**Московский государственный университет им. М.В.Ломоносова**

**Факультет вычислительной математики и кибернетики**

**Кафедра автоматизация систем вычислительных комплексов**



Дипломная работа

**Семантическое отображение канонической информационной модели СИНТЕЗ в формальный язык спецификаций AMN**

студента 522 группы

Сергеева Алексея Юрьевича

Научный руководитель:

с.н.с. ИПИ РАН, м.н.с. ЛОИТ ВМиК МГУ,

к.т.н. Ступников Сергей Александрович

**Москва 2010**

# Аннотация

В настоящее время в различных областях науки наблюдается увеличение числа неоднородных информационных ресурсов, представленных в различных моделях. Представляется актуальной задача их интеграции и повторного использования. Перспективной технологией при решении такого рода задач является технология предметных посредников, в которой выделяются методы унификации информационных моделей ресурсов и регистрация ресурсов в предметных посредниках. Для верификации данных методов используется понятие *уточнения*. Неформально, спецификация *B* *уточняет* спецификацию *A*, если пользователь может использовать *B* вместо *A*, не замечая факта замены *A* на *B*. Задачей унификации моделей является создание *канонической информационной модели*, представляющей однородным образом модели ресурсов. При регистрации ресурсов онтологии и концептуальные схемы посредника и ресурсов для последующей интеграции выражаются в канонической информационной модели. Целью работы является построение трансформации языка СИНТЕЗ, являющегося канонической информационной моделью, в формальный язык спецификаций AMN, для которого существуют специализированные инструментальные средства доказательства уточнения. Трансформация позволяет на основании языка СИНТЕЗ автоматически порождать соответствующие спецификации языка AMN. Это позволяет доказывать уточнение спецификаций языка СИНТЕЗ при помощи средств автоматического и интерактивного доказательства языка AMN. В качестве языка трансформации был выбран ATL (ATLAS Transformation Language).

Оглавление

[Аннотация 2](#_Toc262658738)

[1. Введение 5](#_Toc262658739)

[1.1. Структура дипломной работы 9](#_Toc262658740)

[2. Постановка задачи 10](#_Toc262658741)

[3. Языки спецификаций и языки трансформации моделей 11](#_Toc262658742)

[3.1. Язык СИНТЕЗ 11](#_Toc262658743)

[3.2. Язык AMN 12](#_Toc262658744)

[3.3. Трансформация моделей и язык ATL 14](#_Toc262658745)

[4. Формализация абстрактного синтаксиса языка AMN 16](#_Toc262658746)

[4.1. Конструкции языка AMN 17](#_Toc262658747)

[4.1. Выражения 20](#_Toc262658748)

[4.1.1. Бинарное отношение 20](#_Toc262658749)

[4.1.2. Пересечение множеств 20](#_Toc262658750)

[4.1.3. Переопределение отношений 21](#_Toc262658751)

[4.1.4. Частичная функция 21](#_Toc262658752)

[4.1.5. Полная функция 21](#_Toc262658753)

[4.2. Предикаты 23](#_Toc262658754)

[4.3. Подстановки 24](#_Toc262658755)

[4.3.1. Подстановка присваивания 25](#_Toc262658756)

[4.3.2. Подстановка с предусловием 25](#_Toc262658757)

[4.3.3. Подстановка неограниченного выбора 26](#_Toc262658758)

[4.3.4. Последовательная подстановка 26](#_Toc262658759)

[4.3.5. Подстановка вызова операции 27](#_Toc262658760)

[4.3.6. Параллельная подстановка 27](#_Toc262658761)

[4.3.7. Тождественная подстановка 27](#_Toc262658762)

[4.3.8. Блочная подстановка 27](#_Toc262658763)

[4.3.9. Подстановка с утверждением 28](#_Toc262658764)

[4.3.10. Подстановка ограниченного выбора 28](#_Toc262658765)

[4.3.11. Условная подстановка 28](#_Toc262658766)

[4.3.12. Условная подстановка с ограниченным выбором 28](#_Toc262658767)

[4.3.13. Подстановка локального определения 29](#_Toc262658768)

[4.3.14. Подстановки замены переменной 29](#_Toc262658769)

[4.3.15. Подстановка определения новой переменной 29](#_Toc262658770)

[4.3.16. Подстановка цикла с условием 29](#_Toc262658771)

[5. Построение трансформации языка СИНТЕЗ в язык AMN 30](#_Toc262658772)

[5.1. Основные принципы отображения 31](#_Toc262658773)

[5.1.1. Экстенсиональная интерпретация АТД 31](#_Toc262658774)

[5.1.2. Моделирование структуры спецификации типов при помощи средств композиции абстрактных машин 33](#_Toc262658775)

[5.2. Построение контекстной машины модуля 34](#_Toc262658776)

[5.3. Структура абстрактной машины, соответствующей типу: моделирование атрибутов, методов, инвариантов 39](#_Toc262658777)

[5.4. Отображение формул канонической модели 50](#_Toc262658778)

[5.4.1. Атомарные предикаты 50](#_Toc262658779)

[5.4.2. Условия 51](#_Toc262658780)

[5.4.3. Составные формулы 53](#_Toc262658781)

[5.5. Представление функций доступа 55](#_Toc262658782)

[Заключение 59](#_Toc262658783)

[Список литературы 60](#_Toc262658784)

# Введение

В настоящее время в различных областях деятельности людей и науки в частности наблюдается экспоненциальный рост накапливаемых, экспериментальных данных, описанных с помощью всевозможных информационных моделей. Таким образом нынешний период развития информационных технологий сопровождается созданием множества различных информационных моделей. Наряду с созданием моделей для распределенных систем (таких как архитектуры OMG, архитектуры Семантического Веба, архитектуры цифровых библиотек, архитектуры информационных грид систем), можно выделить разработку в области конкретных информационных моделей - моделей данных (таких как SQL, UML, XML), моделей потоков работ (MQSeries, SAP/R3, Eastman), семантических моделей (включая онтологические модели и модели метаданных). В основе почти всех из них находятся разнообразные понятия и парадигмы, зачастую несовместимые между собой. Также можно выделить другую тенденцию, заключающуюся в накапливании основанных на данных моделях информационных ресурсах, количество которых непрерывно растет. На этом фоне возникает необходимость в использовании и интеграции компонентов и сервисов, представленных в различных моделях и приложениях, также как и их повторное использование для создания новых информационных систем. Можно заметить противоречивость такого рода тенденций, так как чем больше появляется всевозможных моделей наряду с уже использующимися, тем труднее становится задача их интеграции между собой. Данная проблема вынуждает разработчиков новых систем создавать новые ресурсы, вместо грамотного повторного использования уже существующих.

В течение последних лет в Лаборатории композиционных методов проектирования информационных систем (ИС) ИПИ РАН проводились интенсивные исследования по разработке методов решения задач над неоднородными информационными ресурсами, таких как композиционное проектирование ИС [2], регистрация ресурсов в предметных посредниках [3], интеграция моделей ресурсов [4].

Представленные выше методы нашли свое отражение в ряде проектов лаборатории. В частности, была разработана многоуровневая инфраструктура, включающая в себя уровень информационных ресурсов, промежуточный слой, обеспечивающий интероперабельность ресурсов благодаря технической унификации их интерфейсов и введению дистанционных механизмов обращения к ресурсам (адаптеры, осуществляющие преобразование запросов, выраженных в канонической информационной модели посредников, в их представление в информационной модели ресурса, включаются в состав промежуточного слоя), уровень *предметных посредников*, каждый из которых создает спецификацию предметной области для решения некоторого класса задач, используя каноническую информационную модель для представления семантики предметной области и унифицированного отображения разнообразных видов информационных моделей ресурсов (моделей данных, сервисных моделей, онтологических моделей, процессных моделей), уровень задач (приложений), формулируемых в терминах одного или нескольких посредников. Следует заметить, что посредники, в свою очередь, могут рассматриваться как информационные ресурсы для использования в посредниках более высокого уровня. В такой инфраструктуре предметные посредники играют ключевую роль для решения семантических проблем, перечисленных выше. В частности, при интеграции неоднородных ресурсов в посреднике нужно уметь семантически отождествлять объекты, представленные в различных информационных моделях, и семантически правильно отображать схемы интегрируемых ресурсов в схему посредника. Поскольку в общем случае ресурсы неоднородны (представлены в различных моделях), при интеграции неоднородных ресурсов для однородного представления их семантики требуется приведение различных информационных моделей к унифицированному виду в рамках некоторой унифицирующей информационной модели, которая называется *канонической*. Для унификации разнородных спецификаций прежде всего требуется умение сопоставлять спецификации различных ресурсов друг с другом так, чтобы можно было отвечать на вопрос, можно ли при реализации посредника использовать спецификацию существующего ресурса вместо фрагмента спецификации посредника. Для этого достаточно доказать, что рассматриваемые спецификации находятся в отношении уточнения. Говорят, что спецификация A уточняет спецификацию D, если A можно использовать вместо D так, что пользователь D не будет замечать этой замены. Средства доказательства факта уточнения спецификации некоторого компонента спецификацией другого компонента (реализуемые на основе теоретико-модельных нотаций и соответствующего инструментария [6,7]) составляют фундамент применяемых методов конструирования унифицирующих (канонических) моделей представления информации в посредниках. Каноническая информационная модель служит в качестве общего языка для адекватного выражения семантики разнородных моделей представления информации, используемых в разнообразных информационных ресурсах, релевантных посреднику. Методы отображения информационных моделей и синтеза расширяемых канонических информационных моделей для слоя предметных посредников подробно рассмотрены в [7]. Там же предложена архитектура Унификатора информационных моделей. Задачей Унификатора моделей является унификация множества информационных моделей(называемых *исходными*), использующихся в различных информационных ресурсах некоторой информационной системы. Унификация исходной модели R есть приведение ее к канонической информационной модели C, т. е. создание такого расширения E канонической модели (которое может быть и пустым) и такого отображения M исходной модели в расширенную каноническую, что исходная модель *уточняет* расширенную каноническую модель. Уточнение моделей означает, что для любой допустимой спецификации r модели R ее образ M(r) при отображении M уточняется спецификацией r. Синтез канонической модели заключается в построении указанных расширений ядра канонической модели для всех исходных моделей и объединении таких расширений. В качестве ядра канонической информационной модели применяется язык СИНТЕЗ [1], который ориентирован на спецификацию предметных посредников и синтез канонических моделей. Спецификация предметного посредника для класса задач включает определения понятий предметной области, выражаемых соответствующими онтологическими спецификациями, спецификации классов объектов предметной области, спецификации типов экземпляров названных классов и их методов, определяющих их поведение, спецификации процессов решения задач данного класса как совмещенных во времени последовательностей действий, реализуемых методами классов, сервисами и другими процессами. Предполагается, что такие спецификации преобразуются в спецификации канонической модели, имеющей формальную семантику.

Регистрация релевантных посреднику ресурсов рассматривается как задача композиционного проектирования систем [2, 3]. Регистрация ресурсов есть процесс целенаправленной трансформации спецификаций, включающий декомпозицию спецификаций посредника на непротиворечивые фрагменты, поиск среди спецификаций релевантных ресурсов подходящих типов данных — кандидатов для уточнения ими спецификаций типов посредника, построение выражений, определяющих классы ресурсов в виде композиции классов посредника. Для подобного манипулирования спецификациями разработано специальное *исчисление спецификаций* [4].

Принципиальным моментом в этой схеме является реализация доказательства уточнения фрагментов спецификаций посредника спецификациями ресурсов в процессе отображения таких спецификаций.

Для таких манипуляций спецификациями требуется выражение их семантики в некотором формальном языке, позволяющем осуществлять доказательство непротиворечивости уточнения спецификаций. В качестве такого языка выбрана Нотация Абстрактных Машин (Abstract Machine Notation, AMN [6]), которая позволяет осуществлять необходимые рассуждения относительно спецификаций. Для AMN разработаны специальные инструментальные средства, составляющие так называемую В-технологию [7]. Для того, чтобы можно было манипулировать спецификациями информационных моделей в рамках В-технологии (в частности, доказывать корректность отношения уточнения между спецификациями, выраженными в различных моделях), необходимо корректно отобразить такие модели в язык AMN.

## Структура дипломной работы

В разделе 3 дается краткое описание языков спецификаций СИНТЕЗ и AMN а также языка трансформаций ATL.

В разделе 4 формализуется абстрактный синтаксис языка AMN.

В разделе 5 описывается трансформация, реализующая отображение спецификации на языке СИНТЕЗ в спецификацию на языке AMN.

# Постановка задачи

Существенным моментом в методах регистрации ресурсов в предметных посредниках и унификации информационных моделей является обеспечение возможности автоматизированного доказательства факта уточнения спецификации требований спецификацией существующих компонентов. Для этого необходимо корректно отобразить существующие спецификации в язык AMN, позволяющий автоматизированно доказывать факт уточнения при помощи специальных инструментальных средств. Метод такого отображения создан [16] и существует возможность применять его на примере ядра канонической информационной модели (языка СИНТЕЗ).

В настоящее время в Лаборатории композиционных методов проектирования информационных систем разрабатываются прототипы инструментальные средства для регистрации ресурсов в предметных посредниках, а также средства унификации информационных моделей. В качестве объединяющей платформы для этих средств выбран Eclipse [17]. Задачей данной дипломной работы является формализация абстрактного синтаксиса языка AMN, а также программная реализация семантического отображения канонической информационной модели СИНТЕЗ в формальный язык AMN, с использованием языка трансформации моделей ATL (являющегося частью Eclipse Modeling Project). Компонент, реализующий отображение, рассматривается как неотъемлемая часть архитектуры упомянутых средств интеграции неоднородных ресурсов.

# Языки спецификаций и языки трансформации моделей

## Язык СИНТЕЗ

В данной работе в качестве ядра канонической модели был выбран язык СИНТЕЗ [1], разработанный в Лаборатории композиционных методов проектирования информационных систем Института проблем информатики РАН. Он ориентирован на семантическую интероперабельность и композиционное проектирование информационных систем в широком диапазоне существующих неоднородных информационных компонентов. Метаязыком, при помощи которого в ядре канонической информационной модели представляются средства определения информационных источников, является язык фреймов. Фрейм - это структурированная символическая модель некоторого понятия. Фрейм можно рассматривать как множество атрибутов, называемых слотами, каждый из которых используется для представления свойства понятия. Каждый слот может иметь множество значений. Единицей определения канонической модели является модуль. Каждый модуль задает обобщенное представление информационных источников, либо является модулем спецификации предметной области или концептуального проекта информационной системы. Каноническая модель содержит унифицированную систему типов, включающую универсальный конструктор типов (Абстрактный Тип Данных, АТД), а также представительный набор встроенных типов. АТД по умолчанию является объектным, хотя он также может быть объявлен и как необъектный. Описание абстрактного типа данных инкапсулирует спецификации атрибутов, методов и инвариантов типа. Методы и инварианты описываются встроенным типом функции. Типы языка организованы в иерархию обобщения, базирующуюся на отношении тип-подтип. Значение подтипа может использоваться всюду, где ожидается значение супертипа; подтип наследует элементы спецификации супертипа; допускается множественное наследование спецификаций подтипом. Отношение тип-подтип основывается на понятии уточнения. Спецификация типа функции включает описание параметров функции и предикативную спецификацию функции. Для задания предикативных спецификаций функций в канонической модели используется типизированный вариант логики предикатов первого порядка. Предикативная спецификация позволяет задавать смешанные и постусловия, определяющие действия, реализуемые функцией. Для обращения к информационным источникам в языке предусмотрены формулы специального вида - правила. В языке правил не разрешается использование конструкций, связанных с изменением состояния системы, однако допускается использование широкого спектра операций композиции источников, характерных для языков запросов: например, соединение, пересечение, объединение.

Однородные совокупности объектов предметной области (информационные источники) представляются в канонической модели коллекциями. Экземплярами коллекций являются значения АТД. В случае если экземпляр типа объектный, коллекция называется классом. Явно разграничивая объявление типа и объявление коллекции, язык СИНТЕЗ подчеркивает роль коллекции как представителя множества объектов и роль типа как спецификации структуры и поведения объектов, связанных с коллекцией.

## Язык AMN

Формальная семантика необходима для проведения доказательных рассуждений о моделях информационных ресурсов. Имея формальную семантику можно строго формулировать утверждения о непротиворечивости и уточнении спецификаций, выраженных в модели. Появляется возможность доказательства корректности отображений одних моделей информационных ресурсов в другие модели. Для таких манипуляций информационными моделями и выраженными в них спецификациями требуется выражение их семантики в некотором формальном языке, позволяющем осуществлять доказательство непротиворечивости и уточнения спецификаций. Таким языком может являться теоретико-модельный язык Нотация Абстрактных Машин (Abstract Machine Notation, AMN [6]), основанный на теории множеств и логике предикатов первого порядка, который позволяет осуществлять необходимые рассуждения относительно спецификаций. Язык AMN обеспечивает манипулирование теоретико-множественными спецификациями в логике первого порядка и дает возможность доказать правильность уточнения спецификаций. Для AMN разработаны специальные инструментальные средства, составляющие в совокупности так называемую В-технологию[7]. Язык AMN как теоретико-множественная нотация позволяет рассматривать состояния и операции системы интегрировано как спецификацию пространства состояний и поведения (определенного операциями на состояниях) абстрактных машин. Спецификация состояния абстрактной машины вводится переменными состояния вместе с инвариантами - ограничениями, которые должны всегда удовлетворяться. Операции определяются на основе расширения формализма охраняемых команд Дейкстры. Ключевым понятием AMN является уточнение, позволяющее соотносить спецификации систем различных уровней абстракции. Уточняющая спецификация может быть гораздо более детальной, чем уточняемая спецификация. Конструируется уточняющая спецификация на основе алгоритмического уточнения и уточнения данных. Уточнение формализуется в AMN путем формулировки ряда теорем специального вида, так называемых *proof obligations*. Такие теоремы формулируются автоматически при помощи инструментальных средств поддержки В-технологии на основании склеивающих инвариантов, соотносящих состояния уточняемой и уточняющей системы. При генерации данных теорем важную роль играют подстановки. Подстановки представляют собой математические нотации, преобразующие предикаты. Пусть *S* – подстановка, а *P* – предикат. Тогда запись *S* [ *P* ] представляет предикат, полученный после трансформации *P* подстановкой *S*. Подстановки позволяют моделировать динамический аспект В-технологии: позволяют установить, как свойства данных преобразуются операциями компонента.

## Трансформация моделей и язык ATL

Для трансформации моделей созданы специальные языки, которые развиваются в контексте *Движимой моделями архитектуры* (Model-Driven Architecture, MDA [8]) - подхода, поддерживаемого стандартом OMG MOF (Meta-Object Facility) [12]. Базовыми составляющими MDA являются модели. Они рассматриваются как первичные сущности, и наиболее важными операциями манипулирования моделями становятся преобразования моделей, отображения моделей из одной в другую.

Модель определяется в соответствии с семантикой некоторой метамодели, при этом говорят, что модель конформна метамодели. Стандартом MOF определена четырехуровневая архитектура моделей: модели уровня М0 описывают объекты реального мира, модели уровня М1 называются обычно *схемами*, модели уровня М2 представляют собой собственно *информационные модели* (например, в архитектурах систем управления данными представляющие собой совокупность языка определения данных и языка манипулирования данными), модели уровня М3 – метаметамодели, предназначенные для описания моделей уровня М2.

Примером языка трансформации моделей может являться QVT (Query-View-Transformation) [13].

В качестве языка трансформации моделей в данной работе рассматривается язык ATL (ATLAS Transformation Language) [9], разработанный научной группой ATLAS INRIA & LINA. ATL является ответом на RFP (Request for Proposal) языка QVT. Система типов и операций над типами языка ATL очень близка (но не тождественна) системе типов языка OCL [10]. Для языка ATL на базе платформы Eclipse реализована интегрированная среда разработки, называемая ATL Development Tools (ADT). В качестве модели уровня М3 рассматривается модель *Ecore* [11].

Трансформация представляет собой модуль языка ATL. Модуль состоит из заголовка (включающего имя модуля и имена переменных, соответствующих исходной и целевой моделям) и множества правил, определяющих способ построения элементов целевой модели на основе элементов исходной модели. Так называемые *сопоставляющие правила* (matched rules), составляющие ядро трансформации, позволяют описывать:

* какой элемент исходной модели должен быть взят;
* количество и тип порождаемых элементов целевой модели;
* способ, при помощи которого элементы целевой модели инициализируются на основании элементов исходной модели.

Для построения трансформации языка СИНТЕЗ в язык AMN, абстрактные синтаксисы обоих языков представляется в виде моделей уровня М2, конформных модели *Ecore*. Трансформация позволит осуществлять преобразование произвольных моделей уровня М1, конформных модели СИНТЕЗ уровня М2, (т.е. спецификаций языка СИНТЕЗ) в соответствующие модели уровня М1, конформные модели AMN уровня М2, (т.е. спецификации языка AMN).

# Формализация абстрактного синтаксиса языка AMN

Формализация абстрактного синтаксиса языка AMN производится в соответствии с документом [18], описывающим конкретный синтаксис языка AMN. В результате получается модель уровня M2, конформная модели *Ecore,* представляемая в виде набора классов, атрибутов и ссылок на существующие классы. Для простоты объяснения описание класса в данной работе выглядит следующим образом:

class\_name\_1(

attribute\_name\_1: class\_name\_2\*,

attribute\_name\_2: class\_name\_3+,

attribute\_name\_3: class\_name\_4?,

attribute\_name\_4: parent\_class\_name{2}

) : parent\_class\_name;

class\_name\_2();

class\_name\_3();

class\_name\_4();

parent\_class\_name();

Класс представляется своим именем и набором атрибутов. Он также может быть наследован от родительского класса. Данная спецификация классов выглядит в *Ecore* в виде следующего фрагмента XML-кода:

<eClassifiers xsi:type="ecore:EClass" name="class\_name\_1"

eSuperTypes="#/parent\_class\_name">

<eStructuralFeatures xsi:type="ecore:EReference"

name="attribute\_name\_1" upperBound="-1"

eType="#/class\_name\_2"/>

<eStructuralFeatures xsi:type="ecore:EReference"

name="attribute\_name\_2" lowerBound="1"

upperBound="-1" eType="#/class\_name\_3"/>

<eStructuralFeatures xsi:type="ecore:EReference"

name="attribute\_name\_3"

eType="#/class\_name\_4"/>

<eStructuralFeatures xsi:type="ecore:EReference"

name="attribute\_name\_4" lowerBound="2"

upperBound="2" Type="#/parent\_class\_name"/>

</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="class\_name\_2"/>

<eClassifiers xsi:type="ecore:EClass" name="class\_name\_3"/>

<eClassifiers xsi:type="ecore:EClass" name="class\_name\_4"/>

<eClassifiers xsi:type="ecore:EClass" name="parent\_class\_name"/>

## Конструкции языка AMN

Основной синтаксической конструкцией языка являются компоненты. Существуют три вида компонентов: абстрактная машина, уточнение и реализация. При построении модели для компонентов был выделен корневой класс - *Machine* - инкапсулирующий общие для всех компонентов элементы. Данный класс специфицируется следующим образом:

Machine(

sees: AbstractMachine\*,

extendsClause: AbstractMachine\*,

sets: Set\*,

promotes: Operation\*,

properties: Predicate?,

invariant: Predicate?,

assertions: Predicate?,

initialization: Substitution?,

operations: Operation\*,

concreteConstants: NamedConstant\*

concreteVariables: Variable\*

) : Element;

Знак ‘\*’ означает, что может быть ноль и более элементов данного класса. Знак ‘?’ означает, что данный атрибут может как присутствовать, так и отсутствовать. Класс *Machine* наследуется от класса *Element*, чья спецификация выглядит следующим образом:

Element(

name : String

)

От класса *Machine* наследуются классы компонентов:

1) Абстрактная машина:

AbstractMachine(

Constraints: Predicate?,

Includes: AbstractMachine\*,

usesClause: AbstractMachine\*,

abstractConstants: Element\*,

abstractVariables: Element\*

) : Machine;

2) Уточнение:

Refinement(

includes: AbstractMachine\*,

abstractConstants: Element\*,

abstractVariables: Element\*

) : Machine;

3) Реализация:

Implementation(

imports: AbstractMachine\*,

values: Valuation?

) : Machine;

Описание секций атрибутов данных классов приведено в таблице 1.

Таблица 1 – Разделы абстрактной машины

|  |  |
| --- | --- |
| CONSTRAINTS | Определение типа и свойств формальных скалярных параметров |
| SEES | Список экземпляров видимых машин |
| INCLUDES | Список экземпляров включенных машин |
| PROMOTES | Список используемых операций включенных абстрактных машин |
| EXTENDS | Список экземпляров расширяемых машин |
| USES | Список экземпляров используемых машин |
| SETS | Абстрактные множества, либо определение перечислимых множеств |
| CONCRETE\_CONSTANTS | Список конкретных констант |
| ABSTRACT\_CONSTANTS | Список абстрактных констант |
| PROPERTIES | Тип и свойства машинных констант |
| CONCRETE\_VARIABLES | Список конкретных переменных |
| ABSTRACT\_VARIABLES | Список абстрактных переменных |
| INVARIANT | Тип и свойства переменных |
| ASSERTIONS | Определение свойств, выводимых из инварианта |
| INITIALISATION | Инициализация переменных |
| OPERATIONS | Список и определение операций |
| IMPORTS | Список экземпляров используемых машин |
| VALUES | Секция присваивания значения константе, либо отложенному множеству. |

Разделы IMPORTS, SEES, INCLUDES, EXTENDS, или USES являются секциями видимости. С их помощью строятся связи между различными компонентами системы.

Рассмотрим далее классы *Set*, *Operation*, *NamedConstant*, *Variable*, *Valuation*, являющиеся типами атрибутов секций рассмотренных ранее классов.

Set () : Element;

Deferred () : Set;

Enum (identifiers: String\*) : Set;

Задается корневой класс множества, от которого наследуются два класса множеств: перечислимое и отложенное. У перечислимого множества атрибутами является список идентификаторов.

Variable(

name: String

) : Expression;

Класс *Variable* представляется классом с единственным атрибутом - именем переменной. Спецификация класса *NamedConstant* аналогична классу *Variable.*

Operation(

substitution: Substitution{1},

inputParams: Variable\*,

outputParams: Variable\*

) : Element;

Атрибуты класса *Operation* представляют собой атрибут подстановку, атрибуты входных и выходных параметров типа переменная.

Класс *Valuation* специфицируется следующим образом:

Valuation(

valueName: String,

expression: Expression

);

Переменной с именем *valueName* присваивается значение выражения *expression.*

## Выражения

Выражение является основным конструкторским элементом языка. Оно является формулой, которая определяет элемент данных. В языке AMN выражения разбиты на семейства, объединенные по типу данных использующихся в выражении элементов. Например, простейшие выражения, арифметические выражения, выражения над множествами и т.д.

Приведем примеры выражений, наиболее часто использующихся при построении отображения.

### Бинарное отношение

Бинарное отношение определяет отношение двух множеств, которое в свою очередь является множеством. Так, отношение R множества E к множеству F это множество пар (x ↦ y), где x принадлежит E и y принадлежит F. Пример бинарного соответствия:

rel1 = {(0 ↦ FALSE), (1 ↦ TRUE), (2 ↦ FALSE), (3 ↦ TRUE), (4 ↦ FALSE), (5 ↦TRUE)}

Определяет соответствие между множеством целых чисел от одного до пяти и булевым выражением.

### Пересечение множеств

Пересечение множеств определяет множество элементов, принадлежащих обоим множествам. Так, для множеств E1 и E2:

E1 = {-1, 0, 3, 7, 8} и E2 = {-3, -1, 4, 7, 9},

E1∩F1 = {-1, 7}

### Переопределение отношений

Выражение вида R1 <+ R2 определяет переопределение отношения R1 отношением R2. Для отношений R1 и R2:

R1 = {(2 ↦ 1), (2 ↦ 8), (3 ↦ 9), (4 ↦ 7), (4 ↦ 9)} и R2 = {(0 ↦ -1), (1 ↦ 7), (2 ↦ 9)},

R1 <+ R2= {(0 ↦ -1), (1 ↦ 7), (2 ↦ 9), (3 ↦ 9), (4 ↦ 7), (4 ↦ 9)}

### Частичная функция

Частичная функция одного множества в другое – функция, определенная не для всех элементов первого множества.

### Полная функция

Полная функция – это функция, определенная для всех элементов первого множества.

Иллюстрация частичной и полной функции:

If r1 = {(0 ↦ 1), (1 ↦ 2), (2 ↦ 2)},

then r1 {0, 1, 2, 3} ⇸ {0, 1, 2}

and r1 {0, 1, 2} → {0, 1, 2}

При моделировании было решено абстрагироваться от стандартной классификации и объединить выражения по более общему признаку. Было выделено три больших класса выражений, объединенных общими свойствами:

* унарный оператор
* бинарный оператор
* функциональное выражение

Спецификация классов выражений строится в соответствии с выбранной стратегией. На вершине иерархии находится класс *Expression* от которого наследуются все остальные классы.

OperationalExpression(

sign: String

) : Expression;

UnaryOperator(

expression: Expression

) : OperationalExpression;

BinaryOperator(

expression: Expression{2}

) : OperationalExpression;

FunctionalExpression(

expression: Expression+,

sign: String

) : Expression;

Знак ‘+’ показывает, что экземпляров данного типа должно быть не менее одного.

Примером унарного оператора может являться унарный минус.

Примерами бинарного оператора являются арифметические выражения и выражения, определенные на множествах. В арифметические выражения входят операции сложения, умножения, вычитания, деления. В операции с множествами входят объединение, пересечение, разность.

Функциональные выражения представляются операциями определения максимального и минимального элемента множества, определения количества элементов множества.

Также был выделен класс простейших константных выражений: численных, булевых, строковых, множеств. В спецификации классов данный класс разбит на подклассы по типам.

BooleanValue(

value: EBooleanObject

) : Expression;

StringValue(

value: String

) : Expression;

IntegerValue(

value: EIntegerObject

) : Expression;

SetValue(

values: Expression\*

) : Expression;

## Предикаты

Составляющими элементами разделов PROPERTIES, INVARIANT, ASSERTIONS являются предикаты. В языке AMN предикат является формулой, которая может быть доказана либо опровергнута. Предикаты используются для выражения свойств данных или для задания условий применения подстановок.

Все многообразие предикатов языка может быть разбито на два класса:

* атомарный предикат
* составной предикат

Атомарный предикат состоит из выражений, объединенных каким-либо условием.

Составной предикат состоит из предикатов, объединенных отрицанием, конъюнкцией, дизъюнкцией, импликацией либо эквивалентностью.

Секция INITIALISATION представляет собой подстановку, инициализирующую переменные абстрактной машины.

Спецификация классов предикатов представляется в следующем виде.

На вершине иерархии классов предикатов находится класс *Predicate.* Все остальные классы наследуются от него. Данный класс не имеет атрибутов.

Predicate ();

Далее специфицируются формулы с кванторами. Для них создается общий класс, от него наследуются класс для формул с квантором существования и класс для формул с квантором всеобщности:

QuantifiedPredicate(

predicate: Predicate,

variables: Variable\*

) : Predicate;

UniversallyQuantified() : QuantifiedPredicate;

ExistentiallyQuantified() : QuantifiedPredicate;

Для корневого класса специфицируется атрибут предикат и атрибут переменных.

Далее рассматривается атомарный предикат:

AtomicPredicate(

expression: Expression\*,

sign: String

) : Predicate;

В атрибут *sign*  записывается знак, определяющий предикат.

Составные предикаты разделены каждый на свой класс для более удобного построения трансформации. Приведем спецификацию составного предиката на примере конъюнкции:

Conjunction(

predicate: Predicate{2}

) : Predicate;

Запись {2} означает, что экземпляров данных атрибутов должно быть ровно два. Остальные составные предикаты строятся по аналогичной схеме.

## Подстановки

Каждая подстановка специализируется отдельным классом, унаследованным от класса *Substitution.* Его спецификация аналогична спецификации класса *Predicate.*

Рассмотрим наиболее употребляемые в отображении подстановки.

### Подстановка присваивания

При помощи данной подстановки возможно присваивание переменной какого-либо выражения. Возможны разные варианты применения данной подстановки: применение к одной переменной, применение к списку переменных, применение к функции, применение к записи.

Подстановка специфицируется следующим образом:

BecomesEqual(

leftExpression: Expression+;

rightExpression: Expression+;

) : Substitution;

Обозначение + показывает, что экземпляров данного типа должно быть не менее одного.

Рассмотрим примеры применения данной подстановки:

x2 := x1 + 1 ;

tab1 := {(0 ↦ 3), (1 ↦ 1), (2 ↦ -7)} ;

tab1 (1) := 12 ;

tab2 := tab3 ;

y1, y2, y3 := 0, 0, 0 ;

z1, z2 := z2, z1 ;

### Подстановка с предусловием

Подстановка с предусловием используется для задания предусловий перед вызовом операций. Операция вызывается только в том случае, если предусловие является истиной. Класс, определяющий данную подстановку, строится таким образом:

Precondition(

pre: Predicate,

thenPart: Substitution

) : Substitution;

Пример данной подстановки:

PRE

x1 ∈NAT1

THEN

x1 := x1 - 1

END

### Подстановка неограниченного выбора

В общем виде данная подстановка выглядит следующим образом:

ANY X WHERE P THEN S END

где X является непустым списком переменных, чьей областью видимости является подстановка S и предикат P. Переменные доступны только на чтение в подстановке S. Переменные должны быть типизированы в предикате P с использованием конъюнкций.

Данная подстановка позволяет использовать в подстановке S абстрактные данные, объявленные в списке X и удостоверяющие предикат P и специфицируется своим классом следующим образом:

Any(

any: Variable+,

where: Predicate,

thenPart: Substitution

) : Substitution;

Пример подстановки неограниченного выбора:

ANY r1, r2 WHERE

r1 ∈NAT ∧

r2 ∈NAT ∧

r12 + r22 = 25

THEN

SumR := r1 + r2

END

### Последовательная подстановка

Последовательная подстановка отвечает за последовательное выполнение двух подстановок.

Спецификация класса данной подстановки:

SequenceSubst(

substitution: Substitution{2}

) : Substitution;

Пример последовательной подстановки, меняющей местами значения двух переменных:

z := x ; x := y ; y := z

### Подстановка вызова операции

Подстановка вызова операции используется для применения операции путем замены формальных параметров на действительные параметры. Действительные входные параметры являются переменными, а действительные выходные параметры являются именами, ссылающимися на данные, которые можно изменять. Вызов операции определен в четырех разных формах, в зависимости от присутствия входных и выходных параметров. Спецификация класса:

Call(

variable: Variable\*,

actualParam: Expression,

operationName: String

) : Substitution;

Пример подстановки:

op1 ;

op2 (x0 + 1, TRUE) ;

res1, res2 ←op3 ;

res, flag ←op4 (x0)

### Параллельная подстановка

Параллельная подстановка отвечает за одновременное выполнение двух подстановок. Необходимо, чтобы подстановки могли выполняться независимо друг от друга. Данная подстановка обладает свойствами коммуникативности и ассоциативности. Спецификация класса данной подстановки аналогична спецификации последовательной подстановки. В следующем примере значения переменных меняются местами:

x := y ||

y := x

### Тождественная подстановка

Identity() : Substitution;

### Блочная подстановка

Block(

substitution: Substitution

) : Substitution;

### Подстановка с утверждением

Assertion(

pre: Predicate,

thenPart: Substitution

) : Substitution;

### Подстановка ограниченного выбора

LimitedChoise(

substitution: Substitution+

) : Substitution;

### Условная подстановка

Conditional(

if: IfSubstitution+,

thenPart: Substitution

) : Substitution;

IfSubstitution(

if: Predicate,

else: Substitution

) : Substitution;

### Условная подстановка с ограниченным выбором

ConditionalBoundedChoice(

select: Select+,

else: Substitution

) : Substitution;

Select(

predicate: Predicate,

substitution: Substitution

) : Substitution;

Следующая подстановка является особым видом подстановки ограниченного выбора:

Case(

case: Expression,

branch: CaseBranch+,

else: Substitution

) : Substitution;

CaseBranch(

constant: NamedConstant,

substitution: Substitution

) : Substitution;

### Подстановка локального определения

Let(

variable: Variable+,

predicate: Predicate,

in: Substitution

) : Substitution;

### Подстановки замены переменной

Следующая подстановка заменяет переменную значением какого-либо множества:

BecomesElement(

variable: Variable+,

expression: Expression

) : Substitution;

Данная подстановка заменяет переменную значениями, удовлетворяющими предикату:

BecomesSuchThat(

variable: Variable+,

predicate: Predicate

) : Substitution;

### Подстановка определения новой переменной

Var(

variable: Variable+,

substitution: Substitution

) : Substitution;

### Подстановка цикла с условием

While(

while: Predicate,

do: Substitution,

invariant: Predicate,

variant: Expression

) : Substitution;

# Построение трансформации языка СИНТЕЗ в язык AMN

Трансформация представляет собой модуль языка ATL. Модуль состоит из заголовка (включающего имя модуля и имена переменных, соответствующих исходной и целевой моделям) и множества правил, определяющих способ построения элементов целевой модели на основе элементов исходной модели. Так называемые *сопоставляющие правила* (matched rules), составляющие ядро трансформации, позволяют описывать:

* какой элемент исходной модели должен быть взят;
* количество и тип порождаемых элементов целевой модели;
* способ, при помощи которого элементы целевой модели инициализируются на основании элементов исходной модели.

Так, в правиле *SynthesisModule2AMNContextAbstractMachine*

**rule** SynthesisModule2AMNContextAbstractMachine {

**from** m: Synthesis!Module

**to** am: AMN!AbstractMachine(

name <- ‘Context’ + m.name

)

}

входным элементом является элемент типа *Module,* порождается один элемент типа *AbstractMachine*, и имя абстрактной машины формируется из имени с прибавлением к нему приставки ‘Context’. Таким образом, мы породили контекстную абстрактную машину для модуля.

Наряду с сопоставляющими правилами, в языке ATL существуют вызываемые правила (called rules) – специальные правила ATL для явной генерации элементов целевой модели, которые можно вызывать из императивных блоков сопоставляющих правил. К отдельному классу относятся вспомогательные правила (helpers), позволяющие задавать факторизованный ATL код, носящий вспомогательный характер, который можно вызывать из разных частей модуля. Вспомогательные правила не могут генерировать элементы целевой модели. У вспомогательного правила может быть задан контекст, определяющий для какого элемента исходной модели правило может быть вызвано. Если контекст не задан, то обращение к правилу происходит в контексте всего модуля с использованием ключевого слова thisModule. Из сопоставляющих правил можно выделить отдельный подкласс так называемых *ленивых правил* (*lazy rules*). Для того, чтобы ленивое правило отработало, его необходимо явно вызвать из другого правила.

Для построения трансформации языка СИНТЕЗ в язык AMN, абстрактные синтаксисы обоих языков представляется в виде моделей уровня М2, конформных модели *Ecore*. Трансформация позволит осуществлять преобразование произвольных моделей уровня М1, конформных модели СИНТЕЗ уровня М2, (т.е. спецификаций языка СИНТЕЗ) в соответствующие модели уровня М1, конформные модели AMN уровня М2, (т.е. спецификации языка AMN).

## Основные принципы отображения

Построение отображения языка СИНТЕЗ в формальный язык спецификаций AMN было осуществлено в работе [16].

Модуль, как основная единица определения канонической модели, является единицей отображения канонической модели в AMN. Модуль представляется в AMN набором абстрактных машин, состоящим из контекстной машины и машин, соответствующих АТД модуля. Контекстная машина содержит информацию, характеризующую модуль, как целое, машины типов содержат информацию, характерную для отдельных типов. Таким образом, для модуля канонической модели, , где контекстная машина, множество машин, соответствующих АТД.

### Экстенсиональная интерпретация АТД

Моделирование АТД в AMN, также, как и формальная семантика АТД, базируется на экстенсиональной принципе: тип моделируется константным множеством своих экземпляров. Такое множество экземпляров называется *экстенсионалом* типа. В отличие от понятия экстенсионала в формальной семантике канонической модели, экстенсионал типа в AMN представляет собой конечное множество *потенциальных* значений типа. Атрибутные функции не определены изначально на потенциальных значениях. При необходимости получить значение типа с определенными значениями атрибутов, из экстенсионала типа выбирается произвольный незанятый элемент и атрибутные функции доопределяются на нем требуемым образом. Такой подход, в отличие от прямого моделирования в AMN семантики АТД, позволяет избежать бесконечности экстенсионалов и значительно упрощает моделирование функций в AMN операциями абстрактных машин. При этом упрощается процесс автоматического и (или) интерактивного доказательства уточнения.

Потенциальные экземпляры всех абстрактных типов моделируются предопределенным множеством *AVAL* и для каждого типа *T* с экстенсионалом выполнено . Экстенсионалы типов соотносятся в соответствии с отношением тип-подтип; для каждой пары типов , , что является подтипом , выполнено отношение включения на экстенсионалах . Таким образом моделируется иерархия типов.

Заметим, что в общем случае отношение тип-подтип сводится к отношению уточнения типов, и факт уточнения для всех пар тип-подтип, входящих в модуль, является одним из свойств непротиворечивости спецификации модуля. Поэтому непротиворечивость отношения тип-подтип в спецификации канонической модели следует доказывать с использованием B-технологии, отображая его и его подтип в AMN раздельно. Для самого же отображения канонической модели в AMN достаточно экстенсиональной трактовки отношения тип-подтип: имея экстенсиональное представление отношения тип-подтип в AMN и доказательство уточнения соответствуюзих типов, проведенное с использованием B-технологии, можно утверждать, что отношение тип-подтип моделируется в AMN корректно.

Представление множества своих потенциальных экземпляров позволяет естественным образом моделировать спецификацию типа набором функций AMN: атрибут *a*  типа T моделируется функцией , где тип значений, которые может принимать атрибут *a*. Экстенсиональные константы позволяют также строго типизировать переменные в инвариантах и операциях абстрактных машин. Например, для того, чтобы передать значение некоторого типа в качестве параметра в операцию машины, достаточно типизировать переменную-параметр соответствующим экстенсионалом.

Потенциальные экземпляры всех объектных типов моделируются предопределенным множеством . Для обеспечения уникальной идентификации объектов заводится множество объектных идентификаторов *OID*. Биекция моделирует неявный атрибут **self** каждого объектного типа.

### Моделирование структуры спецификации типов при помощи средств композиции абстрактных машин

Спецификации типов канонической модели могут ссылаться друг на друга в спецификациях атрибутов, методов, инвариантов, образуя структуру спецификации типов. Основным принципом отображения структуры спецификации типов канонической модели в AMN является стремление отобразить каждый АТД отдельной абстрактной машиной. Структура спецификации типов находит свое отражение в спецификации абстрактных машин следующим образом:

* абстрактная машина включается в секцию SEES абстрактной машины если методы типов АТД *i* ссылаются на атрибуты, методы типов из АТД *j*
* абстрактная машина включается в секцию USES абстрактной машины если инварианты типов АТД *i* ссылаются на атрибуты, методы типов из АТД *j*
* абстрактная машина включается в секцию INCLUDES абстрактной машины если инварианты типов АТД *i* изменяют атрибуты типов из АТД *j*
* абстрактная машина включается в секцию EXTENDS абстрактной машины если АТД *j* является непосредственным супертипом АТД *i*

## Построение контекстной машины модуля

Контекстная машина модуля *M* получает имя *ContextM,* и содержит разделы SETS, CONSTANTS, PROPERTIES. Основное содержание контекстной машины составляют экстенсионалы типов и отношения между экстенсионалами.

Для каждого типа *T ∊ Types(M)* вводится константное множество *ext\_T*, моделирующее полный экстенсионал и константное множество *extp\_T,* моделирующее собственный экстенсионал. Иерархия типов выстраивается при помощи следующего соотношения, записываемого в раздел PROPERTIES:

*;* длякаждой пары типов , , что является подтипом

В раздел PROPERTIES также помещается условие пустого пересечения собственных экстенсионалов абстрактных типов:

Здесь , … , обозначают все абстрактные типы модуля.

Множества и константы, определенные в *ContextM,* включают:

* множество *AVAL*, элементы которого моделируют потенциальные экземпляры всех абстрактных типов;
* константу *Obj,* типизированную в разделе PROPERTIES как *Obj∊ℙ(AVAL),* обозначающую множество, моделирующее совокупность экстенсионалов объектных типов;
* множество *OID* объектных идентификаторов;
* константу *self,* устанавливающую взаимно-однозначное соответствие между экземплярами объектных типов и объектными идентификаторами и типизированную в разделе PROPERTIES как
* константы *ext\_T, extp\_T* для каждого типа *T*, типизированные в разделе PROPERTIES как

в случае если тип объектный; и типизированная как

Контекстная машина содержит информацию, характеризующую модуль как целое, и поэтому каждая из машин, соответствующих АТД, включает контекстную машину в свой раздел SEES.

**Пример 1. Отображение спецификации модуля в контекстную машину**

Построим отображение модуля в контекстную машину на примере модуля FundingAgency.

{ FundingAgency\_module; in: module;  
type:  
{ Person; in: type;  
 name: string;  
},  
{ Researcher; in: type;  
 supertype: Person;  
 research\_area: string;  
}  
}

В примере представлен модуль FundingAgency, в котором определены два абстрактных типа данных – АТД Person и АТД Researcher. В АТД Person определен атрибут name типа строка, который определяет имя индивида. Из спецификации АТД Researcher видно, что данный АТД является подтипом АТД Person и имеет атрибут *research\_area*, устанавливающий сферу интересов данного исследователя. Стоит отметить, что АТД Researcher наследует атрибут *name* от своего супертипа.

Модулю в соответствие ставится следующая абстрактная машина:

MACHINE FundingAgency\_Context

SETS

AVAL; OID

CONSTANTS

Obj, self,

extp\_Person, ext\_Person,

extp\_Researcher, ext\_Researcher

PROPERTIES

Obj: POW(AVAL) & self: Obj >->> OID &

extp\_Person /\ extp\_Researcher = {} &

extp\_Person : POW(Obj) & ext\_Person : POW(Obj) &

extp\_Person <: ext\_Person &

extp\_Researcher: POW(Obj) & ext\_Researcher: POW(Obj) &

extp\_Researcher <: ext\_Researcher & ext\_Researcher <<: ext\_Person

END

Данное отображение осуществляется при помощи следующей правила трансформации:

**rule** SynthesisModule2AMNAbstractMachine {

**from**

m : Synthesis!ModuleDef(m.oclIsTypeOf(Synthesis!ModuleDef))

**to**

am : AMN!AbstractMachine (

name <- 'Context' + m.name,

sets <- Sequence {avalSet},

abstractConstants <- Sequence {objConst},

properties <- propertiesPredicate),

avalSet : AMN!Deferred (name <- 'AVAL'),

objConst : AMN!NamedConstant (name <- 'Obj' ),

propertiesPredicate : AMN!Conjunction (

predicate <- Sequence {objProp}

),

objProp : AMN!AtomicPredicate (

sign <- ':',

expression <- objConst,

expression <- rightAval

),

rightAval : AMN!FunctionalExpression (

sign <- 'POW',

expression <- avalSet

)

}

На основании данного правила генерируется абстрактная машина, которой в раздел множеств записывается множество *AVAL,* в раздел констант записывается константа *Obj*. Следует отметить, что напрямую эти данные записывать нельзя, сначала происходит их инициализация, при которой создается объект определенного типа, а затем он записывается в соответствующий раздел. В разделах множеств и констант могут присутствовать более одного элемента, в таком случае можно воспользоваться конструкцией последовательности - *Sequence*, для которой через запятую записываются необходимые элементы. Также при помощи последовательности формируется предикат, накладывающий условия на константы*.* Необходимо заметить, что некоторые элементы отображения опущены, поэтому *objProp* не единственный конъюнкт. Элемент *objProp* представляет собой атомарный предикат принадлежности элемента множеству, в данном случае множеством является множество подмножеств *AVAL.* Знак операции задается при помощи символа ‘:’. Данный символ соответствует символу ‘∈’ и является служебным для средства поддержки B-технологии. Символы языка AMN отображаются в символы инструментария в соответствии с [20]. Данное правило создает контекстную машину не полностью. Правило для генерации констант, соответствующих экстенсионалам АТД и условия на них рассматривается далее:

**rule** abstractDataType2ContextMachineSections {

**from**

adt : Synthesis!ADTDef(adt.oclIsTypeOf(Synthesis!ADTDef))

**to**

extp : AMN!NamedConstant ( name <- 'extp\_' + adt.name ),

ext : AMN!NamedConstant ( name <- 'ext\_' + adt.name ),

conjunction : AMN!Conjunction (

predicate <- leftPred,

predicate <- middlePred,

predicate <- rightPred

),

leftPred : …, middlePred : …, rightPred : …

**do** { thisModule.resolveTemp(adt.typeInModule,'am').abstractConstants<- Sequence {extp, ext}; thisModule.resolveTemp(adt.typeInModule,'intersection').expression<- extp; thisModule.resolveTemp(adt.typeInModule,'propertiesPredicate').predicate <-conjunction;

}

}

Данное правило формирует две константы, обозначающие полный и собственный экстенсионалы типа. Также формируется предикат, типизирующий эти константы формулой вида

Интересна секция *do*. Она позволяет использовать императивную семантику при описании трансформаций. В данном случае, она несет в себе и операцию *resolveTemp,* являющуюся неотъемлемым инструментом сложных трансформаций. Данная операция позволяет обращаться из правил языка ATL к любому элементу целевой модели, который будет сгенерирован из исходной модели при помощи сопоставляющего правила.

В данном случае при помощи первой операции *resolveTemp* мы обращаемся к контекстной абстрактной машине, полученной на основании спецификации модуля, и записываем константные множества, моделирующие полный и собственный экстенсионал в раздел абстрактных констант. При помощи второй операции мы обращаемся к элементу *intersection*, являющемуся бинарным оператором, и записываем в выражение константное множество, моделирующее полный экстенсионал. Таким образом, мы получили условие пустого пересечения собственных экстенсионалов. При помощи третьей операции мы записываем формулу, типизирующую эти константы в элемент *propertiesPredicate,* являющийся конъюнкцией, записываемой затем в раздел PROPERTIES контекстной машины.

## Структура абстрактной машины, соответствующей типу: моделирование атрибутов, методов, инвариантов

Рассмотрим, каким образом моделируются атрибуты, методы, инварианты некоторого типа *T* в абстрактной машине *M­T*, инкапсулирующей отображение спецификации типа *T* в AMN.

Для каждого атрибута *a : Ta* типа *T* машина *MT* содержит переменную *a* в разделе VARIABLES. Переменная *a* типизируется в разделе INVARIANT предикатом языка AMN: , где является типом атрибута *a*.

В случае, если атрибут имеет тип boolean, раздел SEES машины *MT* включает предопределенную в AMN машину *Bool\_TYPE*, если атрибут имеет тип string машину *String\_TYPE.*

Методы модификации атрибутов моделируются операциями set\_имя\_атрибута(s) машины *MT* в разделе OPERATIONS. Параметр s представляет собой множество пар , где *o* – значение типа *T, v* – значение, которое должно быть сообщено соответствующему атрибуту значения *o* операцией *set.* В теле операции используется обобщенная подстановка c предварительным условием, где предварительным условием является принадлежность параметра *s* частичной функции , где является типом атрибута *a*. Подстановкой является постановка присваивания определяемому атрибуту параметра *s.*

Инициализация атрибутов осуществляется при помощи подстановки ANY в разделе INITIALIZAITON. В раздел WHERE подставляется предикат , аналогичный предикатам типизации атрибутов.

Инициализирующие подстановки атрибутов соединяются между собой оператором параллельной подстановки ∥.

Поясним данное отображение на примере.

**Пример 2. Моделирование атрибутов АТД**

Рассмотрим спецификацию АТД Dispatcher:

{ Dispatcher; in: type;

experts: {set; type\_of\_element: Specialist;};

experts\_got: boolean;

experts\_speciality: string;

};

В данном АТД определены три атрибута различных типов. Далее идет представление этого АТД в виде абстрактной машины.

MACHINE Dispatcher

SEES

String\_TYPE,

Bool\_TYPE,

FundingAgency\_Context

VARIABLES

experts\_speciality

experts\_got,

experts

INVARIANT

experts\_speciality : ext\_Dispatcher --> STRING &

experts\_got ∈ ext\_Dispatcher --> BOOL &

experts ∈ ext\_Dispatcher --> POW(ext\_Specialist)

INITIALISATION

ANY arelevant\_experts WHERE

arelevant\_experts : ext\_Dispatcher --> STRING

THEN relevant\_experts:= arelevant\_experts

END ||

ANY aexperts\_got WHERE aexperts\_got : ext\_Dispatcher --> BOOL

THEN experts\_got:= aexperts\_got

END ||

ANY aexperts WHERE aexperts : ext\_Dispatcher --> POW(ext\_Specialist)

THEN experts:= aexperts

END

OPERATIONS

set\_experts\_speciality(sav) =

PRE sav : ext\_Dispatcher +-> STRING

THEN relevant\_experts:= relevant\_experts <+ sav

END;

set\_experts\_got(sav) =

PRE sav : ext\_Dispatcher +-> BOOL

THEN experts\_got:= experts\_got <+ sav

END;

set\_experts(sav) =

PRE sav : ext\_Dispatcher +-> POW(ext\_Specialist)

THEN experts:= experts <+ sav

END

END

Следующее правило создает абстрактную машину для АТД и обрабатывает отношения тип-подтип, если таковые имеются.

**rule** abstractDataType2ADTAbstractMachine {

**from**

adt : Synthesis!ADTDef(adt.oclIsTypeOf(Synthesis!ADTDef))

**to**

am : AMN!AbstractMachine (

name <- adt.name,

initialization <- initSubt,

invariant <- predicate,

sees <- thisModule.resolveTemp(adt.typeInModule, 'am')

),

predicate : AMN!Conjunction (),

initSubt : AMN!Simultaneous ()

**do** {

**if** (adt.subtypes->notEmpty()) {

**for** (e **in** adt.subtypes) { thisModule.genStrictInclusion(adt,e);

}

}

**if** (adt.supertypes->notEmpty()) {

**for** (e **in** adt.supertypes) {

am.extendsClause <- thisModule.resolveTemp(e, 'am');

}

}

}

**rule** genStrictInclusion(adt : Synthesis!ADTDef,

subType : Synthesis!ADTDef) {

**to**

strictInclusionSubtypeRelation : AMN!AtomicPredicate (

sign <- '<:',

expression <- Sequence{

thisModule.resolveTemp(subType,'ext'), thisModule.resolveTemp(adt, 'ext')

}

)

**do** {

thisModule.resolveTemp(adt.typeInModule,'propertiesPredicate ').predicate <-strictInclusionSubtypeRelation;

}

}

Имя машины задает имя абстрактного типа данных, для разделов INITIALIZATION, INVARIANT их содержимое будет известно на этапе трансформации атрибутов. Так как инвариант представляется конъюнкцией, для него создается конъюнкция, инициализирующие подстановки соединяются между собой при помощи оператора параллельной подстановки, для чего был создан соответствующий элемент. В императивной секции при помощи конструкции *for* для АТД, имеющих подтипы, производится генерация атомарного предиката включения константного множества экстенсионалов подтипа в множество экстенсионалов супертипа. Генерация производится при помощи вызываемого правила, на вход которому подаются АТД типа и его супертипа. В теле правила при помощи конструкций *resolveTemp* берутся экстенсионалы, и записываются в атомарный предикат. В императивной секции данная подстановка записывается в предикат раздела PROPERTIES модуля, содержащего эти типы. При помощи второй конструкции *for* происходит итерация по всем супертипам данного АТД и машина, сгенерированная для супертипа записывается в раздел EXTENDS абстрактной машины подтипа. При трансформации атрибутов будем считать, что атрибуты абстрактного типа данных не являются функциональными. Правила для функциональных атрибутов рассматриваются в следующих разделах. Рассмотрим правило, при помощи которого происходит наполнение разделов абстрактной машины, созданной для АТД. Для примера было взято правило формирования операции. Операция представляет метод модификации переменной и строится следующим образом:

**rule** attribute2ATDMachineOperation {

**from**

at : Synthesis!AttributeDef(at.isNotFunctional())

**to**

inpVar : AMN!Variable (

name <- 'sav'

),

attrVar : AMN!Variable (

name <- at.name

),

operation : AMN!Operation (

name <- 'set\_' + at.name,

inputParams <- inpVar,

substitution <- subst

),

subst : AMN!Precondition (

pre <- predicate,

thenPart <- substitution

),

predicate : AMN!AtomicPredicate (

sign <- ':',

expression <- inpVar,

expression <- partialFunction

),

partialFunction : AMN!BinaryOperator (

sign <- 'PartialFunction',

expression <- thisModule.resolveTemp(at.attributeOf, 'ext')

),

substitution : AMN!BecomesEqual (

leftExpression <- attrVar,

rightExpression <- overWrite

),

overWrite : AMN!BinaryOperator (

sign <- '<+',

expression <- attrVar,

expression <- inpVar

)

**do** {

thisModule.resolveTemp(at.attributeOf, 'am').operations <- operation;

}

}

Заголовок операции состоит из имени операции, формируемом при помощи конкатенации строки ‘set’ и имени атрибута и входного параметра, значение которого должно быть сообщено соответствующему атрибуту. Тело операции составляет подстановка с предусловием. Сформированная операция заносится в раздел операций машины АТД.

Следует заметить, что данное правило применяется только к нефункциональным атрибутам. Это обеспечивается специфицированием условия *isNotFunctional()* в секции *from*, являющегося вызовом вспомогательного правила вида:

**helper** **context** Synthesis!AttributeDef **def**:isNotFunctional(): Boolean =

**if** self.oclIsTypeOf(Synthesis!AttributeDef)

**then** **if** **not** self.type.oclIsTypeOf(Synthesis!FunctionDef)

**then** true

**else** false

**endif**

**else** false

**endif**;

Генерация инициализации атрибутов и предикатов, специфицирующих атрибуты, производится аналогичным образом. Она реализована совместно с генерацией операции, так как использует одни и те же элементы.

**Пример 3. Отображение метода**

Метод *meth* типа *T* всегда имеет представление в AMN операцией *meth\_op* машины *MT*. Набор входных параметров метода отображается в набор входных параметров операции, аналогично поступаем с выходными параметрами. Также во входные параметры добавляется объект, для которого вызывается метод *meth.* Тело операции формирует подстановка с предусловием. В качестве предусловия выступает конъюнкция ограничений на параметры, так для объекта ограничение заключается в принадлежности этого объекта экстенсионалу АТД. Для других параметров записывается предикат принадлежности экстенсионалу типа данных параметра, либо просто типу, если тип простой. В конструкцию THEN подстановки записывается подстановка ANY. В секцию ANY записываются новые переменные, имена которых не имеют вхождений в формулу спецификации метода. Новые переменные соответствуют выходным параметрам метода, либо атрибутам, значение которых изменяется в теле метода при помощи операции *set*. В секцию WHERE записывается конъюнкция, состоящая из предикатов принадлежности новых переменных типам данных соответствующих им атрибутов и предикатом AMN, смоделированным по формуле спецификации метода. В секцию THEN записывается подстановка, присваивающая атрибуту значение соответствующей ему новой переменной. При применении трансформации прежде создается абстрактная машина, соответствующая АТД. На основании спецификации АТД может быть дополнена контекстная машина модуля, если АТД является подтипом другого АТД.

Возьмем уже использовавшийся нами АТД Dispatcher и расширим его двумя методами get\_experts и count\_experts.

{ Dispatcher;

in: type;

experts: {set; type\_of\_element: Specialist;};

experts\_got: boolean;

relevant\_experts: {set; type\_of\_element: Specialist;};

get\_experts:

{ in: function;

{ predicative:

{ this.experts~ = expert & this.experts\_got~ = true

}

};

};

count\_experts:

{in: function;

params: {-number\_of\_experts/integer};

{ predicative:

{ isempty(this.experts) & this.experts\_got~ = true &

number\_of\_experts = card(this.relevant\_experts)

}

};

};

};

Легко заметить, что оба метода являются методами с явными эффектами, т.е. изменяют состояние ИС. Так метод get\_experts изменяет атрибуты experts и experts\_got. А метод count\_experts изменяет значение атрибута experts\_got. Более того метод count\_experts также имеет один выходной параметр - number\_of\_experts типа integer.

Посмотрим, как будет выглядеть отображение в AMN данной спецификации. Опустим разделы абстрактной машины, описанные ранее, оставив раздел OPERATIONS, в котором и определяется функциональные атрибуты.

OPERATIONS

get\_experts(av)=

PRE av: ext\_Dispatcher

THEN

ANY v2, v1 WHERE

v2: BOOL & v1: POW(ext\_Specialist) &

(v1 = expert &

v2 = TRUE)

THEN

experts\_got(av):= v2;

experts(av):= v1

END

END;

number\_of\_experts <-- count\_experts(av)=

PRE av: ext\_Dispatcher

THEN

ANY v1, number\_of\_experts1 WHERE

v1: BOOL &

number\_of\_experts1: NAT &

(experts(av) = {} &

v1 = TRUE &

number\_of\_experts1 = card(relevant\_experts(av)))

THEN

experts\_got(av):= v1;

number\_of\_experts:= number\_of\_experts1

END

END;

Для метода формируется операция, конструкция которой аналогична конструкциям для нефункциональных атрибутов. Различие состоит в том, что в теле подстановки находится подстановка ANY, с помощью которой моделируется предикативная спецификация атрибута, вводятся новые переменные для атрибутов, меняющих свое значение в ходе работы функции, и ставятся условия на эти переменные. Далее приведен пример отображения:

**rule** functionalattributesToOperations {

**from**

at : Synthesis!AttributeDef(at.isFunctional())

**to**

operation : AMN!Operation (

…

substitution <- subst

…

),

…

subst : AMN!Precondition (

pre <- precondPredicate,

thenPart <- thenSubstAny

),

thenSubstAny : AMN!Any (

where <- whereConj,

thenPart <- sequenceSubstitution

)

**do** {

**for** ( e **in** at.type.parameters ) {

**if** ( e.type.oclIsTypeOf(Synthesis!ADTDef) ) {

thisModule.generateParameters(e, at);

}

}

thisModule.resolveTemp(at.attributeOf,'am').operations<-operation;

}

}

Из тела правила были убраны конструкции, аналогичные правилу для нефункционального атрибута. Секция ANY и WHEREподстановки ANY формируется на основании переменных, изменяющих свое значение, и наполняется в ходе правил обработки выходных параметров и функций, устанавливающих значение атрибута. Также в конструкцию WHERE заносится смоделированный предикат. Операцией *resolveTemp* данная операция заносится в раздел операций абстрактной машины АТД. Рассмотрим правило generateParameters, строящее входные и выходные параметры функции на основе параметров функции и правило generateAdditionalParam строящее дополнительные переменные:

**rule** generateParameters ( param : Synthesis!ParameterDef, attr :

Synthesis!AttributeDef ) {

**to**

var : AMN!Variable (

name <- param.name

),

varPred : AMN!AtomicPredicate (

sign <- ':',

expression <- var,

expression <- thisModule.resolveTemp(

param.type, 'ext')

)

**do** {

**if** ( param.parameterKind = 'input' ) {

thisModule.resolveTemp(attr,'operation').inputParams

<-var;

}

**else** {

**if** ( param.parameterKind = 'output' ) { thisModule.resolveTemp(attr,'operation').outputParams<-var;

thisModule.generateAdditionalParam(param,attr);

}

}

thisModule.resolveTemp(attr,'precondPredicate').predicate<-varPred;

}

}

**rule** generateAdditionalParam ( param : Synthesis!ParameterDef, attr : Synthesis!AttributeDef ) {

**to**

varAny : AMN!Variable (

name <- 'v' + thisModule.counter

),

varPred : AMN!AtomicPredicate (

sign <- ':',

expression <- varAny,

expression <- thisModule.resolveTemp(param.type, 'ext')

)

**do** {

thisModule.resolveTemp(attr,'thenSubstAny').any<-varAny;

thisModule.resolveTemp(attr,'whereConj').predicate<-varPred;

thisModule.counter <- thisModule.counter + 1;

}

}

Вызываемое правило generateParameters создает переменную и предикат принадлежности переменной экстенсионалу ее типа. В секции *do* на основании вида переменной она записывается во входные либо выходные параметры операции, а в раздел PRE подстановки с предусловием записывается предикат. Аналогично для правила generateAdditionalParam, только результаты записываются в подстановку ANY.

## Отображение формул канонической модели

В данном разделе рассматривается отображение формул предикатами AMN. Отображение строится индуктивно, начиная от атомарных предикатов и заканчивая составными формулами.

### Атомарные предикаты

Атомарные предикаты могут являться предикатами-классами, предикатами-методами и встроенными предикатами канонической модели.

Отображение встроенных предикатов same, in осуществляется следующим образом:

Правило трансформации представлено в следующем виде:

**rule** builtInPredicate2AtomicPredicate {

**from**

builtIn : Synthesis!BuiltInPredicate(builtIn.oclIsTypeOf(Synthesis!BuiltInPredicate))

**to**

atomPred : AMN!AtomicPredicate (

expression <- builtIn.terms

)

**do** {

**if** ( builtIn.symbol = 'in' ) {

atomPred.sign <- ':';

} **else** {

**if** ( builtIn.symbol = 'same' **or**

builtIn.symbol = 'is\_equal' ) {

atomPred.sign <- '=';

}

}

**if** ( builtIn.symbol = 'is\_empty' ) {

atomPred.sign <- '=';

thisModule.genEmptySet(builtIn);

}

}

}

При помощи императивных конструкций проверяется символ предиката и на его основе выбирается знак для атомарного предиката. Наряду с вышеупомянутыми предикатами данное правило также отображает встроенные предикаты is\_equal и is\_empty. Для предиката и is\_empty генерируется пустое множество при помощи вызываемого правила genEmptySet.

### Условия

Условия-отношения формируются при помощи операций-отношений

из выражений. Операции-отношения имеют аналоги, определенные на типах AMN, соответствующих целому, перечислимому, булевскому, строковому типам канонической модели. В случае, если операндами операции-отношения являются множества, семантикой операции-отношения является соответствующая операция, определенная на множествах AMN.

Операндами отношений могут быть также объекты АТД.

Данное отображение представляется правилом ATL следующим образом:

**rule** atomToAMNAtomicPredicate {

**from**

atomSynth : Synthesis!Atom(atomSynth.oclIsTypeOf(Synthesis!Atom))

**to**

atomicAMN : AMN!AtomicPredicate (

sign <- atomSynth.symbol,

expression <- atomSynth.terms

)

}

Так как модели данных конструкций почти аналогичны, трансформация состоит только из декларативной секции, напрямую отображающей элементы атомарного предиката языка СИНТЕЗ в атомарный предикат языка AMN.

Арифметические выражения формируются при помощи арифметических операций (+,-,\*,/) из атомарных термов. Арифметические операции имеют аналоги, определенные на типе *NAT*, и поэтому обладают естественной семантикой. Выражения-множества представляют собой атомарный терм.

Арифметические выражения отображаются в соответствии со следующим правилом:

**rule** arithmeticPredicate2BinaryOperator {

**from**

a : Synthesis!ArithmeticPredicate

(arithmPred.oclIsTypeOf(Synthesis!ArithmeticPredicate))

**to**

atomPred : AMN!BinaryOperator (

sign <- arithmPred.symbol,

expression <- arithmPred.terms

)

}

Рассмотрим теперь значения, которые может принимать атомарный терм и их отображение в AMN. Атомарным термом условия может быть значение (идентификатор, целое число, строка, логическое значение), имя переменной, путь в композиционной иерархии АТД, вызов метода, формула-выделение множества, имя класса. Типы значений канонической модели имеют аналоги в AMN, и поэтому значения интерпретируются естественным образом. Имя переменной тождественно отображается по её id. Отображение пути показано в разделе моделирования вызова метода. Выделение множества представляется следующим образом:

Правило, задающее данную трансформацию выглядит следующим образом:

**rule** collectionComprehension2SetComprehension {

**from**

synthCollCompr : Synthesis!CollectionComprehension

(synthCollCompr.oclIsTypeOf(Synthesis!CollectionComprehension))

**to**

amnSetCompr : AMN!SetComprehension (

variable <- synthCollCompr.vars,

predicate <- conjunction

),

conjunction : AMN!Conjunction (

predicate <- synthCollCompr.formula

)

**do** {

**for** ( var **in** synthCollCompr.vars) {

thisModule.genVarTyping( var, synthCollCompr );

}

}

}

Дополнительно для каждой переменной в конъюнкцию добавляется предикат принадлежности типу, генерируемый вызываемой функцией genVarTyping.

### Составные формулы

Составные формулы канонической модели могут быть образованы из атомарных предикатов и условий при помощи логических операций (&, |, ^, ->).

Логические операции в формулах канонической модели моделируются естественным для логики предикатов первого порядка образом:

Приведем пример правила на ATL, реализующий трансформацию конъюнкции:

**rule** conjunctionToAMNConjunciton {

**from**

conjSynth : Synthesis!Conjunction(conjSynth.oclIsTypeOf(Synthesis!Conjunction))

**to**

conjAMN : AMN!Conjunction (

predicate <- conjSynth.formula

)

}

Представление разрешенных в канонической модели формул с кванторами осуществляется следующим образом:

Для предиката с квантором существования:

Для предиката с квантором всеобщности:

Рассмотрим правило на ATL, реализующее данное отображение:

**rule** universllyQuantifiedToAMNUniversallyQuantified {

**from**

quanSynth : Synthesis!UniversallyQuantifiedFormula

(quanSynth.oclIsTypeOf(Synthesis!UniversallyQuantifiedFormula))

**to**

quanAMN : AMN!UniversallyQuantifiedFormula (

predicate <- Sequence {typedVarConj,quanSynth.formula},

variables <- quanSynth.variables

),

typedVarConj : AMN!Conjunction (

predicate <- quanSynth.variables->collect(e |

thisModule.quantVarsToTypedVarPredicate(e))

)

}

## Представление функций доступа

Построение трансформации для функций доступа осуществляется отдельными правилами. Рассмотрим правило для трансформации функции доступа get:

**rule** getFunction2AMNExpression {

**from**

g : Synthesis!GetFunction

(synthGetFunction.oclIsTypeOf(Synthesis!GetFunction))

**to**

f : AMN!FunctionalExpression (

sign <- 'GetFunction',

expression <- namedVar

),

namedVar : AMN!Variable (

name <- synthGetFunction.name

)

**do** {

**for** ( term **in** synthGetFunction.terms) {

**if** ( term.oclIsTypeOf(Synthesis!GetFunction) ) {

functionalExp.expression<-thisModule.resolveTemp(term,'f');

}

}

**if** ( **not** synthGetFunction.attribute.oclIsUndefined() ) {

**for** ( fA **in** Synthesis!AttributeDef.allInstances()->select(m|m.isFunctional) {

**for** ( a **in** fA.type.predicativeSpec.splitToAtoms()) {

**for** ( term **in** a.terms ) {

**if** ( term.oclIsTypeOf(Synthesis!GetFunction) ) {

**if** ( term.name = synthGetFunction.name ) {

**if** ( funcAttr.attributeOf <> synthGetFunction.attribute.attributeOf ) {

**if** ( **not** thisModule.resolveTemp(funcAttr.attributeOf, 'am').sees->

exists(m|m.name=thisModule.resolveTemp(g.attribute.attributeOf, 'am').name)

**and**

**not** thisModule.resolveTemp(fA.attributeOf, 'am').includes->

exists(m|m.name=thisModule.resolveTemp(g.attribute.attributeOf, 'am').name)

) {

thisModule.resolveTemp(fA.attributeOf, 'am').sees <-

thisModule.resolveTemp(g.attribute.attributeOf, 'am');

}…

}

Данное правило формирует функциональное выражение, которое участвует в отображении пути. При помощи операции *resolveTemp* происходит запись аналогичного выражения, созданного для терма функции, если этим термом также является функция доступа. Так строится структура пути, исходя от начального элемента. В секции *do* производится запись машины АТД, инкапсулирующего атрибут, в раздел SEES машины, представляющей АТД, который вызывает данную функцию.

Для функции доступа set правило выглядит следующим образом:

**rule** setFunction2AMNExpression {

**from**

s : Synthesis!SetFunction

(synthSetFunction.oclIsTypeOf(Synthesis!SetFunction))

**to**

var : AMN!Variable (

name <- 'v' + thisModule.counter

),

varType : AMN!AtomicPredicate (

sign <- ':',

expression <- var,

expression <-

thisModule.resolveTemp(s.attribute.attributeOf, 'ext')

),

varSubstBecomesEqual : AMN!BecomesEqual (

leftExpression <- functionalExp,

rightExpression <- var

),

functionalExp : AMN!FunctionalExpression (

sign <- 'SetFunction',

expression <- namedVar

),

namedVar : AMN!Variable (

name <- synthSetFunction.name

),

setExternalVar : AMN!Call (

operationName <- 'set\_' + s.attribute.name,

actualParam <- actualParamExp

),

actualParamExp : AMN!BinaryOperator (

sign <- '|->',

expression <- Sequence {functionalExp,var}

)

**do** {

…

}

}

Данное правило, также как и правило для функции get, формирует путь в иерархии АТД. Если АТД атрибута функции set отличается от АТД, в котором вызывается данная функция, то машина АТД этого атрибута записывается в раздел INCLUDES машины для АТД вызывающей функцию.

В секции *to* генерируется дополнительная переменная, типизирующаяся экстенсионалом типа, включающего данный атрибут. Таким образом, для каждого пути, изменяющего состояние системы, генерируется дополнительная переменная. Данная переменная записывается в раздел ANY подстановки, формирующей операцию для метода. Также строятся две подстановки: подстановка присваивания и подстановка вызова операции, выбор между которыми делается в зависимости от того, является ли данный атрибут атрибутом АТД, в котором вызывается данная функция, либо нет. Данные подстановки при помощи операции *resolveTemp* помещаются в раздел THEN подстановки ANY операции метода. Если атрибут инкапсулирован АТД вызвавший метод, то применяется подстановка присваивания, иначе применяется подстановка вызова операции. В теле подстановок используется сформированный ранее путь.

# Заключение

Для обеспечения возможности повторного использования существующего ресурса в как части интегрированной системы при композиционном проектировании информационных систем необходимо убедиться в уточнении спецификацией ресурса части спецификации системы. Для однородности спецификации ресурсов и спецификации интегрированной системы спецификации выражаются в единой канонической информационной модели.

В рамках работы на основе существующего отображения канонической информационной модели посредников (языка СИНТЕЗ) в язык AMN, поддерживающий уточнение, реализована трансформация языка СИНТЕЗ в язык AMN, с использованием декларативно-императивного языка трансформации моделей ATL. Также был формализован абстрактный синтаксис языка AMN.

Трансформация позволяет на основании спецификаций языка СИНТЕЗ автоматически порождать соответствующие спецификации языка AMN. Это позволяет доказывать уточнение спецификаций языка СИНТЕЗ при помощи средств автоматического и интерактивного доказательства языка AMN.

# Список литературы

1. *Kalinichenko L.A., Stupnikov S.A., Martynov D.O.* SYNTHESIS: a Language for Canonical Information Modeling and Mediator Definition for Problem Solving in Heterogeneous Information Resource Environments. Moscow: IPI RAN, 2007. - 171 p.
2. *Briukhov D.O., Kalinichenko L.A.* Component-based information systems development tool supporting the SYNTHESIS design method // Advances in Databases and Information Systems: Proc. of the Second East European Conference. – Berlin-Heidelberg: Springer-Verlag, 1998. – P. 305-327.
3. *Briukhov D.O., Kalinichenko L.A., Skvortsov N.A.* Information sources registration at a subject mediator as compositional development // Advances in Databases and Information Systems: Proc. of the 5th East European Conference. – Berlin-Heidelberg: Springer-Verlag, 2001. – P. 70-83.
4. *Kalinichenko L.A.* Method for Data Models Integration in the Common Paradigm // Advances in Databases and Information Systems: Proc. of the First East-European Conference. – St. Petersburg: Nevsky Dialekt, 1997. – P. 275-284.
5. *Kalinichenko L.A., Martynov D.O., Stupnikov S.A.* Query rewriting using views in a typed mediator environment // Advances in Databases and Information Systems: Proceedings of the East European Conference. – Berlin-Heidelberg: Springer-Verlag, 2004. – P. 37-53.
6. *Abrial J.-R.* The B-Book: Assigning Programs to Meanings // Cambridge: Cambridge University Press, 1996.
7. *Abrial J.-R.* B-Technology// Technical overview. – B-Core (UK) Ltd., 1993.
8. OMG Model Driven Architecture // http://www.omg.org/mda/
9. ATL Project // <http://www.eclipse.org/m2m/atl/>
10. OMG/OCL Object Constraint Language (OCL) 2.0. OMG Final Adopted Specification. ptc/03-10-14, 2003.
11. *Budinsky, F., Steinberg, D., Ellersick, R., Grose, T.* Eclipse Modeling Framework, Chapter 5 ”Ecore Modeling Concepts”. - Addison Wesley Professional, 2004.
12. Meta Object Facility (MOF) 2.0 Core Specification // <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf>, 2003.
13. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification // http://www.omg.org/cgi-bin/doc?ptc/2007-07-07, 2007.
14. *Kalinichenko L.A., Briukhov D.O., Martynov D.O., Skvortsov N.A., Stupnikov S.A.* Mediation Framework for Enterprise Information System Infrastructures. Proc. of the 9th International Conference on Enterprise Information Systems ICEIS 2007. -- Funchal, 2007. -- Volume Databases and Information Systems Integration. -- P. 246--251.
15. *Захаров В.Н., Калиниченко Л. А., Соколов И. А., Ступников С. А.* Конструирование канонических информационных моделей для интегрированных информационных систем // Информатика и её применения, 2007. Т. 1. Вып. 2. С. 15–38.
16. *Ступников С.А.* Отображение спецификаций, выраженных средствами ядра канонической модели, в язык AMN// Системы и средства информатики, Специальный Выпуск. -- Москва: ИПИ РАН, 2005. -- С. 69--95
17. Eclipse // <http://www.eclipse.org/>
18. B language reference manual. Version 1.8.6.// ClearSy, 2009.
19. Eclipse Modeling Framework // <http://www.eclipse.org/modeling/emf/>
20. В Language Keywords and Operators. Version 1.8.5. // ClearSy, 2009.