семантики самого визуального языка.

В качестве текстового языка для записи этих частей можно использовать различные интерпретируемые языки: C-подобный язык редактора блок-схем, в котором есть можно использовать переменные, арифметико-логические выражения, различные математические функции и присваивание, а также Python и QtScript. Данный язык задается вручную выбором из списка поддерживаемых для каждого правила по отдельности. Таким образом можно

комбинировать разные языки при задании одной семантики в зависимости от сложности необходимого функционала.

Основной выбор при реализации компоненты, отвечающей за исполнение текстовой части семантики, заключался в количестве экземпляров интерпретаторов выбранного текстового языка при непосредственном исполнении модели. Можно создавать один экземпляр на каждое применение правила преобразования графов, тогда невозможно будет осуществлять никакого взаимодействия между правилами, при этом снижается производительность интерпретации, т.к. каждый раз нужно создавать нетривиальный объект интерпретатора.

Поэтому при исполнении модели создается ровно один экземпляр интерпретатора QtScript или текстового языка для языка блок-схем или же один раз запускается процесс интерпретатора языка Python. Таким образом, все объявленные переменные и функции доступны для вызова, чтения и записи при исполнении из любого правила, их существование поддерживается до конца исполнения.

Важно отметить, что экземпляр интерпретатора QtScript создаётся в пространстве работающей системы, таким образом никакого взаимодействия между процессами, как в случае с Python, не происходит. По этому же поводу сгенерированные на QtScript скрипты не сохраняются на диск, а всегда представлены в виде текстовых строк.

5.3 Особенности используемых текстовых языков

Для всех поддерживаемых тектовых языков необходимо было решить проблему доступа к значениям атрибутов, а также проблему исполенения поведения элемента, т.е. вызова значения некторого атрибута как функции. Рассмотрим более подробно, какими конструкциями были расширены соответствующие языки, чтобы обеспечить возможность такого доступа и вызова. Обозначим за *e* и *a* имя элемента и имя атрибута этого элемента, с которыми мы хотим совершить некоторую операцию, и будем придерживаться этого обозначения в следующих параграфах.

5.3.1 Язык блок-схем

Общая функциональность языка блок-схем была описана выше. Для обеспечения возможности доступа к атрибутам и т.п. в текстовой составляющей семантики языка можно использовать его расширенную версию.

- *e.a* — получить доступ к значению атрибута. Может использоваться в выражениях и в левой части присваивания.
- *run(e.a)* — выполнить код, записанный в соответствующем атрибуте.
- *cond(e.a)* — посчитать значение логического выражения, записанного в атрибуте.
- Также в правой части присваивания может быть *null* для деинициализации соответствующего атрибута или логическая коснтанта.

5.3.2 Python

Аналогично языку блок-схем, в коде на языке Python в текстовой составляющей семантики тоже можно использовать некоторые дополнительные конструкции, описанные ниже.

- *e.a* — получить доступ как на чтение, так и на запись к атрибуту элемента.
- *e.a()* — использовать значение атрибута элемента в качестве функции.
- *e@a* — подстановка значения атрибута элемента в генерируемый скрипт реакции как есть (аналог inline-функций в C++).

Для того, чтобы обеспечить поддержку введённых конструкций, был реализован специальный генератор скриптов ограничений и реакций на применение правил, который бы транслировал эти конструкции в совпадающий по смыслу код на языке Python. В результате, *e@a* просто заменяется на соответствующее значение атрибута, а конструкции вида e.a и e.a() заменяются на переменную *e_visint_a* и вызов функции *e_visint_a()*, инициализация начальным значением и объявление которых автоматически добавляется в шапку скрипта. В конец же скрипта добавляется вывод нового значения переменной *e_visint_a*, чтобы после его выполнения можно было изменить значения атрибутов элементов исходной модели.

Чтобы обеспечить синхронизацию интерпретатора визуальных языков и интерпретатора языка Python, был зафиксирован формат вывода исполняемых скриптов. Ограничение на применение правила выводит просто “true” или “false”, а реакция на применение правила — последовательность вида “*{element_name}.{attribute_name} = {new_value};*”. Она легко разбирается главным модулем, и происходит соответствующее ей изменение значений атрибутов. Об ошибках при исполнение скриптов также сообщается главному модулю, при этом интерпретация модели останавливается, а вывод интерпретатора языка Python печатается в специальном окне.

Ещё одной задачей для обеспечения удобства использования языка Python в качестве текстовой составляющей семантики визуального языка явилось подключение к системе QReal специального встроенного текстового редактора, предоставляемого свободной библиотекой QScintilla. Окно с редактором автоматически открывается при попытке изменения инициализации интерпретации, ограничения или реакции на применение правила. Данный редактор предоставляет широкий спектр возможностей, повышающих удобство написания кода на языке Python, такие, как: подсветка синтаксиса, автодополнение, автоматическая табуляция, показ номеров строк и отображение пробелов и т.д.

5.3.3 QtScript

Дополнительные конструкции, которые можно использовать в коде на языке QtScript полностью повторяют конструкции из языка Python. Реализация их поддержки (замены соответветствующих конструкций и формата вывода результатов и ошибок исполняемых скриптов) также полностью совпадает с реализацией для языка Python. Отличием является формат объявления новых функций и переменных в шапке скрипта, а также способ вывода результатов средствами соответствующего интерпретируемого текстового языка.

6 Апробация

6.1 Связь с рефакторингами моделей

Основа предложенного подхода и реализованного средства непосредственно связана с работой Анастасии Кузенковой [5], посвящённой созданию средств рефакторинга моделей в системе QReal.

6.1.1 Рефакторинг как преобразование графов

Рефакторинг модели представляет из себя поиск некоего шаблона в рассматриваемой модели с последующим правильным его преобразованием. Например, изменение направления связей между элементами, изменение имени элементов, вырезка группы элементов модели в отдельную диаграмму и её замена на один блок в исходной.

В общем же случае преобразованием является изменение значений атрибутов элементов в найденном шаблоне, а также изменение направлений связей, замена, удаление и создание новых элементов. Данный набор действий является базой системы преобразований графов, рассмотренной ранее. Поэтому работа над реализацией модуля преобразования графов велась совместно.

6.1.2 Проблема графического расположения элементов

Поскольку при применении правил преобразования графов происходит изменение исходной модели и её графического представления на экране, необходимо учесть проблемы, связанные с отображением добавленных элементов, т.к. при удалении и замене расположение остальных элементов можно не менять.

В работе Анастасии для решения данной задачи было предложено использовать программу dot пакета утилит по автоматической визуализации графов Graphviz[1], и с его помощью реализовано средство автоматической раскладки элементов модели на экране. Впоследствии была добавлена поддержка большего количества программ и алгоритмов автораскладки из пакета Graphviz.

6.2 Исполнимая семантика языка блок-схем

В рамках работы в систему QReal был добавлен визуальный редактор языка блок-схем и с использованием предложенного подхода была задана его исполнимая семантика. Посмотрим более подробно на её детали. Семантику языка блок-схем можно разделить на 3 части. Первая отвечает за инициализацию процесса исполнения, вторая — за непосредственный старт визуализации интерпретации, а также за её корректное завершение. Третья же отвечает за выполнение действий, записанных в элементе Action и за правильную проверку условия в элементе Condition.

Общий принцип исполнения блок-схем можно описать следующим образом. Вначале инициализируем исполнение, записав 0 в переменную логического типа interpretationStarted.

[1] Graph Visualization Software, http://www.graphviz.org/

Далее начинаем визуализацию исполнения, поместив на элемент InititalNode маркер потока исполнения и записав в interpretationStarted единицу. После этого передаём исполнение следующему элементу. Для завершения интерпретации просто удаляем маркер потока исполнения с элемента FinalNode.

Если попали на элемент Action, то исполняем действия внутри него и переходим к следующему элементу. Если же попали на элемент Condition, то дальнейший переход осуществляем по той связи, тип которой совпадает со значением условия внутри рассматриваемого элемента Condition.

На рис. 6 изображена полная семантика для языка блок-схем, записанная при помощи предложенного подхода.

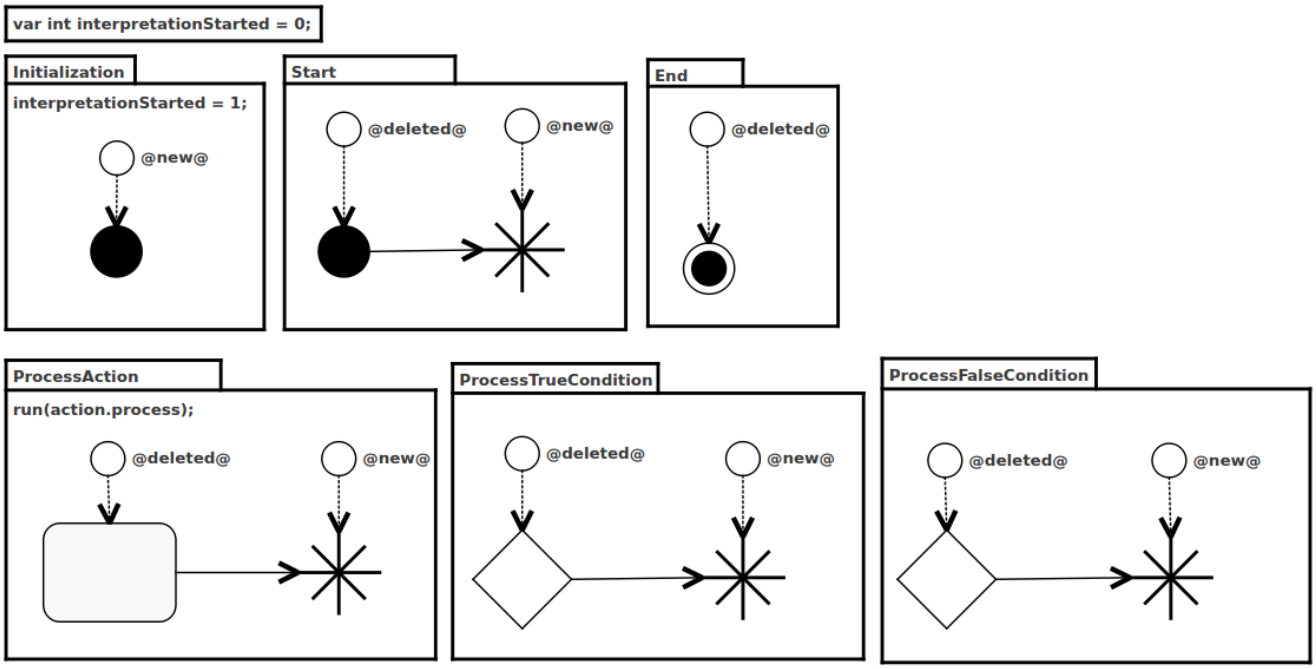

Рис. 6. Исполнимая семантика языка блок-схем

Рассмотрим подробнее данный рисунок. На нем изображено 6 семантических правил, у каждого из которых есть имя, которое будет выводиться на экран при его применении. Правила, соответствующие второй и третьей частям семантики, описанным ранее, расположены в одном ряду для каждой. Инициализацией исполнения является код, написанный в верхнем элементе диаграммы.

Маленький кружок и точечная связь с ним означают текущее положение потока исполнения. Звезда обозначает унификатор узла, т.е. его сравнение с любым элементом-узлом модели возвращает истину. Директивы @deleted@ и @new@ означают удаление и создание соответствующего элемента при применении правила. Код, написанный в левом верхнем углу любого правила, является реакцией на применение правила, через него можно исполнять действия, записанные внутри элементов исходной модели, проверять различные условия, а также считывать и изменять значения атрибутов элементов.

Также стоит отметить, что в правилах Initialization, ProcessTrueCondition и ProcessFalseCondition используются ограничения на применение правил. Initialization можно

применить тогда и только тогда, когда interpretationStarted равна 0, чтобы исполнение началось ровно 1 раз. А ProcessTrueCondition и ProcessFalseCondition должны быть применены тогда и только тогда, когда значение условия в элементе Condition равно типу связи, что и записано в ограничении на применение правила.

6.3 Использование в редакторе алгоритмов работы роботов Lego Mindstorms NXT

Данная работа использовалась при создании редактора алгоритмов работы роботов Lego Mindstorms NXT[1] [6]. Для него была реализована пошаговая интерпретация, притом команды роботу посылаются по bluetooth прямо во время отладки и можно вживую наблюдать за его действиями. При реализации использовался дополненный парсер арифметико-логических выражений для C-подобного текстового языка, использующегося в блок-схемах: в него были добавлены логарифмические и тригонометрические функции.

В рамках применения предложенного подхода была реализована семантика генерации кода по диаграммам поведения роботов. Она позволяет получать код на C++ в нотации nxtOSEK, поддерживает генерацию кода по отдельным элементам редактора, а также условий и циклов. В качестве текстового языка в ней выбран язык Python, активно используются заданные при инициализации переменные и функции, а также ограничения и реакции на применение правил. Непосредственная генерация и запись получившегося кода происходит именно в реакции. Особенностью же данной семантики является использование приоритета у правил, чтобы увеличить производительность исполнения, т.к., например, при переходе к следующему элементу совершенно необязательно пытаться применить правила, связанные с генерацией кода элемента определённого типа.

6.4 Публикации и доклады

По результатам проведённых исследований и работ был сделан ряд публикаций и докладов. Доклады осуществлялись на следующих конференциях: межвузовском конкурсе-конференции студентов, аспирантов и молодых ученых Северо-Запада “Технологии Microsoft в теории и практике программирования” [11, 12] 2011 и 2013 года и на Всероссийской научной конференции по проблемам информатики “СПИСОК-2012” [8]. Кроме публикаций в материалах данных конференций, была опубликована статья в ежегодном сборнике статей “Системное программирование” [9]. Также на момент написания дипломной работы были приняты к публикации и готовятся к печати: статья в журнале, рецензируемом ВАК [10], и статья в материалах французской конференции ENASE’13 (Evaluation of Novel Approaches to Software Engineerin) [30].

[1] Lego Mindstorms NXT, http://mindstorms.lego.com

7 Сравнение и соотнесение предложенного и существующих подходов

Основные сходства и различия разобранных способов задания семантики интерпретации визуальных языков можно увидеть в табл. 1. Сравнение подходов осуществлялось по шкале от 0 до 3 по следующим критериям.

Универсальность — это возможность задания семантики для произвольных визуальных языков, т.е. оценка того, насколько подход подходит под концепцию метамоделирования. Двухуровневая отладка совсем не соответствует концепции метамоделирования и DSM, т.к. данный подход ориентирован на создание единичного отладчика для определённого языка и не подразумевает удобного обобщения (0 баллов), UML и анимационный подход работают так или иначе только с диаграммами UML (0 баллов), в то время как остальные способы не зависят от входного визуального языка (3 балла).

Задание семантики на основе хорошо определённого (математически) подхода добавляет программам доказуемость и предсказуемость поведения, т.е. при таком задании невозможна различная интерпретация спецификации, которая бы присутствовала, например, в большом текстовом описании. За это отвечает *критерий формальности*. DMM, AToM^3 и анимационный подходы основаны на технологии преобразования графов (3 балла), EProvide поддерживает задание семантики, основанной на стандарте QVT Relation, а xUML базируется на PAS, которая была стандартизирована Object Management Group в 2001 году (1 балл, так как основаны на стандартах).

Наглядность средств задания семантики определяется тем, визуально (3 балла) ли это нужно делать или при помощи текста (0 баллов). Подходы, основанные на преобразованиях графов, т.е. DMM, AToM^3 и анимационный, являются большей частью визуальными, а остальные — текстовыми. xUML, благодаря своей визуальной составляющей в виде диаграмм состояний жизненных циклов объектов, мы оцениваем в 1 балл.

Наглядность процесса интерпретации зависит от того, как эта интерпретация будет отображаться пользователю. Самым наглядным из рассмотренных подходов является анимационный подход, т.к. там пользователю представляются реальные картинки, объединённые в анимацию (3 балла). DMM, EProvide и AToM^3 будут по ходу исполнения изменять модель, а также при необходимости подсвечивать отдельные элементы, как, например, в EProvide, поэтому по этому критерию мы оцениваем их все в 2 балла. При использовании xUML или двухуровневой отладки интерпретация позволит лишь подсвечивать текущий исполняемый элемент (1 балл).

Если смотреть на способы задания семантики со стороны разработчика конкретного визуального языка, то важным критерием их сравнения становится *понятность*, отвечающая за количество знаний в области визуального моделирования и других связанных с ней областях, которое необходимо иметь для успешного и осмысленного применения данных способов. На наш взгляд, самым понятным подходом является DMM, т.к. правила преобразования графов воспринимаются интуитивно. У AToM^3 и анимационного подхода есть недостаток в том, что возможно ставить различные ограничения и писать исполняемые инструкции на OCL и Python, а это требует от разработчика дополнительных знаний. Для анимационного подхода также

нужно частичное знание UML, поэтому мы оцениваем его в 1 балл, а AToM^3 — в 2 балла. xUML подразумевает сложную систему зависимостей диаграмм друг от друга, а также активную работу с AL, что снижает понятность, т.к. необходимо хорошо ориентироваться в UML и в PAS или некотором AL (0 баллов). Для использования EProvide нужно изучить стандарт QVT Relation, что по сложности сравнимо с подходом в xUML (0 баллов). Для применения двухуровневой отладки же необходимо знание языка программирования, на котором написана использующаяся система, а также особенностей её реализации для того, чтобы было возможно встраивание в неё функционала отладчиков, что даёт данному подходу оценку в 0 баллов.

Критерий наличия реализации в комментариях не требуется (для xUML их существует много). 3 балла, если реализация существует, 0 баллов, если нет. Двухуровневая отладка имеет 1 балл, т.к. для неё присутствует реализация для конкретного языка. Если для данного подхода уже реализовано средство, позволяющее не только интерпретировать, но и отлаживать, т.е. использовать различные точки останова, просмотр и изменение значений атрибутов и т.п., или для подхода возможно такое расширение, то он имеет 3 балла в графе "*возможность создания отладчика*", иначе — 0 баллов. Как видно из обзора, анимационный подход не подразумевает отладки напрямую, т.к. пользователь видит лишь анимацию интерпретации.

Критерий сравнения	Двухур. отладка	xUML	DMM	EProvide	AToM^3	Анимация	QReal
Универсальность	0	0	3	3	3	0	3
Формальность	0	1	3	1	3	3	3
Наглядность средств задания семантики	0	1	3	0	3	3	3
Наглядность процесса интерпретации	1	1	2	2	2	3	2
Понятность	0	0	3	0	2	1	2
Наличие реализации	1	3	0	3	3	3	3
Возможность создания отладчика	3	3	3	3	3	0	3

Табл. 1. Сравнительная таблица для разобранных подходов

После суммирования полученных для каждого из подходов баллов получается следующее распределение: двухуровневая отладка — 5 баллов, xUML — 9 баллов, EProvide — 12 баллов, анимационный подход — 13 баллов, DMM — 17 баллов, AToM^3 — 19 баллов. Благодаря наличию реализации, подход, использующийся в AToM^3, мы считаем лучшим подходом. Он очень похож на DMM, что подтверждают набранные ими баллы.

Предложенный и реализованный в системе QReal подход набирает 19 баллов, потому что он является наследником DMM-подхода и считается реализованным. Таким образом, у него столько же баллов, сколько и у AToM^3. Главным отличием от AToM^3 является тот факт, что QReal — развивающийся кроссплатформенный проект с удобным интерфейсом, в отличие от системы AToM^3, которая имеет “научный” интерфейс и последнее обновление которой было в январе 2008-го года.

Заключение

Результаты

В рамках данной дипломной работы были получены следующие результаты:

1. Изучен контекст модельно-ориентированной разработки и предметно-ориентированного моделирования, исследованы следующие основные подходы к заданию семантики визуальных языков: двухуровневая схема отладки, анимированная интерпретация диаграмм, DMM, подходы, использующиеся в xUML, AToM^3, EProvide. Произведён подробный обзор, анализ и сравнение этих подходов, выявлены их сильные и слабые стороны, также была оценена степень их применимости к системе QReal.
2. На основе сравнения выбранных походов было сформулировано собственное решение, применимое к системе QReal и включающее в себя положительные черты разобранных методов.
3. Предложенное решение было реализовано на базе проекта QReal. Были разработаны средства задания исполнимой семантики выбранного класса визуальных языков, а также был разработан механизм интерпретации пользовательских моделей по заданной семантике.
4. В качестве тестирования реализованных средств была задана исполнимая семантика для языка блок-схем, а также создана семантика генерации исполнимого кода по моделям в редакторе алгоритмов работы роботов Lego Mindstorms NXT.

Дальнейшее развитие

В будущем планируется развивать работу по следующим направлениям: увеличение функциональности редактора семантики визуального языка и улучшение механизма интерпретации/отладки модели по заданной семантике.

В первое направление входит, например, добавление в редактор семантики унификатора связи, при сравнении которого с любой связью исходной модели получается истина. Также можно добавить более сложные алгоритмы приоритезации правил семантики. Они будут представляться в виде функций на значениях атрибутов элементов правила и, таким образом, будут зависеть от конкретного места применения правила в интерпретируемой модели. Согласно DMM-подходу, в редактор можно добавить поддержку NAC и UQS, при том UQS как для узлов, так и для связей. Необходимо ввести возможность задавать агрегированные функции на атрибутах элементов таких структур. Например, посчитать сумму значений некоторого атрибута для всех элементов указанного типа, связанных с фиксированным узлом.

В рамках улучшения механизма отладки модели можно выделить создание окна просмотра текущих значений переменных в системе при интерпретации, а также повышение интерактивности пользовательского интерфейса. Например, реализация окна выбора применяемого правила на очередном шаге, если количество правил, которые можно применить, или количество мест для применения одного правила больше единицы. Также важным моментом является реализация функциональности точек останова различных видов (подробнее с их видами можно ознакомиться в работе [16]).

Список литературы

1. *Брыксин Т.А., Литвинов Ю.В.* Технология визуального предметно-ориентированного проектирования и разработки ПО QReal // Материалы второй научно-технической конференции молодых специалистов «Старт в будущее», посвященной 50-летию полета Ю.А. Гагарина в космос. СПб. 2011. С. 222-225.
2. *Гуров В., Мазин М., Шалыто А.* UniMod - инструментальное средство для автоматного программирования // Научно-технический вестник информационных технологий, механики и оптики. 2006. № 30. С. 32-45.
3. *Карташев М.* Двухуровневая схема отладки // Системное программирование. Под ред. А. Н. Терехова, Д. Ю. Булычева. Изд-во СПбГУ, 2004. С. 348-365.
4. *Кознов Д.В.* Разработка и сопровождение DSM-решений на основе MSF∗ // Системное программирование. Под ред. А. Н. Терехова, Д. Ю. Булычева. Изд-во СПбГУ, 2008. Т. 3. № 1. С. 80-96.
5. *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. СПб.: Изд-во ВВМ. 2012. С. 24-33.
6. *Кузенкова А.С., Литвинов Ю.В., Брыксин Т.А.* Метамоделирование: 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. СПб. 2011. С. 228-231.
7. *Павлинов А., Кознов Д., Перегудов А. и др.* О средствах разработки проблемно-ориентированных визуальных языков // Системное программирование / Вып. 2, под ред. А. Н. Терехова и Д. Ю. Булычева. СПб.: Изд. СПбГУ, 2006. С. 116-141.
8. *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. СПб.: Изд-во ВВМ. 2012. С. 56-61.
9. *Поляков В.А., Брыксин Т.А.* Подходы к заданию семантики интерпретации диаграмм в рамках DSM-подхода // Системное программирование / Вып. 7, под ред. А. Н. Терехова и Д. Ю. Булычева. СПб.: Изд. СПбГУ. 2012. С. 156-183.
10. *Поляков В.А., Брыксин Т.А.* Подходы к заданию семантики интерпретации диаграмм, основанные на технологии преобразования графов // Компьютерные инструменты в образовании. 2013, готовится к печати.
11. *Поляков В.А., Брыксин Т.А.* Разработка визуального интерпретатора моделей в системе QReal // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада "Технологии Microsoft в теории и практике программирования". СПб.: Изд-во СПбГПУ. 2011. С. 58.
12. *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". СПб.: Изд-во СПбГПУ. 2013. С. 80-81.

13. *Сорокин А., Кознов Д.* Обзор Eclipse Modeling Project // Системное программирование / Вып. 5, под ред. А. Н. Терехова и Д. Ю. Булычева. СПб.: Изд. СПбГУ, 2010. С. 6-31.
14. *Сухов А.О., Серый А.П.* Теория графов и приложения. Graphs Theory and Applications // Изд-во Уральского университета, Екатеринбург, 2012. С. 48—55.
15. *Amyot D., Farah H., Roy J.-F.* Evaluation of Development Tools for Domain-Specific Modeling Languages // 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Springer Berlin Heidelberg, P. 183-197.
16. *Bandener N.* Visual interpreter and debugger for dynamic models based on the Eclipse platform. Diploma Thesis, 2009, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn. 125 p.
17. *Bardohl R., Ermel C., Weinhold l.* GenGED – A Visual Definition Tool for Visual Modeling Environments. Applications of Graph Transformations with Industrial Relevance, Lecture Notes in Computer Science, Volume 3062, 2004, P. 413-419.
18. *Borger E., Stark R.* Abstract State Machines: A Method for High-Level System Design and Analysis // Springer-Verlag, 2003. 35 p.
19. *Boyd G.* Executable UML: Diagrams for the Future. 2003. URL: http://www.devx.com/enterprise/Article/10717
20. *Ehrig H., Ermel C., Taentzer G.* Simulation and Animation of Visual Models of Embedded Systems // Embedded Systems – Modeling, Technology, and Applications, Springer, 2006, P. 11-20.
21. *Ermel C., Ehrig K.* View Transformation in Visual Environments applied to Algebraic High-Level Nets // Electronic Notes in Theoretical Computer Science, Vol. 127, Issue 2, 2005, P. 61–86.
22. *Ermel C., Holscher K., Kuske S., Ziemann P.* Animated Simulation of Integrated UML Behavioral Models based on Graph Transformation // IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), 2005, P. 125-133.
23. *Griebel P.* Parcon – Paralleles Losen von grafischen Constraints. PhD thesis, 1996, Paderborn, University of Paderborn.
24. *Hausmann J.* Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD Thesis, 2005, Paderborn, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn. 326 p.
25. *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM, 12(10):576–580, 1969, P. 576-583.
26. *Isazadeh H.* Architectural Analysis of MetaCASE. A Study of Capabilities and Advances. Thesis for the degree of Master of Science. URL: www.collectionscanada.gc.ca/obj/s4/f2/dsk2/ftp04/mq20654.pdf
27. *Jézéquel J.-M., Barais O., Fleurey F.* Model Driven Language Engineering with Kermeta // Generative and Transformational Techniques in Software Engineering III Lecture, Notes in Computer Science, Volume 6491, 2011, P. 201-221.
28. *Kahn G.* Natural Semantics // Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, London, 1987, P. 22-39.

29. *Kelly S., Tolvanen J.* Domain-Specific Modeling: Enabling Full Code Generation // Wiley-IEEE Computer Society Press, 2008. 448 p.

30. *Kuzenkova A., Deripaska A., Bryksin T., Litvinov Y., Polyakov V.* QReal DSM platform. An environment for creation of specific visual IDEs // Proceedings of ENASE'13, 2013, готовится к печати

31. *Lara J., Vangheluwe H.* AToM^3 : a tool for multi-formalism modelling and meta-modelling // Proceedings of ETAPS/FASE'02, Lecture Notes in Computer Science, Vol. 2306, Springer, Berlin, 2002, P. 174–188.

32. *Lara J., Vangheluwe H.* Defining visual notations and their manipulation through meta-modelling and graph transformation // Journal of Visual Languages and Computing 15, 2004, P. 309–330.

33. *Mathew A.* Software Development Using Executable UML (xUML). 2002. URL: http://se.cs.depaul.edu/ise/zoom/projects/statechart/SE690DetailedPresentation.ppt

34. *Mellor S., Balcer M.* Executable UML: A foundation for model-driven architecture // Addison Wesley, 2002. 416 p.

35. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.0. OMG. 2008.

36. Object Action Language Reference Manual. 2008. 77 p. URL: www.ooatool.com/docs/OAL08.pdf

37. OMG Unified Modeling Language, Superstructure. Version 2.4.1. OMG. 2011. 732 p.

38. *Plotkin G.* A Structural Approach to Operational Semantics // Technical Report DAIMI FN-19, University of Aarhus, 1981. 133 p.

39. *Rensink A.* The GROOVE Simulator: A Tool for State Space Generation // AGTIVE 2003 – Revised Selected and Invited Papers, volume 3062 of Lecture Notes in Computer Science, Berlin, Springer Verlag, 2004, P. 479–485.

40. *Rozenberg G.* Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific, 1997.

41. *Sadilek D., Wachsmuth G.* Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages // ECMDA-FA 2008, LNCS 5095, 2008, P. 64–79.

42. *Scott D., Strachey C.* Towards a Mathematical Semantics for Computer Languages // Computers and Automata, Wiley, 1971, P. 19–46.

43. *Sendall S., Kuster J.* Taming Model Round-Trip Engineering // Proc. Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming , Systems, Languages, Applications). 2004. 13 p.

44. Shlaer-Mellor Action Language. 1997. 29 p. URL: www.modelint.com/downloads/small.pdf

45. *Slonneger K., Kurtz B.* Syntax and Semantics of Programming Languages, A Laboratory Based Approach // Addison-Wesley Publishing, 1995. P.187-222.

46. *Sukhov A.O., Lyadova L.N.* Horizontal Transformations of Visual Models in MetaLanguage System // Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering. SYRCoSE, 2013. 9 p.

47. *Sukhov A.O., Lyadova L.N.* MetaLanguage: a Tool for Creating Visual Domain-Specific Modeling Languages // Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering. SYRCoSE, 2012. 12 p.
48. *Wachsmuth G.* Modelling the Operational Semantics of Domain-Specific Modelling Languages // Generative and Transformational Techniques in Software Engineering II. Lecture Notes in Computer Science Volume 5235, Springer, 2008, P. 506-520.
49. *Warmer J., Kleppe A.* The Object Constraint Language: Precise Modeling with UML // Addison- Wesley Object Technology Services, Reading, MA, 1999. 144 p.