А.О. Сухов

Национальный исследовательский университет
«Высшая школа экономики»
(Пермский филиал)

ASuhov@hse.ru

СРАВНЕНИЕ ЯЗЫКОВ И ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ ТРАНСФОРМАЦИИ ВИЗУАЛЬНЫХ МОДЕЛЕЙ[2]

Введение

Трансформация моделей – центральная часть модельно-ориентированного подхода к разработке программного обеспечения, поскольку существование в системе одной модели, выполненной с разных точек зрения и, возможно, описанной в нотации различных языков моделирования, требует наличия средств преобразования моделей как между различными уровнями иерархии моделей, так и внутри одного уровня: при переходе от одного языка моделирования к другому для построения единой модели системы, описывающей всю систему в целом.

Использование предметно-ориентированных языков (DSL) и инструментальных средств их создания также затрагивает проблему трансформации, поскольку появляется потребность экспорта созданных пользователем моделей во внешние системы, которые, как правило, используют один из стандартных языков моделирования, отличающийся от разработанного DSL. Именно поэтому система MetaLanguage, предназначенная для создания визуальных динамически настраиваемых предметно-ориентированных языков моделирования [30], должна содержать средства описания и выполнения трансформаций визуальных моделей.

[2] Работа выполнена при поддержке РФФИ (проект № 12-07-00763-а)
© Сухов А.О., 2013

Требования, предъявляемые к компоненту трансформации системы MetaLanguage, определяются назначением данного инструментария и могут быть сформулированы следующим образом:

- описание трансформаций должно производиться с использованием визуальных языков, понятных различным категориям специалистов;
- возможность описания как горизонтальных, так и вертикальных трансформаций;
- возможность описания горизонтальных трансформаций вида «модель-модель» и «модель-текст»;
- возможность использования метаязыка, разработанного пользователем, для спецификации исходной и целевой метамодели (модели языка моделирования);
- возможность описания преобразований атрибутов и ограничений элементов метамоделей.

Существуют различные подходы к трансформации моделей, некоторые из них имеют формальную основу. Так, системы AGG, GReAT, VIATRA используют для выполнения трансформаций правила переписывания графов (graph rewriting), а другие – применяют технологии из других областей программной инженерии, например подход MTBE основывается на методе программирования по образцу.

Трансформация моделей с использованием графовых грамматик

Графовые грамматики достаточно часто применяются для описания каких-либо преобразований, выполняемых над графами: трансформация визуальных моделей [5], задание операционной семантики моделей [3], анализ программных систем с динамически развивающимися структурами [2] и др. Такие грамматики позволяют наглядно описать преобразования, которые должны происходить в системе при выполнении над ней заданных в грамматике операций.

Базовым понятием при описании трансформации графов является *продукционное правило*, которое имеет вид $p: L \to R$, где p – имя правила, L – *левая часть правила*, которая называется *паттерном*, а R – *правая часть правила*, которая называется *графом замены*.

Графовая грамматика – это пара $GG = (P, G_0)$, где P – набор продукционных правил, G_0 – исходный граф грамматики.

Правила применяются к исходному графу, называемому *хостграфом*. Если граф из левой части правила был найден в исходном графе, то правило может быть применено.

Пусть дан исходный граф *G*, а также графы *L* и *R*, которые представляют собой левую и правую части продукционного правила $p: L \to R$, причем граф *L* является подграфом графа *G*. Тогда *применением правила p* к исходному графу *G* называется замена в графе *G* подграфа *L* на граф *R*, результатом замены является граф *H*, причём граф *R* – подграф графа *H* [18].

Графовая трансформация – это последовательное применение к исходному помеченному графу G_0 конечного набора правил

$$P = (p_1, p_2 \dots p_n):\ G_0 \overset{p_1}{\Rightarrow} G_1 \overset{p_2}{\Rightarrow} \dots \overset{p_n}{\Rightarrow} G_n,$$

где G_n – результат выполнения трансформации.

По направлению преобразования трансформации можно классифицировать как вертикальные и горизонтальные. *Вертикальные* трансформации управляют ходом преобразования моделей при переходе от одного уровня иерархии к другому, например при отображении объектов метамодели на объекты модели предметной области (рис. 1). *Горизонтальная трансформация* – это преобразование, при котором исходная и целевая модель принадлежат одному уровню иерархии. Примером горизонтальной трансформации является преобразование описания модели, выполненного на одном языке моделирования, в эквивалентное описание на другом языке.

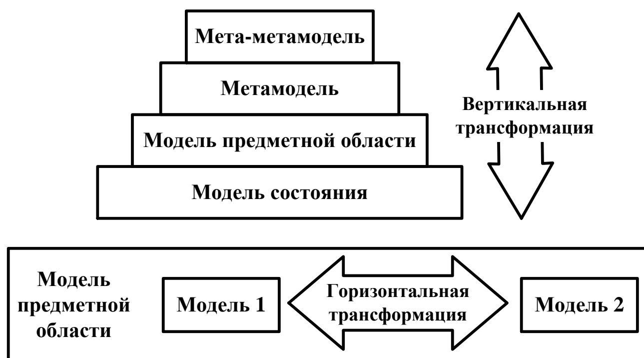

Рис. 1. Вертикальная и горизонтальная трансформации моделей

Для того чтобы выполнить трансформацию моделей необходимо, чтобы они были описаны с использованием некоторых метамоделей. В зависимости от языка, на котором описаны исходная и целевая модель, горизонтальные трансформации можно разделить на два вида:

эндогенные и экзогенные. *Эндогенная трансформация* – это преобразование, при котором исходная и целевая модели описаны на одном языке моделирования. *Экзогенная трансформация* – это преобразование, при котором исходная и целевая модели описаны на различных языках моделирования [25].

Рассмотрим основанные на графовых грамматиках методы трансформации визуальных моделей и реализующие их инструментальные средства.

Attributed Graph Grammar

AGG (Attributed Graph Grammar) – инструментальная среда для описания и выполнения трансформаций типизированных атрибутных графов, реализованная на платформе Java.

Среда AGG состоит из графического интерфейса пользователя, включающего несколько визуальных редакторов, интерпретатора и ряда инструментов проверки допустимости. Визуальные редакторы поддерживают возможность редактирования графов, правил и графовых грамматик.

Благодаря наличию формальной основы – алгебраического подхода к трансформации графов [15], AGG поддерживает возможность проверки допустимости в форме парсинга графа, проверки непротиворечивости графов и правил трансформации.

Граф в AGG представляет собой пару $G = (V, E)$, где V – непустое множество *вершин* графа, E – множество *ребер* графа. Элементы множества, полученного в результате объединения непересекающихся множеств вершин и ребер, называют *объектами* графа.

Следует отметить, что данный инструментарий позволяет проводить кратные ребра, поэтому каждое ребро, как и вершина, имеет уникальный идентификатор.

Каждый элемент графа в AGG имеет определенный тип из заданного множества типов, которое состоит из двух подмножеств: набора типов вершин и набора типов ребер. Создание элементов, не имеющих определенного типа, в AGG запрещено. Тип характеризуется его именем, которое может быть пустым, и графическим представлением, которое определяет форму, цвет, метку вершины или ребра, причем если имена двух типов совпадают, но их графические представления отличны, то соответствующие типы различны.

Описание атрибута в AGG аналогично объявлению переменной в стандартном языке программирования: пользователь должен задать имя и тип атрибута, после чего может присваивать ему любые значения указанного типа. Все элементы атрибутных графов AGG должны иметь ровно одно значение для каждого атрибута. При описании атри-

бутов пользователь может использовать не только простые типы, такие как символы, строки или целые числа, но и любой произвольный класс, описанный на языке Java.

После описания графов моделей пользователю необходимо задать правила их преобразования. Среда AGG поддерживает два способа определения трансформаций: подход одинарного выталкивания (SPO) [16], в этом случае правило состоит из левой (LHS) и правой (RHS) части вместе с частичным морфизмом графа $r: LHS \rightarrow RHS$, и подход двойного выталкивания (DPO) [12], который можно рассматривать как полный морфизм графа с дополнительным графом склеивания *I*, т.е. $r: LHS \leftarrow I \rightarrow RHS$, причем частичные морфизмы графа не обязательно должны быть инъективны.

Правило может содержать отрицательные условия применения правила (NAC), определяющие контекст, в котором правило не должно срабатывать.

LHS правила и NAC в качестве значений атрибутов могут содержать лишь константы и переменные, но не Java-выражения. Это ограничение позволяет устанавливать соответствие атрибутов простым сравнением значений констант или переменных, однако атрибуты RHS правила могут содержать выражения, описанные на языке Java. При описании NAC пользователь может использовать переменные, объявленные в LHS, или переменные, объявленные как входные параметры. Область видимости переменной – это правило, в котором она описана.

В общем случае может быть найдено несколько соответствий LHS в хост-графе либо может не существовать ни одного соответствия. В первом случае должно быть выбрано только одно из найденных соответствий, этот выбор зависит от контекста применения правила, а также от настроек системы: будет ли сделан выбор автоматически или при участии пользователя. В случае, если не было найдено ни одного соответствия, правило не может быть применено к данному графу.

Правила соответствий могут быть не инъективными, тогда несколько объектов левой части правила будут отображены на один единственный объект хост-графа. Однако такие не инъективные соответствия способны вызвать конфликты при удалении и сохранении объектов графа. Эти конфликты могут быть разрешены заданием приоритета операции удаления над операцией сохранения в подходе SPO или неприменимостью правила в подходе DPO, поскольку условие склеивания не будет выполнено.

В результате применения правила $r: LHS \rightarrow RHS$ паттерн, найденный для LHS, будет заменен RHS-частью правила. Соответствие

$m: LHS \rightarrow G$ является полным морфизмом, так как для любого объекта o в LHS найдется его образ $m(o)$ в хост-графе G. Если для объекта o существует образ $r(o)$ в RHS, то его образ $m(o)$ в хост-графе будет сохранен во время преобразования, иначе будет удален. Объекты RHS, не имеющие прообразов в LHS, создаются во время преобразования. Объекты хост-графа, которые не содержатся в соответствии, вообще не участвуют в применении правила, они лишь формируют контекст, который сохраняется во время преобразования. Если условие склеивания установить не удалось, то AGG для трансформации использует подход SPO; в этом случае висячие ссылки будут удалены в результате выполнения преобразования, даже в случае их принадлежности контексту. С другой стороны, если условие склеивания установлено, AGG применяет подход DPO.

Помимо манипулирования вершинами и ребрами графа, правило трансформации может также выполнять вычисление атрибутов, которое не ограничивается применением простых арифметических операций – пользователь может вызывать любые методы, описанные на языке Java.

AGG позволяет пользователю разделять правила по уровням и тем самым определять порядок применения правил. Интерпретатор сначала должен выполнить все правила нулевого уровня, пока это возможно, далее – все правила первого уровня и т.д. Если все правила самого последнего уровня были выполнены, происходит остановка процесса преобразования. Благодаря разделению правил на уровни, появляется возможность задания простого потока управления преобразованием графа.

AGG предоставляет в распоряжение пользователя парсер графа, который способен проверить принадлежность графа языку, определяемому графовой грамматикой. Данная задача эффективно разрешима лишь для ограниченных классов графовых грамматик. Для того чтобы проверить, принадлежит ли граф некоторому языку, необходимо инвертировать все правила грамматики. В результате применения полученных правил в определенном порядке все графы некоторого языка должны быть редуцированы к стартовому графу грамматики.

Однако AGG не может автоматически инвертировать все правила грамматики, поэтому синтаксический анализатор ожидает вместо порождающей грамматики грамматику парсинга, содержащую редуцирующие правила и граф остановки. AGG пытается применить правила парсинга так, чтобы хост-граф был редуцирован к графу остановки. Если это удается сделать, то хост-граф принадлежит языку, определенному грамматикой, и существует последовательность преобразований

хост-графа, приводящая к графу остановки, иначе граф не принадлежит грамматике.

AGG реализует алгоритм парсинга, основанный на переборе с возвратом, при этом парсер создает дерево вывода возможных редукций хост-графа с тупиковыми графами, к которым больше нельзя применить ни одного правила.

Преобразование графа может быть выполнено в двух различных режимах: режиме отладки и режиме интерпретации. Режим отладки позволяет применить выбранное правило только к текущему хост-графу. Соответствующий морфизм может быть определен пользователем. Поскольку задание соответствия «вручную» – достаточно сложная работа, AGG поддерживает возможность автоматического завершения частичных соответствий. Если существует несколько вариантов завершения, то все они будут вычислены и показаны один за другим в редакторе графа. После выбора пользователем соответствия правило будет применено к хост-графу. Режим интерпретации является более сложным и применяется не только к одному правилу, но и к целой последовательности правил. Правило применяются столько раз, сколько это возможно, пока соответствие для другого правила не будет найдено. Следует отметить, что результирующий граф не является единственным возможным, поскольку применение одного правила может повлечь за собой отказ от выполнения другого.

AGG поддерживает возможность обмена графами в формате GXL, разработанном на основе XML. Хост-граф может быть сохранен в GXL, а грамматика графа – в формате XML, который в дальнейшем авторы планируют заменить специальным форматом GTXL [23]. Благодаря такому подходу появляется возможность выполнять трансформации моделей, описанных с помощью некоторого CASE-средства, предварительно сохранив их в формате GXL.

Подводя итог, можно гвоорить о том, что AGG – инструментальное средство описания графовых грамматик, которое поддерживает трансформацию графов. AGG использует для описания исходной и целевой моделей ориентированные типизированные атрибутные графы, что позволяет применять данный инструментарий практически в любой предметной области. Использование в качестве формальной основы алгебраического подхода к трансформации графов позволяет произвести парсинг графа, проверить графовую модель на непротиворечивость.

Расширение возможностей графовых грамматик средствами языка Java позволяет приблизить формальную модель к предметной области и реализовать сложные функции поведения системы.

Преобразование модели задается правилами переписывания графа, которые применяются недетерминировано до тех пор, пока ни одно из них не может быть больше применено. Если необходимо явно указать порядок применения правил, то пользователь может сгруппировать их по уровням. Данное инструментальное средство поддерживает два способа определения трансформации: SPO, DPO.

Преимуществом данного подхода является то, что он был реализован в инструментальном средстве [32], которое содержит визуальные редакторы, интерпретатор, инструменты проверки допустимости, а также текстовый редактор описания Java-выражений.

Поскольку левая и правая части правила трансформации должны быть описаны в одной графической нотации (с помощью одного типизированного графа), то данный инструментарий может быть использован лишь для описания операционной семантики исходной модели, но не для трансформации модели из одной графической нотации в другую. Это является основным ограничением, которое не позволяет использовать данный инструментарий в системе MetaLanguage. Кроме того, в случае существования нескольких вхождений левой части правила в хост-граф, лишь для одного из них правило будет выполнено, хотя в MetaLanguage необходимо выполнить правило для всех вхождений.

Graph Rewriting and Transformation

GReAT (Graph REwriting And Transformation) – язык описания преобразований моделей, базирующийся на тройных графовых грамматиках [7]. Трансформация, описанная на этом языке, представляет собой набор упорядоченных правил переписывания графа, которые применяются к исходной модели и в результате создают целевую модель. Правило – это операция переписывания, представляющая собой паттерн, который необходимо найти в исходной модели, и подграф, который будет создан в результате применения правила. Правила всегда выполняются в некотором контексте, которым является исходный граф.

Данный язык использует модифицированный аппарат графовых грамматик. Грамматика по-прежнему состоит из набора правил, но правило уже не разбивается на левую и правую части, как это предполагается в классических графовых грамматиках. Вместо этого строится один граф, содержащий как левую, так и правую часть правила. Каждый элемент (вершина или ребро) может быть помечен одним из специальных символов, который указывает способ обработки этого элемента: «создание», «удаление» и др.

Язык GReAT включает в себя три языка: язык спецификации ме-

тамоделей, язык описания правил преобразования графа и язык описания потока управления.

Для спецификации исходной и целевой метамоделей используются диаграммы классов UML и ограничения, описанные на языке OCL. UML позволяет определять структурные ограничения, в то время как OCL может быть использован для задания неструктурных ограничений. Таким образом, диаграмма классов UML играет роль графовой грамматики, с помощью которой могут быть созданы метамодели предметных областей. Кроме того, описание метамоделей с помощью UML позволило разработчикам GReAT создать генератор объектно-ориентированного кода. Данный язык предоставляет пользователю возможность определять произвольное число метамоделей, которые могут использоваться в дальнейших преобразованиях.

Процесс создания трансформации в GReAT состоит из нескольких шагов. Сначала пользователю необходимо импортировать описание исходных и целевых метамоделей. Далее следует создать новую метамодель, содержащую все объекты исходных и целевых метамоделей, дополнительные вспомогательные вершины и ребра. После создания единой метамодели пользователь может приступить непосредственно к процессу описания правил преобразования.

Графы, являющиеся формальной основой GReAT, – *типизированные атрибутные мультиграфы*. Графы описываются с помощью диаграмм классов UML, в которых классы и ассоциации являются типами для вершин и рёбер графа, соответственно.

Вершина графа *v* представляет собой тройку (*class*, *id*, *attrs*), где *class* – класс UML, *id* – уникальный идентификатор вершины, *attrs* – отображение, которое каждому атрибуту класса ставит в соответствие некоторое значение.

Ребро e – это пятерка (*assoc*, *id*, *src*, *dst*, *attrs*), где *assoc* – ассоциация или класс ассоциации языка UML, *id* – уникальный идентификатор ребра, *src* и *dst* – инцидентные ребру вершины, *attrs* – отображение, которое каждому атрибуту ассоциации ставит в соответствие некоторое значение.

Граф *G* – это пара (*V*, *E*), где *V* – непустое множество вершин графа, *E* – множество рёбер графа, причём:

$$\forall e = (assoc, id, src, dst, attrs) \in E : src \in V \wedge dst \in V .$$

Соответствие – это пара (*MV*, *ME*), где *MV* – множество привязок вершин, *ME* – множество привязок ребер. Соответствие связывает элементы графа-паттерна и хост-графа. *Привязка вершины* – это пара (*pv*, *hv*), где *pv* – вершина графа-паттерна, *hv* – вершина хост-графа.

Привязка ребра – это пара (*pe*, *he*), где *pe* – ребро графа-паттерна, *he* – ребро хост-графа.

Следует отметить, что вершина графа-паттерна соответствует вершине хост-графа, если они имеют совместимый тип. Кроме того, соответствие является инъективным, т.е. один элемент графа-паттерна отображается ровно на один элемент хост-графа, однако, наборы соответствий не инъективны: один и тот же элемент хост-графа может присутствовать в нескольких соответствиях.

Основной объект преобразования в GReAT – *порождающее правило*, или *продукция*. Продукция содержит:

- граф паттерна, в котором каждый объект имеет свой тип: класс или ассоциация;
- роль;
- входной интерфейс – множество входных портов, на которые поступают объекты графа от предшествующих правил;
- выходной интерфейс – множество выходных портов, через которые объекты графа передаются следующим правилам;
- фильтр – выражение логического типа, которое определяет, должно ли правило быть выполнено для данного подграфа;
- отображение атрибутов – программный код, который выполняется для каждого допустимого соответствия, чтобы установить значения атрибутов вершин и рёбер графа.

Роль в продукции представляет собой отображение вершин и дуг паттерна на ограниченный набор действий, которые могут быть выполнены над ними. Выделяют три вида ролей:

- связывание: объект используется для сопоставления объектов в графе;
- удаление: объект используется для сопоставления объектов, но как только соответствие определено, объект удаляется;
- создание: после того, как соответствие определено, создаётся новый объект.

Если сопоставление прошло успешно, то для каждого соответствия объекты, имеющие роль «удаление» (в визуальном представление изображается в виде «крестика»), будут удалены, после чего объекты с ролью «создание» (в визуальном представление изображается в виде «галочки»), будут созданы. Операция удаления исключает вершину и все инцидентные ей ребра. Во время сопоставления может возникнуть ситуация, когда объект хост-графа уже удален из соответствия, в то время как следующее соответствие все еще содержит привязку для данного объекта, поэтому перед выполнением операции система проверяет существование объекта и, если он был уже удален,

сообщает о невозможности выполнения действия.

Иногда возникает ситуация, когда имеющихся паттернов недостаточно для задания точного соответствия. Так, например, необходимо указать, что целочисленный атрибут некоторой вершины должен находиться в пределах определенного диапазона. Существует также потребность инициализировать значения атрибутов созданных объектов и/или изменять атрибуты существующих объектов. Для этих целей авторы предлагают использовать C++ в качестве языка реализации фильтров и кода отображения атрибутов.

На рис. 2 представлено простое правило преобразования в GReAT [8]. Объекты «House» и «PurchaseOrder» имеют входной интерфейс, это означает, что они получают информацию от предыдущего правила, которое было выполнено. Эти объекты формируют начальный контекст правила для осуществления поиска паттерна, т.е. правило ищет объекты «Room1», «Room2» и «AdjacentTo» в объекте «House». Как только эти элементы найдены, система приступит к проверке фильтра «HasDoor», содержащего процедурный код на C++, позволяющий получить доступ к атрибутам объектов. Если фильтр вернул истинное значение, то будет создан новый объект «OrderItem» в связанном объекте «PurchaseOrder» и будет выполнен процедурный код отображения атрибутов, содержавшийся в блоке «AttributeMapping». В итоге, элементы «House» и «OrderItem» будут переданы следующему правилу через выходные интерфейсы.

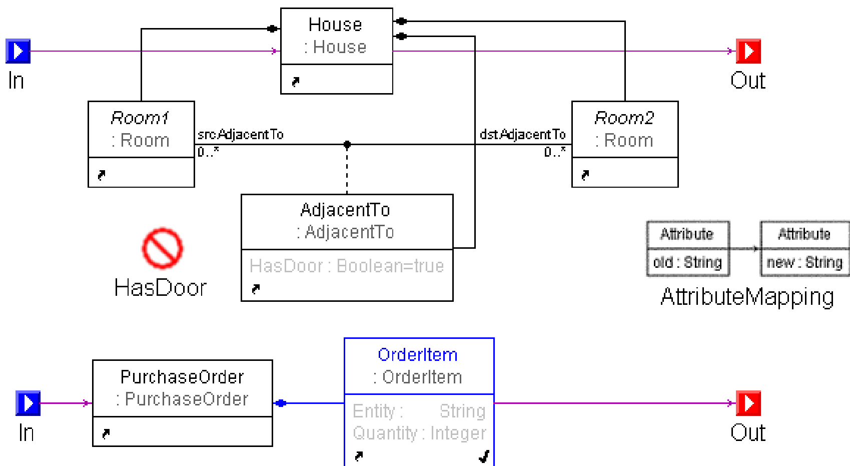

Рис. 2 Правило трансформации в GReAT

Важно отметить, что в GReAT и LHS, и RHS правила преобразования определяются вместе. Объекты, отмеченные как «связывание», можно рассматривать как левую часть правила, а объекты, отмеченные как «создание» или «удаление», – как правую часть правила.

В классических графовых грамматиках на очередном шаге применяется та продукция, которая может быть выполнена. Однако такой подход эффективен лишь для генерации языков, поскольку для модельных трансформаций часто появляется необходимость контроля последовательности срабатывания правил. Для этих целей в GReAT был встроен дополнительный компонент – язык потока управления, который позволяет управлять процессом преобразования моделей. Этот язык включает следующие конструкции потока управления [37]:

1. Упорядочение: если выходной интерфейс правила соединен с входным интерфейсом другого правила, то по окончании выполнения первого правила будет выполнено второе правило. Связь правил предполагает передачу потока пакетов от одного правила другому.
2. Недетерминизм: если правила не соединены последовательно, они могут быть выполнены параллельно, причем порядок срабатывания параллельных правил будет недетерминирован.
3. Иерархия: продукции могут иметь иерархическую структуру, которая создается с помощью конструкции Blocks. Blocks может содержать в себе правила и другие блоки. Есть два вида этой конструкции: Block и ForBlock. Единственное отличие их семантики в том, что каждый входной пакет в ForBlock проходит через все содержащие его правила прежде, чем следующий входной пакет будет подан на вход блоку, в то время как в Block все пакеты, обработанные первым правилом, передаются на вход следующему правилу.
4. Рекурсия: продукционное правило может вызывать себя.
5. Условное ветвление: GReAT предоставляет в распоряжение пользователя специальные блоки Test-Case для разветвления потока управления. Блок Test может иметь несколько блоков Case.

Последовательность выполнения правил завершается, если какое-либо правило не имеет выходного интерфейса, или если правило, имеющее выходной интерфейс, не производит выходных пакетов.

Подводя итог, можно говорить о том, что GReAT является языком описания трансформаций моделей, который использует диаграммы классов UML и язык OCL для представления преобразований. Данный язык позволяет производить трансформацию сразу для нескольких исходных метамоделей [6], что является значительным преимуществом

по сравнению с другими подходами. Язык преобразования состоит из трех подъязыков: языка спецификации метамоделей, языка описания правил трансформации графа и языка потока управления. Язык спецификации предоставляет возможность описывать достаточно сложные метамодели с помощью диаграмм классов UML. Язык описания правил трансформации используется для определения шагов преобразования графов. Язык потока управления позволяет управлять последовательностью срабатывания правил.

К преимуществам данного подхода следует отнести, во-первых, наличие возможности работы с несколькими исходными и целевыми метамоделями одновременно, что позволяет выполнять преобразование моделей, построенных с использованием нескольких языков моделирования; во-вторых, явное упорядочение продукционных правил позволяет управлять процессом их срабатывания, а использование операторов языка потока управления предоставляет возможность императивно описать процесс преобразования моделей.

Однако этот язык описания трансформаций обладает также и ограничениями. Так, в GReAT отсутствует возможность выбора языка спецификации метамоделей и изменения его описания, поэтому пользователю приходится довольствоваться возможностями, предоставляемыми UML и OCL. Кроме того, язык GReAT предъявляет высокие требования к уровню квалификации пользователя, поскольку фильтры описываются на языке OCL, а правила отображения атрибутов – на языке C++. Создание при описании трансформации единого домена, содержащего как левую, так и правую части правила, делает описание правил менее наглядным и более запутанным. Кроме того, отсутствует единая математическая основа данного подхода: так, в источниках [6] и [8] даны различные формальные определения графа, вершины, дуги.

Visual Automated Model Transformations

VIATRA (VIsual Automated model TRAnsformations) – основанный на правилах и паттернах фреймворк преобразования графовых моделей, который комбинирует в единую парадигму спецификации два формализма: первый основан на правилах трансформации графов [19] и используется для описания моделей, второй основан на абстрактных конечных автоматах и предназначен для описания потока управления.

Текущая версия реализации фреймворка [13] поддерживает текстовый язык преобразования VTL, который состоит из трех подъязыков: языка метамоделирования, языка описания трансформаций моделей и языка описания шаблонов кода. Язык метамоделирования (VPM) позволяет выполнять многоуровневое моделирование. Изначально для данного языка был создан лишь абстрактный синтаксис [36], однако в

дальнейшем разработчики предоставили в распоряжение пользователя два представления конкретного синтаксиса: текстовое (VTML) и визуальное. Текстовый формат удобен для автоматической генерации метамоделей, а графическая нотация, используемая для описания небольших метамоделей, является более наглядной для пользователей знакомых с языком MOF.

Язык VPM состоит из двух конструкций: сущность (обобщение пакета, класса и объекта в MOF) и отношение (обобщение ассоциации, атрибута и ссылки в MOF). Сущности описывают базовые понятия исходной и целевой предметной области, в то время как отношения описывают связи между этими понятиями.

Сущности метамодели представляют строгую иерархию включения, которая составляет пространство имен метамодели. В рамках сущности-контейнера каждый элемент метамодели имеет уникальное локальное имя, помимо этого, каждый элемент может также иметь глобальное уникальное имя.

Конструкция «отношение» имеет следующие свойства:

- свойство *isAggregation* указывает, что данное отношение является агрегацией;
- свойство *inverse* указывает на двунаправленный характер отношения;
- свойство *multiplicity* позволяет ввести ограничения на структуру модели.

Между сущностями метамодели могут быть установлены два вида специальных отношений:

- *supertypeOf* (наследование, обобщение) представляет отношение «суперкласс-подкласс» аналогично обобщению в UML;
- *instanceOf* представляет отношение «экземпляр-класс» между уровнями метамодели.

Рассмотрим пример описания метамодели с помощью языка VTML. Пусть дана упрощенная метамодель диаграмм классов языка UML (см. рис. 3). Ее описание на языке VTML будет следующим:

```
entity(UML)
{    entity(Class);
     entity(Association);
     entity(Attribute);
     relation(src, Association, Class);
     relation(dst, Association, Class);
     relation(parent, Class, Class);
     relation(attrs, Class, Attribute);
     multiplicity(attrs, many-to-many)
     relation(type, Attribute, Class);
}
```

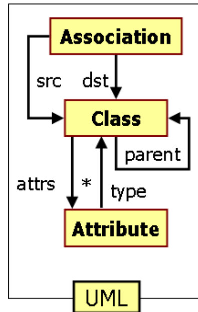

Рис. 3. Упрощенная метамодель диаграмм классов UML

Основные элементы VTML – декларативные конструкции, которые описываются по аналогии с фактами языка Prolog. Сущность должна быть объявлена в виде

Тип (*Имя*).

Если сущность не имеет конкретного типа, то она должна быть экземпляром системного типа *entity*. Поскольку сущности могут содержать в себе другие элементы модели, то разработчиками VIATRA была предусмотрена возможность описания вложенных блоков аналогично тому, как это определено в языке C, т.е. с помощью фигурных скобок ({}). Так, в рассматриваемом примере сущность-контейнер содержит в себе блок описания других объектов метамодели.

Отношение должно быть задано таким же образом, как и сущность, с той лишь разницей, что дополнительно необходимо указать исходную и целевую сущности. Синтаксис описания отношения следующий:

Тип (*Имя*, *Сущность-источник*, *Сущность-приемник*).

Особый вид отношений может быть представлен с помощью ключевых слов *supertypeOf* и *subtypeOf* для отношений специализации, и *typeOf* и *instanceOf* для отношений инстанцирования. Синтаксис задания этих отношений:

Отношение (*Класс*, *Экземпляр*).

Так, например, выражение `typeOf(UML.Class, Dog)` определяет, что сущность `Dog` является экземпляром элемента метамодели `UML.Class`. Благодаря использованию отношений такого вида, элементы модели могут иметь разные типы.

Для описания преобразований моделей VIATRA использует язык задания трансформаций моделей, основанный на паттернах преобразования. Паттерны графа задают условия, налагаемые на контекст моделей, правила преобразования определяют элементарные действия над моделью, а абстрактные конечные автоматы используются для описа-

ния потока управления. Язык, используемый для реализации всех этих операций, – это текстовый язык VTCL. В последующих версиях разработчики планируют реализовать графический редактор, который будет поддерживать возможность визуального описания преобразований моделей.

Паттерны графа – атомарные модули преобразования моделей. Они описывают контекст: условия или ограничения, которые должны быть соблюдены для выполнения определенных операций над моделями. Модель удовлетворяет паттерну графа, если между паттерном и подграфом модели может быть установлено соответствие на основе обобщённого метода сопоставления с паттерном, описанного в работе [34].

В следующем примере представлен паттерн, который может быть применен только для тех экземпляров классов, которые не имеют родительских классов:

```
pattern isTopClass(C) =
{    UML.Class(C);
     neg pattern negCondition(C) =
     {
          UML.Class(C);
          UML.Class.parent(C,CP);
          UML.Class(CP);
     }
     check (name(C)!="")
}
```

Паттерны задаются с помощью ключевого слова *pattern*, они могут иметь параметры, которые должны быть перечислены после имени паттерна, тело паттерна содержит определения сущностей и отношений модели, аналогично тому, как они описываются на языке VTML. Ключевое слово *neg* задает новый паттерн, который содержится в текущем и представляет собой отрицательное условие применения. Если для класса *C* существует родительский класс *CP*, то отрицательное условие будет истинным и сопоставление с паттерном будет неуспешным.

В VTCL паттерн может быть вызван другим паттерном с помощью конструкции *find*, благодаря этому поддерживается возможность повторного использования существующих паттернов.

Трансформации моделей задаются правилами преобразования графа, которые используют паттерны для определения критериев применимости преобразования. При выполнении правила трансформации подграф, представленный в левой части правила (LHS), будет заменен паттерном, представленным в правой части правила (RHS).

Язык VTCL поддерживает два способа задания правил трансформации графа. Первый – традиционная нотация, содержащая два паттерна: паттерн предусловия для LHS и паттерн постусловия, который определяет RHS правила. Элементы, которые присутствуют в LHS, в результате применения правила будут заменены на элементы RHS, а остальные элементы модели останутся неизменными. Такие правила преобразования графов задаются с помощью ключевого слова *gtrule* и могут иметь направленные параметры (in/out/inout). Обмен информацией между паттернами одного правила происходит через передачу параметров.

Второй способ задания правил преобразования в VIATRA – графическая нотация, заимствованная из подхода Fujaba [27]. Правило содержит простой паттерн, который определяет контекст применения правила преобразования графа, и действия, которые должны быть выполнены. Элементы паттерна, отмеченные с помощью ключевого слова *new*, создаются после того, как соответствие для LHS было установлено, а элементы, отмеченные ключевым словом *del*, удаляются после выполнения сопоставления.

Для повторного использования алгоритмов преобразования, не зависимых от конкретной метамодели, VIATRA поддерживает универсальные метапреобразования, которые основаны на применении многоуровневого моделирования.

Управление порядком и режимом выполнения правил преобразования осуществляется командами языка ASM, которые имеют формальную семантику и аналогичны конструкциям языков программирования. Основные элементы программы, описанной на ASM, – правила, аналог методов объектно-ориентированных языков, переменные и функции. Помимо стандартных математических функций, VIATRA предоставляет возможность определять собственные функции, которые пользователь может описать на языке Java. Это предоставляет возможность реализовать сложные вычисления, которые могут использоваться при выполнении преобразований моделей.

Для поддержки трансформации вида «модель-текст» язык VTCL включает конструкцию *print*, которая позволяет получить исходный код всех преобразований на языке Java. Этот код может быть отформатирован с помощью специальных операторов форматирования кода. Для комбинирования в одном файле блоков статичного текста и динамических данных используется понятие шаблона кода. Шаблон – текстовый блок, который может включать ссылки на переменные ASM и какие-либо конструкции.

Подводя итог, можно говорить о том, что VIATRA является под-

ходом к описанию преобразований моделей, который интегрирует в себе графовые трансформации и абстрактные конечные автоматы. Благодаря использованию конструкций ASM разработчикам фреймворка удалось значительно повысить семантику стандартных языков описания паттернов и преобразования графов.

К преимуществам VIATRA следует отнести, во-первых, возможность использования универсальных метапреобразований, что позволяет производить многоуровневое моделирование и повторное использование уже созданных алгоритмов трансформаций. Во-вторых, основанный на шаблонах метод генерации кода для преобразований вида «модель-текст» позволяет генерировать файлы, содержащие как статичные части, так и изменяемые данные. В-третьих, возможность использования как текстовой, так и графической нотации языка метамоделирования предоставляет пользователю возможность выбирать наиболее удобное для него представление языка.

Недостатком подхода является невыразительность и сложность текстового языка описания метамоделей и трансформаций. Хотя разработчики фреймворка критикуют стандарт MOF за отсутствие возможности многоуровневого моделирования [9], они все же остаются в рамках той же парадигмы при использовании визуального языка описания метамоделей, который используется чаще, чем текстовое представление. Данный инструментарий предъявляет высокие требования к уровню квалификации конечного пользователя, поскольку для описания трансформаций пользователь должен не только знать язык метамоделирования MOF, но и уметь строить абстрактные конечные автоматы. Кроме того, у пользователя VIATRA отсутствует возможность выбора языка спецификации метамоделей и трансформаций, так как для этих целей может быть использован только язык MOF.

Query/View/Transformation

QVT (Query/View/Transformation) – стандарт трансформации моделей, предложенный OMG.

Стандарт QVT предоставляет в распоряжение пользователя несколько языков преобразования моделей, все они работают с моделями, метамодели которых описаны с помощью стандарта MOF. QVT для описания преобразований использует язык OCL, расширенный императивными функциями. Ключевыми понятиями этого стандарта являются: запрос, представление, трансформация [24].

Запрос – некоторое выражение, применяемое к модели, результатом выполнения которого является один или несколько объектов этой модели. Формулировка запроса с помощью QVT аналогична построению запросов на языке SQL к реляционной базе данных (БД). Так,

например, запрос к модели диаграмм классов языка UML может быть следующим: «Выбрать все объекты, которые не имеют родительского класса». Чаще всего запросы в QVT используются для задания фильтров. Запросы также могут быть созданы с помощью операционной семантики [33].

Представление – модель, которая получена из другой (основной) модели. Представление не может быть изменено независимо от модели, из которой оно получено. Изменения в основной модели приводят к соответствующим изменениям в представлении. В общем случае метамодель представления отлична от исходной метамодели.

Представления обычно не сохраняются независимо от их исходных моделей и часто доступны только для чтения. Если представление доступно для редактирования, то все произведенные в нем изменения будут отражены в основной модели, поэтому для редактирования представления необходимо иметь определенное его отображение на основную модель. Представление может быть частичным, т.е. основанным на некотором подмножестве исходной модели, или полным, т.е. охватывающим тот же объем информации, что и исходная модель, но реорганизованный для конкретной задачи или пользователя. Следует отметить, что запрос является частным случаем представления.

Трансформация – преобразование, позволяющее получить целевую модель из исходной. Результатом трансформации может быть независимая или зависимая модель. В первом случае нет никакой связи между исходной и целевой моделью с момента, как только последняя была сгенерирована. Во втором случае преобразование связывает исходную и целевую модели.

Преобразования могут быть однонаправленными и двунаправленными. В случае двунаправленных трансформаций пользователь может модифицировать как исходную, так и целевую модель, при этом изменения будут распространены в любом направлении. Если изменения были произведены одновременно в двух моделях, то появляются конфликтные ситуации, которые автоматически разрешены быть не могут, – в этом случае необходимо вмешательство пользователя.

Представление является частным случаем трансформации, в которой целевая модель не может быть изменена независимо от исходной модели. Если представление доступно для редактирования, то соответствующее преобразование должно быть двунаправленным, чтобы существовала возможность изменения исходной модели. Таким образом, базовым элементом стандарта QVT является трансформация, состоящая из набора правил.

Правила – это модули, из которых состоят трансформации. Пра-

вило ответственно за преобразование определенного подмножества элементов исходной модели в соответствующие элементы целевой модели. Правило может содержать описание и/или реализацию. Чисто декларативное правило будет содержать только описание, чисто императивное правило – только реализацию, гибридное правило содержит как декларативные, так и императивные элементы.

Описание является спецификацией отображения между элементами исходной и целевой модели. Описание может содержать достаточно информации для полного определения однонаправленного или двунаправленного преобразования. Альтернативой является использование описания только для определения контекста исходной и целевой модели, в то время как для выполнения самой трансформации вызывается реализация.

Реализация – императивная спецификация действий, позволяющих явно создавать элементы целевой модели из элементов исходной модели. Реализации, как правило, направлены, т.е. они выполняются слева направо или справа налево, однако существует возможность создания реализаций, которые могут работать в любом направлении.

Еще одним понятием QVT является *соответствие*, оно устанавливается во время применения преобразования, если элементы левой и/или правой модели удовлетворяют условиям фильтра правила. Соответствие инициирует создание элементов целевой модели, управляемой описанием и/или реализацией соответствующего правила.

Взаимосвязь между декларативным и императивным уровнем спецификации QVT представлена на рис. 4 [17].

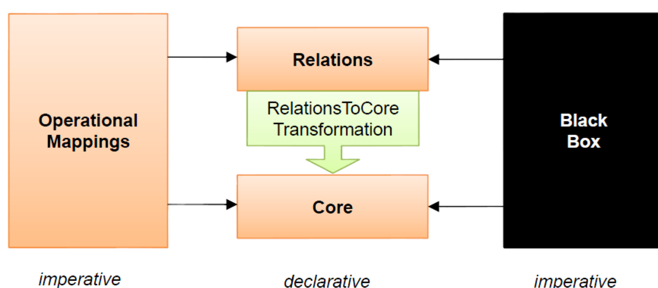

Рис. 4. Связь между языками QVT

Как видно из рисунка, QVT содержит два декларативных языка описания преобразований, которые формируют два уровня абстракции. Есть более абстрактный и более удобный для пользователя *язык отношений* (*Relations*), конструкции которого отображаются на более

конкретный *язык ядра* QVT (Core). В обоих языках для описания паттернов моделей используются выражения OCL.

Язык отношений – декларативный язык, позволяющий описывать однонаправленные и двунаправленные трансформации моделей. Для проверки трансформации на непротиворечивость используется режим выполнения *checkonly*; если набор правил трансформации является непротиворечивым, то система вернет результат «истина», иначе – «ложь». Язык отношений поддерживает сопоставление с паттерном составного объекта и неявно создает классы трассировки и их экземпляры для записи действий, выполненных во время преобразования.

Мощности языка отношений и языка ядра одинаковы, хотя семантика последнего является более простой. В языке ядра модели трассировки определяются явно, а не выводятся из описания преобразований, как в языке отношений. Пользователь может описывать трансформацию непосредственно на языке ядра или на языке отношений, конструкции которого будут в дальнейшем отображены на элементы ядра. Следует отметить, что язык ядра не имеет визуального синтаксиса.

Рассмотрим пример. Опишем с помощью языка отношений двунаправленное преобразование метамодели диаграмм классов UML в метамодель реляционной БД [29]:

```
transformation umlToRdbms
      (uml:SimpleUML, rdbms: SimpleRDBMS)
{  relation ClassToTable
   {  domain uml c:Class
      {  namespace = p:Package {},
         kind='Persistent',
         name=cn
      }
      domain rdbms t:Table
      {  schema = s:Schema {},
         name='t_' + cn,
         column = cl:Column
         {name=cn+'_tid',
           type='NUMBER'
         },
         primaryKey = k:PrimaryKey
         {name=cn+'_pk'
           column=cl
         }
      }
      when { PackageToSchema(p, s);}
      where { AttributeToColumn(c, t);}
   }
}
```

Описание преобразования начинается с ключевого слова *transformation*, за которым следует имя трансформации. Далее следует

описание метамоделей диаграмм классов и реляционной БД, каждое из которых начинается с ключевого слова *domain*. По окончании описания метамоделей необходимо определить связь между ними с помощью разделов *when* и *where*.

В дополнение к декларативным языкам, которые реализуют одну семантику на двух разных уровнях абстракции, QVT поддерживает два способа описания императивных конструкций: стандартный язык – операционные отображения (Operational Mappings), а также нестандартные реализации операций из MOF – чёрный ящик (Black Box).

Язык операционных отображений является стандартным способом описания императивных преобразований моделей. Этот язык использует конструкции OCL, расширенные возможностями процедурного стиля и конкретным синтаксисом. Операционные отображения могут использоваться для реализации одного или нескольких отображений, если их сложно описать с помощью лишь декларативных спецификаций. Преобразования, полностью описанные на языке операционных отображений, называются операционными преобразованиями.

Существование в стандарте QVT *черного ящика* позволяет использовать в процессе задания трансформаций средства преобразования, описанные на других языках, например XSLT или XQuery. Благодаря этому появляется возможность создавать сложные алгоритмы преобразования на языках высокого уровня с привязкой к MOF. Кроме того, такой подход позволяет использовать специализированные библиотеки для вычисления значений свойств элементов модели, например библиотеки с математическими функциями.

Одной из модификаций стандарта QVT являются подходы, описанные в работе [1]. Трансформация представляет собой последовательность правил, применяемых к конкретным элементам модели или совокупности элементов. Правило состоит из *секции выборки*, описывающей контекст правила, и *секции генерации*, определяющей само преобразование. Секция выборки состоит из операторов выборки и уточняющих условий. Секция генерации содержит операторы создания, модификации, удаления элементов модели. При выполнении трансформации создается *трансформационная связь*, которая содержит служебную информацию о сработавших правилах и выполненных операциях создания, модификации, удаления элементов модели. Трансформационная связь используется для поддержания исходной и целевой моделей в согласованном состоянии. Предложенные в работе подходы применяются для описания и выполнения трансформаций моделей, созданных в нотации UML.

Таким образом, QVT – предложенный OMG стандарт трансфор-

мации моделей, предоставляющий в распоряжение пользователя как декларативные, так и императивные языки. Преобразование определяется на уровне метамоделей, описанных с помощью MOF. Достоинством данного подхода является существование стандарта его описания, а также использование в процессе построения преобразований моделей стандартных языков: OCL и MOF. Еще одним преимуществом QVT является широкий набор языков описания трансформаций, которые позволяют использовать как стандартные средства, так и их расширения.

Однако эти преимущества имеют и обратную сторону. Использование MOF в качестве языка метамоделирования не позволяет пользователю выбрать удобный для него метаязык, а также изменить описание метаязыка интегрированного в QVT. Кроме того, данный стандарт не позволяет производить трансформации вида «модель-текст», так как каждая метамодель должна быть описана с помощью стандарта MOF.

Для использования в процессе определения трансформаций внешних преобразований в виде черного ящика необходимо не только иметь хорошо отлаженные библиотеки функций, но и запретить модификацию элементов моделей из этих библиотек напрямую.

Как видно из приведенного примера описания трансформации с помощью QVT, синтаксис языка преобразования моделей достаточно сложен по сравнению с другими подходами. Хотя разработчики стандарта заявляют о поддержке как визуального, так и текстового представления языка описания трансформаций, в большинстве работ авторы предпочитают использовать лишь текстовый конкретный синтаксис, поскольку визуальное представление оказывается еще более сложным для понимания.

Следует также отметить, что декларативный язык ядра никогда не имел полной реализации, поэтому в настоящее время нет ни одного инструментального средства, поддерживающего весь стандарт QVT.

ATLAS Transformation Language

Язык трансформации ATL (ATLAS Transformation Language) [20] базируется на стандарте QVT. ATL разработан как часть платформы AMMA (ATLAS Model Management Architecture) [10]. *Абстрактным* синтаксисом ATL является язык MOF, а в качестве *конкретного* синтаксиса используется текстовый гибридный язык, который интегрирует в себе как декларативные, так и императивные конструкции. Дополнительно существует графический синтаксис, который позволяет частично определить правила трансформации моделей.

По сравнению с QVT, язык ATL имеет *ограничение на создаваемые правила*: он позволяет описывать лишь такие правила, в левой ча-

сти которых находится только один элемент модели – вершина или дуга. Это дает возможность автоматически проверять правильность описанной трансформации, хотя и сильно сужает выразительные возможности языка и область его применения.

Трансформации в ATL описываются в специальных файлах-модулях [21]. *Модуль* содержит обязательный раздел *заголовка*, несколько *помощников* и *правила преобразования*. В разделе заголовка указывается имя модуля преобразования, и объявляются исходные и целевые метамодели. Помощники и правила преобразования – конструкции, используемые для определения функциональной составляющей трансформации.

Помощники используются для навигации по элементам исходной модели и ее атрибутам. Термин «помощник» заимствован из спецификации языка OCL, который содержит два вида помощников: *помощник операции* и *помощник атрибута*. В ATL помощник может быть определен только для OCL-элементов или объектов исходной модели. Основная цель помощника операции – выполнение навигации по исходной модели. Помощники атрибута позволяют производить навигацию по атрибутам элементов исходной модели. Как и у помощников операции, у них есть имя, контекст и тип, отличие лишь в том, что они не могут иметь входных параметров. Помощники атрибута могут использоваться для установления связи между исходными моделями, так как тип помощника атрибута может быть классом метамодели, отличной от исходной метамодели. Существует возможность рекурсивного определения помощников атрибута.

Основной конструкцией языка ATL, используемой для выражения логики трансформации, является *правило преобразования*. Преобразование определяется на уровне метамоделей. Правило преобразования может быть вызвано явно с помощью его имени (императивное правило) или быть выполнено в результате обнаружения соответствия – вхождения паттерна в исходную модель (декларативное правило).

Декларативные правила состоят из двух элементов: исходный и целевой паттерн.

Исходный паттерн представляет собой объединение множества типов, полученных из исходной метамодели, и типов, доступных в языке OCL, с множеством фильтров, налагающих ограничения на исходную модель. *Фильтр* представляет собой логическое выражение и используется для выбора элементов исходной метамодели, удовлетворяющих определенным условиям, описанным на языке OCL. Таким образом, исходный паттерн – это множество элементов исходной метамодели, удовлетворяющих условиям фильтров.

Целевой паттерн состоит из множества типов, полученных из целевой метамодели, и привязок. Привязка определяет выражение, значение которого используется для инициализации некоторого элемента (атрибута, отношения или роли ассоциации). Таким образом, целевой паттерн – множество вершин целевой метамодели, которые могут быть соединены привязкой.

Рассмотрим пример. Опишем правило преобразования понятия «Класс», содержащего один атрибут «Имя», в понятие «Таблица» [21]:

```
rule Класс2Таблица
{
     from
          c: Класс (c.parent.oclIsUndefined())
     to
          t: Таблица (Имя <- c.Имя)
}
```

Как видно из примера, имя правила задается после ключевого слова *rule*. Исходный паттерн правила начинается с ключевого слова *from* и содержит определение переменной типа «Класс». Фильтр задает условие преобразования: только к классам без суперклассов правило может быть применено. Целевой паттерн начинается с ключевого слова *to*, в этом паттерне определены переменная типа «Таблица» и привязка, которая содержит выражение, используемое для инициализации атрибута «Имя».

Более подробное описание примера задания трансформации на языке ATL представлено в работе [4].

По *способу вызова* все правила можно разделить на

- *Стандартные правила*, которые вызываются один раз для каждого соответствия, найденного в исходной модели.
- *Ленивые правила*, вызываемые другими правилами. Они вызываются столько раз, сколько это указано в содержащих их правилах. Это означает, что ленивое правило может быть применено многократно в одном соответствии, каждый раз создавая новый набор целевых элементов.
- *Уникальные ленивые правила*, которые инициируются другими правилами. Они могут быть вызваны только один раз для данного соответствия. Если уникальное ленивое правило вызывается повторно в том же соответствии, то вместо создания новых элементов модели будут использоваться уже существующие.

У декларативного стиля спецификации трансформации есть много преимуществ. Он основан на определении отношений между исходны-

ми и целевыми паттернами и соответствует интуитивному определению трансформации. Этот стиль скрывает детали, связанные с выбором исходных элементов, срабатыванием правил, их упорядочиванием и т.д., поэтому он позволяет скрыть сложность алгоритмов трансформации за простотой синтаксиса языка.

Если в программе ATL используется вызываемое правило или блок операций, то эта программа не является полностью декларативной.

В ATL существует возможность наследования правил, которая может быть применена для повторного использования кода и/или задания полиморфных правил. Также существует возможность определить абстрактное правило, которое не может быть выполнено, но используется в качестве родительского.

Рассмотрим алгоритм выполнения правил трансформации моделей [11]. В первую очередь выполняется правило, помеченное как точка входа (entrypoint). Тело этого правила может содержать произвольное число вызываемых правил соответствия.

При первом вызове каждого правила соответствия создаются и инициализируются элементы целевой модели. Инициализация происходит на основе привязки. Все паттерны каждого правила сопоставляются с исходной моделью, и проверяются фильтры. Если возвращаемое фильтром значение – истина, то паттерн был распознан и правило может быть применено к некоторому подмножеству элементов исходной модели.

В результате выполнения правила преобразования, помимо элементов целевой модели, дополнительно создается ссылка трассировки. Эта ссылка связывает три компонента: правило, исходные и целевые элементы.

Для выполнения правил в ATL используется специальный алгоритм, получивший название алгоритма разрешения. Прежде чем присвоить значение привязки целевой функции, оно должно быть разрешено. Разрешение значения зависит от его типа. Если тип простой, то значение будет просто присвоено соответствующей функции. Если тип значения – класс метамодели, то возможно два варианта:

- значение привязки – целевой элемент, тогда оно будет просто присвоено этому элементу;
- значение привязки – элемент исходной модели, тогда сначала он должен быть преобразован в целевой элемент на основе ссылки трассировки, затем значение должно быть записано в соответствующий элемент целевой модели.

По окончании выполнения всех декларативных правил должно

быть выполнено правило, помеченное как точка выхода (endpoint).

Преобразования в ATL могут быть описаны с помощью конкретного графического синтаксиса, однако визуальный эквивалент есть не для всех конструкций языка, а только для базовых. Так, ни фильтры исходных паттернов, ни привязки целевых паттернов не могут быть представлены графически. Основное назначение этого синтаксиса состоит в том, чтобы сделать использование графических конструкций удобным и наглядным для определения паттернов декларативных правил. Этот синтаксис может использоваться для облегчения понимания преобразования.

Таким образом, ATL является языком описания трансформаций, позволяющим задавать преобразования любой исходной модели в указанную целевую модель. Преобразование производится на уровне метамоделей. Интеграция в рамках одного языка декларативного и императивного подхода позволяет производить преобразования для сложных предметных областей, при этом декларативный подход позволяет скрыть сложность алгоритмов трансформации.

В основе ATL лежит стандартизованный язык описания ограничений OCL, что позволяет использовать в процессе описания трансформаций все преимущества данного языка: стандартные типы, фильтры, помощники и др.

К ограничениям данного языка следует отнести высокие требования, предъявляемые к уровню квалификации разработчика преобразования. Поскольку ATL в большинстве случаев использует лишь текстовое определение трансформации, то, помимо знания исходной и целевой метамоделей, разработчик должен знать сам язык преобразования и язык описания ограничений OCL.

Кроме того, использование императивных конструкций элиминирует преимущества декларативного подхода. Отсутствие навигации по целевой модели затрудняет процесс определения обратных правил преобразования.

Использование языка MOF для спецификации метамоделей ограничивает возможность выбора метаязыка и изменения его описания. В силу этих же причин в ATL отсутствует возможность многоуровневого моделирования.

Трансформация моделей по образцу

Подход к трансформации моделей по образцу (Model Transformation By-Example, MTBE) основан на подходах программирования по образцу и формирования запросов по образцу.

Программирование по образцу [14] позволяет автоматизировать работу конечного пользователя, которая требует навыков разработки программ и знания высокоуровневых языков программирования за счет записи действий пользователя, например с помощью моделей трассировки, и генерации на основе этих моделей исходного кода программы. Это подобно тому, как Microsoft Office позволяет произвести запись взаимодействия пользователя с программным интерфейсом и сгенерировать код для всех произведенных пользователем операций. Полученный таким образом макрос может использоваться для автоматического воспроизведения записанных действий.

Цель формирования запросов по образцу [39] состоит в том, чтобы облегчить работу пользователя при построении запросов к реляционной БД за счет их автоматической генерации. Это достигается путем использования шаблонов таблиц, состоящих из образцов кортежей, заполненных некоторыми константами, ограничениями и командами. Команды описывают операции, которые необходимо произвести над данными (такие как выборка, вставка, удаление кортежей и др.). На основе шаблонов таблиц строятся запросы на языке SQL, которые могут быть выполнены во время работы с БД.

Основная цель MTBE – автоматическая генерация правил трансформации на основе начального набора образцов. Исходные образцы представляются тройками (*SMD*, *TMD*, *MB*), где *SMD* – исходная модель, *TMD* – целевая модель, *MB* – множество отображений, которое каждому элементу из *SMD* ставит в соответствие эквивалентный элемент из *TMD*. Начальный набор образцов определяет различные варианты преобразования моделей. Таким образом, пользователю достаточно описать преобразование на уровне моделей, не касаясь особенностей метамоделей.

Основное преимущество MTBE по сравнению с рассмотренными ранее подходами состоит в том, что при выполнении трансформации используются понятия исходного и целевого языков моделирования для спецификации моделей, в то время как правила трансформации автоматически генерируются на некотором языке, например на ATL [38] или на языке трансформации графов [35].

Процесс автоматической генерации правил трансформации моделей состоит из следующих шагов:

Шаг 1: Построение начального набора образцов. Разработчик преобразования составляет начальный набор взаимосвязанных исходных и целевых пар моделей. Это может быть как единственная модель, которая будет содержать множество всех концептов языка, так и несколько моделей, каждая из которых сосредотачивается на некотором

определенном концепте. Построенные модели, во-первых, должны соответствовать метамоделям языков, с помощью которых они созданы; во-вторых, они должны покрывать множество всех конструкций обоих языков моделирования.

Шаг 2: Автоматическая генерация правил трансформации. MTBE-система создает правила, которые максимально точно преобразуют множество исходных моделей в их целевые эквиваленты. Различные реализации используют для описания правил трансформации различные языки.

Шаг 3: Усовершенствование правил преобразования моделей. После автоматической генерации правил преобразования может потребоваться некоторая дополнительная информация от пользователя для того, чтобы разрешить возникшие конфликты отображения. Разработчик преобразования может модифицировать правила, добавляя дополнительные условия и обобщая существующие правила.

Шаг 4: Автоматическое выполнение правил. Разработчик проверяет корректность сгенерированных правил, выполняя их на тестовых парах моделей. Если результат трансформации неудовлетворителен, то процесс разработки правил может быть запущен вновь. В качестве тестовых моделей авторы работы [38] предлагает использовать исходный набор образцов. В других работах при тестировании предлагается использовать наборы, отличные от исходного.

В работе [35] авторы описывают полуавтоматический процесс генерации правил трансформации моделей, использующий индуктивное логическое программирование [26]. Начальный набор образцов на внутреннем уровне представляется на языке исчисления предикатов первого порядка, что значительно упрощает получение правил преобразования на языке трансформации графов. Для получения результирующего набора правил преобразования часто требуется усовершенствование сгенерированных правил «вручную».

Еще одна реализация MTBE описана в работе [22]. В ней авторы рассматривают проблему трансформации при отсутствии исчерпывающего набора образцов. Авторы сводят проблему трансформации моделей к задаче оптимизации методом роя частиц [28]. Процесс получения правил преобразования становится поиском решения в многомерном пространстве, где каждая размерность представляет определенный элемент модели, а каждая точка в пространстве поиска – возможный вариант трансформации этого элемента.

Поскольку правила трансформации модели заранее не известны, то вариантов преобразования конструкции может быть несколько, степень применимости каждого из которых характеризуется *качеством*

преобразования. Качество преобразования зависит от двух критериев: адекватности преобразования и когерентности преобразования относительно трансформации остальных конструкций. Качество преобразования исходной модели – сумма качеств преобразований каждого из ее элементов, следовательно, поиск оптимального преобразования эквивалентен поиску комбинации преобразований отдельных элементов модели. Однако пространство поиска может быть достаточно большим, если число элементов велико. Эвристический поиск методом роя частиц позволяет сократить это пространство.

Такой подход предоставляет целый ряд преимуществ. Во-первых, для любой исходной модели трансформация производится без генерации правил преобразования. Во-вторых, этот подход не зависит от способа представления исходной и целевой модели. В-третьих, подход не требует никакой дополнительной информации, кроме исходного набора образцов.

Чтобы еще более упростить процесс генерации правил трансформаций моделей авторы работы [31] предлагают новый подход – преобразование модели на основе демонстрации (MTBD). Вместо идеи MTBE получить правила преобразования из начального набора образцов, пользователей просят продемонстрировать, как преобразование модели должно быть выполнено с помощью определенного набора команд, например добавление, удаление элементов, их соединение и др. Эта демонстрация является основой для анализа трансформации моделей и получения паттерна преобразования. Система запоминает все произведенные пользователем операции. Последовательность записанных операций указывает, как должно быть выполнено преобразование. Однако не все операции значимы, так, например, возможно, что пользователь сначала создает некоторый элемент, модифицирует его, а затем понимает, что он является лишним, и удаляет, поэтому после записи демонстрации автоматически выполняется процесс оптимизации, который устраняет все лишние операции.

Данный подход не предполагает генерации правил преобразования моделей, вместо этого он строит паттерн преобразования, описывающий контекст преобразования, т.е. условия его применения, и содержание преобразования, т.е. набор действий, которые должны быть выполнены. После того, как паттерн был получен, система сохраняет его в БД. В процессе работы системы он может быть применен к любой модели, удовлетворяющей контексту.

Таким образом, MTBD позволяет задавать преобразования без необходимости использования какого-либо языка трансформации моделей. Кроме того, описание преобразования не требует от конечного

пользователя знания метамоделей.

Однако этот подход для эффективной работы требует большого количества демонстраций, полученных от пользователя, и его активного участия. Фактически, пользователь должен выбрать подходящий паттерн преобразования. Кроме того, авторы в своей работе не показывают, как именно MTBD может выполнить трансформацию всей исходной модели, а ограничиваются лишь применением паттернов к определенным ее фрагментам. Еще одним существенным ограничением данного подхода является то, что он может быть использован лишь для эндогенных трансформаций.

Таким образом, можно говорить о том, что трансформация модели по образцу является новым подходом, который пытается устранить ряд ограничений традиционных подходов к описанию преобразований моделей, связанных с явным заданием пользователем правил трансформации. Вместо того чтобы описывать правила преобразования «вручную», MTBE позволяет пользователю определить набор взаимосвязанных отображений между исходными и целевыми моделями, а правила преобразования на уровне метамоделей будут сгенерированы автоматически.

Основное преимущество MTBE заключается в том, что для получения правил преобразования пользователю не требуется знание какого-либо языка трансформации моделей, будь то графовые трансформации, ATL и др.

Однако ни один из упомянутых подходов не гарантирует того, что сгенерированные правила преобразования моделей корректны или полны. Более того, полученные правила сильно зависят от исходного набора образцов. Одним из требований, предъявляемых к образцам, является то, что они должны покрывать все множество конструкций языка, однако это нетривиальная задача, особенно в тех случаях, когда язык моделирования достаточно мощный. Также следует отметить, что поскольку правила преобразования генерируются на основе исходного множества пар моделей, то даже небольшие ошибки в определении отображения одной модели на другую могут привести к существенным недочетам в сгенерированных правилах трансформации.

MTBE является итеративным и интерактивным процессом. На практике получить заключительный набор правил трансформации на первой итерации практически невозможно. Это означает, что после автоматической генерации правил преобразования из начального набора образцов, эти образцы должны быть скорректированы или правила преобразования изменены, если пользователь не удовлетворен полученным результатом. Однако в большинстве случаев решение о том,

требуется ли модификация образцов и/или правил, является неочевидным для конечного пользователя – непрофессионального ИТ-специалиста.

Текущие реализации подхода MTBE позволяют выполнять лишь полные эквивалентные отображения, не учитывая сложные преобразования атрибутов, хотя на практике достаточно часто требуется провести преобразование атрибута некоторого элемента исходной модели в соответствующий атрибут элемента целевой модели, выполнив над ними некоторые арифметические или строковые операции, которые описаны на некотором языке трансформации.

Еще одним существенным ограничением большинства реализации MTBE является то, что в них не затрагивается проблема трансформации ограничений, налагаемых на элементы модели.

Выводы

Выше были рассмотрены различные языки и инструментальные средства описания трансформаций визуальных моделей. По результатам анализа можно сделать следующие выводы.

Различные модификации алгебраического подхода используются в AGG, GReAT, VIATRA. В AGG в качестве левой и правой части продукционного правила выступают типизированные атрибутные графы, причем обе части правила должны быть описаны в одной графической нотации, т.е. эта система позволяет выполнять лишь эндогенные преобразования, что делает невозможным ее применение в системе MetaLanguage. Кроме того, данный инструментарий не позволяет производить трансформации вида «модель-текст». Однако использование в качестве формальной основы алгебраического подхода к трансформации графов позволяет произвести парсинг графа, проверить графовую модель на непротиворечивость, а расширение графов возможностями языка Java делает трансформации достаточно мощными с функциональной точки зрения.

Язык GReAT основывается на алгебраическом подходе с двойным выталкиванием, поэтому для описания трансформации необходимо создать домен, содержащий как левую, так и правую часть продукционного правила одновременно с указанием того, какие элементы следует создать, а какие удалить. Такой вид правила является непривычным для конечного пользователя и немного запутанным. Однако такое правило предоставляет возможность выполнять трансформацию сразу для нескольких исходных метамоделей, что является значительным преимуществом по сравнению с другими подходами. Для описания метамоделей GReAT использует UML и OCL, что не позволяет пользо-

вателю выбирать язык спецификации метамоделей, изменять его описание. Это ограничивает применение данного подхода в системе MetaLanguage.

VIATRA – основанный на правилах и паттернах фреймворк, который комбинирует в единую парадигму спецификации два подхода: алгебраический подход для построения моделей и абстрактные конечные автоматы, предназначенные для описания потока управления. Благодаря использованию конструкций конечных автоматов, разработчикам удалось значительно повысить семантику стандартных языков описания паттернов и преобразований графов.

Одним из ограничений VIATRA является невыразительность текстового языка описания метамоделей. Так, для задания отношения между сущностями пользователю необходимо вновь определить сами сущности. Хотя разработчики фреймворка критикуют стандарт MOF за отсутствие возможности многоуровневого моделирования, они все же остаются в рамках той же парадигмы при использовании визуального языка описания метамоделей, который применяется чаще, чем текстовое представление, особенно пользователями, не являющимися профессиональными разработчиками.

Фреймворк VIATRA не предназначен для выполнения горизонтальных трансформаций моделей. Его основное назначение – верификация и валидация построенных моделей путем их трансформации. Стандарт QVT оказался более подходящим для этих целей, поэтому авторы смогли разработать простой и надежный алгоритм верификации. В результате, они отказались от наглядности в пользу надёжности.

QVT – предложенный OMG стандарт трансформации моделей, предоставляющий в распоряжение пользователя как декларативные, так и императивные языки. Преобразование определяется на уровне метамоделей, описанных с помощью MOF. Достоинством данного подхода является существование стандарта его описания, а также использование стандартных языков OCL и MOF в процессе построения преобразований моделей. Еще одним преимуществом QVT является наличие нескольких языков описания трансформаций, которые позволяют использовать как стандартные средства, так и их расширения.

Однако эти преимущества имеют и обратную сторону. Использование MOF в качестве языка метамоделирования не позволяет пользователю выбрать удобный для него метаязык, а также изменить описание метаязыка интегрированного в QVT. Кроме того, данный стандарт не позволяет производить трансформации вида «модель-текст», так как каждая метамодель должна быть описана с помощью MOF. Все это накладывает ряд ограничений на возможность применения стандарта

QVT в системе MetaLanguage.

Синтаксис текстового языка описания трансформаций в QVT достаточно сложен по сравнению с другими подходами. Хотя разработчики стандарта заявляют о поддержке как визуального, так и текстового представления языка описания трансформаций, в большинстве работ авторы предпочитают использовать лишь текстовый язык, поскольку визуальное представление оказывается еще более сложным для понимания.

Следует также отметить, что декларативный язык ядра никогда не имел полной реализации, поэтому в настоящее время нет ни одного инструментального средства, поддерживающего весь стандарт QVT.

ATL – язык, позволяющий описывать трансформации любой исходной модели в указанную целевую модель. Преобразование производится на уровне метамоделей. В основе ATL лежит стандартизованный язык описания ограничений OCL, что делает возможным использование в процессе описания трансформаций все преимущества языка OCL: стандартные типы, фильтры, помощники и др.

К ограничениям данного языка следует отнести высокие требования, предъявляемые к уровню квалификации разработчика преобразования. Поскольку ATL в большинстве случаев использует лишь текстовое определение трансформации, то, помимо знания исходной и целевой метамоделей, разработчик должен знать сам язык описания преобразований. Кроме того, использование императивных конструкций элиминирует преимущества декларативного подхода. Отсутствие навигации по целевой модели затрудняет процесс определения правил преобразования.

ATL является диалектом языка QVT и, как следствие, наследует все его ограничения. Одним из отличий от QVT является очень жесткое ограничение на создаваемые трансформации: в левой части правила в ATL допускается наличие лишь одного элемента. Это сильно усложняет описание правил, увеличивая их количество. Все это делает невозможным использование данного подхода в системе MetaLanguage.

Однако ATL имеет и существенное преимущество по сравнению с другими подходами. В рамках ATL достаточно просто реализуется трансформация ограничений, описанных на OCL. Такие ограничения могут транслироваться в новую модель, оставаясь корректными для неё.

Основная цель MTBE – автоматическая генерация правил трансформации на основе начального набора образцов. Однако ни одна из реализаций данного подхода не гарантирует, что генерация правил

преобразования моделей корректна и полна. Более того, сгенерированные правила преобразования сильно зависят от исходного набора образцов.

Текущие реализации подхода MTBE позволяет выполнять лишь полные эквивалентные отображения атрибутов, не учитывая их сложные преобразования.

Заключение

Подводя итог, можно говорить о том, что все рассмотренные подходы обладают теми или иными недостатками, которые ограничивают их применимость для описания трансформаций в системе MetaLanguage. Наиболее подходящим и перспективным, с точки зрения авторов, является алгебраический подход с одинарным выталкиванием при условии внесения в него следующих расширений:

- наличие возможности многоуровневого описания метамоделей из левой и правой части правила;
- описание правил трансформаций должно производиться на одном уровне иерархии, а их применение – на другом;
- наличие возможности преобразования модели, созданной с использованием одного языка моделирования, в эквивалентную модель, описанную на другом языке;
- наличие возможности описания трансформаций вида «модель-модель» и «модель-текст»;
- наличие возможности трансформации атрибутов элементов метамодели.

Но не следует забывать о преимуществах рассмотренных подходов, некоторые из них способны сделать компонент трансформации моделей системы MetaLanguage достаточно мощным инструментарием.

Библиографический список

1. *Кузнецов М.Б.* Трансформация UML-моделей и ее использование в технологии MDA // Программирование. 2007. Вып. 33. С. 65-79.
2. *Миков А.И., Борисов А.Н.* Графовые грамматики в автономном мобильном компьютинге // Математика программных систем: межвуз. сб. науч. ст. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. Вып. 9. С. 50-59.

3. *Поляков В.А., Брыксин Т.А.* Разработка визуального интерпретатора моделей в системе QReal // Материалы межвуз. конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада «Технологии Miscrosoft в теории и практике программирования». СПб.: Изд-во СПбГПУ, 2011. С. 58.
4. *Ступников С.А., Калиниченко Л.А.* Методы автоматизированного построения трансформаций информационных моделей // Системы и средства информатики. 1. 2009. № 19. С. 34-62.
5. *Сухов А.О., Серый А.П.* Использование графовых грамматик для трансформации моделей // Материалы конференции «CSEDays 2012». Екатеринбург: Изд-во Урал. ун-та, 2012. С. 48-55.
6. *Agrawal A., Karsai G., Neema S., Shi F., Vizhanyo A.* The design of a language for model transformations // Journal on Software and Systems Modeling. 2006. Vol. 5. P. 261-288.
7. *Agrawal A., Karsai G., Shi F.* Graph Transformations on Domain-Specific Models // International Journal on Software and Systems Modeling. Nashville: Vanderbilt University Press, 2003. P. 1-43.
8. *Balasubramanian D., Narayanan A., Buskirk C.P., Karsai G.* The Graph Rewriting and Transformation Language: GReAT // Electronic Communications of the EASST. 2006. Vol. 1. P. 1-8.
9. *Balogh A., Varro D.* Advanced model transformation language constructs in the VIATRA2 framework // ACM Symposium on Applied Computing. New York: ACM New York, 2006. P. 1280-1287.
10. *Bezivin J., Jouault F., Touzet D.* An Introduction to the ATLAS Model Management Architecture // Research Report № 05.01. LINA, 2005. 24 p.
11. *Chiprianov V.*, *Kermarrec Y.*, *Alff P.D.* An Approach for Constructing a Domain Definition Metamodel with ATL // 1st International Workshop on Model Transformation with ATL. Nantes, 2009. P. 18-33.
12. *Corradini A., Montanari U., Rossi F., Ehrig H., Heckel R., Loewe M.* Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach // Handbook of Graph Grammars and Computing by Graph transformation. 1997. Vol. 1. P. 163-246.
13. *Csertan G., Huszerl G., Majzik I., Pap Z., Pataricza A., Varro D.* VIATRA – Visual Automated Transformations for Formal Verifica-

tion and Validation of UML Models // Proceedings of the 17th IEEE international conference on Automated software engineering. Washington: IEEE Computer Society, 2002. P. 267-270.

14. *Edwards J.* Example centric programming // ACM SIGPLAN Notices. 2004. Vol. 39, No. 12. P. 84-91.
15. *Ehrig H., Ehrig K., Prange U., Taentzer G.* Fundamentals of Algebraic Graph Transformation. New York: Springer-Verlag, 2006. 388 p.
16. *Ehrig H., Heckel R., Korff M., Loewe M., Ribeiro L., Wagner A., Corradini A.* Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach // Handbook of Graph Grammars and Computing by Graph transformation. 1997. Vol. 1. P. 247-312.
17. *Gardner T., Griffin C., Koehler J., Hauser R.* A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard // Proceedings of the 1st International Workshop on Metamodeling for MDA. York, 2003. P. 1-20.
18. *Grabska E., Strug B.* Applying Cooperating Distributed Graph Grammars in Computer Aided Design // Parallel Processing and Applied Mathematics. 2006. Vol. 3911/2006. P. 567-574.
19. Handbook on Graph Grammars and Computing by Graph Transformation / ed. by G. Rozenberg. New Jersey: World Scientific Publishing Co., 1999. 677 p.
20. *Jouault F., Allilaire F., Bezivin J., Kurtev I.* ATL: A model transformation tool // Science of Computer Programming. 2008. Vol. 72. P. 31-39.
21. *Jouault F., Kurtev I.* Transforming Models with ATL // Proceedings of the 2005 international conference on Satellite Events at the MoDELS 2005 Conference. Berlin: Springer-Verlag, 2006. P. 128-138.
22. *Kessentini M., Sahraoui H., Boukadoum M.* Model Transformation as an Optimization Problem // Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems. Berlin: Springer-Verlag, 2008. Vol. 5301/2008. P. 159-173.
23. *Lambers L.* A New Version of GTXL: An Exchange Format for Graph Transformation Systems // Electronic Notes in Theoretical Computer Science. 2004. Vol. 127(1). P. 51-63.

24. *Markovic S., Baar T.* Semantics of OCL specified with QVT // Software and Systems Modeling. 2008. Vol. 7, No. 4. P. 399-422.
25. *Mens T., Czarnecki K., Gorp P.V.* A Taxonomy of Model Transformations // Electronic Notes in Theoretical Computer Science. Amsterdam: Elsevier Science Publishers, 2006. Vol. 152. P. 125-142.
26. *Muggleton S., Raedt L.* Inductive logic programming: Theory and methods // Journal Logic Program. 1994. Vol. 19-20. P. 629-679.
27. *Nickel U., Niere J., Ziindorf A.* Tool Demonstration: The FUJABA Environment // Proceedings of the the 22nd International Conference on Software Engineering (ICSE). New York: ACM Press, 2000. P. 1-4.
28. *Sedighizadeh D., Masehian E.* Particle Swarm Optimization Methods, Taxonomy and Applications // International Journal of Computer Theory and Engineering. 2009. Vol. 1, № 5. P. 486-502.
29. *Stevens P.* Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions // Model Driven Engineering Languages and Systems. International Book Series «Lecture Notes in Computer Science». 2007. Vol. 4735/2007. P. 1-15.
30. *Sukhov A.O., Lyadova L.N.* MetaLanguage: a Tool for Creating Visual Domain-Specific Modeling Languages // Proc. of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering. Moscow: ISP RAS, 2012. P. 42-53.
31. *Sun Y., White J., Gray J.* Model Transformation by Demonstration / Model Driven Engineering Languages and Systems. Berlin: Springer-Verlag, 2009. P. 712-726.
32. *Taentzer G.* AGG: A Graph Transformation Environment for Modeling and Validation of Software // Applications of Graph Transformations with Industrial Relevance. International Book Series «Lecture Notes in Computer Science». 2004. Vol. 3062/2004. P. 446-453.
33. *Tratt L.* Model transformations and tool integration // Journal on Software and System Modeling. 2005. Vol. 4. P. 112-122.
34. *Varro D.* Automated Model Transformations for the Analysis of IT Systems: Ph.D. thesis. Budapest: Budapest University of Technology and Economics, 2003. 240 p.
35. *Varro D., Balogh Z.* Automating Model Transformation by Example Using Inductive Logic Programming // Proceedings of the 2007

ACM Symposium on Applied Computing. New York: ACM Press, 2007. P. 978-984.

36. *Varro D., Pataricza A.* VPM: A Visual, Precise and Multilevel Metamodeling Framework for Describing Mathematical Domains and UML // Journal of Software and Systems Modeling. 2003. Vol. 2(3). P. 187-210.
37. *Vizhanyo A.* Improving the Usability of a Graph Transformation Language // Electronic Notes in Theoretical Computer Science. 2006. Vol. 152. P. 207-222.
38. *Wimmer M., Strommer M., Kargl H., Kramler G.* Towards Model Transformation Generation By-Example // Proceedings of the 40th Annual Hawaii International Conference on System Sciences. Washington: IEEE Computer Society, 2007. P. 1-10.
39. *Zloof M.M.* Query-by-example: the invocation and definition of tables and forms // VLDB '75 Proceedings of the 1st International Conference on Very Large Data Bases. New York: ACM Press, 1975. P. 1-24.