Подходы к заданию семантики интерпретации диаграмм в рамках DSM-подхода

В. А. Поляков
vap.polyakov@gmail.com

Т. А. Брыксин
timofey.bryksin@gmail.com

В статье рассматривается задача интерпретации визуальных языков моделирования. Приводится обзор основных способов описания семантики визуальных языков, необходимой для осуществления такой интерпретации. Анализируются следующие подходы: двухуровневая схема отладки, Executable UML, Dynamic Meta Modeling и EProvide. Делаются выводы о применимости данных подходов при разработке DSM-платформ.

Ключевые слова: визуальное моделирование, модельно-ориентированная разработка UML, DSM-подход, предметно-ориентированное моделирование, DSM-платформы, отладка.

Введение

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

Помимо текстовых языков при разработке ПО в последнее время часто используется модельно-ориентированные средства разработки (Model-Based Development). Этот подход предназначен для

© В. А. Поляков, Т. А. Брыксин, 2012

спецификации системы и её предметной области в виде множества моделей, описывающих её с разных сторон. При этом стало принято для конкретных прикладных задач создавать новые специализированные предметно-ориентированные языки (Domain Specific Languages, DSL). При наличии такого языка модели становятся более наглядными и понятными, оказывается возможной эффективная автоматическая кодогенерация. Такой подход называют предметно-ориентированным моделированием (Domain Specific Modeling, DSM) [6–8, 10, 12, 20].

В настоящее время существует множество DSM-платформ — средств для разработки таких языков и соответствующего ПО (графические редакторы, кодогенераторы и т. д.); обзоры DSM-платформ можно найти в работах [9, 11]. Один из способов сделать создаваемые с помощью DSM-платформ средства более удобными — снабдить их средствами генерации пошаговых интерпретаторов и отладчиков. Подобный инструмент позволяет разработчику искать и исправлять ошибки на ранних этапах разработки. Также визуальная интерпретация диаграмм способствует лучшему пониманию проектировщиком принципов работы разрабатываемой системы, что, в итоге, повышает её качество. В некоторых UML-пакетах (например, в Borland Together[1]) возможность визуальной интерпретации и отладки уже присутствует. Однако данный подход было бы полезно распространить и на DSM-платформы.

Цель данной статьи — дать обзор основных способов спецификации исполняемой семантики визуальных языков, сделать выводы относительно их применимости на практике, наметить направления по улучшению таких средств. В работе рассматриваются следующие подходы: двухуровневая схема отладки, Executable UML, EProvide и Dynamic Meta Modeling.

1. Семантика языков

Любой новый язык является набором знаков, при помощи которых пользователь, следуя определённым правилам, может составить некий текст. Традиционно любой язык раскладывается по трём измерениям: синтаксис, определяющий правила построения текстов из знаков, семантика, задающая проекции знаков на пред-

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

метную область языка, и прагматика, описывающая способы использования языка пользователем.

Для языков визуального моделирования выделяют три вида синтаксиса: абстрактный, конкретный и служебный.

Служебный синтаксис — это формат хранения визуальных спецификаций. *Абстрактный синтаксис* определяет набор правил, при помощи которых можно объединять простые составляющие языка (знаки и конструкции) в сложные (тексты, т. е. визуальные модели). Обычно он задаётся в виде грамматики или метамодели. *Конкретный синтаксис* (графическая нотация или просто нотация) — это спецификация внешнего вида моделей, создаваемых с помощью данного языка (форма графических символов, правила размещения на графических элементах текста и т. д.).

Семантика визуального языка позволяет придавать конструкциям смысл, т. е. определяет их соответствие реальным объектам предметной области, называемой *семантической областью*.

В теории языков программирования выделяют четыре вида семантики: денотационная, операционная, аксиоматическая и трансляционная.

Денотационная семантика обычно использует в качестве семантической области 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 [30]. 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 (семантика малого шага [19]) и естественная (семантика большого шага [27]). 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, выполняющиеся в дальнейшем.

Аксиоматическая семантика представляют собой набор логических аксиом, а любые выражения, записанные на данном языке,

рассматриваются в виде логических формул, значения которых будут формально выведены из исходного набора аксиом [18].

Трансляционная семантика строится из правил преобразования моделей, с помощью которых модель на исходном языке переводится в модель на другом языке, для которого уже задана семантика исполнения [33]. Это часто сводится к генерации кода на языке общего назначения, для которого 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, так как понятия абстрактной исходной предметной области переходят в конкретные конструкции исполнимого языка.

Теперь перейдём к рассмотрению ряда конкретных подходов, предложенных в различных работах и использующихся в программных средствах.

2. Двухуровневая схема отладки

2.1. Описание

Двухуровневая схема отладки основана на идее двухуровневой трансляции, которая состоит в том, что программа на исходном языке транслируется в программу на другом языке (Java, C++, C#), для которого существуют отладчик и компилятор в бинарный исполняемый код.

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

Более подробно данную проблему можно описать на примере следующим образом. По коду программы на исходном языке генерируется код на некотором объектном языке, и установка точек останова происходит в нём в терминах номеров строк кода. Каждой

конструкции программы на исходном языке соответствует несколько строк кода на объектном, поэтому для того чтобы установить точку останова в терминах исходного языка, необходимо знать на какую из этих строк кода нужно ставить точку останова в объектном языке. Также это нужно учитывать и при простой пошаговой отладке, так как не всегда переход к следующему элементу потока исполнения в исходной программе будет соответствовать переходу на следующую строку кода в объектном.

Таким образом, создание отладчика исходного языка нужно производить на основе существующего отладчика объектного языка. Этот отладчик должен иметь интерфейс доступа к таким функциям, как запуск, приостановка и принудительное завершение программы, установка и снятие точек останова в терминах объектного языка, получение информации о стеках потоков вызовов в программе и др. В статье [5] приведено описание того, как данные стандартные функции могут быть реализованы в терминах исходного языка.

2.2. Анализ

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

В связи с высокой технической сложностью задачи и тем, что данный подход ориентирован на создание единичного отладчика для определённого языка, обобщение подхода двухуровневой отладки на предметно-ориентированное моделирование является неочевидным. Для нового языка процесс нужно будет повторять заново со всеми техническими деталями, что не даёт существенного упрощения, ускорения и автоматизации процесса создания отладчика по сравнению с ручным кодированием.

3. Язык xUML

Появление унифицированного языка моделирования UML (Unified Modeling Language) [26] способствовало сильному продвижению ви-

зуального проектирования среди разработчиков. Однако сам по себе UML — это язык именно проектирования и документирования, а не программирования, семантика многих его элементов либо не задана, либо многозначна. Для того чтобы применять диаграммы UML для генерации кода, был создан язык, получивший название исполняемого UML (Executable UML, xUML [16,21,22]). Формально этот язык является профилем UML, т. е. подмножеством UML с чётко определённой семантикой для каждого элемента. Это подмножество выбрано так, чтобы сделать на основе UML полноценный визуальный язык программирования общего назначения. Также как и UML, xUML основан на объектно-ориентированном подходе, т. е. описание системы ведётся в терминах классов, атрибутов и т. п.

3.1. Задание семантики при помощи xUML

Для задания семантики отдельных элементов в xUML используется так называемая семантика действий (Precise Action Semantics, PAS[2] [22]). PAS — это фиксированный набор семантических операций, не имеющий конкретного синтаксиса и реализации. PAS обеспечивает независимость, например, от конкретной целевой платформы или языка, на которых будут исполняться модели. Благодаря тому, что элементы обладают фиксированной семантикой, появляется возможность однозначно генерировать исполняемый код по модели, т. е. можно говорить об интерпретации модели.

Программирование на xUML основано на трёх следующих компонентах.

- Диаграммы классов, позволяющие описывать структурную модель системы. Эти классы имеют атрибуты, а связи между классами представляются в виде отношений.
- Диаграммы состояний и переходов, позволяющие задавать набор действий, которые совершают активные объекты, в виде конечного автомата.
- Язык действий (Action Language, AL), позволяющий задавать поведение объекта при нахождении в состоянии диаграммы состояний и переходов. Он необходим для создания экземпляров классов, установки связей между ними, выполнения

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

различных операций над атрибутами и т. п. AL является конкретной реализацией PAS, выбор которой зависит от предпочтений пользователя. xUML явно определяет только PAS.

В качестве примера на рис. 1 изображён фрагмент исполняемой модели в xUML для задачи автоматизации системы контроля поездов. Главными доменами этой предметной области являются непосредственное управление поездами (Train Management) и аппаратный интерфейс (Hardware Interface) всей системы.

В домене по управлению поездами выделяются две основные сущности (Classes) — поезд (Train) и поездка (Hop). Каждая поездка была осуществлена (is being made by) ровно одним поездом, и могут существовать поезда, которые ещё никуда не ездили (is making). Поездку характеризуют начальные дата (атрибут startDate) и время (атрибут startTime), а также пройденное расстояние (атрибут distanceCovered) и текущее состояние (атрибут currentState). Также у поездки есть один метод reportDistanceCovered, который позволяет узнать, чему равно пройденное поездом расстояние. Поезд характеризуется своей текущей скоростью (атрибут currentSpeed), состоянием (атрибут currentState) и таймером (атрибут timerId).

Описание диаграмм состояний и операций для данных классов (States and Operations) выполнено с помощью языка действий ASL [38]. Метод reportDistanceCovered выдаёт нужное значение и в комментариях не нуждается. Диаграмма состояний для поезда также довольно проста. На ней показано всего два состояния: ожидание следующей поездки (Waiting for Next Hop) и инициализация новой (Negotiating Hop). При ожидании новой поездки происходит следующее: из отношения R2 между поездом и поездкой, описанного на уровне классов, выбирается следующая поездка, и после этого выставляется таймер negotiateNextHop, переводящий поезд во второе состояние, на начальное время поездки nextHop. В состоянии инициализации новой поездки её значение опять берётся из отношения R2, и после этого вызывается метод negotiateHop для данной поездки. Его завершение означает конец поездки, и по событию hopNegotiated поезд возвращается в первое состояние.

Действие и PAS являются ключевыми понятиями в исполняемой части xUML. Действие — это конкретная операция, определённая

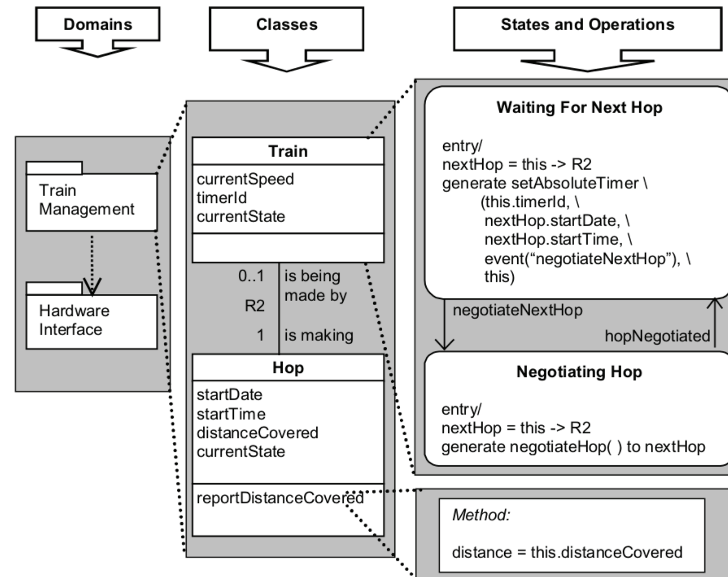


Рис. 1. Пример описания системы в xUML (из [37])

на элементе модели, принадлежащая классу и характеризующая его поведение после инициализации.

После задания модели в xUML нужно завершить спецификацию, используя конкретный AL. На данный момент существует довольно много готовых AL: OAL [25], SMALL [32], TALL [22] и т. д. Заметим, что достаточно ознакомиться только с одним AL, так как остальные будут иметь тот же семантический набор операций.

Пример спецификации на языке действий OAL [21] можно увидеть ниже (фрагмент взят из [21]). Этот код по своей структуре и внешнему виду похож на обычное текстовое описание и мало напоминает текст на языке программирования.

```
create object instance d of dog;
d.name = "sparky";
create object instance h of doghouse;
h.name = "sparkys house";
relate  d to h across R1;
unrelate d from h across R1;
delete object instance d;
delete object instance h;
select any d from instances of dog where .....
select many dogset from instances of dog;
```

Прокомментируем данный пример. Сначала создаётся (create object instance) объект d класса dog, т. е. в систему добавляется ещё одна собака. Ей даётся имя «sparky» присваиванием атрибуту name объекта d соответствующего значения. Дальше создаётся собачья будка h (экземпляр класса doghouse) с именем «sparkys house». Затем собака и будка связываются (relate d to h) при помощи отношения R1 (across R1), а потом эта связь убирается (unrelate d from h across R1). Объекты d и h удаляются из памяти при помощи оператора delete object instance. В конце приведён пример того, как можно делать различные выборки среди всех объектов, присутствующих в системе. Оператор select any выбирает любой инициализированный объект указанного класса, удовлетворяющий условию, находящемуся в блоке where. В примере, представленном выше, ищется произвольная собака d класса dog, удовлетворяющая некоторому условию. При помощи select many можно получить всё множество объектов, удовлетворяющих определённому условию. В примере после выполнения select many переменная dogset будет содержать все объекты класса dog.

3.2. Анализ

Существует несколько программных средств (например, CASSANDRA/xUML[3], Cameo Simulation Toolkit[4], Telelogic Tau[5], UniMod[6] [1]), позволяющих интерпретировать диаграммы на xUML. Однако этот язык широко использует конструкции текстовых язы-

[3] CASSANDRA/xUML, http://www.knowgravity.com/ger/value/cassandra.htm
[4] Cameo Simulation Toolkit, http://www.magicdraw.com/simulation
[5] Telelogic Tau, http://www-01.ibm.com/software/awdtools/tau/
[6] UniMod, http://unimod.sourceforge.net/index.html

ков программирования (ветвления, циклы и т. п.), поэтому на практике значительного повышения уровня абстракции он не даёт. Поведение системы в любом случае нужно описывать в терминах диаграмм состояний, что бывает совсем не удобно и приводит к весьма громоздким спецификациям.

4. Dynamic Meta Modeling

Другим интересным способом задания семантики визуальных языков является подход под названием Dynamic Meta Modeling (DMM) [17]. Несмотря на то, что приведённое в работе исследование изложено, опираясь на UML, DMM позволяет определять семантику для любых визуальных языков. DMM обладает высоким уровнем формализации, ясен и адекватен при определении семантики интерпретации визуальных языков, основывается на концепции семантики действий (action semantics), совместившей в себе черты как денотационной, так и операционной семантик.

4.1. Описание

Общая схема DMM-подхода изображена на рис. 2. Данный рисунок построен, исходя из предположения, что абстрактный язык программирования должен состоять из трёх компонент: описание синтаксиса (syntax definition), описание семантики (semantics definition, т. е. множества возможных действий), описание семантического соответствия (semantic mapping), которое отвечает за придание определённым синтаксическим конструкциям некого семантического смысла.

Для всех визуальных языков характерно наличие двух уровней: уровень языка и уровень модели (в литературе также встречаются другие термины — уровень метамодели и уровень модели, соответственно). Уровень языка содержит описание синтаксиса и семантики языка. На уровне модели происходит непосредственное создание конкретной спецификации на этом языке, состоящей из элементов метамодели (model elements). На рис. 2 эти уровни помечены буквами L (language) и M (model). Наличие такого разделения подразумевает, что компоненты на уровне модели должны быть согласованы (связь conforms to) с компонентами на уровне

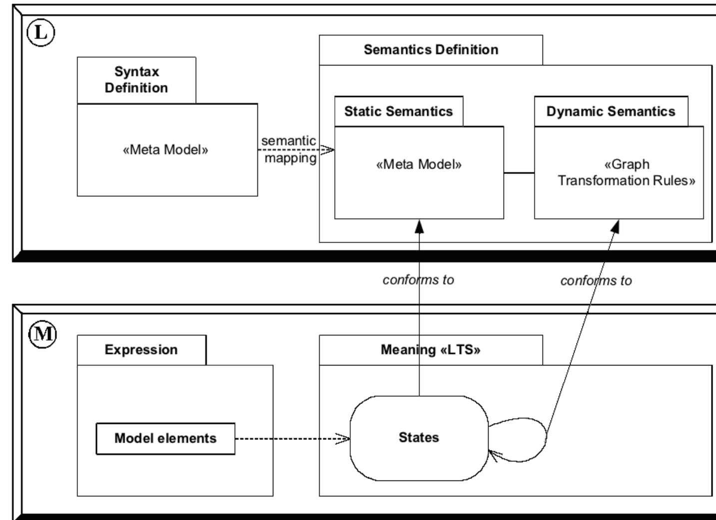


Рис. 2. Общая схема DMM-подхода (этот рисунок и все последующие в этом разделе взяты из работы [17])

языка, т. е. конкретная модель должна удовлетворять синтаксическим правилам языка, конкретная реализация поведения системы должна удовлетворять семантическим языковым правилам и т. п.

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

Для организации семантического соответствия используется концепция мета-отношений (Meta Relations). По своей структуре она значительно сложнее, чем просто соответствие одного синтаксического элемента одному семантическому, так как нужно учитывать вложенные соответствия — см. рис. 3. Допустим, что в синтаксической метамодели языка (Syntax Definition) присутствует агрегирование, и элемент Class агрегирует элемент Attribute. В опреде-

лении семантики (Semantics Definition) элемент Class соответствует элементу Object, а элементу Attribute — Slot. Данная схема предполагает, что каждый атрибут класса соответствует ровно одному слоту соответствующего объекта, а не слоту произвольного другого. Таким образом, соответствие Class/Object в некотором смысле задаёт ограничения на соответствие Attribute/Slot, которое является вложенным в него.

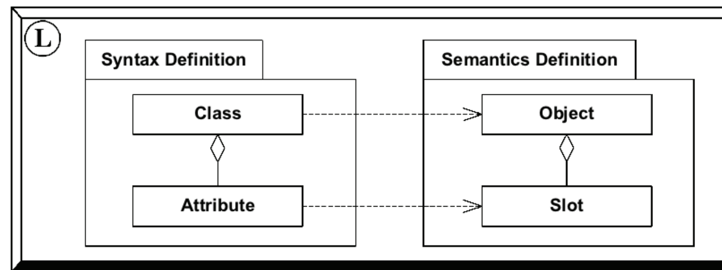


Рис. 3. Пример вложенного соответствия

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

В результате, на уровне модели в качестве реализации подходов, описанных на уровне языка, имеем систему LTS, согласованную с ними, т. е. состояния (states) такой системы соответствуют состояниям, описанным в статической семантике на уровне языка. Правила перехода между этими состояниями являются правилами преобразования графов, которые реализованы на уровне языка

и согласованы со статической семантикой, а также удовлетворяют синтаксическим особенностям создания правил, о которых будет рассказано в следующем разделе.

4.2. Преобразования графов

Модель, созданная средствами DMM, рассматривается в виде типизированного мультиграфа с атрибутами и метками на узлах и рёбрах, допускающего наследование. Множество типов узлов и рёбер этого мультиграфа может быть расширено.

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

Такой поиск подграфа в графе модели очень похож на способ задания ограничений на данные в технологии REAL-IT, описанный в работах А.Иванова [2–4]. Этот способ позволяет в наглядном графическом виде создавать ограничения на модель данных, заданную с помощью диаграмм сущность-связь. Например, часто может возникать ситуация, когда нельзя тот или иной объект соединить связью определённого типа с другим объектом. Для таких случаев можно задать соответствующее ограничение, корректность которого будет впоследствии проверяться.

В DMM существуют также отрицающее применение условия (Negative Application Conditions, NAC) и универсальное замыкание (Universal Quantification, UQ). При использовании NAC к правилу добавляются несколько перечёркнутых элементов, означающих, что правило применимо только в случае отсутствия данных элементов в графе (правильным образом соединённых с остальными элементами искомого подграфа). Наличие же в правиле универсального замыкания означает, что правило будет применено сразу ко всем подходящим местам в исходном графе.

Чтобы такие правила было удобно использовать, для каждого из них задаётся имя, список параметров и, по необходимости, ссылка на базовый элемент, для которого оно должно применяться. Это позволяет внутри одних правил вызывать другие.

На рис. 4 приведена общая сигнатура правила (Rule Signature) и вызовов других правил (Invocation). Сигнатура правила состоит из четырёх частей:

- название правила (rule name);
- индикатор (big-step indicatior, *) того, является ли данное правило правилом большого шага, которое может применяться к исходному графу в любой момент, или правилом малого шага, которое может вызываться только внутри других правил;
- имя базового элемента, для которого можно вызывать данное правило (context node name); на рис. 4 таким базовым элементом является lman типа ListManager;
- список элементов-параметров (parametername:type); на рис. 4 это элемент e типа Element.

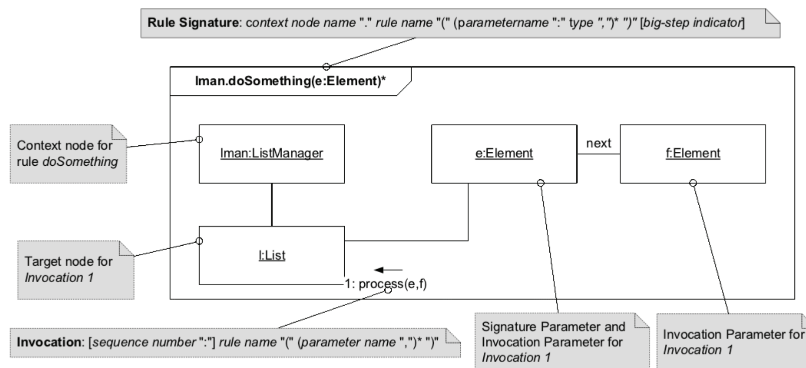


Рис. 4. Сигнатура правила преобразования графов

Сигнатура вызова одного правила внутри другого состоит из трёх частей:

- порядковый номер (sequence number), задающий порядок выполнения этих вызовов внутри данного правила;
- название правила (rule name);
- список элементов-параметров (parametername:type).

На рис. 4 приведён пример такого вызова — process (1:process(e, f)). Внутренний вызов в этом правиле один, поэтому он имеет порядковый номер 1.

4.3. Примеры

Рассмотрим несколько примеров использования DMM. На рис. 5, *а* изображено правило r1. Левая часть этого правила состоит из элементов типа Element и типа Buffer, правая — из этих же элементов, только теперь они соединены связью. Данное правило ищет любую пару элементов типа Element и Buffer и добавляет между ними связь.

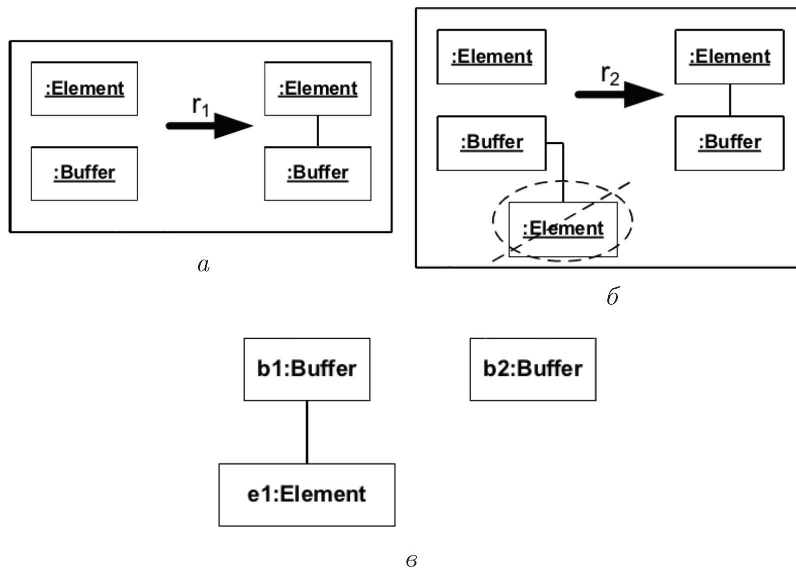


Рис. 5. *а* — правило r1 без использования NAC; *б* — правило r2 с использованием NAC; *в* — исходная модель

На рис. 5, *б* изображено правило r2, правая часть которого совпадает с правой частью правила r1, а левая часть содержит зачёркнутый элемент типа Element, связанный с элементом типа Buffer. Наличие такого перечёркнутого элемента и есть NAC для прави-

ла r2: это означает, что данное правило может быть применено к паре элементов типа Buffer и Element только при условии отсутствия связи элемента типа Buffer с произвольным элементом типа Element.

Модель на рис. 5, *в* состоит из двух буферов, один из которых связан с элементом типа Element. Правило r1 может быть применено к паре (b1, e1) и к паре (b2, e1), тогда как правило r2 — только к паре (b2, e1).

На рис. 6, *а* показано, как при помощи UQ создать правило для полной очистки буфера. Данное правило называется flush и удаляет как все связи элемента типа Buffer с элементами типа Element, так и все эти элементы. Это достигается путём использования в правиле универсальной структуры, на что указывает двойная рамка у элемента с типом Element в левой части правила.

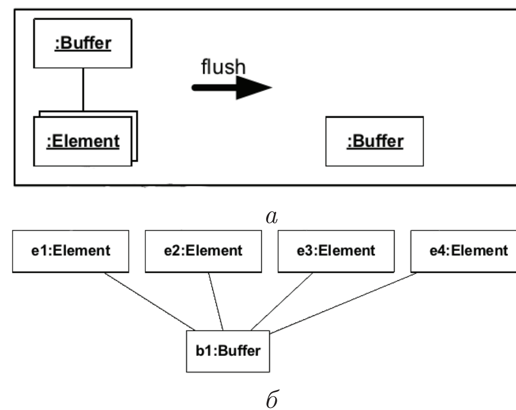
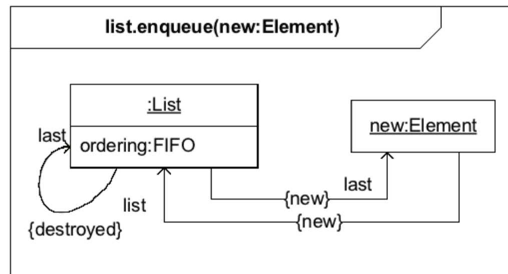


Рис. 6. а — правило очистки буфера с использованием UQS; *б* — исходная модель

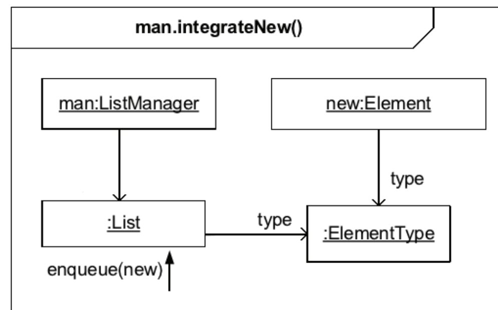
Если применить данное правило к модели, изображённой на рис. 6, *б* и состоящей из буфера b1 и четырёх связанных с ним элементов, то элементы e1, e2, e3, e4 вместе со связями будут удалены и в модели останется всего один буфер.

В качестве примера вызова одного правила из другого рассмотрим следующую ситуацию. Есть набор списков, перечисляющих элементы Element типа ElementType, и сущность ListManager, ко-

торая ими управляет. Тип элемента и тип списка задаются при помощи направленной связи с меткой type (на рис. 7, *а* она показывает тип списка и элемента new).



а



б

Рис. 7. *а* — правило enqueue со стратегией FIFO; *б* — правило integrateNew, осуществляющее поиск нужного списка и вызов другого правила

Правило enqueue, непосредственно присоединяющее новый элемент к списку, может быть реализовано разными способами в зависимости от выбранной стратегии работы со списком списка (например, FIFO или LIFO). На рис. 7, *б* изображено правило enqueue, действующее по стратегии FIFO. Оно также является правилом малого шага, может вызываться только для элемента типа List и принимает в качестве параметра элемент new типа Element. Для того чтобы можно было использовать различные стратегии работы

со списком, у элемента типа List есть атрибут ordering, который в этом случае предполагается равным FIFO. Как видно из рис. 7, *a* данное правило работает следующим образом: оно создаёт связь с меткой last между списком и элементом (и элемент оказывается последним в списке), добавляет обратную связь с меткой list между этим элементом и списком (по этой связи всегда можно найти, какому списку принадлежит элемент), а также удаляет старую связь с меткой last, зацикленную на самом списке, так как изначально список был пустым.

4.4. Анализ

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

DMM потенциально можно использовать и для визуализации процесса отладки диаграмм. Можно организовать настраиваемое пошаговое исполнение визуальной спецификации (т. е. задать, сколько правил следует применить перед следующим обновлением диаграммы), определить 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 могут быть применены на данном шаге, целесообразно 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 [14].

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

Если сравнивать DMM с xUML, то можно выделить следующие существенные отличия: DMM может применяться к любым визуальным языкам, а xUML фиксирован; семантика в DMM задаётся, в основном, визуально, а в xUML важную часть составляет код на

AL; DMM подразумевает, что за ходом интерпретации можно будет наблюдать визуально, следя за изменениями модели, а для xUML эта визуализация будет состоять из подсветки текущего исполняемого элемента модели.

5. EProvide

Данный пакет (полное название Eclipse Plugin for Prototyping Visual Interpreters and Debuggers[7]) является подключаемым модулем (плагином) для среды разработки Eclipse[8]. Он позволяет задавать операционную семантику для предметно-ориентированных языков, основанных на технологиях моделирования Eclipse Modeling Framework [13]. Подход, использующийся в этом плагине, приведен в статьях [29, 35].

5.1. Описание подхода

Для задания правил преобразования визуальных спецификаций (в исполняемый код, в другие модели и т. д.) применяется трансформационный подход (Model-Transformation Approach), например, используя стандарт OMG Query/View/Transformation (QVT)[9]. Набор формально заданных трансформаций позволяет полностью определить проекции (семантику) из одного языка в другой. Часто целевым языком трансформации моделей является язык общего назначения, обладающий возможностью компиляции или интерпретации (например, Java, C++ или Python).

Весь набор способов задания семантики для произвольных языков можно разделить на несколько классов. Одним из основных классов является операционная семантика (ОС)[10]. ОС языка представляет любой конкретный объект, записанный при помощи этого языка, в виде набора вычислительных шагов. Формально её можно представить в виде пары $<\Gamma, \rightarrow>$, где Γ — множество конфигураций, а $\rightarrow$ — бинарное отношение перехода на множестве конфигураций. ОС бывают двух типов: структурная операционная семантика (СОС), предложенная Плоткиным [27], и естественная семантика.

---

[7] EProvide, eprovide.sourceforge.net

[8] Eclipse, www.eclipse.org

[9] QVT, http://www.omg.org/spec/QVT

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

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

Г. Ваксмут (Wachsmuth) предложил использовать СОС, записанную декларативным текстовым способом при помощи языка QVT Relations, являющегося частью стандарта OMG QVT. QVT Relations — это высокоуровневый декларативный язык, разработанный для создания как односторонних, так и двухсторонних преобразований моделей. Позже в работе [29] этот подход был совмещён с технологиями визуального моделирования и представлена утилиты EProvide, позволяющая прототипировать визуальные интерпретаторы и отладчики для предметных языков. Также были добавлены и другие языки для описания семантики: Java, Abstract State Machines (ASM) [15], Prolog и Scheme[11].

Важно отметить, что при интерпретации или отладке происходит изменение состояния модели, которое, во-первых, в конце интерпретации или при ручной остановке нужно возвращать к исходному состоянию, а во-вторых, это состояние через графический интерфейс должен уметь изменять пользователь. Для этого необходимо «вручную» создать отдельную от исходной метамодель, состояние экземпляров элементов которой и будет изменяться. Также туда можно добавить дополнительные структуры данных, необходимые, например, для обеспечения функциональности точек останова. Рассмотрим предлагаемый подход более подробно на примере сетей Петри.

5.2. Редактор и семантика для сетей Петри в EProvide

Сети Петри в простейшем случае можно представить в виде набора позиций с некоторым числом меток (токенов) и переходов, которые

[11] Scheme, http://www.r6rs.org

могут иметь имена. У перехода может быть набор входящих и исходящих позиций.

На рис. 8, *а* изображена метамодель для сетей Петри с двумя элементами: позиция и переход. Сеть (Net) содержит в себе произвольное количество переходов (Transition) и позиций (Place). У перехода и позиции могут быть имена, поэтому соответствующие элементы содержат атрибут name. В сетях Петри у позиции может быть несколько токенов, количество которых хранится в атрибуте token, который, таким образом, должен быть неотрицательным. Переходы содержат наборы входящих (src) и исходящих (snk) позиций.

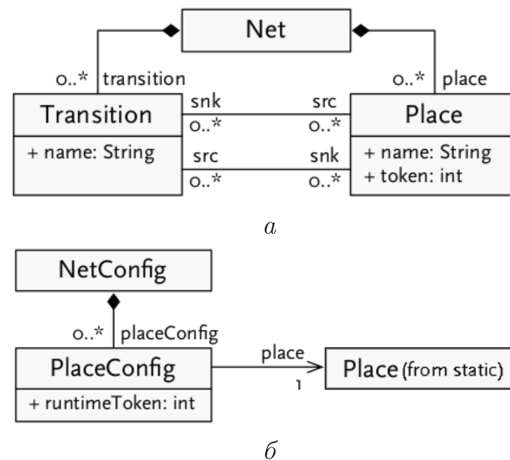


Рис. 8. *а* — общая метамодель сетей Петри (этот рисунок и все последующие в этом разделе взяты из работы [29]); *б* — метамодель времени исполнения для сетей Петри

На рис. 8, *б* представлено расширение общей метамодели для хранения состояний элементов во время исполнения. Конфигурируемая сеть (NetConfig) содержит в себе некоторое число (совпадающее с числом позиций в исходной сети) конфигурируемых позиций (PlaceConfig). Каждой такой позиции соответствует одна и ровно одна позиция из статической метамодели, и изменения происходят

только с ней, т. е. во время исполнения она содержит в атрибуте runtimeToken текущее количество токенов.

Ниже приведен фрагмент семантики сетей Петри, заданный с помощью QVT Relations — трансформация petri_sos.

```
transformation petri_sos(input:petri, output:petri) {

    top relation run {
        trans: Transition;
        checkonly domain input net:Net{};
        enforce domain domain output net:Net{};
        where { trans = getActivated(net); fire(trans); }
    }

    query isActivated(trans: Transition): Boolean {
         trans.src -> forAll(place | place.token > 0)
    }

    query getActivated(net: Net): Transition {
         net.transition -> any(trans | isActivated(trans))
    }

    relation fire {
        checkonly domain input trans:Transition{};
    }

    top relation produce {
        checkonly domain input place:Place{
            src = trans:Transition{}, token = n:Integer{}
        };
        enforce domain output place:Place{ token = n+1 };
        when { fire(trans); trans.src -> excludes(place); }
    }
}
```

Любое преобразование в QVT Relations состоит из запросов (queries) и отношений (relations). Запрос, в отличие от отношения, возвращает результат некоторого OCL выражения [36]. При задании семантики сетей Петри используются два вида запросов: isActivated — проверяет, может ли сработать данный переход или

нет, и getActivated — возвращающий переход, который может сработать. Запрос isActivated возвращает булевское значение и имеет своим аргументом некоторый переход trans. У этого перехода берутся все входящие в него позиции (trans.src) и с помощью цикла forAll проверяется, что число токенов в них положительно. getActivated в качестве параметра принимает всю сеть (net) и возвращает некоторый элемент типа Transition. У сети берутся все переходы (net.transition) и при помощи OCL-предиката any вычисляется произвольный переход, удовлетворяющий условию isActivated, т. е. первый переход, который может сработать.

Отношения определяют новые области (domains) и переменные и позволяют производить некоторые операции над самой моделью. Отношение, имеющие модификатор top, будет применено (hold) для успешного проведения процедуры преобразования целиком. В противном случае отношение производит преобразования над элементами выходной модели, объявленные при помощи модификатора enforce. Также отношение может содержать where- или when-блоки. Where-блок позволяет устанавливать дополнительные ограничения в отношении используемых элементов, причём у этих ограничений может быть указан модификатор enforce. When-блок определяет дополнительные условия на применение отношений. Отношения без модификатора top могут быть применены только внутри where-блока другого отношения.

В заданной нами семантике сетей Петри присутствует одно отношение fire, не объявленное как top. Оно проверяет, присутствует ли переход во входной модели. Отношение run определяет две области. Первую, содержащуюся в input и являющуюся моделью сети Петри, оно связывает с переменной net. Вторую же, содержащуюся в output, оно «насильно» (enforce) связывает с переменной net, показывая таким образом, что результат всех операций, производимых над входной моделью, в ней же и останется. В where-блоке отношение run берёт переход, который может сработать, и применяет к нему отношение fire. Таким образом, отношение run может быть применено тогда и только тогда, когда в модели сети Петри есть переход, который может сработать.

Отношения produce, consume и preserve продолжают начатую с run активацию перехода и соответствуют ситуациям, когда у позиции нужно изменить число токенов (т. е. для тех позиций, которые

непосредственно связаны со сработавшим переходом): увеличить или уменьшить на единицу, оставить неизменённым.

Разберём подробнее отношение produce, которое, так же как и run, объявлено top-отношением. В качестве входных данных это отношение берёт позицию place из области input, переход (src), из которого в неё можно перейти, обозначается как trans, число токенов в позиции — n. После этого в выходной области output (которая в данном случае совпадает со входной) увеличивается число токенов у place на единицу.

Данный набор правил изменяет исходную модель сети Петри, а не модель времени исполнения, поэтому нужно добавить в него пару правил, инициализирующих последнюю. Также можно расширить метамодель времени исполнения точками останова и добавить соответствующие правила в семантику для обеспечения их поддержки (в данной статье не рассматривается). Подробнее ознакомиться с тем, как создать полноценный отлаживаемый и исполняемый визуальный язык для сетей Петри при помощи EProvide, можно в статье [29].

Все метамодели, используемые в EProvide, согласованы со стандартом MOF [23], и работа с ними на уровне редактора моделей осуществляется при помощи технологий EMF и GMF [13]. Модуль, отвечающий за правильное применение преобразований QVT Relations, реализован авторами самостоятельно.

Также была разрешена следующая проблема. GMF не может автоматически генерировать код, который отвечает за работу с моделью времени исполнения, так как для этого нужно расширить множество элементов логического представления (добавить PlaceConfig) и сделать ссылку на графическое представление Place для PlaceConfig. Поэтому в EProvide создаётся объединённая метамодель: в сущность Net добавляется атрибут running, сигнализирующий о том, симулируется ли сейчас данная сеть или нет, а в сущность Place добавляются initToken (начальное значение) и runtimeToken (текущее значение при симуляции). Исходный атрибут token теперь просто ссылается на initToken или runtimeToken, в зависимости от значения Net.running.

Как видно из рис. 9, EProvide позволяет изменять состояние элементов модели прямо в процессе отладки (окно слева), подсвечивает сработавший переход, отображает состояние модели времени ис-

полнения, а не исходной. В конце интерпретации или при «ручной» остановке модель возвращается в исходное состояние. Это происходит автоматически, вследствие разделения на исходную статическую модель и модель времени исполнения, отображаемую пользователю.

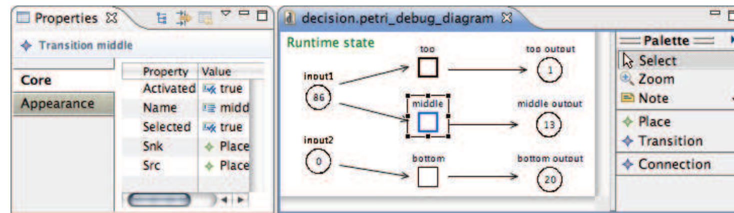


Рис. 9. Графический интерфейс отладчика для сетей Петри

5.3. Анализ

Данный подход является достаточно общим и позволяет работать с различными языками спецификаций, с помощью которых будет происходить непосредственное задание семантики, имеет гибкую позицию относительно реализации точек останова и прочей стандартной функциональности отладчиков, полноценно реализован под платформу Eclipse. Но если смотреть со стороны удобства использования, то спецификация семантики «вручную» на каком-либо текстовом языке понижает достигнутый уровень абстракции. И для успешного применения этого подхода в средах, отличных от Eclipse, необходимо реализовать язык задания семантики, что является нетривиальной задачей.

От xUML EProvide отличается тем, что для задания семантики достаточно лишь описать метамодель языка и способ трансформации моделей, например, при помощи стандарта QVT Relations, и нет необходимости создавать много UML-диаграмм. В EProvide легко можно организовать пошаговое исполнение, совмещённое с поддержкой различного вида точек останова и отслеживанием значений атрибутов элементов. С другой стороны, для xUML, как было отмечено в разделе 3.2, существует множество поддерживающих его программных средств.

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

6. Дискуссия и заключение

Предметно-ориентированное моделирование при определённых условиях может значительно повышать производительность труда разработчиков. Одним из таких условий является адекватная инструментальная поддержка, в которую входит большой набор средств, начиная от визуального редактора и репозитория и заканчивая средствами отладки, версионирования или рефакторинга создаваемых моделей. Для эффективной поддержки предметно-ориентированной парадигмы DSM-платформы должны обладать средствами автоматизированного создания всех необходимых инструментов, «ручное» кодирование даже хотя бы одного инструмента существенно уменьшает эффективность подхода.

Традиционно создание нового языка начинается с анализа предметной области и выделения в ней сущностей и связей между ними, важных для решаемой задачи. Для графических языков эта информация чаще всего задаётся в виде метамодели (модели абстрактного синтаксиса языка) и является достаточной для того, чтобы автоматизировано получить работающий редактор визуальных моделей. Метамодель задаёт правила создания моделей из конструкций языка моделирования, но никак не объясняет, что эти конструкции означают (т. е. метамодель не задаёт семантику языка). Для многих языков моделирования семантика определяется неформально, с помощью описаний на естественном языке, подкреплённых примерами использования. И если для языков моделирования (как, например, UML) это вполне приемлемо, то для языков программи-

рования наличие формально заданной семантики является ключевым. Отсутствие формально заданной семантики приводит к неоднозначности трактовки моделей на этом языке, что делает автоматизированное построение и использование генераторов, интерпретаторов и отладчиков моделей практически невозможным.

В статье были рассмотрены существующие подходы к описанию исполнимой семантики, выделены и описаны некоторые основные методы по заданию семантики визуальных языков.

На основе анализа данных методов можно предложить следующий подход, сочетающий в себе их лучшие стороны. Для задания семантики можно использовать технологию преобразования графов, описанную в DMM. Дополнительные ограничения на применение правил можно записывать при помощи OCL, о котором упоминается в EProvide. Для визуализации потока исполнения можно ввести специальный маркер, являющийся отдельным графическим элементом семантики. Связь с ним означала бы подсветку соответствующего элемента интерпретируемой модели (либо любое другой действие, отличающее текущий исполняемый элемент модели от других).

Из семантики xUML можно взять её текстовую составляющую, а именно, способ задания поведения объектов в целом, а также при проходе каждого состояния в диаграмме состояний. Для этого можно ввести понятие реакции на применение правила, которую можно задавать либо на AL, либо на другом интерпретируемом текстовом языке, обеспечивающем взаимодействие между элементами правила, изменение их атрибутов и т. п.

Сложность реализации данного подхода зависит от целевой платформы. Наиболее существенная и сложная часть в реализации — это модуль, отвечающий за организацию преобразований графов. При использовании Java-технологий для этого можно воспользоваться готовыми средствами AGG [34] или GROOVE [28], однако конвертацию исходной визуальной модели в формат, поддерживаемый данными продуктами, нужно реализовывать самостоятельно. Для вычисления OCL-выражений можно воспользоваться, например, средством USE[12].

[12] USE, http://sourceforge.net/apps/mediawiki/useocl

Список литературы

[1] *Гуров В., Мазин М., Шалыто А.* UniMod — инструментальное средство для автоматного программирования // Научно-технический вестник информационных технологий, механики и оптики. 2006. № 30. С. 32–45.

[2] *Иванов А.* Графический язык описания ограничений на диаграммы классов UML // Программирование. 2004. № 4. С. 204–208.

[3] *Иванов А.* Технологическое решение REAL-IT: Создание информационных систем на основе визуального моделирования // Системное программирование. Под ред. Терехова А. Н., Булычева Д. Ю. Изд-во СПбГУ. 2005. Т. 1. № 0. С. 89–100.

[4] *Иванов А., Стригун С.* Технологическое решение REAL-IT: автоматизированная разработка пользовательского интерфейса информационных систем // Системное программирование. Под ред. Терехова А. Н., Булычева Д. Ю. Изд-во СПбГУ. 2005. Т. 1. № 0. С. 124–147.

[5] *Карташев М.* Двухуровневая схема отладки // Системное программирование. Под ред. Терехова А. Н., Булычева Д. Ю. Изд-во СПбГУ. 2005. Т. 1. № 0. С. 348–365.

[6] *Кознов Д. В.* Визуальное моделирование компонентного программного обеспечения. Автореферат диссертации на соискание ученой степени кандидата физико-математических наук. Санкт-Петербург. 2000. 16 с.

[7] *Кознов Д. В.* Разработка и сопровождение DSM-решений на основе MSF // Системное программирование. Под ред. Терехова А. Н., Булычева Д. Ю. Изд-во СПбГУ. 2008. Т. 3. № 1. С. 80–96.

[8] *Кознов Д. В.* О спецификации диаграммных преобразований в графических редакторах // Вестн. С.-Петерб. ун-та. Серия 10: Прикладная математика. Информатика. Процессы управления. 2011. № 3. С. 100–111.

[9] *Кознов Д. В., Иванов А. Н.* Поддержка концептуального моделирования при разработке визуальных языков с использованием Microsoft DSL Tools // Системное программирование. Под ред. Терехова А. Н., Булычева Д. Ю. Изд-во СПбГУ. 2009. Т. 4. С. 105–127.

[10] *Кознов Д. В., Ольхович Л. Б.* Визуальные языки проектов // Системное программирование. Под ред. Терехова А. Н., Булычева Д. Ю. Изд-во СПбГУ. 2005. Т. 1. С. 148–167.

[11] *Павлинов А., Кознов Д., Перегудов А., Бугайченко Д., Казакова А., Чернятчик Р., Фесенко Т., Иванов А.* Комплекс средств разработки проблемно-ориентированных визуальных языков // Вестник

Санкт-Петербургского Университета. Серия 10: Прикладная математика. Информатика. Процессы управления. 2007. № 2. С. 86.

[12] *Павлинов А., Кознов Д., Перегудов А. и др.* О средствах разработки проблемно-ориентированных визуальных языков // Системное программирование / Вып. 2, под ред. Терехова А. Н., Булычева Д. Ю. СПб.: Изд. СПбГУ. 2006. С. 116–141.

[13] *Сорокин А., Кознов Д.* Обзор Eclipse Modeling Project // Системное программирование / Вып. 5, под ред. Терехова А. Н., Булычева Д. Ю. СПб.: Изд. СПбГУ. 2010. С. 6–31.

[14] *Bandener N.* Visual Interpreter and Debugger for Dynamic Models Based on the Eclipse Platform. Diploma Thesis, 2009, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn. 125 p.

[15] *Borger E., Stark R.* Abstract State Machines: A Method for High-Level System Design and Analysis // Springer-Verlag. 2003. 35 p.

[16] *Boyd G.* Executable UML: Diagrams for the Future. 2003. http://www.devx.com/enterprise/Article/10717

[17] *Hausmann J.* Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD Thesis, 2005, Paderborn, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn. 326 p.

[18] *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM, 12(10). 1969. P. 576–583.

[19] *Kahn G.* Natural Semantics // Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, London. 1987. P. 22–39.

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

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

[22] *Mellor S., Balcer M.* Executable UML: A Foundation for model-driven architecture // Addison Wesley, 2002. 416 p.

[23] Meta Object Facility (MOF) 2.0 Core Specification. OMG. 2006.

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

[25] Object Action Language Reference Manual. 2008. 77 p. www.ooatool.com/docs/OAL08.pdf

[26] OMG Unified Modeling Language, Superstructure. Version 2.4.1. OMG. 2011. 732 p.

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

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

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

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

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

[32] Shlaer-Mellor Action Language. 1997. 29 p. www.modelint.com/downloads/small.pdf

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

[34] *Taentzer G.* AGG: A Graph Transformation Environment for System Modeling and Validation // Proc. Tool Exihibition at Formal Methods'03. 2003. 9 p.

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

[36] *Warmer J., Kleppe A.* The Object Constraint Language: Precise Modeling with UML // Addison- Wesley Object Technology Services, Reading, MA. 1999. 144 p.

[37] *Wilkie I.* Executable UML and SPARK Ada: The Best of Both Worlds // Kennedy Carter Ltd. 2006. 6 p.

[38] *Wilkie I. et al.* The Action Specification Language Reference Manual // Kennedy Carter Ltd. 2003. 112 p.