Санкт-Петербургский государственный университет

На правах рукописи

УДК 004.4’22

Брыксин Тимофей Александрович

Платформа для создания специализированных визуальных сред разработки программного обеспечения

Специальность 05.13.11 —
Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

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

Санкт-Петербург
2016

Оглавление

Введение

С появлением первых языков программирования стали также развиваться инструменты, упрощающие процесс создания программных систем и повышающие его эффективность. В настоящее время интегрированные среды разработки (integrated development environments, IDE) являются многофункциональными инструментальными системами, которые позволяют освободить разработчиков от многих рутинных действий, в частности, снижая порог вхождения разработчиков в программные проекты на новых языках. В конце XX века получили популярность визуальные языки проектирования ПО. Считается, что человек гораздо лучше воспринимает графические диаграммы, чем большие объёмы текста, а значит, переход от текстового программирования к визуальному можно рассматривать как следующий шаг, позволяющий сделать процесс разработки ПО более наглядным и удобным для людей.

В 90-е годы XX века основной упор в этом направлении делался на языки общего назначения (такие, как UML [156]), однако практика показала, что модели, создаваемые с использованием таких языков, получаются чрезвычайно громоздкими. В последние годы активно развиваются идеи визуального предметно-ориентированного моделирования (domain-specific modeling, DSM [102]), в основе которого лежит идея создания специализированных языков под конкретные задачи. Это позволяет существенно поднять уровень абстракции создаваемых моделей, перенося разработку с уровня программных конструкций типа ветвлений и циклов в область терминов предметной области. Разработчик взаимодействует только с наглядными и понятными визуальными моделями, а код разрабатываемой системы полностью генерируется автоматически по этим моделям. Такой подход хорошо себя зарекомендовал в случаях, когда есть серия похожих задач и хочется переиспользовать полученные знания, однако, практика показывает, что и для

одиночных средних и крупных по размеру задач такой подход также имеет право на существование.

Для того, чтобы данный подход к разработке ПО был экономически оправдан, необходимо уметь быстро создавать визуальные языки и инструментальные средства для них — так называемые предметно-ориентированные решения. При этом речь идет не только о графическом редакторе, но и о наборе генераторов (генераторы исходных кодов, документации, скриптов сборки и размещения целевой системы и т.д.), репозитории для хранения создаваемых моделей, средствах многопользовательской работы и многом другом. Такие среды стали называть CASE-системами (computer-aided software engineering) или DSM-решениями, а среды разработки таких предметно-ориентированных решений – metaCASE-системами или DSM-платформами.

За несколько десятилетий своего развития DSM-решения адаптировали для своих нужд многие инструменты, считающиеся уже традиционными для текстовых IDE. Среди них, например, визуальные интерпретаторы и отладчики создаваемых моделей, средства задания и осуществления рефакторингов над ними, синтаксическая подсветка элементов диаграмм, средства версионирования моделей. В связи с этим крайне актуальной видится задача переноса всех этих инструментов на уровень DSM-платформ, т.е. возможность быстрого автоматизированного создания полноценных визуальных интегрированных сред, поддерживающих полный цикл разработки ПО.

Среди существующих открытых metaCASE-систем лучше всего описанную проблему решает проект Eclipse Modeling Project (EMP) [9], развиваемый силами множества исследовательских групп и промышленных компаний по всему миру. Включая в свой состав десятки специализированных проектов, EMP предоставляет инструментарий для создания мощных CASE-систем, однако требует длительного обучения для полноценного использования своих возможностей. Такая ситуация

указывает на необходимость продолжения исследований в этой области с целью создания более простых в использовании и легковесных DSM-платформ, позволяющих быстро создавать современные полнофункциональные DSM-решения для разработки ПО в различных предметных областях.

Целью диссертационной работы является ускорение процесса разработки инструментальных средств поддержки визуальных языков путём создания программной платформы, позволяющей разрабатывать полнофункциональные визуальные среды, поддерживающие все основные этапы жизненного цикла программного обеспечения, даже неспециалистам в короткие сроки без специальной подготовки.

Для достижения цели были сформулированы следующие задачи.

1. Предложить проектировщикам ПО средства повышения скорости выполнения типовых задач при работе с диаграммными редакторами и разработать метод для реализации подобных средств для новых языков.
2. Предложить метод формальной спецификации исполнимой семантики моделей с целью ускорения создания интерпретаторов и отладчиков визуальных языков.
3. Спроектировать DSM-платформу, реализующую предложенные методы.
4. Реализовать и провести апробацию созданной DSM-платформы на практических задачах.

Цели и задачи диссертационной работы соответствуют области исследований паспорта специальности 05.13.11 «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»: пункту 1 (модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования) и пункту 2 (языки программирования и системы программирования, семантика программ).

Объектом исследования являются визуальные языки, **предметом**

исследования являются технологии для разработки инструментальных средств визуальных языков.

Методология исследования типична для решения задач в области предметной инженерии и сводится к последовательной идентификации и анализу проблемы, проектированию её возможного решения, выбору подходящих средств и технологий программирования, реализации и применения созданного решения, а также проведения инженерных экспериментов с целью обоснования эффективности полученного решения. В качестве **методов** исследования используются теория формальных языков, теория графов, методы объектно-ориентированного программирования, эмпирические методы (анализа литературы, постановки и проведения инженерных экспериментов).

Научная новизна данной работы заключается в следующем.

1. Предложенный метод, позволяющий автоматически генерировать средства распознавания жестов для графического редактора создаваемого языка по описанию конкретного синтаксиса этого языка, является новым.
2. Созданная на базе разработанной платформы система программирования диаграмм машин состояний для задачи распознавания жестов в проекте компьютерного зрения является оригинальной.
3. Разработанная программная платформа для создания предметно-ориентированных решений превосходит существующие аналоги по объему функциональных возможностей, предоставляемых разработчикам инструментальных средств с помощью единого программного интерфейса.

В основу разработанной предметно-ориентированной инструментальной платформы и технологии, предложенной на ее основе, положены следующие принципы.

- Конечные среды разработки создаются на основе обобщенной

инструментальной платформы, в каждом конкретном случае расширяя ее спецификой выбранной области. Это позволяет реализовывать различные функциональные средства максимально абстрактно на уровне DSM-платформы и с минимальными усилиями переиспользовать их между различными DSM-решениями.

- Число шагов, необходимых для создания готовой к работе специализированной среды разработки с минимальным комплектом функциональности, должно быть относительно небольшим, что позволит разработчику языка как можно быстрее получить работающую первую версию DSM-решения и продолжить создание остальных интересующих его инструментов итеративно. Технология не должна заставлять разработчика выполнять действия по созданию инструментов, которые в данный момент ему не интересны или не понятны.
- Создаваемые с помощью предлагаемой платформы интегрированные среды разработки по функциональности должны быть сравнимы с существующими в настоящее время CASE-средами, разработанными «вручную».
- Создаваемые с помощью предлагаемой платформы интегрированные среды разработки должны удовлетворять повышенным требованиям к удобству их использования, предоставляя своим пользователям подходящие средства моделирования и автоматизируя как можно больше рутинных и повторяющихся действий.

Положения, выносимые на защиту.

1. Предложен метод для создания инструментов распознавания жестов для диаграммных редакторов предметно-ориентированных языков.
2. Разработан метод формального задания операционной семантики предметно-ориентированных визуальных языков, позволяющий автоматически создавать для них интерпретаторы и отладчики.

3. Предложена модель (архитектура) программного комплекса (DSM-платформы), позволяющего автоматизированно создавать большинство типовых компонентов современных CASE-систем.
4. Выполнена реализация и апробация созданной DSM-платформы на практических задачах, подтвердившая работоспособность созданных инструментов и предложенных решений.

Степень разработанности темы исследования.

Исследованиями процесса разработки DSM-платформ занимается целый ряд научных коллективов: группа под руководством S. Kelly и J.-P. Tolvanen из университета г. Jyväskylä (Финляндия), группа под руководством J. de Lara из Автономного университета Мадрида (Испания), международная некоммерческая организация Eclipse Foundation и другие. В России вопросами визуального моделирования занимается исследовательские группы под руководством Л. Н. Лядовой, Ф. А. Новикова, А. А. Шалыто, В. П. Котлярова и другие. Результаты некоторых из этих исследований были воплощены в инструментальных средствах, как коммерческих (MetaEdit+, Microsoft Modeling SDK), так и открытых (Eclipse Modeling Project, Generic Modeling Environment, AToM3, MetaLanguage). Коммерческие системы недоступны для модификации и настройки сторонним пользователям, а самая зрелая открытая система EMP представляет собой объединение более десятка других проектов, которые активно развиваются, но часто бывает непросто наладить их взаимодействие друг с другом.

Среда QReal [3, 7, 53, 111] разрабатывается в рамках деятельности научно-исследовательской группы по изучению модельно-ориентированного подхода к разработке ПО. Коллектив данной группы работает под руководством проф. А. Н. Терехова с 2007 года и базируется на более чем двадцатилетнем опыте коллектива кафедры системного программирования Санкт-Петербургского государственного университета в области создания и внедрения средств

программирования, основанных на графических языках (см., например, работы [26-28, 51-52]). Проект имеет открытый исходный код, разработка ведется силами студентов, аспирантов и преподавателей кафедры [6]; автор данной диссертации — создатель первых прототипов QReal, разработчик и технический руководитель проекта. Хочется особо отметить вклад студентов, работавших над данным проектом под руководством автора диссертации: Мордвинова Дмитрия Александровича, Полякова Владимира Александровича, Подкопаева Антона Викторовича, Тарана Кирилла Сергеевича, Еварда Вадима Евгеньевича, Колантаевской Анны Сергеевны, Соболева Артёма Александровича, Новожилова Евгения Алексеевича, Самарина Алексея Владимировича, Агаповой Татьяны Юрьевны, Терешина Романа Юрьевича, Шквиро Ирины Алексеевны, Семеновой Анастасии Владимировна, Сенина Ивана Игоревича, Глазачева Владимира Александровича, Азимова Рустама Шухратулловича, всех студентов, работавших под руководством Литвинова Юрия Викторовича, а также вклад Никандрова Георгия Александровича и Симоновой Александры Андреевны. На момент начала работы над данным исследованием в проекте QReal были разработаны первые прототипы графо-графической библиотеки, средств быстрой разработки языков в тот момент создано не было. На данный момент среда существует в виде работающего прототипа.

Теоретическая и практическая значимость работы. Теоретическая значимость диссертационного исследования заключается в создании метода автоматического создания механизмов распознавания жестов диаграммных редакторов, а также в разработке архитектуры программной платформы, позволяющей гибко настраиваться под требуемый набор функциональности, формируя среду разработки для выбранного визуального языка. Практическая значимость определяется использованием полученных результатов для разработки DSM-платформы QReal, а также создания на ее основе ряда DSM-решений для различных предметных областей: среды визуального программирования роботов

QReal:Robots, используемой для обучения школьников основам программирования и кибернетики, редактора диаграмм машин состояний и генератора кода по нему для приложений компьютерного зрения, использовавшихся в проектах ЗАО «ЛАНИТ-Терком», а также нескольких исследовательских сред разработки для апробации различных аспектов технологии и инструментальных средств платформы: визуальной среды разработки, основанной на языке блок-схем, среды для разработки мобильных приложений для платформы Ubiq Mobile [37].

Среда QReal:Robots демонстрировалась на Открытых состязаниях Санкт-Петербурга по робототехнике в 2012, 2013, 2014 и 2015 году, на робототехнических фестивалях «Робофест 2012» и «Робофест 2013» в Москве. На базе QReal:Robots был проведен ряд мастер-классов, в том числе и на международных конференциях Skolkovo Robotics 2014 и 2015. В настоящий момент среда QReal:Robots переименована в TRIK Studio и используется как основное средство программирования кибернетической платформы ТРИК, в ряде робототехнических кружков России, применяется для обучения студентов Поволжской государственной социально-гуманитарной академии (г. Самара), а также на мастер-классах, проводимых ООО «КиберТех».

Проект поддержан грантом Санкт-Петербургского государственного университета 6.39.1054.2012.

Степень достоверности и апробация результатов. Достоверность и обоснованность результатов исследования опирается на использование формальных методов исследуемой области и инженерные эксперименты. По результатам работы были сделаны доклады на второй всероссийской научно-практической конференции, посвященной памяти заслуженного деятеля науки РФ, профессора В.Ф. Волкодавова (Самара, 2009); на второй научно-технической конференции молодых специалистов «Старт в будущее» (Санкт-Петербург, 2011); на конференции SYRCoSE 2012 (Пермь, 2012); на межвузовском конкурсе-конференции студентов, аспирантов и

молодых ученых Северо-Запада «Технологии Miscrosoft в теории и практике программирования» (Санкт-Петербург, в 2011 и 2013 году); на международной конференции «Информационные технологии в образовании и науке» (Самара, 2011); на III Всероссийской конференции «Современное технологическое обучение: от компьютера к роботу» (Санкт-Петербург, 2013); на Девятой независимой научно-практической конференции «Разработка ПО 2013» (CEE SEC(R)-2013, Москва); на международной конференции «10th Conference of Open Innovations Association FRUCT» (Тампере, Финляндия, 2011); международной конференции «Evaluation of Novel Approaches to Software Engineering» (ENASE 2013, Анже, Франция).

Личный вклад автора. Результаты, представленные в диссертационной работе, получены соискателем либо самостоятельно, либо при его непосредственном участии. Над проектом QReal работала большая группа студентов, аспирантов и преподавателей кафедры системного программирования СПбГУ, автор претендует лишь на результаты, явно перечисленные в списке положений, выносимых на защиту.

Публикация результатов. Результаты диссертационной работы представлены в 22 публикациях. Основные результаты опубликованы в 4 статьях ([3], [30], [42], [53]) в журналах из перечня российских рецензируемых научных журналов, в которых должны быть опубликованы основные научные результаты диссертаций на соискание учёных степеней доктора и кандидата наук.

Работы [5], [7], [29], [30], [32], [38-43], [47-50], [53], [72], [111], [132] написаны в соавторстве. В работах [48-50] и [53] диссертанту принадлежит разработка архитектуры и реализация основных модулей платформы QReal, А. Н. Терехову – постановка задачи, Ю. В. Литвинову – разработка средств метамоделирования ([53]) и разработка компонент, специфичных для программирования роботов ([48-50]). В работе [72] диссертанту принадлежит разработка генератора кода и редактора экранных форм, А. Н. Терехову и В. В. Оносовскому – интеграция решения с

платформой Ubiq Mobile, Ю. В. Литвинову – реализация диаграммных языков и редакторов для них. В работах [32] и [132] диссертанту принадлежит реализация механизма распознавания жестов мышью на уровне платформы QReal, Ю. В. Литвинову – создание инструментов задания «идеальных жестов», М. С. Осечкиной – реализация алгоритмов классификации жестов мышью и постановка экспериментов. В работах [29], [30] и [111] диссертанту принадлежит архитектура решения и реализация основных компонент платформы, Ю. В. Литвинову – разработка средств метамоделирования, А. С. Кузенковой – реализация некоторых частей метаредактора QReal, А. В. Подкопаеву – реализация средств генерации кода, А. О. Дерипаска – реализация редактора форм элементов, В. А. Полякову – реализация механизма преобразования графов. В работах [5] и [7] диссертанту принадлежит архитектура решения и реализация основных компонент платформы, Ю. В. Литвинову – разработка средств метамоделирования. В работах [38-43] и [47] автору принадлежит постановка задачи, реализация архитектуры и интеграция решений, запрограммированных соавторами.

Ниже приведён краткий план последующих глав диссертации.

В **главе 1** приводится общее описание модельно-ориентированного подхода к разработке программного обеспечения, приводятся примеры подходов и методологий, основывающихся на модельно-ориентированной разработке, дается описание предметно-ориентированной парадигмы к разработке ПО, обсуждается структура визуальных языков, уровни моделей, вводятся определения и понятия, важные для дальнейшего изложения.

Глава 2 содержит описание функциональности типичных CASE-систем, рассматриваются некоторые из существующих DSM-платформ, используемые для создания промышленных DSM-решений, для каждой из которых делаются выводы о возможности создания сред визуальной разработки с определенным набором функциональности. Делается вывод о необходимости разработки современных DSM-

платформ, позволяющих создавать визуальные интегрированные среды разработки, достаточно функциональные, чтобы быть применимыми в промышленности, и достаточно удобные, чтобы приносить реальную пользу разработчикам.

Глава 3 посвящена обсуждению влияния степени удобства использования визуальной IDE на желание и продуктивность использования подобных средств разработчиками ПО. Приводится описание средств, реализованных в платформе QReal для повышения удобства использования DSM-решений на основе данной платформы — распознавание жестов мышью как средство быстрого создания элементов и связей между ними на диаграммах, использование графических элементов управления для изменения свойств элементов как части визуального представления этих элементов на диаграммах, реализация некоторых эвристик языка ДРАКОН.

Глава 4 посвящена вопросу создания исполнимых визуальных языков. Рассматриваются виды семантик и существующие способы задания исполнимой семантики для визуальных языков. Приводится описание решения, реализованного в платформе QReal — описывается язык задания семантики визуальных языков, а также инструментальные средства для использования этого описания семантики для автоматизированного создания визуальных интерпретаторов языков на основе формальной модели сетей Петри.

Глава 5 содержит описание предметно-ориентированной платформы QReal, в этой главн обсуждается назначение и общая архитектуры платформы, кратко описывается назначение основных модулей и взаимодействия между ними. Далее приводится состав типового DSM-решения на основе платформы QReal, перечисляется набор инструментов, его составляющих. Для каждого инструмента приводится описание его функциональности и ситуаций при разработке ПО, в которых данный инструмент будет полезен пользователям DSM-решения. Также приводятся некоторые реализационные особенности, показывающие способы

интеграции описанных инструментов с базовой платформой QReal.

Глава 6 содержит примеры применения результатов, описанных в данной диссертации, для разработки DSM-решений — среды для программирования роботов QReal:Robots, среды на основе языка блок-схем, DSM-решения для разработки мобильных приложений для платформы Ubiq Mobile, а также решения на основе диаграмм машин состояний для проекта компьютерного зрения. Для каждой среды разработки приводится описание конечной функциональности, обсуждаются расширения платформы QReal, которые было необходимо сделать для реализации данного решения, а также специфичная функциональность, которую пришлось реализовывать “вручную”. Описывается эксперимент по сравнению платформы QReal с двумя другими популярными платформами (MetaEdit+ и Eclipse Sirius), делаются выводы о применимости полученных в работе результатов.

Приложение A содержит обзор исследований, проводимых на тему внедрения CASE-систем в производственный процесс, анализируются причины их возможного неиспользования разработчиками.

В **Приложении B** приводится обзор и сравнение подходов к заданию исполнимой семантики визуальных языков.

В **Приложении C** описывается реализованный в рамках диссертационной работы алгоритм поиска подграфа в графе модели, хранящейся в репозитории QReal.

В **Приложении D** приводятся копии актов о внедрении разработанной в рамках исследования платформы.

В **заключении** приведены итоги выполненного исследования, рекомендации и перспективы дальнейшего развития.

Глава 1. Модельно-ориентированная разработка

1.1. Введение

В данной главе описан контекст работы и базовые понятия, которые будут использоваться в последующих главах. Рассматриваются наиболее популярные подходы к разработке ПО, использующие модели, описывается структура визуальных языков и уровни моделирования.

1.2. Разработка ПО и модели

Традиционно в инженерных дисциплинах технологический процесс состоит из двух принципиально различающихся этапов — проектирования и реализации. На первом этапе разрабатывается модель создаваемого изделия, которая в дальнейшем используется как эталон на этапе реализации. Чаще всего модели представляются в виде чертежей, описывающих ключевые особенности создаваемого изделия. Попытки применить данный подход напрямую к программной инженерии сталкиваются с рядом проблем, связанных с невидимостью и нематериальностью программ [70]. Например, при вытачивании детали на станке или строительстве дома человек имеет в голове мысленный образ изделия, который как в процессе, так и по завершении изготовления может быть сравнен с получающимся продуктом. С программным обеспечением всё по-другому — ни исходные тексты в силу своего объема и сложности, ни определенные внешние проявления работы программы (экранные формы, создаваемые файлы, посылаемые сообщения и т.п.) не могут полностью охарактеризовать ту или иную программу как объект физического мира. А как можно эффективно и подконтрольно создавать то, что невозможно представить?

Возникают различные метафоры визуализации [27] — способы формально

сопоставить абстрактные части ПО зрительно воспринимаемым объектам. В настоящее время наиболее часто применяемой метафорой визуализации ПО являются графы — вершины обозначают определённые сущности, а рёбра — определённые связи между ними. Для разных языков сущности и связи между ними могут быть как абстрактными (объекты, классы, состояния, блоки ветвления и циклов и т.п.), так и вполне конкретными объектами и действиями целевой предметной области (например, база данных, температурный датчик, присылающий значения в систему, внешнее устройство, которым система может управлять посылкой сообщений, и многое другое).

Модельно-ориентированная разработка (Model-driven engineering, MDE) ПО основывается на представлении программы в виде набора моделей, представляющих ее с различных точек зрения. При этом обычно используются визуальные языки моделирования, с помощью которых создаются разного уровня абстракции описания предметной области, разрабатываемой системы и взаимодействующего с ней окружения. Различные методологии и подходы к разработке в рамках MDE по-разному используют полученные при моделировании диаграммы: например, для фиксирования и формализации знаний при сборе и анализе требований, для общения с заказчиком или между членами команды разработчиков, для спецификации и визуализации архитектуры, для разделения задач и организации многопользовательской разработки или даже для генерации по ним целевого кода разрабатываемой системы.

Рассмотрим некоторые методологии и подходы к разработке, основанные на использовании визуальных моделей.

1.3. ПО для работы с данными

Одна из областей, в которых человек активно использует ПО — это работа с данными. Более того, обработка больших объемов данных исторически была одной

из первых задач, для которых стали применяться компьютеры. В эру мейнфреймов подобные вычислительные устройства могли себе позволить лишь большие компании, университеты или крупные государственные учреждения, и наличие большого хранилища данных и информационной системы для работы с ним для всех них было ключевым моментом.

Типичный процесс разработки информационной системы состоит из следующих этапов [89] — анализ требований, проектирование базы данных (БД), разработка пользовательского интерфейса к данным, средств создания отчетов и прочих инструментов системы. При этом проектирование адекватной структуры хранимых данных является одним из самых критичных мест разработки, поскольку неэффективная структура БД повлечет за собой неоправданный рост либо хранимых данных, либо времени их обработки, либо и того, и другого сразу.

При разработке приложений для работы с данными моделирование осуществляется на нескольких уровнях [59].

- Концептуальный уровень описывает предметную область в терминах, понятных и знакомых людям. Модели на этом уровне (называемые концептуальными схемами) описывают структуру предметной области: какие типы объектов там присутствуют, какие роли играют, какие на них накладываются ограничения и т.п.
- На логическом уровне происходит выбор подходящей логической модели данных (реляционная, объектно-ориентированная и т.п.), после чего концептуальные схемы превращаются в логические схемы, выражаемые в терминах абстрактных структур данных и операций, поддерживаемых выбранной моделью данных. Например, в реляционной схеме факты хранятся в таблицах, а ограничения выражены с использованием первичных и внешних ключей.
- На физическом уровне логическая схема может быть представлена физической

схемой в выбранной системе управления базами данных (СУБД). Например, реляционная схема может быть реализована в MS SQL Server или IBM DB2. Физическая схема включает в себя детали описания физического хранилища данных, структур его внутренней организации и т.п. (например, индексы, кластеризация файлов и т.д.). Для каждой СУБД обычно этот выбор может быть сделан по-разному, к тому же даже в рамках одной СУБД возможно использование различных средств. Таким образом, для одной логической схемы могут быть выбраны разные физические схемы.

Также часто выделяют еще внешний уровень, на котором описывается предметная область в том виде, как она представляется различным группам пользователей (для разных групп пользователей могут создаваться разные интерфейсы к создаваемым моделям относительно набора отображаемых данных, прав доступа к ним и т.п.).

Модели, создаваемые на концептуальном уровне, чаще всего представляются визуально. Наглядное и понятное для всех участников процесса описание структуры предметной области значит очень много на начальном этапе разработки — от того, насколько удачно данная модель будет отражать картину реального мира, часто зависит, насколько удобным в работе станет данное приложение, и сколько ресурсов уйдет на его разработку. Дальнейшие переходы к логическому и физическому уровню чаще всего обеспечиваются настройками самой СУБД и/или средств, предоставляемых разработчиками СУБД для создания приложений для работы с ней.

В области разработки ПО для работы с данными моделирование диаграммами стало применяться более 30 лет, многие из этих методов в том или ином виде используются и сейчас. Рассмотрим несколько популярных подходов к концептуальному моделированию данных [89].

1.3.1. Модель “Сущность-связь” (Entity-Relationship Model)

Данный подход был предложен в 1976 году Питером Ченом [73] и до сих пор

является одним из самых распространенных подходов к моделированию данных. Он отображает реальный мир в виде сущностей, у которых есть атрибуты и которые участвуют в некоторых отношениях. Например, факт, что у человека есть день рождения, отражается тем, что у сущности “человек” есть атрибут “дата рождения”, а то, что некий работник работает в некоем отделе — связью между сущностями “работник” и “отдел”. Такой подход довольно интуитивен, и даже несмотря на популярность других подходов, ER все еще самый популярный для моделирования приложений, работающих с БД. Со временем появилось много разных версий данного подхода [61, 64, 85, 97], так что в настоящее время нет одной стандартной ER-нотации.

1.3.2. Семантическое моделирование

При семантическом моделировании модели строятся, основываясь на фактах (fact-oriented), а не на сущностях предметной области. Факты при этом представляются в виде бинарного отношения между элементами модели, а сами элементы характеризуются ролями, которые они в этих отношениях играют. Факты о предметной области составляются в виде предложений на естественном языке, которые потом формализуются в виде отношений, предоставляемых выбранной нотацией.

В отличие от моделей типа “Сущность-связь”, при семантическом моделировании практически не используются атрибуты сущностей. Все факты представляются в терминах объектов и их ролей. Несмотря на то, что это часто приводит к большим по объёму диаграммам, этот подход имеет свои преимущества для концептуального анализа, которые выражаются в большей простоте и наглядности для непрофессионалов, стабильности и простоте валидации получаемых моделей.

Основная нотация — ORM [131].

1.3.3. Объектно-ориентированное моделирование

Несмотря на то, что объектно-ориентированное моделирование создавалось для проектирования программных систем, для моделирования данных оно тоже может быть использовано. Существует много подходов объектно-ориентированного моделирования, наиболее известным из которых является UML — унифицированный язык моделирования, предложенный консорциумом OMG [127]. В текущую на данный момент версию UML 2.5 входит целый набор типов диаграмм, одним из которых являются диаграммы классов, описывающие статические структуры данных. Диаграммы классов способны задавать как операции, так и низкоуровневые решения из области реализации (например, видимость атрибутов или направленность ассоциаций). Абстрагировавшись от подобных элементов, относящихся к реализации, можно считать диаграммы классов UML расширением ER-нотации.

Достоинством использования UML для моделирования данных является то, что в эту нотацию возможно заложить более подробную информацию о предметной области и создаваемой системе, которую в дальнейшем можно использовать будет не только при разработке схемы БД, но и самого приложения. Также в UML 2.5 помимо диаграммы классов входят и другие диаграммы (например, диаграммы деятельности или диаграммы машин состояний), которые могут помочь визуализировать поведенческий аспект предметной области.

1.4. Модельно-ориентированная архитектура

Модельно-ориентированная архитектура (Model-driven architecture, MDA) — подход к разработке программных систем, продвигаемый консорциумом OMG. MDA определяет несколько типов моделей и фиксирует поэтапный процесс разработки ПО, состоящий в последовательном продвижении от одной модели системы к другой. При этом последующие модели получаются на основе

предыдущих как автоматическим преобразованием с использованием соответствующего инструментария, так и “ручным” дополнением силами проектировщиков. Последовательность преобразований моделей завершается генерацией кода готовой системы для выбранной платформы. Под платформой в MDA понимается совокупность технологий и подсистем с набором функциональности, которым любое приложение, работающее на этой платформе может использовать безотносительно того, как эта функциональность платформы реализована [117].

Стоит отметить, что методология MDA фиксирует процесс и состав каждого этапа, при этом никак не ограничивая класс создаваемых с ее помощью приложений, т.е. может быть использована при разработке произвольных программных систем.

Рассмотрим виды моделей, которые определяет MDA [117].

- Независимая от вычислений модель (Computation Independent Model, CIM) используется для описания общих требований к системе, словаря используемых терминов предметной области, а также окружения системы. В данную модель должны включаться только те сущности, которые будут детализироваться и использоваться в последующих моделях. CIM не должна раскрывать каких-либо деталей внутреннего устройства системы и предназначена для формирования общего видения системы экспертами предметной области и техническими специалистами. Часто эти модели называют моделями предметной области или бизнес-моделями. Модели CIM представляют собой общее концептуальное описание системы, а поэтому при необходимости могут быть опущены при создании небольших систем.
- Независимая от платформы модель (Platform Independent Model, PIM) описывает структуру и бизнес-логику разрабатываемой системы. Модели на этом уровне могут содержать сколь угодно подробные сведения о поведении, архитектуре приложения или пользовательском интерфейсе, однако они не

должны касаться вопросов реализации на конкретных платформах. Модель PIM строится на основе модели CIM, если последняя была создана на предыдущем этапе.

- Зависимая от платформы модель (Platform Specific Model, PSM) дополняет описания PIM деталями, специфицирующими способ реализации системы для конкретной платформы. Для каждой платформы, на которой планируется функционирование разрабатываемой системы, создается отдельная модель PSM. Получаемая на этом уровне модель содержит всю техническую информацию, необходимую для генерации кода и других целевых артефактов разработки.

Также MDA выделяет модель платформы, содержащую в себе набор технических описаний составляющих частей платформы и сервисов, предоставляемых ею приложениям. Модель платформы используется при преобразовании модели PIM в модель PSM.

Ввиду общности методологии все модели в MDA рекомендуется создавать с помощью унифицированного языка моделирования UML.

Процесс преобразования моделей и роли, осуществляющие изменения моделей на каждом этапе, отражены на рис. 1.

MDA определяет целый ряд способов преобразования моделей, правила преобразования моделей рекомендуется задавать с помощью языка, описанного стандартом QVT (Query/View/Transformation) [136]. Преобразования могут быть параметризованы, что позволит подстраивать их под нужды конкретных проектов. Создано большое количество средств разработки, поддерживающих MDA[1], в которых процесс преобразования моделей максимально автоматизирован.

[1] Например, Enterprise Architect, Rhapsody, Rational Software Architect, UModel и другие

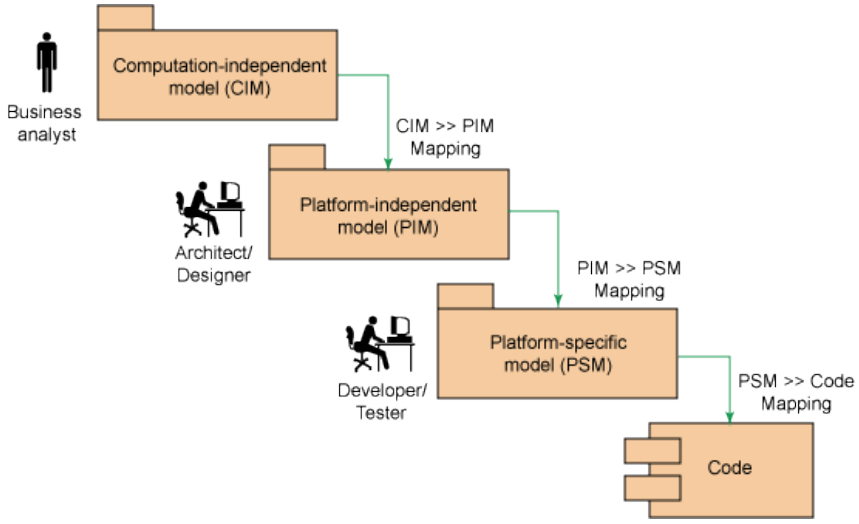

Рисунок 1. Преобразование моделей в MDA[2]

Рассмотренный подход к разработке ПО несомненно имеет свои преимущества: минимизация ресурсов по переносу приложения на другую платформу, потенциальная независимость моделей от средств разработки, автоматизация и четкая структуризация процесса разработки. Однако, существенным, на наш взгляд, недостатком предлагаемого подхода является то, что каждый переход от одной модели к другой (и переход от моделей к коду) может сопровождаться необходимостью внесения “ручных” изменений в модели или получаемый исходный код. Для разработчика системы это означает следующее:

- Итеративный подход к разработке возможен только в случае, если используемые инструментальные средства поддерживают механизм циклической разработки ПО (round-trip engineering), при котором возможен возврат к предыдущему этапу разработки, и перегенерация моделей или кода с учетом «ручных» изменений, сделанных на предыдущей итерации.
- Даже если 90% кода или промежуточных моделей получается автоматически посредством используемого инструментария, для того, чтобы внести оставшиеся 10% изменений, требуется разбираться со сгенерированными

[2] Рисунок взят из работы [80]

артефактами, что в случае большого проекта может быть крайне сложной задачей.

1.5. Предметно-ориентированное моделирование

С появлением стандарта UML как унифицированного визуального языка, понятного всем, многие ожидали перехода на использование графических языков программирования как более наглядных и удобных для использования человеком по сравнению с языками текстовыми. Создаваемые и развиваемые в это время методологии преподносили визуальные модели как основной инструмент описания создаваемых систем. Однако практика показала, что создание графических спецификаций при разработке ПО не является принципиально более простым или быстрым процессом, чем программирование на текстовых языках — на низком уровне проектировщик также оперирует абстрактными понятиями типа циклов, операторов, условных переходов. Разумеется, использование визуальных моделей дает свои преимущества по сравнению с текстом — б*о*льшая наглядность, возможность построения иерархии моделей и быстрое переключение между уровнями абстракции системы (при соответствующей поддержке инструментальных средств), — однако существенного повышения уровня производительности разработчиков при этом не происходит. Существующие немногочисленные исследования показывают 35% повышение производительности труда [121], в то время как некоторые из них не отмечают никакого роста производительности вовсе, а иногда даже и ее снижение [66, 81]. На наш взгляд, это вызвано следующими причинами.

Любая модель — упрощение того, что эта модель описывает. В случае MDE, модель — упрощенное представление того, как проектировщик понимает разрабатываемую программную систему. Это вполне ожидаемо, поскольку модель по определению должна быть проще моделируемой сущности, однако из-за этого

при переходе к формальным спецификациям определенная часть знаний о системе теряется, образуя семантический разрыв между идеей и её реализацией. Однако, обладая неполной информацией, инструментальные средства оказываются не в состоянии сгенерировать полноценный код по создаваемым моделям. Требование автоматической генерации кода по моделям на языках общего назначения влечёт за собой серьёзное усложнение описаний систем, что приводит к нежеланию разработчиков работать с подобными моделями и инструментами.

К примеру, в попытках превратить UML в полноценный язык программирования[3] для описания алгоритмов стали использовать диаграммы деятельности (activity diagrams). Язык UML является языком общего назначения, а значит, оперирует практически теми же сущностями, что и текстовые языки программирования, однако текстовые программы являются более компактными, чем соответствующие им диаграммы активностей. При детальном описании системы модели становятся настолько крупными и сложными, что теряется их основное преимущество — наглядность и понятность. Если же сохранять модели компактными и понятными в ущерб информативности, инструментальные средства будут способны создать автоматически лишь часть кода системы (например, скелет архитектуры программы или только заголовочные файлы классов для C++ с функциями-заглушками для методов в файлах реализации). Как уже было отмечено выше, для поддержки итеративности процесса разработки в таких случаях используются механизмы возвратного проектирования, инструментальная поддержка которых во многих средствах реализована далеко не самым удобным образом.

Учитывая общую инерционность людей при использовании чего-то нового и незнакомого, многие программисты предпочли вернуться к уже освоенным техникам и инструментам текстового программирования. В итоге в настоящее время

---

[3]Исполнимый UML (Executable UML, xUML) **[**115, 150**]**

модельно-ориентированная разработка с помощью языков общего назначения (и UML в частности) используется большей частью для документирования на этапах выявления требований и проектирования, а разработка ведется “традиционным” способом программированием на текстовых языках.

Дальнейшее развитие идея визуального программирования нашла в парадигме предметно-ориентированного моделирования (domain-specific modeling, DSM). Основу данного подхода к разработке составляет активное использование специализированных языков, создаваемых специально для решения текущей задачи. Языки общего назначения пригодны для решения широкого круга задач, однако они оперируют абстрактными сущностями. Ограничив определенную предметную область, возможно создать инструментальные средства, идеально подходящие для выбранного круга задач — элементами языка могут быть не абстрактные сущности, как в случае языков общего назначения, а высокоуровневые элементы из предметной области, связями между ними — операции, которые выполняются над этими элементами. Например, соединив на диаграмме элементы “Кнопка” и “Сообщение”, можно при генерации получить код, осуществляющий отправку SMS-сообщения по нажатии определенной кнопки на экране мобильного телефона. В случае DSM это становится возможно, поскольку инструментальная поддержка — визуальный редактор, генератор, библиотеки времени исполнения и прочее — также созданы специально для этого круга задач и обладают достаточными знаниями о предметной области, чтобы автоматически получать даже довольно объемные и сложные участки кода по компактным и простым моделям. При этом, использование подобных языков приводит к преодолению описанного выше семантического разрыва — проектировщик строит модели напрямую из сущностей целевой предметной области, необходимость в промежуточном уровне формализации в виде абстрактных объектов и связей между ними пропадает (см. рис. 2).

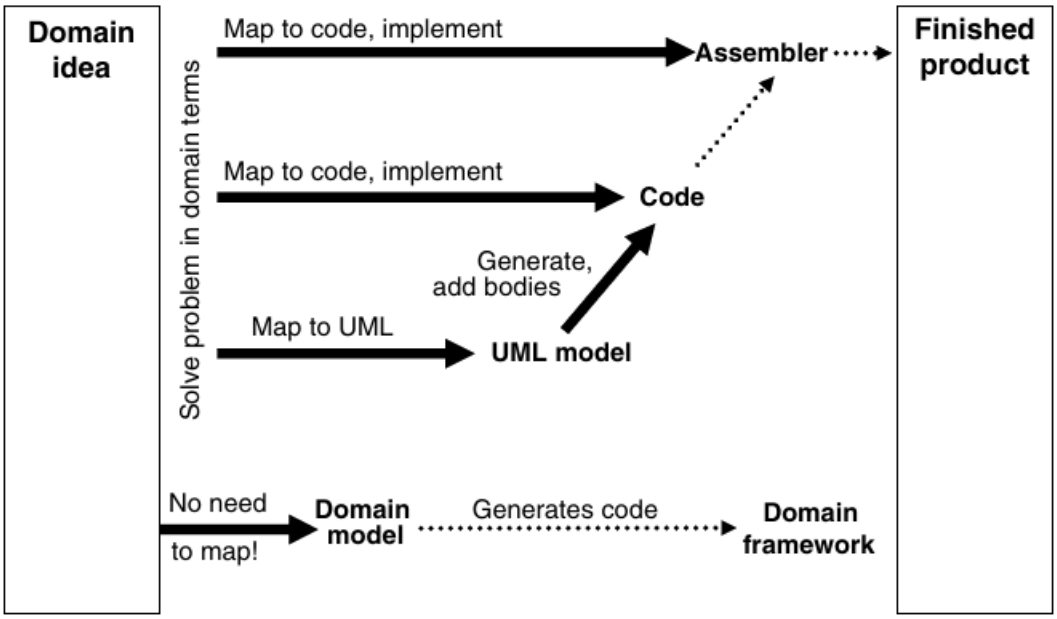

Рисунок 2. Преодоление семантического разрыва между моделями и предметной областью в DSM[4]

Явное ограничение круга решаемых задач и ориентации языка (и соответствующих инструментов) на выбранную часть предметной области делает также возможным стопроцентную генерацию исходного кода по моделям. В этом смысле DSM позиционируется как аналог перехода на структурное программирование с ассемблерных языков — повышение уровня абстракции с отказом к возврату разработчиков на нижний уровень. При написании программ, например, на языке C или C++ программист никогда не исправляет ассемблерные коды, генерируемые компилятором при сборке проекта. По аналогии с этим в рамках предметно-ориентированного моделирования предлагается отказаться от «ручной» доработки кода, генерируемого по визуальным моделям: для проектировщика существуют только визуальные модели, и он работает только с ними (см. рис. 3). Именно отказ от редактирования ассемблерных кодов и привёл к скачку в продуктивности при переходе от ассемблерных языков к структурному

[4] Рисунок взят из работы [102]

программированию, и идеологами DSM делается упор на то, что именно те же идеи приносят большой прирост производительности при переходе с визуальных языков общего назначения на предметно-ориентированные.

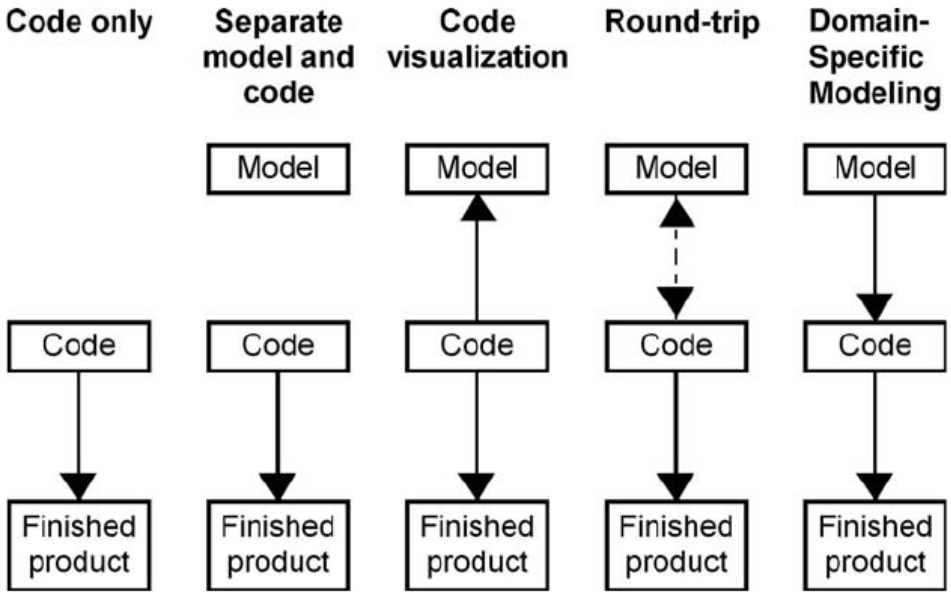

Рисунок 3. Связь моделей и исходного кода[5]

По разным оценкам DSM по сравнению с языками общего назначения дает прирост производительности в 300-1000%. Так, Nokia опубликовала результаты применения DSM подхода для разработки мобильных приложений, показывающие десятикратное увеличение производительности [103, 123]. Применение предментно-ориентированных языков для разработки ПО в BBC США дало трехкратный рост производительности [106]. Исследования, проводимые компанией Lucent, показывают рост производительности и снижение затрат в 3-5 раз для начинающих разработчиков и в 10 и более для опытных [159].

Разумеется, создание нового языка и полноценной инструментальной поддержки для него под каждую конкретную задачу — очень ресурсоёмкая работа, выполнять которую «вручную» для каждой задачи крайне сложно и экономически не выгодно. Для автоматизации данного процесса используются инструментальные

[5] Рисунок взят из работы [102]

среды, называемые предметно-ориентированными платформами (или, ранее, metaCASE-системами). Обзор функциональности современных DSM-платформ представлен в главе 2.

1.6. Метамоделирование

Для эффективной реализации предметно-ориентированной парадигмы необходимо средство, позволяющее быстро и униϕицированно проектировать семантически богатые языки и инструментальную поддержку для них. Таким механизмом является метамоделирование — процесс создания языков на основе их метамоделей.

1.6.1. Структура языка

В соответствии со стандартом языка MOF [129], метамодель — это модель, которая задает язык моделирования. Эта модель содержит в себе описание существенных свойств и возможностей этого языка. В то время, как состав, назначение и применимость языков могут сильно различаются от одного языка к другому, основные их черты остаются неизменными. Структуру языков традиционно представляют следующим образом (см. рис. 4):

- абстрактный синтаксис;
- конкретный синтаксис;
- служебный синтаксис;
- семантика;
- отображения в другие языки.

Абстрактный синтаксис описывает набор элементов, составляющих язык (сущности, описываемые языком, и возможные связи между ними), и то, как эти элементы будут объединяться вместе для создания моделей (правила, определяющие синтаксическую корректность моделей). Важным является то, что абстрактный

синтаксис языка не должен зависеть от конкретного синтаксиса и семантики. Назначение абстрактного синтаксиса — в задании структуры языка, набора его сущностей и их свойств, а не того, как эти элементы выглядят или что значат. Язык описания абстрактного синтаксиса принято называть метаязыком.

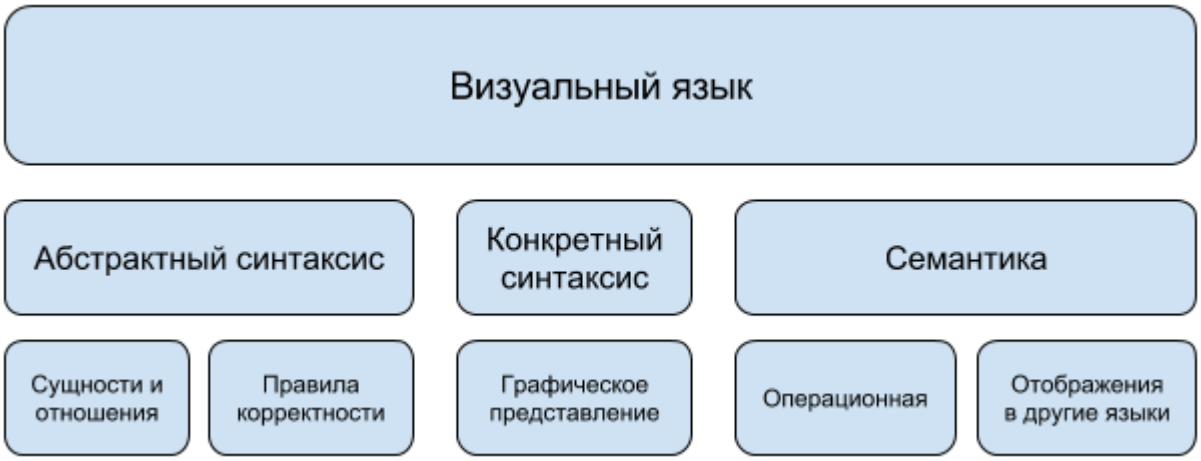

Рисунок 4. Структура языка

Конкретный синтаксис задает нотацию языка, определяет внешний вид элементов языка и моделей, с их помощью создаваемых. Традиционно графические языки представляются в виде диаграмм, состоящих из набора пиктограмм. Каждый элемент языка может иметь несколько различных представлений: например, по-разному отображаться на разных диаграммах. Более того, часто для графических языков бывает полезно иметь и некое текстовое представление.

Служебный синтаксис определяет формат хранения моделей. Например, в случае, если модели сериализуются с помощью XML-формата, служебный синтаксис будет задаваться соответствующей XML-схемой.

Семантика дает представление о том, что же значат элементы языка и модели, из них составляемые. Абстрактный синтаксис дает об этом слишком мало информации, лишь фиксируя правила синтаксической корректности моделей. Для многих языков семантика определяется неформально с помощью описаний на естественном языке (русском, английском или любом другом), подкреплённых

примерами использования. И если для языков моделирования (как, например, UML) это вполне приемлемо, то для языков программирования наличие формально заданной семантической модели является ключевым фактором исполнимости программ, созданных с помощью данного языка. Отсутствие формально заданной семантики приводит к неоднозначной трактовке моделей на этом языке, что делает автоматизированное построение и использование генераторов, интерпретаторов и отладчиков моделей практически невозможным.

Отображения (mappings) конструкций языка в другие языки позволяют задать связь этого языка с другими. Этот механизм может быть использован для создания генераторов кода, механизма преобразования моделей (например, для выполнения рефакторингов на уровне моделей), связи элементов между собой в рамках одного или между различными визуальными языками (например, задание трассировки между элементами моделей разных уровней абстракции).

Традиционно в литературе метамодель понимается как модель абстрактного синтаксиса языка, а метамоделирование как процесс создания этой модели. Однако также часто метамоделирование понимается как процесс разработки языка в целом, т.е. задание всех формальных спецификаций, достаточных для создания инструментальных средств для этого языка в рамках предметно-ориентированного моделирования. Таким образом, метамодель рассматривается как целостный источник знаний о языке, содержащий в себе все описанные выше компоненты — синтаксис, семантику и т.д. В данной работе мы понимаем под метамоделью именно описание абстрактного синтаксиса, для описания целостного источника знаний о языке будем говорить о “метамодели в широком смысле”.

1.6.2. Уровни моделирования

Метамодель в широком смысле (модель, описывающая язык моделирования) состоит из ряда формальных описаний, задающих каждый аспект разрабатываемого языка. Для создания каждого из этих описаний могут использоваться различные

языки, текстовые или графические. Например, для задания конкретного синтаксиса может быть использован специальный графический редактор фигур, или вид графического представления элемента может быть описан в XML файле в форме SVG [141]. Правила трансформаций моделей могут быть заданы визуально графовыми грамматиками [138], либо в текстовом виде на языке QVT (Query/View/Transformation). При этом каждый из этих языков также может быть задан соответствующей моделью. Это приводит нас к понятию метаметамодели — модели описания языка, который используется для задания других языков. Идея подобной иерархия была описана более, чем 30 лет назад [107], а также упоминается в стандартах ISO [98] и OMG [129].

Рассмотрим описанную иерархию моделей на примере системы, реализующей работу с каталогом фильмов (см. рис. 5). Сама предметная область — каталог фильмов, это первый уровень моделирования. В стандартах OMG этот уровень называется нулевым (M0), поскольку никакого моделирования в общепринятом смысле на этом уровне не происходит — модель в точности совпадает с моделируемой сущностью.

Второй уровень — собственно модель, описывающая предметную область. Состоит эта модель из диаграмм, которые создает проектировщик при использовании визуальной среды разработки, для моделирования используется подходящий для этого диаграммный язык, общего назначения или специализированный. На рис. 5 это диаграмма классов с элементами “Фильм”, “Режиссер” и связями между ними, задающими отношения между соответствующими сущностями предметной области. В терминологии OMG этот уровень называется M1, уровень модели.

Следующий уровень моделей получается, когда мы попытаемся описать визуальный язык, используемый на уровне M1. Результатом этого будет метамодель, которая также является моделью на некотором языке (или на нескольких языках,

если рассматривать метамодель в широком смысле). Для создания этой модели используется метаязык — язык описания других языков. Примером графического метаязыка является язык MOF [129] или Ecore (используемый в качестве метаязыка в проекте Eclipse Modeling Project и представляющий собой реализацию варианта EMOF языка MOF). В примере на рис. 5 в метаязык входят элементы “Класс”, “Атрибут” и “Ассоциация”, описывающие элементы языка, используемого на уровне M1. В терминологии OMG этот уровень называется M2, уровень метамодели.

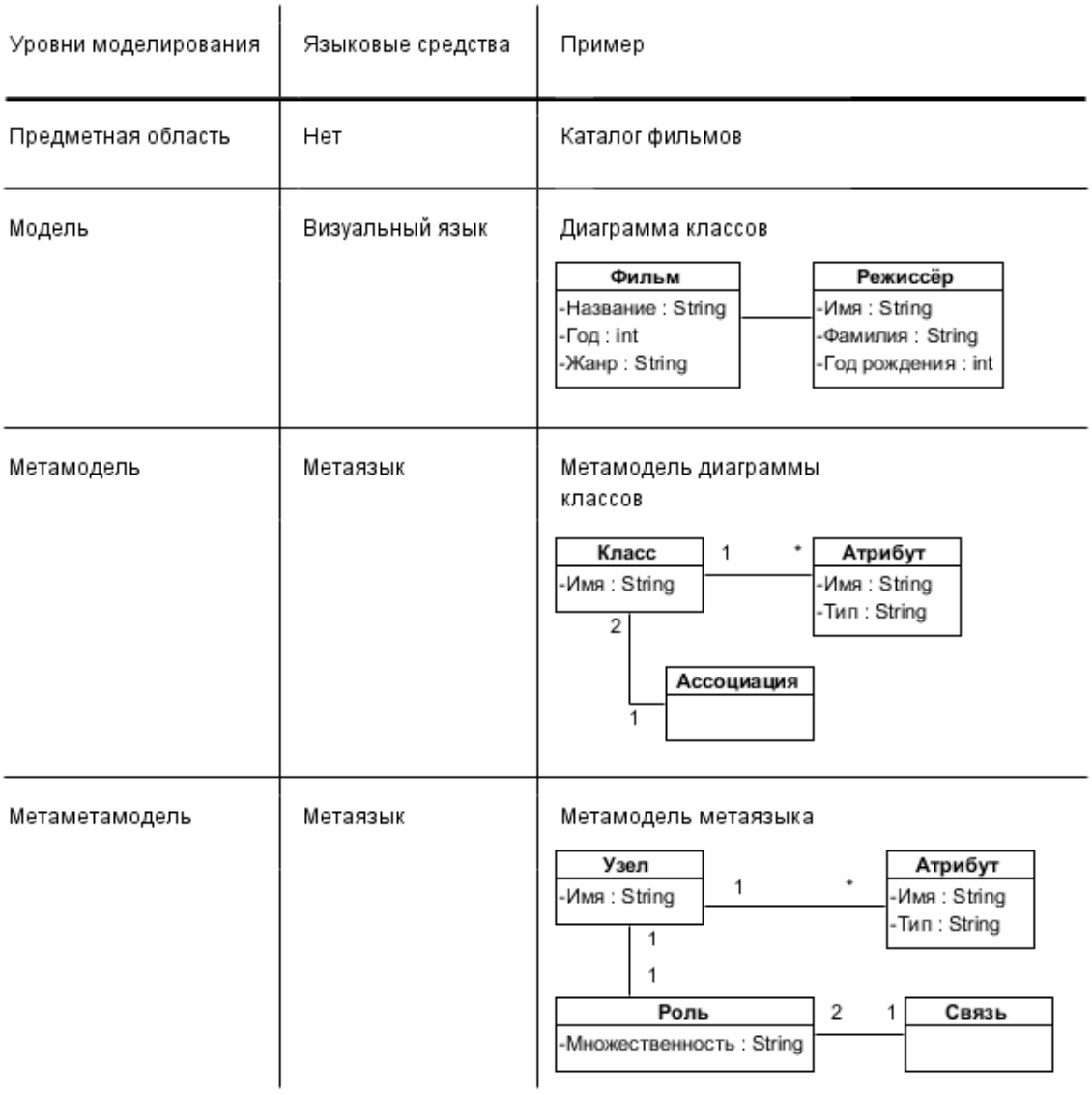

Рисунок 5. Иерархия уровней моделей

Но ведь метаязык — это тоже визуальный язык, и нет никаких препятствий

построить модель, описывающую его. Данная модель будет содержать абстрактные элементы типа “Узел” или “Связь” — элементы метаязыка. Подобная модель называется метаметамодель, а сам уровень в терминологии OMG называется M3. Подобную иерархию можно было бы продолжить, однако несложно заметить, что все последующие уровни будут структурно совпадать с M3, поэтому обычно на этом останавливаются.

1.7. Заключение

В этой главе были описаны основные подходы к разработке программного обеспечения, опирающиеся на диаграммные модели: моделирование данных (модели «сущность-связь», семантические и объектно-ориентированное моделирование), модельно-ориентированная архитектура (MDA) и предметно-ориентированное моделирование (DSM). Работа посвящена развитию последнего подхода, который при должной инструментальной поддержке способен как упростить процесс моделирования и повысить продуктивность проектировщиков, так и привлечь к разработке профессионалов предметной области, не имеющих навыков моделирования.

Рассматривается структура визуального языка: синтасис (абстрактный, конкретный и служебный) и семантика, а также уровни моделирования (предметная область, модель, метамодель, метаметамодель). Предметно-ориентированная платформа в силу своего назначения должна быть способной переключаться как минимум между уровнями модели и метамодели, что накладывает дополнительные сложности при её проектировании и реализации. Дальнейшее исследование раскрывает эти вопросы более подробно.

Глава 2. Обзор

2.1. Введение

В данной главе вводится понятие CASE-инструментария, рассматривается их назначение, обсуждаются достоинства и недостатки. Приводится состав типовой CASE-среды: описываются входящие в её состав инструменты и их назначение. Подобная классификация необходима, учитывая общее стремление DSM-платформ создавать инструменты, по возможности функционально не уступающие созданным «вручную». В обзор также входит анализ ряда наиболее популярных на данный момент DSM-платформ. В рамках данного исследования не важно, как происходит работа с этими платформами, нас интересует функциональность получаемых на их основе решений. По результатам анализа CASE-сред и DSM-платформ формулируются требования к современной платформе, позволяющей создавать решения, которые поддерживают все основные этапы жизненного цикла разработки ПО.

2.2. CASE-инструменты

Основное назначение инструментов — упрощение и автоматизация совершаемых человеком действий. В области программной инженерии ситуация ровно такая же. На ранних стадиях развития компьютерной техники процесс разработки программ состоял из непосредственно программирования. В этом программистам помогали редакторы текста, трансляторы этих текстов программ в ассемблерные и бинарные машинные коды, линковщики и загрузчики. С ростом сложности создаваемых систем и подходов к их разработке набор доступных разработчикам инструментальных средств стал существенно расширяться. В частности, повысилась ценность стадий анализа и проектирования создаваемого ПО,

тестирования, верификации программ и т.п., а вместе с ними и инструментов, автоматизирующих данные виды деятельности.

С начала 1980-х годов в активное обращение входит термин *CASE-инструмент* (Computer Aided Software Engineering), изначально используемый для описания любых инструментов, используемых для автоматизации разработки ПО: редакторов текста и документации, компиляторов и кодогенераторов, отладчиков, хранилищ данных и репозиториев и т.п. С развитием возможностей вычислительной техники и развитием подходов и методологий разработки ПО появляются инструменты поддержки сбора требований, анализа и дизайна, отладки и эмуляции создаваемых систем, поддержки итеративности процесса разработки и многое другое. Со временем в дополнение к текстовым инструментам появляются графические инструменты и методологии их использования, возросшая сложность и многочисленность которых приводит к тому, что с 1990-х годов под CASE-инструментами уже понимают не просто любые инструменты, используемые для разработки компьютерных программ, а инструменты (и даже целые инструментальные системы), автоматизирующие определенные этапы какой-либо методологии создания ПО. Появляется понятие *CASE-инструментария* или *CASE-пакета* (CASE workbench) [86], т.е. программного средства, интегрирующего в себе несколько CASE-инструментов: например, в случае текстовых языков, это может быть среда разработки, в состав которой входят текстовый редактор с подсветкой синтаксиса, компилятор и отладчик. В целом считалось, что использование CASE-инструментариев помогает упорядочить и стандартизировать процесс разработки ([57, 58, 130]), тем самым снизить время и затраты, затрачиваемые на создание ПО, повысив его качество ([57, 63, 51, 84, 130, 134]), иметь единый источник знаний о проекте ([119]), автоматически получать документацию о системе по создаваемым моделям ([74, 154]), повышать тестируемость и сопровождаемость системы ([82]) и т.п. Однако также активно обсуждались и недостатки существующих

инструментариев, большей частью ориентированность на определенные этапы разработки, сложность интеграции друг с другом для формирования единого процесса [87, 125, 149] (более подробно о критике CASE-инструментов см. в Приложении А). С появлением инструментальных сред, поддерживающих полный цикл процесса разработки ПО в соответствии с выбранной методологией, в оборот входит термин *CASE-среда* или *CASE-окружение* (CASE environment). Соответственно, подходы к разработке, подразумевающие активное использование подобных инструментов, называют CASE-подходами. Подобная терминология сохраняется и по сегодняшний день. Исходя из того, что в настоящее время инструментальные средства редко используются в отрыве друг от друга и наибольшую пользу дают при совместном применении, в данной работе, если не указано явно, термины “CASE-инструмент”, “CASE-инструментарий” и “CASE-среда” используются как синонимы.

В 1990-х годах в попытке упорядочить и взять под контроль ход разработки ПО популярность набирают идеи о моделировании деятельности по созданию программных систем в виде процесса, вовлекающего работников компании в деятельность по достижению определенных бизнес-целей компании. При этом CASE-инструменты видятся как подходящее средство для автоматизации и формализации отдельных этапов процессов [83]. В отсутствии единого понимания набора и состава таких этапов многие компании пытаются выстраивать процессы, исходя из собственных особенностей и потребностей, что приводит к существенному росту количества создаваемых CASE-инструментов, большей частью несовместимых друг с другом. Ситуация меняется с появлением языка UML, разработанного консорциумом OMG для унификации представления моделей ПО. Появляются подходы, использующие UML в качестве основного языка моделирования (например, RUP [95] или MDA [120]), и инструментальные средства для реализации данных подходов в рамках производственного процесса (первым

инструментарием, поддержившим UML, был Rational Rose [19], ставший весьма успешным коммерчески).

2.3. Состав типовой CASE-среды

В настоящее время существует больше сотни различных CASE-инструментариев [23-25], различающихся как лицензией распространения, границами своей применимости, так и качеством реализации и удобством использования, однако набор инструментов, входящих в их состав, в большинстве случаев совпадает.

- Набор редакторов, с которыми работает пользователь среды, создавая модели разрабатываемого ПО. Чаще всего используются графические языки, модели на которых представляются набором диаграмм, однако нередко в состав CASE-среды входит редактор табличных данных или текстов (для задания документации, правки скриптов, конфигурационных файлов и т.п.). Для ряда инструментов также характерно наличие редактора форм, позволяющего задавать пользовательский интерфейс разрабатываемого приложения.
- Репозиторий, являющийся центральным хранилищем создаваемых моделей и предоставляющий доступ к ним всем остальным компонентам среды.
- Набор инструментальных средств, осуществляющих генерацию целевых артефактов разработки по создаваемым моделям. Тип и форма создаваемых сущностей зависят напрямую от назначения каждого конкретного CASE-инструментария. Это могут быть файлы с кодом на одном из текстовых языков программирования (например, “скелеты” отдельных частей приложения в виде функций-заглушек или полноценный код, готовый к компиляции, если в модели достаточно информации для его генерации), автоматические тесты или техническая документация для создаваемого кода, различные отчеты и документация по созданным моделям, скрипты сборки и размещения,

инсталляторы готовых версий создаваемого ПО и многое другое, для автоматизированного создания чего могут использоваться CASE-среды. В эту же категорию можно отнести инструменты автоматических или автоматизированных преобразований моделей (например, для осуществления рефакторингов моделей или выполнения над ними часто повторяющихся рутинных операций).

- Механизмы циклической разработки ПО (round-trip engineering), позволяющие дополнять “вручную” код, автоматически сгенерированный по моделям, так, чтобы изменения остались в силе при последующих перегенерациях. Стоит отметить, что для CASE-средств, основанных на UML, наличие подобных инструментов является крайне важным, поскольку UML является языком проектирования, и сгенерировать полноценный код возможно далеко не по всем UML-моделям.
- Средства обратного проектирования (reverse engineering), позволяющие строить модели по целевым артефактам (документации, коду и т.п.).
- Средства эмуляции и отладки создаваемых систем или их частей. Для ряда программных систем отладка может происходить посредством генерации исходного кода и отладки его с помощью соответствующих средств (например, запуска программ на языке C в отладчике gdb), однако часто процесс отладки целевой системы может происходить и на уровне моделей (например, при дороговизне или требований особых условий для реального запуска системы). В этом случае инструментальная среда может выдавать пользователю наглядную информацию о ходе процесса исполнения (например, подсвечивая исполняемую в текущий момент часть модели).
- Инструменты поддержки процесса командной разработки. Может быть поддержана как синхронная командная работа (например, одновременное изменение одной и той же диаграммы проектировщиками с разных

компьютеров посредством сетевого доступа к центральному репозиторию), так и асинхронная (например, посредством обмена изменениями в моделях между CASE-средствами с помощью системы контроля версий).

- Набор верификаторов и анализаторов создаваемых моделей, обеспечивающих корректность целевого ПО и предоставляющих статистику по моделям в соответствии с используемыми в проекте метриками и т.п.
- Библиотеки шаблонов и готовых решений для набора типовых ситуаций, возникающих при проектировании выбранного типа ПО.
- Средства экспорта и импорта моделей для обеспечения возможности обмена данными между CASE-инструментами.
- Набор пользовательской документации к данной CASE-среде — руководства пользователя, учебные пособия, механизм всплывающих подсказок во время работы с системой и т.п.

2.4. Современные DSM-платформы

Существует два способа анализа DSM-платформ — можно анализировать сами metaCASE-системы как среды разработки других визуальных сред и можно анализировать те DSM-решения, которые получаются на выходе подобных систем. В данной работе нас больше интересует второй способ, поэтому DSM-платформы будут рассматриваться не с точки зрения методологии или процесса создания визуальных языков и инструментов для них [145], а с точки зрения инструментальных средств в составе DSM-платформ, которые позволяют создавать те или иные инструменты в DSM-решениях.

В настоящее время существует целый ряд DSM-платформ, являющихся как коммерческими, так и исследовательскими разработками. Рассмотрим возможности наиболее зрелых из них, а затем сформулируем требования к современной DSM-платформе.

2.4.1. MetaEdit+

MetaEdit+ [15], на наш взгляд, на данный момент является наиболее зрелым коммерческим продуктом в данной области. По сути эти целых два инструментария: MetaEdit+ Modeler, являющийся гибко настраиваемой CASE-средой, и MetaEdit+ Workbench — DSM-платформа, с помощью которого создаются предметно-ориентированные решения на основе MetaEdit+ Modeler.

Являясь поначалу академической разработкой коллектива университета Jyväskyla, Финляндия (по данным [102], данной исследовательской группой на 2008 год было опубликовано более 150 статей), в 1990-х годах продукт становится коммерческим и активно применяется в индустрии до сегодняшнего дня. На момент написания текста текущей является версия MetaEdit+ 5.1.

MetaEdit+ является продуктом с почти двадцатилетней историей и представляет собой пример инструмента, очень хорошо решающего базовые задачи DSM-подхода: создание новых графических языков с помощью метамоделирования, быстрая разработка генераторов, автоматическое создание графического редактора путем конфигурации платформы метамоделью языка, наличие сетевого репозитория с многопользовательским доступом и т.п. Однако то, что MetaEdit+ является закрытым коммерческим продуктом, не дает научному сообществу и промышленным компаниям полноценно расширять имеющийся инструментарий (только с помощью открытого API). А большое количество коммерческих клиентов делает продукт консервативным и негибким к новым тенденциям в области. В результате создаваемые DSM-решения обладают лишь базовым функционалом.

2.4.2. Microsoft Modeling SDK

Другим распространенным коммерческим продуктом в данном направлении является технология Modeling SDK компании Microsoft [76]. Проект развивается с 2003 года и представляет собой надстройку для среды разработки Visual Studio,

позволяющая с помощью метамоделирования задавать предметно-ориентированные решения, интегрируемые с данной средой. Visual Studio является профессиональной средой программирования, поддерживающей целый ряд текстовых языков и технологий, и Microsoft Modeling SDK призвана дополнить существующие в Visual Studio средства разработки также и инструментами, основанными на диаграммах.

Microsoft Modeling SDK предоставляет своим пользователям все стандартные средства для метамоделирования в широком смысле: редактор абстрактного синтаксиса, редактор форм, средства задания шаблонов генерации. Особенностью технологии является то, что зачастую требуется ручное кодирование на C# – например, для задания нетривиальных фигур для элементов, для задания ограничений на модели, для определения управляющих конструкций при задании правил генерации кода и т.п.

Технология Microsoft Modeling SDK является довольно зрелой и хорошо подходит для программистов, использующих стек технологий и инструменты Microsoft, для остальных же ее применимость видится сомнительной – создание нетривиальных DSM-решений практически неизбежно потребует навыков владения Visual Studio и программирования на C#.

2.4.3. Eclipse Modeling Project

Eclipse Modeling Project (EMP) является открытым проектом, развиваемым при участии академических и промышленных организаций с начала 2000-х годов. Проект базируется на основе платформы Eclipse и фактически состоит из нескольких десятков более мелких проектов, каждый из которых имеет свою направленность. Нередки случаи, когда создаются новые проекты с единственной целью интеграции между собой инструментов, созданных в рамках других проектов (например, Modeling Amalgamation Project [17]). Данная платформа по факту является самой популярной площадкой для исследований в области модельно-ориентированной разработки ПО, однако на ее основе созданы также несколько известных

коммерческих систем визуального моделирования (например, Borland Together [8] или Rational Software Architect [18]).

Платформа имеет большое количество разработчиков и активно развивается, регулярно появляются новые проекты. Однако, порог вхождения для использования данной платформы довольно большой. Это происходит потому, что проекты зачастую развиваются отдельно и независимо друг от друга. Каждая группа разработчиков выпускает документацию только по своему проекту (в виде документации онлайн или научных публикаций). Актуальной информации по взаимодействию инструментов, созданных в разных проектах, крайне мало (редкими исключениями можно считать [44] или [88]). При этом, учитывая темпы развития проектов, составляющих EMP, информация теряет актуальность довольно быстро, в том числе и проектная документация, представленная на веб-сайтах проектов. Все это приводит к тому, EMP является проектом, предоставляющим широкий спектр возможностей для хорошо разбирающихся в платформе разработчиков, при этом не очень дружественным для начинающих.

2.2.4. AToM^3

Платформа AToM^3 также является некоммерческой академической разработкой, создававшейся для исследований в области трансформации моделей. Проект открытый и разрабатывался сотрудниками университета Макгилла (Квебек, Канада) при участии исследователей из Мадридского автономного университета. На данный момент проект закрыт, авторы занимаются разработкой платформы для текстовых предметно-ориентированных языков MetaDepth [14].

Платформа предоставляет пользователю стандартные инструменты метамоделирования, особенностью AToM^3 является мощный механизм трансформации моделей, основанный на графовых грамматиках. Определяя трансформации, автор языка может задавать преобразования между языками, определять генераторы кода и создавать интерпретаторы/отладчики диаграмм.

Среда активно использовалась авторами для создания предметно-ориентированных решений, демонстрирующих использование описанных механизмов, однако после 2009 года проект по сути не развивается. Представляя собой интересную академическую разработку, платформа AToM^3, на наш взгляд, не может считаться достаточно зрелой для использования в коммерческих проектах.

2.4.5. MetaLanguage

Проект MetaLanguage [31, 45, 151] развивается силами исследователей из Высшей Школы Экономики в городе Пермь. Платформа реализует традиционный набор средств метамоделирования, позволяя создавать графические редакторы, репозиторий, валидаторы, средства трансформации моделей и др. Особенностью системы является возможность изменения синтаксиса языка во время работы с ним. Это достигается за счет интерпретации метамодели языка при создании моделей с помощью данного языка (подобный подход также реализован в платформе MetaEdit+, рассмотренной выше).

Исходные коды системы MetaLanguage недоступны, готовую версию для скачивания в рамках данного исследования также найти не удалось, поэтому сделать выводы относительно применимости ее в промышленной или академической разработке не представляется возможным.

2.4.6. Сравнительный анализ

Завершает данный обзор сравнительный анализ DSM-решений, которые могут быть созданы на базе рассмотренных платформ (см. табл. 1). Анализ таблицы показывает, что из четырех представленных аналогов самая богатая функциональность получаемых DSM решений у платформы Eclipse. Однако платформа крайне сложна в применении. Видятся актуальными работы как по упрощению процесса использования DSM-платформ, так и упрощение самих решений, созданных на их основе – разработка средств, позволяющих сделать их

более удобными при использовании непрофессионалами.

Таблица 1. Сравнение возможностей сред разработки, создаваемых на базе некоторых DSM платформ

	MetaEdit+	EMP	AToM^3	MS Modeling SDK	QReal
Условия распространения	Закрытый коммерческий продукт	Открытое ПО (EPL)	Открытое ПО	Закрытый коммерческий продукт	Открытое ПО (Apache License Version 2.0)
Отделимость решения от платформы	Да, в рамках Modeler	Да	Нет	Нет	Нет
Кроссплатформенность	Windows, Linux, Solaris, Mac OS X	Windows, Linux, Solaris, Mac OS X	Windows, Linux, Solaris, Mac OS X	Windows	Windows, Linux, Solaris, Mac OS X
Особенности редакторов диаграмм	Работа с таблицами и матрицами	Автоматическая раскладка элементов и другие проекты	Механизм автодополнения моделей	Только базовая функциональность редактора	Распознавание жестов, поддержка ряда эвристик моделирования
Поддержка командной разработки	Сетевой репозиторий	Посредством систем контроля версий	Нет	Посредством интеграции с Visual Studio	Посредством систем контроля версий
Совместное использование языков	Да	Да	Да	Нет	Да
Средства трансформации моделей	Нет	Да	Да	Реализуется кодированием на C#	Да
Визуализация отладки	Да, путем интеграции со сторонними средами	Да, внутренний механизм	Да, внутренний механизм	Реализуется кодированием на C#	Да, внутренний механизм
Верификаторы и анализаторы моделей	Ограничения только на синтаксис языка	Заданные на языке OCL или Java	Реализованные на Python	Реализуется кодированием на C#	Заданные с помощью языка ограничений или реализованные на C++

Последним столбцом в таблице представлена платформа QReal, разработка которой велась в рамках данного исследования и которой посвящены последующие главы работы.

2.5. Требования к современной DSM платформе

Минимальный набор инструментальных средств, входящих в предметно-ориентированное решение, которое может быть приносить пользу на практике, составляет графический редактор и средство трансформации создаваемых моделей в описания на другом языке (например, генератор кода в случае текстового целевого языка). В соответствии с [102], простейший редактор должен быть способен сохранять и загружать модели с диска; добавлять и удалять экземпляры объектов; соединять объекты с помощью связей; отображать и раскладывать (автоматически или с помощью перетаскивания) элементы и связи между ними на диаграммах; изменять свойства уже существующих объектов и связей между ними. Более функционально сложные (и востребованные пользователями) DSM-решения также могут содержать большую часть средств, описанных в разделе 2.3.

Современная DSM платформа должна обладать инструментами для автоматизированного создания решений, обладающих следующей функциональностью.

- Набор редакторов (графических и текстовых), реализующих набор типовых операций:
  - механизм отмены совершенных действий;
  - копирования-вставки элементов диаграмм;
  - средства декомпозиции и построения иерархий моделей;
  - поддержка нескольких визуальных языков и интеграции создаваемых с их помощью моделей.
- Репозиторий для хранения моделей.

- Средства преобразования моделей (как в другие модели, так и в текст).
- Средства интерпретации/отладки создаваемых диаграмм.
- Средства многопользовательской работы.
- Анализаторы и верификаторы моделей.
- Средства интеграции со сторонними инструментами (например, средства экспорта и импорта моделей или открытый API для взаимодействия с компонентами системы).

Отметим, что важный для средств на основе UML механизм циклической разработки ПО (round-trip engineering) в рамках DSM-парадигмы теряет смысл благодаря тому, что полноценный код разрабатываемой системы генерируется автоматически и последующее его «ручное» редактирование запрещается. Использование средств обратного проектирования (reverse engineering) при DSM подходе в литературе не нашло своего отражения, хотя не видится технических сложностей для создания их в metaCASE-средствах (по аналогии со средствами задания генераторов кода или трансформаций моделей).

2.6. Заключение

Выполненный обзор показал существенный интерес к области предметно-ориентированного моделирования как со стороны промышленности, так и научных исследований. При этом промышленные решения хорошо решают задачу создания только самых базовых инструментов (чаще всего это графические редакторы, хранилища моделей и генераторы кода), в то время как исследовательские разработки далеки от промышленного уровня стабильности работы и простоты использования разработчиками решений (чем больше функциональности предоставляет платформа, тем большее количество её компонент разработчик должен увязать вместе). Также данный обзор формулирует требования к функциональности современных CASE-систем. Данное исследование призвано

исследовать вопрос создания единой платформы, позволяющей строить полнофункциональные решения в соответствии с этими требованиями.

Глава 3. Подходы к повышению удобства моделирования

3.1. Введение

Как показано в Приложении А, по мнению большого количества исследователей, одной из главных причин отказа от массового использования средств визуального моделирования и проектирования является их чрезмерная сложность, перегруженность функциональностью и неудобство в использовании в повседневных процессах разработки. Неудобство использования таких инструментов складывается в первую очередь из неудобства самого инструментария (много рутинных операций, сложный для понимания интерфейс, нетривиальный процесс), так и из неудобства используемого языка моделирования. Упомянутые работы относятся к 90-м годам XX века, т.е. к CASE-средствам, основанным на UML и крупных методологиях разработки ПО (например, Rational Unified Process, RUP), которые создавались исходя из соображений общности и единообразия, а не удобства использования (возможно поэтому ожидаемый методологами CASE-подхода массовый переход на визуальное программирование в девяностых годах так и не произошел). DSM-решения призваны исправить ситуацию с удобством языков моделирования, предлагая проектировщикам оперировать не абстрактными классами или объектами, а вполне конкретными понятиями из целевой предметной области. Однако, проблема сложности и неудобства визуальных сред разработки остается и при использовании предметно-ориентированных языков: проектировщику в процессе моделирования приходится совершать большое число рутинных действий, что не лучшим образом сказывается на привлекательности и технологии в целом.

Данная глава посвящена вопросам улучшений удобства визуального

моделирования с точки зрения используемых инструментов. В частности, рассматривается новый метод для создания инструментов распознавания жестов в диаграммных редакторах.

3.2. Использование распознавания жестов мышью

Эффективность любого используемого инструмента определяется тем, насколько удобно и быстро он позволяет выполнять те операции, для которых предназначен. В процессе разработки моделей одними из наиболее часто выполняемых действий над объектами на диаграммах являются их создание и удаление. В большинстве визуальных сред разработки для того, чтобы создать нужный объект на диаграмме, необходимо найти его либо на панели инструментов, либо выбрать в меню, а затем указать место на диаграмме, где бы мы хотели этот элемент разместить. Также в большинстве инструментариев возможен вариант создания объектов «перетаскиванием» (drag-and-drop) их из палитры. При этом надо учитывать, что количество видов диаграмм и объектов в палитре каждой диаграммы может быть довольно большим (например, 13 видов диаграмм в одном только UML 2). Не всегда возможно оставить в палитре только специфичные для данного типа диаграмм элементы, поскольку может возникать задача быстрого прототипирования с использованием нескольких языков на одной диаграмме. То есть даже для такой базовой операции, как создание нового элемента, разработчику нужно совершить не только набор чисто механических действий, но ещё и, скажем, вспомнить, на какой вкладке палитры или в каком меню находится нужный ему элемент, тем самым переключая контекст с продумывания иерархии создаваемых моделей на особенности использования выбранного инструмента. Нам кажется, что данную операцию можно и нужно автоматизировать, причём её нужно сделать максимально удобной для пользователей CASE-средств.

В качестве замены традиционного интерфейса для создания элементов и связей

между ними предлагается использовать жесты мышью — определенные траектории, обрисованные курсором мыши с неким модификатором (в нашем решении это зажатая правая клавиша мыши). Жесты мышью уже используются для управления некоторыми программными средствами (например, в веб-браузерах или играх), однако особенность данного решения в том, что число элементов, которые хочется создавать таким способом, гораздо больше, чем число простейших жестов (вверх, вниз, вправо, влево) и их тривиальных комбинаций. Возможным решением в таких случаях было бы по жесту предоставлять список команд, из которых пользователь может выбрать. Но если список будет слишком большим или будет вызываться при каждом жесте мышью, то смысл введения жестов пропадает, так как большинство инструментов уже предоставляет пользователю список объектов, которые с помощью операции drag-and-drop можно перетаскивать на рабочее поле.

Данная идея уже нашла свое воплощение в инструментах моделирования Visual Paradigm [21] и Ideogramic UML [13, 78]. К примеру, в Visual Paradigm выделяются три вида жестов — для создания объектов, для создания связей и для вызова определенных команд. Каждый жест задается набором направлений (вверх, вниз, влево, вправо), значение жеста помимо последовательности направлений определяется также и текущей диаграммой. Не все элементы палитры имеют отдельные жесты для их создания, лишь самые часто употребляемые, по мнению авторов инструмента. Имеется контекстная справка, анимировано отображающая рисование жеста, однако сами жесты сильно отличаются от формы создаваемых ими фигур, и для эффективной работы требуется время на привыкание и заучивание жестов. Стоит также отметить, что описанные среды не предоставляют возможности изменять набор жестов и связывание их с выполняемыми действиями. К тому же реализация этих инструментов является закрытой, что мешает их явной параметризации и расширению.

Подобный подход плохо применим к DSM-платформам, поскольку процесс их

разработки разнесен во времени с процессом создания DSM-решений на их основе: необходимо дать пользователям DSM-платформы возможность самостоятельно задавать наборы жестов и связывать их с выполняемыми действиями. Продолжая мысль об автоматизации необязательных шагов, из которых состоит создание нового DSM-решения, необходимо иметь в составе платформы средства автоматического создания инструментов распознавания жестов для создаваемых языков, включающих в себя генерацию жеста для каждого элемента языка, а также интеграцию механизма распознавания с создаваемым визуальным редактором.

В соответствии с обозначенными наблюдениями был предложен новый метод, призванный ускорить и автоматизировать процесс создания такого рода инструментов в DSM-решениях. Основной принцип данного метода состоит в том, что создание инструментов распознавания жестов мыши для редактора нового языка происходит автоматически по описанию конкретного синтаксиса без участия разработчика этого языка.

Основные этапы предлагаемого метода представлены ниже.

1. Формальное описание внешнего вида элементов языка.
2. Генерация по этим описаниям «идеального жеста» для каждого элемента.
3. Выполнение жеста мыши при моделировании с использованием языка.
4. Сопоставление введённого жеста с набором «идеальных жестов» и создание на диаграмме наиболее подходящего элемента.

Отметим, что, хотя первый этап требует явных действий разработчика языка, внешний вид элементов языка неизбежно задаётся им в процессе создания редактора (независимо от используемой методологии разработки языка). В рамках предлагаемого подхода используются уже созданные ранее описания конкретного синтаксиса, дополнительных действий от разработчика языка не требуется.

Ключевым понятием предлагаемого метода является «идеальный жест» – некий эталон траектории движения мыши, который ставится в соответствие каждому

объекту на диаграмме и с которым будет осуществляться сравнивание вводимых пользователем жестов в процессе распознавания. По умолчанию для всех элементов языка, чье графическое представление задается в векторном виде с помощью редактора форм QReal, идеальный жест генерируется автоматически максимально похожим на фигуру, представляющую элемент на диаграмме. Для элементов, чье графическое представление задается сторонним растровым изображением, идеальный жест может быть задан с помощью специального графического редактора (с помощью этого же редактора можно при необходимости задать произвольный идеальный жест любому элементу). При генерации DSM-решения в его состав будут входить дополнительные компоненты, хранящие идеальные жесты для всех элементов языка и поддерживающие процесс распознавания как таковой.

Во время работы с созданным языком пользователи DSM-решения выполняют жест с зажатой правой кнопкой мыши, графический редактор записывает соответствующую траекторию и передает модулю распознавания, который, в свою очередь, сравнивает траекторию введенного жеста с набором идеальных жестов элементов. При успешном сравнении (один из идеальных жестов “похож” на введенный с заданной точностью) на диаграмме создается элемент, ассоциированный с найденным идеальным жестов. Для ассоциаций языка специальных жестов не создается, создание ассоциации происходит, если пользователь осуществит жест произвольной формы, начинающийся и заканчивающийся на уже существующих на диаграмме элементах. В таком случае специальная компонента QReal проверит, какого типа связи возможны между этими двумя элементами, и предложит выбрать одну из списка (если возможна только связь одного типа, она будет создана автоматически).

В настоящее время в QReal реализована поддержка как одноштриховых, так и многоштриховых жестов. Более подробно о реализованных алгоритмах распознавания см. в [32] и [132].

3.3. Особенности графических редакторов

3.3.1. Линкеры элементов

Помимо распознавания жестов в QReal существует еще одна возможность быстрого создания связей между элементами — использование так называемых линкеров элементов. Линкеры — небольшие элементы графического интерфейса, отображаемые рядом с элементами на сцене при выделении их мышью (см. рис. 6).

Рисунок 6. Элемент «Начало» графического языка QReal:Robots с отображаемым линкером

При нажатии на линкер левой кнопкой мыши (и не отпуская ее при движении) из элемента начинает «вытягиваться» связь, следующая за курсором мыши. При отпускании левой кнопки мыши возможны два варианта. Если курсор находился над уже существующим элементом на диаграмме, будет создана связь между элементом, которому принадлежит линкер, и элементом, на котором остановился курсор мыши. Если же левая кнопка мыши была отпущена при нахождении курсора на пустом пространстве диаграммы, будет показано служебное окно (см. рис. 7), с помощью которого пользователь может либо удалить данную связь, либо убрать данное меню (и оставить связь не присоединенной ни к какому элементу), либо выбрать из списка элемент, который может быть соединен с данным текущим видом связи (с учетом её направленности). В последнем случае элемент выбранного типа будет создан и присоединен к другому концу новой связи. Подобный список элементов в меню линкера создается для каждого элемента автоматически во время выполнения кода DSM-решения на основе возможных связей между элементами, заданных в метамодели языка.

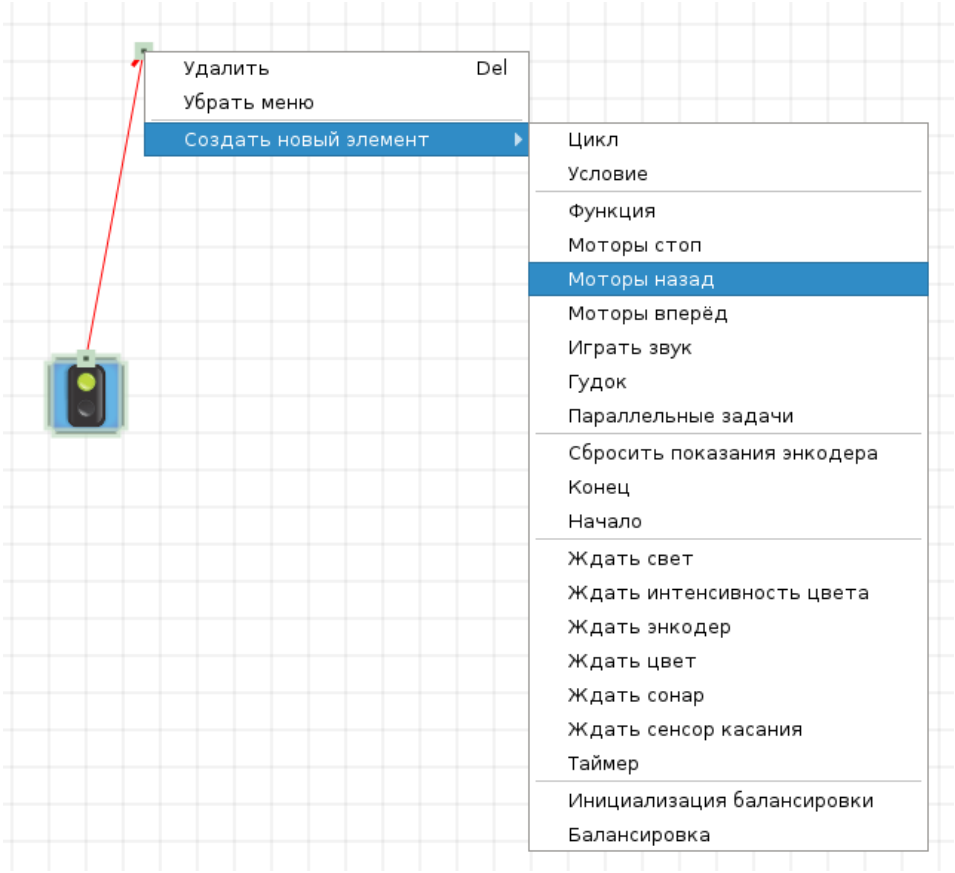

Рисунок 7. Меню линкера элемента «Начало» языка QReal:Robots

В случае, когда из одного элемента может исходить несколько типов связей, рядом с элементом будет находиться несколько линкеров, представляемых кружками разных цветов. Возможен также режим редактора, когда рядом с каждым линкером отображается также и имя связи, этому линкеру соответствующее (по умолчанию данные режим в QReal выключен во избежание загромождения диаграмм большим количеством служебной информации).

3.3.2. Элементы управления, встроенные в объекты на сцене

Любые нетривиальные элементы языка имеют набор атрибутов (свойств) определённых типов. Значения этих атрибутов в средствах моделирования обычно изменяются либо в специальном окне, появляющемся при нажатии на элемент, либо в традиционном для сред разработки редакторе свойств. Однако, частое обращение к

редактору свойств или использование дополнительных окон вносит в работу проектировщика много дополнительных действий, которые, на наш взгляд, также имеет смысл сокращать. Другая проблема состоит в том, что часто единственным способом увидеть значение того или иного свойства элемента является выделение (в случае редактора свойств) или двойное нажатие на этот элемент на диаграмме (в случае специального окна свойств). Обе этих проблемы, на наш взгляд, решила бы возможность отображения ряда свойств прямо на диаграмме рядом с элементом (по аналогии с тем, как многие среды моделирования включают имя элемента в его графическое представление на диаграмме), а также изменения их значений с помощью этих же средств.

С помощью редактора конкретного синтаксиса QReal возможно включать в графические представления элементов на диаграммах стандартные элементы управления — выпадающие списки, поля для ввода текста, флажки и ряд других. Автор языка решает сам, какие свойства будут отображаться на диаграмме всегда, отображение каких может отключаться в настройках QReal, а какие не будут показаны. Это позволяет добиться наибольшей наглядности и удобства работы со свойствами элементов.

На рис. 8 представлен внешний вид элемента “Ждать сонар” языка, используемого в DSM-решении QReal:Robots. Как видно из рисунка, три свойства (порт, к которому присоединен датчик сонар, расстояние, на котором датчик будет отслеживать предметы, и факт наличия или отсутствия предметов в заданной области) отображены на диаграмме вместе с пиктограммой элемента, причём прямо же на диаграмме можно изменить значение свойства перечислимого типа с помощью выпадающего списка.

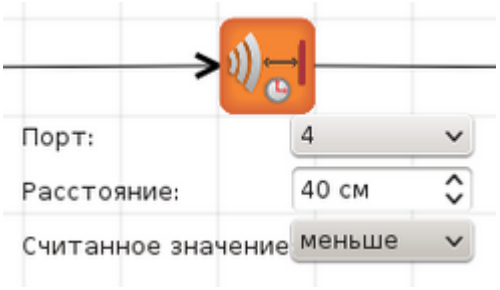

Рисунок 8. Элемент языка QReal:Robots со встроенными элементами работы со свойствами

3.3.3. Эвристики языка ДРАКОН

Вопросы эргономики визуальных языков были исследованы авторами языка ДРАКОН [34-36], который основывается на языке блок-схем, дополняя его набором эвристик, по мнению автора языка, способствующих повышению читаемости и понимаемости создаваемых с помощью данного языка диаграмм. К сожалению, автор не проводил количественных замеров на тему того, насколько проще описывать алгоритмы на языке ДРАКОН по сравнению с традиционными блок-схемами, и ограничивается лишь общими рассуждениями о базовых особенностях восприятия информации человеком (например, восприятие информации справа-налево и сверху-вниз представителями европейской культуры).

Несмотря на то, что понятие удобства использования является крайне субъективным и, вследствие этого, слабо формализованным, можно выделить определенные объективные метрики, которые будут описывать данную характеристику языка и процесса его использования. Как было описано выше, в рамках данной работы ставилась цель минимизировать число механических действий, производимых пользователем для выполнения определенных операций. На наш взгляд, например, сокращение числа нажатий кнопок мыши в процессе моделирования в несколько раз может позволить улучшить отношение пользователей к среде моделирования и снизить дискомфорт от ее использования.

Язык ДРАКОН был проанализирован на предмет того, насколько его эвристики

могли бы быть применимы в других визуальных языках, в основе которых лежит поток управления. В результате ряд таких эвристик был обобщён и реализован на уровне инструментов платформы QReal, что позволило потенциально использовать их в редакторах всех поведенческих языков, создаваемых на базе платформы. Вкратце рассмотрим некоторые из них и то, как они сокращают число совершаемых проектировщиком операций.

3.3.3.1. Возможность создания разделяемых ассоциаций

При расширении существующего алгоритма часто бывает необходимо вставить определенный элемент между уже существующими элементами, соединенными определенным видом связи (рис. 9, а). Традиционно для этого нужно совершить следующую последовательность действий:

- создать требуемый элемент рядом с тем местом, куда необходимо его вставить (рис. 9, б);
- отсоединить связь, которую расширяет данный элемент и присоединить ее к новому элементу (рис. 9, в);
- создать новую связь, соединяющую новый элемент с уже существующим (рис. 9, г);
- расположить элементы на диаграмме необходимым образом (рис. 9, д);

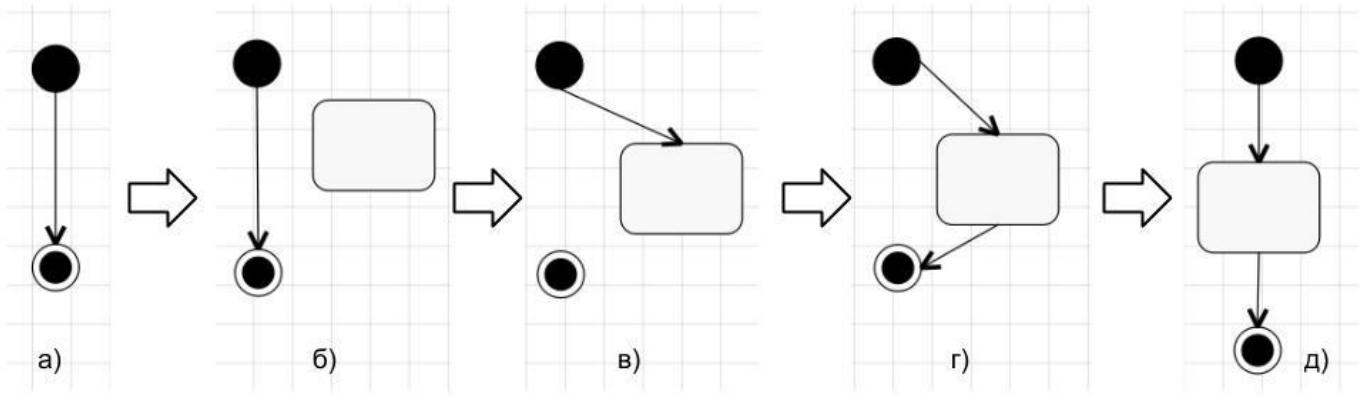

Рисунок 9. Шаги процесса вставки элемента «внутрь» существующего алгоритма

Для упрощения этого процесса был реализован механизм разделяемых ассоциаций: при перетаскивании элемента из палитры на уже существующую связь, соединяющую два других элемента, все перечисленные выше действия выполняются автоматически, визуально разделяя ассоциацию на две и вставляя новый элемент между ними (см. рис. 10). Стоит отметить, что при выполнении подобного действия во избежание наложения элементов друг на друга производится сдвиг всех элементов ниже и правее вставляемого на размер этого элемента.

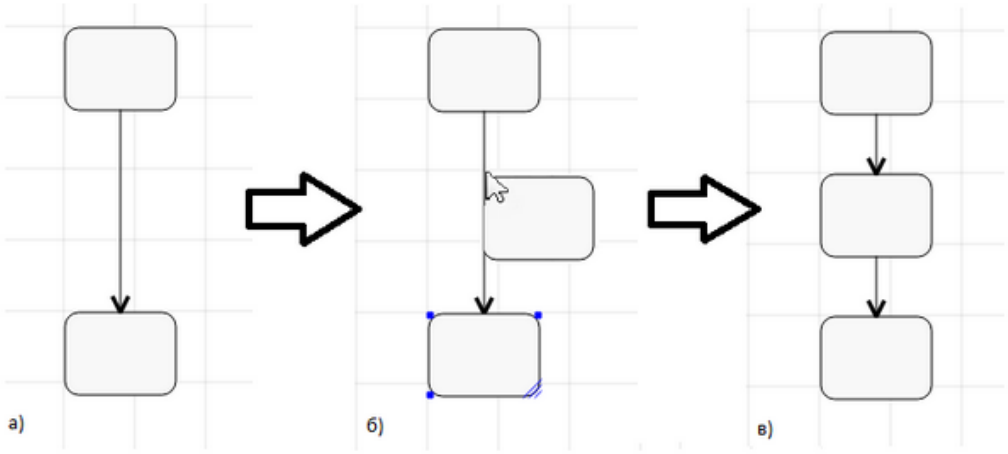

Рисунок 6. Пример вставки элемента “внутрь” разделяемой ассоциации

Также возможен и обратный процесс — при удалении элемента, имеющего входящую и исходящую связь, одна из этих связей будет удалена, а оставшаяся будет соединять элементы, к которым был присоединен удаленный (переход от рис. 18, в к рис. 18, а).

3.3.3.2. Создание групп элементов

Зачастую в визуальных языках присутствуют элементы, которые на диаграммах могут использоваться вместе с другими элементами, формируя более высокоуровневые элементы-шаблоны с заданной семантикой. Например, блок “Цикл” может состоять из нескольких блоков: блок начала цикла и блок конца цикла с некоторым числом блоков между ними. Или блок распараллеливания диаграмм

активностей UML 2.x должен начинаться и заканчиваться определенным элементом распараллеливания и синхронизации потоков выполнения алгоритма. Создание этих элементов перетаскиванием из палитры по отдельности видится чрезмерно избыточным, при этом проектировщик вполне может ошибиться при организации этих элементов в шаблон, создав семантически некорректную диаграмму.

Для решения этой проблемы в метаязык была добавлена возможность описания групп элементов как отдельных блоков в палитре элементов, а также ядро платформы QReal было соответствующим образом расширено для того, чтобы давать возможность перетаскивать эти новые блоки на диаграмму и создавать при этом сразу все элементы, входящие в группу. Например, на рис. 11 приведен пример вставки группы элементов, состоящей из двух элементов ActionNode языка блок-схем, соединенных ассоциацией в виде ломаной линии. Вставка происходит «внутрь» разделяемой связи, элемент FinalNode при вставке группы соответствующе сдвигается вниз.

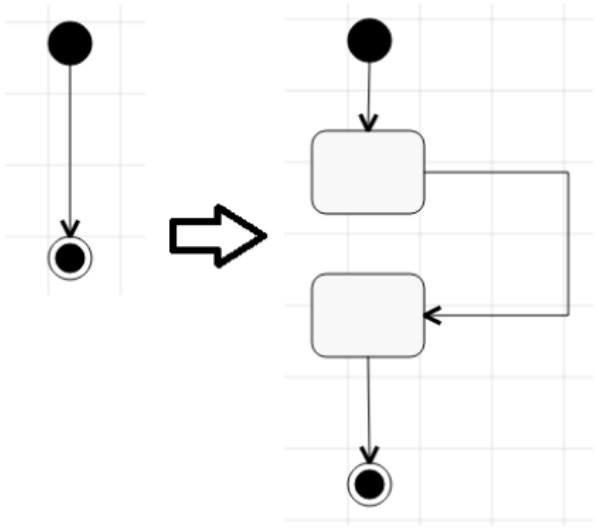

Рисунок 11. Вставка группы элементов “внутрь” ассоциации

3.4. Заключение

В данной главе рассмотрены способы упрощения работы с диаграммным

редактором платформы QReal. Самым важным результатом здесь является метод автоматического создания компонент распознавания жестов DSM-решений. Подобные компоненты создаются без какого-либо участия разработчика по формальному описанию визуального представления элементов языка и позволяют автоматизировать наиболее часто повторяющиеся при моделировании действия: создание и удаление элементов на диаграммах и связей между ними. Также в редактор была добавлена реализация некоторых эвристик языка ДРАКОН, которые позволили уменьшить число операций, необходимых при добавлении элементов на уже существующие диаграммы и при создании групп элементов.

Глава 4. Средства задания исполнимой семантики

4.1. Введение

В данной главе рассматривается метод, позволяющий создавать отладчики и интерпретаторы моделей посредством задания операционной семантики визуальных языков. Кратко описываются существующие для этого формализмы (подробный обзор см. в Приложении В), приводится описание подхода, выбранного к реализации в данной работе. Рассматриваются особенности реализации подхода в рамках платформы QReal.

4.2. Семантика языков

В теории языков программирования выделяют четыре основных типа семантик: операционная, денотационная, аксиоматическая и трансляционная. Подобное деление существует и для визуальных языков.

Денотационная семантика была описана в работе [142]. 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 это соответствие задаётся при помощи λ-исчисления. Данный тип семантик является математически строгим и формальным, но в чистом виде порой не обладает достаточной степенью понятности для неспециалистов.

Операционная семантика также является математически строгой и бывает двух типов: структурная операционная семантика (семантика малого шага, [101]) и естественная семантика (семантика большого шага, [135]). Структурная операционная семантика состоит из набора правил, при помощи которых

исполнение конкретной программы на данном языке будет 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 [91] 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 [148]. Они состоят из набора правил преобразования моделей, с помощью которых модель на исходном языке переводится в модель на другом языке, для которого уже задана семантика исполнения. Например, часто это сводится к генерации кода на языке общего назначения, для которых 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.

В приложении В и [42] рассмотрен также ряд конкретных подходов к заданию семантики визуальных языков, предложенных в различных работах и использующихся в программных средствах. Средства задания семантики рассматриваются в первую очередь как инструмент автоматизированного создания пошаговых интерпретаторов и отладчиков целевых языков, а также генераторов исходного кода по визуальным моделям.

4.3. Задание исполнимой семантики в QReal

По результатам исследования и анализа существующих способов задания семантики визуальных языков был предложен метод, реализованный применительно к платформе QReal [40, 43]. Он основан на правилах преобразованиях графов, задаваемых, как в DMM-подходе и AToM^3 в графическом виде, а также имеет некоторое подобие языка действий из xUML для задания операций, осуществляемых при выполнении правила.

Рассмотрим основные особенности предлагаемого подхода.

4.3.1. Семантическая модель

При формализации семантики визуального языка и последующего использования этой семантики для исполнения моделей может понадобиться добавить элементам данного языка дополнительный набор специфических атрибутов. Например, элементам языка, описываемого формализмом сетей Петри, можно добавить дополнительный атрибут, указывающий, содержит ли данный элемент в настоящее время токен исполнения или нет. Начальному элементу диаграммы можно поставить логический атрибут, указывающий, исполняется ли данная диаграмма в настоящий момент или нет.

В предлагаемом решении для задания семантики элементов используется специальное расширение метаредактора QReal, содержащего стандартный набор средств для добавления новых атрибутов к элементам и создания новых элементов. После всех необходимых добавлений генерируется новая метамодель целевого визуального языка, содержащая все его элементы вместе с их семантикой. Эта метамодель автоматически компилируется средствами QReal в подключаемый модуль редактора и последний 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.

Разумеется, эти дополнительные атрибуты имеют смысл только в рамках работы с семантикой данного элемента и не должны пересекаться с другими атрибутами: например, семантические атрибуты не должны отображаться пользователю в редакторе свойств, а использоваться лишь при интерпретации правил семантики (например, вышеупомянутое свойство элементов языка сетей Петри будет использоваться для отслеживания перемещения токенов по модели при ее интерпретации). Любая модель на некотором визуальном языке представляется в QReal в виде логической и графической моделей, взаимосвязанных друг с другом (см. раздел 5.3.4). К данному разделению имеет смысл добавить новую семантическую модель, которая бы содержала эти новые специфические служебные атрибуты и процедуры. По аналогии с логическими и графическими моделями доступ к семантическим атрибутам элементов для интерпретаторов и генераторов должен осуществляться через специальное API репозитория (SemanticRepoApi). Также к семантике элементов возможен доступ из редактора семантики визуального языка.

4.3.2. Редактор семантики визуального языка

Редактор семантики визуального языка генерируется автоматически также на основе полученной на предыдущем шаге расширенной метамодели исходного языка. Семантика визуального языка в QReal представляет собой набор правил преобразования графов, пошаговое применение которых к графу исходной модели позволяет визуализировать процесс интерпретации или осуществить генерацию кода по модели. При исполнении модели на каждом шаге выбирается правило с наивысшим приоритетом (если таких несколько, то берётся произвольное) или случайное правило, которое может быть применено к модели, если приоритеты правил не заданы. Далее правило применяется в первом найденном подходящем месте. Принцип работы такого процесса напоминает выполнение алгоритмов Маркова на строках, а наличие приоритета позволяет упорядочить возможное

применение правил.

Редактор семантики позволяет задавать правила преобразования моделей визуально с помощью специального языка, состоящего из следующих элементов.

- Элемент SemanticsRule является контейнером для всех остальных элементов. В него помещаются левые и правые части правил, при необходимости совмещенные вместе (по аналогии с тем, как это описано в DMM-подходе, см. Приложение В). Задание факта создания и удаления элементов происходит при помощи добавления к ним меток “@new@” и “@deleted@” соответственно.
- Связь Replacement позволяет задать факт замены одного элемента на другой (с сохранением всех связей, входящих и исходящих из заменяемого элемента).
- Элемент ControlFlowMark используется в правилах для того, чтобы задать текущее положение потока управления (аналог токена в сетях Петри) — показать, какой элемент будет считаться текущим с точки зрения исполнения модели после применения данного правила. Существует два варианта его использования — либо соединение элемента ControlFlowMark с целевым элементом правила с помощью специальной связи ControlFlowLocation, либо помещение его внутрь целевого элемента как в контейнер (второй способ для ряда случаев позволяет создавать более компактные и наглядные правила). Во время выполнения модели, т.е. при применении очередного правила, происходит подсветка элементов, ассоциированных с элементами типа ControlFlowMark в рамках данного правилах.
- Элемент Initialization используется для определения типа семантики языка — будет ли она использоваться для создания интерпретатора и/или генератора кода по моделям. Также этот элемент используется для определения правил начальной инициализации процесса интерпретации, задаваемых в текстовой форме (см. раздел 4.3.3).
- Элемент Wildcard вляется унификатором элемента модели. При поиске

шаблона во время применения правила элемент Wildcard будет давать успешный результат сопоставления с любым элементом модели.

- Все элементы целевого языка также могут быть использованы при описании правил. Элемент или связь типа X, помещенный в правило, будет соответствовать любому элементу или связи типа X модели, над которой это правило применяется.

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 правиле не важны, то можно использовать унификатор узла или связи. Как было отмечено выше, 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 (см. раздел 4.3.3). Простейший вариант приоритета — целое число. В общем случае, приоритет — это значение некоторой функции, заданной на атрибутах элементов правила, т.е. приоритет правила зависит от конкретного места в исходной модели. Например, если у каждого элемента модели задан целочисленный вес, а правила в первую очередь должны применяться к подграфам с максимальным суммарным весом элементов. Данная функциональность может понадобиться, например, при визуализации различных формальных алгоритмов на графах.

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 в настройках модуля задания семантики нужно указывать разное время задержки между шагами исполнения, т.е. применения правил к модели.

Рассмотрим пример одного из правил исполнимой семантики языка блок-схем, отображенного на рис. 12.

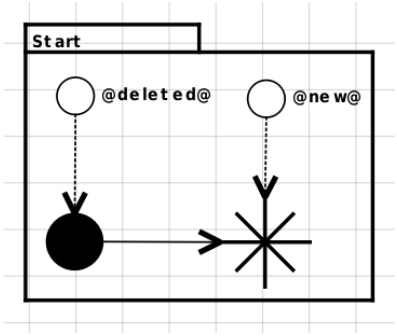

Рисунок 127. Пример правила преобразования графа модели

Данное правило позволяет начать интерпретацию модели и состоит из следующих элементов:

- элемента SemanticsRule с идентификатором “Start”, являющегося контейнером для всех остальных элементов правила;
- элемента InitialNode языка блок-схем, отображаемого на диаграмме в виде черного круга;
- элемента Wildcard, отображаемого в виде звезды;
- связи ControlFlow языка блок-схем, соединяющей элементы InitialNode и Wildcard;
- двух элементов ControlFlowMark, отображаемых незакрашенными кругами меньшего размера;
- двух связей ControlFlowLocation, соединяющих элементы ControlFlowMark с элементами InitialNode и Wildcard.

Данное правило означает следующее — необходимо найти в графе исполняемой модели элемент типа InitialNode (начальный узел блок-схемы), который в настоящее

является текущим элементом исполнения (на это указывает связь этого элемента с ControlFlowMark), и передать исполнение элементу модели, который соединен с InitialNode связью ControlFlow. Последнее осуществляется за счёт того, что элемент ControlFlowMark, ассоциированный с InitialNode будет удалён (атрибут “@deleted@”), зато будет создан новый ControlFlowMark, и он будет ассоциирован с элементом, связанным с InitialNode. При этом тип элемента, которому передастся исполнение, в данном случае совершенно не важен, поэтому используется элемент Wildcard, который при поиске шаблона в модели будет сопоставлен с элементом любого типа.

4.3.3. Реакция на применение правила

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

На каждое исполнение модели создаётся новый экземпляр интерпретатора выбранного текстового языка[6], использующегося для задания поведения элементов, разного рода ограничений на элементы и реакций на применение правил. Таким образом, определённые в этом интерпретаторе сущности (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 элементе Initialization и гарантированно будет исполнен до применения первого подходящего правила преобразования графов.

[6] В настоящее время в качестве текстового языка может быть использован Python или некий собственный язык, используемый в редакторе блок-схем, созданном на базе QReal (см. раздел с описанием блок-схем). Также ведется работа над реализацией возможности использования языка QtScript, входящего в состав инструментария Qt.

Ограничения на применение правил нужны для задания более сложных условий, чем соответствие типу или равенство определенному значению заданного атрибута. Такие ограничения представляют собой функцию на выбранном интерпретируемом языке, возвращающую значение логического типа и имеющую доступ к значениям атрибутов элементов правила (а также к прочим сущностям, определённым при инициализации или в ходе интерпретации при выполнении других правил). Если ограничение на соответствующем правилу участке модели возвращает ложное значение, то это правило не может быть применено для данного подграфа. Например, в случае моделей на языке блок-схем для создания правила перехода потока исполнения с условного элемента далее по одной из веток, соответствующей истине, в ограничение на применение правила можно поместить значение типа связи, которая из него выходит и условие равенства истине, записанного в атрибуте соответствующего элемента.

Для исполнения поведения элементов правила и организации их взаимодействия между собой, а также организации взаимодействия правил между собой было введено понятие реакции на применение правила. Это процедура на выбранном 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.

4.3.4. Модуль работы с графами

В предложенном подходе модель на визуальном языке рассматривается как 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. Основными подзадачами, которые были

решены при реализации данного модуля, являются создание алгоритма поиска подграфа в графе и корректное размещение новых элементов в модели. Разберем их подробнее.

4.3.4.1. Поиск подграфа модели

Существует ряд открытых программных средств, предназначенных для применения преобразований графов и автоматической верификации моделей относительно итоговой системы преобразований. Среди них GROOVE [12, 137], AGG [155], GenGED [11, 65] и некоторые другие (см., например, обзор подобных средств [46]). Однако, интеграция их на уровне исходного кода в платформу QReal видится проблематичной из-за несоответствия языков реализации, а использование их в качестве стороннего инструмента сделает платформу QReal тяжеловесной и менее однородной. Кроме того, большинство средств нацелены на специалистов в области преобразования графов и плохо подходят к использованию непрофессионалами. В результате было принято решение сделать собственную реализацию на основе существующих алгоритмов.

В итоге для организации поиска заданного шаблона в модели был предложен итеративно-рекурсивный алгоритм, накопление результата в котором будет происходить постепенно. Данный алгоритм находит все вхождения заданного шаблона в модели, промежуточным результатом работы будет одно найденное вхождение. Более подробное описание алгоритма см. в приложении С.

4.3.4.2. 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 репозитория, предоставляемого QReal, но для корректного его исполнения необходимо сделать еще несколько дополнительных действий.

При создании и замене элементов необходимо, во-первых, установить созданному элементу значения атрибутов, равные значениям соответствующих элементов из правила. Также этот вновь 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 повторно подсоединить все связи, одним из концов которых был заменяемый элемент.

Также возникает вопрос, нужно ли удалять все связи элемента с другими 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 (в этом случае добавленный элемент имеет те же координаты, что и заменяемый). Создание же нового элемента требует расчёта его места на диаграмме. В текущем решении новые элементы либо просто располагаются правее всех остальных на диаграмме, либо после добавления элемента осуществляется автоматическое перераскладывание элементов на диаграмме (позиции элементов рассчитываются с помощью утилиты dot из распространенного пакета визуализации графов graphviz [10]).

4.4. Архитектура реализованного решения

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, были вынесены в модуль qrutils (см. раздел 5.3), чтобы другие компоненты системы (например, модуль задания и осуществления рефакториингов моделей, см. раздел 5.4.7) при необходимости также могли ими воспользоваться.

Для корректной работы реализованного визуального интерпретатора в запущенной системе должен присутствовать модуль метаредактора, 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, ограничениями 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.

4.5. Заключение

В данной главе рассмотрен вопрос использования аппарата операционной семантики для создания инструментов интерпретации и отладки поведенческих визуальных языков. Выбран подкласс языков, семантика которых обычно основывается на понятии токена исполнения и которые формализуется с помощью сетей Петри (например, диаграммы деятельности UML 2 или язык блок-схем), для них реализован инструментарий, позволяющий интерпретировать формально заданную семантику языка, тем самым выполняя определённые полезные действия при обходе диаграммы (например, исполнение кода на некотором языке или генерацию текста). В главе описываются основные компоненты созданного решения и обсуждается их использование разработчиком и пользователями создаваемого языка.

Глава 5. Платформа QReal

5.1. Введение

В данной главе описывается архитектура предлагаемой в работе DSM-платформы QReal, позволяющая гибко настраивать компоненты платформы под конкретные решения. Рассматривается состав платформы, функциональность, предлагаемая разработчикам: графический интерфейс пользователя, хранилище моделей, генераторы кода, отладчики и интерпретаторы моделей, механизмы рефакторингов, проверки ограничений и другое.

5.2. Технология QReal

Типичный процесс работы с платформой QReal выглядит следующим образом. Используя набор инструментов платформы QReal, разработчик специализированного графического языка создает ряд формальных описаний этого языка — задает его метамодель (перечисляет набор сущностей языка, возможные связи между ними и их свойства), определяет внешний вид элементов, описывает ограничения, налагаемые на них или их группы, задает правила преобразования моделей для получения исходного кода, описывает семантику сущностей языка и т.п. (про методику разработки визуальных языков и инструменты поддержки процесса метамоделирования в QReal см. диссертационную работу Ю. В. Литвинова). По завершении формальных описаний разработчик языка из них посредством инструментов QReal автоматически получает среду визуальной разработки, имеющую в своей основе данный язык. Так как созданное DSM-решение использует для своей работы инфраструктуру и компоненты платформы QReal, то весь базовый функционал платформы автоматически доступен и для создаваемых на ее основе решений. Это позволяет разработчику языка сосредоточиться лишь на

функционале, являющимся особенным для создаваемого языка. Однако, если графический редактор подразумевает какую-то функциональность, которая отсутствует в базовой платформе, ее придется реализовывать отдельно программированием на C++. Также в текущей версии QReal “вручную” приходится реализовывать и сложные генераторы исходного кода, требующие серьезного анализа создаваемых моделей (стоит отметить, что платформа предоставляет для этого довольно удобные возможности в виде соответствующих программных интерфейсов и уже готовых примеров). Более простые генераторы (например, шаблонные генераторы, использующие уже написанные ранее шаблоны конечных программ) в платформе QReal возможно создавать автоматизированно на основе моделей, задающих операционную семантику языка. К тому же, в большинстве случаев по визуальным моделям происходит генерация не полноценных приложений, а код, активно использующий API уже существующих библиотек, в которых уже реализована значительная часть функциональности, полезной для данной предметной области. Как правило, это приводит к уменьшению размера генерируемого кода, а значит, и к сложности генератора кода. Такие генераторы также могут быть эффективно созданы автоматизированно по модели описания операционной семантики языка.

Рассмотрим подробнее архитектуру платформы QReal и то, как она взаимодействует с создаваемыми на ее основе DSM-решениями.

5.3. Общая архитектура платформы QReal

Технология QReal изначально задумывалась как развитие технологии REAL [52], основывающееся на использовании более современной версии языка UML — 2.0. При этом на разрабатываемые средства накладывались требования многоплатформенности (возможность работы на наиболее популярных операционных системах MS Windows, Linux и Mac OS X), поддержка

многопользовательской разработки, возможность удаленного доступа к репозиторию системы и другая актуальная для современных сред визуальной разработки ПО функциональность. Однако, быстро стало очевидно, что создание большого числа редакторов диаграмм вручную является довольно утомительным занятием, к тому же получаемая система оказывается плохо масштабируемой: создание дополнительного редактора, являющегося типовым для данного CASE-средства, чаще всего будет осуществляться методом копирования-вставки с дальнейшими доработками полученного после копирования кода, что влечет как к размножению уже существующих ошибок, так и к появлению дополнительных, например, связанных с неполнотой вносимых правок. Возникла необходимость в средствах быстрого создания визуальных языков и инструментальной поддержки для них [30].

Для автоматизации процесса создания таких редакторов метамодели соответствующих языков должны быть описаны формально с помощью некоторого языка[7], текстового или графического (или даже набора языков). Далее эти описания могут быть использованы следующим образом:

- в рамках генеративного подхода [4] по описанию метамодели генерируется программный код, который тем или иным образом дополняет код самого DSM-решения, привнося в него особенности данного языка;
- в рамках интерпретативного подхода в состав DSM-решения входит механизм, который по запросу читает нужные данные прямо из описания метамодели и обрабатывает их нужным образом (например, считывает количество и тип свойств элементов и отображает их в редакторе свойств). То есть, во время работы с редактором происходит интерпретация метамодели языка.

Исторически в QReal первым был реализован генеративный подход, поскольку он более прост в реализации и, как правило, откомпилированный код работает быстрее, чем некий сложный механизм, интерпретирующий внешние данные. Это

[7] В случае текстовых языков для этого чаще всего используются формы Бэкуса-Наура.

решение привело к тому, что архитектура как платформы, так и сред разработки, созданных на ее основе, становилась все более и более модульной (см. рис. 13) — конкретные DSM-решения на базе QReal получаются дополнением и параметризацией кода самой DSM-платформы. Абстрактная функциональность редакторов (например, такая, как способность двигать и масштабировать элементы на диаграммах, соединять их связями, хранить в репозитории значения их свойств и т.п.) максимально абстрагируется и формирует так называемое “ядро” системы, в то время как специфика каждого конкретного визуального языка оформляется в виде подключаемого модуля-плагина. Код каждого такого модуля генерируется автоматически по описанию метамодели языка, компилируется в плагин и не зависит от каких-либо других частей QReal. Плагины загружаются диспетчером модулей, который, в свою очередь, предоставляет интерфейс остальным частям QReal для доступа к информации, содержащейся в плагинах редакторов: именам и типам элементов и связей между ними, изображениям элементов, числу и типу их свойств, правилам соединения элементов связями и т.п.

Помимо плагинов редакторов в QReal существуют также плагины инструментов. В них выносится функциональность различных инструментальных средств CASE-пакета, которые могут быть полезны проектировщику — механизм версионирования моделей, генераторы кода, отладчики и интерпретаторы, механизм задания и проверки ограничений на создаваемые модели, механизм задания и проведения рефакторингов моделей и т.п. Это позволяет максимально переиспользовать инструменты между создаваемыми на базе QReal DSM-решениями — стоит добавить какую-нибудь функциональность в «ядро» системы или оформить ее в отдельный подключаемый модуль, и все DSM-решения при желании получат эту функциональность автоматически.

Рассмотрим подробнее архитектуру отдельных частей платформы. Основные библиотеки QReal и их взаимодействие показаны на рис. 14.

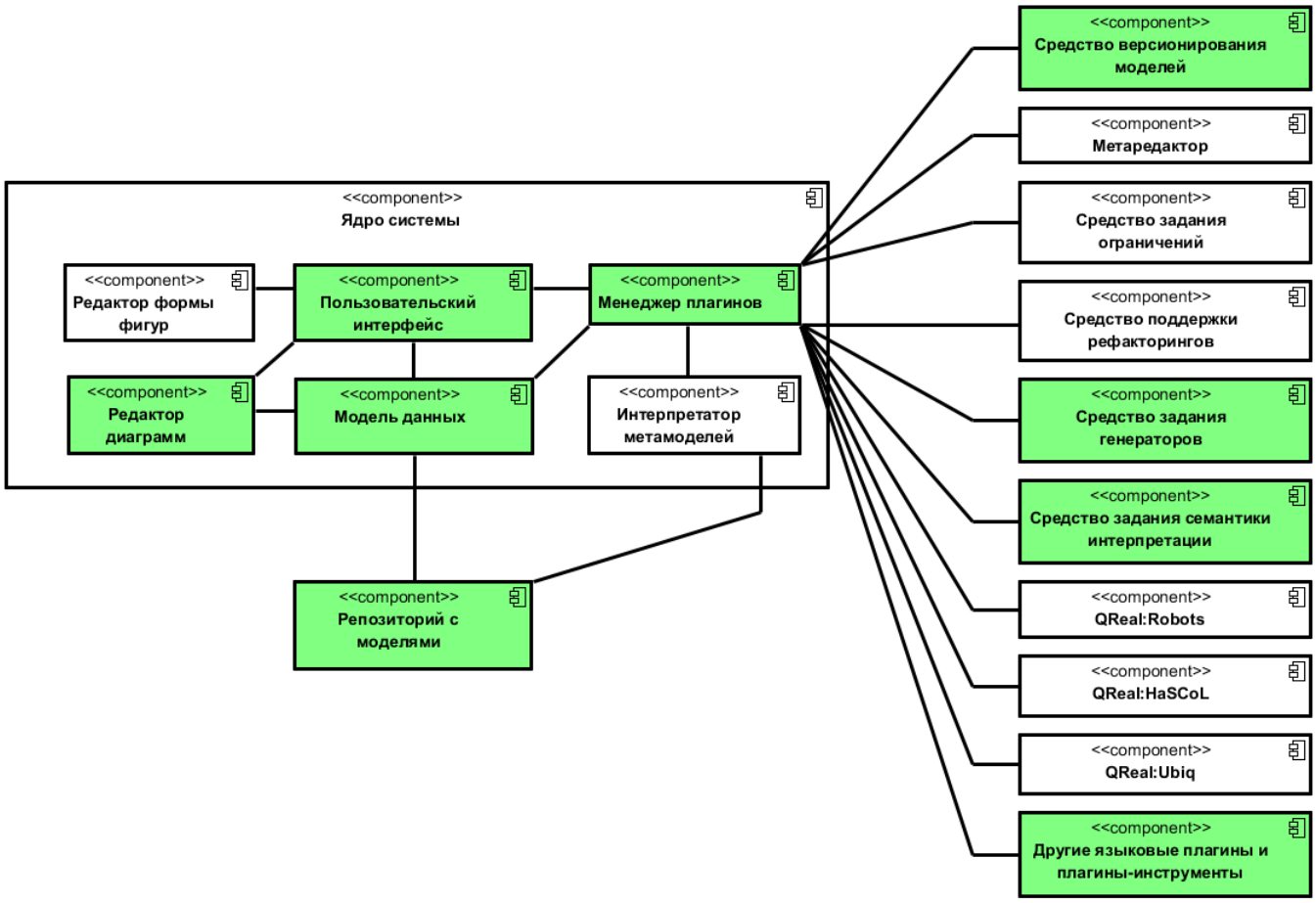

Рисунок 13. Общая архитектура DSM-решений на основе QReal[8]

Библиотека qrkernel представляет библиотеку классов, используемых во многих других модулях: класс для работы с конфигурационными файлами настроек, класс идентификаторов сущностей в моделях QReal, классы исключений и т.п. Как видно из рис. 14, этот модуль активно используется другими.

Назначение библиотеки qrutils сходно с qrkernel, однако если содержимое последней используется без исключения всеми другими компонентами, то в qrutlis находится код, который может понадобиться лишь некоторым другим компонентам. Например, в qrutils находятся абстрактные классы для кодогенераторов, базовая часть модуля трансформации графов, некоторые графические утилиты для работы с диаграммами, средства для вызовов из кода сторонних программ и другие.

[8] Цветом выделены компоненты, разработанные автором в рамках диссертационного исследования

Библиотека qrrepo реализует функциональность репозитория QReal. Репозиторий предоставляет остальным компонентам платформы интерфейс RepoApi, который позволяет осуществлять все необходимые операции над моделями — загрузку и сохранение проектов, создание и удаление элементов и связей между ними, получение и установка свойств элементов и связей и многое другое.

Компонента qrgui представляет собой основное приложение, реализующее графический интерфейс пользователя.

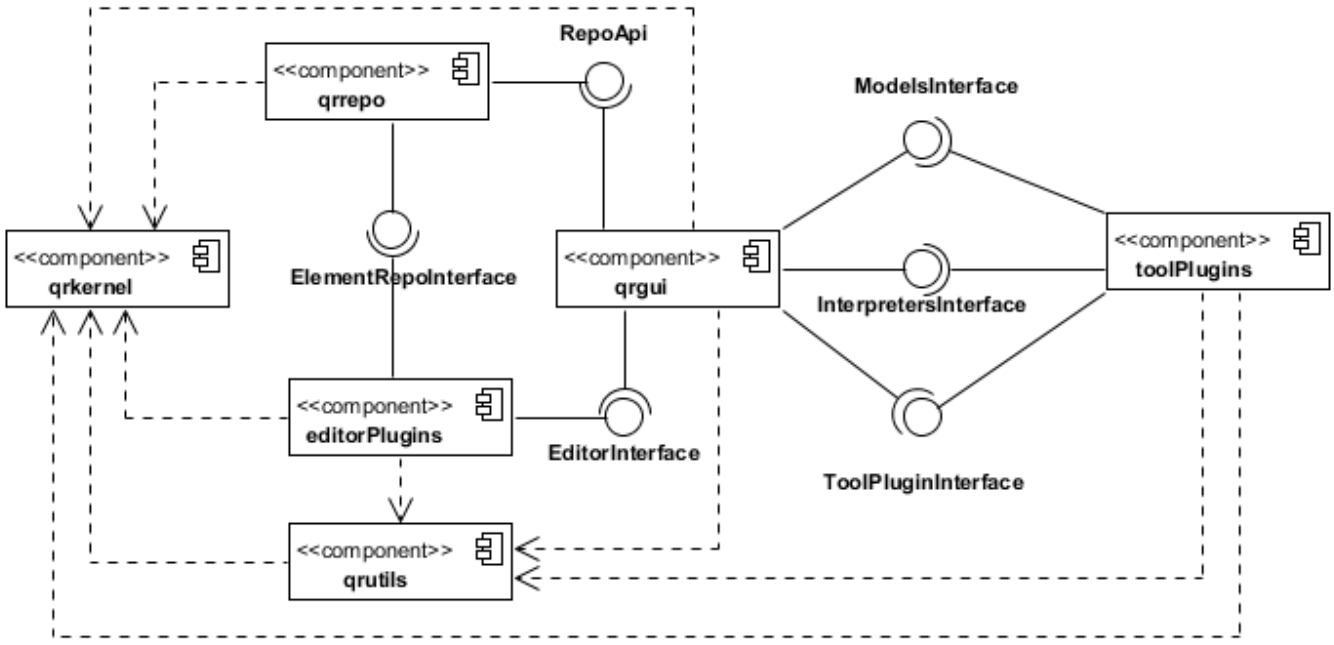

Рисунок 14. Основные модули QReal

Компонента editorPlugins представляют собой подключаемые модули визуальных редакторов. Через интерфейс EditorInterface каждый плагин предоставляет информацию о наборе диаграмм, содержащихся в данном модуле, о типе и числе элементов и связей в языке, о числе, типах и значениях по умолчанию свойств каждого типа элемента и связи, о способе группировки элементов языка в палитре QReal, о правилах соединения элементов связями и о правилах помещения одних элементов в другие как в контейнер, предоставляет фабрику создания элементов данного языка и некоторые другие средства, используемые «ядром»

QReal. Сами плагины редакторов используют общую функциональность платформы, вынесенную в модули qrkernel и qrutils, а также репозиторий через интерфейс ElementRepoInterface для получения значений свойств элементов.

Компонента toolPlugins представляет набор подключаемых модулей инструментов QReal. В подобные плагины выносится любая функциональность, которая напрямую не относится к ядру QReal, то есть потенциально может быть не нужна для определенных DSM-решений. Каждый такой плагин предоставляет интерфейс ToolPluginInterface, через который с ним можно совершать следующие действия: запросить список меню и элементов панелей для встраивания в главное окно QReal, запросить страницу настройки для встраивания в общее окно настроек среды, название главного окна и иконку приложения. С основной частью QReal плагины инструментов взаимодействуют через интерфейсы InterpretersInterface и ModelsInterface. Первый позволяет изменять состояние главного окна (выделять и подсвечивать элементы на диаграммах и некоторые другие), загружать и выгружать другие плагины. ModelsInterface представляет доступ к данным моделей — доступны операции создания новых элементов, получения свойств текущих, проверка принадлежности элемента контейнеру и т.п.

5.4. Состав DSM-решения на основе QReal

Получаемое типовое предметно-ориентированное решение на базе QReal включает в себя следующие инструменты.

5.4.1. Графический интерфейс пользователя

Графический интерфейс (graphical user interface, GUI) получаемых сред разработки предоставляет пользователям типичные инструменты для подобного рода инструментариев (внешний вид редактора блок-схем представлен на рис. 15): обозреватели модели, редактор свойств элементов, рабочую область редактора

диаграмм, палитру доступных для моделирования элементов, “миникарту”, являющуюся уменьшенной версией текущей диаграммы (бывает весьма полезной при работе с крупными диаграммами, занимающими более одного экрана), окно для отображения предупреждений и сообщений об ошибках, а также ряд набор меню и панелей главного окна для управления остальными инструментами.

При желании любые элементы графического интерфейса можно скрыть.

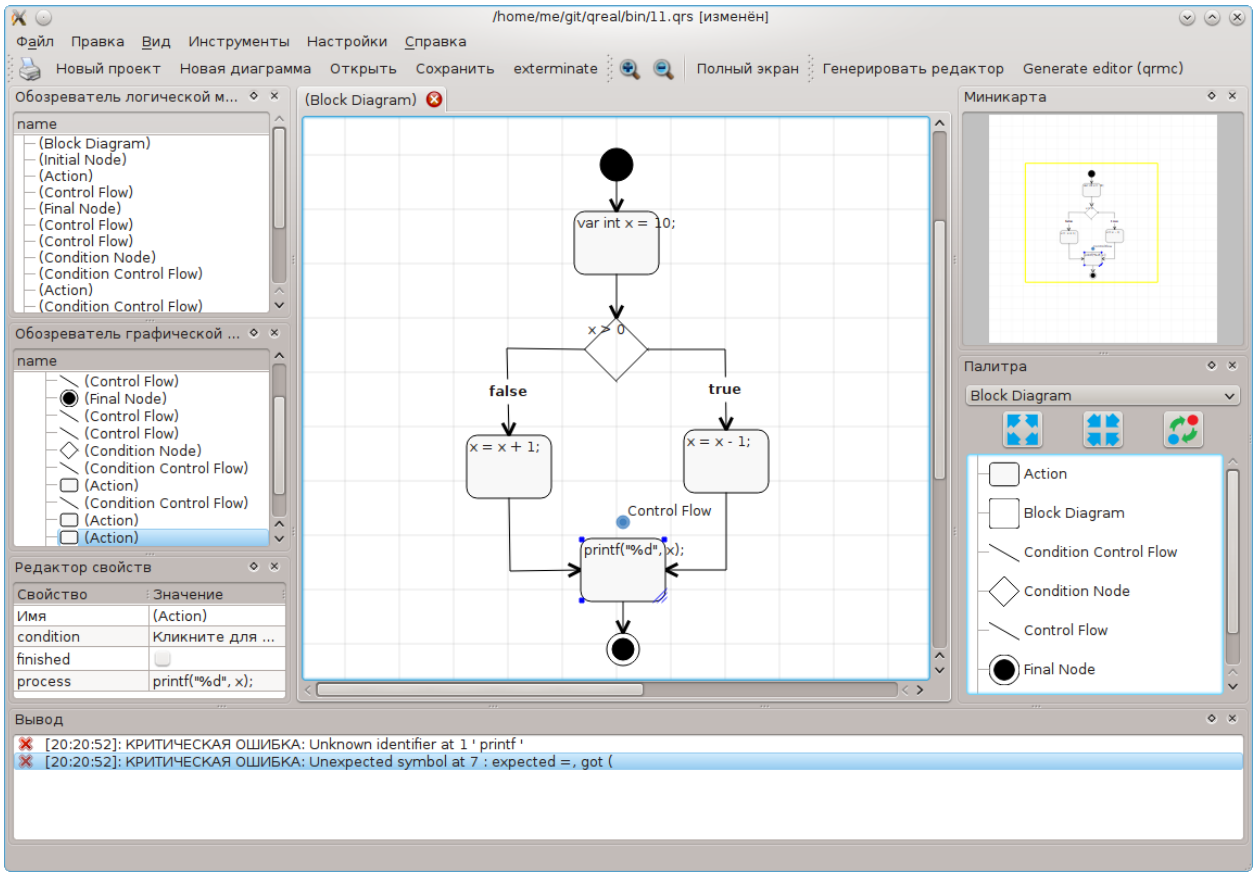

Рисунок 15. Графический интерфейс DSM-решения на основе QReal

5.4.2. Репозиторий

Репозиторий QReal представляет собой простую объектно-ориентированную базу данных, хранящую все создаваемые в QReal модели. Репозиторий представлен в виде отдельной динамически загружаемой библиотеки, реализованной на языке C++, в результате чего может быть использован сторонними программами (например, генераторами кода или анализаторами моделей) через специальный

программный интерфейс. Из других модулей платформы QReal репозиторий использует qrkernel и qrutils.

Доступ к данным репозитория осуществляется через интерфейс RepoApi, который соединяет в себе три других интерфейса: интерфейсы моделей GraphicalRepoApi и LogicalRepoApi и интерфейс служебных операций RepoControlInterface. Интерфейс RepoControlInterface содержит несколько операций загрузки и сохранения содержимого репозитория, а также методы, реализующие поиск среди содержимого репозитория. Интерфейсы моделей предоставляют полный доступ к данным: диаграммам, элементам и связям между ними. Доступны операции создания и удаления произвольных элементов и связей, запрос и изменение свойств, помещение элементов в другие как в контейнеры, проверка элемента с заданным идентификатором на существование и другие.

5.4.3. Средства для работы с логическими и графическими моделями

Логическая модель системы отражает её внутреннюю структуру, и это то, с чем работают генераторы и другие инструменты. Графическая модель системы — это набор диаграмм, отображающих её логическую модель, с ней работает пользователь. В самом простейшем случае между элементами логической и графической моделей устанавливается соотношение «один к одному», однако это удобно далеко не всегда. Некоторые элементы модели могут визуального представления не иметь вовсе (например, значения перечислимых типов или специальные служебные вещи типа настроек генерации), некоторые, наоборот, имеют только визуальное представление и на логику системы никак не влияют (например, декоративная линия или прямоугольник на диаграмме), некоторые могут иметь несколько представлений (например, один и тот же класс UML может присутствовать на нескольких разных диаграммах, причём выглядеть по-разному).

Поддержка разделения логических и графических моделей реализована в QReal на нескольких уровнях.

- На уровне графического интерфейса присутствуют отдельные обозреватели логической и графической моделей. С их помощью можно создавать, перемещать и удалять элементы каждого типа моделей, помещать логические элементы на диаграммы (создавать графические элементы), число «копий» одного логического элемента на диаграмме не ограничено. В редакторе свойств для элемента на диаграмме отображаются как его графические свойства (позиция, размеры), так и свойства соответствующего логического элемента (свойства элемента, описанные в метамодели языка). Например, изменение имени приведет к изменению имен всех элементов на диаграммах, соответствующих данному логическому элементу.
- На уровне репозитория существует два различных интерфейса для работы с логической и графической моделями. Операции данных интерфейсов различаются: через API графической модели в репозитории можно оперировать данными о диаграммах (позиция и размеры элементов, отношение вложенности контейнеров и т.п.), через API логической модели работать с логическими свойствами элемента (например, управлять трассировкой — невизуальным типом связи между элементами, используемым в некоторых редакторах). Для связи элементов моделей для каждого элемента графической модели в репозитории возможно получить идентификатор соответствующего ему элемента логической модели (если он существует).

5.4.4. Средства обеспечения многопользовательской работы

Для обеспечения коллективной работы над проектом в QReal нескольких проектировщиков была добавлена возможность версионирования создаваемых моделей с помощью систем контроля версий (version control system, VCS): доступны операции сохранения моделей на удаленном сервере и загрузки на рабочее место (операции check-in и check-out систем контроля версий соответственно), откат к предыдущим версиям, просмотр изменений между версиями.

Рассмотрим, как в QReal устроена работа с файлами сохранений. Файлы сохранений имеют расширения .qrs (QReal Save file) и получаются следующим образом: содержимое репозитория сериализуется на диск в виде иерархии директорий, разделяющих элементы логической и графической моделей (далее элементы каталогизируются по типу элемента для логической модели, для графической — вначале по принадлежности диаграмме, а затем уже по типу элемента). Сам элемент сериализуется в виде XML-файла, содержащего всю информацию о нем: идентификатор элемента, его имя, значения всех свойств и т.п. Для получения единого файла сохранения данная иерархия каталогов архивируется. При загрузке файла сохранения на рабочее место репозиторием производятся обратные операции: разархивирование каталогов и рекурсивное чтение их содержимого.

Благодаря подобному внутреннему устройству файлов сохранения поддержка версионирования моделей выглядит довольно несложной задачей, однако встает следующая техническая проблема. Очевидно, что функциональность версионирования должна быть потенциально отключаемой, т.е. быть реализована в виде плагина инструментов. Однако, описанные выше операции над файлом сохранения производит объект класса Serializer, который является внутренним объектом плагина репозитория. Расширять интерфейс репозитория операциями системы контроля версий видится неразумным, так как добавляет этому объекту еще одну функцию, слабо связанную с основной — предоставлением доступа к данным. К тому же, существенным требованием была максимальная гибкость и расширяемость — добавление поддержки новой системы контроля версий должно требовать минимума изменений существующего кода. Использование еще одного экземпляра объекта Serializer в плагине версионирования также было затруднено: для этого потребовалось бы раскрывать детали реализации репозитория.

Очевидным решением было бы версионировать уже заархивированные файлы

сохранения, но в таком случае будет осуществляться версионирование бинарных файлов, и вся информация о семантике изменений, осуществляемых в моделях, теряется.

В результате было реализовано следующее решение (см. рис. 16). Для того, чтобы дать произвольным компонентам QReal возможность распаковать файл сохранения без нарушения инкапсуляции, был создан интерфейс WorkingCopyManagementInterface, содержащий две операции: prepareWorkingCopy(), распаковывающую файл сохранения в каталог с рабочей копией, и processWorkingCopy(), соответственно запаковывающую директорию с рабочей копией в файл сохранения. Интерфейс репозитория RepoControlApi, содержащий служебные операции над репозиторием, был унаследован от WorkingCopyManagementInterface.

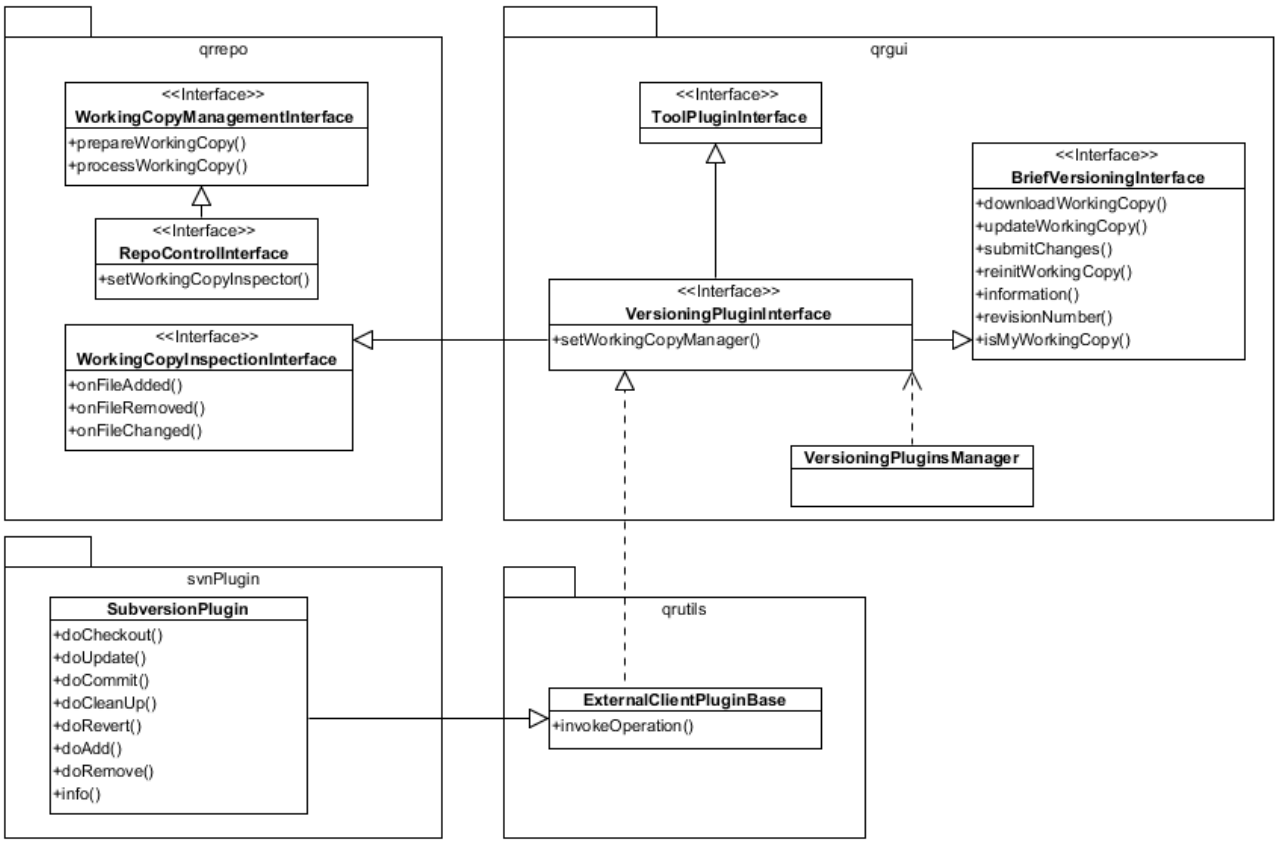

Рисунок 16. Встраивание компонент версионирования в архитектуру QReal

Следующая задача — обеспечение контроля за изменениями файлов в рабочей копии. Например, если на диаграмме был создан новый элемент, в файле сохранения появится соответствующий файл, который нужно добавить в рабочую копию VCS соответствующей командой. Для обработки изменений в рабочей копии был создан интерфейс WorkingCopyInspectionInterface, а в RepoControlApi был добавлен метод setWorkingCopyInspector(), принимающий подобный объект-инспектор и передающий его в сериализатор. Сериализатор, выполняя сохранение, узнает, какие файлы добавились, удалились или изменились и вызывает соответствующие методы объекта-инспектора. Это единственный способ для других компонент узнать об изменениях в рабочей копии.

Как показано на рис. 16, интерфейс плагина инструмента версионирования VersioningPluginInterface унаследован от трех других интерфейсов. Таким образом, каждый плагин, реализующий поддержку какой-либо VCS, должно состоять из трех частей.

- Функционал обычного плагина инструмента.
- Реализация интерфейса инспектора WorkingCopyInspectionInterface. Например, в случае поддержки subversion метод onFileAdded() будет вызывать команду svn add, a onFileRemoved() — команду svn remove.
- Реализация интерфейса BriefVersioningInterface, содержащего операции абстрактной VCS: получение рабочей копии с удаленного сервера, синхронизация рабочей копии с сервером, откат к выбранной версии и т.п. Этот интерфейс, по сути, и есть API плагинов версионирования, используемый другими компонентами в рамках графического интерфейса пользователя.

Для удобства выполнения внешних команд над клиентами систем контроля версий в модуле qrutils был реализован класс ExternalClientPluginBase, инкапсулирующий в себе механизм общения с внешними процессами, выдавая наружу только методы по синхронному/асинхронному запуску этих процессов.

На основе модуля версионирования был реализован механизм визуального отображения изменений выбранной диаграммы между рабочей копией и определенной версией на сервере. Сравнение осуществляется следующим образом. После выбора интересующей проектировщика версии она загружается с сервера во временную директорию, создается новый экземпляр репозитория, в которой и загружается полученная с сервера версия проекта. После этого имеется две загруженные в память модели, которые сравниваются поэлементно: через методы репозитория можно запросить информацию обо всех элементах, их свойствах и т.п. При этом стадия сопоставления является тривиальной, поскольку каждому элементу в репозитории присвоен уникальный идентификатор, не изменяющийся на протяжении всей жизни объекта.

Когда построение модели разницы завершается, отображается окно, наглядно демонстрирующее изменения между диаграммами (см. рис. 17). Левая часть окна отображает диаграмму, хранящуюся на сервере, правая — в рабочей копии. Элементы на диаграммах подсвечены в соответствии с их состоянием. Состояние может быть одним из следующих: “добавлено”, “удалено”, “изменено” или “без изменений”. На рисунке удаленные и добавленные элементы подсвечены зеленым цветом, модифицированные — оранжевым (цвета, соответствующие состояниям, могут быть изменены в диалоговом окне настроек QReal).

Изменение значений свойств, не отображаемых визуально на диаграмме, можно отследить на вкладках в нижней части окна.

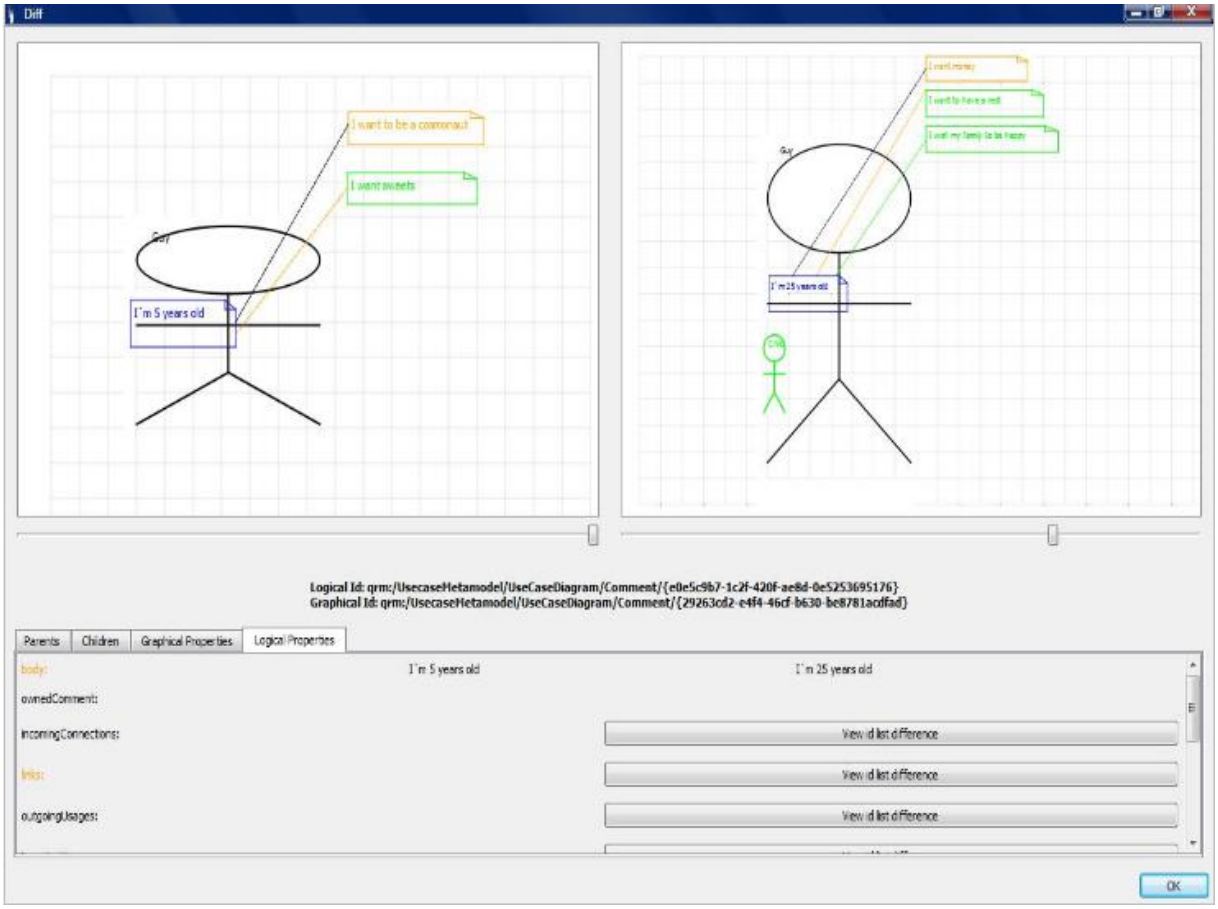

Рисунок 87. Окно визуального сравнения версий диаграммы

5.4.5. Отладчики и интерпретаторы моделей

Для поведенческих языков, описывающих систему в динамике (например, диаграммы деятельности UML 2 или язык блок-схем), очень полезными могут быть отладчики и интерпретаторы, работающие на уровне моделей. Подобные инструменты позволяют разработчику наглядно проследить ход работы заданного им алгоритма и либо убедиться в его корректности, либо внести нужные изменения. Семантика таких языков обычно основывается на понятии токена исполнения и формализуется с помощью сетей Петри, причём класс поведенческих языков достаточно широк, чтобы стоило потратить усилия на формализацию их семантики (например, упоминавшиеся выше диаграммы деятельности UML 2 и блок-схемы имеют именно такую семантику).

В QReal в данном направлении реализовано несколько инструментов.

- Интерпретатор моделей, позволяющий пошагово «исполнять» модель, подсвечивая текущий блок и отображая текущие состояния переменных, если это применимо для данного языка.
- Отладчик сгенерированного кода с привязками к диаграммам.

Описанные инструменты были реализованы «вручную» в виде плагинов инструментов QReal, однако в дальнейшем создание интерпретаторов было автоматизировано путем описания операционной семантики разрабатываемого языка (как описано в главе 4).

Создание отладчиков моделей с привязкой к отладке исходного кода все ещё остается неавтоматизированной задачей ввиду большого числа целевых языков генерации и существующих отладчиков/интерпретаторов для них.

5.4.6. Генераторы кода

Визуальные модели могут использоваться практически на всех этапах жизненного цикла: для фиксирования знаний при сборе и анализе требований, в качестве наглядного материала при общении с пользователями и заказчиками, при декомпозиции задач разработчиками и т.п. Основное требование к таким диаграммам — наглядность и понятность. Однако, многие желают пойти дальше и использовать знания о предметной области и разрабатываемой системе, отраженные в моделях, для того, чтобы автоматически (или хотя бы автоматизированно) получать по ним программный или тестирующий код, скрипты сборки или установки, документацию и т.д. Построение подобных артефактов разработки по моделям — задача генераторов.

С архитектурной точки зрения генераторы в QReal могут быть внутренние и внешние. Внутренние генераторы реализованы в виде плагина инструментов, динамически подгружаются в QReal и могут через стандартные интерфейсы воздействовать на графический интерфейс пользователя (например, отображать

сгенерированный код в отдельной вкладке, отображать сообщения об ошибках средствами QReal и т.п.). Внешние генераторы реализуются и запускаются как отдельные приложения и никак не связаны с платформой QReal. Возможность создавать такие генераторы может быть полезна, если разработчик по каким-то причинам выбирает язык программирования, отличный от C++.

Независимо от типа генератора получение данных происходит через интерфейсы логической (и, реже, графической) моделей репозитория. В случае генератора, реализованного на отличном от C++ языке программирования, взаимодействие с библиотекой репозитория может происходить, например, посредством инструментария ZeroC ICE [2], обеспечивающего взаимодействие кода на различных языках программирования.

В области модельно-ориентированной разработки с генераторами кода тесно связан вопрос о возможности циклической разработки (round-trip engineering [55]). Основная проблема, решаемая данным подходом, — синхронизация изменений, внесенных в сгенерированный код, с исходными моделями. Эта задача далеко не тривиальна, и разработчики практически каждого инструмента моделирования вынуждены как-то решать ее для себя [54]. Однако, как было описано в главе 1, предметно-ориентированное моделирование постулирует отказ от «ручного» изменения сгенерированного кода — проектировщик ведет работу только с моделями, по которым автоматически генерируется код, не требующий последующих изменений. При необходимости внести в получаемый исходный код изменения необходимо либо поправить исходную модель и произвести перегенерацию кода (если это логическое изменение), либо исправить генератор кода (если требуемое изменение — это исправление ошибки), либо если в языке не хватает выразительных средств для описания требуемого изменения — поднять вопрос о расширении языка и инструментов, поддерживающих его.

Для платформы QReal существует несколько механизмов для упрощения

процесса создания генераторов кода. Один из них основан на специализированном текстовом языке [38, 39], позволяющем описывать логику генерации. Это позволяет ускорить процесс создания генератора, однако всё еще требует от пользователя серьёзных навыков программирования. Другой способ создания генераторов подразумевает задание исполнимой семантики языка (см. главу 4). Генератор в этом случае по сути является интерпретатором, обходящим (возможно неоднократно) диаграммы, на каждом элементе осуществляя запись исходного кода в выходной файл.

5.4.7. Механизм рефакторингов моделей

Рефакторинг определяют как изменение внутренней структуры системы для того, чтобы сделать ее проще для понимания и внесения изменений. При этом внешнее поведение такой системы не должно меняться. При переносе уровня абстракции с кода на визуальные модели имеет смысл перенести на уровень моделей и понятие рефакторинга. Рефакторинг на уровне моделей является довольно молодой областью знания, многие вопросы в ней остаются открытыми для дальнейших исследований. В настоящее время очень немногие инструментарии способны осуществлять интегрированную поддержку рефакторинга моделей (см. главу 2). Кроме того, класс моделей, для которых реализована поддержка рефакторинга, весьма ограничен: чаще всего исследования ведутся применительно лишь к диаграммам классов UML.

В настоящее время вопрос организации рефакторингов моделей в QReal все еще исследуется, однако уже сейчас реализована инфраструктура и несколько примеров автоматического преобразования моделей: групповое изменение имени элементов, инвертирование выбранных связей на диаграмме, выделение набора элементов в подпрограмму и некоторые другие. Несложно заметить, что практически все эти рефакторинги применимы для произвольных языков.

С точки зрения пользователя процесс применения рефакторинга выглядит

следующим образом: с помощью соответствующего пункта меню необходимо открыть окно рефакторингов в QReal и выбрать интересующее правило (см. рис. 18). Правило представляется в виде левой и правой части, между которыми находится стрелка, символизирующая преобразование модели. Левая часть задает шаблон поиска элементов в моделях, а правая часть описывает то, во что преобразуется элемент (или группа элементов для более сложных правил), подходящий под шаблон левой части.

Нажатие кнопки “Найти” приводит к поиску элементов, подходящих под шаблон левой части правила. Если было найдено хотя бы одно соответствие, становится активной кнопка “Применить”, которая осуществляет выбранное преобразование модели.

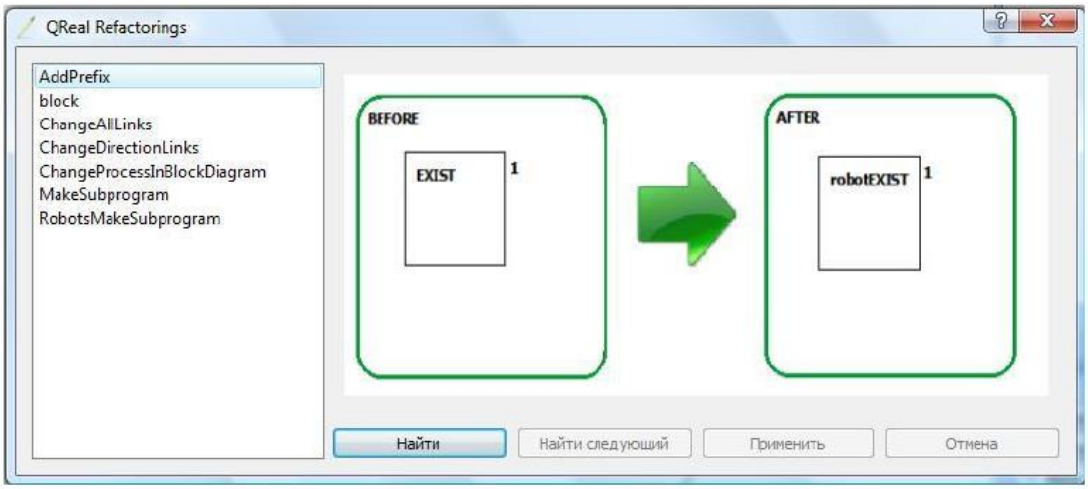

Рисунок 18. Окно применения рефакторингов в QReal

С архитектурной точки зрения модуль применения рефакторингов представлен отдельным плагином инструментов QReal. Правила являются отдельными диаграммами, хранящимися в репозитории. Для осуществления поиска по моделям используется базовый механизм трансформации графов QReal, вынесенный в модуль qrutls (данный механизм используются и другими модулями QReal, например, при выполнении правил исполнимой семантики языка).

5.4.8. Механизм проверки правил ограничений

Несмотря на то, что визуальные программы в большинстве своем нагляднее и проще для понимания, чем текстовые, модели также могут содержать в себе ошибки. Часть ошибок могут быть синтаксическими (например, из элемента “Начало” диаграмм деятельности UML должна выходить только один связь, и не должно входить ни одной), так и семантическими (поле “Возраст” элемента “Человек” не должно быть отрицательным). Данные ошибки естественным образом проявятся либо на фазе кодогенерации, либо уже во время работы целевого приложения, и чем раньше проектировщик узнает о некорректности создаваемой им модели, тем проще будет исправить имеющиеся ошибки.

В QReal был реализован механизм выполнения правил ограничений во время работы с моделями. Проверка правил происходит при выполнении какого-либо из следующих событий:

- изменение любого свойства элемента (логической или графической модели);
- при помещении элемента в контейнер или извлечении элемента из контейнера;
- при создании или удалении элементов;
- при присоединении и отсоединении связей.

У каждого правила задан уровень ошибки:

- предупреждение — элемент, для которого не выполняется заданное правило, будет подсвечен на диаграмме красным цветом;
- критичная ошибка — аналогично предупреждению, но при этом еще выдается текстовое сообщение с текстом ошибки;
- настраиваемый режим, при котором уровень критичности будет задаваться пользователем в настройках QReal. Например, это может быть полезно в режиме отладки, когда на время хочется отключить текстовые сообщения об ошибках.

На рис. 19 и 20 показано несколько примеров выполнения правил ограничения

для языка программирования роботов (см. главу 6). Так, на рис. 19 не выполняется правило о том, что сразу после блока “Включить моторы” не должен идти блок “Выключить моторы” (на диаграмме эти блоки идут последовательно). Элемент “Выключить моторы” подсвечен красным цветом.

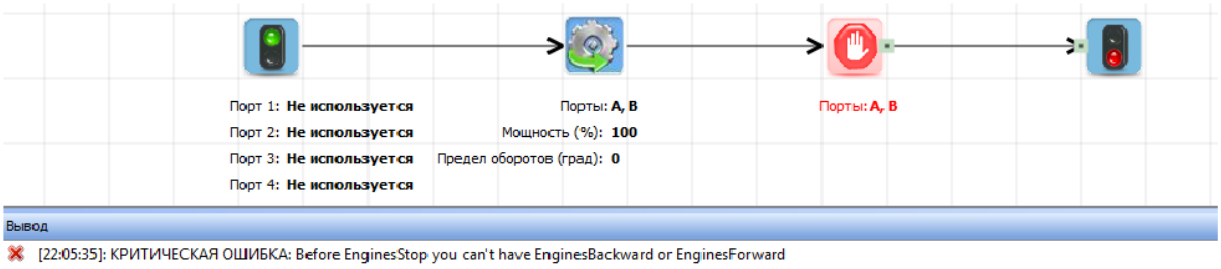

Рисунок 19. Нарушение правила ограничения

На рис. 20 нарушено правило о том, что на диаграмме не может быть более одного блока “Начало”. В данном случае элементом, для которого не выполнено ограничение, является сама диаграмма, поэтому никаких блоков не подсвечивается.

При нажатии на текст ошибки происходит выделение элемента, которому она соответствует (диаграмма, на которой располагается этот элемент, при необходимости будет открыта).

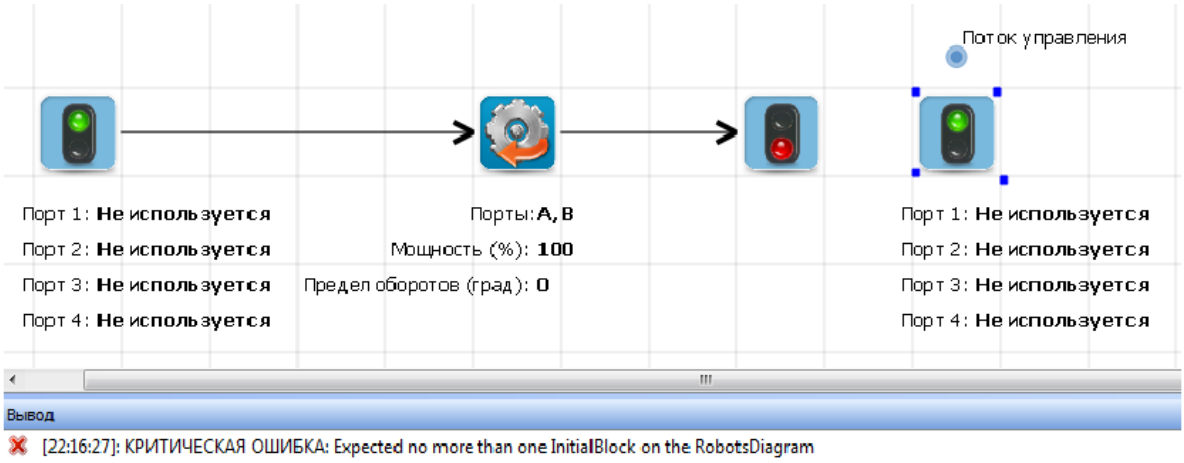

Рисунок 20. Нарушение правила ограничения

5.5. Заключение

В данной главе рассмотрена архитектура разработанной автором предметно-ориентированной платформы: описаны её принципы, основные типы подключаемых модулей (плагины редакторов, плагины инструментов) и их взаимодействие, способы расширения платформы. Описано назначение и правила использования основных компонент платформы, которые могут быть использованы для формирования конечных предметно-ориентированных решений. Как видно из данной главы, разработанное решение удовлетворяет требованиям к современной DSM-платформе, сформулированным в разделе 2.5.

Глава 6. Апробация

6.1. Введение

Описанная в данной работе DSM-платформа была использована для создания ряда как отдельных визуальных языков и редакторов для них, так и полноценных DSM-решений, включающих в себя интерпретаторы моделей, генераторы исходного кода и прочие инструменты. В данной главе рассматриваются некоторые примеры таких решений.

6.2. Среда разработки QReal:Robots

Среда QReal:Robots [49, 50] предназначена для программирования роботов Lego Mindstorms NXT 2.0. В настоящее время робототехнические конструкторы активно используются в школах, в частности в рамках уроков информатики и кружков для обучения детей программированию и основам кибернетики. Одним из самых популярных таких конструкторов в российских школах является Lego Mindstorms NXT 2.0. Программирование этих роботов старшеклассниками происходит на языке Си, в младшей и средней школе используются более наглядные визуальные средства программирования — NXT-G [16] для совсем простых задач и Robolab [20] для более сложных. При этом у учителей, использующих эти средства на практике, накопился ряд замечаний к ним: отсутствие наглядных механизмов отладки создаваемых программ, неполная русификация, высокая стоимость, и др., а также больше число замечаний к графическому интерфейсу сред и представлению используемых в них блоков. В результате было принято решение разработать отечественную среду программирования робототехнических конструкторов Lego Mindstorms NXT 2.0 на базе платформы QReal.

Внешний вид QReal:Robots представлен на рис. 21 и характерен для

большинства создаваемых на базе QReal сред (как описано в разделе 5.4). Визуальный язык составляют 22 блока, разбитые на четыре логические группы:

- алгоритмические блоки (например, блоки “Условие” или “Цикл”);
- блоки инициализации программы (например, блоки “Начало” или “Сбросить показания количества оборотов моторов”);
- блоки действий (например, блоки “Моторы вперед” или “Гудок”);
- блоки ожиданий (например, блоки “Ждать срабатывания сенсора касания” или “Таймер”).

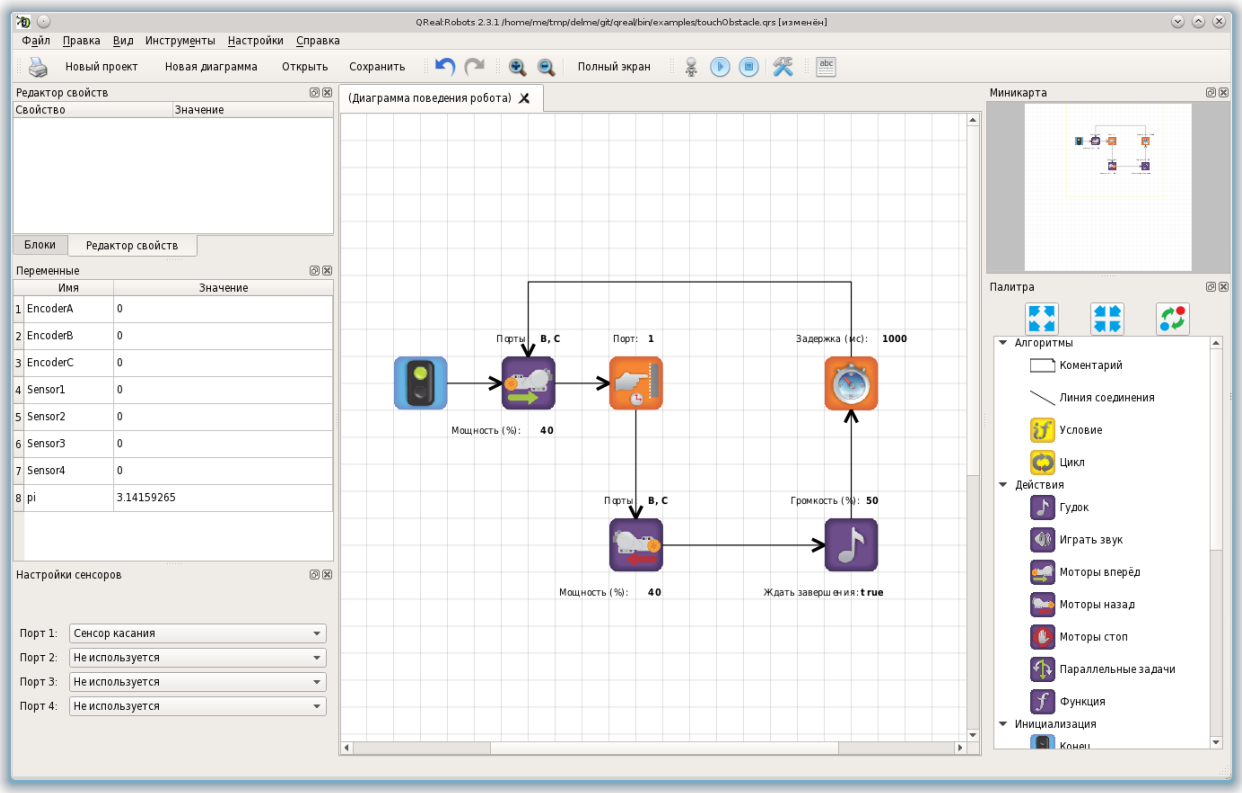

Рисунок 91. Внешний вид среды QReal:Robots

Отметим, что при возникновении требования разделения блоков в палитре на логические группы используемый для описания редакторов метаязык был расширен, и была добавлена поддержка этой функциональности на уровне DSM-платформы. Таким образом, появилась возможность задавать группы элементов в палитре не только в редакторе QReal:Robots, но и в других, созданных на базе платформы.

QReal:Robots расширяет стандартный графический интерфейс платформы QReal некоторыми дополнительными виджетами, характерными для предметной области роботов: например, панелью настройки сенсоров, используемой для конфигурирования сенсоров, подключаемых к роботу. При этом ряд виджетов, к моменту создания QReal:Robots уже существовавших в рамках других сред, созданных на базе QReal, был обобщен и перенесен на уровень DSM-платформы, что позволило использовать их в других средах, в частности, в QReal:Robots. Например, таким виджетом оказалась панель отображения переменных, показывающая состояние всех используемых в программе переменных и констант, как служебных (например, значения показателей количества оборотов моторов), так и созданных пользователем.

Существует несколько вариантов выполнения создаваемых в QReal:Robots программ.

- Автоматическая генерация по диаграмме кода на языке Си, компиляция его в бинарный код и запись его в память робота (робот должен быть подключен по USB или Bluetooth к компьютеру, на котором работает QReal:Robots). В этом случае программа будет выполняться на роботе автономно.
- Пошаговая интерпретация диаграммы с посылкой роботу команд, соответствующих выполняемому блоку диаграммы. В этом случае робот также должен быть подключен к компьютеру по USB или Bluetooth, однако, в отличие от предыдущего случая, робот должен оставаться подключенным на все время выполнения программы.
- Пошаговая интерпретация диаграммы с использованием двухмерной модели робота в качестве исполнителя (см. рис. 22). С помощью специального окна возможно конфигурирование как самой модели робота (например, определение набора и расположения сенсоров), так и окружающего мира (создание препятствий, линий на полу и т.п.).

Функциональность, связанная с двухмерной моделью робота, была создана программированием вручную в виде подключаемого к QReal:Robots модуля, поскольку является слишком специфичной для данной предметной области, чтобы быть частью DSM-платформы.

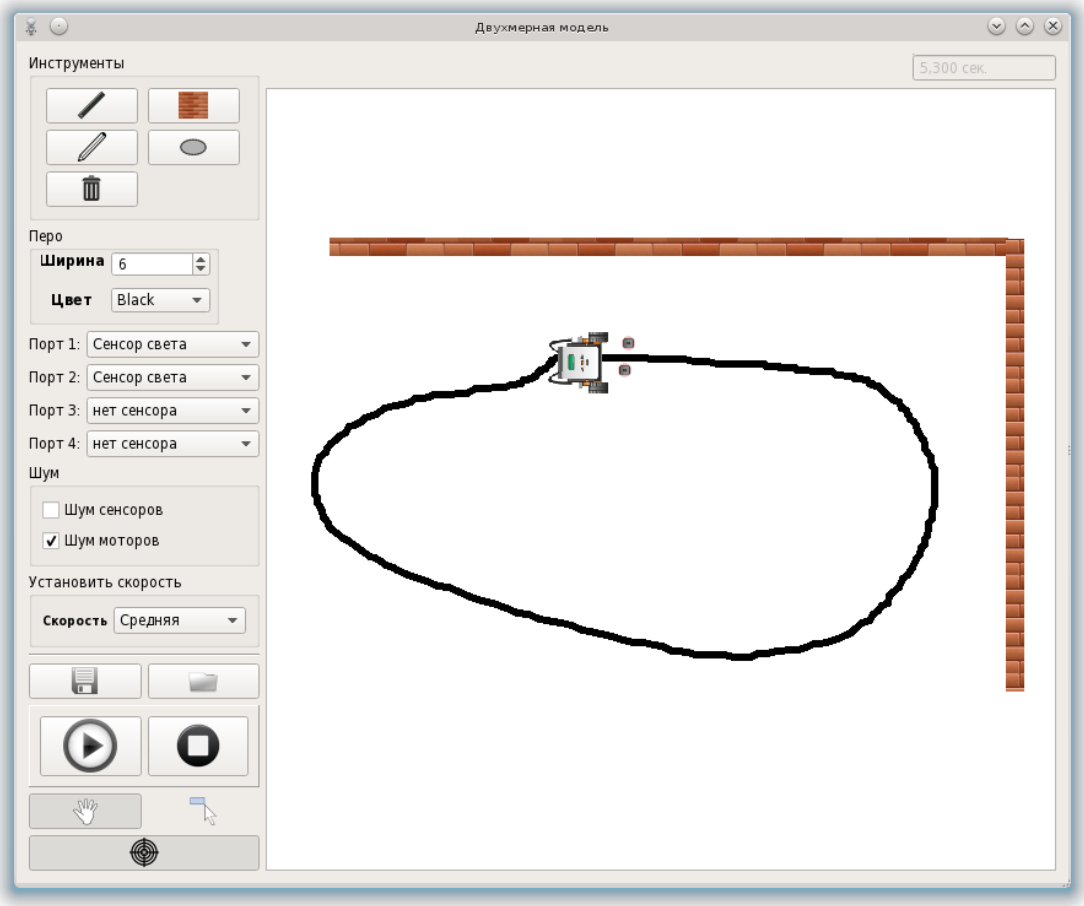

Рисунок 10. Окно двухмерной модели робота в QReal:Robots

При старте проекта QReal:Robots в платформе QReal еще не было средств автоматизированного создания визуальных интерпретаторов и генераторов исходного кода по диаграммам, поэтому описанные выше интерпретаторы диаграмм и генератор кода программы на Си были созданы кодированием вручную на C++/Qt. После появления средств задания операционной семантики языков генератор был описан и с помощью этого инструмента.

6.3. Среда разработки, основанная на языке блок-схем

Блок-схемы являются примером широко известного визуального языка общего назначения с небольшим количеством элементов и конкретной семантикой этих элементов. Это позволило использовать редактор блок-схем в качестве средства апробации новых инструментальных средств, разработанных для платформы QReal. Например, изначально работа над созданием средств задания исполнимой семантики поведенческих языков велась применимо именно к редактору блок-схем — изначально визуальный интерпретатор был создан для этого редактора вручную, затем велась работа по обобщению подобной функциональности на другие языки, основанные на задании потока управления.

Внешний вид среды представлен на рис. 23. Используемый визуальный язык является подмножеством языка, определяемого ГОСТ 19.701-90, и состоит из пяти типов элементов (блоки “Начало”, “Конец”, “Действие”, “Условие” и “Диаграмма”) и двух связей (одна из которых используется для задания вариантов условных переходов, а другая — для связи остальных блоков). В качестве текста, записываемого внутри блоков, используются конструкции языка Си.

Помимо редактора диаграмм в состав среды входят следующие инструментальные средства.

- Визуальный интерпретатор диаграмм, позволяющий выполнять программу пошагово блок за блоком, подсвечивая текущий. При этом происходит анализ кода, содержащегося в блоках — объявленные переменные добавляются в список текущих переменных (и отображаются в специальном окне QReal), при выполнении определённых действий над переменными их значения нужным образом изменяются. При интерпретации условных блоков происходит вычисление выражения, содержащегося в тексте блока, и осуществляется переход по одной из веток алгоритма. В случае некорректных выражений (как на языке Си внутри блоков, так и некорректных диаграмм в терминах блок-

схем) выдается соответствующее предупреждение с указанием на проблемный блок.

- Генератор исходного кода, дающий возможность получить по диаграмме программу на языке Си. При генерации проверяется синтаксическая корректность диаграммы (неструктурные диаграммы не поддерживаются, в этом случае выдается предупреждение). Корректность кода внутри блоков не проверяется, текст просто копируется без изменений в целевой выходной файл.
- Визуальный отладчик диаграмм осуществляет генерацию по диаграмме файлов с исходным кодом на Си, осуществляет компиляцию его в отладочном режиме, после чего запускает отладчик GDB [33], загружает в него скомпилированный бинарный файл и позволяет осуществлять его пошаговую отладку. При этом каждый блок диаграммы привязывается к сгенерированным по этому блоку строкам (одной или нескольким) в файле исходных текстов. Это позволяет в процессе отладки подсвечивать блок, которому соответствует исполняемая в отладчике строка кода. Возможно также выполнение некоторых команд GDB посредством интерфейса QReal: построчное выполнение программы, установка точек останова и переходы между ними, завершение отладки. При выборе соответствующих пунктов в меню QReal нужная команда передается в запущенный экземпляр GDB, после чего в информационном окне QReal отображается вывод отладчика после выполнения данной команды.

Как было отмечено выше, изначально описанные инструменты были запрограммированы вручную, однако в дальнейшем интерпретатор и отладчик были описаны с помощью средств описания исполнимой семантики.

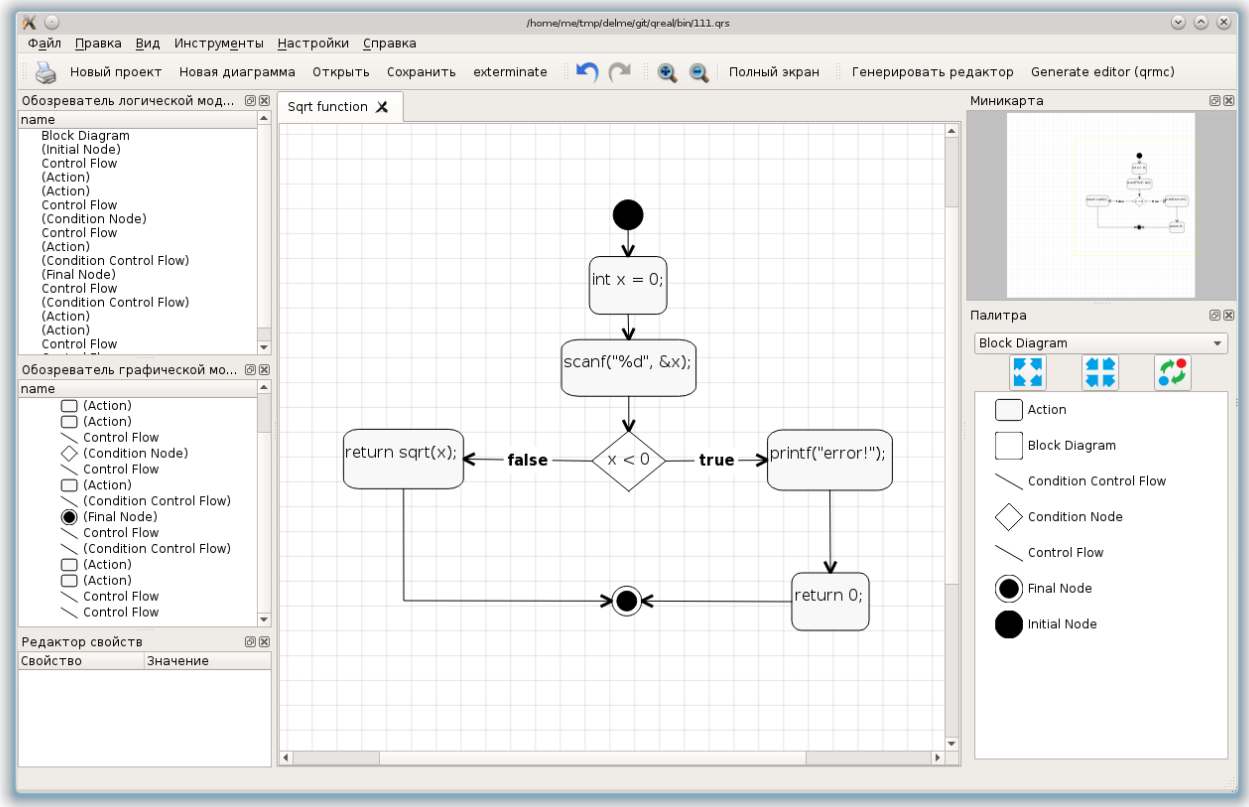

Рисунок 23. Внешний вид среды, основанной на языке блок-схем

6.4. Среда разработки QReal:Ubiq

На кафедре системного программирования СПбГУ уже несколько лет разрабатывается платформа Ubiq Mobile [37], упрощающая создание мобильных приложений, являющихся компонентами распределенных информационных систем. Такие системы характеризуются большим числом пользователей, сложной бизнес-логикой, выполняемой большей частью на сервере, а также большим набором мобильных платформ, на которых будут работать создаваемые приложения. Типовое приложение, создаваемое с помощью Ubiq Mobile, состоит из сервера и мобильного клиента (или набора клиентов), общающихся между собой посредством протокола, определяемого платформой. Платформа инкапсулирует в себе бо́льшую часть специфики, связанной с клиент-серверным взаимодействием, работой на мобильном устройстве и т.п., предоставляя удобные средства разработки кроссплатформенных

мобильных приложений C#-программистам, не владеющим специальными навыками программирования под мобильные устройства. Целью проекта QReal:Ubiq стало создание среды визуального программирования приложений с помощью платформы Ubiq Mobile, которая позволила бы еще больше упростить процесс разработки мобильных систем за счет замены программирования на C# моделированием и, следовательно, расширить круг людей, которые могут создавать такие приложения самостоятельно.

Первая версия QReal:Ubiq была создана в 2011 году и представлена на конференции FRUCT 2011 [72]. Данное решение позволило повысить наглядность и понятность программ, не требовало от разработчика глубоких знаний объектно-ориентированного программирования, однако все же для создания приложений было необходимо писать в блоках довольно много текстового кода, а также понимать общее внутреннее устройство платформы Ubiq Mobile. В дальнейшем была предпринята попытка ограничения предметной области целевых мобильных приложений и создания инструментов для их моделирования, которые бы не требовали от проектировщика знания целевого языка программирования (C# в случае платформы Ubiq Moblie). Так, был выбран класс онлайн игр для двух пользователей с игровым полем (например, “Морской бой”, “Крестики-нолики” и тому подобные игры), а также класс простых информационных приложений (так называемые, приложения-визитки, отражающие статичную текстовую и мультимедийную информацию, размещенную на ряде экранов). В результате были созданы следующие инструменты.

1. Редактор экранных форм, позволяющий схематично описывать внешний вид экранов приложения. Каждому элементу управления возможно задать имя обработчика события, с ним связанного. Простые переходы между экранами (например, переход на заданный экран при нажатии на кнопку) возможно задавать прямо на этой диаграмме с помощью

специального вида связей. Пример диаграммы, задающей экранную форму приложения для игры в “Морской бой”, см. на рис. 24.

2. Редактор логики обработчиков событий позволяет задать алгоритм обработки событий элементов управления. Язык содержит в себе операторы условий, циклов, поддерживает работу с переменными и позволяет считывать и изменять параметры элементов управления на экранных формах. На рис. 25 приведен пример диаграммы, которая проверяет условие завершения игры и выполняет соответствующую инициализацию внутренних переменных приложения.
3. Редактор диаграмм логических условий позволяет задавать нетривиальные логические выражения, которые могут быть в дальнейшем использованы в операторах на диаграммах обработчиков событий. На рис. 26 приведен пример диаграммы, описывающей условие выигрыша одной из сторон.
4. Генератор исходного кода позволяет по созданным диаграммам автоматически получить код прототипа соответствующего приложения на языке C# с использованием средств платформы Ubiq Mobile.

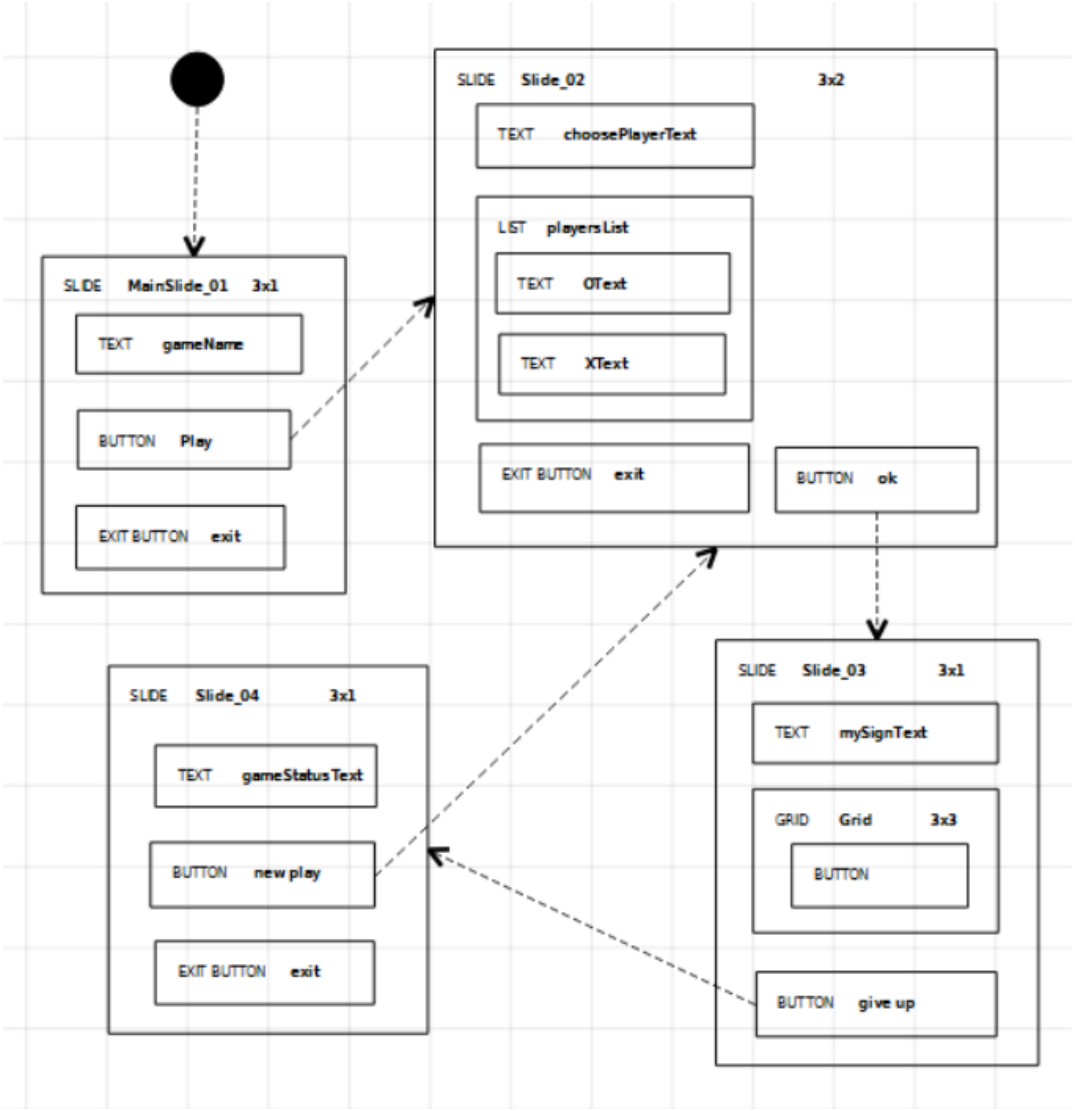

Рисунок 114. Диаграмма редактора форм QReal:Ubiq

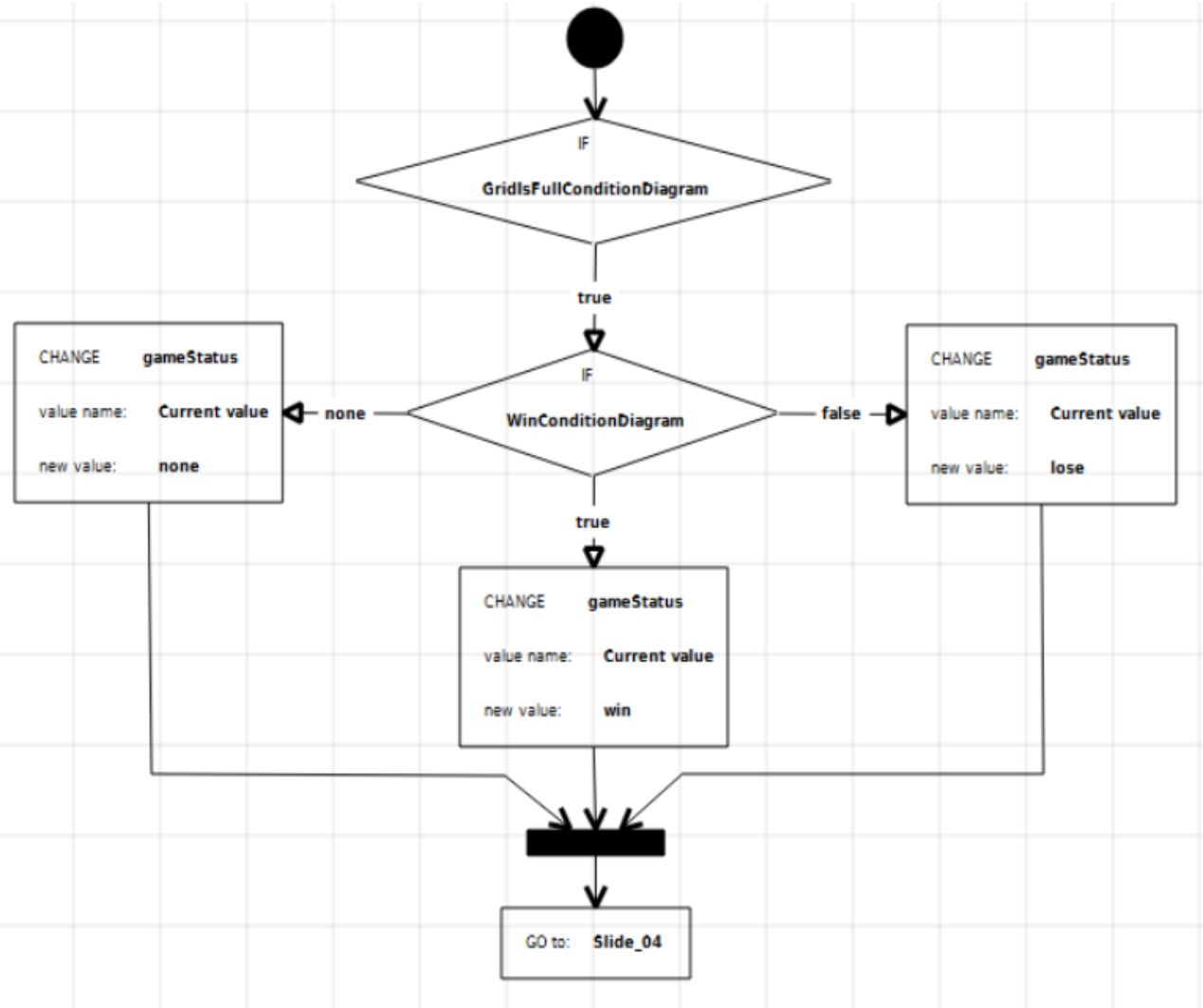

Рисунок 12. Пример диаграммы обработчика событий для игрового поля

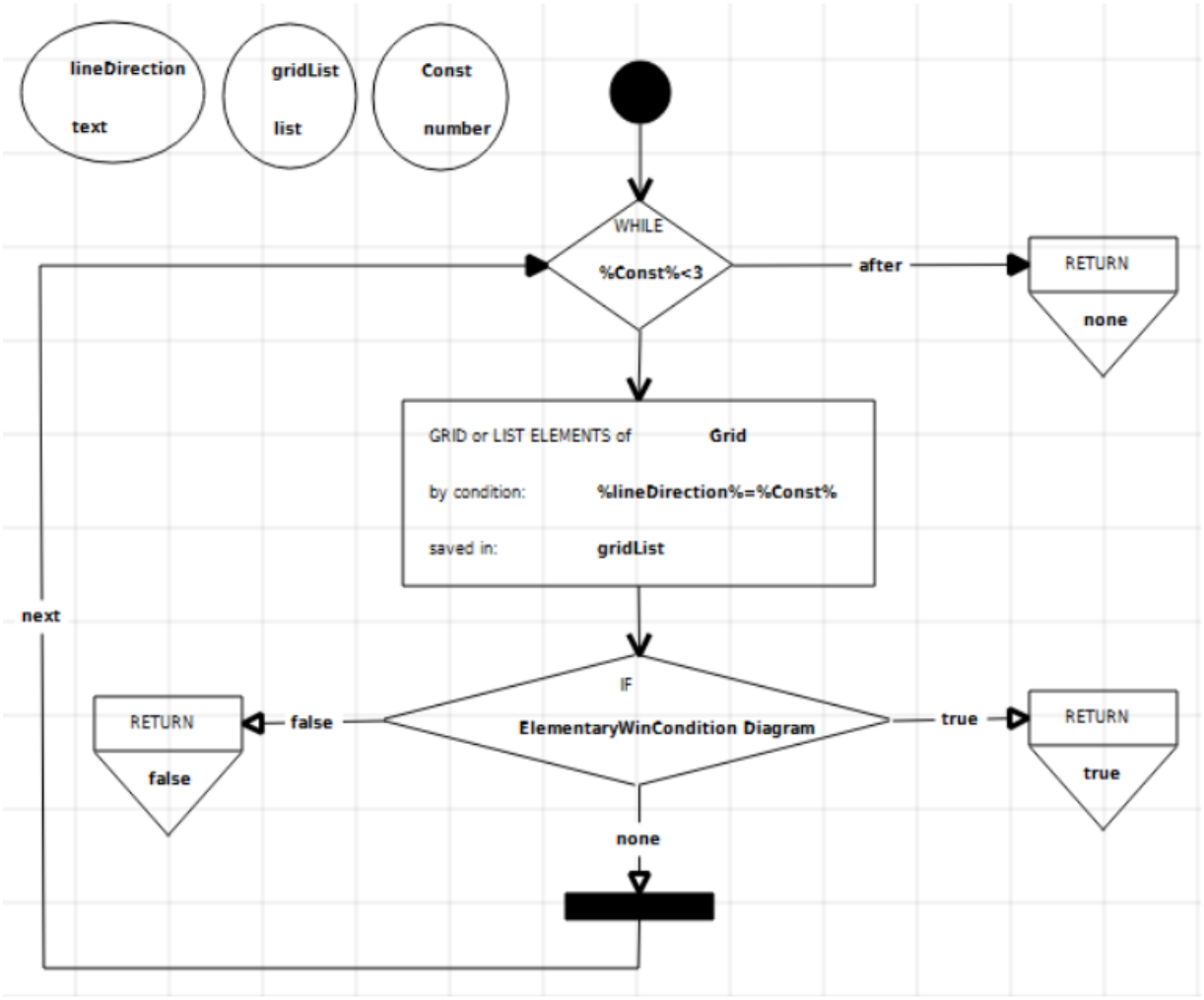

Рисунок 13. Пример диаграммы, описывающей условие завершения игры

6.5. Редактор диаграмм машин состояний для проекта компьютерного зрения

Редактор диаграмм машин состояний был создан при работе над проектом в ООО “Системы компьютерного зрения”. Проект был нацелен на создание системы управления телевизором взмахами руки вверх-вниз и влево-вправо. Так как при проектировании архитектуры решения было принято решение основывать реализацию на машинах состояний, с помощью QReal был создан подходящей графический язык и соответствующий генератор кода на C++. Язык состоял из четырех типов элементов: начальное состояние, состояние завершенного жеста,

состояние незавершенного жеста и таймер, который присоединялся к элементу состояния, в который переходило управление по истечении определенного промежутка времени неактивности системы. Интересной особенностью языка и генератора стало то, что это был неграфовый язык: помимо связей между элементами имело значение и геометрическое расположение элементов на диаграмме. Так, движение руки вправо переводило автомат в правое состояние, возврат руки в начальное положение — возврат в среднее состояние ожидания. Пример создаваемых на данном языке диаграмм см. на рис. 27.

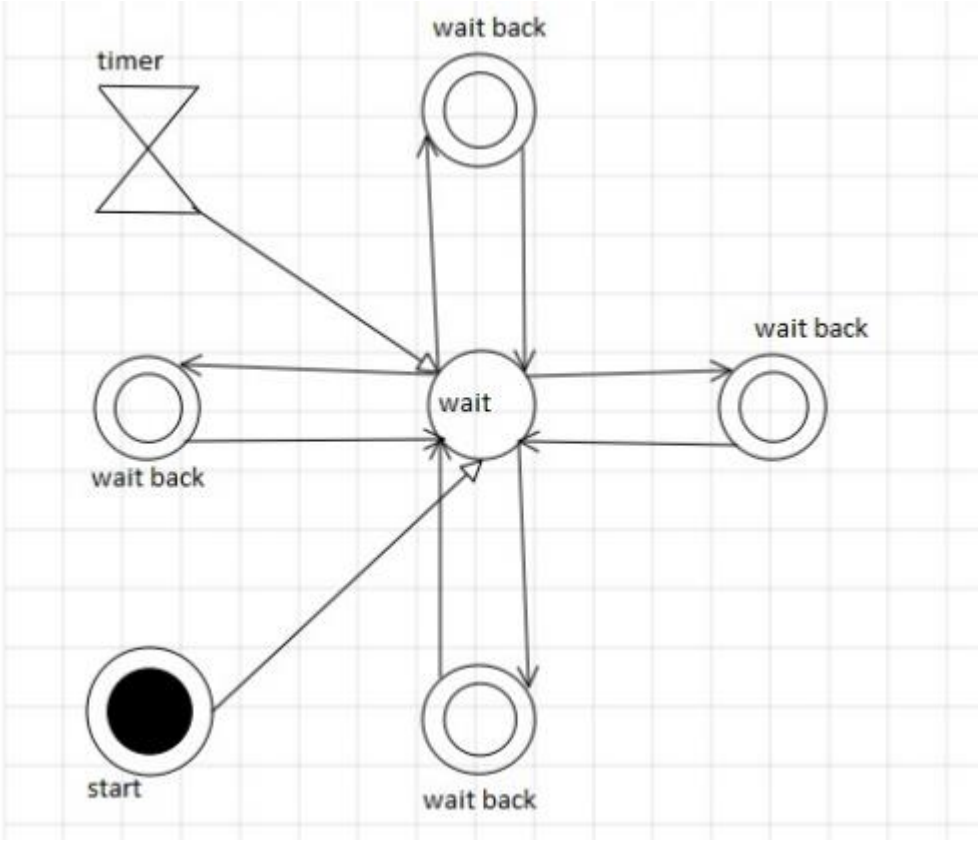

Рисунок 147. Пример диаграммы машин состояний для проекта компьютерного зрения

6.6. Сравнение DSM-платформ

Для сравнения функциональности разработанной платформы QReal с наиболее

популярными аналогами и доказательства соответствия полученных результатов целям работы был проведён следующий эксперимент.

В дополнение к платформе QReal было выбрано ещё две программных системы: MetaEdit+, которая является наиболее распространённой среди коммерческих платформ, и Eclipse Sirius – наиболее зрелый открытый продукт, построенный на базе Eclipse Modeling Project. Была поставлена задача создания визуальной среды для описания схем данных, основанной на нотации Чена «Сущность-связь» [73]. Разрабатываемая среда в идеале должна содержать:

- визуальный редактор;
- генератор скриптов на языке SQL DDL для создания баз данных;
- механизм ограничений, проверяющий, что любая сущность на диаграмме имеет хотя бы один атрибут;
- средство осуществления рефакторинга моделей, позволяющее выделить ряд атрибутов в отдельную сущность.

Поставленную задачу выполняли совместно два человека, до этого имевших опыт работы со всеми перечисленными платформами. Замерялось время создания инструментов, результаты отражены в таблице 2.

Очевидно, что полученные результаты могут характеризовать эффективность разработанных средств лишь в первом приближении. Для более аккуратного сравнения необходимо провести масштабный эксперимент, в котором примут участие не менее 10 человек разной квалификации и степени знакомства с тестируемыми платформами. Кроме времени выполнения задач необходимо ввести и другие метрики: степень соответствия решения постановке задачи, количество выполненных операций (например, открытых экранов, нажатий мышью, длина пути, пройденного курсором мыши) и т.п. Также следует оценивать субъективную оценку об удобстве использования и удовлетворённости полученным результатом. Кроме этого в эксперимент требуется добавить задачу создания поведенческого языка

(например, языка описаний сетей Петри). Это позволит также сравнить средства платформ для создания интерпретаторов и отладчиков такого рода языков. Также в список создаваемых инструментов следует добавить средства поддержки многопользовательской работы при работе с диаграммами и другие инструменты из п. 2.5. Всё это требует аккуратного планирования и серьёзных ресурсов. Подобный эксперимент будет проведён после устранения пробелов в документации и обучающих материалах платформы QReal, выявленных в рамках описанного тестирования.

Таблица 2. Результаты сравнения DSM платформ

	Редактор диаграмм	Генератор	Механизм ограничений	Средства рефакторинга
Eclipse Sirius	2 часа	50 минут	17 минут	40 минут
MetaEdit+	20 минут	30 минут	-	-
QReal (режим метаредактора)	15 минут	15 минут	10 минут	15 минут
QReal (режим «метамоделирования на лету»[9])	5 минут	15 минут	-	-

Сформулируем некоторые выводы из проведённого сравнения.

- Разработанная в рамках диссертационной работы платформа QReal успешно позволяет создавать среды разработки для специализированных визуальных языков. При этом для создания типовых инструментов (таких, как редактор, генератор, средства проверки ограничений и осуществления рефакторингов моделей) требуется меньше времени, чем при использовании других платформ. Особенно хорошо себя показал

[9] Подробнее про «метамоделирование на лету» см. в диссертации Ю.В.Литвинова

режим «метамоделирования на лету» – подход к созданию диаграммных редакторов, разработанный на базе платформы QReal Ю.В.Литвиновым в его диссертационной работе.

- Платформа MetaEdit+ подтвердила свою стабильность и эффективность, однако её функциональных средств явно недостаточно для создания полноценных визуальных сред разработки. Все необходимые инструменты, не входящие в стандартную поставку, авторы платформы рекомендуют реализовывать вручную посредством использования внешних API.
- Eclipse Modeling Project объединяет в себе десятки проектов, ряд которых разработчик конечного решения вынужден интегрировать друг с другом самостоятельно. Для сравнения была выбрана среда Eclipse Sirius, являющаяся на момент проведения эксперимента наиболее зрелым комплексным средством метамоделирования, однако инструменты для создания генераторов, ограничений и рефакторингов в нём отсутствовали, и их пришлось искать и добавлять «вручную». В этом смысле большое и активно развивающееся сообщество проекта оборачивается дополнительными проблемами для пользователей: решению одних и тех же задач может быть посвящено несколько проектов, и они могут находиться в разной степени зрелости. К примеру, для задания рефакторингов вначале нами был выбран инструмент EMF Refactor, который не заработал в связке с последней версией Sirius. Затем была предпринята попытка использовать средство Refactory, однако примеры из стандартной поставки инструмента в условиях практически полного отсутствия актуальной документации также не удалось запустить на выполнение. В итоге работающее решение было получено при помощи инструмента Epsilon (интеграцию которого с Sirius также

пришлось выполнять посредством серии проб и ошибок). Такая ситуация может крайне отрицательно сказаться на продуктивности разработчиков, имеющих мало опыта в использовании подобных инструментариев.

6.7. Заключение

В этой главе описано несколько предметно-ориентированных решений, созданных на базе созданной в рамках данной работы платформе QReal. Апробация показала, что архитектура платформы хорошо подходит для такого рода задач, позволяя создавать полнофункциональные среды в базовой комплектации (графический редактор и репозиторий моделей) минимальными усилиями разработчиков (по сути, необходимо лишь задать абстрактный и конкретных синтаксис языка), а также далее включать в конечное решение нужные компоненты платформы. В случае необходимости добавления в решение специфичных для задачи компонент (таких, как среда двумерного имитационного моделирования QReal:Robots) платформа предоставляет краткий, но ёмкий программный интерфейс для расширения функциональности кодированием «вручную».

Заключение

Итоги диссертационной работы

Итоги выполненного диссертационного исследования, таковы.

1. Предложен метод для создания инструментов распознавания жестов для диаграммных редакторов предметно-ориентированных языков.
2. Разработан метод формального задания операционной семантики предметно-ориентированных визуальных языков, позволяющий автоматически создавать для них интерпретаторы и отладчики.
3. Предложена модель (архитектура) программного комплекса (DSM-платформы), позволяющего автоматизированно создавать большинство типовых компонентов современных CASE-систем.
4. Выполнена реализация и апробация созданной DSM-платформы на практических задачах, подтвердившая работоспособность созданных инструментов и предложенных решений.

Код реализованного решения опубликован на сервисе GitHub под лицензией Apache License Version 2.0 [22].

Рекомендации по применению результатов работы

При применении результатов данной работы в научных исследованиях или промышленности необходимо учитывать следующие аспекты.

- Архитектура подобной программной платформы должна быть как можно более модульной, причём должна быть возможность отключать большую часть модулей на стадии компиляции или во время исполнения программы. Это необходимо для более точной настройки функциональности платформы под нужды конкретного предметно-

ориентированного решения.

- При разработке промышленных DSM решений может понадобиться реализовать дополнительные инструменты, специализированные для решаемой задачи. Например, это может быть среда трёхмерного имитационного моделирования для среды программирования роботов или эмулятор для среды разработки мобильных приложений. Такие инструменты являются слишком нестандартными и вряд ли смогут быть реализованы автоматизированно средствами DSM платформы, поэтому платформа должна предоставлять функционально богатый интерфейс прикладного программирования (API) для расширения своей функциональности сторонними компонентами. Например, в такой интерфейс могут входить операции с содержимым репозитория или с элементами пользовательского интерфейса DSM решения.
- Графические языки, инструменты поддержки которых возможно создавать с помощью платформы QReal, должны быть основаны на концепции графов (то есть явно иметь узлы и связи между ними). На данный момент специальных средств для удобного создания инструментов поддержки неграфовых языков (например, языков, в которых семантика моделей зависит от взаимного расположения элементов и связей на диаграммах) в платформе нет, однако в простых случаях создание таких инструментов всё же возможно (см., например, раздел 6.5).

Перспективы дальнейшей разработки тематики

Перечислим также перспективные направления развития данного проекта.

- Поддержка эволюции языков (автоматизация миграций моделей при выходе новых версий языка) [47].

- Реализация средств автоматической генерации механизма автодополнения моделей по метамодели языка.
- Расширения границ применимости механизма рефакторингов.
- Создание законченного решения задачи автоматизированной разработки генераторов кода.
- Доработка компонент платформы для поддержки неграфовых визуальных языков.
- Проведение ряда экспериментов по использованию как DSM-платформы, так и созданных на ее основе решений. Сбор данных, построение и анализ метрик позволит сделать выводы касаемо степени удовлетворенности пользователей текущим интерфейсом системы, позволит найти и оптимизировать проблемные места, усложняющие работу с платформой.
- Адаптация и перенос частей платформы в веб-окружение, реализация среды моделирования, доступной онлайн через стандартные браузеры.

Список литературы

1. Библиотека QScintilla [Электронный ресурс]. – http://www.riverbankcomputing.com/software/qscintilla (дата обращения: 18.04.2015).
2. Библиотека ZeroC Ice [Электронный ресурс]. – URL: http://www.zeroc.com/ice.html (дата обращения: 18.04.2015).
3. Брыксин, Т. А. Опыт проведения студенческих проектов на примере реализации metaCASE-системы QReal [Текст] / Т. А. Брыксин // Компьютерные инструменты в образовании. – №5. – 2011. – С. 46–63.
4. Брыксин, Т. А. О генеративном подходе к созданию визуальных редакторов [Текст] / Т. А. Брыксин // Материалы Второй всероссийской научно-практической конференции, посвященной памяти засл. деятеля науки РФ профессора В. Ф. Волкодавова. – 2009. – С. 207–209.
5. Брыксин, Т. А. Среда визуального программирования роботов QReal:Robots [Текст] / Т. А. Брыксин, Ю. В. Литвинов // Материалы международной конференции «Информационные технологии в образовании и науке». – 2011. – С. 332–334.
6. Брыксин, Т. А. Студенческие проекты по программированию как средство формирования профессиональных навыков [Текст] / Т. А. Брыксин // Системное программирование. – №6. – 2011. – С. 116–135.
7. Брыксин, Т. А. Технология визуального предметно-ориентированного проектирования и разработки ПО QReal [Текст] / Т. А. Брыксин, Ю. В. Литвинов // Материалы второй научно-технической конференции молодых специалистов «Старт в будущее», посвященной 50-летию полета Ю.А. Гагарина в космос. – 2011. – С. 222–225.

8. Инструментарий Borland Together [Электронный ресурс]. – URL: http://www.borland.com/Products/Requirements-Management/Together (дата обращения: 18.04.2015).
9. Инструментарий Eclipse Modeling Project [Электронный ресурс]. – URL: http://www.eclipse.org/modeling/ (дата обращения: 18.04.2015).
10. Инструментарий Graphviz [Электронный ресурс]. – URL: http://www.graphviz.org/ (дата обращения: 18.04.2015).
11. Инструментарий GenGED [Электронный ресурс]. – URL: http://user.cs.tu-berlin.de/~genged (дата обращения: 18.04.2015).
12. Инструментарий GROOVE [Электронный ресурс]. – URL: http://groove.sourceforge.net/groove-index.html (дата обращения: 18.04.2015).
13. Инструментарий Ideogramic UML [Электронный ресурс]. – URL: http://ideogramic-uml.software.informer.com/ (дата обращения: 18.04.2015).
14. Инструментарий MetaDepth [Электронный ресурс]. – URL: http://astreo.ii.uam.es/~jlara/metaDepth/ (дата обращения: 18.04.2015).
15. Инструментарий MetaEdit+ [Электронный ресурс]. – URL: http://www.metacase.com/ (дата обращения: 18.04.2015).
16. Инструментарий NXT-G [Электронный ресурс]. – URL: http://www.legoengineering.com/program/nxt-g/ (дата обращения: 18.04.2015).
17. Инструментарий Modeling Amalgamation Project [Электронный ресурс]. – URL: https://eclipse.org/modeling/amalgam/ (дата обращения: 18.04.2015).
18. Инструментарий Rational Software Architect [Электронный ресурс]. – URL: http://www-03.ibm.com/software/products/us/en/ratisoftarch (дата обращения: 18.04.2015).
19. Инструментарий Rational Rose [Электронный ресурс]. – URL: http://www-01.ibm.com/software/awdtools/developer/rose/ (дата обращения: 18.04.2015).

20. Инструментарий Robolab [Электронный ресурс]. – URL: http://www.legoengineering.com/program/robolab/ (дата обращения: 18.04.2015).
21. Инструментарий Visual Paradigm [Электронный ресурс]. – URL: http://www.visual-paradigm.com/ (дата обращения: 18.04.2015).
22. Исходный код проекта QReal [Электронный ресурс]. – URL: https://github.com/qreal/qreal (дата обращения: 18.04.2015).
23. Каталог CASE-инструментариев [Электронный ресурс]. – URL: http://www.dmoz.org/Computers/Programming/Methodologies/Modeling_Languages/Unified_Modeling_Language/Tools/ (дата обращения: 18.04.2015).
24. Каталог CASE-инструментариев [Электронный ресурс]. – URL: http://modeling-languages.com/uml-tools/ (дата обращения: 18.04.2015).
25. Каталог CASE-инструментариев [Электронный ресурс]. – URL: http://www.diagramming.org/ (дата обращения: 18.04.2015).
26. Кознов, Д. В. Визуальное моделирование компонентного программного обеспечения [Текст] / Д. В. Кознов. — Диссертация на соискания учёной степени кандидата физико-математических наук. — СПбГУ. — 2000. — 74 с.
27. Кознов, Д. В. Основы визуального моделирования [Текст] / Д. В. Кознов. – М: Бином. Лаборатория знаний. – 2008. – С. 248.
28. Кознов, Д. В. Языки визуального моделирования. Проектирование и визуализация программного обеспечения / Д. В. Кознов. — Изд-во Санкт-Петербургского университета. — 2004. — 150 с.
29. Кузенкова, А. С. Метамоделирование: современный подход к созданию средств визуального проектирования [Текст] / А. С. Кузенкова, Ю. В. Литвинов, Т. А. Брыксин // Материалы второй научно-технической конференции молодых специалистов «Старт в будущее», посвященной 50-летию полета Ю.А. Гагарина в космос. – 2011. – С. 228–231.
30. Кузенкова, А. С. Средства быстрой разработки предметно-ориентированных решений в metaCASE-средстве QReal [Текст] / А. С. Кузенкова,

А. О. Дерипаска, К. С. Таран, А. В. Подкопаев, Ю. В. Литвинов, Т. А. Брыксин // Научно-технические ведомости СПбГПУ. Информатика, телекоммуникации, управление. – № 4 (128). – 2011. – С. 142–145.

31. Лядова, Л. Н. Языковой инструментарий системы MetaLanguage [Текст] / Л. Н. Лядова, А. О. Сухов // Математика программных систем. Межвузовский сборник научных статей. — 2008. — С. 40–51.
32. Осечкина, М. С. Поддержка жестов мышью в мета-CASE-системах [Текст] / М. С. Осечкина, Т. А. Брыксин, Ю. В. Литвинов и др. // Системное программирование. – 2010. – №5. – С. 52-75.
33. Отладчик GDB [Электронный ресурс]. – URL: http://www.gnu.org/software/gdb/ (дата обращения: 18.04.2015).
34. Паронджанов, В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! [Текст] / В. Д. Паронджанов. – М: Дело, 2001. – 360 с. – ISBN 5–7749–0211–0.
35. Паронджанов, В. Д. Учись писать, читать и понимать алгоритмы. Алгоритмы для правильного мышления. Основы алгоритмизации [Текст] / В. Д. Паронджанов. – М: ДМК Пресс, 2012. – 520 с. – ISBN 978-5-94074-800-7.
36. Паронджанов, В. Д. Язык ДРАКОН. Краткое описание [Текст] / В. Д. Паронджанов. – Москва, 2009. – 124 с.
37. Платформа Ubiq Mobile [Электронный ресурс]. – URL: http://www.ubiqmobile.com/ (дата обращения: 18.04.2015).
38. Подкопаев, А. В. Генерация кода на основе графической модели [Текст] / А. В. Подкопаев, Т. А. Брыксин // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада «Технологии Microsoft в теории и практике программирования». – 2011. – С. 112–113.
39. Подкопаев, А. В. Средства описания генераторов кода для предметно-

ориентированных решений в metaCASE-средстве QReal [Текст] / А. В. Подкопаев, Т. А. Брыксин // СПИСОК-2012: Материалы всероссийской научной конференции по проблемам информатики. – 2012. – С. 49–55.

40. Поляков, В. А. Средство разработки визуальных интерпретаторов и отладчиков диаграмм в проекте QReal [Текст] / В. А. Поляков, Т. А. Брыксин // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада «Технологии Microsoft в теории и практике программирования». – 2013. – С. 80–81.
41. Поляков, В. А. Подходы к заданию семантики интерпретации диаграмм в рамках DSM-подхода [Текст] / В. А. Поляков, Т. А. Брыксин // Системное программирование. – №7. – 2012. – С. 187–216.
42. Поляков, В. А. Подходы к заданию семантики интерпретации диаграмм, основанные на технологии преобразования графов [Текст] / В. А. Поляков, Т. А. Брыксин // Компьютерные инструменты в образовании. – №2. – 2013. – С. 3–17.
43. Поляков, В. А. Разработка визуального интерпретатора моделей в системе QReal [Текст] / В. А. Поляков, Т. А. Брыксин // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада «Технологии Miscrosoft в теории и практике программирования». – 2011. – С. 58.
44. Сорокин, А. В. Обзор Eclipse Modeling Project [Текст] / А. В. Сорокин, Д. В. Кознов // Системное программирование. – 2010. – Т. 5. – С. 6–31.
45. Сухов, А. О. Разработка инструментальных средств создания визуальных предметно-ориентированных языков [Текст] : Дис. канд. ф.-м. наук: 05.13.11: защищена 05.12.2013 / А. О. Сухов; Институт системного программирования Российской академии наук. – 2013. – С. 157.

46. Сухов, А. О. Методы трансформации визуальных моделей [Текст] / А. О. Сухов. // Материалы III международной научно-технической конференции «Технологии разработки информационных систем ТРИС-2012». –2012. – Т. 1. – С. 120-124.
47. Таран, К. С. Проблема эволюции визуальных языков метамоделирования в metaCASE-системе QReal [Текст] / К. С. Таран, Т. А. Брыксин // Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада «Технологии Microsoft в теории и практике программирования». – 2011. – С. 159.
48. Терехов, А. Н. Архитектура среды визуального моделирования QReal [Текст] / А. Н. Терехов, Т. А. Брыксин, Ю. В. Литвинов, К. К. Смирнов, Г. А. Никандров, В. Ю. Иванов, Е. И. Такун // Системное программирование. – №4. – 2009. – С. 171–196.
49. Терехов, А. Н. Среда визуального программирования роботов QReal:Robots [Текст] / А. Н. Терехов, Т. А. Брыксин, Ю. В. Литвинов // III Всероссийская конференция «Современное технологическое обучение: от компьютера к роботу». – 2013. – С. 2–5.
50. Терехов, А. Н. Среда для обучения информатике и робототехнике QReal:Robots [Текст] / А. Н. Терехов, Ю. В. Литвинов, Т. А. Брыксин // Девятая независимая научно-практическая конференция «Разработка ПО 2013» (CEE SEC(R)-2013). – 2013.
51. Терехов, А. Н. Технологии программирования: учебное пособие [Текст] / А. Н. Терехов. – М.: Бином. Лаборатория знаний, 2007. – С. 152. – ISBN978-5-94774-669-3.
52. Терехов, А. Н. Real: методология и CASE-средство для разработки систем реального времени и информационных систем [Текст] / А. Н. Терехов,

К. Ю. Романовский, Д. В. Кознов, П. С. Долгов, А. Н. Иванов // Программирование. – 1999. – №5. – С. 44-52.

53. Терехов, А. Н. QReal: платформа визуального предметно-ориентированного моделирования [Текст] / А. Н. Терехов, Т. А. Брыксин, Ю. В. Литвинов // Программная инженерия. – № 6. – 2013. – С. 11–19.
54. Холтыгина, Н. А. Обзор реализации механизма циклической разработки диаграмм классов и программного кода в современных UML-средствах [Текст] / Н. А. Холтыгина, Д. В. Кознов // Системное программирование. – 2010. – №4. – С. 76-94.
55. Циклическая разработка [Электронный ресурс]. – URL: http://en.wikipedia.org/wiki/Round-trip_engineering (дата обращения: 18.04.2015).
56. Aaen, I. CASE Tool Bootstrapping – how little strokes fell great oaks [Text] / I. Aaen // Next Generation CASE Tools, edited by K. Lyytinen, V.-P. Tahvanainen. – IOS Press. – 1992. – P. 8-17.
57. Aaen, I. A tale of two countries: CASE experiences and expectations [Text] / I. Aaen, A. Siltanen, C. Sørensen, V.-P. Tahvanainen // Proceedings of the IFIP WG8. 2 Working Conference on The Impact of Computer Supported Technologies in Information Systems Development. – North-Holland Publishing Co. – 1992. – P. 61-91.
58. Albizuri-Romero, M. B. A retrospective view of CASE tools adoption [Text] / M. B. Albizuri-Romero // ACM SIGSOFT Software Engineering Notes. – Volume 25, Issue 2. – ACM Press. – 2000. – P. 46-50.
59. American National Standards Institute. 1975. ANSI/X3/SPARC Study Group on Data Base Management Systems; Interim Report. FDT (Bulletin of ACM SIGMOD) 7:2.
60. Arnold, B. Analysis of Industrial Challenges and Capabilities in Computer Science and Software Development Sector: Model Driven Engineering [Text] / B. Arnold,

M. R. Shadnam // Lecture Notes in Electrical Engineering. – Vol. 129, №6. – Springer. – 2012. – P. 499-505.

61. Bachman, C. W. Data structure diagrams [Text] / C. W. Bachman // ACM Sigmis Database. – ACM. – 1969. – Vol. 1, №2. – P. 4-10.
62. Bandener, N. Visual interpreter and debugger for dynamic models based on the Eclipse platform [Text] / N. Bandener // Diploma thesis, University of Paderborn. – 2009. – P. 125.
63. Banker, R. D. Reuse and productivity in integrated computer-aided software engineering: An empirical study. [Text] / R. D. Banker, R. J. Kauffman // MIS Q. 15, 3. – 1991, – P. 375–401.
64. Barker, R. CASE Method: Entity Relationship Modelling [Text] / R. Barker // Addison-Wesley Professional. – 1990. – P. 224.
65. Bardohl, R. GenGED – A Visual Definition Tool for Visual Modeling Environments [Text] / R. Bardohl, C. Ermel, I. Weinhold // Applications of Graph Transformations with Industrial Relevance. – Springer. – Vol. 3062. – 2004. – P. 413-419.
66. Bettin, J. Measuring the potential of domain-specific modeling techniques [Text] / J. Bettin // Proceedings of the Second Domain Specific Modeling Languages Workshop, Helsinki School of Economics. – Working Paper series. – 2002. – P. 39-44.
67. Borger, E. Abstract State Machines: A Method for High-Level System Design and Analysis / R. Stark, E. Borger // Springer-Verlag. – 2003. – P. 35.
68. Bouldin, B. Implementing CASE: From Strategy to Reality [Text] / B. Bouldin // Computer-world. – 1987. – P. 518.
69. Briand, L. Research-Based Innovation: A Tale of Three Projects in Model-Driven Engineering [Text] / L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, T. Yue // MODELS 2012. – Springer Berlin Heidelberg. – 2012. – P. 793-809.

70. Brooks, Jr., F. P. The Mythical Man-Month: Essays on Software Engineering / F. P. Brooks // 20th Anniversary Edition. – Reading, MA: Addison-Wesley. – 1995. – P. 322
71. Brown, A. Principles of CASE Tool Integrations [Text] / A. Brown, D. J. Carney, E. J. Morris, D. B. Smith, et. al. // Oxford University Press. – 1994. – P. 271.
72. Bryksin, T. Ubiq Mobile + QReal a Technology for Development of Distributed Mobile Services [Text] / T. Bryksin, Y. Litvinov, V. Onossovski, A. N. Terekhov // Proceedings 10th Conference of Open Innovations Association FRUCT and the 2nd Finnish-Russian Mobile Linux Summit. –State University of Aerospace Instrumentation (SUAI). – 2011. – P. 27-35.
73. Chen, P. The Entity-Relationship Model - Toward a Unified View of Data [Text] / P. Chen // ACM Transactions on Database Systems. – Vol. 1. – 1976. – P. 9-36.
74. Chikofsky, E. J. Reliability Engineering for Information Systems: The Emerging CASE Technology [Text] / E. J. Chikofsky // Index Technology Corporation. –1987.
75. Clark, T. Exploiting model driven technology: a tale of two startups [Text] / T. Clark, P.-A. Muller // Software & Systems Modeling. – Vol. 11, Issue 4. – Springer-Verlag New York. – 2012. – P. 481-493.
76. Cook, S. Domain-specific development with Visual Studio DSL Tools [Text] / S. Cook, G. Jones, S. Kent, A. C. Wills // Addison-Wesley Professional. – 2007. – P. 576.
77. Damm, C. H. An Evaluation of Workspace Awareness in Collaborative, Gesture-based Diagramming Tools [Text] / C. H. Damm, K. M. Hansen // People and Computers XVIII – Design for Life. – Springer London. – 2005. – P. 35-49.
78. Damm, C. H. Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools [Text] / C. H. Damm, K. M. Hansen, M. Thomsen, M. Tyrsted // 14th European Conference on Object-Oriented Programming. – Springer London. – 2000. – P. 27-43.

79. Davis, E. D. Extrinsic and Intrinsic Motivation to Use Computers in the Workplace [Text] / E. D. Davis, R. P. Bagozzi, P. R. Warshaw // Journal of Applied Social Psychology. – Vol. 22, №14. – 1992. – P. 1111-1132.
80. Dodani, D. A Picture is Worth a 1000 Words? [Text] / D. Dodani // Journal of Object Technology. – Vol. 5, №2. – 2006. – P. 35-40
81. Dubinsky, Y. Industrial ROI, Assessment, and Feedback, ModelWare Report No. 511731. / Y. Dubinsky, A. Hartman, M. Keren // 2006.
82. Elshazly, H. A Study on the Evaluation of CASE Technology [Text] / H. Elshazly, V. Grover // Journal of Information Technology Management. – Vol. 4, №1. – 1993.
83. Finkelstein, A. Software Process Modelling and Technology [Text] / A. Finkelstein, J. Kramer, B. Nuseibeh // John Wiley & Sons, Inc. – 1994.
84. Finlay, P. N. Perceptions of the benefits from introduction of CASE: An empirical study [Text] / P. N. Finlay, A. C. Mitchell // MIS quarterly. – Vol. 18, №4. – 1994. – P. 353-370.
85. FIPS Publication 184, Computer Systems Laboratory of the National Institute of Standards and Technology (NIST). – 1993.
86. Fuggetta, A. A Classification of CASE Technology [Text] / A. Fuggetta // Computer. – Vol. 26, №12. – IEEE Computer Society Press. – 1993. – P. 25-38.
87. Gillies, A. C. Managing Software Engineering [Text] / A. C. Gillies, P. Smith // Case studies and solutions. – Chapman & Hall. – 1994.
88. Gronback, R. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit [Text] / R. Gronback // Addison-Wesley Professional. – 2009. – P. 736.
89. Halpin, T. Information Modeling and Relational Databases [Text] / T. Halpin, T. Morgan // Morgan Kaufmann. – 2008. – P. 976.
90. Hausmann J. Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages [Text] / J. Hausmann // PhD Thesis, Paderborn, Faculty

of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn. – 2005. – P. 326.

91. Hoare C. A. R. An axiomatic basis for computer programming [Text] / C. A. R. Hoare // Communications of the ACM. – Vol. 12, №10. – P. 576–580.
92. Huang, R. Making Active CASE Tools – Toward the Next Generation CASE Tools [Text] / R. Huang // ACM SIGSOFT Software Engineering Notes. – Vol. 23, №1. – 1998. – P. 47-50.
93. Huff, C. C. Elements of a realistic CASE tool adoption budget [Text] / C. C. Huff // Communications of the ACM. – Vol. 35, №4. – 1992. – P. 45–54.
94. Hutchinson, J. Model-driven engineering practices in industry [Text] / J. Hutchinson, M. Rouncefield, J. Whittle // 33rd International Conference on Software Engineering. – IEEE. – 2011. – P. 633-642.
95. IBM Rational Unified Process [Electronic resource]. – URL: http://www-01.ibm.com/software/awdtools/rup/ (online, accessed: 18.04.2015).
96. Iivari, J. Why are CASE tools not used? [Text] / J. Iivari // Communications of the ACM. – Vol. 39, №10. – ACM New York. – 1996. – P. 94-103.
97. ISO I. S. O. 10303–11: 2004 Industrial automation systems and integration— Product data representation and exchange: Part 11 [Text] // Implementation methods: Clear text encoding of the exchange structure. – 2004.
98. ISO-IEC I. S. O. 10027: Information technology-Information Resource Dictionary System (IRDS)-Framework [Text] //ISO/IEC International standard. – 1990.
99. Jamart, P. A reflective approach to process model customization, enactment and evolution [Text] / P. Jamart, A. van Lamsweerde // 3rd International Conference on the Software Process. – 1994. – 21–32.
100. Jarzabek, S. The case for user-centered CASE tools [Text] / S. Jarzabek, R. Huang // Communications of the ACM. – Vol. 41, №8. – ACM New York. – 1998. – P. 93-99.

101. Kahn, G. Natural Semantics [Text] / G. Kahn // 4th Annual Symposium on Theoretical Aspects of Computer Science. – Springer-Verlag. – 1987. – P. 22-39.

102. Kelly, S. Domain-Specific Modeling: Enabling Full Code Generation [Text] / S. Kelly, J.-P. Tolvanen // Wiley-IEEE Computer Society Press. – 2008. – P. 448.

103. Kelly, S. Visual domain-specific modeling: benefits and experiences of using metaCASE tools [Text] / S. Kelly, J.-P. Tolvanen // International workshop on Model Engineering. – 2000.

104. Kemerer, C. How the learning curve affects CASE tool adoption [Text] / C. Kemerer // IEEE Software. – Vol. 9, №3. – 1992. – P. 23-28.

105. Kemerer, C. Learning curve models for integrated CASE tool management [Text] / C. Kemerer // Working paper 231, MIT Center for Information Systems Research. – Cambridge. – 1991.

106. Kieburtz, R. A software engineering experiment in software component generation [Text] / R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, L. Walton // 18th International Conference on Software Engineering. – IEEE Computer Society Press. – 1996.

107. Kotteman, J. Information systems planning and development: strategic postures and methodologies [Text] / J. Kotteman, B. Konsynski // Journal of Management Information Systems. – Vol. 1, №2. – 1984. – P. 45-63.

108. Kuhn, A. An exploratory study of forces and frictions affecting large-scale model-driven development [Text] / A. Kuhn, G. C. Murphy, C. A. Thompson // 15th International Conference MODELS 2012. – Springer Berlin Heidelberg. – 2012. – P. 352-367.

109. Kull, D. The Rough Road to Productivity [Text] / D. Kull // Computer Decisions. – 1987. – P. 30-41.

110. Kusters, R. J. On the practical use of CASE-tools: results of a survey [Text] / R. J. Kusters, G. M. Wijers // 6th International Workshop on CASE. –IEEE Computer Society Press. – 1993. – P. 2-10.

111. Kuzenkova, A. QReal DSM Platform: An Environment for Creation of Specific Visual IDEs [Текст] / A. Kuzenkova, A. Deripaska, T. Bryksin, Y. Litvinov, V. Polyakov // Proceedings of 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2013). – 2013. – P. 251–257.

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

113. Lending, D. The Changing Systems Development Job: A Job Characteristics Approach [Text] / D. Lending, N. L. Chervany // Proceedings of the 1997 ACM SIGCPR Conference. – ACM Press. – 1997, – P. 127-137.

114. Lending, D. The use of CASE tools [Text] / D. Lending, N. L. Chervany // Proceedings of the 1998 ACM SIGCPR conference on Computer personnel research. – ACM Press. – 1998. – P. 49-58.

115. Mathew A. Software Development Using Executable UML (xUML) [Electronic resource]. – 2002. – URL: http://se.cs.depaul.edu/ise/zoom/projects/statechart/SE690DetailedPresentation.ppt (online, accessed 18.04.2015).

116. Martin, M. P. The Case against CASE [Text] / M. P. Martin // Journal of Systems Management. – Vol. 46. – 1995. – P. 54-57.

117. MDA Guide [Electronic resource]. – URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (online, accessed 18.04.2015).

118. Mellor S. Executable UML: A foundation for model-driven architecture [Text] / S. Mellor, M. Balcer // Addison Wesley. – 2002. – P. 416.

119. McClure, C. The CASE Experience [Text] / C. McClure // Byte Magazine. – 1989. – P. 235-236.

120. Model-driven architecture [Electronic resource]. – URL: http://www.omg.org/mda/ (online, accessed: 18.04.2015).

121. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach [Text] // Productivity Analysis. – 2003. – URL: http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf (online, accessed 18.04.2015).

122. Mohagheghi, P. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases [Text] / P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez // Empirical Software Engineering. – Vol. 18, №1. – Springer US. – 2013. – P. 89-116.

123. Narraway, D. Designing and generating mobile phone applications [Text] / D. Narraway // Presentation at MetaEdit+ Method Seminar. – 1998.

124. Norman, R. J. CASE productivity perceptions of software engineering professionals [Text] / R. J. Norman, J. F. Nunamaker Jr. / Communications of ACM. – Vol. 32, №9. – 1989. – P. 1102–1108.

125. Norman, R. J. CASE at the Start of the 1990's [Text] / R. J. Norman, W. Stevens, E. J. Chikofsky, J. Jenkins, B. L. Rubenstein, G. Forte // Proceedings of the 14th International Conference on Software Engineering. – IEEE. – 1991, P. 128-139.

126. Object Action Language Reference Manual [Text]. – 2008. – 77 p., URL: www.ooatool.com/docs/OAL08.pdf (online, accessed 18.04.2015).

127. Object Management Group [Electronic resource]. – URL: http://www.omg.org/ (online, accessed: 18.04.2015).

128. Ocampo C. Is CASE Technology Still Alive? [Text] / C. Ocampo, B. Albizuri // Actas de las III Jornadas de Ingenieria del Software. – 1998. – P. 127-139.

129. OMG, Meta Object Facility (MOF) specification [Electronic resource]. – URL: http://www.omg.org/spec/MOF/Current (online, accessed 18.04.2015).

130. Orlikowksi, W. J. CASE Tools and the IS Workplace [Text] / W. J. Orlikowksi // Proceedings of the 1998 ACM SIGCPR Conference on the Management of Information Systems Personnel. – 1988. – P. 88-97.

131. Object-Role Modeling (ORM) [Electronic resource]. – URL: http://en.wikipedia.org/wiki/Object-role_modeling (дата обращения: 18.04.2015).

132. Osechkina, M. Multistroke Mouse Gestures Recognition in QReal metaCASE Technology / M. Osechkina, Y. Litvinov, T. Bryksin // SYRCoSE 2012: Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering. – 2012. – P. 194-200.

133. Paige, R. F. Revealing Complexity through Domain-Specific Modelling and Analysis [Text] / R. F. Paige, P. J. Brooke, X. Ge, C. Power, F. R. Burton, S. Poulding // Proceedings of the 17th Monterey conference on Large-Scale Complex IT Systems: development, operation and management. – Springer Berlin Heidelberg. – 2012. – P. 251-265.

134. Perrone, G. Primary Product in the Development Life Cycle [Text] / G. Perrone // Software Magazine. – 1988. – P. 35-41.

135. Plotkin, G. A Structural Approach to Operational Semantics [Text] / G. Plotkin // Technical Report DAIMI FN-19. – University of Aarhus. – 1981. – P. 133.

136. Query/View/Transformation language [Electronic resource]. – URL: http://www.omg.org/spec/QVT/ (online, accessed: 18.04.2015).

137. Rensink, A. The GROOVE Simulator: A Tool for State Space Generation [Text] / A. Rensink // Applications of Graph Transformations with Industrial Relevance. – Springer Verlag. – 2004. – P. 479–485.

138. Rozenberg, G. Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. [Text] / G. Rozenberg / World Scientific. – 1997. – P. 572.

139. Russell, F. The case for Case [Text] / F. Russell // Software Engineering: A European Perspective. – IEEE Computer Society Press. – 1993. – P. 531-547.

140. Sadilek D. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages [Text] / D. Sadilek, G. Wachsmuth // ECMDA-FA. – 2008. – P. 64–79.

141. Scalable Vector Graphics [Electronic resource]. – URL: http://www.w3.org/Graphics/SVG/ (online, accessed: 18.04.2015).

142. Scott D. Towards a Mathematical Semantics for Computer Languages [Text] / D. Scott, C. Strachey // Computers and Automata. – Wiley. – 1971. – P. 19–46.

143. Selic, B. What will it take? A view on adoption of model-based methods in practice [Text] / B. Selic // Software and Systems Modeling (SoSyM). – Vol. 11, №4. – Springer-Verlag New York. – 2012. – P. 513-526.

144. Senn, J. A. The Other Side of CASE Implementation: Best Practices for Success [Text] / J. A. Senn, J. L. Wynekoop // Information Systems Management. – №12. – 1995. – P. 7-14.

145. Sgirka, R. A Quality Model of Metamodeling Systems [Text] / R. Sgirka, E. Eessaar // Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering: International Conference on Systems, Computing Sciences and Software Engineering (SCSS 10). – Springer. – 2013. – P. 543 - 555.

146. Shlaer-Mellor Action Language [Text]. – 1997. – P. 29. – URL: www.modelint.com/downloads/small.pdf (online, accessed 18.04.2015).

147. Sharma, A. Framework to define CASE tool requirements for distributed environment [Text] / A. Sharma // ACM SIGSOFT Software Engineering Notes. – Vol. 19, №1. – ACM Press. – 1994. – P. 86-89.

148. Slonneger K. Syntax and Semantics of Programming Languages, A Laboratory Based Approach [Text] / K. Slonneger, B. Kurtz // Addison-Wesley Publishing. – 1995. – P. 187-222.

149. Sommer, M. The Impact of Computer-Assisted Engineering on Systems Development [Text] / M. Sommer // IFIP Transactions. – 1992. – P. 43-60.

150. Starr, L. Executable UML: How to Build Class Models [Text] / L. Starr // Prentice-Hall. – 2002. – P. 448.

151. Sukhov, A. An Approach to Development of Visual Modeling Toolkits [Text] / A. Sukhov, L. N. Lyadova // Advances in Information Science and Applications. Volumes I & II. – Proceedings of the 18th International Conference on Computers (part of CSCC '14). – 2014. – P. 61-66.

152. Sumner, M. Making the Transition to Computer-Assisted Software Engineering [Text] / M. Sumner // Proceedings of the 1992 ACM SIGCPR Conference. – ACM Press. – 1992. – P. 81-92.

153. Sumner, M. The Impact of Computer-Assisted Software Engineering on Systems Development [Text] / M. Sumner // The Impact of Computer Supported Technologies on Information Systems Development. – 1992. – P. 43-60.

154. Suydam, W. CASE Makes Strides Toward Automated Software Development [Text] / W. Suydam // Computer Design. – Vol. 26, №1. – 1987. – P. 49-70.

155. Taentzer, G. AGG: A Graph Transformation Environment for Modeling and Validation of Software [Text] / G. Taentzer // Applications of Graph Transformations with Industrial Relevance. Lecture Notes in Computer Science. – Vol. 3062. – Springer Berlin Heidelberg. – 2004. – P. 446-453.

156. Unified Modeling Language [Electronic resource]. – URL: http://uml.org/ (online, accessed 18.04.2015).

157. Vessey, I. CASE tools as collaborative support technologies [Text] / I. Vessey, A. P. Sravanapudi // Communications of the ACM. – Vol. 38, №1. – ACM Press. – 1995. – P. 83-95.

158. Wachsmuth, G. Modelling the Operational Semantics of Domain-Specific Modelling Languages [Text] / G. Wachsmuth // Generative and Transformational Techniques in Software Engineering II, Lecture Notes in Computer Science. – Vol. 5235. – Springer. – 2008. – P. 506-520.

159. Weiss, D. Software Product-line Engineering [Text] / D. Weiss, C. T. R. Lai // Addison Wesley, Longman. – 1999.

160. Wynekoop, J. L. The Implementation of CASE Tools: An Innovation Diffusion Approach [Text] / J. L. Wynekoop, J. A. Senn, S. A. Conger // The Impact of Computer Supported Technologies on Information Systems Development. – 1992. – P. 25-41.

161. Yourdon, E. Serious CASE in the 90's: What Do We Do When the Novelty Wears Off? [Text] / E. Yourdon // Show CASE Conference IV. – Vol. 10. – 1989.

162. Zagorsky, C. Case Study: Managing the Change to CASE [Text] / C. Zagorsky // Journal of Information Systems Management. – 1990. – P. 24-32.

Приложение А. Критика CASE-инструментов

В 1990-х годах рынок CASE-инструментов испытал существенный рост (по данным [71] с 4.8 млрд долларов США в 1990 году до примерно 12.11 млрд в 1995 году), однако в это же время одно за другим начинают появляться исследования, показывающие, что далеко не все приобретенные компаниями CASE-средства успешно внедряются в производственный процесс.

Так, в [104] утверждается, что после года внедрения в одной из крупных компаний 70% CASE-инструментов не применяются совсем, 25% используются одной-двумя группами разработчиков, а оставшиеся 5% широко используются, но при этом не раскрывая своих полных возможностей. По другим данным ([105]), компании не используют от 80 до 90 процентов покупаемых CASE-инструментов. В [57] сообщается, что из 102 изученных компаний в Дании и Финляндии лишь 20% использовали CASE-средства на постоянной основе при том, что 24% из этих компаний внедряли подобные инструменты уже более трех лет. Исследование [96] сообщает о том, что в 57% из организаций, сообщавших о использовании CASE-инструментов, более 75% программного обеспечения создавалось без использования CASE-средств. О схожих результатах говорит и ряд других работ [56, 82, 84, 110, 116, 124, 139, 144, 152].

Также зачастую оказывалось, что высшее руководство сильно переоценивает использование CASE-сред в своих компаниях: например, в [114] описывается организация, в которой данные, представленные вице-президентом компании, были завышены более, чем в шесть раз относительно показаний непосредственных руководителей команд разработчиков. Авторы данного исследования утверждают, что подобная ситуация типична для большинства из рассматриваемых ими 87 компаний. По их мнению, это объясняется тем, что компании тратят большие средства на внедрение CASE-средств большей частью по инициативе высшего руководства, и, если эти инструменты не находят популярности у разработчиков,

менеджерам нижнего и среднего звена приходится скрывать реальное положение вещей.

Рассмотрим причины подобной непопулярности CASE-инструментов среди разработчиков.

Введение в производственный процесс любой новой технологии требует первоначальных вложений и/или понижения продуктивности работы из-за адаптации этой технологии[10]. По мнению ряда авторов [58, 68, 104, 161], именно это является причиной неудач внедрения большинства CASE-инструментов, покрывающих весь процесс разработки ПО. Пытаясь использовать сложные инструменты, меняющие большую часть устоявшихся в компании процессов, разработчики разочаровываются результатами первых пилотных проектов, выполненных с использованием новых технологий, и со временем отказываются от них [125, 152]. В [105] дается следующая оценка при попытке внедрения CASE-инструментов: только вслед за падением продуктивности на 50% на протяжении шести месяцев и возвращением на привычный уровень в последующие шесть месяцев можно ожидать от 30 до 50% роста производительности труда разработчиков. Разумеется, далеко не все компании готовы на подобные вложения в долгосрочную перспективу с учетом высокой конкуренции в области разработки ПО [58, 143, 149].

Стоимость внедрения CASE-средств в производственный процесс — еще одна из главных причин, по которой компании отказывались от использования CASE-средств [58, 93, 144, 152, 161]. Цена самих инструментов (хоть и немалая — по данным [93], на начало 1990-х годов в пределах от $500 до $10000 в зависимости от состава инструментария) была лишь небольшой частью от общей стоимости внедрения этих инструментов. Существенная часть приходилась на обучение, на соответствующее аппаратное обеспечение, а также на потери производительности на

[10] Кривая обучаемости, URL: http://en.wikipedia.org/wiki/Learning_curve (дата обращения: 18.04.2015)

время пилотных проектов, апробации новых программных средств и подстройки существующих процессов. По оценкам [93], внедрение CASE-средств стоит для компании порядка $18000 в год на одного разработчика. В [144] приводится похожая оценка — $22000. О сложности и серьезных затратах (как временных, так и материальных) на внедрение CASE-инструментов в процесс разработки ПО также пишут [82] и [125].

Ситуация усугублялась тем, что большинство CASE-инструментариев были довольно плохо документированы и предоставляли крайне мало или не предоставляли вовсе информации о том, как их использовать для разработки ПО [125, 128, 149]. Предполагалось, что пользователи будут знакомы с методологией, которую поддерживает тот или иной инструментарий. При этом многие из организаций, приобретавших CASE-инструменты, в силу дороговизны не предоставляли своим сотрудникам специального обучения по использованию внедряемых инструментов [162], что заставляло разработчиков разбираться с ними самостоятельно, еще больше понижая их продуктивность на время адаптации инструментов.

Довольно очевидно, что невозможно создать универсальное инструментальное средство, которое бы оптимально подходило для разработки произвольного ПО в любой компании, купившей это средство. Так или иначе компаниям приходится приспосабливать инструменты под особенности своих процессов, создаваемых программ, разработчиков и клиентов [58]. Однако, большинство инструментариев продавалось в жестко заданной готовой конфигурации и почти никак не умели настраиваться на реалии использующих их людей — например, заменить встроенный текстовый редактор на уже имеющийся у разработчика привычный ему редактор или быть в состоянии адаптироваться к изменениям в процессе разработки, на который ориентирован данный инструмент [92, 125]. Тратя крупные суммы денег на внедрение какой-либо технологии, компании часто попадали в состояние

зависимости от поставщика[11], в котором не могли сменить используемые инструменты без существенных затрат, и уже им приходилось подстраиваться под приобретенные инструменты, а не наоборот. Ряд более сложных сред разработки, основанных на моделях поддерживаемых ими процессов (метамоделях) [99], позволял конфигурировать инструмент для поддержки изменений в процессе, однако это было доступно лишь для экспертов. Для рядовых пользователей процесс разработки выглядит как взаимодействие человеческих ресурсов и программных средств, вовлеченных в ряд работ [92]. Имея понятные и наглядные (например, диаграммные) средства описания взаимодействия этих сущностей, можно было бы эффективно настраивать CASE-среду под изменения в имеющихся в компании процессах.

Ряд исследований [153, 169] отмечает, что при использовании CASE-средств происходит существенное перераспределение работ и обязанностей — разработчики начинают выполнять другие действия, чем когда разработка происходила без использования CASE-инструментов. К примеру, по данным [114], при использовании CASE-средств в два раза больше времени уходит на фазу анализа и проектирования, чем без использования этих инструментов. Далеко не все люди одинаково реагируют на подобную смену акцентов: люди, которые склоняются к формальным методологиям, более охотно используют CASE-инструменты, чем те, кто таким методологиям предпочитает не следовать. Этот фактор оказался довольно существенным, поскольку многие организации оказались совершенно не готовы использовать CASE-среды из-за незрелости внедряемых в этих компаниях процессов разработки [125]. Резкая попытка жестко формализовать и автоматизировать процесс у многих разработчиков вызывает естественное сопротивление и нежелание использовать новые непривычные средства. Также введение формальных процессов плохо воспринимается разработчиками, предпочитающими работать автономно

[11] Зависимость от поставщика, URL: http://en.wikipedia.org/wiki/Vendor_lock-in (дата обращения: 18.04.2015)

относительно остальных членов команды (а это в той или иной степени верно для значительной части разработчиков [109, 113]).

Многие инструменты, используемые в 1990-х годах, ставили своей целью лишь решение определенного ряда задач и не претендовали на поддержку всего процесса разработки [58, 125, 147, 154]. В результате для автоматизации всех этапов разработки ПО приходилось использовать несколько различных инструментариев, чаще всего плохо или вовсе не интегрируемых друг с другом. Это также приводило к недовольству и нежеланию применять данные инструменты на практике. Практически все авторы, рассуждающие на эту тему, приходят к выводу, что инструментальная среда должна быть целостной и поддерживать как можно больше этапов разработки, в то же время давая возможность гибко настраивать себя под конкретные нужды пользователей.

Помимо экономических и организационных причин непопулярности CASE-инструментов среди разработчиков в 1980-1990-х годах существует также ряд причин технических, среди которых отмечают ориентированность на определенную платформу и операционную систему [92], слабое развитие средств конфигурационного управления [147] и обратного проектирования [92, 147], отсутствие средств командной разработки (как при синхронном, так и асинхронном взаимодействии) [77, 87, 147, 157], однако наиболее значимым в этой области ряд авторов выделяет неудобство использования CASE-средств [56, 78, 79, 87, 92, 100, 114, 147].

Так, [92] отмечает ориентированность имеющихся на рынке инструментов на техники и процессы, а не на их пользователя. По мнению автора, инструменты должны уметь настраиваться на уровень пользователя — больше помогать новичкам (разного рода подсказками, помощниками и мастерами) и давать больше возможностей экспертам, показывать разным группам пользователей разный набор элементов управления, по-разному представлять доступный функционал: например,

разделять сложные операции на несколько более простых или скрывать ряд операций для новичков, давать дополнительные возможности (например, средства метамоделирования) для экспертов. Это позволит сократить временны*е* и материальные затраты на обучение, сделает инструменты доступными более широкому кругу пользователей. Также, по мнению [56] и [92], инструменты должны давать своим пользователям немедленную пользу от их использования, а не только быть рассчитанными на долгосрочную перспективу в рамках всей компании — повышение качества создаваемого ПО, сокращение времени разработки и т.п.

В работе [100] отмечается слишком сильная ориентированность имеющихся CASE-сред на процессы и методологии разработки, излишняя их формальность. Это приводит к тому, что разработчик больше времени уделяет выполнению тех действий, которых от него требует CASE-инструмент, а не тех, которые непосредственно необходимы для решения задач, из которых состоит разработка ПО. Например, строго формализованные языки моделирования имеют большое количество ограничений на создаваемые с их помощью модели: правила использования элементов и связей между ними, которые, с одной стороны, не дают проектировщику создавать некорректные диаграммы и позволяют отлавливать ошибки на ранних этапах разработки, но с другой — мешают естественному мыслительному процессу разработчика, заставляя его совершать предписанные методологией действия. Авторы [100] предупреждают, что подобный инструмент будет встречен своими пользователями негативно и не будет использоваться с полной эффективностью. Инструмент должен помогать пользователю решать задачи, а не ставить перед ним новые. Например, в описанном выше случае мог бы быть полезен режим графического редактора, когда диаграммы создаются в произвольном виде, без каких-либо ограничений, накладываемых семантикой языка, целостностью диаграмм или ограничениями самого редактора. Это бы более способствовало свободному креативному мышлению разработчика. Об этой

проблеме пишут и авторы [78]: CASE-среды хорошо поддерживают поздние фазы разработки, такие как проектирование и реализация, и довольно плохо — начальные фазы, на которых проходит понимание и анализ предметной области и разрабатываемой системы. Для лучшей поддержки этих стадий CASE-средства должны иметь более простой, менее формальный и более гибкий интерфейс, позволяя разработчику думать о задаче, а не о том, как ее реализовать с помощью данного инструмента.

Удобный графический интерфейс используемых инструментов играет большую роль даже сам по себе. Так, исследование [79], проведенное психологами, показало, что чем больше разработчикам нравится внешний вид CASE-среды, чем более удобными для них кажутся входящие в ее состав инструменты, то не только они будут пользоваться подобной средой с большей охотой, но и будут считать ее более полезной, чем неудобная и непривлекательная среда. [114] также подтверждает, что удобство использования инструментального средства может быть не менее сильной мотивацией к использованию этого средства, нежели повышение в продуктивности, к которому этот инструмент может привести.

В 2000-х годах CASE-инструменты продолжали свое развитие, но, большей частью, в направлении совершенствования своих технических средств. Ряд современных исследований показывает успешные применения MDE-подхода к промышленной разработке ПО ([60, 69, 75, 94, 108, 122, 133, 143]), однако в целом авторы соглашаются, что подобные случаи не являются характерными для индустрии. Использование модельно-ориентированных средств в настоящее время — это скорее элемент общей культуры компаний и работающих в них программистов, нежели общеупотребимая практика. Так, в [143] утверждается, что не более 15% компаний в области программной инженерии в США используют модельно-ориентированные средства разработки. Основной проблемой этого авторы видят социально-экономически причины: даже в настоящее время программисты

обучаются с использованием традиционных текстовых средств разработки ПО, и это то, к чему они привыкают и не хотят менять. Моделирование воспринимаются большинством разработчиков как “рисование красивых диаграмм” — деятельность, которая отрывает от основного занятия (программирования), а потому зачастую осуществляется неохотно (и, зачастую, постфактум). Кроме того, для большинства людей, сознательно выбравших для себя разработку ПО в качестве профессии, программирование является крайне творческой деятельностью, причем для многих привлекательным в работе является не создание конечного продукта, а решение возникающих технических задач. Это приводит к тому, что любые попытки так или иначе убрать программирование из ежедневной работы и заменить его моделированием воспринимаются враждебно, аргументы о повышении продуктивности работы, качества создаваемого ПО или улучшения коммуникаций как внутри команды разработчиков, так и за ее пределами оказываются для программистов неубедительными, и они находят способы вернуться к привычному им программированию.

И действительно, язык UML не является языком программирования, и столь широкое использование его обуславливается в большей степени тем, что он стандартизован и в определенной степени понятен даже людям, не являющимся техническими специалистами (например, диаграммы UML могут использоваться при общении с заказчиками). К тому же, даже при условии наличия четко заданной семантики элементов языка (как, например, сделано в языке xUML) попытки программировать на таком языке приводят к моделям огромной сложности, перегруженным деталями.

Данную проблему стараются решать CASE-средства, основанные на специализированных языках (предметно-ориентированные решения), создаваемые под решение конкретных задач [102]. В рамках предметно-ориентированной парадигмы разработка ведется только в терминах визуальных моделей, а весь код

генерируется автоматически (и дальнейшее его редактирование “вручную” не допускается). Для быстрого создания таких средств используются инструментальные среды, называемые metaCASE-средствами или DSM-платформами. В отличие от традиционных CASE-средств, разрабатываемых годами большим коллективом разработчиков, DSM-решения, создаваемые на базе выбранной платформы одним или несколькими экспертами в течение нескольких дней или недель, не могут быть настолько же мощными функционально, однако большая часть необходимых для работы инструментов в их составе присутствует. Достигается это тем, что сама DSM платформа, как правило, содержит в себе довольно обширный набор инструментов, которые автоматически используются всеми DSM-решениями, создаваемыми на основе данной платформы. Ориентация на конкретную предметную область достигается настройкой и расширением средств платформы. Так, например, стандартная графо-графическая библиотека может быть дополнена конкретными изображениями фигур элементов для создания необходимого редактора диаграмм; по формальному описанию семантики, созданному разработчиком DSM-решения, средствами DSM-платформы может быть автоматически создан отладчик или генератор исходного кода по моделям на данном языке и т.п.

Приложение B. Обзор подходов к заданию исполнимой семантики визуальных языков

B.1. Непосредственное создание интерпретаторов

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. При работе данного модуля 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 и т.п.

Однако 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 при самой генерации машинного кода, так и при организации отслеживания хода исполнения. Поэтому следующим шагом может быть реализация отладчиков визуальных языков по двухуровневой схеме, то есть при помощи генерации кода на некоем текстовом языке и использования существующего отладчика этого языка.

Процесс отладки должен происходить в терминах этого исходного языка, т.е. отображать положение потока исполнения в исходных моделях, осуществлять установку точек останова в определённых местах моделей и др. Поэтому, например, команда выполнения следующего шага отладки (step over) должна учитывать изменение стека вызовов исходной программы, а не генерирующегося кода. Это

важно, поскольку каждому элементу модели на исходном визуальном языке может соответствовать несколько строк кода на целевом текстовом языке, поэтому для того, чтобы установить точку останова в терминах исходного языка, необходимо знать, на какую из этих строк кода нужно ставить точку останова в целевом языке. Кроме того, не всегда переход к следующему элементу потока исполнения в исходной модели будет соответствовать переходу на следующую строку кода в сгенерированном коде.

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 визуального языка конструкциям целевого текстового 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.

B.2. Исполнимый UML

Появление унифицированного языка моделирования UML 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 [115, 150]). 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.

Для задания семантики отдельных элементов в xUML используется явно определённая семантика действий (Precise Action Semantics, PAS [118]). 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.

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 создателях инструментов и пользователях, а не заложен в xUML. xUML явно определяет только PAS. Действия 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 [126], SMALL [146], TALL [118] и т.д.

Таким образом, упрощённо данный подход основан на представлении структуры разрабатываемой системы в виде диаграмм классов, а поведения в виде набора диаграмм состояний для каждого активного объекта. Действия при проходе каждого состояния в таких диаграммах записываются при помощи AL. При этом благодаря тому, что элементы xUML обладают чётко фиксированной семантикой, появляется возможность как однозначно генерировать исполняемый код по модели, так и отлаживать создаваемые модели: симулируются описанные объекты и обмен сообщениями между ними, при переходе из одного состояния в другое выполняется код, записанный на AL.

Разработка ПО с использованием xUML несомненно имеет свои преимущества: разработка ведётся с помощью подмножества широко известного языка, создаваемые диаграммы чётко формализованы и знакомы большинству разработчиков, существует мощная инструментальная поддержка, включающая симуляторы и отладчики создаваемых систем и многое другое. Однако, данный подход основан на традиционных понятиях объектно-ориентированного программирования и на стандартных конструкциях текстовых языков программирования (ветвления, циклы и т.п.), поэтому на практике значительного повышения уровня абстракции (и, как следствие, продуктивности труда разработчиков) он не даёт.

B.3. EProvide

EProvide (Eclipse Plugin for Prototyping Visual Interpreters and Debuggers) — это

подключаемый модуль (плагин) для среды разработки Eclipse, позволяющий быстро задавать операционную семантику для предметно-ориентированных языков, основанный на технологиях моделирования Eclipse [44]. Подход, использующийся в этом плагине, описан в статьях [140, 158].

В данном подходе для задания исполнимой семантики языка метамодель этого языка расширяется дополнительными элементами, обеспечивающими хранение состояния элементов создаваемой модели во время ее исполнения, функциональность точек останова, слежение за значениями свойств элементов модели и т.п. Правила, задающие семантику языка, в EProvide задаются в текстовом виде с помощью языков QVT Relations, Java, Abstract State Machines (ASM [67]), Prolog или Scheme. С помощью данных правил на основе расширенной метамодели языка при запуске процесса отладки создается и инициализируется модель времени исполнения (соответствующая исходной модели) и выполняются ее последовательные преобразования.

Данный подход позволяет описывать исполнимую семантику и создавать отладчики и интерпретаторы для произвольных языков, что существенно расширяет область его практической применимости. Инструмент EProvide реализован в рамках платформы Eclipse и хорошо интегрирован с другими инструментами платформы. Однако, описание семантики на текстовых языках сильно понижает наглядность этих описаний. Стоит также отметить, что создание полноценного отладчика и интерпретатора языка в платформе Eclipse — довольно сложный процесс, требующих серьезных технических навыков[12]. Перенос же созданного средства на другую платформу потребует повторной реализации серьезной части инструментов (например, создания интерпретатора QVT Relations), что весьма нетривиально.

[12] См., например, инструкцию от авторов EProvide: http://sourceforge.net/apps/mediawiki/eprovide/index.php?title=Tutorial:_Developing_a_Robot_DSL_with_EProvide (дата обращения: 18.04.2015)

B.4. Dynamic Meta Modeling

Другим способом задания семантики визуальных языков, использующим в качестве своей основы технологию преобразования графов, является динамическое метамоделирование (Dynamic Meta Modeling, DMM) [90]. Основной идеей в DMM является тот факт, что модель, созданная при помощи визуального языка, рассматривается как типизированный мультиграф с атрибутами и метками на узлах и рёбрах, допускающий наследование, связанное с возможностью расширения типов узлов и рёбер. Сам же процесс исполнения связан с преобразованиями таких типизированных графов. Задаётся соответствующий набор правил преобразования графов, которые впоследствии поочерёдно применяются к модели.

В общем случае ([138]), правило преобразования графов состоит из правой и левой части. При применении правила к модели происходит следующее: в модели ищется подграф, совпадающий с левой частью, и заменяется на то, что стоит в правой. При такой замене могут удаляться и добавляться элементы, переставляться концы связей и т.п.

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 [62].

Основным отличием данного подхода является его наглядность: семантика в

DMM задается визуально в виде правил графовых грамматик. Среди главных недостатков этого подхода можно выделить высокую алгоритмическую сложность, связанную с задачей поиска подграфа в графе, а также отсутствие реализации. Стоит, однако, отметить, что в левых частях правил преобразования графов в основном помещают небольшие фрагменты графов, что позволяет добиться приемлемого быстродействия инструментов, осуществляющих поиск данных шаблонов в моделях.

B.5. AToM^3

Примером реализации идей, заложенных в подходе DMM, является задание семантики в DSM-платформе AToM^3 (A Tool for Multi-Formalism and Meta-Modelling, [112]), написанной целиком на языке Python. Трансляция и интерпретация моделей, а также генерация кода по моделям осуществляется в AToM^3 при помощи графовых грамматик и последовательного преобразования моделей в соответствии с правилами грамматики. В рамках интерпретации и отладки AToM^3 поддерживает пошаговое и непрерывное исполнение модели, причем даже с учетом изменения модели «на лету», т.е. прямо перед исполнением очередного шага.

Модель каждого правила задается графически и сохраняется отдельно в виде скрипта на языке Python. По этой модели генерируется ещё один скрипт, являющийся кодом, который будет производить необходимые изменения согласно правилу. Первый файл необходим для того, чтобы можно было удобно изменять состав и логику работы этих правил, а второй — чтобы быстро видеть, какие операции будут выполнены при применении правила.

Другой отличительной от DMM-подхода особенностью является возможность при помощи специальных меток определять, что значение свойства элемента для применения правила может быть любым, задавать, оставить ли значение свойства элемента модели как есть или изменить его при выполнении правила. Также в

AToM^3 можно ставить более сложные ограничения на применение правила, связывающие атрибуты нескольких элементов левой части, при помощи OCL или задавать их на языке Python. Кроме дополнительных условий можно на тех же языках указывать список действий, которые необходимо исполнить после применения правила.

B.6. Сравнение подходов

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

Универсальность — возможность задания семантики для произвольных визуальных языков, т.е. оценка того, насколько подход может быть использован в предметно-ориентированном моделировании. Двухуровневая отладка совсем не соответствует парадигме DSM, т.к. данный подход ориентирован на создание единичного отладчика для определённого языка и не подразумевает удобного обобщения (0 баллов), xUML работает 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 и AToM^3 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).

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 и AToM^3, являются большей частью визуальными, а остальные — текстовыми. xUML, благодаря своей визуальной составляющей в виде диаграмм состояний жизненных циклов объектов, по данному критерию можно оценить в 1 балл.

Наглядность процесса интерпретации зависит от того, как эта интерпретация будет отображаться пользователю. DMM, EProvide и AToM^3 по ходу исполнения изменяют модель, а также при необходимости подсвечивают отдельные элементы, как, например, в EProvide, поэтому все эти подходы оценены 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.

Самым понятным подходом, на наш взгляд, является DMM, так как правила преобразования графов воспринимаются довольно интуитивно. У AToM^3 есть недостаток в том, что возможно ставить различные ограничения и писать исполняемые инструкции на OCL и Python, а это требует от разработчика дополнительных знаний. 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.

Таблица 3. Сравнительная таблица для разобранных подходов

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

Критерий наличия реализации: 3 балла, если реализация существует (для xUML, например, их существует много), 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.

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

Приложение С. Реализованный алгоритм поиска подграфа в модели репозитория QReal

Рассмотрим набор множеств и отображений, необходимых для работы этого алгоритма. Пусть исходная модель является графом $G = V \cup E$, где G — множество его вершин, E — множество рёбер, а искомый шаблон — граф $G' = V' \cup E'$. В дальнейшем будем придерживаться соглашения, что обозначения с штрихом относятся к сущностям, связанным с шаблоном, а без штриха — 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 представляется в виде списка, поэтому порядок добавления элементов в него имеет значение.
- Индекс вершины в W', рассматриваемой на текущем шаге, обозначим за i. В начале работы i инициализируем 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.

Отметим дополнительно, что так как 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. Рассмотрим принцип работы алгоритма целиком.

На первом шаге выбираем произвольную вершину из V' и ищем все вершины в V, совпадающие с ней по описанным выше критериям. Запускаем цикл, проходящий по ним и при проходе каждого выполняем следующие операции: обнуление i, очищение H', W' от старых значений и заполнение их информацией о соответствующих первых вершинах.

Второй шаг данного алгоритма представляет собой рекурсивную функцию, возвращающую значение логического типа. Работает функция следующим образом: берем вершину с индексом i из списка W', находим для нее первое ребро e' из E', не ведущее в H', т.е. вторая вершина, инцидентная с данным ребром, не должна лежать в H'. Если такого ребра не нашлось, то увеличиваем индекс i на единицу и запускаем второй шаг снова, если i не равно мощности множества W' (если же равно, то, значит, искомый граф найден целиком и в Φ будет содержаться промежуточный результат, то есть одно найденное вхождение данного графа. Добавляем его в список найденных подграфов и возвращаем истину). Обозначим вершину, выбранную из W', за x', а вершину, которой e' инцидентно помимо x', за a'. Вершину $\Phi(X')$обозначим за b. Далее ищем у b все ребра, совпадающие с e' и не ведущие в $\Phi(H')$. Обозначим полученное множество ребер за E_1. Если таких ребер 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 обозначим вершину, с которой оно соединено, за C. Ищем все ребра в E', имеющие одним из своих концов a', а вторым — вершину из H', и проверяем, есть ли такие же в E из c. Если ребер не нашлось, то переходим к следующему ребру из E_1. Если нашлись все нужные ребра, то $\Phi(A') = C$, а также $\forall r' \in E$ один из концов r' равен a', а второй принадлежит 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 единицу, убираем информацию об удаленной вершине в (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.

Приложение D. Акты о внедрении

ООО «КиберТех»
ТРИК

http://www.trikset.com
info@trikset.com

ООО "Кибернетические Технологии"
ОГРН 1127847447485
ИНН 7820329684
КПП 782001001
196606, Россия, Санкт-Петербург, г.Пушкин,
ул. Железнодорожная, д.68, оф. 26

05.12.15 № 15/12-15-1

АКТ
о внедрении результатов диссертационного исследования Брыксина Т.А. на тему
"**Платформа для создания специализированных визуальных сред разработки программного обеспечения**"
на соискание учёной степени кандидата технических наук

Результаты диссертационного исследования Брыксина Тимофея Александровича на тему "Платформа для создания специализированных визуальных сред разработки программного обеспечения" применяются для разработки и сопровождения среды программирования TRIK Studio. Эта среда разрабатывается на базе платформы QReal, архитектура которой предложена Т.А. Брыксиным в диссертационном исследовании. Наличие готовой программной платформы позволило существенно снизить затраты на разработку средства программирования контроллеров "ТРИК", разрабатываемых и производимых ООО "Кибернетические Технологии". Кроме того, наличие в составе платформы предложенных Т.А. Брыксиным средств улучшения пользовательского интерфейса, таких как распознавание жестов мышью, интерактивных элементов на диаграмме позволило существенно улучшить показатели удобства использования среды TRIK Studio.

Генеральный директор
ООО "Кибернетические Технологии" Р.М. Лучин

ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ "КиберТех" "CyberTech" Co. Ltd. САНКТ-ПЕТЕРБУРГ

ЗАО «Ланит-Терком»
198504, Санкт-Петербург, Ст. Петергоф, Университетский пр.д.28
Тел. (812) 428-71-09 факс: (812)428-70-29
ИНН 7819019624, КПП 781901001, Р/СЧЕТ 40702810222000001890 ПАО "БАНК"САНКТ-ПЕТЕРБУРГ", КОР/СЧЕТ 30101810900000000790, БИК 044030790, ОКПО 48004088, ОКОНХ 95120, ОКВЭД 73.10.00 , ОКАТО 1140413000, ОГРН 1037841001681

Исх. № 124 от 08.12.2015 г.

АКТ

о внедрении результатов диссертационного исследования Брыксина Т.А.
на тему "**Платформа для создания специализированных визуальных сред разработки программного обеспечения**"
на соискание учёной степени кандидата технических наук

Результаты диссертационного исследования Брыксина Тимофея Александровича на тему "Платформа для создания специализированных визуальных сред разработки программного обеспечения" применялись в ЗАО "ЛАНИТ-Терком" для разработки системы управления на основе компьютерного зрения. На базе платформы QReal, разработанной в ходе диссертационного исследования Т.А. Брыксина, был создан предметно-ориентированный язык для описания жестов пользователя, распознаваемых системой компьютерного зрения, по диаграммам на котором был сгенерирован код распознавателя жестов. Наличие платформы QReal позволило, во-первых, существенно сократить трудозатраты на разработку решения, во-вторых, уменьшить количество ошибок по сравнению с ручным кодированием – то, что код на C++ генерируется по визуальной модели, а не пишется вручную, исключает возможность опечаток.

Генеральный директор
ЗАО «Ланит – Терком» **А. Н. Терехов**

Директор департамента
видеотехнологий **М.Н. Смирнов**