Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Шигаров Никита Алексеевич

Создание интерпретатора OCL для глубокого метамоделирования в REAL.NET

Магистерская диссертация

Научный руководитель:
к.т.н., доц. Литвинов Ю. В.

Рецензент:
разработчик “СКБ Контур” Перешеина А. О.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering Department

Shigarov Nikita

Creation of OCL interpreter for deep metamodeling in REAL.NET

Graduation Thesis

Scientific supervisor:
docent Litvinov Yurii

Reviewer:
software engineer at Kontur Peresheina Anna

Saint-Petersburg
2019

Оглавление

Введение

Визуальные языки имеют большое значение в современном мире программирования. С их помощью можно более продуктивно и наглядно работать в предметной области, оперируя объектами.

Например, они служат хорошим инструментом для составления диаграмм различной сложности, а также обучения школьников программированию. На кафедре системного программирования СПбГУ разрабатывается платформа QReal [18], а также её преемник REAL.NET [16], являющийся платформой для создания своих языков программирования. QReal написан на C++ с использованием библиотеки Qt [7], а REAL.NET на платформе .NET с использованием языков C# и F#.

В QReal используется концепция метамоделирования, заключающаяся в создании метамоделей и моделей. Первые являются средствами построения вторых. Модели являются конкретными реализациями метамоделей и описаниями конкретных структур данных. Так, метамодель является описанием языка на уровне предметной области, задавая множество всех синаксически корректных диаграмм на данном визуальном языке, а модель является конкретной программой на этом языке.

В основу концепции работы с метамоделями в REAL.NET заложено так называемое глубокое метамоделирование [13], поддерживающее много метауровней. Последние являются метамоделями, которые с одной стороны могут представлять лингвистическое инстанциирование, т.е характировать создание экземпляра объекта в терминах языка моделирования, а также онтологическое, которое представляет собой передачу "идеи сущности" между уровнями. Таким образом промежуточные метамодели выполняют роль как полноценных объектов, так и типов. Достигается это путем введения целого числа, так называемого потенциала к свойствам метамоделей и уменьшением его значения на каждом уровне. Если данное число положительно, то полю не может быть присвоено конкретное значение. Сделать это можно несколькими метамоделями ниже, у соответствующего объекта потенциал поля

которого станет равным 0.

При создании своих метамоделей требуется обеспечить корректность потенциально созданных программ на данном визуальном языке. Синтаксическую корректность программ обеспечивает сама метамодель языка. А для создания семантически корректных моделей требуется накладывать некоторые логические ограничения на метамодели.

В платформе REAL.NET было предложено использовать для этих целей декларативный язык OCL [1], являющийся известным языком ограничений, позволяющий задавать их, а также выполнять запросы к метамоделям. Проверка этих ограничений должна быть выполнена средствами интерпретатора.

Однако реализация обычного интерпретатора OCL не является достаточным условием для создания ограничений на глубокие метамодели. Поэтому требуется модифицировать грамматику OCL, так как OCL поддерживает только два уровня абстракций (метамодель и модель). Таким образом целью данной работы является создание модифицированного интерпретатора OCL для задания ограничений в платформе REAL.NET.

В рамках данной работы были поставлены следующие задачи:

- выполнить обзор публикаций по глубокому метамоделированию;
- определить требования к интерпретатору;
- выбрать подмножество языка OCL и модифицировать его в соответствии с требованиями;
- реализовать интерпретатор OCL;
- выполнить интеграцию с REAL.NET;
- выполнить апробацию с помощью тестов и опросов пользователей.

1. Обзор

1.1. Язык OCL

Язык OCL является декларативным языком описания ограничений, задаваемых в UML. Он был разработан компанией IBM, и сейчас является частью стандарта UML, позволяя задавать ограничения на диаграммы. Выражение на OCL состоит из четырех частей:

- контекст (модель и узел), в рамках которого выражение будет верным;
- свойства, которые представляют характеристики контекста;
- вычисления, арифметические или операции над множествами, которые управляют свойствами;
- ключевые слова, определяющие условные выражения.

Выражения на OCL могут состоять из запросов или логических условий, применяемых к моделям. Также они могут быть заданы в предусловии или постусловии. Таким образом, корректность данного выражения может быть проверена до или после выполнения выражения.

Для указания типа объектов, на экзепляры которых будет действовать ограничение, используется ключевое слово context. В приведененных примерах ниже контекстами являются типы Person, Address, Job. Для обращения к собственным полям объектов этих типов используется ключевое слово self, для статических свойств его писать не нужно. Сами ограничения заданы в блоках inv, pre и post, соответствующих инвариантам, предусловиям и постусловиям. Их может быть больше одного в рамках одного контекста. Внутри этих блоков может находится одно логическое условие, использующие операторы and, or, implies, not, а также конструкции if-then-else-endif и т. п. В постусловии можно использовать переменную result, записав значение в которую можно определить возвращаемое значение метода. Также после ключевых слов inv, pre и post и перед самим логическим выражением можно опре-

```
context Person inv :
    self.wife->notEmpty() implies self.wife.age >=18 and
    self.father->notEmpty() implies self.age < self.father.age

context Address inv:
    self.town.notEmpty() implies (self.region.otEmpty() and
    self.region.town.includes(self.town) ) or (self.region.isEmpty() and
    self.country.town.includes(self.town) )

context Job
    inv: self.employer.numberOfEmployees >= 1
    inv: self.employee.age > 21

context Person inv:
let income : Integer = self.job.salary->sum()
let hasTitle(t: String) : Boolean = self.job->exists(title = t)
    if isUnemployed then income < 100 else income >= 100 and hasTitle('manager')
    endif

context Person::income(d: Date): Integer
pre: d.value >= self.job.startDate.value
post: result = 5000
```

делить одну или несколько переменных или функций, используя ключевое слово let.

В вышеприведенных примерах приведены ограничения в контексте типа Person, задающие ограничения на возраст жены и возраст отца. Далее следуют ограничения в контексте Address, которые проверяют наличие города в регионе и стране. В контексте Job заданы два ограничения на количество и возраст сотрудников. Затем снова задается ограничение в контексте Person: теперь задана переменная income, равная сумме зарплат, а также функция hasTitle, проверяющая существование работы с данным именем в портфолио человека. В самом ограничении происходит проверка трудоустроенности человека и в зависимости от этого выполняется соответствующее условие на доход и наличие должности в портфолио. Далее приведена функция дохода в контексте типа Person от даты, в которой сначала происходит проверка корректности даты, а затем возврат значения.

1.2. Обзор интерпретаторов OCL

На момент начала работы отсутствовали интерпретаторы OCL с открытым исходным кодом, написанные для платформы .NET последних версий.

Существует проект OCL Compiler от Dave Arnold [6], написанный под .NET 1.1, который поддерживать и модифицировать было бы затруднительно.

Незадолго до окончания работы над проектом, описанным в этой работе, был реализован проект CrossEcore for C# [14], который позволяет генерировать код на C# по модели классов, ограничения на которую делаются с помощью OCL. Это подтверждает актуальность работы, но потенциально использовать данные наработки стало возможным лишь к окончанию работы.

На кафедре системного программирования СПбГУ в 2017 году была сделана бакалаврская работа Когутича Дениса [15], в которой было осуществлено создание интерпретатора OCL на языке JavaScript с использованием ANTLR.

1.3. Глубокое метамоделирование

На сегодняшний день существуют два основных вида метамоделирования. Так называемый “тип-объект” и подход глубокого моделирования.

В первом классическом двух-уровневом подходе задается метамодель и модель, объекты в которой являются экземплярами метамодели. Во втором количество уровней не ограничено и пользователь может задавать любое количество, а промежуточные звенья могут выполнять роль как типов, так и инстанциированных объектов.

Минусы первого подхода видны на примере модели, где нужно иметь и тип (например книги) и сам объект (конкретная книга). В таком случае пользователь вынужден настраивать связи между типом и экземпляром вручную в модели, а также иметь выделенный тип экземпляра объекта в метамодели. С использованием глубокого метамоделирова-

ния уменьшается сложность для такого случая, так как типы нижней метамодели являются экземплярами (см. рис. 1).

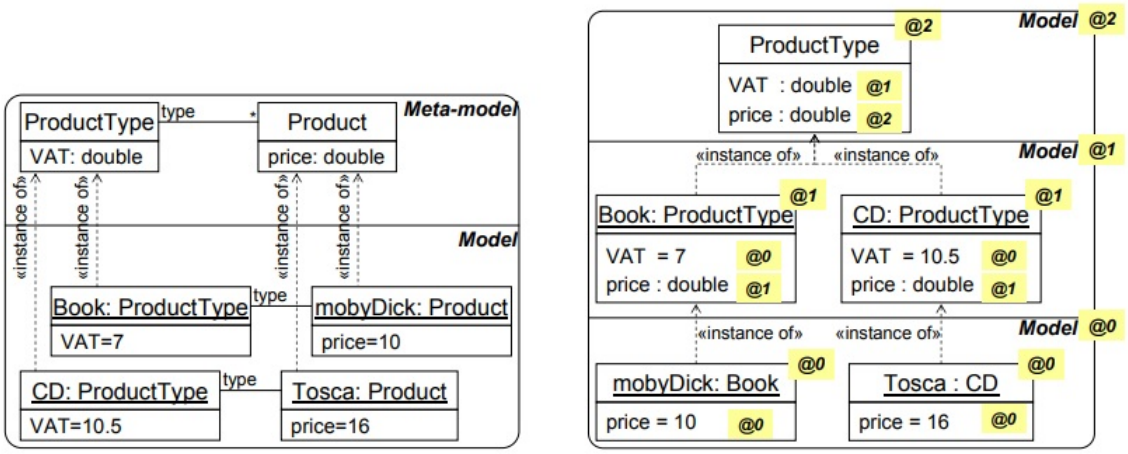

Рисунок 1. Сравнение двух типов метамоделирования [13].

Можно заметить, что связи между типом и экземпляром, настраиваемые вручную в метамодели слева, являются отношением инстанциирования объектов в предметной области, т.е наделяют объект ”идеей” соответствующей сущности, это так называемое онтологическое инстанциирование, которое реализуется на рисунке справа между предпоследнем и последним уровнем.

Онтологическое инстанциирование имеет абстрактный характер, зависящий от предметной области. Так как существует инстанциирование класса в терминах языка моделирования, мы можем ввести второе измерение инстанциирования (так называемое лингвистическое), которое каждую сущность модели делает экземпляром некоторой сущности языка. На рис. 2 приведен пример, в котором весь онтологический стек моделей является экземпляром лингвистической модели. Слева задана лингвистическая метамодель с описанием объектов, которые могут инстанциировать себя, а справа реализация этой метамодели в виде онтологического стека.

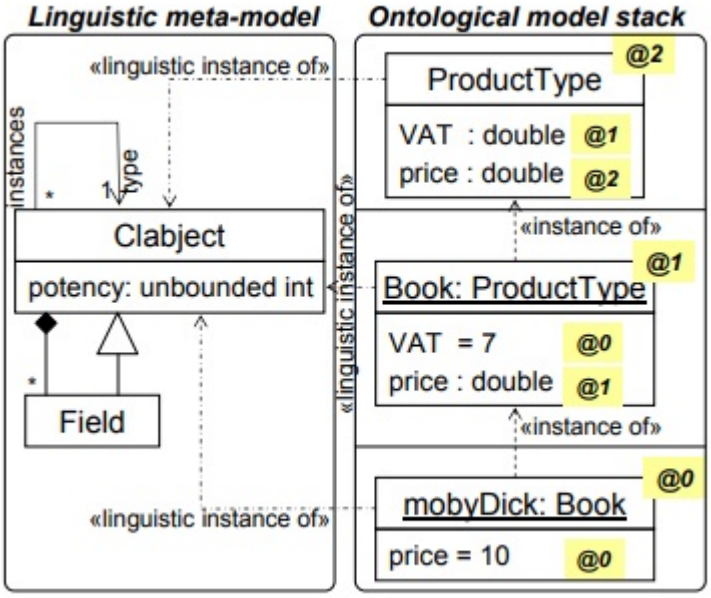

Рисунок 2. Два измерения инстанциирования [13].

Смотреть на одну и ту же модель можно двумя способами: в лингвистическом и онтологическом виде. На рисунке 3 приведен лингвистический взгляд на стек моделей, в котором между уровнями происходит инстанциирование в терминах языка: объект собака (Lassie) является экземпляром абстрактного объекта (Object), который в свою очередь инстанциирует объект класс (Class). На рисунке 4 та же модель представлена в онтологическом формате: собака (Lassie) является экземпляром породы колли (Collee), который в свою очередь является экземпляром объекта породы (Breed). Все эти объекты являются лингвистическими наследниками соответствующих объектов, находящихся справа от них, которые в свою очередь тоже представляют онтологический стек.

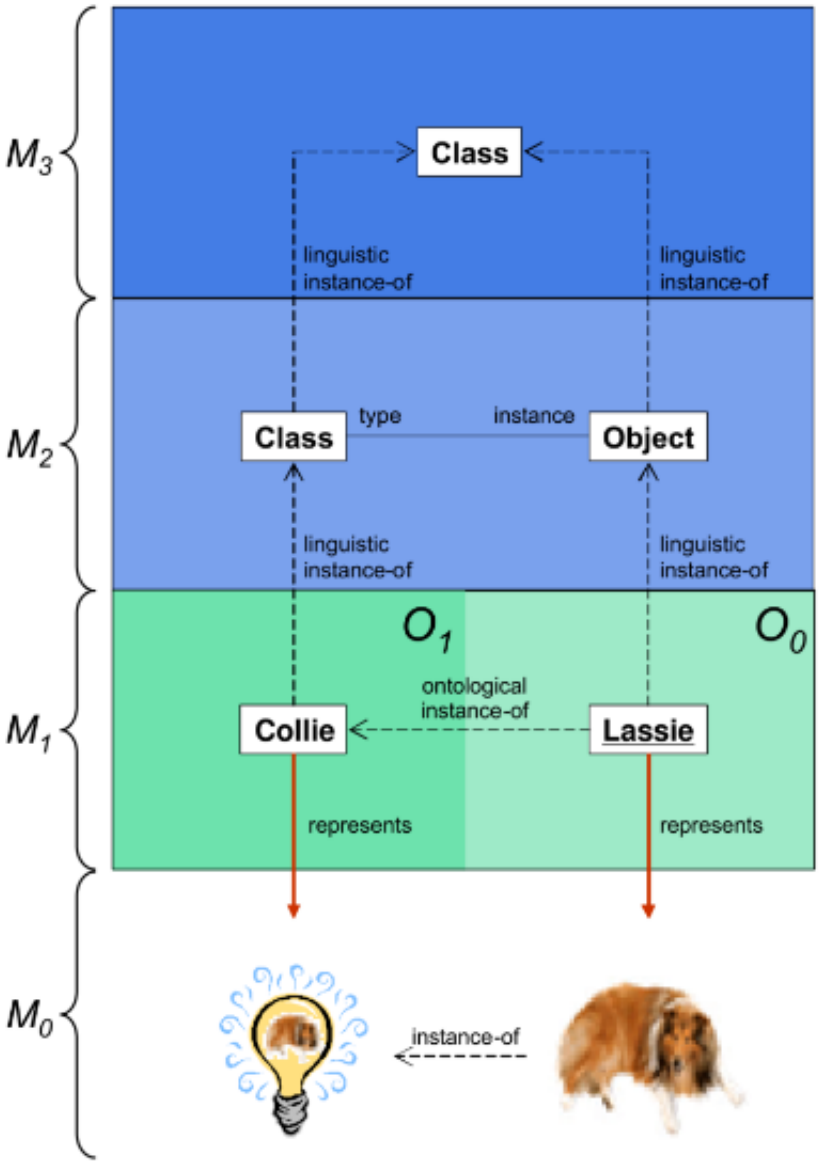

Рисунок 3. Лингвистический вид модели [3].

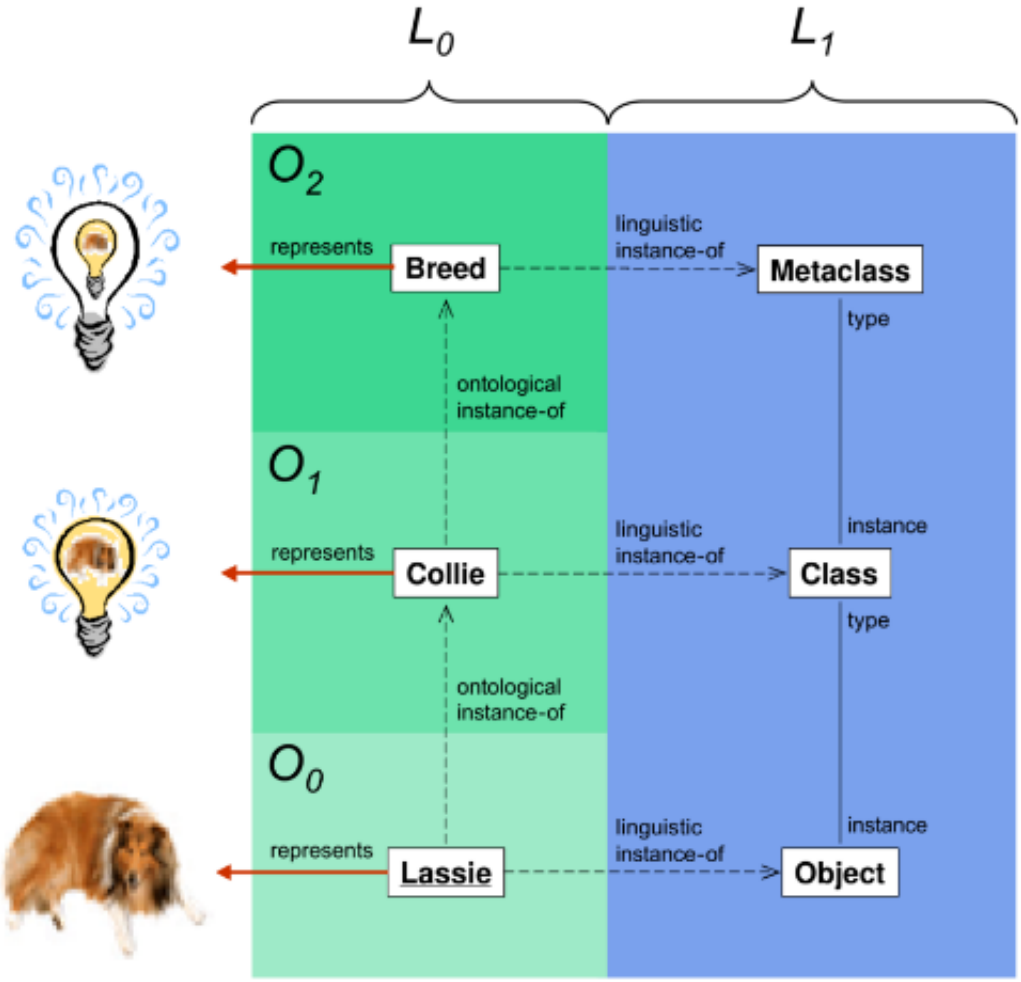

Рисунок 4. Онтологический вид модели [3].

1.4. Платформа REAL.NET

Интегрировать данный интерпретатор предлагается в платформу REAL.NET [8], которая является средой предметно-ориентированного визуального моделирования [5, 10, 9] и написана для платформы .NET. Предлагается создать плагин для данной среды, который будет реализовывать интерпретатор OCL для задания ограничений на метамодели.

Предшественником данной платформы является система QReal, написанная на C++/Qt. Данная система служит средством создания своих визуальных языков, редакторов с помощью метаязыка и метаредактора. Ограничения на метамодели задаются с помощью модели на

специальном языке ограничений с помощью диаграмм, состоящих из элементарных ограничений, внутри которых можно графически задать логические условия, применимые к типу или нескольким элементам типа метамодели [17].

В основе задания синтаксических правил визуального языка в REAL.NET лежит принцип “глубокого метамоделирования”, который заключается в том, что объекты и типы сущностей моделей рассматриваются как одно целое. Такое представление особенно хорошо в случаях работы с элементами подпрограмм, которые нужно представлять и как типы сущностей вызовов, и как сами объекты, являющимися конкретными вызовами.

Общая архитектура платформы REAL.NET приведена на рисунке 5. Основным компонентом является репозиторий, в нем инкапсулировано создание моделей и метамоделей. Более высокие слои предоставляют API для работы с этими метамоделями, например, создание атрибутов. Структуры данных, использующиеся в репозитории, определены через свои модели.

Модели хранятся с помощью своего внутреннего представления, один редактор реализован с использованием библиотеки хранения графов MS AGL на Windows Forms, что позволяет запускать REAL.NET на Linux / Mac OS, а второй использует GraphX и WPF, что дает более развитый интерфейс на ОС Windows. Репозиторий реализован на F#, а средство ограничений на C#. Также есть специализированные редакторы для платформ вроде REAL.NET.

Доменную область можно определить как с помощью визуальных редакторов, так и через систему плагинов, которая позволяет добавлять функционал в редактор.

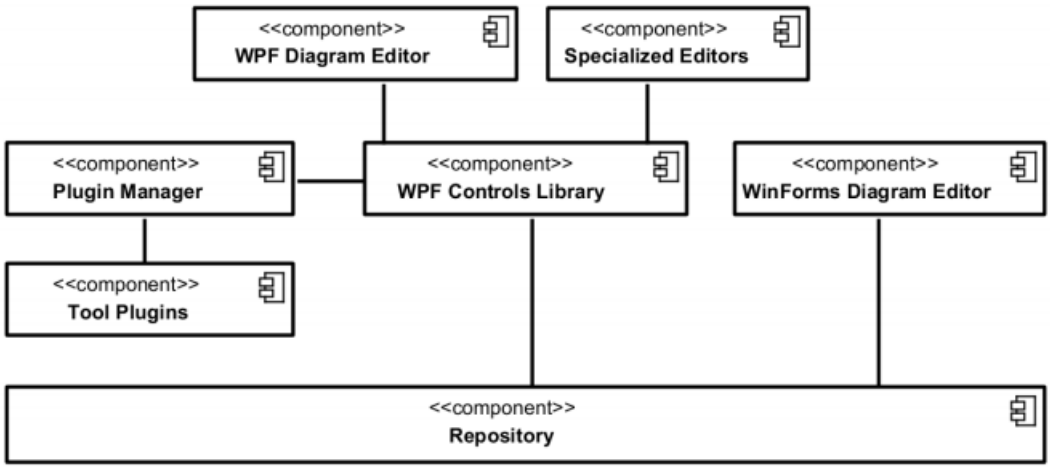

Рисунок 5. Архитектура REAL.NET [4].

На рисунке 6 показано окно редактора GraphX/WPF: диаграмма, палитра, окно свойств.

Рисунок 6. Окно редактора GraphX/WPF.

Средство проверки ограничений ализовано следующим образом:

1) пользователь описывает ограничения в специальном редакторе, который берет информацию из репозитория;

2) правила переводятся во внутреннее представление и кладутся рядом с метамоделью;

3) в результате при работе с моделью в случае изменения объектов и свойств будет проведена проверка ограничений.

Интеграции визуального средства ограничений в REAL.NET посвящена бакалаврская работа на кафедре системного программирования Дарьи Алымовой, находящаяся в разработке в момент написания этого текста. Таким образом, в REAL.NET планируется поддержка двух способов задания ограничений, так как не все ограничения можно удобно задать визуально.

1.5. Фреймворк MetaDepth

MetaDepth [13] – фреймворк, который поддерживает двойное лингвистическое / онтологическое моделирование, и который позволяет выполнять его с произвольным числом онтологических мета-уровней.

Для контроля атрибутов у полей в MetaDepth вводится потенциал, для того, чтобы контролировать на каком уровне мы можем присвоить некое значение атрибуту. Он задается возле свойства, после символа @. Так, поле price на рис. 9 имеет потенциал 2, а значит может быть инициализировано двумя метауровнями ниже.

Модели задаются в MetaDepth, в отличие от REAL.NET, на языке Epsilon Object Language, который является расширением языка OCL с поддержкой операций с побочными действиями, такие как присваивания и методы. На рисунке 7 показан пример модели Store на языке EOL. Создается единственный узел ProductType с тремя полями и соответствующими потенциалами, а также три ограничения, оперирующие этими полями. Поляледняя строчка в узле ProductType является переопределением унаследованного поля.

```
1 Model Store@2 {
2  Node ProductType@2 {
3   VAT@1      : double = 7.5;
4   price@2    : double = 10;
5   discount@2: double = 0;
6   minVat@1   : $self.VAT>0$
7   minPrice@2: $self.price>0$
8   maxDisc@2 : $self.VAT*self.price
8    *0.01+self.price<self.discount$
9   /finalPrice@2: double =
9    $self.VAT*self.price/100
9    +self.price-self.discount$;
10  }
11 }
```

Рисунок 7. Модели и ограничения в MetaDepth.

1.6. Платформа WebDPF

Еще одной платформой для работы с губокими метамоделями является WebDPF [12]. Это облачное веб-приложение, разработанное на HTML5 и JavaScript и поддерживающие визуализацию глубоких метамоделей, их ограничений, а также автоматическое исправление некорректных моделей. Данная платформа позволяет также визуализировать только часть больших моделей.

Окно редактора WebDPF приведено на рисунке 8. Оно состоит из четырех элементов. Панель управления слева позволяет пользователю выбирать метамодель и также предоставляет такие операции, как проверка корректности. Просмотрщик метамодели позволяет определять типы для моделей, а просмотрщик сигнатур позволяет графически задавать арность и визуализировать предикаты.

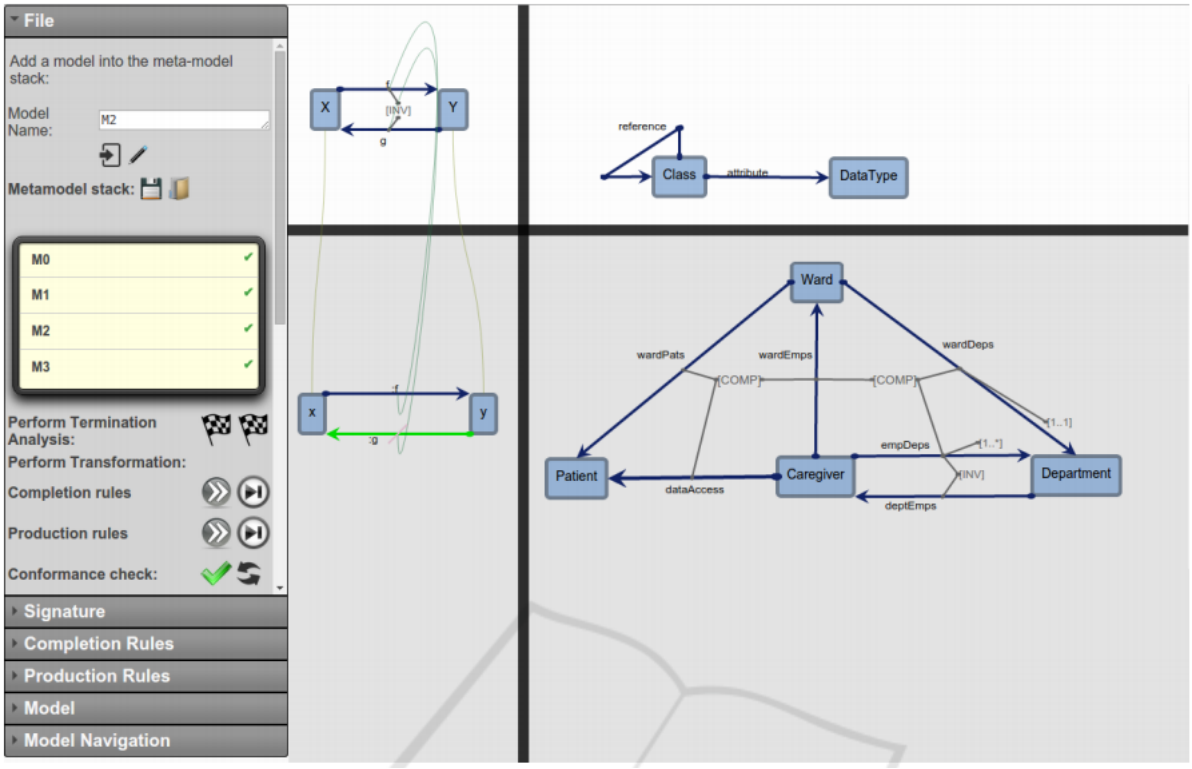

Рисунок 8. Редактор метамоделей WebDPF.

Для каждого предиката платформа генерирует JavaScript объект. Проектировщик языка должен создать свой метод, который будет вызываться платформой, чтобы определить удовлетворяет ли модель своей метамодели. Объекты с некорректным состоянием подсвечиваются в редакторе.

Для больших моделей можно воспользоваться запросами в собственном формате, которые отфильтруют элементы. Так, на рисунке 9 приведен пример большой модели, а на рисунке 10 только узлы, имеющие отношение к пациенту “Stephen”. Таким образом платформа может выделять зависимые группы элементов от данного.

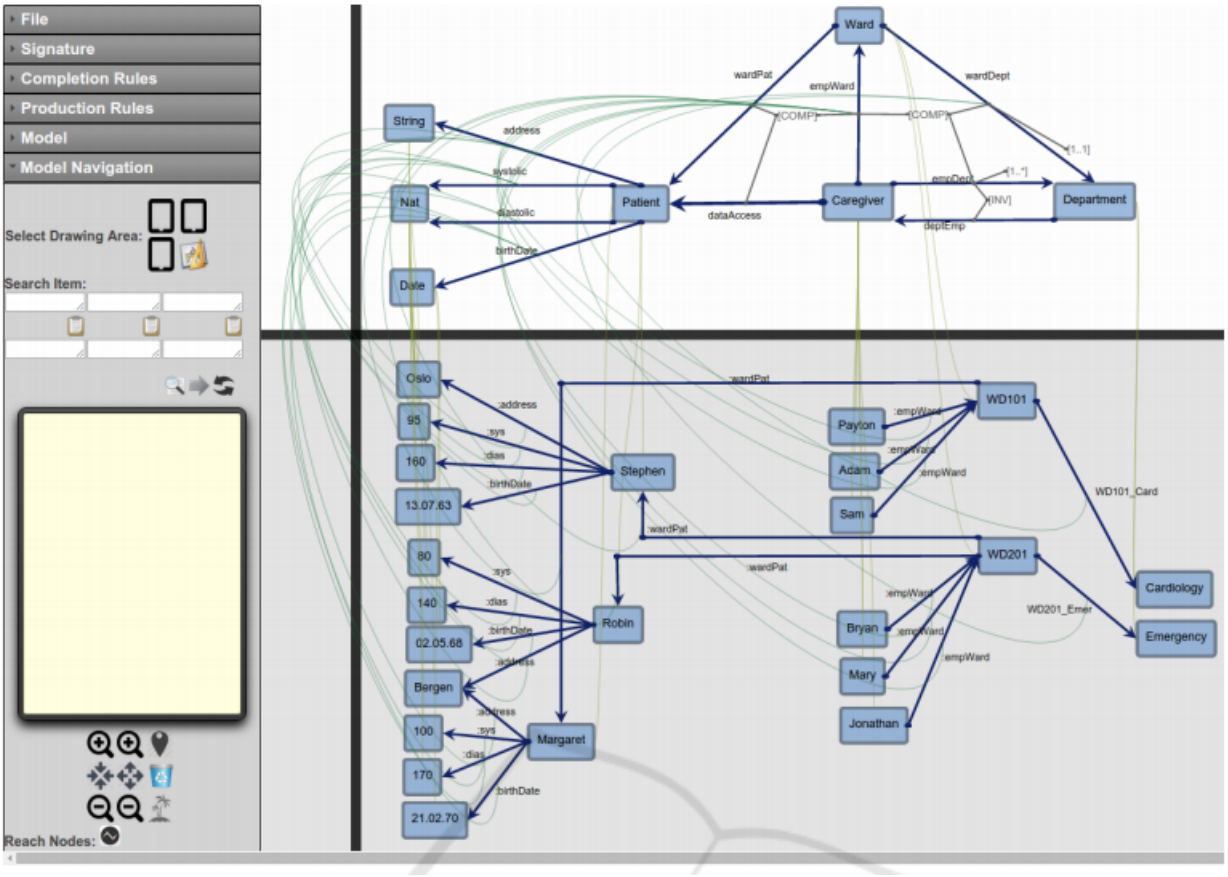

Рисунок 9. Большая модель.

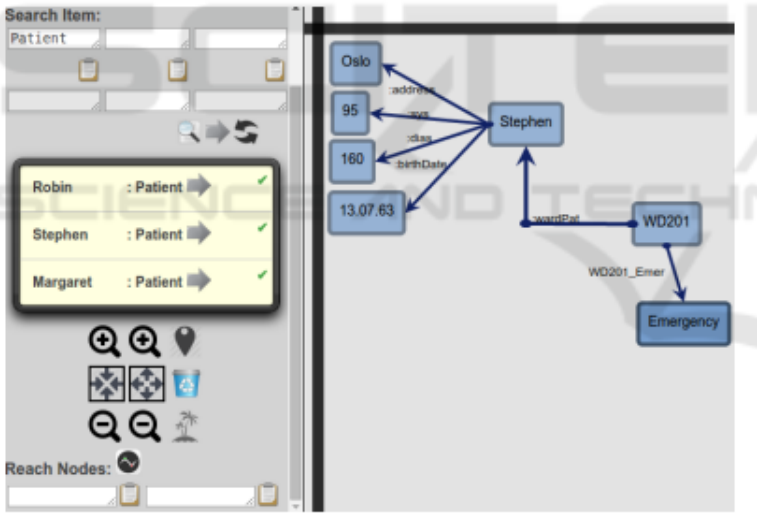

Рисунок 10. Часть отфильтрованной модели.

Платформа WebDPF позволяет автодополнять некорректные модели до правильного состояния, основываясь на заданных правилах дополнения моделей. Примеры таких правил приведены на рисунке 11.

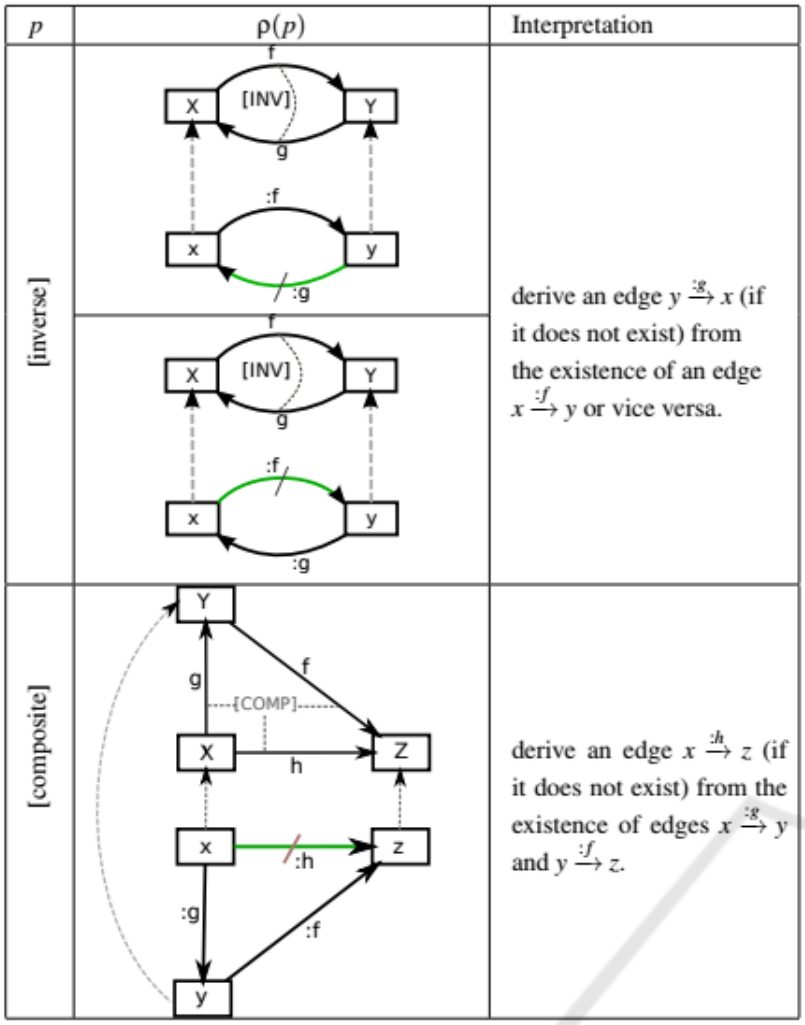

Рисунок 11. Примеры правил автодополнений.

1.7. Фреймворк ANTLR

Для реализации OCL-интерпретатора был выбран фреймворк ANTLR [2].

ANTLR – средство для генерации нисходящих анализаторов для формальных языков. Использовалось в данной работе для генерации синтаксического и лексического анализаторов на языке C# для интеграции в REAL.NET.

Схема работы с ANTLR подразумевает создание грамматики и запуск генерации классов по ней. По заданной грамматике ANTLR генерирует вспомогательные файлы на заданном языке (C#, Java, C++ и другие).

При запуске, ANTLR получает конкретное выражение, строит по нему конкретное синтаксическое дерево, которое можно использовать для обработки с помощью паттернов Visitor и Listener. Первый способ позволяет контролировать вызов функций для обработки дочерних узлов, так как управление обхода дерева предоставляется пользователю. Во втором используется принцип "инспектора": при каждом входе в вершину вызывается сгенерированный фреймворком метод Enter, а при выходе Exit. Данные методы могут быть переопределены пользователем. Был выбран первый паттерн, так как он предоставляет более наглядную работу и контроль при обходе дерева в отличие от первого. ANTLR использует LL(*)-разбор.

2. Модификация грамматики OCL

2.1. Требования

Для работы с глубокими метамоделями требовалась модификация грамматики OCL, так как спецификация этого не поддерживает.

Так, например, в контексте для задания ограничения на выражение с функциями в модели "Узел" -> "Функция" -> "Определение функции" -> "Вызов этой функции в конкретной диаграмме" ограничение может быть такое: если на диаграмме присутствует вызов функции, то должно быть представлено и соответствующее опеределение. Соответственно использовать стандартный OCL для написания такого ограничения не представляется возможным, так как OCL позволяет работать только с двумя метауровнями.

Были составлены требования к интерпретатору:

- возможность задавать ограничения на элементы более глубоких метамоделей, соответствующим данному;
- возможность обращаться к полям из более глубоких метамоделей;
- возможность комбинировать обращения к полям из разных метамоделей.

2.2. Модифицированная грамматика

Для реализации данных требований было предложено следующее решение:

- помечать каждое ограничение номером самой глубокой метамодели относительно данного элемента через символ @;
- помечать обращения к полям из более глубоких метамоделей номерами через символ @.

Реализация данного решения заключается в следующем. При обработке каждого ограничения интерпретатор получает все элементы из репозитория на глубине, равной заданному номеру ограничения относительно элемента, на который задается ограничение. Затем он заменяет

номера у полей на отрицательное число, равное разнице между максимальной глубиной и глубиной данного поля. Это означает, что для вычисления значения поля интерпретатор должен обратиться к типам от которого инстанциирован данный элемент заданное количество раз.

Соответствующее OCL-выражение, которое может быть задано в новой реализации интерпретатора OCL для выполнения ограничения, приведенного выше:

```
package P
context Function
inv@2:
    self.allInstances@2->select(x.type == "Call").forAll(true implies
self.allInstances@2->select(y.type == "Def")->exists(z.name == x.name))
endpackage
```

При обработке данного инварианта интерпретатор запрашивает все объекты из репозитория с глубиной инстанциирования равной двум, фильтрует их по типу, применяет для всех них существование соответствующего по имени объекта определения функции.

Дополняющий пример работы с глубокими метамоделями является ограничением “для любой функции вызов функции должен передавать столько параметров, сколько принимает определение функции” (ограничение пишется на узел “Функция” (уровня 1), но применяется к экземплярам конкретных функций (уровня 3)):

```
package P
context Function
inv@2:
    self.callParams@2.size() == self.params@1.size()
endpackage
```

Таким образом, при обработке такого инварианта происходит получение всех объектов с глубиной инстанциирования 2 из репозитория, и к ним применяется симметричное OCL выражение: self.callParams@0.size() == self.params@-1.size(), отрицательные поля в семантике которого означают получение типа от которого инстанциирован данный несколькими уровнями выше.

Грамматика модифицированного языка OCL, которая является исходными данными для ANTLR, представлена в приложении.

3. Реализация интерпретатора

3.1. Архитектура решения

Было решено создавать интерпретатор с динамической системой типов. В отличие от статической системы это дает возможность писать более короткий и понятный код, а также делает язык более выразительным. Платформа REAL.NET предназначена для быстрого создания предметно-ориентированных языков, поэтому статическая система не подходит из-за больших затрат на контроль типов, а её плюсы типобезопасности не так существены для небольших конструкций, использующихся для задания OCL-ограничений. Рассмотрим следующее OCL-выражение:

```
Sequence{Bag{1}, Set{'a'}, Set{1.0}}->first()->asSequence()->first()+1
```

Во время выполнения статического анализа типов будет ошибка, так как тип после второго вызова first не гарантируется что будет численным. Динамический анализ не будет запрещать выполнять данное корректное для данного случая выражение.

Диаграмма классов решения приведена на рисунке 12.

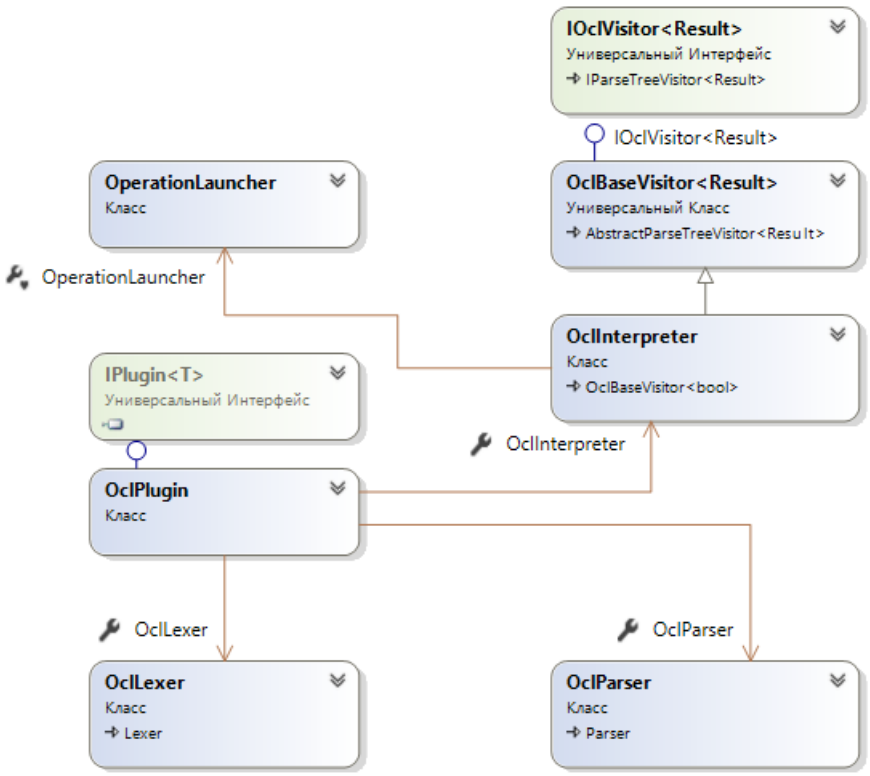

Рисунок 12. Диаграмма классов интерпретатора

Сам плагин REAL.NET реализован в классе OclPlugin, который унаследован от интерфейса IPlugin платформы REAL.NET и ссылается на OclInterpretor, в котором реализован обход абстрактного синтаксического дерева, и который унаследован от класса OclBaseVisitor системы ANTLR, который в свою очередь реализует интерфейс IOclVisitor, также сгенерированный фреймворком ANTLR. OclParser и OclLexer являются сгенерированными классами ANTLR по грамматике и выполняют роль парсера и лексера соответственно, их экземпляры хранятся в классе OclPlugin. Класс OperationLauncher, являющийся также свойством OclPlugin, инкапсулирует логику запуска конкретного обработчика операции OCL.

Обработчики операций реализованы в классах, унаследованных от базового класса Operation. Их частичная диаграмма классов приведена на рисунке 13.

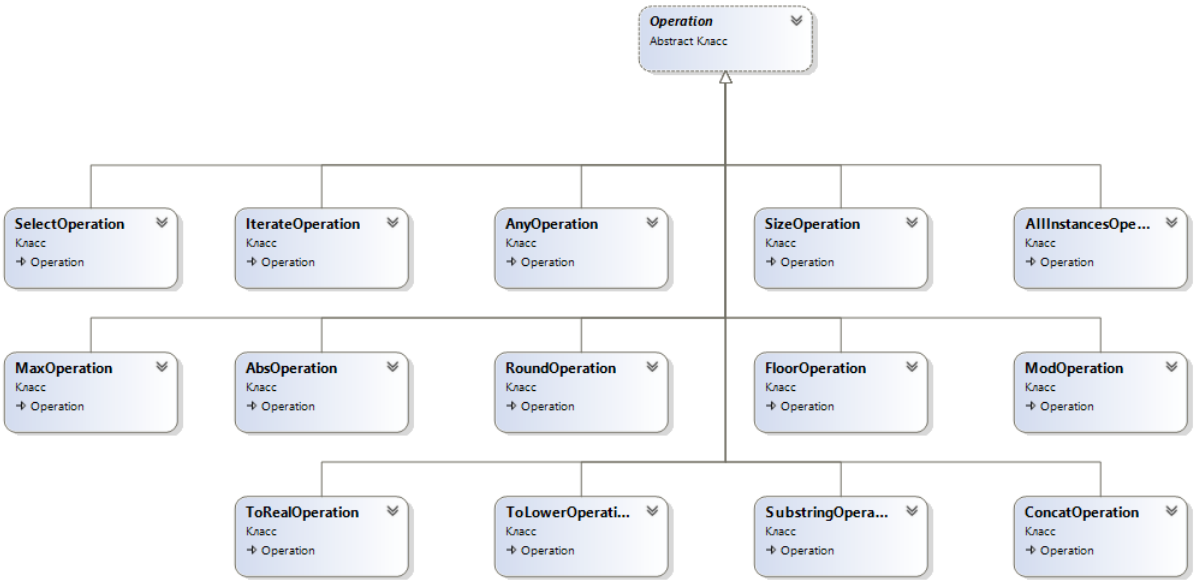

Рисунок 13. Диаграмма классов операций

На рисунке 13 приведены операции работы с коллекциями в первой строчке, с числами во второй и со строками в третьей.

Результаты всех операций инкапсулированы в специальных классах, иерархия которых приведена на рисунке 14. Так, например, тип Int явялется подклассом Double, что позволяет выполнять вычисления с разными числовыми данными. А своя система типов позволяет определять тип во время выполнения.

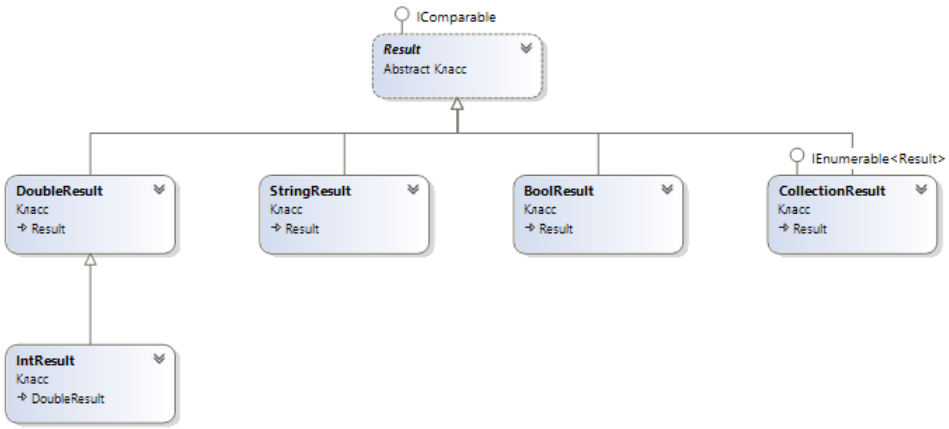

Рисунок 14. Иерархия классов результатов вычислений

В целом же, был реализован семантический анализ абстрактного синтаксического дерева подмножества языка OCL с поддержкой арифметических, логических операций, операций работ со строками и некоторых операций с коллекциями. Реализована поддержка своих пользовательских опередедений функций в том числе рекурсивных. При вызове данных функций происходит переход к вершине AST-дерева, соответствующей данной функции. Информация о данном соответствии хранится в отдельной хэш-таблице.

Пример определения своей рекурсивной функции и вызова проверки ограничения, а также другого простого ограничения на равенство значений:

```
package P
context A
inv:
    let fact(n: var) = if n = 1 then n else n * fact(n - 1) endif;
    let n = 5;
    fact(n+1) = 720
inv:
    let a = 6;
    let b = 7;
    a = b
endpackage
```

Само выражение задается в текстовом редакторе или файле, который считывается и обрабатывается плагином, используя ANTLR Runtime, затем плагин пишет результат проверки в консоль.

3.2. Плагин REAL.NET

В рамках задачи был создан плагин интерпретатора OCL. Архитектура поддержки плагинов в REAL.NET приведена на рисунке 15:

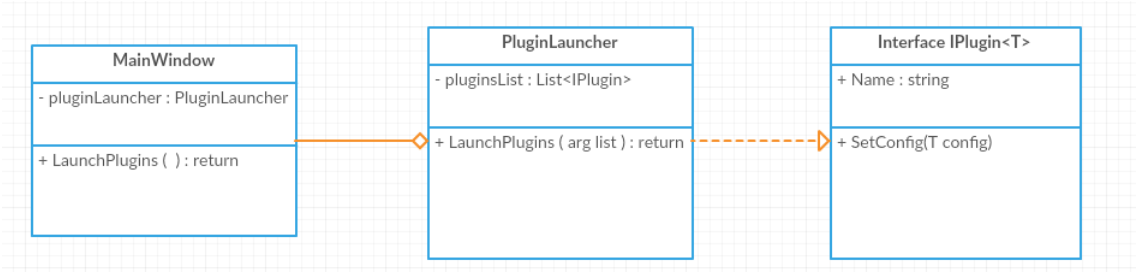

Рисунок 15. Плагины в REAL.NET.

Главное окно MainWindow содержит экземпляр класса PluginLauncher и вызывает его метод LaunchPlugins при инициализации. В методе LaunchPlugins происходит обработка всех библиотек, лежащих в папке, создание их экземпляров через методы рефлексии, запуск и установка их конфигурации, передавая функциям инициализации плагинов модель, панель инструментов, сцену, консоль и палитру. Таким образом, плагин может конфигурировать их под свои нужды. Например, OCL-плагин таким образом получает доступ к моделям, позволяя ссылаться на элементы моделей в выражениях.

Плагин интерпретатора реализован в классе OclPlugin.

3.3. Редактор

Редактор реализован с помощью WPF-компонента RichTextBox. Он поддерживает задание OCL-ограничений в нескольких строчках, а его содержимое предварительно заполняется шаблоном OCL-ограничения на открытую в данный момент модель. Внешний вид редактора с заданным шаблоном ограничения приведен на рисунке 16.

Рисунок 16. Внешний вид редактора.

Ограничения задаются в редакторе в отдельной вкладке ”Ocl” в нижней части панели. После задания ограничения пользователю нужно нажать на кнопку, расположенную справа от редактора. В случае успеха, в консоль во вкладке ”Messages” будет выведено сообщение ”Ok”, а в случае неуспеха – ”Error”, а некорректные элементы будут подсвечены красным цветом на сцене.

3.4. Интеграция со сценой

При выполнении OCL-выражения происходит перебор всех элементов моделей и у тех, которые соответствуют ограничению, выставляется атрибут isValid. Последний является источником данных для свойств Color и ShadawColor для текстового и картиночного отображения соответственно.

Пример объекта с некорректным состоянием приведен на рисунке 17.

Рисунок 17. Объект с некорректным состоянием подсвечен красным

4. Апробация

4.1. Тесты

Для проверки корректности реализации интерпретатора были написаны тесты в открытой среде юнит-тестирования приложений для .NET NUnit. В инициализации данных тестов происходит заполнение репозитория тестовыми моделями. Сами тесты покрывают проверку работы OCL с моделями, строками, числами, коллекциями, проверяя возвращаемый результат. Данные тесты предназначены для проверки корректности разработанного инструмента.

Для подсчета покрытия кода использовался стандартный инструмент, интегрированный в Visual Studio Enterprise. Тестовое покрытие исходного кода реализации интерпретатора, выполненной в данной работе (не учитывая сгенерированных классов), равняется 90%.

4.2. Анкетирование

Был проведен юзабилити-эксперимент на нескольких студентах, знакомых с OCL и понятиями метамоделирования. Им предлагалось составить программу на языке OCL для задания ограничения на наличие у функции соответствующего определения с заданной метамоделью в среде REAL.NET. Время не фиксировалось. После решения этой задачи предлагалось заполнить анкету System Usability Scale [11] и оставить отзыв.

Анкета System Usability Scale представляет инструмент измерения удобства пользования приложением. Она состоит из 10 вопросов с пятью вариантами ответов для участников от полного согласия до полного несогласия. Данные вопросы приведены ниже.

1. Я бы хотел(-а) поработать еще с этой программой
2. Программа слишком сложная
3. Этой программой легко пользоваться

4. Мне понадобится помощь, чтобы научиться пользоваться этой программой
5. Разные функции в этом приложении правильно сгруппированы
6. В приложении слишком много несоответствий
7. Большая часть людей очень быстро научиться пользоваться этой программой
8. Это приложение очень трудно использовать
9. Я уверенно себя чувствовал(-а), используя это приложение
10. Мне пришлось многому научиться, прежде чем я смог(-ла) работать с приложением

В зависимости от степени согласия с вопросами 1, 3, 5, 7, 9 начисляется от 0 (полное несогласие) до 4 (полное согласие) баллов. За вопросы 2, 4, 6, 8, 10 от 4 (полное несогласие) до 0 баллов (полное согласие). Затем сумма очков домножается на 2.5 для получения номированного относительно 100 результата. Исследования показали, что средний балл находится в районе 68 баллов.

Результаты тестирования составили 90, 70, 57.5, 77.5 и 85 баллов, что говорит об уровне удобства пользования выше среднего.

5. Заключение

Таким образом, в рамках данной работы были реализованы следующие поставленные задачи:

- выполнен обзор публикаций по глубокому метамоделированию и аналогичных решений;
- модифицирована грамматика OCL для поддержки нескольких метаслоев в соответствии с составленными требованиями;
- реализован интерпретатор модифицированного языка OCL на языке C#;
- выполнена интеграция с REAL.NET с помощью плагина и создания редактора;
- для проверки корректности проведены тесты с помощью технологии NUnit и опросы пользователей с оценкой по метрике System Usability Scale.

Потенциальные дальнейшие направления развития платформы заключаются в создании более совершенной системы ограничений. Так, например, можно показывать варианты исправления нарушений, обнаруженных на диаграммах. Также можно расширить функциальность редактирования кода: добавить подсветку синтаксиса и автодополнения.

Список литературы

[1] ABOUT THE OBJECT CONSTRAINT LANGUAGE SPECIFICATION. — https://www.omg.org/spec/OCL/ (дата обращения: 23.12.2018).

[2] ANTLR. — https://www.antlr.org/ (дата обращения: 23.12.2018).

[3] Atkinson Colin Kuhne Thomas. Model-driven development: a metamodeling foundation. — Software, IEEE. Т. 20, № 5. С. 36–41., 2003.

[4] Elizaveta Kuzmina Yurii Litvinov. Implementation of “smart greenhouse” visual programming tool using deep metamodeling. — The Fourth Conference on Software Engineering and Information Management (SEIM-2019). P. 3., Готовится к публикации.

[5] J. Luoma, S. Kelly, J.-P. Tolvanen. Defining domain-specific modeling languages: Collected experiences. — Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04) / OOPSLA. — Jyvävaskylä, Finland : University of Jyvävaskylä,, 2004.

[6] OCL Compiler. — http://www.davearnold.ca/csocl/ (дата обращения: 23.12.2018).

[7] Qt Documentation. — http://doc.qt.io/ (дата обращения: 21.05.2017г).

[8] REAL.NET: A set of .NET libraries for quick creation of visual languages and related tools. — https://github.com/yurii-litvinov/REAL.NET.

[9] S. Kelly, J.-P. Tolvanen, Domain-specific modeling: enabling full code generation. — Hoboken, New Jersey, USA : Wiley-IEEE Computer Society Press. P. 448, 2008.

[10] S. Kelly, J.-P. Tolvanen, Visual domain-specific modeling: Benefits and experiences of using metaCASE tools. — International Workshop on Model Engineering, at ECOOP, 2000.

[11] Sauro Jeff. A Practical Guide to the System Usability Scale: Background, Benchmarks and Best Practices. — CreateSpace Independent Publishing Platform.

[12] WebDPF: A Web-based Metamodelling and Model Transformation Environment. — 4th International conference on Model-Driven Engineering and Software Development, At Rome, Italy., 2016.

[13] de Lara1 Juan, Guerra Esther. Deep Meta-Modelling with MetaDepth. — Universidad Aut´onoma de Madrid. P. 3-5., 2010.

[14] ecore-csharp. — https://github.com/crossecore/ecore-csharp (дата обращения: 17.04.2019).

[15] Д. Когутич. Реализация механизма поддержки ограничений в проекте WMP. — кафедра системного программирования СПбГУ, 2017.

[16] Литвинов Ю.В. Кузьмина Е.В. Небогатиков И.Ю. Алымова Д.А. Среда визуального предметно-ориентированного программирования REAL.NET. — Всероссийская научная конференция по проблемам информатики СПИСОК-2017. — URL: http://spisok.math.spbu.ru/2017/txt/SPISOK-2017.pdf (дата обращения: 14.04.2018)., 2017.

[17] О. Дерипаска А. Языки задания ограничений. — Компьютерные инструменты в образовании. С. 21-22., 2013.

[18] Терехов А.Н. Брыксин Т.А. Литвинов Ю.В. QReal: платформа визуального предметно-ориентированного моделирования. — Программная инженерия, 2013, № 6, С. 11-19.

6. Приложение

Контексто-свободная грамматика OCL, используемая фреймфорком ANTLR представлена ниже. Ключевым нетерминалом которой является constraint, описывающий ненулевое число выражений на OCL. Он в свою очередь раскарывается в произовольное число let-выражений и само выражение expression. Нетериминал expression поочередно раскрывается в логические выражения через нетерминалы implExpression, orExpression, andExpression в соответствие с приоритетом, а затем в нетерминалы соответствующие сложению additiveExpression, умножению multiplicativeExpression, а также вычислению основного выражения primaryExpression и опционального вызова у него свойства propertyCall.

```
grammar Ocl;

oclFile : ( 'package' packageName
oclExpressions
'endpackage'
)+;
packageName : pathName;
oclExpressions : ( constraint )*;
constraint : contextDeclaration
( Stereotype '@' NUMBER NAME? ':' oclExpression)+;
contextDeclaration : 'context'
( operationContext | classifierContext );
classifierContext : ( NAME ':' NAME ) | NAME;
operationContext : NAME '::' operationName
'(' formalParameterList ')' ( ':' returnType )?;
Stereotype : ( 'pre' | 'post' | 'inv' );
operationName : NAME | '=' | '+' | '-' | '<'
 | '<=' | '>=' | '>' | '/' | '*' | '<>' | 'implies'
 | 'not' | 'or' | 'xor' | 'and';
formalParameterList : ( NAME ':' typeSpecifier
```

```
(',' NAME ':' typeSpecifier )*)?;
typeSpecifier : simpleTypeSpecifier | collectionType;
collectionType : collectionKind '(' simpleTypeSpecifier ')';
oclExpression : ( letExpression )* expression;
returnType : typeSpecifier;
expression : implExpression;
letExpression : 'let' NAME ( '(' formalParameterList ')' )?
 ( ':' typeSpecifier )? '=' expression ';';
ifExpression : 'if' expression 'then' expression
 'else' expression 'endif';
implExpression : orExpression ( 'implies' orExpression)*;
orExpression : andExpression ( orOperator andExpression)*;
andExpression : eqExpression ('and' eqExpression)*;
eqExpression : relationalExpression
(eqOperator relationalExpression)*;
relationalExpression : additiveExpression
(relationalOperator additiveExpression)?;
additiveExpression : multiplicativeExpression
( addOperator multiplicativeExpression)*;
multiplicativeExpression : unaryExpression
( multiplyOperator unaryExpression)*;
unaryExpression : ( unaryOperator postfixExpression)
 | postfixExpression;
postfixExpression : primaryExpression ( ('.' | '->')propertyCall)*;
primaryExpression : literalCollection | literal | propertyCall
| '(' expression ')' | ifExpression;
propertyCallParameters : '(' ( declarator )?
( actualParameterList )? ')';
literal : NUMBER | enumLiteral | stringLiteral | booleanLiteral;
stringLiteral : '''' (NUMBER | NAME)? '''';
enumLiteral : NAME '::' NAME ( '::' NAME )*;
simpleTypeSpecifier : pathName;
literalCollection : collectionKind '{' ( collectionItem
```

```
(',' collectionItem )*)? '}';
collectionItem : expression ('..' expression )?;
propertyCall : pathName ('@' NUMBER)? ( timeExpression )?
 ( qualifiers )? ( propertyCallParameters )?;
qualifiers : '[' actualParameterList ']';
declarator : NAME ( ',' NAME )* ( ':' simpleTypeSpecifier )?
 ( ';' NAME ':' typeSpecifier '=' expression )? '|';
pathName : NAME ( '::' NAME )*;
timeExpression : '@' 'pre';
actualParameterList : expression (',' expression)*;
orOperator : 'or' | 'xor';
collectionKind : 'Set' | 'Bag' | 'Sequence' | 'Collection';
eqOperator : '=' | '<>';
relationalOperator : '>' | '<' | '>=' | '<=';
addOperator : '+' | '-';
multiplyOperator : '*' | '/';
unaryOperator : '-' | 'not';
booleanLiteral : 'true' | 'false';
fragment LOWERCASE  : 'a'..'z' ;
fragment UPPERCASE  : 'A'..'Z' ;
fragment DIGITS  : '0'..'9' ;
NAME : (LOWERCASE | UPPERCASE | '_')
( LOWERCASE | UPPERCASE | DIGITS | '_' )* ;
NUMBER : DIGITS (DIGITS)* ( '.' DIGITS (DIGITS)* )?;
WS : [ \t\r\n]+ -> skip ;
```