Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Разработка метамоделирования “на лету” в metaCASE-системе QReal

Курсовая работа студентки 445 группы
Птахиной Алины Ивановны

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

Санкт-Петербург
2013

Оглавление

Введение

Причиной возникновения визуального программирования [1] стало желание упростить процесс разработки приложений и сделать его как можно более наглядным и удобным. При таком подходе программа представляется в виде набора диаграмм, каждая из которых представляет отдельный аспект ПО. Диаграммы формально не связаны друг с другом. Это позволяет моделировать ПО с разных точек зрения и при разработке не рассматривать все многообразие предметной области, а сосредоточиться на конкретном ее участке, функциональности. Такой подход широко применяется в случае командной разработки проекта и позволяет достичь взаимопонимания между различными участниками проекта.

Для реализации такого подхода применяются специализированные программно-технологические средства, называемые CASE-системами. Термин CASE (Computer Aided Software Engineering) появился в индустрии разработки ПО в начале 1980-х годов. CASE-системы предоставляют средства генерации кода и используются в качестве инструментов для разработки, анализа и проектирования программного обеспечения.

Часто возникает потребность в создании множества моделей конкретной предметной области. Соответственно возникает потребность в специальном языке, который существенно упростил бы разработку языков, позволяющих описывать такие модели. Этот язык должен содержать описание всех абстракций, которые нужны при моделировании. Модель языка моделирования называется метамоделью.

Существуют системы, которые по формальному описанию CASE-систем позволяют их автоматически генерировать. Такие системы, упрощающие процесс создания CASE-систем, называются metaCASE-системами. MetaCASE-системы применяются для создания предметно-ориентированных языков (DSL, Domain Specific Language), с помощью которых можно решать конкретные, специфичные для данной предметной области, задачи. Их преимуществом перед универсальными языками общего назначения является то, что их использование может привести к более быстрому решению.

В данной работе рассматривается metaCASE-система QReal - проект научно-исследовательской группы кафедры системного программирования СПбГУ. Система QReal позволяет автоматически генерировать код произвольных визуальных редакторов по описаниям их метамоделей. Таким образом, с помощью метамоделирования мы получаем быструю реализацию предметно-ориентированного языка.

1 Постановка задачи

Часто на практике возникают ситуации, когда для решения задачи в рамках данной предметной области приходится описывать сложные, нетривиальные структуры. А с фиксированными наборами сущностей и диаграмм, предоставляемыми большинством визуальных редакторов, это бывает затруднительно: пользователю приходится думать не в терминах предметной области, а в терминах конкретной диаграммы. Это приводит к тому, что пользователь сначала пытается нарисовать эскиз на простом листе бумаги, а потом думает, как его полученное решение перенести в визуальный редактор. И, как правило, этот процесс длится долго: нужных элементов языка нет, а добавление новых и изменение существующих затруднительно.

Безусловно, свести работу с CASE-средством до уровня бумаги и ручки не получится, однако хочется предоставить пользователю большую свободу действий. Хочется, чтобы пользователь мог сам создать нужный для его предметной области язык без особых усилий. Ведь решение любой задачи проще всегда начинать с эскиза, с простых элементов, которые в дальнейшем можно уточнять и детализировать.

В связи с этим было предложено реализовать так называемое метамоделирование “на лету”, которое позволило бы быстро и легко расширять систему прямо в процессе разработки, а именно: добавлять новые элементы языка, удалять ненужные и изменять существующие. Реализация метамоделирования “на лету” в системе QReal стала возможной благодаря интерпретатору метамоделей, который был написан в рамках моей курсовой работы третьего курса [2]. С помощью интерпретатора появилась возможность изменять метамодель прямо в процессе моделирования.

Таким образом, задача заключается в следующем: в режиме интерпретируемой метамодели предоставить пользователю возможность:

- Изменять существующие свойства элементов (под этим понимается изменение типа свойства, его описания и имени), добавлять новые свойства элементу и удалять неиспользуемые.
- Изменять графическое представление элементов
- Создавать новые элементы языка и удалять ненужные с точки зрения пользователя элементы.
- Создать свой собственный язык “с нуля”

При этом, все новые и измененные сущности должны быть сразу доступны для построения диаграммы, и в случае возможных конфликтов и некорректности системы пользователю должна быть предоставлена информация о возможных последствиях.

2 Обзор существующих решений

Рассмотрим наиболее распространенные CASE-средства и языковые инструментарии, основанные на технологии предметно-ориентированного моделирования (DSM, Domain-Specific Modeling) и использующие интерпретативный подход к созданию редакторов, с точки зрения удобства внесения изменений в метамодель. Основными аналогами являются системы MetaEdit+ и MetaLanguage. Подробнее о них можно почитать в дипломной работе Е.И.Такун “ Реализация режима быстрого прототипирования в CASE-системе QReal” [4], в статье А.О.Сухова “Сравнение систем разработки визуальных предметно-ориентированных языков” [5] и в моей курсовой работе третьего курса “Интерпретация метамоделей в metaCASE-системе QReal” [2]. Здесь же кратко приведем их основные особенности.

2.1 MetaEdit+

MetaEdit+ - это среда для создания и использования предметно-ориентированных языков, разработанная в рамках научно-исследовательского проекта MetaPHOR в University of Jyväskylä в качестве расширения среды MetaEdit, созданной ранее проектом SYTI в сотрудничестве с финской компанией MetaCase в период с конца 1980-х – начала 1990-х годов. Среда MetaEdit поддерживала создание только одного модельного языка для одного пользователя в текущий момент и обеспечивала моделирование лишь диаграмм. MetaEdit+ предоставляет несколько встроенных языков моделирования, обеспечивает одновременный доступ к среде для нескольких пользователей, а также позволяет создавать диаграммы, матрицы и таблицы[6].

В состав MetaEdit+ входит средство создания языков моделирования и генераторов MetaEdit+ Workbench, которое предоставляет простой, но мощный язык метамоделирования и инструмент для проектирования модельного языка. С помощью него можно определять основные понятия языка (concepts), устанавливать на них ограничения и правила, задавать атрибуты, взаимосвязи между понятиями и различные правила интеграции нескольких языков в одной системе. Графическое представление понятий языка можно задать, воспользовавшись встроенным в систему редактором Symbol Editor, который позволяет как создать собственное изображение, так и загрузить уже существующее в формате SVG или BMP. Далее полученные в MetaEdit+ Workbench модельные языки можно использовать для создания и редактирования моделей во встроенном в MetaEdit+ инструменте MetaEdit+ Modeler. Более подробно данный процесс проиллюстрирован на рис 1.

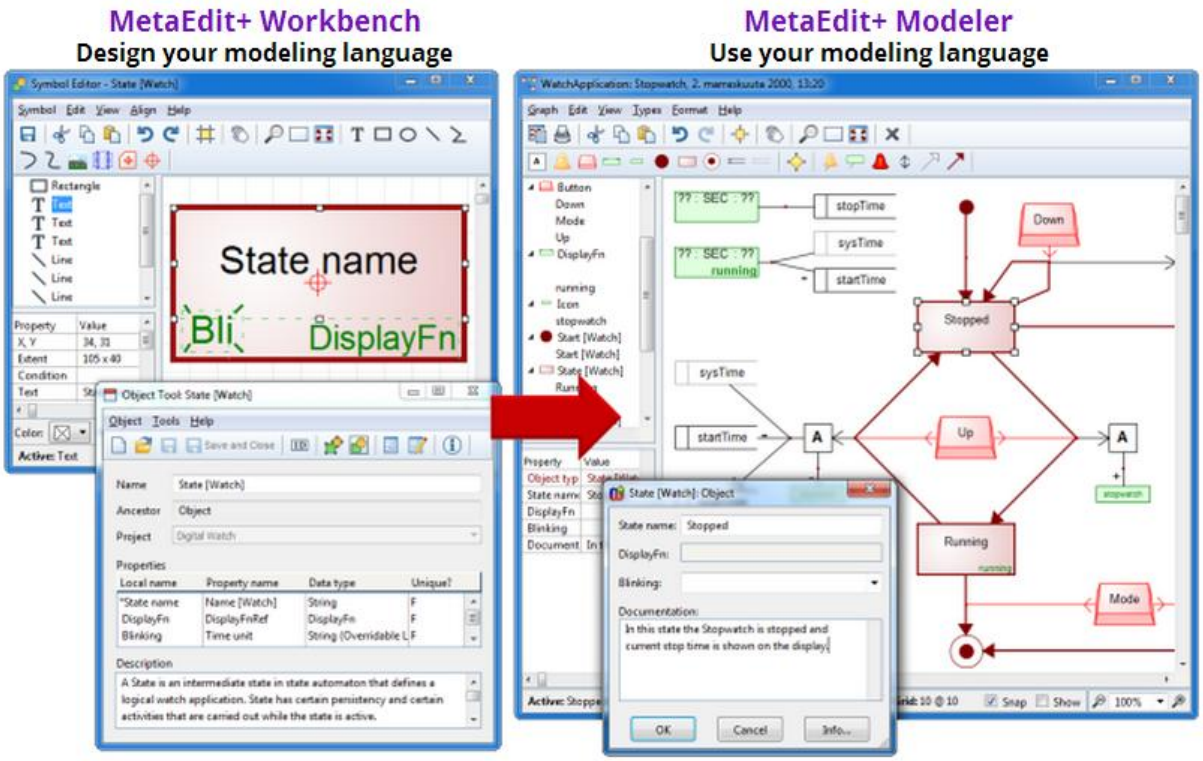

Рис 1. MetaEdit+ Domain-Specific Modeling (DSM) environment

Вся необходимая информация о модельных языках и моделях, включающая все их элементы и свойства, хранится в репозитории. Она используется для генерации кода DSL-редактора и документации, содержащей название модели, ее графическое представление со ссылками на описание отдельных ее частей и информацию о разработчиках.

При построении метамодели в MetaEdit+ используется язык метамоделирования GOPPRR (Graph-Object-Property-Port-Relationship-Role) [3], получивший свое название от основных понятий, которыми он оперирует:

- граф (Graph) – создаваемый язык моделирования (совокупность объектов и связей);
- объект (Object) – сущность модельного языка;
- свойство (Property) – атрибут какого-либо объекта графа;
- порт (Port) - место на объекте, куда можно прицепить роль;
- связь (Relation) – задает отношение между объектами;
- роль(Role) – определяет, каким образом объект участвует в связи.

С построения GOPRR-модели и начинается создание метамодели. Далее она может быть открыта в MetaEdit+ Workbench для задания семантики разрабатываемого языка моделирования.

MetaEdit+ использует подход, основанный на интерпретации метамоделей. Это позволяет вносить изменения в метамодель во время работы системы без перезапуска

приложения. Для этого пользователю достаточно внести изменения в описание метамодели, вновь импортировать ее в MetaEdit+ Workbench и продолжить работу над моделью. А все необходимые изменения в соответствующих моделях DSM-платформа произведет сама.

Такое динамическое изменение метамоделей близко к тому, что мы хотим реализовать. Однако при таком подходе пользователь вынужден работать отдельно с моделью и метамоделью. Задачей метамоделирования “на лету” же является объединение редактирования модели и метамодели.

2.2 MetaLanguage

MetaLanguage представляет собой инструментальную среду разработки визуальных языков моделирования. Впервые данная система была упомянута в 2008 году в межвузовском сборнике научных статей “Математика программных систем” в статье Л.Н.Лядовой и А.О.Сухова “Языковой инструментарий системы MetaLanguage”[7], в которой описывались базовые элементы метаязыка, и статье А.О.Сухова под названием “Среда разработки визуальных предметно-ориентированных языков моделирования”[8]. На основе этих статей и рассмотрим данную среду, поскольку она отсутствует в открытом доступе.

MetaLanguage использует интерпретативный подход для описания моделей различных уровней абстракции. А сам процесс создания нового предметно-ориентированного языка представляется схемой, изображенной на рис 2. Он начинается с построения метамодели: задаются основные конструкции языка, соотношения между ними и накладываются необходимые ограничения. Далее можно переходить непосредственно к построению модели. После описания пользователем сущностей предметной области и установления связей между ними, построенная модель проходит валидацию. В случае невыполнения каких-либо ограничений, пользователю будет предоставлена соответствующая информация. Полученную модель можно сохранить в виде XML-файла, содержащего все основные данные о модели: свойства, сущности, атрибуты и накладываемые ограничения. Помимо этого, можно сгенерировать на основе модели документацию с информацией о разработчиках, названии модели и ее графическом представлении.

Модификация метамодели может быть произведена на любом этапе создания визуального языка. А все внесенные изменения в метамодель система автоматически применит и к моделям, созданным с помощью данной метамодели.

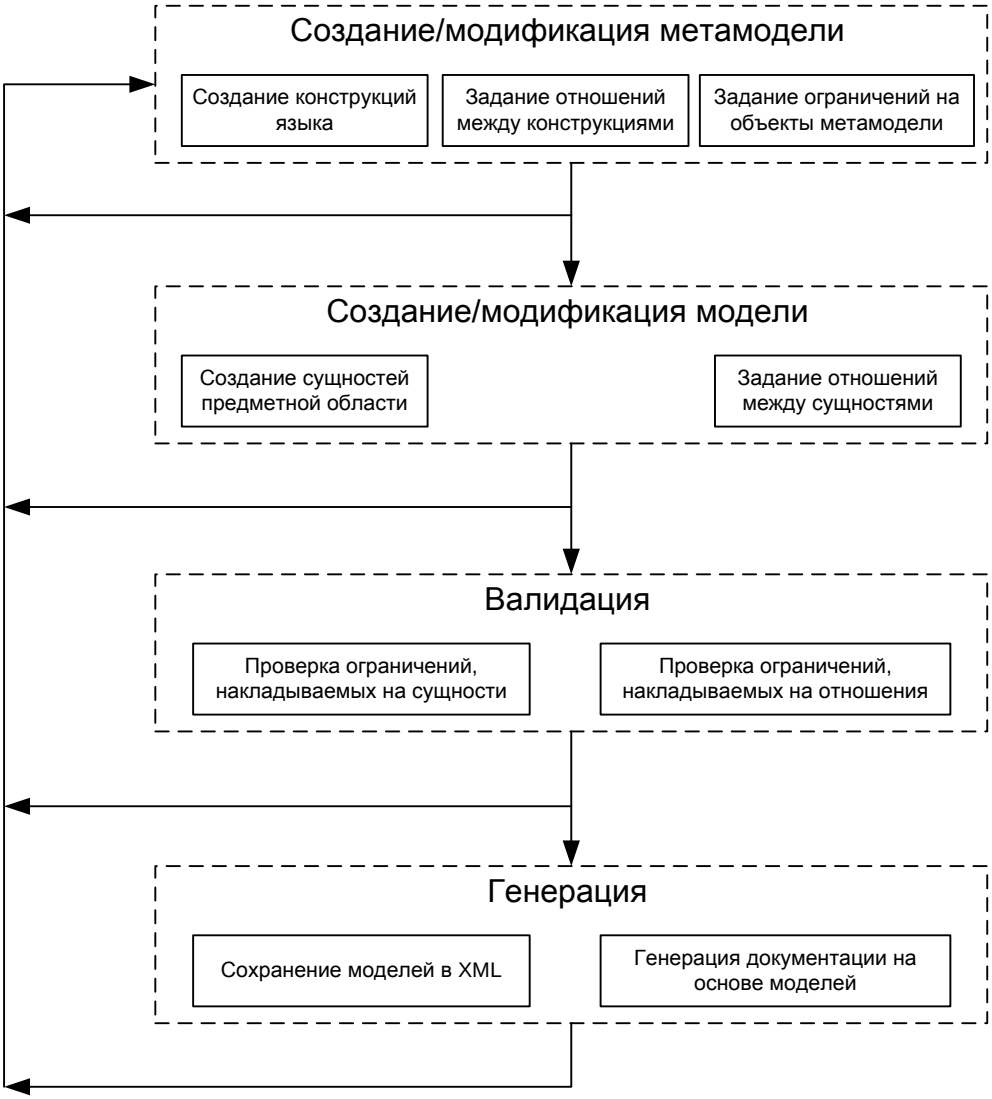

Рис 2. Процесс создания/модификации языка предметной области с помощью системы MetaLanguage

Из этого обзора можно сделать вывод, что процесс создания предметно-ориентированного языка в системе MetaLanguage очень похож на тот, который описывался в системе MetaEdit+.

2.3 QReal

В QReal метамоделированием “на лету” в рамках дипломной работы занималась Е.И.Такун. Предлагаемое решение изображено на рис 3. Как мы можем видеть, пользователю предоставлялась возможность в несколько кликов создать свой собственный элемент, изменить его путем редактирования его свойств, имени и смены графического изображения. Также была возможность создать копию элемента и удалить ненужный элемент из палитры.

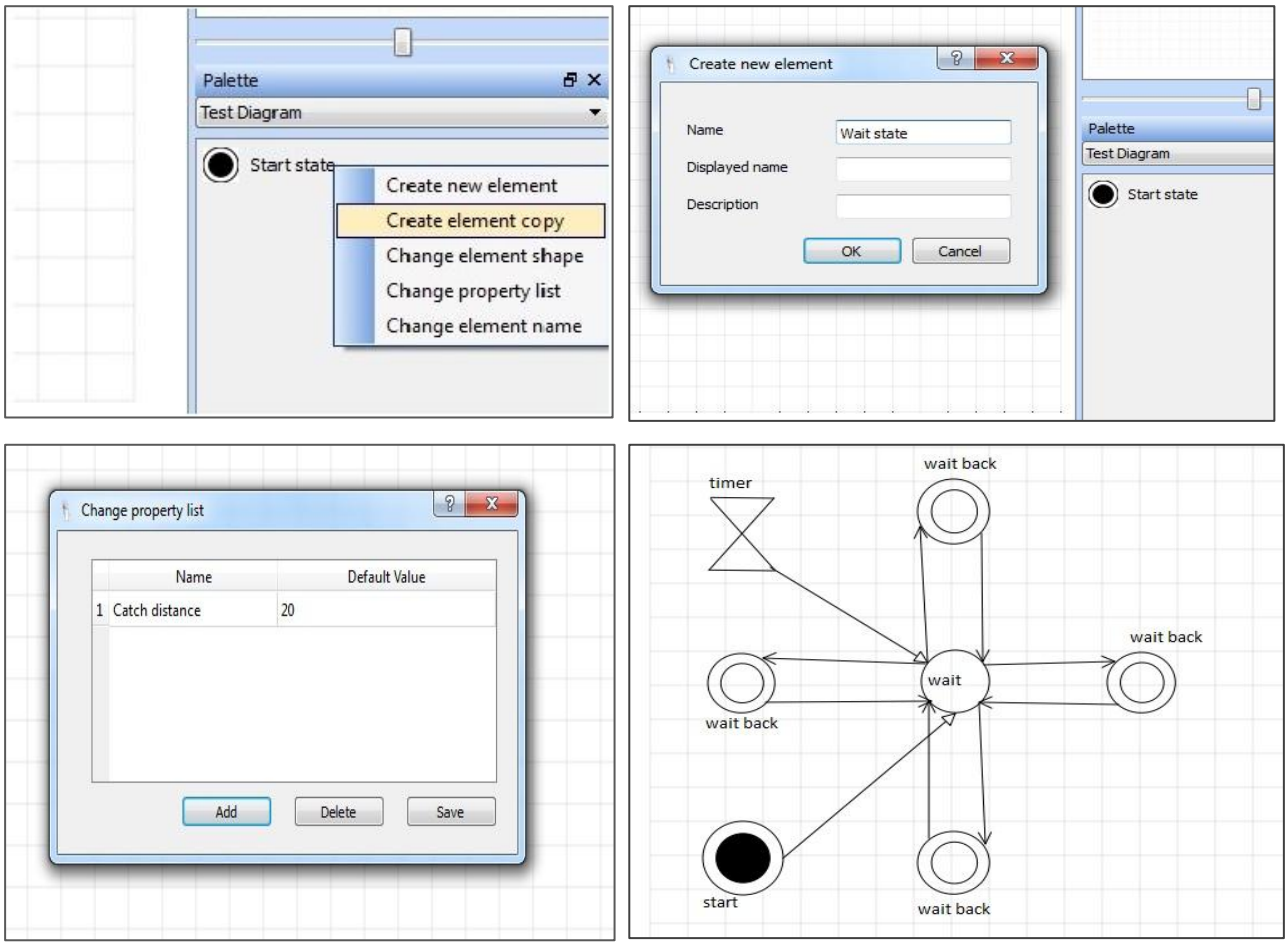

Рис 3. Процесс создания предметно-ориентированного языка

Однако в QReal данная реализация встроена не была, а в настоящий момент встраивание невозможно по причине многочисленных изменений, произошедших в системе. В связи с этим было предложено реализовать данную концепцию с нуля.

3 Архитектура

Визуальные языки в современных DSM-платформах, как правило, задаются с помощью метамоделей, то есть моделей синтаксиса языка. Они описывают, какие элементы могут находиться на диаграмме, какие свойства есть у этих элементов, и как элементы могут быть связаны друг с другом. Существует два принципиально разных подхода по работе с метамоделью. Один из них заключается в том, что описание языка может быть сгенерировано как код на текстовом языке программирования по метамодели. Второй - метамодель может интерпретироваться во время работы DSM-платформы. Основное отличие интерпретативного подхода от генеративного заключается в том, что интерпретация моделей происходит непосредственно во время работы приложения, в то время как генерация обязана завершиться до начала работы.

Рассмотрим, в чем заключается отличие данных подходов с точки зрения архитектуры в системе QReal (см рис 4). В случае генеративного подхода GUI для отображения информации, касающейся языка, такой как: имеющиеся сущности, их свойства и их графическое представление, использует класс EditorManager. EditorManager содержит в себе набор редакторов, каждый из которых реализует интерфейс EditorInterface. Реализация редакторов генерируется во время компиляции приложения с помощью xml-парсера из файла метамодели в код на C++ - плагины по описанию метамоделей. Во время запуска приложения библиотечные файлы плагинов используются для загрузки редакторов.

При реализации интерпретатора в системе QReal из класса EditorManager был выделен интерфейс EditorManagerInterface, за реализацию которого отвечают класс EditorManager и созданный класс интерпретатора. Он работает по несколько иному принципу. В отличие от генеративного подхода, вся необходимая информация извлекается не посредством вызова сгенерированного метода, а путем непосредственного обращения к репозиторию с сохраненной метамоделью. Класс InterpreterEditorManager отвечает за интерпретацию метамодели и тем самым эмулирует функциональность сгенерированного редактора.

Для того, чтобы поддержать возможность использования обоих режимов, был реализован шаблон проектирования Proxy. В зависимости от выбора пользователем соответствующего режима EditorManagerProxy передает управление либо классу EditorManager, либо InterpreterEditorManager.

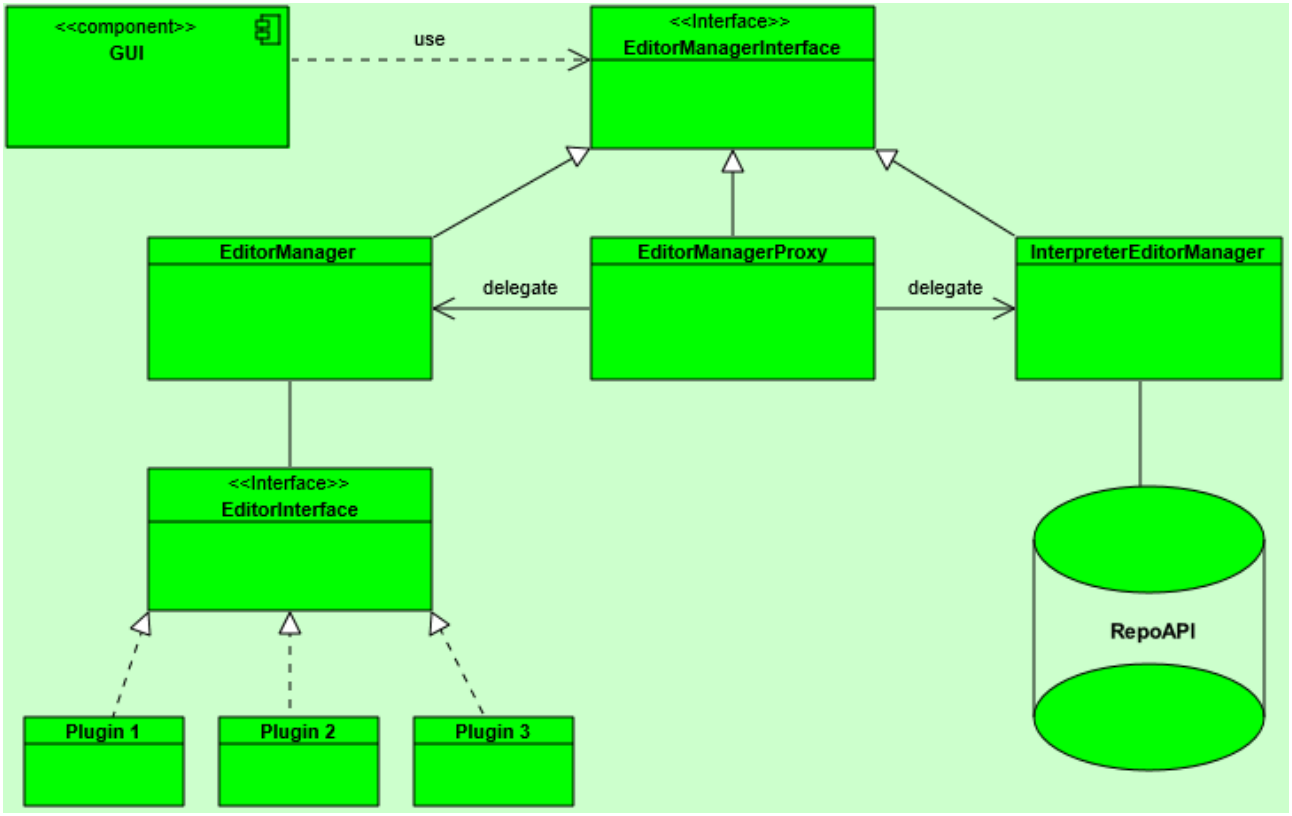

Рис 4. Архитектура. Генеративный и интерпретативный подходы.

Сложно сказать, какой из данных режимов предпочтительнее использовать. Все зависит от поставленной задачи. Если необходимо быстро набросать эскиз, не вдаваясь в подробности (не требуется указывать связи наследования и другие отношения между элементами), метамоделирование на «лету» будет удобнее и быстрее для пользователя. Однако задать в таком случае более сложное поведение элементов нельзя.

С другой стороны, отношение вложенности (свойство элемента быть вложенным в другой элемент) очень удобно использовать при построении диаграммы. Поэтому было принято решение поддержать данную функциональность, Для этого при создании интерпретируемой диаграммы в нее будет добавляться абстрактный элемент, который будет содержать сам себя, и от которого будут наследоваться все создаваемые в дальнейшем сущности. Таким образом, все элементы являются по сути контейнерами и могут содержать друг друга.

4 Реализация

3.1 Режим интерпретируемой диаграммы

Как упоминалось ранее, реализация метамоделирования “на лету” возможна только в режиме интерпретируемой метамодели. Чтобы в него перейти, достаточно при запуске системы QReal выбрать в стартовом диалоге пункт меню “Открыть интерпретируемую диаграмму” (см. рис 5) и загрузить файл с сохраненной метамоделью. Метамодель хранится в формате сохранений QReal, который представляет собой сжатые наборы XML-файлов. Метамодели в системе QReal можно создавать и изменять при помощи встроенного метаредактора. Он представляет собой удобный инструментарий, позволяющий задавать сущности языка и устанавливать между ними отношения.

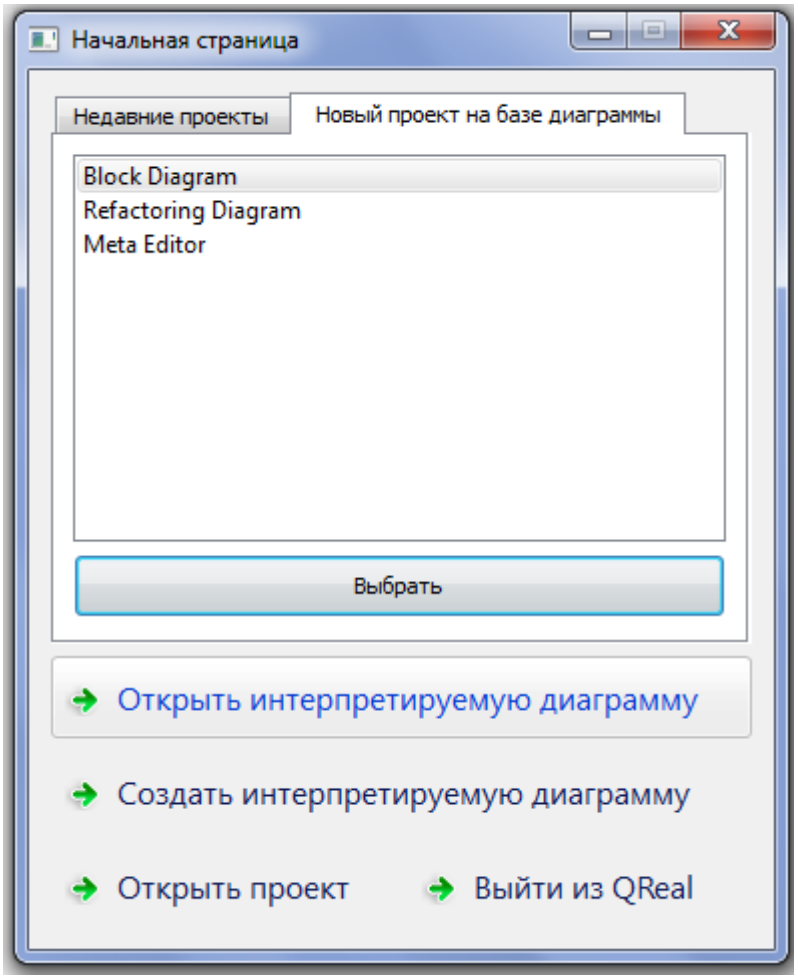

Рис 5. Стартовый диалог QReal

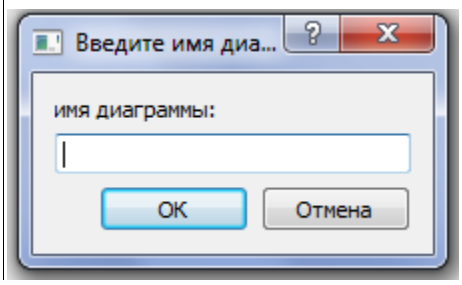

Рис 6. Окно ввода

В случае, если не имеется сохраненной метамодели, или хочется создать свой собственный визуальный язык, можно воспользоваться пунктом меню “Создать интерпретируемую диаграмму”. В открывшемся окне ввода (см рис 6.) необходимо ввести имя создаваемой диаграммы. После этого будет создана диаграмма и редактор с одноименным названием. На данный момент диаграмма не содержит больше никакой

информации. Палитра элементов и обозреватели логической и графической модели, а соответственно и редактор свойств QReal, будут пусты, поскольку они пока не содержат элементов. Далее можно переходить к созданию новых элементов диаграммы.

Полученную с помощью метамоделирования «на лету» метамодель можно сохранить и далее уточнять и детализировать в метаредакторе. В нем можно задать более сложные отношения между элементами: наследование, связи и пр. После внесения изменений в метамодель ее можно снова использовать в режиме метамоделирования «на лету».

3.2 Добавление нового элемента на диаграмму

Для добавления нового элемента на диаграмму необходимо кликнуть правой кнопкой мыши по палитре и в появившемся меню выбрать пункт “Добавить элемент” (см рис 7).

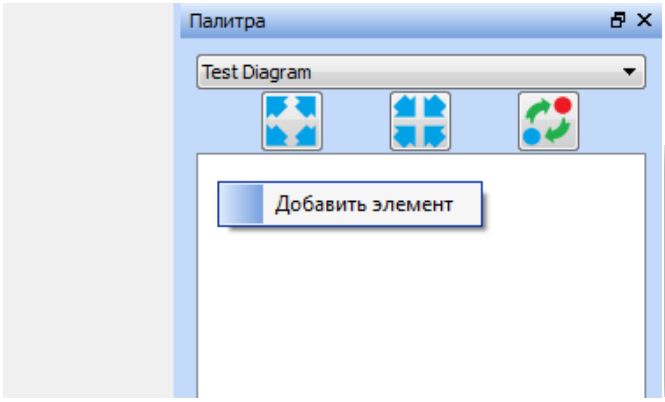

Рис 7. Добавление нового элемента

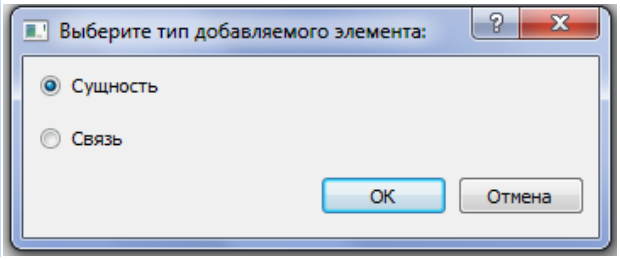

Рис 8. Выбор типа элемента

Затем надо выбрать тип добавляемого элемента: сущность или связь. И в соответствии с этим задать необходимые свойства создаваемому элементу в окне, представленном на рис 9 или рис 10. После этого, в случае заполнения всех обязательных полей, пользователь увидит созданный им элемент в палитре. Вот и все: созданный элемент готов к использованию. Для простоты все создаваемые сущности имеют изначально форму квадрата. Далее в данной работе будет описано, как можно будет изменить это графическое представление. Тип связи же задается в ее свойствах.

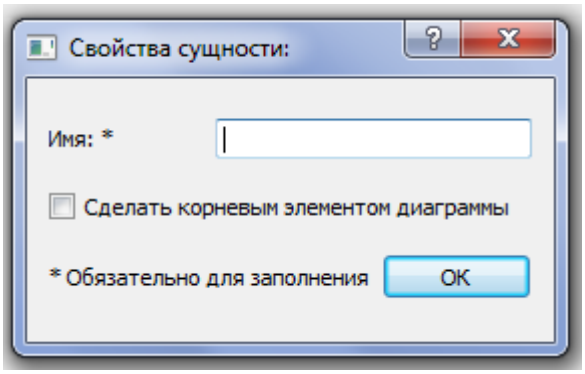

Рис 9. Свойства сущности

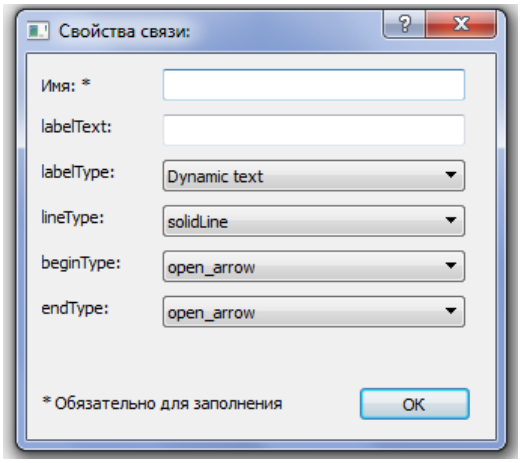

Рис 10. Свойства связи

Так выглядит процесс создания нового элемента с точки зрения пользователя. Рассмотрим подробнее, какие изменения при этом происходят в метамодели. Вся информация о метамодели хранится в отдельном репозитории. Работу с ним осуществляет класс интерпретатора. Он позволяет извлекать из репозитория все необходимые данные. В нашем же случае класс интерпретатора необходимо расширить функциями добавления и изменения информации. При создании нового элемента, в репозитории находится нужный редактор, из списка его диаграмм выбирается соответствующая диаграмма, и ей в качестве ребенка добавляется новый созданный элемент метамодели. Далее ему задается, в зависимости от того, сущность это или связь, соответствующий набор свойств.

3.3 Удаление имеющегося элемента

Удалить элемент можно, кликнув правой кнопкой мышки по нему и выбрав соответствующий пункт меню (см рис 11).

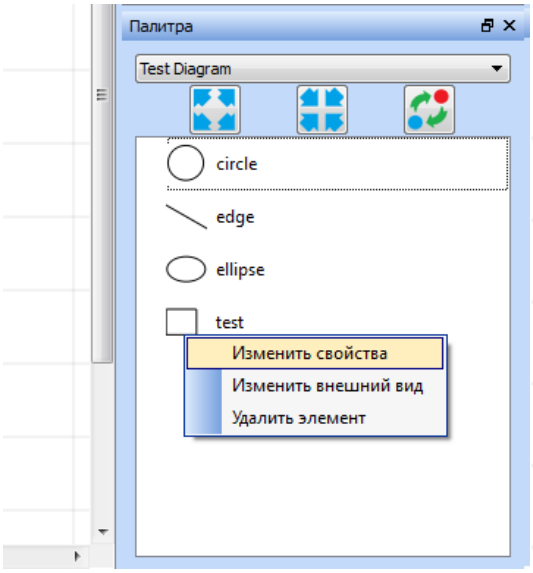

Рис 11. Меню элемента палитры

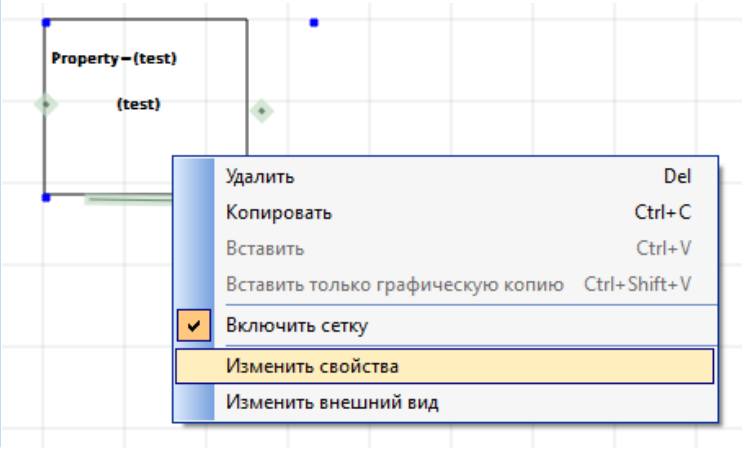

Рис 12: Меню элемента, находящегося на сцене

В случае удаления элемента, дело обстоит несколько более сложным образом. Необходимо удалить все его графические представления со сцены, из графического и логического обозревателей модели, а так же сам элемент из палитры. Кроме того, возможны ситуации, при которых удаление одного элемента затрагивает другие элементы диаграммы. В таких ситуациях пользователю перед удалением предоставляется информация, и он в любой момент может прервать процесс удаления. Рассмотрим эти ситуации подробнее:

1) Удаляемый элемент является корневым элементом диаграммы. В таком случае удаление элемента приведет к удалению всей построенной диаграммы.
2) Удаляемый элемент имеет наследников. В этой ситуации было принято решение, что все унаследованные свойства будут удалены у его наследников. Однако пользователю будут

предоставлено предупреждение, в котором будут перечислены все фигуры-наследники, и пользователь сможет при желании отменить операцию удаления.

Рассмотрим подробнее, как происходит удаление информации об элементе из метамодели. Система QReal устроена таким образом, что удаление элемента из обозревателя логической модели приводит к автоматическому удалению элемента из обозревателя графической модели и при этом удалению элемента со сцены. Таким образом, первым шагом необходимо удалить элемент из обозревателя логической модели.

Чтобы удалить информацию об элементе из репозитория, для начала надо по объекту модели найти для него соответствующий элемент метамодели. В качестве следующего шага нужно получить список всех связей, как входящих, так и выходящих, которые присутствуют в метамодели у данного элемента. Например, это могут быть связи наследования, говорящие в зависимости от направления о том, имеются ли у элемента наследники, или он сам является наследником какого-то элемента. Или это могут быть связи вложенности, характеризующие тот факт, что элемент может быть вложен внутрь другого элемента, а также то, какие элементы могут вкладываться в него самого. Все эти связи должны быть удалены из репозитория, чтобы не допустить возможность ссылки на удаленный из метамодели объект. Далее нужно удалить элемент из списка детей диаграммы, и только после этого можно удалить сам элемент из репозитория.

Что касается удаления элемента из палитры, то для этого можно просто обновить ее содержимое. Поскольку мы находимся в режиме интерпретируемой метамодели, а метамодель после проделанных нами действий выше ничего уже не знает о данном элементе, то информация о нем не будет добавлена в палитру.

3.4 Редактирование свойств существующего элемента

Под редактированием свойств понимается возможность добавлять новые свойства элементу, изменять существующие и удалять ненужные свойства. Рассмотрим подробнее, каким образом происходит каждое перечисленное выше действие.

Все возможности видоизменения свойств становятся доступными для пользователя после выбора им пункта меню “Изменить свойства”, изображенного на рисунке 11. Если элемент присутствует на сцене, то можно также кликнуть по нему правой кнопкой мыши и выбрать идентичный пункт из предложенного меню (см рис 12). После этого появляется окно со списком всех свойств, которые есть у выбранного элемента на данный момент (см рис 13).

Как ни странно, в связи с особенностью архитектуры системы QReal, свойство у элемента проще удалить, чем добавить новое. Это связано с тем, что, если элемент присутствует на сцене, то у него не будет установлено свойство, которое мы только что

добавили, он ничего про него не знает. Запрос созданного свойства у элемента диаграммы приведет к возникновению ошибки в системе. В связи с этим было принято решение запретить пользователю добавлять свойства элементу в случае, если хотя бы одно его представление присутствует на сцене, либо в обозревателях графической или логической модели. Однако добавлять свойства и изменять существующие в такой ситуации, конечно, можно.

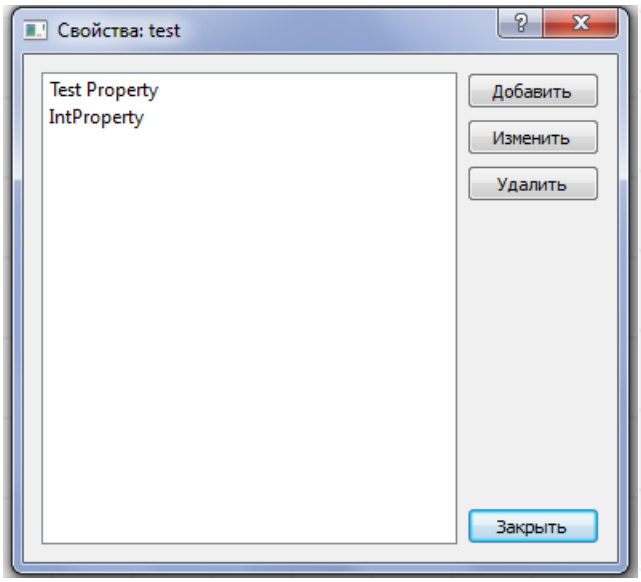

Рис 13. Список свойств элемента

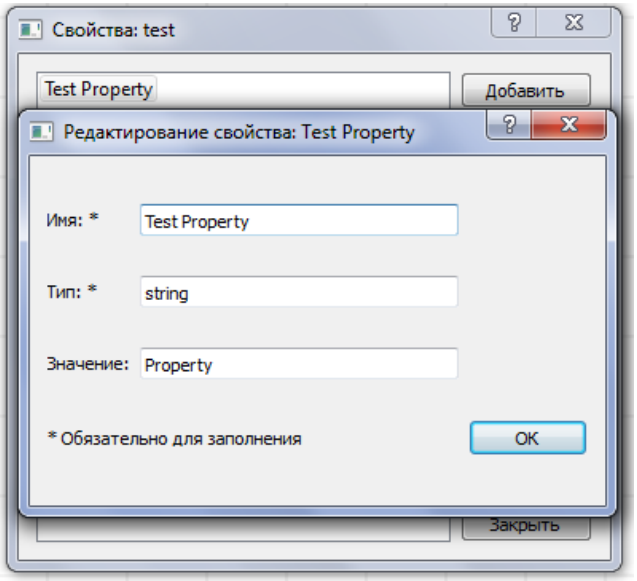

Рис 14. Редактирование выбранного свойства

При добавлении нового свойства, а также при редактировании существующего пользователю предлагается заполнить форму, изображенную на рисунке 14. Обязательными для заполнения являются поля, содержащие имя и тип свойства. Значение по умолчанию, в случае не указания пользователем иного, будет считаться нулевым, либо пустой строкой. В случае если элемент, которому добавляется новое свойство, имеет потомков, то данное свойство будет автоматически добавлено и всем наследникам данного элемента. То же произойдет и в случае удаления свойства родителя.

Отдельно следует рассмотреть операцию изменения типа элемента. Она требует особого внимания, поскольку не всегда возможны корректные преобразования значений одного типа в другой. В качестве примера рассмотрим приведение значения типа string к значению типа int. Следует осторожно относиться к такого рода преобразованиям, но чтобы все-таки позволить пользователю изменять тип свойства, было принято решение, что в случае некорректного преобразования значение по умолчанию просто обнулится. А соответствующая информация о возможных последствиях будет доведена до пользователя в качестве предупреждения.

3.5 Изменение графического изображения элемента

Изначально все создаваемые пользователем элементы имеют форму квадрата. Для того, чтобы изменить изображение элемента, нужно в меню элемента выбрать пункт “Изменить внешний вид”. После этого в открывшемся встроенном в QReal редакторе фигур можно изменить форму элемента. У пользователя есть возможность как загрузить существующий рисунок, сохраненный в графическом формате SVG, так и создать свой собственный с помощью имеющегося инструментария. При завершении редактирования необходимо нажать кнопку «save» для сохранения полученного результата. Внешний вид элемента будет изменен автоматически. Новое изображение будет применено как ко всем представителям данного элемента на сцене, так и к его иконкам в палитре и в обозревателе графической и логической модели.

5 Апробация

В качестве апробации метамоделирования “на лету” рассмотрим, как языки визуального моделирования применяются в разработке аппаратного обеспечения. Создадим визуальный язык для HasCol [9][10]. В HaSCol для моделирования аппаратных систем используется понятие процесс - сущность, инкапсулирующая в себе ресурсы (данные), обработчики сигналов и другие процессы, и имеющая входы и выходы (порты). Процесс имеет тип – перечень его входов и выходов с указанием типов их аргументов. Процессы могут наследоваться друг от друга.

Проведем апробацию метамоделирования “на лету” на задаче об арбитре динамических приоритетов 4 в 1. Сама задача формулируется следующим образом: “На один из четырех входов поступают данные, первый параметр пришедших данных – приоритет. Если в одном такте данные поступили на несколько входов, на выход выдаются данные с наибольшим приоритетом, остальные входы объявляются неготовыми. Если приходит только одно сообщение, оно отправляется на выход”.

Решать поставленную задачу будем в соответствии с оригинальным решением: сначала напишем арбитр динамических приоритетов 2 в 1 (с двумя входами и одним выходом), затем создадим арбитр-функтор 4 в 1, использующий 3 арбитра 2 в 1, который и решит задачу. Для нас принципиально то, что мы взяли известную задачу, с существующим уже решением и не придумываем нового, а просто реализуем известное при помощи метамоделирования “на лету”. Анализируем сущности, необходимые для получения решения и строим их, так сказать, “по ходу дела”, прямо в процессе разработки.

Перенесем понятия языка HaSCol в визуальный язык, добавив сущности “процесс” и “порт”. И начнем рисовать диаграмму типов процессов (см. рис 15). В ходе рисования мы понимаем, что нам нужно выразить наследование процессов, кроме того оказывается удобным понятие языка HaSCol “функтор”, т.е. процесс, параметризованный другим процессом. Добавим эти понятия в язык. Для того, чтобы получить полноценную работающую программу, надо добавить реализацию процессам. Поскольку мы пока не хотим создавать отдельный язык для задания логики реализации, будем писать реализацию в текстовом виде, для чего добавим соответствующий элемент. В нем напишем код процесса DynamicArbiter на языке системы CoolKit, описывающий реализацию арбитра 2 в 1. Код из комментария при синтезе добавляется к сгенерированному описанию процесса, таким образом, мы получаем полностью специфицированный арбитр 2 в 1, который может быть использован как фактический параметр функтора Arbiter4to1 (на это указывает отношение генерализации, связывающее DynamicArbiter и безымянный тип процесса – формальный

параметр функтора). Про процесс Arbiter4to1 мы указали пока только то, что он является функтором (готов использовать любой процесс, поддерживающий интерфейс арбитра 2 к 1, который мы определили с помощью безымянного типа процесса), и указали, что у него есть 4 входа и один выход – то есть просто “нарисовали” условие задачи. И “нарисовали” просто: смотрим, какой нужен элемент, создаем элемент в палитре с соответствующим названием и задаем ему нужное графическое изображение.

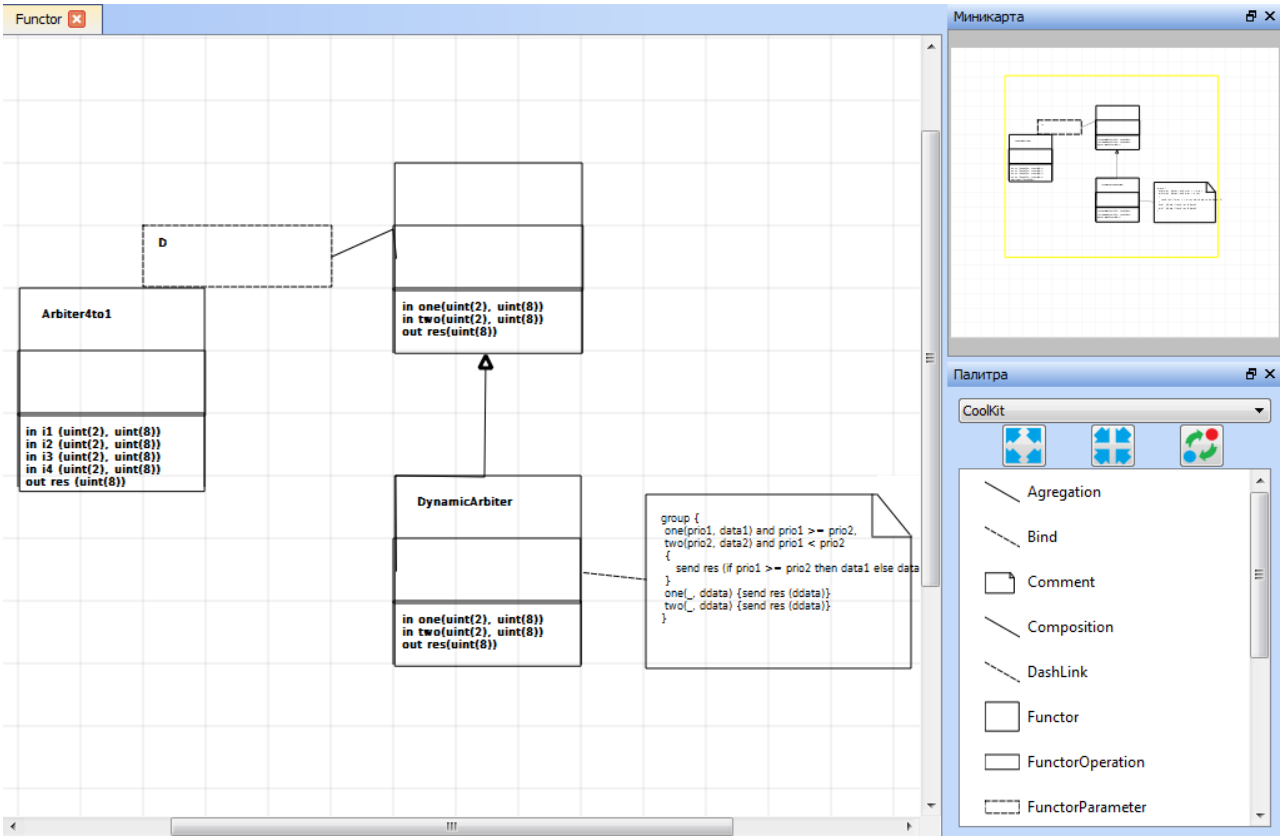

Рис 15. Диаграмма типов процессов для задачи "Арбитр 4 в 1”

Аналогичным образом нарисуем диаграмму отображения портов для задачи "Арбитр 4 в 1", указанную на рисунке 16. Здесь мы видим процесс верхнего уровня Arbiter4to1 – без имени, но с типом и с формальным параметром функтора. Arbiter4to1 имеет 4 входных порта и 1 выходной, их типы здесь уже можно не указывать, они были на диаграмме типов процессов. Процесс имеет три вложенных процесса A12, A34 и A1234 типа D, который, как следует из диаграммы типов процессов, описывает процессы, имеющие два входа и один выход. Если мы считаем, что в качестве параметра функтора передается арбитр 2 в 1, изображенное на рисунке соединение портов решает задачу.

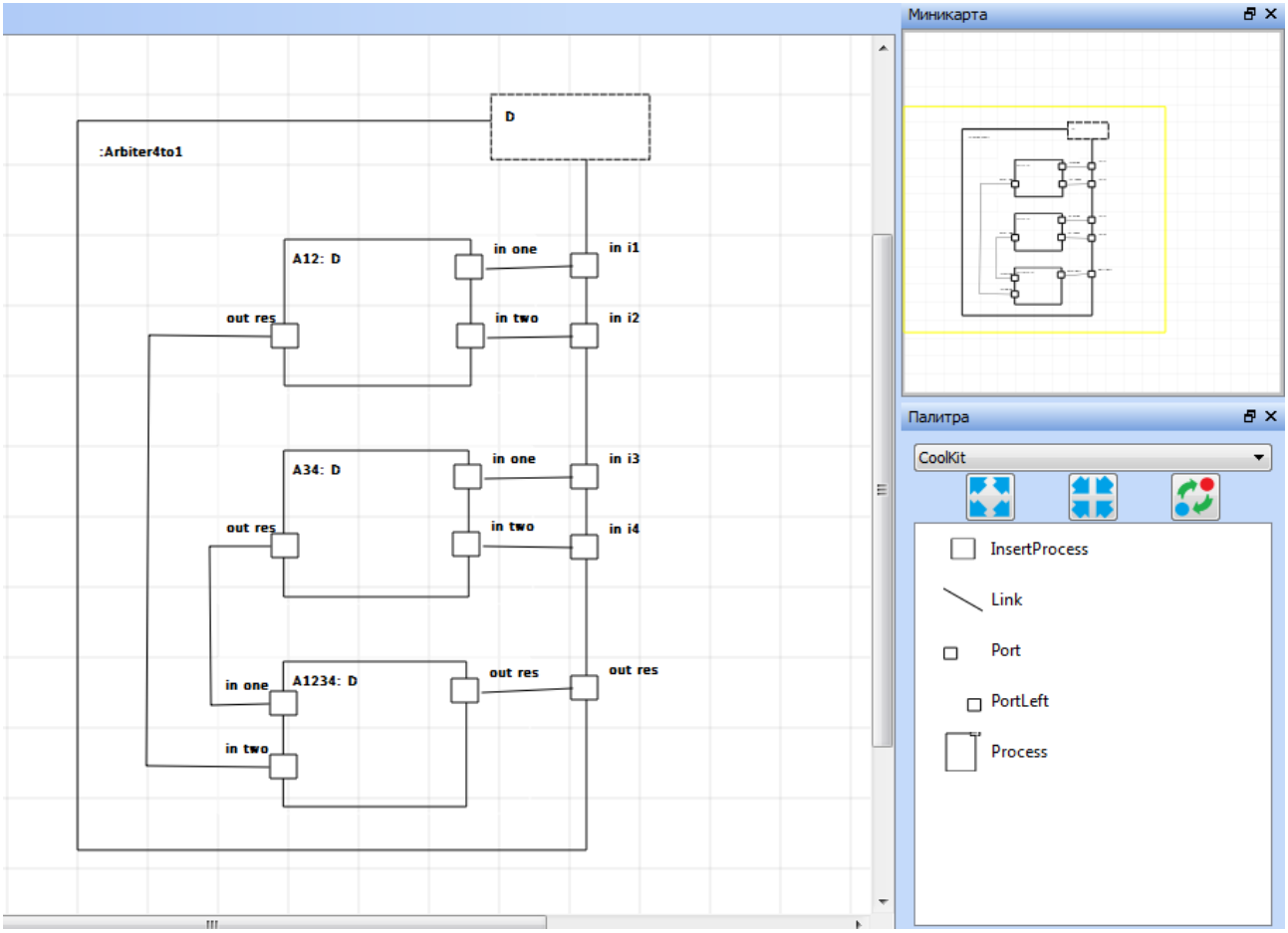

Рис 16: Диаграмма отображения портов для задачи "Арбитр 4 в 1”

В рассмотренной нами задаче логика поведения тривиальна, а чтобы продемонстрировать язык описания логики, рассмотрим другую задачу, посложнее. Процесс может принимать сообщения от других процессов или от самого себя. В языке HaSCoL есть два оператора посылки сообщения: send и inform. Рассмотрим, в чем заключается разница между ними. Пусть есть сообщение m. В случае inform m - доставка сообщения не гарантируется, ненадежная посылка. После того, как сообщение послано, исполнение посылающего процесса продолжается без всяких условий (т.е. даже если сообщение никто не получил). Посылка сообщения оператором send называется надежной, потому что сообщение никогда не пропадает, когда приемник будет готов к приему, сообщение будет доставлено.

Роль оператора получения сообщения играет условие, с которого начинается каждый обработчик процесса. Пример условия:

m(a) when a>Ø

Здесь из разъема m принимается сообщение (если оно там есть), и, если параметр a этого сообщения больше Ø, то условие возвращает значение истина, и начинается исполнение тела. Если сообщения не было, или параметр был меньше или равен Ø, то возвращается значение ложь, а исполнение тела приостанавливается.

Построим диаграмму поведения, описывающую аппаратную реализацию очереди FIFO с буфером из целых элементов и двумя указателями first – индекс в очереди первого кандидата на отправку и last – индекс в очереди последнего неотправленного сообщения (см. рис 17).

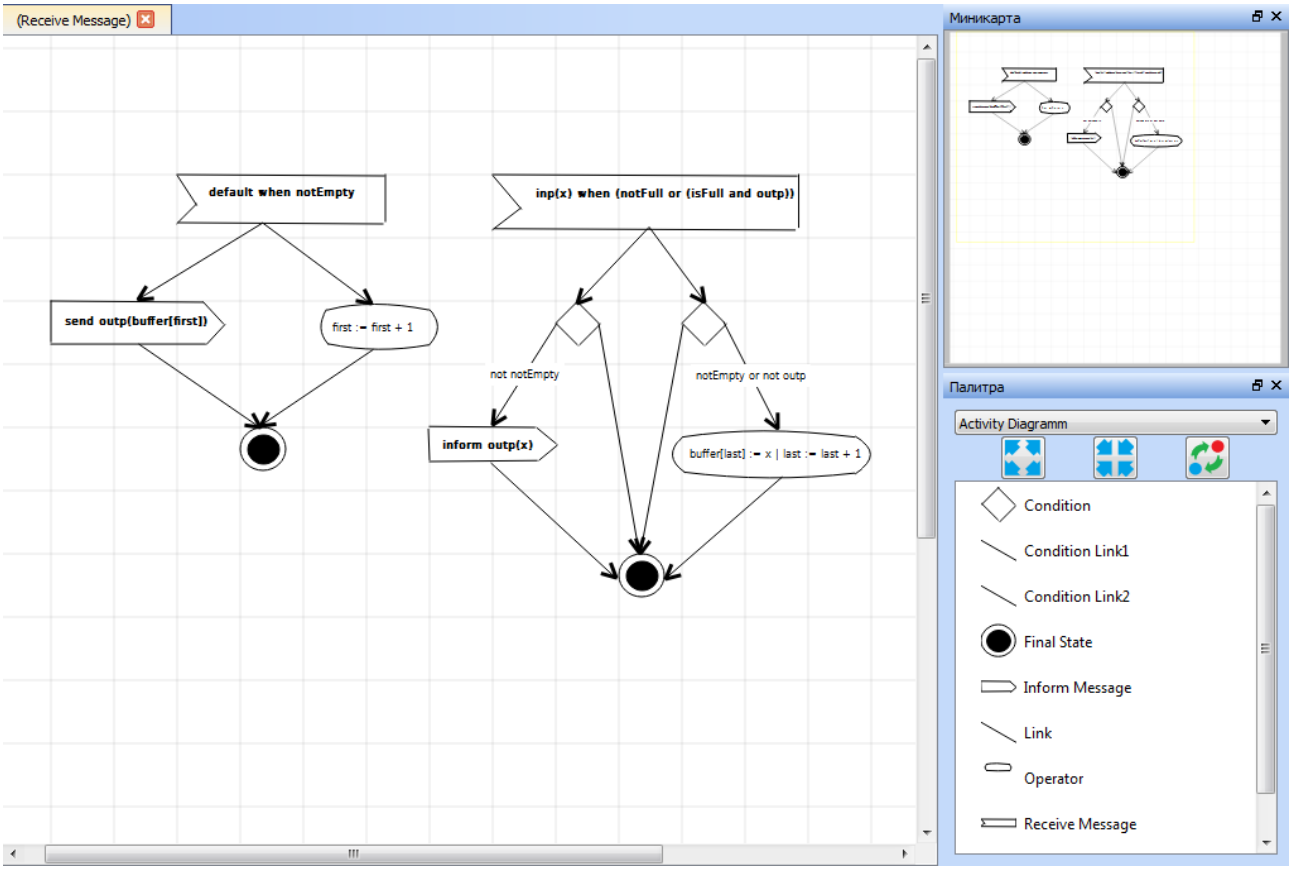

Рис 17. Диаграмма поведения.

Опишем два обработчика. Первый из них запускается каждый такт в случае, если очередь не пуста. Он посылает в выходной разъем первый элемент очереди и продвигает указатель `first`. Второй обработчик запускается, если во входном разъеме есть сообщение и в очереди есть место, куда его записать, или же, если очередь полна, но выходной порт готов принять сообщение. Последнее означает, что первый обработчик точно пошлет первое сообщение из очереди, освободив тем самым одну позицию, которую второй обработчик займет в конце такта. Тело второго обработчика состоит из двух параллельно исполняемых

операторов – в первом принятое сообщение сразу же передается на выход, если очередь пуста (в этом случае последнее принятое сообщение является и первым), а во втором – записывается в конец очереди.

Существенным является то, что во втором обработчике используется оператор inform вместо send. Это связано с тем, что от приостановки первого обработчика никто не пострадает: очередь будет принимать сообщения во втором обработчике, пока не переполнится. Но если бы мы во втором обработчике использовали оператора send, то любая задержка в нем заблокировала бы дальнейшее заполнение очереди. Поэтому вместо send использован неблокирующий оператор inform, а в конце добавлено условие or not outp. Если очередь была пуста, но входной разъем не был готов принять сообщение, то принятое из входного разъема сообщение могло бы потеряться, а с добавлением этого условия оно запишется в конец очереди по обычным правилам.

Заключение

В результате данной работы было реализовано метамоделирование “на лету”, предоставляющее пользователю возможность быстро и легко вносить изменения в визуальный язык программирования: добавлять новые элементы языка, удалять ненужные и изменять существующие. При этом все новые и измененные сущности сразу доступны для построения диаграммы, и в случае возможных конфликтов и некорректности системы пользователю предоставляется соответствующая информация о возможных последствиях.

В рамках данной концепции было произведено объединение редактирования модели и метамодели, что избавило пользователя от необходимости мыслить в терминах двух абстракций. Теперь уровень метамодели скрыт от пользователя, что позволяет ему полностью сконцентрироваться на задаче, а не на создании инструментария. Полученное решение было апробировано на реальной задаче.

Список литературы

1) Д.В.Кознов, Основы визуального моделирования. –М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория Знаний, 2008. – 246 с.: – (Серия “Основы информационных технологий”).
2) А.И.Птахина, Интерпретация метамоделей в metaCASE-системе QReal, курсовая работа, СПбГУ, кафедра системного программирования, 2012. URL: http://se.math.spbu.ru/SE/YearlyProjects/2012/list (дата обращения: 08.04.2013)
3) А.О.Сухов. Инструментальные средства создания визуальных предметно-ориентированных языков моделирования, Научный журнал ISSN 1812-7339 «Фундаментальные исследования», 2013. - № 4 часть 4. –С. 848-852.
4) Е.И.Такун, Реализация режима быстрого прототипирования в CASE-системе QReal, дипломная работа, СПбГУ, кафедра системного программирования, 2011 http://se.math.spbu.ru/SE/diploma/2011 (дата обращения: 08.04.2013)
5) А.О.Сухов. Сравнение систем разработки визуальных предметно-ориентированных языков / Математика программных систем: межвуз. сб. науч. ст. – Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 84–111.
6) Domain-Specific Modeling with MetaEdit+. URL: http://www.metacase.com/ (дата обращения: 08.04.2013)
7) Л.Н. Лядова, А.О. Сухов, Языковой инструментарий системы MetaLanguage, Математика программных систем: межвуз. сб. науч. ст. / М34 Перм. гос. ун-т. – Пермь, 2008. С. 40-51.
8) А.О. Сухов, Среда разработки визуальных предметно-ориентированных языков моделирования, Математика программных систем: межвуз. сб. науч. ст. / М34 Перм. гос. ун-т. – Пермь, 2008. С. 84-94.
9) Dmitri Boulytchev, Oleg Medvedev. Hardware Description Language Based on Message Passing and Implicit Pipelining // EWDTS '10 Proceedings of the 2010 East-West Design & Test Symposium. – 2010. – Pages 438-441.
10) Ю.В.Литвинов, Использование визуального моделирования для проектирования аппаратного обеспечения (не опубликовано).