Трансформация UML-моделей и ее применение в технологии MDA

М.Б. Кузнецов
mikle.kuz@mtu_net.ru

Аннотация. Модельно-ориентированный подход к разработке ПО позволяет решить проблемы, связанные с постоянно увеличивающимся количеством технологических платформ, а так же может ускорить разработку и интеграцию систем. Но он станет по настоящему эффективным только когда будут разработаны программные инструменты для его поддержки, что в свою очередь требует разработки технологии автоматизированной трансформации моделей ПО. В данной статье рассмотрены различные подходы к разработке такой технологии, а так же предложен язык программирования, предназначенный для описания трансформаций объектных моделей.

1. Введение

На данный момент разработаны и активно используются множество стандартов, middleware-технологий и платформ (CORBA, DCOM, Dot-Net, WebServices, JAVA-технологии и т.д.), и в будущем их количество будет расти. Попытки создать универсальную платформу, которая бы заменила все существующие, привели только к увеличению количества используемых технологий. Всё более важными становятся вопросы выбора технологии для конкретного проекта, осуществления взаимодействия и интеграции разнородных систем и переносе систем на новую технологическую платформу, когда используемая платформа устаревает и уже не может удовлетворить потребности заказчика. Решение может лежать в использовании новой, более совершенной методики разработки ПО.

Model Driven Architecture (MDA) – новый подход к разработке ПО, предложенный консорциумом OMG [9]. Архитектура MDA может предоставить ряд преимуществ по сравнению с существующими методиками: упрощение разработки многоплатформенных систем, простота смены технологической платформы, повышение скорости разработки и качества разрабатываемых программ и многое другое. Однако всё это станет возможным только тогда, когда среды разработки будут поддерживать новую технологию и полностью реализовывать её потенциал. К сожалению, на данный момент большинство коммерческих продуктов, для которых заявлена поддержка MDA, реально не

предоставляют соответствующих инструментов; в связи с этим эффективное применение MDA для решения реальных задач затруднено.

В основе MDA лежат понятия платформо-независимой и платформо-зависимой моделей (platform-independent and platform-specific models, PIMs and PSMs) [10]. В процессе разработки системы сначала создаётся PIM – модель, содержащая бизнес-логику системы без конкретных деталей её реализации, относящихся к какой либо технологической платформе. Принципиальным является именно тот факт, что на этапе создания этой модели не принимается никаких решений по поводу её реализации, разрабатываемый программный продукт не привя-зывается к технологиям. На этом этапе в модель закладывается бизнес-логика, сценарии использования, функциональные требования и другая информация о взаимодействии системы с пользователем и о желаемом поведении системы. При использовании MDA рекомендуется доводить платформо-независимую модель по достаточно высокой степени детализации, вплоть до использования высокоуровневого платформо-независимого языка программирования для описания функциональности и создания исполняемой модели. Однако следует отличать детали функциональности, описывающие поведение системы с точки зрения пользователя, от деталей её практической реализации: последние не должны присутствовать в платформо-независимой модели.

После того, как PIM в достаточной степени детализирована, выполняется переход к платформо-зависимой модели. Эта модель описывает уже не только функциональность системы, но и её реализацию с использованием конкретной (выбранной для данного проекта) технологической платформы. Происходит дальнейшая детализация модели и добавление элементов и конструкций, специфичных для выбранной технологии реализации. После того, как модель достаточно разработана, выполняется генерация кода, затем производится доработка этого кода и его компиляция, так же как и в традиционных методиках разработки ПО.

Разумеется, реально процесс разработки не столь линеен. Для сложного проекта практически невозможно сразу создать платформо-независимую модель, которая бы не потребовала изменений на более поздних стадиях. В процессе разработки платформо-зависимой модели и даже при написании кода может возникнуть необходимость в изменении любой из моделей. Это вполне допускается технологическим процессом MDA, однако необходимо следить, чтобы сохранялось соответствие между моделями: изменения в одной должны быть отображены на другие. Таким образом, при использовании технологии MDA одновременно разрабатываются и изменяются сразу три модели (PIM, PSM и код), представляющие разрабатываемую систему с разных точек зрения и с различными уровнями детализации.

В принципе, идея, положенная в основу MDA, не зависит от инструментов и языка моделирования. Но так как эта технология создана консорциумом OMG, который занимается, помимо прочего, развитием языка моделирования UML [1], предполагается, что именно этот язык будет использоваться для описания

моделей при разработке ПО по технологии MDA. В последних версиях стандарта UML появились дополнения, которые делают этот язык более удобным для такого использования: язык действий (action semantics) [2] позволяет описывать функциональность системы на платформо-независимом уровне, а профили (profiles) облегчают создание платформо-зависимой модели.

Разделение платформо-зависимой и платформо-независимой моделей обеспечивает ряд преимуществ по сравнению с традиционным подходом. Во-первых, облегчается перенос системы на другую платформу и модернизация, так как при этом можно использовать старую платформо-независимую модель и разрабатывать заново только платформо-зависимую. Кроме того, уменьшается риск ошибки проектирования: на относительно простой и полностью основанной на требованиях заказчика (в отличие от традиционной модели, загромождённой деталями реализации) PIM легко находить и исправлять подобные ошибки. Благодаря разделению двух моделей может быть также упрощено создание документации, интеграция систем, создание гетерогенных систем и так далее.

Но главное достоинство подхода MDA состоит в том, что с его помощью можно ускорить разработку ПО, несмотря на то, что создаётся две модели вместо одной. Это достигается за счёт автоматизированной генерации платформо-зависимой модели по платформо-независимой. Процесс перехода к платформо-зависимой модели, основанной на конкретной технологической платформе, в значительной степени формализован. Профили UML, существующие для большинства распространённых технологий, содержат рекомендации по отображению различных конструкций UML в специфичные для выбранной технологии формы – но это рекомендации для разработчика, а не инструкции для автоматического исполнения. Если удастся задать подобное отображение так, чтобы оно выполнялось автоматически, то уже не потребуется создавать платформо-зависимую модель вручную, и процесс разработки существенно ускорится, так как значительное количество элементов, специфичных для технологии реализации будет вноситься в модель автоматически [7]. Кроме того, сократится количество ошибок, неизбежно возникающих при ручном моделировании. Описание перехода к платформо-зависимой модели может быть произведено и тщательно отлажено всего один раз для каждой технологии, и потом может использоваться во всех проектах, использующих данную технологию. Итак, при использовании автоматизированного перехода к платформо-зависимой системе цикл разработки ПО с использованием MDA будет выглядеть следующим образом:

- Постановка задачи, создание сценариев использования и списка формальных требований.
- Создание платформо-независимой модели.
- Автоматизированная трансформация PIM в платформо-зависимую модель, используя ранее разработанное или стандартное описание трансформации для выбранной технологии реализации.

- Модификация и доработка как платформо-зависимой, так и платформо-независимой моделей, добавление необходимых деталей. При этом должно поддерживаться соответствие между моделями.
- Автоматизированная генерация кода по детальной платформо-зависимой модели.
- Ручная доработка кода, компиляция.

Переход от PSM-модели к коду достаточно хорошо разработан, до появления MDA это называлось «генерацией кода по UML-модели» и в большей или меньшей степени поддерживалось всеми средствами для работы с UML и для популярных платформ, языков и технологий. Автоматизированный же переход к PSM – это новая и недостаточно разработанная идея. Так как и PIM, и PSM – это модели, представленные на языке UML, то переход между ними, по сути, является трансформацией UML-модели по заданному описанию трансформации (содержащему формальным образом заданное описание перехода от UML-модели общего вида к конкретной технологии). Именно разработке средства для задания и выполнения подобных трансформаций и посвящена данная работа.

Следует отметить, что не всякий способ задания трансформации моделей может быть эффективно применён в разработке ПО с использованием MDA. Можно сформулировать ряд требований, которым должно соответствовать средство трансформации моделей для его применения в MDA:

- **Формальность и полнота.** С помощью языка описания трансформаций должно быть возможно задать любую необходимую трансформацию для любой платформы. При этом описание должно быть формализованным настолько, чтобы было возможно автоматическое его выполнение.
- **Универсальность правил трансформации**. MDA имеет преимущество по сравнению со стандартными подходами к разработке ПО за счёт того, что для перехода к PSM можно использовать стандартные описания трансформаций. Если бы для каждой системы трансформацию приходилось описывать заново, это оказалось бы даже менее эффективно, чем переход от PIM к PSM вручную. Поэтому необходимо, чтобы язык описания трансформации позволял задавать трансформации, применимые для множества проектов, а не только для одной конкретной UML-модели. Желательно, чтобы в языке были предусмотрены средства для настройки и параметризации трансформации, не требующей кардинальной переделки всего описания трансформации.
- **Поддержка целостности при модификации**. Во время разработки ни одна модель не остаётся неизменной. Это значит, что уже после трансформации как исходные модели, так и модели, созданные в процессе трансформации, могут быть изменены. При этом должен иметься способ поддерживать соответствие между этими моделями, достигнутое в процессе трансформации. Для этого необходимо сохранять информацию о ходе трансформации и о получившихся в

процессе трансформации зависимостях между элементами модели, чтобы специализированный инструмент редактирования UML мог использовать эту информацию для автоматического поддержания соответствия между моделями.

- **Наглядность правил трансформации**. Одно и то же описание трансформации может использоваться во многих проектах. При этом, скорее всего, потребуется его изменение и настройка под специфические требования. Это означает, что описание трансформации должно быть понятно не только создавшему его программисту, но и тому, кто планирует его использовать. Язык описания трансформации должен быть понятным для человека и иметь легко читаемую нотацию. Также желательно, чтобы язык допускал эффективное структурирование описания трансформации, то есть его разбиение на слабо зависимые части. Без подобного структурирования будет очень сложно изменять описание трансформации большого объёма, так как для оценки результата изменения придётся проверять всё описание трансформации, а не только его структурно независимую часть.
- **Взаимосвязанная трансформация нескольких моделей.** При разработке ПО с использованием MDA в рамках одного проекта может иметься несколько UML-моделей (соответствующих разным метамоделям или разным технологическим платформам). Поэтому язык трансформации должен предусматривать наличие более чем одной исходной и генерируемой модели. Если трансформация всех моделей происходит одновременно и по общему описанию, то процесс трансформации проще описывать (по сравнению со случаем, когда для каждой генерируемой модели существует отдельное описание трансформации), так как обычно трансформация моделей одного проекта имеет много общего. Кроме того, при одновременной трансформации проще выявлять соответствие между элементами моделей и генерировать разного рода вспомога-тельные модели (межплатформенные адаптеры, описания интерфейсов и т.д.).

2. Разные подходы к трансформации UML-моделей

Существует несколько способов описания и выполнения трансформаций UML-моделей [5]. Простейший способ – это явное императивное описание процесса трансформации с использованием любого алгоритмического языка. При этом подходе в среду разработки на этапе её создания встраивается набор трансформаций, которые позднее могут быть задействованы пользователем. У этого подхода имеются существенные недостатки, которые делают его малопригодным для использования в средах разработки, поддерживающих MDA. Прежде всего, у пользователя отсутствует возможность добавлять новые описания трансформаций или изменять существующие; он вынужден использовать то, что сделано разработчиками инструмента. Кроме того, из-за

отсутствия единого стандарта описания трансформаций разные среды разработки неминуемо будут выполнять трансформации по-разному даже для одной и той же технологической платформы, что может привести к возникновению случаев несовместимости и затруднит смену сред разработки: вместо зависимости от технологической платформы программист окажется в зависимости от выбранной им среды разработки. И наконец, подобный подход означает, что для каждой среды разработки придётся писать полный набор описаний трансформаций для всех популярных технологий, вместо того чтобы использовать стандартные описания трансформаций, созданные независимыми разработчиками.

Другой подход – использование уже разработанных механизмов трансформаций и преобразований из других областей информатики. В частности, можно представить UML-модель в виде графа и использовать математический аппарат трансформации графов [6]. Главный недостаток такого подхода состоит в том, что в нем используется собственный понятийный аппарат, не имеющий отношения к UML-моделированию. Это значит, что от пользователей такой системы требуется знание не только UML-моделирования, но и теории графов и принципов их трансформации. Кроме того, поскольку UML-модель несёт семантическую нагрузку, отличную от формального графа, правила трансфор-мации, сформулированные для графа, будут трудны для понимания с точки зрения UML-модели: для понимания трансформации придётся мысленно совершать переход от графа к породившей его UML-модели, а для внесения изменений в описание трансформации – от UML к графу.

Ещё один вариант заключается в использовании методик трансформации XML-документов и стандарта XMI [3]. XMI (XML Metadata Interchange) - это стандарт, позволяющий представить UML модель в виде XML документа. Он предназначен главным образом для хранения UML-данных, а так же любых других данных, метамодель которых задана с помощью MOF (Meta Object Facility) [12] и обмена ими между различными инструментами и средами разработки. MOF – это стандарт описания метамоделей, с помощью которого в частности можно описать структуру и общий вид UML-модели. Для XML существует несколько хорошо развитых методик трансформации, в частности XSLT [11] и XQuery [13]. Для трансформации UML-модели можно преобразовать её в XMI-представление, выполнить трансформацию средствами работы с XML, и затем преобразовать результат обратно в UML. Но XMI разрабатывался прежде всего как стандарт хранения и обмена UML-данными, он сложен для чтения и понимания пользователем. Также очень сложно понять функционирование трансформации, описывающей преобразование XML-документа, с точки зрения UML-модели, которой соответствует этот документ. Из-за того, что трансформация описывается в терминах XML, а не UML большая часть описания трансформации оказывается направленной на то, чтобы в результате получить XML-документ, соответствующий стандарту XMI, а не на собственно описание трансформации UML.

Язык моделирования UML считается универсальным, и, конечно же, он содержит средства описания трансформаций. В частности стандарт CWM (Common Warehouse Metamodel) позволяет описывать трансформации и преобразования [7]. Идея использовать UML для описания трансформаций UML-моделей подобно тому, как на UML описывается синтаксис UML, выглядит заманчиво, но трудно реализуема на практике. CWM даёт возможность только описывать сам факт того, что между определёнными элементами модели существует отображение, но не содержит развитых средств для задания трансформаций в общем виде (декларативно или императивно). Ещё один стандарт из семейства UML – QVT (Query, View, Transformation) – представляет значительно больший интерес с точки зрения его применения в MDA. Но, к сожалению, на данный момент этот стандарт находится на ранних этапах разработки. Кроме того, вызывает опасение тот факт, что в рамках этого стандарта предполагается решить сразу несколько общих задач, и то, что QVT ориентирован прежде всего не на практическое применение, а на развитие концепции метаметамоделирования в рамках MOF (Meta Object Facility). Существует вероятность того, что из за подобной направленности на теорию и на решение задач в общем виде этот стандарт будет неудобен для использования на практике в задачах, специфичных для MDA, хотя ни в чём нельзя быть уверенным, пока стандарт ещё не принят хотя бы в ранней версии.

Представляет интерес разработка языка и средства трансформации моделей, предназначенного именно для применения в MDA. Возможно, такое средство трансформации окажется более удобным и эффективным в данной узкой области, чем адаптация более универсального стандарта. Ниже будет рассмотрен один из таких языков трансформации, предлагаемый автором.

3. Принципиальная схема инструмента трансформации

Инструмент трансформации может быть самостоятельным программным продуктом или компонентом среды разработки. При разработке с использованием MDA он предназначен для частичной автоматизации генерации платформо-зависимой модели [4]. Входными данными для него являются:

- Одна или несколько *исходных моделей*
- Метамодель для каждой модели, принимающей участие в трансформации.
- Описание трансформации на определённом языке трансформации. Описание трансформации существенно зависит от используемых метамоделей, но по возможности универсально относительно исходных моделей.

Результатом работы инструмента являются:

- Набор исходных моделей с изменениями, внесёнными в процессе трансформации.

- Одна или несколько новых *сгенерированных моделей*, созданных в процессе трансформации. Наличие и количество таких моделей зависит от используемого описания трансформации. Каждая сгенерированная модель соответствует одной из метамоделей, заданных в качестве исходных данных.
- Информация о связях и отображениях между элементами модели, образованных в процессе трансформации. Такая информация необходима для того, чтобы далее можно было поддерживать соответствие между моделями при их модификации.

С точки зрения инструмента трансформации нет принципиальной разницы между исходными и генерируемыми моделями: и те и другие в процессе трансформации могут подвергаться изменениям и дополнениям. Поэтому в дальнейшем будем говорить о *совокупности моделей*, подвергаемых трансформации, понимая под этим как исходные, так и генерируемые модели.

4. Пример простейшей трансформации платформо-независимой модели в модель, предназначенную для реализации на платформе CORBA

Для того чтобы лучше понять требования к языку описания трансформаций, рассмотрим преобразования, необходимые для перехода от платформо-независимой модели к модели, основанной на технологической платформе CORBA [14]. Пока будем описывать их неформально, на естественном языке. При этом будем стараться задавать эти преобразования в общем виде, не ориентируясь на какую-либо конкретную UML-модель.

- Для каждого класса из PIM следует создать одноимённый класс реализации в PSM и связанный с ним класс-интерфейс.
- Все приватные атрибуты должны быть скопированы в соответствующие классы реализации.
- Публичные атрибуты следует также поместить в класс реализации, но уже как приватные; в интерфейс следует добавить методы для доступа и модификации этих атрибутов.
- Публичные методы следует скопировать в интерфейс, приватные – в класс реализации.
- Ассоциации с множественностью «единица» преобразуются в атрибут-объектную ссылку.
- Связи-обобщения преобразуются в аналогичные связи между классами-интерфейсами генерируемой модели.
- Для интерфейсов исходной модели, в отличие от обычных классов, реализаций создавать не надо.

В результате таких преобразований из платформо-независимой UML модели произвольного вида мы получим модель, которая лучше соответствует объектной модели CORBA и более удобна для дальнейшей реализации, чем

исходная модель. Разумеется, данное описание является только примером и не задаёт всех аспектов преобразования в CORBA-ориентированную модель, в частности не описано преобразование множественных связей и специальных видов ассоциаций. Описывать трансформацию удобно в виде правил – модификаций модели, которые следует выполнить для определённых элементов исходной модели или групп таких элементов. Формальный язык описания трансформации целесообразно наделить такой же структурой, чтобы облегчить его понимание человеком. Далее мы рассмотрим предлагаемый автором язык описания трансформации, основной структурной единицей которого является правило трансформации.

5. Язык описания трансформации

Описание трансформации представляет собой один или более *блоков трансформации*. Каждый блок имеет уникальное имя и состоит из последовательности *правил трансформации*. В заголовке блока может быть также указан параметр, определяющий порядок выполнения этого блока. Ниже приведены фрагменты синтаксической нотации языка трансформации, заданной с помощью расширенных БНФ (форм Бэкуса-Наура).

```
transformation::= <stage>*;
stage::= stage <name> [<sequence>] { <transformation_rule>*
};
sequence::= [reversed] (linear | loop | rollback |
rulebyrule);
```

Каждое правило имеет уникальное имя и состоит из *секции выборки*, определяющей, в каких случаях применимо правило, и *секции генерации*, в которой задаются действия, совершаемые при применении правила.

```
transformation_rule::= rule <name> {
                    <select_section> <generate_section> };
```

Секция выборки состоит из последовательности *операторов выборки.* Каждый оператор выборки объявляет новую переменную, называемую *переменной выборки*. Имя этой переменной должно быть уникальным в пределах данного правила, а область значений – множество элементов модели, задаваемое с помощью *навигационного выражения*. Кроме того, секция выборки может содержать *уточняющие условия* – логические выражения, в котором могут использоваться переменные выборки, объявленные вышестоящими (в рамках правила) операторами выборки.

```
select_section::= (<select_operator>|<constraint>)*;
select_operator::=forall <name> from <nav_expression>;
constraint::= where <condition>;
```

Навигационное выражение – это последовательность *направлений навигации*, начинающаяся с имени ранее объявленной переменной (назовём её *базовой переменной*), в качестве разделителя используется символ «/». Направление

навигации – это имя ассоциации в метамодели UML, соответствующей трансформируемой UML-модели (если в трансформации участвует несколько моделей с разными метамоделями, то метамодель определяется по тому, на элемент какой модели указывает начальная переменная).

```
nav_expression::=<name> iteration_pair(/, <nav_direction>);
```

При вычислении навигационного выражения происходит последовательный переход от одного UML-элемента к другому по ассоциации метамодели, соответствующей очередному направлению навигации, начиная со значения базовой переменной. Под кардинальностью направления навигации будем понимать множественность (multiplicity) соответствующей ассоциации метамодели. Если кардинальность очередного направления навигации больше единицы и существует неоднозначность в выборе элемента модели для перехода, то переход осуществляется по всем вариантам, то есть в результат включаются все подходящие элементы. Формально результат вычисления навигационного выражения можно описать с помощью индукции:

- для выражения, состоящего только из имени переменной, результатом является значение этой переменной;
- процесс вычисления разобьём на шаги: первый шаг – выражение, состоящее только из базовой переменной; на каждом следующем шаге будем добавлять к вычисляемому выражению очередное (слева направо в исходном выражении) направление навигации;
- результатом вычисления очередного шага будет множество элементов модели, достижимых хотя бы из одного элемента, полученного на предыдущем этапе, переходом по текущему (то есть последнему добавленному) направлению навигации;
- если исходное выражение содержит последовательность из N направлений навигации, то на (N+1) шаге будет получен результат вычисления этого выражения.

Таким образом, результатом вычисления навигационного выражения является список из всех возможных элементов модели, которых можно достичь из начального элемента, определяемого значением базовой переменной, по заданным направлениям перехода. У навигационного выражения можно выделить следующие характеристики:

- **Тип** – элемент метамодели, соответствующий элементам модели, входящим в результат. Так как навигация задаётся с помощью метамодели, все элементы результата вычисления выражения имеют общий тип. Тип выражения определяется статически (то есть не зависит от конкретной модели, на которой вычисляется выражение).
- **Кардинальность** – максимальное число элементов в результате. Кардинальность выражения равна 1, если кардинальность всех направлений навигации равна 1. Кардинальность определяется статически, по метамодели.

- **Значение** – множество элементов модели, являющееся результатом вычисления выражения.

Навигационное выражение может начинаться с любой ранее объявленной локальной или глобальной переменной. Под локальными переменными понимаются переменные, объявленные в исполняемом на данный момент правиле. Разумеется, в каждом конкретном операторе допустимо использовать только локальные переменные, объявленные в вышестоящих (в описании правила) операторах: так как выполнение правила происходит сверху вниз, локальные переменные, порождённые нижестоящими операторами, на момент выполнения текущего оператора ещё не инициализированы. Глобальными переменными являются имена моделей, участвующих в трансформации. Очевидно, что навигационное выражение самого первого оператора выборки каждого правила может начинаться только с глобальной переменной, так как ни одна локальная переменная ещё не объявлена.

Секция генерации – это последовательность операторов создания, изменения или удаления элементов модели. Оператор создания элемента позволяет добавить новый элемент модели в последовательность элементов, оператор удаления – исключить элемент из списка. Соответствующие навигационные выражения указывают на изменяемый список и определяют тип добавляемого элемента. Кардинальность навигационного выражения должна быть больше 1. Оператор изменения позволяет менять значение того или иного поля данных или атрибута; при этом навигационное выражение указывает на изменяемый элемент (и кардинальность выражения должна быть равна единице). Также существуют операции добавления в список уже существующего элемента и исключения элемента из списка; они предназначены для использования в тех случаях, когда в метамодели имеются циклы или один и тот же метаэлемент доступен сразу по нескольким ассоциациям: в таких случаях для формирования модели может оказаться недостаточно операций создания и удаления элементов.

```
generate_section::=
 (
 <create_operator> | <update_operator> | <delete_operator> |
 <include_operator> | <exclude_operator> )*;
create_operator::= make <name> in <nav_expression>; ;
update_operator::= <nav_expression> = <expression>; ;
delete_operator::= delete <nav_expression>; ;
 include_operator::= include <name> in <nav_expression>; ;
exclude_operator::= exclude <name> in <nav_expression>; ;
```

6. Выполнение трансформации

Теперь, когда определён синтаксис языка трансформации, опишем процедуру выполнения трансформации. Инструмент трансформации, получив в качестве входных данных исходную модель (или несколько моделей) и описание трансформации, а также имея информацию об используемой метамодели,

может выполнить заданную трансформацию. При этом, в зависимости от описания трансформации, возможно как изменение исходной модели, так и создание новой.

Выполнение трансформации заключается в последовательном применении бло-ков трансформации. Выполнение блока состоит в нахождении в этом блоке правила, которое может быть применено в данный момент, и применении этого правила.

Применимость правила определяется по его секции выборки.

Каждый оператор выборки объявляет новую переменную и с помощью навигационного выражения определяет множество возможных значений этой переменной на данной модели. Секция выборки может также содержать уточняющие условия; конъюнкцию всех таких условий назовём *обобщённым уточняющим условием*. Множество возможных выборок – это декартово произведение множеств значений переменных выборки, ограниченное уточняющим условием, то есть множество всевозможных наборов значений переменных, для которых обобщённое уточняющее условие истинно. Выборкой назовём произвольный элемент этого множества. Правило применимо, если для текущего состояния трансформируемой модели (совокупности моделей) существует такая выборка, для которой это правило не было применено ранее.

Применение правила к выборке состоит в последовательном выполнении всех операторов секции генерации. При этом значениями переменных выборки являются соответствующие компоненты выборки, для которой применяется правило. Каждый раз, когда применяется правило, создаётся новый экземпляр трансформационной связи, порождённой этим правилом. (О том, что такое трансформационная связь, будет сказано позже.)

Если правило применимо более чем к одной выборке, то для выполнения случайным образом выбирается одна из них (вообще говоря случайным образом, в зависимости от реализации). Далее, в зависимости от порядка применения правил в блоке, правило может быть применено к другим выборкам, или может начаться выполнение другого правила.

Порядок применения правил определяется заданной при создании описания трансформации *последовательностью применения правил* в блоке. Она задаётся с помощью параметра <sequence> в заголовке блока. Если значением параметра является “linear”, то блок выполняется линейно сверху вниз, то есть поиск следующего применимого правила осуществляется в порядке объявления правил в описании трансформации, начиная с последнего выполненного правила. Когда процесс доходит до последнего правила в блоке, выполнение блока завершается. При значении параметра “loop” процесс выполнения также происходит сверху вниз по описанию трансформации, но после достижения последнего правила выполнение блока не заканчивается, а продолжается с первого правила в описании блока. Если значением параметра является“rollback”, то после каждого применения правила поиск следующего

применимого правила начинается с первого правила в описании блока, а не с последнего применённого правила, как в предыдущих вариантах. При значении параметра “rulebyrule” поиск применимого правила также начинается с первого правила в описании блока, но после нахождения применимого правила оно выполняется для всех возможных значений выборки, и только после того, как все возможности применения этого правила исчерпаны, происходит поиск другого применимого правила (начиная с первого правила в описании блока). Также возможно применение описанных выше параметров совместно с ключевым словом “reversed”. В этом случае процесс поиска применимого правила происходит не в порядке объявления правил в описании блока, а в обратном порядке. Значение параметра <sequence> по умолчанию – “rulebyrule”, так как такая последовательность выполнения правил приводит к наиболее быстрому завершению трансформации.

Если ни одно правило из блока трансформации больше не может быть применено, или если достигнут конец описания блока при линейном (“linear”) порядке выполнения, то этот блок трансформации считается завершённым и начинает выполняться следующий. Трансформация считается завершенной, когда выполнен последний блок.

Следует отметить, что для произвольного описания трансформации не гарантируется завершение процесса трансформации для любой модели; при использовании правил с операторами создания элемента модели возможны бесконечные циклы. Это необходимо учитывать как при создании описания трансформации, так и при его использовании.

7. Трансформационная связь и её использование в описании трансформации

Каждый раз, когда выполняется то или иное правило, помимо выполнения действий, описанных в секции генерации, создаётся специальная структура данных, называемая трансформационной связью. Имя этой структуры совпадает с именем правила, а полями данных являются локальные переменные, объявленные в описании правила (в операторах выборки и операторах создания). Значения этих полей данных совпадают со значениями соответствующих переменных, определённых в процессе выполнения правил.

Трансформационные связи, образованные в процессе трансформации, объединяются в *совокупность трансформационных связей*. Она содержит информацию о ходе трансформации, применённых правилах и о том, как и почему был изменён или создан тот или иной элемент модели. Эту информацию после выполнения трансформации можно использовать для поддержания соответствия между моделями при их модификации.

Совокупность трансформационных связей может использоваться не только после завершения трансформации, но и в процессе ее выполнения. В операторах выборки, заданных в описании трансформации, вместо навигации

по модели возможна навигация по трансформационным связям, порождённым ранее применёнными правилами. Для этого в операторе выборки используется навигационное выражение особой структуры. В отличие от обычного навигационного выражения, навигация по совокупности трансформационных связей начинается не с локальной переменной или имени модели, а с имени блока трансформации. После имени блока следует имя правила и имя локальной переменной из этого правила. Так как любое правило в процессе трансформации может быть применено несколько раз, и, соответственно, в совокупности связей может присутствовать несколько экземпляров соответствующей трансформационной связи, кардинальность подобного навигационного выражения всегда больше единицы.

```
<special_nav_expression>::=<block_name>/<rule_name>/<variable_name>;
```

Вместо имени блока можно использовать ключевое слово “rules”, которое эквивалентно имени текущего блока. Следует отметить, что, поскольку блоки трансформации исполняются последовательно, в каждом блоке возможно использование информации о трансформационных связях данного блока и блоков, стоящих в описании трансформации перед ним. Блоки, располагающиеся в описании трансформации после текущего, ещё не выполнены и не имеют трансформационных связей.

Использование трансформационных связей в описании трансформации позволяет создавать зависимости между применением правил и более точно определять, в каком случае должно быть применено то или иное правило. Это, в свою очередь, позволяет описывать нетривиальные преобразования моделей в компактной и простой для понимания форме.

8. Механизмы уточнений правил и шаблоны

Для улучшения структуры языка трансформации в него добавлено понятие уточнения правил. При описании любого правила можно объявить, что оно *уточняет* одно из ранее объявленных правил в рамках того же блока трансформации. Для этого в заголовке правила после его имени указывается имя уточняемого правила. Возможно создание многоярусных древовидных иерархий за счёт дальнейшего уточнения уточняющих правил и создания нескольких уточняющих правил для одного уточняемого. При написании операторов выборки уточняющего правила помимо переменных, объявленных в предыдущих операторах данного правила могут использоваться все переменные выборки из уточняемого правила, а при написании секции генерации – все переменные секций выборки и генерации уточняемого правила.

```
transformation_rule::= rule <name> [ : <name>] {
        <select_section> ; <generate_section> };
```

Уточняющее правило применимо в тех случаях, когда выполняются все условия из следующего списка:

- применимо уточняемое правило;
- найдена выборка, удовлетворяющая секции выборки уточняющего правила;
- это уточняющее правило ранее не применялось к данному применению уточняемого правила и данной выборке.

Применение уточняющего правила заключается в выполнении операторов секции генерации. После выполнения секции генерации происходит замена трансформационной связи, порождённой применением уточняемого правила, на расширенную трансформационную связь, включающую как старые переменные, так и переменные, объявленные уточняющим правилом. Трансформационная связь при этом именно расширяется за счёт добавления новых полей, старые же поля сохраняются, по аналогии с классическим механизмом наследования. Такая трансформационная связь будет соответствовать как выборкам из совокупности трансформационных связей, нацеленных (с помощью навигационного выражения) на связи, порождённые уточняемым правилом, так и выборкам из связей, порождённых уточняющим правилом.

Применение уточнений позволяет структурировать совокупность трансформационных связей и создавать древовидные иерархии наследования среди связей. Так как трансформационные связи можно использовать в описаниях других правил, подобные иерархии позволяют более эффективно описывать трансформации, а также задавать трансформации, которые было бы невозможно задать по-другому.

Шаблон – это специальная разновидность правила, описание которого начинается с ключевого слова «abstract». Сам по себе шаблон не применим никогда, вне зависимости от его секции выборки. Но на основе шаблона за счёт механизма уточнения можно создавать новые правила, которые уже могут быть применены при соответствии выборки и трансформируемой модели. Как и при применении обычных уточнений, использование шаблонов позволяет структурировать совокупность трансформационных связей для её последующего использования в других правилах трансформации. Возможно создание шаблонов как уточнений других шаблонов, но нельзя создать шаблон, являющийся уточнением обычного правила.

9. Пример описания трансформации

Ранее был описан пример трансформации, предназначенной для преобразования платформо-независимой UML-модели классов в платформо-зависимую модель, ориентированную на технологию CORBA. Но этот пример трансформации был описан на естественном языке, а теперь попробуем задать его с помощью формального языка трансформации. В данной трансформации будут участвовать две модели с именами «source» и «target», первая из которых

является исходной моделью, а вторая создаётся в процессе выполнения трансформации. Будет использоваться упрощённая метамодель классов UML, приведённая на рисунке.

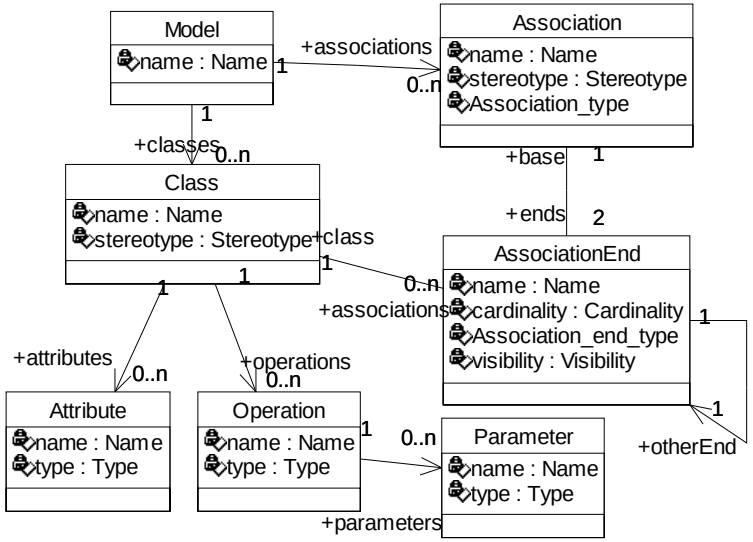

Рис. 1. Упрощённая метамодель диаграмм классов языка UML.

Трансформация будет состоять из одного блока. Ниже показан заголовок этого блока и первое правило, соответствующее фразе «Для каждого класса из PIM следует создать класс реализации в PSM и связанный с ним класс-интерфейс» из неформального описания трансформации.

```
stage example_CORBA_transformation {

rule class_to_class {
forall srcclass from source/classes
make implclass in target/classes;
implclass/name= srcclass /name+"_implementation";
implclass/stereotype="implementation"
make interclass in target/classes;
interclass/name=srcclass/name;
interclass/stereotype="interface";
make d in target/associations;
d/stereotype="realize";
make e in implclass/associations;
e/base=d;
e/cardinality=1;
e/association_end_type="navigable";
include e in d/ends;
make f in interclass/associations;
```

```
f/base=d;
f/cardinality=1;
f/association_end_type=”navigable”;
include f in d/ends;
e/otherend=f;
f/otherend=e;
}
```

Правило «class_to_class» для каждого класса исходной модели создаёт класс-интерфейс и класс-реализацию в генерируемой модели, а так же связь между ними. Далее напишем пару правил, соответствующих утверждениям «Все приватные атрибуты должны быть скопированы в соответствующие классы реализации» и «Публичные атрибуты следует также поместить в класс реализации, но уже как приватные; в интерфейс следует добавить методы для доступа и модификации этих атрибутов».

```
rule priv_attributes {
forall srcclass from source/classes
forall srcattrib from a/attributes
where ((srcattrib /visibility=”private”) or (srcattrib
/visibility=”protected”))
forall trgtclass from target/classes
forall d from rules/class_to_class
where ((d/srcclass=srcclass) and (d/implclass=trgtclass))
make trgtattrib in trgtclass/attributes;
trgtattrib/name=srcattrib/name;
trgtattrib/visibility=srcattrib/visibility;
trgtattrib/type=srcattrib/type;
}
```

```
rule pub_attributes {
forall srcclass from source/classes
forall srcattrib from a/attributes
where srcattrib/visibility=”public”
forall implclass from target/classes
forall interclass from target/classes
forall d from rules/class_to_class
where ((d/srcclass=srcclass) and (d/implclass=implclass) and
(d/interclass=interclass))
make trgtattrib in implclass/attributes;
trgtattrib/name=srcattrib/name;
trgtattrib/visibility=”private”;
trgtattrib/type=srcattrib/type;
make getoperation in interclass/operations;
getoperation/name=”get_”+srcattrib/name;
getoperation/type=srcattrib/type;
make setoperation in interclass/operations;
setoperation/name=”set_”+srcattrib/name;
make h in setoperation/parameters;
h/type=srcattrib/type;
}
```


В этих правилах использована трансформационная связь, образованная применением правила «class_to_class». Благодаря этой связи возможно определить класс-интерфейс и реализацию, порождённые заданным классом исходной модели. Следующее правило из неформального описания – «Публичные операции класса копируются в интерфейс, приватные – в класс реализации», оно описывается с помощью нескольких формальных правил.

```
rule methods1 {
forall srcclass from source/classes
forall srcoper from a/operations
where srcoper/visibility=”public”
forall trgtclass from target/classes
forall d from rules/class_to_class
where ((d/srcclass=srcclass) and (d/interclass=trgtclass))
make trgtoper in trgtclass/operations;
trgtoper/name=srcoper/name;
trgtoper/type=srcoper/type;
trgtoper/visibility=srcoper/visibility;
}
```

```
rule methods1a {
forall a from rules/methods1
forall b from a/srcoper/parameters
make c in a/trgtoper/parameters;
c/name=b/name;
c/type=b/type;
}
```

```
rule methods2 {
forall srcclass from source/classes
forall srcoper from a/operations
where srcoper/visibility=(”private” or “protected”)
forall trgtclass from target/classes
forall d from rules/class_to_class
where ((d/srcclass=srcclass) and (d/implclass=trgtclass))
make trgtoper in trgtclass/operations;
trgtoper/name=srcoper/name;
trgtoper/type=srcoper/type;
trgtoper/visibility=srcoper/visibility;
}
```

```
rule methods2a {
forall a from rules/methods2
forall b from a/srcoper/parameters
make c in a/trgtoper/parameters;
c/name=b/name;
c/type=b/type;
}
```

Правила с именами methods1 и methods2 отображают публичные и приватные операции соответственно, а вспомогательные правила methods1a и methods2a

нужны для копирования параметров операций. Далее формально зададим утверждение «Ассоциации «один к одному» и «многие к одному» преобразуются в атрибут - объектную ссылку», а так же «Связи-обобщения преобразуются в аналогичные связи-обобщения между классами-интерфейсами».

```
rule associations_single {
forall srcclass from source/classes
forall assoc from srcclass/associations
where assoc/base/association_type=″navigation″
where ((assoc/otherEnd/cardinality=″1″) or
(assoc/otherEnd/cardinality=″0..1″))
where assoc/otherEnd/association_end_type=″navigable″
forall d from rules/class_to_class
where d/srcclass=srcclass
forall e from rules/class_to_class
where e/srcclass=assoc/otherEnd/class
make objptr in d/implclass/attributes;
objptr/type=e/interclass/name+″_ptr″;
objptr/name=assoc/otherEnd/name;
objptr/visibility=″private″;
}

rule association_generalization {
forall srcclass from source/classes
forall srcassoc from a/associations
where srcassoc/base/association_type=″generalization″
where srcassoc/otherEnd/association_end_type=″navigable″
forall c from rules/class_to_class
where c/srcclass=srcclass
forall d from rules/class_to_class
where d/srcclass=srcassoc/otherEnd/class
make trgtassoc in target/associations;
trgtassoc/association_type=″generalization″;
trgtassoc/stereotype=srcassoc/base/stereotype;
trgassoc/name=srcassoc/base/name;
make f in c/interclass/associations;
f/name=srcassoc/name;
f/cardinality=srcassoc/cardinality;
f/association_end_type=srcassoc/association_end_type;
include f in trgtassoc/ends;
make g in d/interclass/associations;
g/name=srcassoc/otherEnd/name;
g/cardinality=srcassoc/otherEnd/cardinality;
g/association_end_type=srcassoc/otherend/association_end_type;
include g in trgtassoc/ends;
f/otherend=g;
g/otherend=f;
}
```


И, наконец, последнее правило соответствует утверждению «Интерфейсы исходной модели и их связи копируются в генерируемую модель без создания классов-реализаций».

```
rule interface_mapping {
forall a from rules/class_to_class
where a/srcclass/stereotype=″interface″
delete a/implclass;
      }
}
```

На самом деле данное правило просто удаляет лишние классы реализации, созданные первым правилом. Последняя фигурная скобка обозначает окончание блока трансформации. Итак, полученное описание трансформации формально определяет те преобразования модели, которые необходимы при переходе к PSM, основанной на CORBA и которые были заданы на естественном языке в начале статьи. В результате выполнения этого описания трансформации на какой-либо модели классов будет получена платформо-зависимая модель «target» и совокупность связей между этой моделью и исходной.

10. Полнота языка трансформации моделей.

Важным требованием к языку описания трансформаций при его использовании в MDA является универсальность – возможность описать с его помощью любое преобразование из PIM в PSM для любой технологии. Причём необходимо учитывать не только все существующие технологии, но и те, которые будут разработаны в будущем. Это означает, что язык должен позволять задавать любые отображения моделей.

Будем считать, что все метамодели связны, то есть любой элемент метамодели достижим из корневого элемента метамодели с помощью навигационных выражений.

Теорема 1. С помощью предложенного языка трансформации можно задать любое однозначное отображение моделей.

Для доказательства нам понадобятся две леммы.

Лемма 1. Для любой конечной модели можно написать правило, которое бы было применимо к ней и не применимо к любой другой модели.

Доказательство.

Применимость правила определяется его секцией выборки. Приведём алгоритм построения нужной нам секции выборки, которая бы соответствовала определённой модели $\boldsymbol{a}$ и не соответствовала никакой другой. Для каждого элемента модели r_i из $\boldsymbol{a}$ построим оператор выборки ***forall*** r_i ***from model/nav_expression***, где ***nav_expression*** – навигационное выражение, позволяющее выбрать данный элемент (и все остальные элементы того же типа) в модели. Для каждого атрибута e_j со значением f_j построим уточняющее

выражение ***where*** $r_i/e_j=f_j$. Для каждой относящейся к данному элементу связи s_j с кардинальностью 1 добавим уточняющее условие ***where*** $r_i/s_j=r_k$, где r_k – элемент модели, на который указывает связь. Для каждой связи s_j с кардинальностью больше 1, которая связывает данный элемент с элементами $r_{j1}…r_{jn}$, добавим уточняющее условие ***where*** $(r_{j1}$ ***belongsto*** $r_i/s_j)$ ***and … and (*** r_{jn} ***belongsto*** $r_i/s_j)$ ***and*** $((r_i/s_j$ ***exclude*** r_{j1} ***exclude*** r_{j2} *…* ***exclude*** $r_{jn})=null)$ (если связь s_j пуста, то есть $n=0$, то такое уточняющее условие вырождается в ***where*** $r_i/s_j=null$). Если создать такие операторы выборки и уточнения для всех элементов модели ***a***, то полученная секция выборки будет удовлетворять необходимым свойствам: она применима к модели ***a*** и не применима ни к какой другой.

Лемма 2. Для любой модели можно написать правило, которое бы при выполнении создавало эту модель.

Доказательство этой леммы элементарно. Секция генерации позволяет императивно описывать процесс создания модели элемент за элементом. Поэтому для того, чтобы правило генерировало определённую модель a, следует в секцию генерации добавить операторы создания для каждого элемента модели и инициализировать все атрибуты и ссылки их значениями, взятыми из модели.

Доказательство теоремы 1.

Пусть есть множество исходных моделей ***A*** и множество генерируемых моделей ***B***, и отображение ***U***, которое каждой исходной модели ставит в соответствие определённую генерируемую модель. Необходимо доказать, что это отображение можно задать с помощью правил трансформации, то есть что $\exists\, T:\ \forall a \in A\ T(a)=U(a)$, где ***T*** это описание трансформации, а ***T(a)*** – результат применения этой трансформации к модели ***a*** из ***A***.

Будем строить это описание трансформации следующим образом. Представим отображение ***U*** в виде множества пар вида (<исходная модель>,<генерируемая модель>). Для каждой такой пары ***(a,b)*** в описание трансформации добавим правило специального вида.

Его секция выборки должна быть применима только на модели ***a***, причём ровно один раз. То, что такая секция выборки возможна, доказано в лемме 1. А для того, чтобы правило никогда не применялось дважды (это могло бы случиться, если модель ***a*** обладает симметрией), добавим уточняющее условие вида ***where rules/rule_name=null***, где ***rule_name*** – имя данного правила. После первого применения правила соответствующее множество трансформационных связей будет не пусто, а условие – ложно, что гарантированно не допустит повторного применения правила.

Секция генерации этого правила при выполнении должна создавать модель ***b*** со всеми содержащимися в ней элементами. Существование такой секции генерации доказано в лемме 2.

Итак, правило такого вида будет порождать модель ***b***, если исходная модель – ***a***, и не будет делать ничего (ни разу не будет применено), если исходная

модель отлична от ***a***. Если создать такие правила для всех пар из исходного отображения ***U*** и объединить их в блок трансформации, то полученное описание трансформации ***T*** и будет искомым. Порядок следования и выполнения правил в блоке может быть любым. Доказательство того, что T-искомое описание трансформации, элементарно:

Возьмём произвольную модель ***a*** из множества исходных моделей, ей однозначно соответствует некоторая модель из множества генерируемых моделей, назовём её ***b***. То есть выполняется ***U(a)=b***. Тогда в описании трансформации ***T*** содержится правило ***t***, соответствующее паре моделей ***(a,b)***, то есть ***t(a)=b***. Так как отображение ***U*** однозначно, никаких других правил, применимых к модели ***a***, в описании ***T*** не существует. Тогда ***T(a)=t(a)=b=U(a)***. Так как ***a*** – произвольная исходная модель, доказано что $\forall a \in A\ T(a)=U(a)$, то есть построенное нами описание трансформации ***T*** задаёт отображение ***U***. Доказательство окончено.

Существенным недостатком данной теоремы является то, что не гарантируется конечность описания трансформации, то есть если отображается бесконечное множество моделей, то и количество правил трансформации будет не ограничено. Даже если ограничить максимальное число элементов в модели, описание трансформации всё равно может быть бесконечным. Например, существует бесконечно много моделей, состоящих ровно из одного класса без атрибутов и операций, и различающихся только именами класса. Разумеется, на практике такое описание трансформации, состоящее из бесконечного количества правил, реализовано быть не может. Поэтому необходимо знать условия, при которых бы гарантированно существовало конечное описание трансформации.

Определение. Структурой модели назовём набор её элементов и связей между ними, без учёта значений атрибутов. Будем говорить, что две модели (определённые на общей метамодели) имеют общую структуру, если существует взаимно однозначное соответствие между элементами этих моделей, сохраняющее тип элемента и связи между элементами. На практике это означает, что модели имеют одинаковые элементы и связи между ними, но возможно разные значения атрибутов.

Определение. Будем называть модель конечной, если она состоит из конечного числа элементов метамодели. На практике, очевидно, все модели конечны.

Теорема 2. Пусть существует однозначное отображение ***U(x)*** множества исходных моделей A на множество генерируемых моделей B, и пусть все модели конечны. Пусть структура любой генерируемой модели зависит только от структуры исходной модели, но не от значений её атрибутов (то есть для любых исходных моделей a_1 и a_2 с общей структурой их образы $U(a_1)$ и $U(a_2)$ так же должны иметь общую структуру). И, наконец, пусть значение любого атрибута генерируемой модели может быть выражено как функция от исходной модели с использованием только алгебраических, строковых и теоретико-множественных операций. Тогда существует конечное описание

трансформации ***T***, задающее отображение ***U***, причём это описание трансформации всегда выполнимо за конечное время.

Доказательство.

Множество исходных моделей A разобьём на минимальное число групп так, чтобы все модели внутри группы имели одинаковую структуру (то есть отличались только значениями атрибутов). Число таких групп будет конечно, так как по условию число элементов в любой модели конечно, а из конечного числа элементов можно составить конечное число моделей с различной структурой. Так как структура генерируемой модели зависит только от структуры исходной модели, отображение такой группы оператором ***U*** так же будет состоять из моделей с одинаковой структурой. Назовём её структурно однородной группой. Следовательно, отображение ***U*** можно представить в виде конечного множества отображений $U_1 \ldots U_n$, каждое из которых отображает структурно однородную группу моделей в структурно однородную группу. Представим это множество отображений в виде конечного множества пар ***(a, b=U(a))***, где ***a*** и ***b*** – группы моделей с одинаковой структурой. Так же как и в доказательстве теоремы 1, построим правило трансформации для каждой такой пары.

Секция выборки строится аналогично теореме 1, за исключением того, что не добавляются уточняющие условия на значения атрибутов, так как правило пишется сразу для группы моделей и следовательно эти значения не определены.

В секции генерации, так же как и в доказательстве теоремы 1, явным образом используем операторы создания элемента и установления связей между ними. Так как модели в группе имеют общую структуру, генерация элементов и связей одинакова для всех моделей группы. А вот значения атрибутов для разных моделей могут отличаться, поэтому уже невозможно явно присвоить значения атрибутов. Но по условию теоремы, значение любого атрибута ***r*** из генерируемой модели $b_i \in b$ может быть представлено как $r=f(a_i)$, где $a_i \in a$, а ***f()*** – функция, которая может быть выражена через простейшие операции. Функция ***f()*** одна и та же для всех $a_i \in a$. Следовательно, если для каждого атрибута r использовать оператор присваивания ***r=f(a)***, можно задать значения атрибутов сразу для всех $b_i \in b$.

Итак, полученное правило ***t*** будет применимо ко всем моделям группы ***a***, и ни к каким другим. При этом генерируемая модель всегда будет иметь структуру, соответствующую группе ***b***, а значения атрибутов будут соответствовать отображениям конкретной исходной модели, то есть $t(a_i)=U(a_i)$. Объединив множество таких правил для всех групп, получим описание трансформации ***T*** и $\forall a \in A\ T(a)=U(a)$. Число правил в описании трансформации равно числу групп моделей с одинаковой структурой. Так как число групп конечно (при условии что все модели конечны), описание трансформации ***T*** конечно. Так как для любой исходной модели при трансформации применяется ровно одно правило и только один раз, порядок применения правил в блоке трансформации ***T*** не влияет на результат. То есть мы можем назначить линейный порядок, в таком

случае трансформация гарантированно завершится за конечное время. Все утверждения теоремы доказаны.

Замечание: на самом деле не обязательно наличие одной общей функции, выражающей атрибуты. Достаточно, чтобы значение каждого атрибута для всего множества генерируемых моделей задавалось с помощью конечного числа функций, представимых с помощью простейших операций над элементами, связями и значениями атрибутов исходных моделей.

Доказательство аналогично доказательству теоремы 2. Но вместо того, чтобы строить отдельное правило для каждой структурно однородной группы, эти группы следует сначала разбить на подгруппы так, чтобы значения атрибутов всех моделей в подгруппе выражались в виде одинаковых функций. Так как общее число функций, задающих атрибуты, конечно, то и число подгрупп будет конечно. Построив правило трансформации для каждой подгруппы так же, как и в доказательстве теоремы 1, и объединив все эти правила в единый блок трансформации, получим искомое описание трансформации.

Условия теоремы 2 можно смягчить ещё больше, но, к сожалению, невозможно совсем от них избавиться, придя к условиям теоремы 1 и сохранив гарантированную конечность описания трансформации. В частности, если значения атрибутов генерируемой модели не выражаются с помощью элементарных операций, то единственный способ задать такое отображение – перечисление. Если множество возможных значений атрибута бесконечно, то и количество правил трансформации, необходимых для задания такого перечисления, будет бесконечно большим. Однако описания трансформаций для большинства практических задач могут быть успешно заданы с помощью предложенного языка трансформации. Необходимо отметить, что используемый в доказательстве теорем метод построения описаний трансформации не рекомендуется для применения на практике, он крайне громоздок и использован только для формального доказательства полноты языка трансформации.

11. Практическая реализация инструмента трансформации

В настоящее время ведутся работы по практической реализации прототипа инструмента трансформации, поддерживающего описанный выше язык. В целях упрощения работы было принято решение сделать прототип, поддерживающий только одну метамодель (метамодель классов, показанную на рис. 1), однако при этом используются универсальные решения, которые можно легко перенести на любую другую модель и внедрить в инструмент поддержку любых метамоделей. На данный момент инструмент поддерживает работу только с парой моделей – исходной и генерируемой, но его функциональность будет расширена для поддержки одновременной трансформации нескольких моделей.

Для внешнего представления UML на данный момент используется собственная текстовая нотация, но в будущем планируется включить в

инструмент трансформации модуль, позволяющий импортировать и экспортировать модели в XMI-представлении. Это позволит интегрировать инструмент трансформации с различными средами UML-разработки, поддерживающими этот стандарт.

Отдельной и пока не решённой задачей является создание UML-редактора, который мог бы поддерживать соответствие между моделями при внесении в них изменений уже после трансформации. Необходимая для этого информация о трансформационных связях сохраняется, но пока не используется после окончания процесса трансформации.

12. Заключение

Технология MDA может оказаться новой ступенью эволюции средств и методик разработки программных систем. Но это станет возможно только в том случае, если появятся инструменты разработки, специально предназначенные для этой технологии и максимально использующие её потенциал. Существующие на данный момент инструменты являются в основном адаптациями и расширениями старых разработок, они только поддерживают подход MDA, но не предназначены для него. Таких инструментов недостаточно, чтобы реализовать среду разработки ПО нового поколения. Поэтому на данный момент ведётся множество разработок инструментария для MDA.

Одна из важнейших и самых сложных возникающих при этом задач состоит в автоматизации трансформации моделей. Ранее этой задаче не уделялось должного внимания, так как область применения возможных результатов была ограничена. Существующие методики и заимствования из других областей информатики оказываются малоэффективными и неудобными для решения практически задач, поставленных технологией MDA. Поэтому актуальна задача разработки методики описания и выполнения трансформации моделей с учётом особенностей её применения для MDA.

В данной работе предложен язык описания трансформации, предназначенный, прежде всего, для применения к моделям UML (впрочем, благодаря поддержке различных метамоделей, он может быть использован и для трансформации других моделей). При разработке языка учитывались особенности его применения в технологии MDA. Особое внимание было уделено понятности языка для человека и хорошей структурированности описаний трансформации на этом языке.

Литература

1. Martin Flower. UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition. Addison Wesley, 2003.
2. Daniel Varro, Andras Pataricza. UML Action Semantics For Model Transformation Systems. Periodica Polytechnica, no. 3-4, 2003.
3. Jernej Kovse, Theo Härder. Generic XMI-Based UML Model Transformations. ZDNet UK Whitepapers, 2002.

4. Keith Duddy, Anna Gerber. Declarative Transformation for Object-Oriented Models. Transformation of Knowledge, Information, and Data: Theory and Applications, 2003.
5. Krzysztof Czarnecki, Simon Helsen. Classification of Model Transformation Approaches. University of Waterloo, Canada, 2003.
6. Agrawal, G. Karsai and F. Shi. Graph Transformations on Domain-Specific Models. Journal on Software and Systems Modeling, 2003.
7. Common Warehouse Metamodel (CWM) Specification. OMG Documents, Feb. 2001. http://www.omg.org/cgi-bin/apps/doc?formal/03-03-02.pdf
8. Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, Andrew Wood. Transformation: The Missing Link of MDA. Proceedings 1st International Conference on Graph Transformation (ICGT 2002), 2002. http://www.dstc.edu.au/Research/Projects/Pegamento/publications/icgt2002.pdf
9. Anneke Kleppe, Jos Warmer, Wim Bast. MDA Explained. The Model Driven Architecture: Practice and Promise. Pearson Education, 2003.
10. MDA Guide Version 1.0. Joaquin Miller and Jishnu Mukerji(eds.), 2003. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf
11. XSL Transformations (XSLT) v1.0. W3C Recommendation, Nov. 1999. http://www.w3.org/TR/xslt
12. Meta-Object Facility (MOF) specification, version 1.4 . OMG Documents, Apr. 2002. http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf
13. D. Chamberlin. XQuery: An XML query language. IBM systems journal, no 4, 2002.
14. Александр Цимбал. “Технология CORBA для профессионалов”. Санкт-Петербург, ЗАО “Питер Бук”, 2001.