ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ УЧРЕЖДЕНИЕ
«ФЕДЕРАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЦЕНТР
«ИНФОРМАТИКА И УПРАВЛЕНИЕ»
РОССИЙСКОЙ АКАДЕМИИ НАУК»
(ФИЦ ИУ РАН)

на правах рукописи

Ступников Сергей Александрович

МЕТОДЫ И СРЕДСТВА ВЕРИФИЦИРУЕМОЙ ИНТЕГРАЦИИ НЕОДНОРОДНЫХ ДАННЫХ

2.3.5 - Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
доктора технических наук

Научный консультант
д-р техн. наук, доцент
В. Н. Захаров

Москва
2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность темы исследования.

Роль данных в различных областях деятельности человека – научных исследованиях, здравоохранении, образовании, промышленности и т.д. – непрерывно растет в последние годы. Укрепляется новая парадигма в науке и информационных технологиях, связанная с интенсивным использованием данных – так называемая *четвертая парадигма* [1]. *Эмпирическая парадигма,* господствовавшая в науке в течение тысяч лет, состояла в описании природных явлений. Сотни лет назад ее расширила *теоретическая парадигма*, предполагающая использование моделей и обобщений. Десятки лет назад возникла *вычислительная парадигма*, дополнившая способы научного познания симуляцией сложных явлений. Современная научная парадигма объединяет теорию, эксперимент и симуляцию. Данные поступают от инструментов или симуляторов, обрабатываются (очищаются) программным обеспечением, а затем ученые производят анализ извлеченной информации. Собственно научное знание образуется в процессе интенсивного анализа накапливаемых данных, приводящего в конечном счете к извлечению знаний из данных. Данные используются для решения задач в различных предметных областях (называемых *областями с интенсивным использованием данных*) – астрономии, материаловедении, управлении землепользованием, биологии, медицине, физике и других [2].

Основная проблема новой парадигмы состоит в том, что объемы поступающих данных значительно превышают возможности современных методов и средств хранения, обработки и анализа данных. Наблюдается экспоненциальный рост объема накапливаемых экспериментальных (наблюдательных) данных, оформляемых в виде источников данных [3]. В соответствии с принципами Открытой науки [4] источники данных концентрируются в *исследовательских инфраструктурах данных*, предоставляющих доступ к данным, вычислительным сервисам и процессам. Такие инфраструктуры предназначены для поддержки полного цикла управления и обработки данных, от сбора и курирования данных[1] до хранения и анализа. Примеры наиболее крупных исследовательских инфраструктур – это GEANT[2] (объединяющая

[1] *Курированием данных* (data curation) называется процесс создания и поддержки наборов данных таким образом, чтобы пользователи имели возможность находить их и использовать при решении своих задач.

[2] https://www.geant.org/

национальные Европейские исследовательские и образовательные организации в высокоскоростную широкополосную сеть, EGI[3] (вычислительные облачные и грид-сервисы), PRACE[4] (высокопроизводительные вычисления), IDGF[5] (федерация персональных компьютеров для массовых вычислений), OpenAIRE[6] (хранилище публикаций и данных исследований), EUDAT[7] (совместная инфраструктура для синхронизации, обмена, хранения, поиска, репликации, предоставления доступа к данным и вычислений над ними), ELIXIR[8] (распределенная инфраструктура для обнаружения, хранения, анализа данных в науках о жизни), EOSC[9] (Европейское облако данных открытой науки). Важность развития подобных инфраструктур в России подчеркивается, в частности, дорожной картой по развитию высокопроизводительных вычислений, алгоритмов искусственного интеллекта, грид-технологий и суперкомпьютерной инфраструктуры[10].

Однако, существующие инфраструктуры данных достаточно редко предоставляют комплекс средств, обеспечивающий извлечение максимальных выгоды из инвестиций в сбор данных и исследования. Для того, чтобы подвигнуть провайдеров данных и инфраструктур преодолеть этот разрыв, академическим сообществом Force11 были разработаны и опубликованы принципы управления данными FAIR Data Principles [5] (FAIR – Findability, Accessibility, Interoperability, Reusability). В соответствии с принципами FAIR, данные должны быть *обнаруживаемыми* (данные аннотированы достаточным количеством метаданных с уникальными и постоянными идентификаторами), доступными (метаданные и данные должны понятны и читаемы как людьми, так и компьютерами, и размещены в хранилище с разделяемым доступом), обеспечивать *совместное использование* (данные или инструменты из разных

---

[3] https://www.egi.eu/

[4] http://www.prace-ri.eu/

[5] http://desktopgridfederation.org/

[6] https://www.openaire.eu/

[7] https://eudat.eu/

[8] https://www.elixir-europe.org/

[9] https://ec.europa.eu/research/openscience/index.cfm?pg=open-science-cloud

[10] Распоряжение Правительства Российской Федерации от 12 марта 2026 г. № 482-р.

источников могут совместно работать и интегрироваться друг с другом; для описания метаданных должен использоваться общий формальный язык представления знаний) и *повторное использование* (данные и коллекции прозрачно лицензированы и содержат метаданные о своем происхождении). Повышение уровня реализации принципов FAIR в инфраструктурах данных облегчает и упрощает процесс обнаружения, оценки и повторного использования данных. Для этого развиваются новые информационные технологии, в которых данные становятся доминирующим фактором, новые подходы к концептуализации, организации и реализации информационных систем. При этом требуется не только создание методов и средств оперирования данными, объемы которых выходят за рамки возможностей современных технологий баз данных, но и разработка новых подходов, позволяющих справляться с разнообразием массово и хаотично развивающихся *моделей данных*[11], поскольку одна из важнейших проблем инфраструктур данных – это очень большая неоднородность данных.

Разнообразие моделей данных включает традиционную реляционную модель (диалекты языка SQL – Structured Query Language[12] [6]) и ее объектно-реляционные расширения, объектные модели (ObjectStore[13]) [7], тензорные [8] и графовые модели [9], семантические модели (как RDF – Resource Description Framework [10] и OWL – Web Ontology Language [11]), модели слабоструктурированных данных (XML – Extensible Markup Language [12], JSON – JavaScript Object Notation [13]), модели потоков работ и бизнес-процессов (XPDL – XML Process Definition Language [14], BPEL – Business Process Execution Language [15], BPMN – Business Process Model and Notation [16], событийные процессные цепи EPC – Event-driven Process Chain [17]), документные модели (системы MongoDB, Couchbase, CouchDB) [18], модели с семействами столбцов (системы Cassandra, HBase), модели «ключ-значение» (системы Redis, Memcached). Эти модели предоставляют существенно различные языки манипулирования и запросов для доступа к данным и их изменения. В обзоре шестидесятилетней истории моделирования данных [19] подчеркивается современный тренд на использование в информационных

---

[11] В данной работе под термином «модель данных» понимается комбинация языка определения данных (ЯОД) и языка манипулирования данными (ЯМД). Таким образом модель данных – это язык, предназначенный для описания схем баз данных и сервисов и запросов к ним.

[12] Следует отметить, что современный стандарт SQL [6] делает язык существенно мультимодельным: в стандарте включены возможности по представлению слабоструктурированных данных (XML), документных данных (JSON), многомерных массивов, графовых запросов. Однако, ядром стандарта остается реляционная модель.

[13] https://ignitetech.com/softwarelibrary/objectstore

системах, основанных на озерах данных, *гибридной* смеси технологий баз данных – традиционных (реляционных) и разнообразных нереляционных баз данных.

Наблюдаемый в настоящий период развития информационных технологий рост числа новых моделей данных сопровождается другой тенденцией - накоплением использующих подобные модели источников данных, число которых растет экспоненциально.

Такой рост вызывает все увеличивающуюся потребность совместного использования (*интеграции*) модельно неоднородных данных и сервисов в различных применениях, а также их повторного использования и композиции при решении новых задач в науке или промышленности. Различные по семантике и структурированности данные могут быть необходимы при решении одной задачи. Вопросы интеграции данных в инфраструктурах данных с течением времени становятся все более и более важными и сложными [20] [21] [22]. Интеграция данных делает их более обнаруживаемыми, доступными и пригодными для совместного и повторного использования, чем разнородные данные, хранимые в никак не взаимодействующих источниках. Фактически, интеграция данных – это необходимое предусловие для решения задач анализа данных, в частности, в рамках фундаментальных научных исследований с использованием высокопроизводительных вычислительных инфраструктур.

Может быть выделено несколько *уровней интеграции данных*. Эти уровни соответствуют уровням моделей в архитектуре метамоделирования MOF – MetaObject Facility (рисунок 0.1).

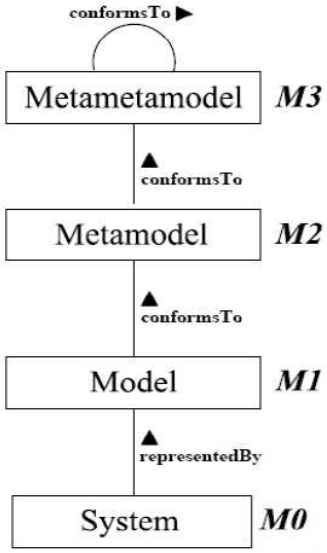

Рисунок 0.1 – Архитектура метамоделирования MOF

Модель определяется в соответствии с семантикой некоторой *метамодели*, при этом говорят, что модель *соответствует* или *конформна* (conforms to) метамодели. Стандартом OMG MOF [23] определена четырехуровневая архитектура моделей: модели уровня M0 описывают объекты реального мира, модели уровня M1 называются обычно *схемами*, модели уровня M2 представляют собой собственно модели данных, модели уровня M3 – мета-метамодели, предназначенные для описания моделей уровня M2. В современных исследованиях эта терминология может допускать различные вариации, например, в [24] модели M3 называются метаязыками, модели M2 – языками, модели M1 – описаниями моделей (model descriptions), модели M0 – исполнениями моделей (системами).

В соответствии с уровнями моделей, могут быть выделены следующие *уровни интеграции данных*, от более высокого к более низкому: интеграция моделей данных (унификация), сопоставление и интеграция схем данных (интеграция метаданных) и собственно интеграция данных. На уровне моделей данных сопоставляются элементы языков определения и манипулирования данными. На уровне схем сопоставляются структуры (типы) данных, их атрибуты и методы. На уровне собственно данных определяются правила трансформации коллекций данных, их экземпляров и значений атрибутов. Примеры сопоставления элементов моделей данных, элементов схем и значений атрибутов приведены на рис. 0.2.

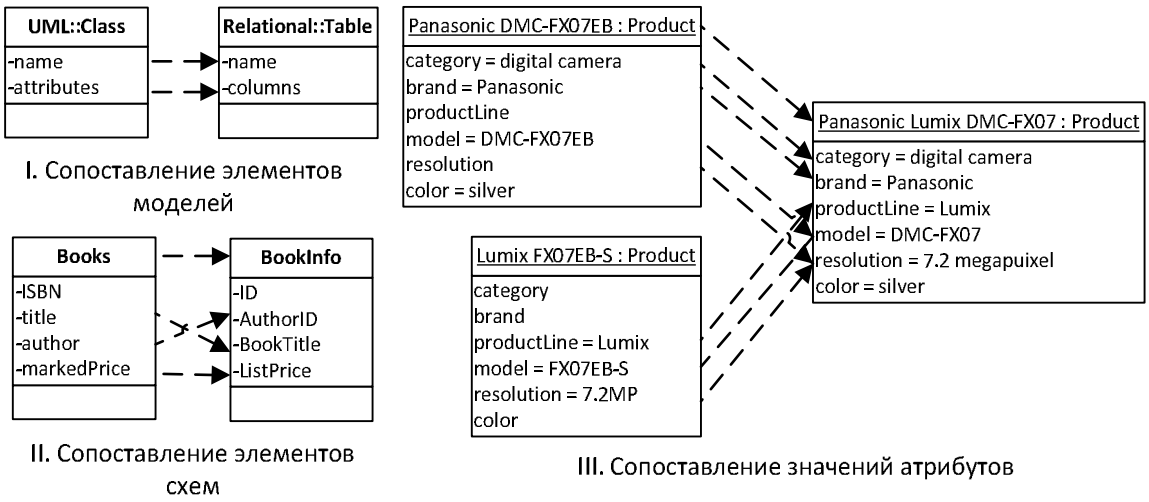

Рисунок 0.2 – Сопоставление элементов моделей данных, схем и собственно данных для интеграции данных

На уровне моделей данных класс UML (Unified Modeling Language) сопоставлен таблице реляционной модели, атрибуты класса сопоставлены столбцам таблицы. На уровне схем тип *Books* исходной схемы сопоставлен с таблицей *BookInfo* целевой схемы, сопоставлены также их атрибуты. На уровне собственно данных два

взаимодополняющих описания цифровой камеры сливаются в одно целевое описание. Обычно завершение интеграции на более высоком уровне является необходимым предусловием для интеграции на более низких уровнях.

Наиболее зрелый уровень интеграции обеспечивается в системах интеграции данных, таких как *предметные посредники* и *хранилища данных [22]*.

Предметные посредники [25] [26] – специальный вид программного обеспечения, образующий промежуточный слой между пользователем (приложением) и источниками данных и сервисов. При разработке посредников предпочтительным является *движимый приложениями* подход. Описание предметной области приложения в таком подходе образуется независимо от источников (в терминах понятий, структур данных, функций, процессов), а затем релевантные приложению источники отображаются в это описание). Преимуществами подхода являются масштабируемость по отношению к числу источников, возможность достижения семантической интеграции источников в контексте конкретного приложения и доказательной идентификации релевантных приложению источников, а также повышение стабильности спецификации посредника в процессе эволюции источников, релевантных приложению.

Создание предметного посредника состоит из двух фаз. На фазе *консолидации* происходит концептуальное моделирование предметной области посредника силами некоторого экспертного сообщества. Во время *операционной* фазы провайдеры источников данных регистрируют их в посреднике, т.е. представляют спецификации источников в терминах концептуальной модели посредника.

Предметные посредники реализуют *виртуальную интеграцию данных.* Запросы (программы) пользователя определяются в некоторой унифицирующей модели данных. Эти запросы декомпозируются во множества подзапросов, которые передаются на разнородные источники и исполняются. Источники соединяются с посредником при помощи *адаптеров*, которые преобразуют запросы в языки исходных моделей данных и преобразуют результаты исполнения запросов из исходной модели данных в целевую. Результаты исполнения запросов передаются через адаптеры в посредник, соединяются и передаются пользователю. Одна из современных тенденций применения технологии предметных посредников – их конструирование над озерами данных [27] [28] [29].

Хранилища данных, реализующие *материализованную интеграцию данных*, в настоящее время являются одной из основных составляющих инфраструктур поддержки систем бизнес-аналитики. Данные извлекаются из различных коллекций, преобразуются к единой схеме хранилища, интегрируются и загружаются в хранилище. Над хранилищем выстраивается слой приложений, осуществляющих анализ данных

(например, при помощи методов математической статистики, машинного обучения и др.) и выдающих результат пользователю в необходимой форме.

Ввиду роста неоднородности источников данных, их количества и объемов данных, актуальными остаются вопросы разработки методов спецификации и реализации интеграции данных в масштабируемых инфраструктурах распределенного хранения и обработки данных, подобных Apache Hadoop [30].

Традиционные промышленные решения по интеграции данных и разработке хранилищ (например, IBM InfoSphere Information Server [31]) предлагают визуальные средства проектирования потоков работ интеграции данных, оперирующие, преимущественно, понятиями и операциями реляционной модели и порождающие программы трансформации и интеграции данных на языке SQL. Отдельно выделяются визуальные средства сопоставления схем исходных коллекций данных и схемы хранилища (целевой схемы), например, InfoSphere FastTrack, позволяющие автоматически порождать части потоков работ интеграции данных. Для обеспечения интеграции данных, представленных в различных моделях данных, такие средства предоставляют отдельные компоненты, позволяющие преобразовывать исходные данные к реляционной модели (например, компонент преобразования XML в IBM InfoSphere DataStage [32]).

Параллельно промышленным решениям, проводятся исследования методов формальной спецификации правил интеграции данных. Классическим примером работы в данном направлении является подход обмена данными (data exchange) [33] [34]. Подход позволяет описывать интеграцию данных реляционной модели с использованием логических правил, которым сообщается формальная семантика в логике предикатов первого порядка.

Для интеграции неоднородных источников данных и сервисов FAIR-инфраструктуры данных могут поддерживать как виртуальную, так и материализованную интеграцию данных, а также их комбинацию, когда хранилища данных рассматриваются как источники для предметных посредников (рис. 0.3).

В общем случае источники данных неоднородны, представлены в различных моделях данных. А потому, при их интеграции с использованием систем виртуальной или материализованной интеграции для однородного представления семантики требуется приведение различных моделей данных к унифицированному виду в рамках некоторой унифицирующей модели данных, которая называется *канонической*. Такое семантическое преобразование разномодельных спецификаций к унифицированному

виду в канонической модели нуждается в специальных методах и инструментальных средствах.

После решения вопросов интеграции на уровне моделей в системах виртуальной интеграции необходима интеграция на уровне схем и построение представлений (взглядов), связывающих схемы источников данных и схему посредника. При этом необходимо семантическое сопоставление элементов схем источников данных и элементов схемы посредника.

Рисунок 0.3 – FAIR-инфраструктура данных с комбинированной виртуальной и материализованной интеграцией данных

Идеальным описанием комбинированного процесса интеграции схем и собственно данных в хранилищах можно было бы считать совокупность декларативных правил (подобную программе на языке Datalog), как при обмене данными. Однако, на практике, процесс интеграции данных представляет собой набор операций трансформации данных, представленных в SQL или подобном ему языке, причем важным является

порядок исполнения операций. Обычно интеграция данных организуется в виде потоков работ.

Обычно разработчикам систем интеграции данных на всех уровнях приходится полагаться на интуитивные приемы сопоставления элементов моделей, без точного учета семантики используемых моделей и подтверждения правильности выполняемых преобразований. Такой учет возможен только при помощи определения формальной семантики моделей данных и *верификации процесса интеграции данных на всех уровнях*, т.е. формальной доказательной проверки на соответствие заданным требованиям.

Существует два основных метода верификации сложных систем: *проверка моделей* (model checking) и *доказательство теорем* (theorem proving). Верификация моделей основана на построении конечной модели системы и проверке выполнения заданных свойств в этой модели. В силу конечности модели, такая верификация всегда может быть выполнена автоматически. Основная проблема верификации моделей состоит в том, что для проверки свойств сложных систем далеко не всегда возможно построить конечную модель приемлемого размера, и даже просто конечную модель.

Доказательство теорем представляет собой метод, в котором система описывается на языке некоторой формальной логики, а требуемые свойства системы описываются как формулы логики. Доказательство теорем представляет собой процесс вывода заданных формул из аксиом логики с использованием правил вывода. Некоторые выводы могут быть получены автоматически, для получения более сложных выводов используются средства интерактивного доказательства.

В контексте методов верификации систем формализуется одно из наиболее важных понятий в области формальных методов – понятие уточнения [35]. В настоящей работе уточнение понимается следующим образом: система *B* уточняет систему *A*, если пользователь может использовать систему *B* вместо системы *A*, не замечая факта замены *A* на *B*. Уточнение формализовано в различных языках спецификаций, и изначально предназначалось для разработки программных компонентов и комплексов[14] «сверху-вниз» методом пошагового уточнения (stepwise refinement). При этом система описывается на высоком уровне абстракции, затем уточняется серией конкретизаций вплоть до кода на языке программирования. Каждая последующая конкретизация системы является более детальной, чем предыдущая. При этом каждый шаг уточнения

[14] Согласно определению ГОСТ 19.101-2024 [207], программный компонент — программа, рассматриваемая как единое целое, выполняющая законченную функцию; программный комплекс — программа из двух или более программных компонентов и/или программных комплексов, выполняющих взаимосвязанные функции.

формально доказывается, и полученная в результате система обладает всеми необходимыми свойствами.

Формальная верификация с успехом применялись в большом количестве проектов по разработке систем в различных областях: медицине, ядерной энергетике, телефонии, транспорте, космической технике, разработке микропроцессорной техники, верификации протоколов и пр. [36]. Применение формальных методов доказательства сопряжено с известными сложностями, однако при разработке современных сложных информационных систем стоимость исправления ошибки на этапе эксплуатации превышает стоимость исправления ошибки на этапе проектирования в 10-100 раз [37]. Поэтому издержки на привлечение специалиста для интерактивного доказательства корректности представляются оправданными, особенно при разработке систем, ошибки которых критичны для безопасности функционирования человека (safety-critical systems).

Все вышесказанное позволяет сделать вывод о необходимости и актуальности разработки методов и средств верифицируемой интеграции неоднородных данных.

Степень разработанности темы исследования.

В 1980-90х годах Леонидом Андреевичем Калиниченко было заложено направление сохраняющих семантику отображений моделей данных [38][39][40]. Был предложен метод аксиоматического расширения реляционной модели данных для выражения различных видов зависимостей и построения эквивалентных отображений с сетевой и иерархической моделями данных. Предложен принцип коммутативного отображения моделей данных, сводящийся к коммутативности диаграмм отображения схем и последовательностей операций языка манипулирования данными. Предполагается, что семантика операций моделей данных выражается при помощи одинакового набора базовых функций и функциональных форм, а эквивалентность операций доказывается при помощи применения ряда правил эквивалентного преобразования функций. Предложен подход к однородному определению семантики реляционной и сетевой моделей данных. Построено отображение подмножества сетевой модели данных в реляционную. На примере показано определение семантики операций моделей и коммутативность диаграмм их отображения. В работе [41] предложено усовершенствование принципа коммутативного отображения моделей данных с использованием языка AMN (Abstract Machine Notation) [35] для определения семантики моделей и уточнения моделей вместо эквивалентного отображения. Применение принципа проиллюстрировано на отображении типа связи модели ODMG’93 в метатип

ассоциаций языка СИНТЕЗ [42]. Семантика конструкций представлена в виде AMN-спецификаций.

В мире в области определения формальной семантики моделей данных существенные результаты были получены Lano, Butler, France, Kim, Carrington, Beckert, Schmitt, Facon, Meyer, Souquieres, Ledang, Conen, Fikes, Mossakowski, de Bruijn, Schneider, Franconi, Guagliardo, Benzaken. Конструирование трансформаций моделей исследовались Fabro, Valdurez, Falleri, Wimmer, Diskin, Bohannon, Bollati. Методы и средства преобразования схем данных развивались группой Atzeni. Методы и средства верификации трансформаций моделей исследовались Lano, di Ruscio, Stenzel, Buttner, Ledang, Cabot, Cheng, Larsen, Vassiliadis.

Данная работа нацелена на целостное решение *проблемы формальной верификации интеграции неоднородных данных на всех трех уровнях: моделей данных, схем данных и собственно данных* с обеспечением формальной методической и программной базы в отличие от известных работ, уделяющих внимание отдельным аспектам верификации интеграции данных на отдельных уровнях. В частности, работа развивает и расширяет основные составляющие заложенного Л. А. Калиниченко направления сохраняющих семантику отображений моделей данных: формализацию методов, формирование доказательной базы, программную реализацию методов, применение методов на широком спектре моделей данных. Работа объединяет результаты, полученные автором в области верификации интеграции данных. Методы, развиваемые в данной работе, призваны служить формальными основаниями для совместного использования (мета)данных, интеграции структурированных данных (данных, конформных некоторой схеме, определяющей структуры данных) и повторного использования в FAIR-инфраструктурах данных.

Цель диссертационной работы состоит в разработке методов и средств верифицируемой интеграции неоднородных данных, обеспечивающих формальное доказательство корректности интеграции.

Для реализации этой цели необходимо решить следующие основные **задачи**:

1. Разработать метод конструирования сохраняющих семантику отображений моделей данных, обеспечивающий формальное доказательство корректности интеграции моделей данных.

2. Разработать метод верификации интеграции схем данных в системах виртуальной интеграции данных, обеспечивающий формальное доказательство корректности интеграции схем данных.

3. Разработать метод верификации интеграции данных в системах материализованной интеграции данных, обеспечивающий формальное доказательство корректности интеграции данных.

4. Разработать схемы функциональной структуры сред решения задач над неоднородными источниками данных с поддержкой верифицируемой интеграции данных.

5. На основе полученных методов разработать программные средства их автоматизации, включая: средства поддержки верифицируемой интеграции моделей данных, верификации интеграции схем, верификации интеграции данных.

Объектом исследования является процесс интеграции неоднородных данных.

Предметом исследования выступают методы и программные средства верифицируемой интеграции неоднородных данных.

Методы исследования. При решении поставленных задач использовались методы математической логики первого порядка и теории множеств, методы формальной спецификации предметных областей, методы теории уточнения спецификаций, методы денотационной и аксиоматической семантики, методы отображения и трансформации моделей данных и схем данных.

Научная новизна диссертационной работы заключается в следующем:

1. Разработан новый метод конструирования сохраняющих семантику отображений моделей данных, и в отличие от известных, сочетающий формальное определение синтаксиса и семантики моделей данных, интеграцию синтаксисов исходной и целевой моделей, создание расширений целевой модели, автоматизированное конструирование трансформаций моделей, верификацию корректности отображения моделей. Метод обеспечивает доказательство корректности интеграции моделей данных с использованием формально определенного уточнения спецификаций.

2. Разработан новый метод верификации интеграции схем данных в системах виртуальной интеграции данных, основанный, в отличие от известных, на уточнении абстрактных типов данных и запросов пользователя к системе интеграции. Метод обеспечивает формальное доказательство корректности интеграции схем данных.

3. Разработан новый метод верификации интеграции данных в системах материализованной интеграции данных. В отличие от известных методов, формальная семантическая спецификация сопоставляется высокоуровневой программе интеграции данных, и необходимые свойства программы интеграции проверяются с использованием

инструментальных средств доказательства. Метод обеспечивает формальное доказательство корректности интеграции на уровне данных.

4. Разработаны новые программные средства автоматизации методов интеграции данных, включающие средства поддержки верифицируемой интеграции моделей данных, верификации интеграции схем, верификации интеграции данных.

5. Разработаны новые схемы функциональной структуры сред решения задач над неоднородными источниками данных с поддержкой верифицируемой интеграции данных как методологическая основа для разработки систем решения аналитических задач над множествами неоднородных источников данных и сервисов.

Теоретическая значимость работы. Разработанные методы позволяют определять семантику, отображения и трансформации моделей данных различных классов: объектных, реляционных, онтологических, графовых, тензорных, процессных; составляют собой формальные основания для автоматизированной верификации процессов интеграции неоднородных данных.

Практическая значимость работы. Разработанные методы и средства предназначены для применения при разработке программных комплексов интеграции неоднородных данных[15], необходимых как при решении задач в исследовательских инфраструктурах в различных предметных областях, так и в промышленных информационных системах. В рамках диссертационного исследования разработан комплекс программ для поддержки методов автоматизированной верификации интеграции данных[16].

Полученные результаты диссертационного исследования были реализованы или использованы в научно-исследовательской и практической деятельности следующих организаций (акты внедрения приведены в Приложении Д):

- ФГБНУ ФИЦ «Почвенный институт им. В.В Докучаева»;
- Институт астрономии РАН;

---

[15] При разработке программных комплексов разработанные методы и средства применяются совместно с существующими методами и средствами, реализующими управление процессами жизненного цикла программных комплексов (ГОСТ Р ИСО/МЭК 12207–2010 [380]), защиту информации (ГОСТ Р 59352–2021 [214]), управление рисками (ГОСТ Р 59352–2021 [381]).

[16] Список программ, составляющих комплекс, с реквизитами свидетельств об их регистрации, приведен в Приложении В.

– Институт металлургии и материаловедения им. А.А. Байкова Российской академии наук (ИМЕТ РАН);

– ООО «Паллада».

Материалы диссертационной работы используются в курсе «Виртуальная интеграция неоднородных данных и унификация моделей данных» совместной магистерской программы факультета Вычислительной математики и кибернетики МГУ им. М. В. Ломоносова и ФИЦ ИУ РАН «Большие данные: инфраструктуры и методы решения задач».

Результаты диссертационного исследования были реализованы или использованы в рамках научно-исследовательских работ Российского научного фонда, Российского фонда фундаментальных исследований, Минобрнауки РФ, Отделения информационных технологий и вычислительных систем (Отделения нанотехнологий и информационных технологий) РАН, Президиума РАН, выполненных в ФИЦ ИУ РАН, МГУ им. М. В. Ломоносова, Институте астрономии РАН, Национальном исследовательском ядерном университете «МИФИ», ФГБНУ ФИЦ «Почвенный институт им. В.В Докучаева» (полный список приведен в Приложении Г).

Достоверность и обоснованность полученных в диссертации результатов подтверждена применением формальных методов определения семантики и отображений моделей данных, а также средств автоматизированного доказательства, опирающихся на логику предикатов первого порядка и теорию множеств; использованием теоретических результатов и опытом практической реализации результатов исследования в ряде НИР, сопоставлением результатов исследования с данными зарубежного и отечественного опыта; обсуждением результатов диссертационного исследования на международных и всероссийских научных мероприятиях.

Основные положения, выносимые на защиту:

1. Метод конструирования сохраняющих семантику отображений моделей данных, сочетающий формальное определение синтаксиса и семантики моделей данных, интеграцию синтаксисов исходной и целевой моделей, создание расширений целевой модели, автоматизированное конструирование трансформаций моделей, и основанный на уточнении семантических спецификаций, обеспечивает формальное доказательство корректности интеграции данных на уровне моделей данных и применим для интеграции широкого спектра моделей данных.

2. Метод верификации интеграции схем данных, основанный на уточнении спецификаций типов и запросов, обеспечивает формальное доказательство корректности интеграции данных в системах виртуальной интеграции данных.

3. Метод верификации интеграции данных, основанный на доказательстве истинности интегрирующих соотношений, связывающих исходную и целевую базы данных, после исполнения программы интеграции данных, обеспечивает формальное доказательство корректности интеграции данных в системах материализованной интеграции данных.

4. С использованием схем функциональной структуры среды комбинированной виртуально-материализованной интеграции данных и основанной на правилах мультидиалектной среды как методологической основы, осуществима разработка программных комплексов решения аналитических задач над множествами неоднородных источников данных и сервисов.

Апробация результатов. Результаты диссертационного исследования прошли апробацию на научных мероприятиях в России и за рубежом:

– Международная конференция «Аналитика и управление данными в областях с интенсивным использованием данных» DAMDID/RCDL 2015-2025 гг.

– Second Workshop on Modelling to Program (M2P). Lappeenranta University of Technology, Lappeenranta, Finland, March 11-12, 2020.

– Конференция «Наука о данных в Европе» (Data Science Europe), Институт современной науки, Белград, 15-19 ноября 2020 г.

– Международная конференция «Иванниковские чтения», Великий Новгород, 13-14 сентября 2019 г.

– Международная конференция по базам данных и информационным системам ADBIS. Блед, Словения, 2019 г.; Охрид, Сербия, 2014 г.; Genova, Италия, 2013; Познань, Польша, 2012; Вена, Австрия, 2011; Рига, Латвия, 2009, Пори, Финляндия; 2008; Салоники, Греция, 2006; Будапешт, Венгрия, 2004; Братислава, Словакия, 2002.

– Научная сессия «Исследовательские и образовательные компетенции для цифровой экономики». Отделение нанотехнологий и информационных технологий Российской академии наук. ФИЦ ИУ РАН, Москва, 6 июня 2018 г.

– Конференция «Технологии баз данных». 29-30 ноября 2016, Издательство «Открытые системы», Москва.

– Конференция «Центры коллективного пользования и уникальные научные установки в организациях, подведомственных ФАНО России». 20-21 октября 2015 г., Москва.

– Международная конференция по Открытым и большим данным OBD 2015. Рим, Италия, 24-26 августа 2015.

– Российская конференция по электронным библиотекам RCDL 2006, 2010-2014.

– Семинар Московской секции ACM SIGMOD. Москва, 2009.

– V Conference of the Italian Chapter of Association for Information Systems itAIS. Paris, France, 2008.

– Scientific Information for Society – from Today to the Future: 21st CODATA Conference. Kiev, 2008.

– Проблемы и методы информатики. II Научная сессия ИПИ РАН. Москва, ИПИ РАН, 2005.

– Emerging Database Research in Eastern Europe: Pre-Conference Workshop of VLDB 2003. Berlin, Germany, 2003.

– Fifth International Baltic Conference on Databases and Information Systems, BalticDB&IS'2002. Tallinn, Estonia, 2002.

– Международная научно-практическая конференция по программированию УкрПРОГ: Проблемы программирования. Киев, 2002.

– XXIV Конференция молодых ученых. Москва, МГУ им. М.В. Ломоносова, 2002.

– Научные семинары ФИЦ ИУ РАН, ВМК МГУ им. М.В. Ломоносова.

Материалы диссертационной работы использовались при формировании курса «Виртуальная интеграция неоднородных данных и унификация моделей данных», читаемого автором на магистерской программе «Большие данные: инфраструктуры и методы решения задач» факультета Вычислительной математики и кибернетики МГУ им. М. В. Ломоносова с 2015 г. по настоящее время.

Публикации. По теме диссертации автором опубликовано 66 статей, 2 монографии [42] [43].

Из них 23 статьи опубликованы в журналах из списка ВАК (К1 и К2) [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66], 33 статьи - в изданиях, включенных в наукометрические базы данных Scopus, Web of Science (1- в Scopus Q1 [67], 8 - в Scopus Q2 [68] [69] [70] [71] [72] [73] [74] [75], 6 - в Scopus Q3 [76] [77] [78] [79] [80] [81], 3 - в Scopus Q4 [82] [83] [84], а также 15 – в других

изданиях, индексированных Scopus [26] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98]). В других трудах российских и международных конференций опубликованы еще 10 работ [99] [100] [101] [102] [103] [104] [105] [106] [107] [108].

Получено 12 свидетельств о регистрации программ для электронных вычислительных машин ЭВМ (Приложение В).

Личный вклад автора. Диссертационная работа представляет собой многолетнее исследование автора, объединенное тематикой и методами исследования. Все выносимые на защиту результаты получены лично автором.

В работах [50][78][85][97][102][103] автором разработан метод унификации моделей данных, на примерах проиллюстрированы его этапы, проведено обсуждение применения метода для интеграции данных в исследовательских инфраструктурах. В работе [82] автором изложены основные идеи реализации метода унификации моделей данных с использованием движимой моделями инженерии, выделены преимущества этой реализации по сравнению с реализацией на основе метакомпиляции и переписывания термов. В работе [26] автором изложен краткий обзор методов унификации и определения формальной семантики для сервисных и процессных моделей. В работах [104][64] автором разработан метод автоматизации построения трансформаций моделей.

В работах [42][47][73][51] автором определено и проиллюстрировано понятие редукта абстрактного типа данных (АТД) канонической модели данных, формализованы определения соединения и пересечения АТД, определены общие правила интерпретации формул объектного исчисления, определена формальная семантика конъюнктивных запросов. В работе [100] автором разработаны программные средства синтаксического и семантического анализа спецификаций канонической модели. В работе [48] автором разработан метод представления конструкций языков UML и OCL (Object Constraint Language) в канонической модели данных. В работах [87][54] автором были созданы предикативные спецификации инвариантов в канонической модели данных.

В работах [49][109] автором определена формальная денотационно-аксиоматическая семантика ядра канонической объектной модели данных. В работах [110][46][107] автором разработан метод отображения спецификаций, выраженных средствами ядра канонической модели, в язык AMN. В работах [99][43][68][65][111] автором определена формальная семантика ядра канонической модели процессов, определены общие принципы построения расширений канонической модели процессов, определено отображение расширенных сетей потоков работ в AMN, проиллюстрирован

на примере метод доказательства корректности уточняющего расширения канонической модели процессов.

В работах [52][108] автором проведен анализ семантик логических программ. В работе [105] автором рассмотрены принципы создания каркаса логических диалектов RIF (Rule Interchange Format), синтаксические конструкции и семантические структуры каркаса, правила интерпретации синтаксических конструкций в семантических структурах, пример синтаксиса и семантики диалекта как специализации каркаса. В работе [79] автором были созданы логические правила на языке RIF-FLD (RIF Framework for Logic Dialects), используемые для аннотирования элементов онтологии.

В работе [70] автором определены общие принципы конструирования разрешимых расширений канонической модели на основе классов логических зависимостей в данных с сохранением семантики языков запросов. В работах [86][106] автором разработано взаимное отображение канонической модели данных и языка OWL 2 QL (Query Language), разработан и применен метод верификации отображения моделей на основе эквивалентного представления в единой семантической структуре. В работах [88][89][93][58][60] автором проведена унификация тензорной и графовой моделей данных в канонической объектной модели. В работе [80] автором разработаны основные идеи взаимообратного отображения графовой и реляционной моделей данных, а также доказательство корректности отображения.

В работах [101][45] автором разработан метод автоматизации верификации уточнения при композиционном проектировании информационных систем и предметных посредников. В работе [69] автором разработан метод верификации корректности интеграции веб-сервисов с использованием представления их семантики в языке AMN. В работе [72] автором разработан метод основанной на запросах верификации корректности интеграции данных. В работе [67] автором разработаны основные положения метода повторного использования и композиционного проектирования потоков работ.

В работах [76][95][63] автором разработан и применен метод верификации интеграции собственно данных путем определения семантики высокоуровневых программ интеграции данных в языке AMN. В работах [77][96] автором разработан метод спецификации мультимодельных правил интеграции данных с использованием логического языка на правилах и их реализации с возможностью последующей верификации. В работе [83] автором определены основные принципы реализации правил интеграции схем данных в модели RDF при помощи высокоуровневого языка анализа данных для исполнения в распределенной вычислительной среде. В работах [98] [81]

автором определены общие принципы расширяемого подхода к материализованной интеграции больших данных.

В работах [92][61] автором разработаны основные принципы построения и реализации комбинированной виртуально-материализованной среды интеграции больших неоднородных коллекций данных. В работе [94] автором определены этапы процесса извлечения информации из разноструктурированных данных и ее приведения к целевой схеме, определены отдельные компоненты программной архитектуры средств извлечения информации, ее интеграции и анализа, и связи между ними, и правила трансформации данных. В работе [57] автором разработано преобразование коллекций данных графовых моделей и модели RDF в интегрированное представление. В работе [62] автором разработана архитектура системы извлечения и интеграции информации из данных по Арктической зоне и правила трансформации данных. В работе [44] автором проведено описание задачи анализа системного риска, определены этапы процесса решения задач в виртуально-материализованной среде интеграции, определена концептуальная схема предметной области задачи и схемы источников данных, определено описание задачи в виде декларативной программы, определены семантические отображения схем источников в концептуальную схему. В работе [66] автором разработаны основные логические структуры данных для решения задач в области управления землепользованием.

В работах [91][55] автором разработана программная архитектура мультидиалектной среды для решения задач над неоднородными источниками данных, основные принципы ее реализации, концептуальные спецификации примера применения. В работах [71][56] автором определены основные принципы представления потоков работ в мультидиалектной среде и программная архитектура расширения мультидиалектной среды, ориентированного на потоки работ. В работе [53] автором разработан пример спецификации задачи в мультидиалектной архитектуре предметных посредников. В работе [59] автором описан метод мультидиалектной спецификации потоков работ, онтология структуры потоков работ. В работе [90] автором определен состав деятельностей по семантическому контролю научных потоков работ.

Результаты диссертационного исследования **соответствуют паспорту специальности** 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей», а именно – пунктам «1. Модели, методы и алгоритмы проектирования, анализа, трансформации, верификации и тестирования программ и программных систем», «2. Языки программирования и системы

программирования, семантика программ», «3. Модели, методы, архитектуры, алгоритмы, языки и программные инструменты организации взаимодействия программ и программных систем», «8. Модели и методы создания программ и программных систем для параллельной и распределенной обработки данных, языки и инструментальные средства параллельного программирования».

Структура и объем работы. Диссертационная работа состоит из введения, пяти глав, заключения, списка литературы (344 наименования) и четырех приложений. Работа содержит 39 рисунков и 55 таблиц. Общий объём работы - 433 страницы (313 стр. – основной текст и 120 стр. – приложения).

1 Основные понятия интеграции неоднородных данных и методы интеграции, подлежащие верификации

1.1 Модели данных

Под *моделью данных* в работе понимается совокупность двух языков – языка определения данных (ЯОД) и языка манипулирования данными (ЯМД).

История моделей данных [112] [113] прослеживается с конца 1960-х годов, когда появилась *иерархическая модель данных* [114]. Эта модель ввела понятие *тип записи* (коллекцию именованных полей и ассоциированных с ними типов данных), а также *ключа* – подмножества полей, уникально идентифицирующих *экземпляр* типа записи. Типы записей организуются в древовидную структуру, в которой у каждого типа записи (кроме корневого) есть уникальный *родительский* тип записи.

В 1970х развивалась *сетевая модель данных* [115], в которой типы записей были организованы в сетевую структуру (а не древовидную, в отличие от иерархической модели), фактически, представляющую собой связанный граф. Ребро сети называется *набором*: оно означает, что для с каждым экземпляром типа записи – *владельца набора* связаны 0 или несколько экземпляров типа записи – *члена набора*.

В 1970 году Коддом была предложена *реляционная модель данных* [116]. Основные структуры данных модели – это отношения (таблицы), хранящие кортежи (состоящие из значений атрибутов). Отношения могут быть связаны друг с другом *внешними ключами*. ЯМД для реляционной модели алгебраически замкнут (результатом операций являются отношения) и оперирует множествами кортежей, в отличие от сетевой модели, в которой записи перебираются по одной.

В середине 1970-х Чен предложил концептуальную модель Сущность-связь (Entity-Relationship, ER) [117]. Абстрактная база данных в этой модели представляет собой коллекцию экземпляров *сущностей*. Сущности могут иметь *атрибуты* – элементы данных, характеризующие сущность. Также между сущностями могут быть установлены *связи* различных типов множественности (от один-к-одному до много-ко-многим).

На основе реляционной модели и модели сущность-связь в начале 1980-х была разработана концептуальная модель IDEF1X [118]. Основные элементы этой модели – классы сущностей, связи, атрибуты, первичные и внешние ключи.

С середины 1980-х начали активно развиваться *объектные модели и базы данных* [7], направленные на преодоление «несоответствия импеданса» между реляционными СУБД и объектно-ориентированными языками программирования. В объектных моделях данных к структурам данных добавилось *поведение* в виде методов. В

дальнейшем появились *объектно-реляционные* модели данных и СУБД [119], в которых реляционная модель расширилась определяемыми пользователем типами данных, операциями, функциями, методами доступа к данным.

В начале 2000-х возник большой интерес к *слабоструктурированным* данным, и стала популярной модель XML [120] (с ЯОД XMLSchema и ЯМД XQuery). Основная идея слабоструктурированных моделей состоит в том, что экземпляры данных (записи) представляются в самоописываемом формате и не обязательно соответствуют какой-либо определенной схеме. Записи в XMLSchema могут быть иерархическими, могут ссылаться на другие записи, могут иметь атрибуты типа множества, могут наследоваться от других записей.

В 2000-х и далее в 2010-х происходил значительный рост разнообразия новых моделей данных (или, под видом новых моделей, возвращение популярности некоторым хорошо известным моделям данных) ввиду бума «больших данных».

В 2000-х несколько Интернет-компаний разработали узкоспециализированные СУБД (Memcached [121], Redis[17], Dynamo [122]), основанные на простейшей модели данных – *ключ-значение*. Структуры данных в этой модели – это ассоциативные массивы, отображающие ключи в значения, реализующие единственное бинарное отношение *(key, value)*.

В *документной модели данных* база данных представляет собой коллекцию объектов-записей [18]. Каждый документ содержит иерархию пар атрибут-значение, где атрибут идентифицируется именем, а значение атрибута может быть скалярного типа, массивом значений или другим документом. Модель данных *семейств столбцов* (column-family или wide-column), используемая в СУБД BigTable [123], Cassandra[18], HBase[19], фактически, представляет собой урезанную документную модель, поддерживающую лишь один уровень вложенности. Структуры данных похожи на отношения реляционной модели, но записи могут иметь опциональные атрибуты, и значениями атрибутов могут быть массивы значений.

Онтологические модели данных предназначены для представления онтологий [124], моделирующих знания об индивидах, их атрибутах и связях между индивидами в

---

[17] https://redis.io/

[18] https://cassandra.apache.org/

[19] https://hbase.apache.org/

различных предметных областях. Знания формализуются в виде сущностей (классов), обладающих атрибутами (свойствами), и связей между сущностями. Онтологии обычно рассматриваются как более высокий уровень абстракции, чем схемы баз данных. Для определения семантики онтологий используются *дескриптивные логики* [125], в которых задачи определения непротиворечивости онтологии или поглощения понятий разрешимы.

В *тензорных* моделях данных основной структурой является многомерный массив – коллекция элементов данных одного типа, где каждый элемент имеет координаты, располагающиеся в прямоугольной решетке с параллельными осями – подмножестве конечномерного Евклидова пространства [8]. ЯМД тензорной модели поддерживают декларативные операции манипулирования многомерными массивами. *Векторная* модель данных представляет собой урезанную тензорную модель, в которой массивы имеют единственное измерение. Векторные СУБД нацелены на хранение представлений (embeddings) для инструментов искусственного интеллекта, и реализуют поиск приближенного ближайшего соседа (approximate nearest neighbor, ANN) в векторном пространстве.

Графовые модели данных оперируют графами как основными структурами данных [9]. В *атрибутированных* графах как вершины, так и ребра могут быть аннотированы метками ключ-значение. Аналитические запросы к графам могут включать сложные операции, подобные нахождению кратчайшего пути или клики.

Процессные модели или модели *потоков работ* [126] предназначены для описания деятельности различных организаций при решении соответствующих им сложных задач [127]. Например, модели виртуальных организаций основаны на композиции процессов реальных организаций, вовлеченных в сферу деятельности виртуальной организации. При помощи потоков работ обычно формализуются задачи интеграции данных и сервисов.

Далее в данном разделе более подробно описаны те виды моделей данных, которым в работе уделено наибольшее внимание.

1.1.1 Семантические и онтологические модели данных

Одна из самых популярных семантических моделей данных – это RDF [10], позиционируемая консорциумом W3C как каркас для представления знаний в Веб [128]. Знания описываются в RDF при помощи утверждений, состоящих из трех элементов (называемых *RDF-триплетами*): *субъекта*, *предиката* и *объекта*. Множество триплетов формирует *RDF-граф*.

Субъектами и предикатами могут быть *идентификаторы ресурсов* (Internationalized Resource Identifier, IRI), объектами могут быть идентификаторы ресурсов и *литералы встроенных типов данных*. Идентификаторы и литералы обозначают сущности из Веб (включая вещи, документы, абстрактные понятия, числа и строки), называемые *ресурсами*. Предикаты в триплетах задают связи между ресурсами, называемые *свойствами*.

RDF Schema (RDFS) [129] представляет собой семантическое расширение RDF для моделирования данных. Основное понятие RDFS – *класс*, представляющий собой группу ресурсов. Ресурсы, составляющие класс, называются его *экземплярами*. На классах определяется рефлексивное и транзитивное отношение класс-подкласс (*subclassOf*): все экземпляры подкласса есть экземпляры суперкласса. Также в RDFS определяются метасвойства *domain* и *range*, позволяющие задавать утверждения о том, что элементы области определения или области значений некоторого свойства соответственно являются экземплярами одного или нескольких классов.

Язык OWL [11] – это дальнейшее расширение RDF и RDFS, предназначенное для описания онтологий в Семантическом вебе. Язык включает средства описания онтологий и составляющих их классов, свойств, индивидов, а также выражений для представления составных понятий предметной области и аксиом предметной области (утверждений, которые всегда должны быть истинными).

Язык SPARQL [130] направлен на представление запросов над RDF. В языке выражаются обязательные и опциональные графовые шаблоны, наряду с их конъюнкциями и дизъюнкциями. Поддерживаются агрегация, подзапросы, отрицание, ограничения. Результатами запросов могут быть множества кортежей значений или RDF-графы.

В последние годы распространение онтологий на область баз данных и информационных систем становится все более заметным. В начале 90-х годов в связи с разработкой стандартов взаимодействия компонентов информационных систем онтологии были обозначены как важнейшие составляющие систем, основанных на знаниях. Онтология была определена как «явная формальная спецификация разделяемой концептуализации» [131]. Позже онтологии стали рассматриваться как уровень абстракции моделей данных, предназначенных для моделирования знаний об индивидах [124]. Говорят, что онтологии находятся на *семантическом* (*концептуальном*) уровне, тогда как схемы баз данных - это модели данных на *логическом* или *физическом* уровне. Ввиду независимости онтологий от моделей данных более низкого уровня, появилось направление использования онтологий для интеграции неоднородных баз данных и

обеспечения интероперабельности разрозненных систем. Появились такие понятия, как основанные на онтологиях информационные системы (OBIS – Ontology-Based Integration Systems), основанный на онтологиях доступ к данным (OBDA – Ontology-Based Data Access), основанная на онтологиях интеграция данных (OBDI – Ontology-Based Data Integration) [132]. Основанные на онтологиях технологии предполагают использование баз данных на основе концептуализации прикладных областей и доступа к реляционным базам данных, опосредованного концептуальным представлением данных. Например, разработаны системы поддержки рассуждений над онтологиями, выраженными в дескриптивной логике семейства DL-Lite [133], и исполнения конъюнктивных запросов над онтологией на данных, хранимых во внешних реляционных СУБД. Перед выполнением запросы дедуктивно расширяются на основе аксиом концептуальной схемы и механизма логического вывода в соответствующей дескриптивной логике. К классу OBDI систем относится, например, Ontop [134].

Также онтологические модели применяются в системах материализованной интеграции данных, например PoolParty Semantic Suite[20], позиционирующейся как платформа семантического промежуточного (middleware) программного обеспечения. Входящий в нее комплекс UnifiedViews представляет собой каркас для создания ETL-процессов с RDF в качестве канонической модели данных. Извлечение данных обеспечивается из файлов форматов CSV, XML, JSON, текстовых документов, реляционных СУБД.

Исследования в области основанных на онтологиях информационных систем включают разработку онтологических языков для концептуального моделирования (концептуальных моделей данных), основанных на дескриптивных логиках. Наиболее наглядные результаты таких работ отражены в профиле OWL 2 QL [135], основанном на дескриптивной логике DL-Lite$_R$.

Онтологические модели, методы и средства существенным образом используются при создании предметных посредников. Так, концептуальная модель посредника включает онтологию, описывающую понятия и отношения, характерные для предметной области. При регистрации источников данных важным моментом является согласование (интеграция) онтологии посредника и онтологий источников [136].

В процессе концептуального проектирования и интеграции онтологий возникает ряд задач, для решения которых необходимы методы формального определения и

[20] https://www.poolparty.biz/product-overview

инструментальные средства верификации онтологий. К таким задачам относятся доказательство непротиворечивости онтологий и поглощения одного понятия другим, определение полной иерархии понятий и т.д. Автоматическое решение подобных задач возможно в рамках дескриптивных логик [137] при помощи специализированных инструментальных средств – систем вывода (например, Jena [138]).

Необходимым шагом, обеспечивающим применение автоматических систем вывода в дескриптивных логиках для верификации концептуального моделирования и интеграции онтологий в предметных посредниках, является построение взаимного отображения канонической модели данных и онтологической модели, служащей входным языком для систем вывода.

1.1.2 Языки на правилах

Раздел основан на работах автора [108][52][105].

Использование и развитие языков на правилах для логического программирования, дедуктивных баз данных, представления знаний, создания интеллектуальных информационных систем продолжается уже более сорока лет. Технология языков на правилах созревала в этот период в результате теоретических исследований, практического и коммерческого применения таких языков. В частности, накопленный в этой области опыт существенно превосходит опыт в области дескриптивных логик, активно развиваемых для онтологического моделирования.

Для эффективного использования потенциала подходов, основанных на правилах, сообщества применения языков на правилах в области искусственного интеллекта, в бизнесе, в Семантическом Вебе организовали проект RIF (Rule Interchange Format) [139][140] по выработке решений по унифицированному, интероперабельному использованию спецификаций, представленных на различных языках на правилах. Применение результатов этого проекта должно быть общеупотребительным и не ограничиваться Семантическим Вебом. Идея RIF состоит в создании расширяемого унифицированного языка на правилах, обеспечивающего возможность построения сохраняющих семантику отображений в него различных языков на правилах. Такой унифицированный язык представляется как семейство диалектов, которые имеют общее ядро (корневой диалект) и совокупность расширяющих его диалектов, образующих ориентированный граф без циклов. Каждое ребро графа (отношение расширения диалектов) отправляется от более простого к расширенному диалекту.

В RIF по отношению к диалекту каждая система программирования (система вывода) на правилах может выступать в двух независимых ролях – роли поставщика и роли потребителя. Первая роль означает, что система на правилах обеспечивает

преобразование собственных программ в программы на унифицированном диалекте. Вторая роль означает, что система на правилах может воспринимать программы на диалекте и преобразовывать их в программы на собственном языке системы. Таким образом, для полновесного включения каждого языка на правилах в совокупность интероперабельных языков достаточно снабдить соответствующую систему программирования двумя сохраняющими семантику преобразователями – из собственного языка в адекватный диалект (роль поставщика) и из диалекта в собственный язык (роль потребителя).

Для работы по проекту RIF в 2005 г. была образована рабочая группа W3C RIF WG с целью выработки «обменного формата» правил – так кратко именуется семейство определяемых RIF унифицированных диалектов языка на правилах. Изначально было декларировано, что целью RIF не является обеспечение единственного языка, объемлющего черты всех известных языков на правилах. Различные средства различных языков и систем на правилах зачастую не совместимы. Поэтому и была принята концепция диалектов RIF, обеспечивающая возможность обмена модулями правил между различными системами на правилах. По замыслу проекта, для правил, созданных в рамках некоторого приложения, должна быть обеспечена возможность их публикации, использования совместно с другими правилами, повторного использования в других приложениях и других системах на правилах.

Кроме того, важной задачей RIF WG было обеспечение возможности совместной работы диалектов RIF и существующих стандартов Семантического Веба – таких как RDF и OWL, не совместимых с большинством существующих языков на правилах. В частности, важно было показать, что существует возможность совместной работы дескриптивных логик и систем на правилах:

– сформировать общий базис для онтологических языков на правилах и на дескриптивной логике с целью достижения их интероперабельности;

– проанализировать пересечение этих двух формализмов представления знаний для понимания того, какой выразительности можно достичь при их объединении (комбинации);

– обеспечить использование машин на правилах как масштабируемых служб рассуждений в онтологиях, определяемых на таком пересечении.

В октябре 2009 г. был завершен важный этап RIF – публикация документов спецификации RIF в качестве кандидатов для стандартизации W3C. В 2013 г. было опубликовано второе издание документов RIF.

Список категорий примеров использования RIF включает следующие варианты:

– интеграция информации;

– принятие решений;

– кросс-платформенная разработка и развертывание правил;

– стратегии авторизации транзакций и управления доступом;

– обмен бизнес-правилами, ориентированными на взаимодействие с пользователем;

– публикация программ на правилах;

– сервисы обмена правилами для третьей стороны;

– развитое представление знаний.

Работа RIF WG была сосредоточена на двух видах диалектов – диалектах, основанных на логике, и диалектах на продукционных правилах. Логические диалекты используют логику первого порядка (часто ограниченную Хорновскими правилами) или другой вид логики. Диалекты на правилах с действиями включают языки продукционных правил и системы с реактивными правилами (или правилами «событие – условие – действие»). Были определены только два логических диалекта – базовый логический диалект (RIF-BLD [141]) и его подмножество - диалект-ядро RIF, расширяемое также диалектом продукционных правил (RIF-PRD [142]).

Диалект RIF-BLD соответствует логике Хорна с различными синтаксическими и семантическими расширениями. Основные синтаксические расширения включают синтаксис фреймов и предикаты с поименованными аргументами. Основные семантические расширения включают типы данных и внешне определенные предикаты. Хотя этот диалект не является достаточно выразительным для многих применений правил, он охватывает большое число существующих систем на правилах, что позволяет считать его необходимой предпосылкой создания более выразительных диалектов в будущем.

RIF-PRD – это другой крупный диалект, охватывающий основные аспекты различных систем на продукционных правилах. Серьезный интерес к технологиям на продукционных правилах был проявлен крупными компаниями. Продукционные правила обычно определяются с использованием неформальных вычислительных схем не основанных на логике. Поэтому RIF-PRD не является частью набора логических диалектов RIF. Вместе с тем значительное внимание было уделено тому, чтобы выделить максимально возможную общую часть продукционного и логических диалектов. Выделение такой общей части послужило основанием для создания диалекта-ядра RIF.

Предполагается, что существующие и будущие диалекты RIF будут использовать один и тот же набор типов данных, встроенных функций и предикатов в соответствии с их определением в документе RIF Datatypes and Built-Ins (RIF-DTB [143]).

Для упрощения создания новых диалектов и сокращения времени их разработки RIF WG определила каркас расширения RIF, называемый каркасом логических диалектов (RIF-FLD) [144]. RIF-FLD не является собственно диалектом, он определен как универсальный каркас для создания новых логических диалектов, расширяющих существующие. Он был разработан для существенного упрощения процесса определения и верификации новых логических диалектов, расширяющих возможности RIF-BLD.

Разработка каркаса RIF-FLD оказалась возможной, поскольку, несмотря на различия логических теорий, составляющих основу различных логических систем на правилах, в них используется много общих синтаксических и семантических конструкций. Более того, способы образования комбинаций различных подобных конструкций для образования соответствующих систем хорошо изучены. Спецификация RIF-FLD является результатом тщательного разбора этих знаний и представляет их в согласованном виде.

RIF-FLD представляет собой весьма общий логический язык, использующий значительную часть широко используемых синтаксических и семантических конструкций: при этом, однако, намеренно ряд параметров оставлен неопределенным для того, чтобы конструкторы конкретных диалектов могли добавить необходимые детали. Например, RIF-FLD предоставляет способ представления правил синтаксиса при помощи понятия сигнатур. В нем также можно специфицировать некоторые семантические понятия, такие как модели и логическое следование, оставляя при этом открытым выбор конкретных вариантов (например, какие конкретно модели следует использовать при рассмотрении следования). Конструктор конкретного диалекта может определить его синтаксис и семантику заданием их специализации на основе синтаксиса и семантики RIF-FLD.

1.1.3 Тензорные модели данных

Раздел основан на работах автора [145] [58].

Один из важных видов моделей данных, нацеленных на параллельную обработку и анализ данных в распределенных средах – гридах и облаках – это модели данных, основанные на многомерных массивах (array-based data models, или array data models) [8] [146]. В последние годы эти модели данных и соответствующие СУБД стали называться *тензорными* [147].

Родственны данным моделям так называемые «кубы данных», используемые в OLAP-технологии [148] [149]. Исследования тензорных моделей начались достаточно давно [150] [151] и продолжают развиваться. Первой тензорной СУБД считается Rasdaman [152] (система широко распространена и популярна и в настоящее время).

Другой хорошо известный в мире пример тензорной модели – это модель Array Data Model (ADM), используемая в СУБД SciDB [153].

История SciDB начинается с 2007 года, когда на симпозиуме по экстремально большим базам данных (XLDB) представителями науки и промышленности был сделан вывод о том, что существующие СУБД не в состоянии манипулировать объемами данных, которые появятся в ближайшем будущем. Одним из примеров поставщиков таких данных служит строящийся телескоп LSST (Large Synoptic Survey Telescope) [154]. Был также сделан вывод о необходимости разработки СУБД нового поколения, которая должна удовлетворять, в частности, следующим требованиям [155]:

- модель данных основывается на многомерных массивах, а не на кортежах;
- модель хранения базируется на версионности, а не на обновлении значений;
- масштабируемость до сотен петабайт и высокая отказоустойчивость;
- СУБД является свободно распространяемым программным обеспечением.

Некоторое время спустя был запущен международный проект под руководством Майкла Стоунбрейкера, целью которого стало создание новой системы управления базами данных, получившей название SciDB. В настоящее время свободно распространяется очередная версия системы для ОС Red Hat Enterprise Linux7 и CentOS 7 [156].

Нужно отметить, что модель ADM подвергается некоторой критике со стороны исследователей, продолжающих развитие моделей, основанных на многомерных массивах. Так, авторы языка SciQL [157] отмечают, что язык ADM является смесью SQL и деревьев алгебраических операций. По их мнению, язык для СУБД, основанных на многомерных массивах, должен быть интегрирован с синтаксисом и семантикой SQL:2003. Несмотря на эти замечания, модель ADM представляет несомненный практический интерес для интеграции баз данных. SciDB используется как в научных проектах, связанных с LSST (предполагается после запуска телескопа) и физикой высоких энергий, так и в коммерческих, связанных с генетикой, страхованием, финансами. Сравнительное тестирование SciDB с СУБД Postgres и статистическим ПО R показало преимущества SciDB по производительности и масштабируемости [158].

Тензорные СУБД продолжают активно развиваться в настоящее время. Так, одна из современных тензорных СУБД, показывающая существенное повышение производительности по сравнению с SciDB – это ChronosDB [159].

1.1.4 Процессные модели данных

Раздел основан на работе автора [43].

Процессные модели необходимы для описания деятельности различных организаций при решении стоящих перед ними задач. Например, модели виртуальных организаций основаны на композиции процессов реальных организаций, вовлечённых в сферу деятельности виртуальной организации.

Процессы относятся к классу интерактивных вычислений [160][161]. Понятие интерактивных вычислений характеризуется недетерминизмом (выбор проявляемого поведения осуществляется под влиянием среды), одновременностью (рассматриваемой для композиции взаимодействующих одновременных процессов), взаимодействием между компонентами и внешней средой, а также ограничениями, накладываемыми средой на вычисления.

Процессные модели активно разрабатываются в мире в течение нескольких десятилетий для практических целей и целей исследований. Выделяют *интенсиональные модели* (LTS, сети Петри и ряд других моделей, имитирующих процессы состояниями и переходами между ними), а также *экстенсиональные модели* (CSP [162], CCS [163], ACP и другие процессные алгебры, имитирующие системы в терминах внешнего наблюдателя как трассы событий, разнообразные деревья синхронизации, событийные структуры). В то же время ни одна из классических существующих моделей не может служить расширяемым ядром канонической модели процессов в силу отсутствия общей теории одновременного поведения. Существующие процессные языки базируются на разнообразных понятиях и парадигмах, не совместимых напрямую между собой.

Наиболее широко практическое использование процессные модели находят в разнообразных Системах Управления Потоками Работ (СУПР), обеспечивающих описание процессов, которые необходимо поддерживать в информационных системах, и управление ими. Процессные модели СУПР обычно не формализованы и являются чрезвычайно разнообразными: их число измеряется многими десятками и определяется большим числом независимых компаний – разработчиков СУПР.

Группой ученых во главе с Ван дер Аальстом [164] была выполнена работа по анализу процессных моделей систем управления потоками работ (включая Staffware, COSA, Meteor, Mobile, SAP/R3, MQSeries, FLOWer, Pegasystems SmartBPM Suite, IBM Business Process Manager, Oracle SOA Suite и Oracle BPM Suite), а также языков

композиции Веб-сервисов (BPML, BPMN, XPDL). При этом был выполнен анализ типовых конструкций в рассмотренных процессных моделях. Этот анализ позволил выявить полный набор образцов процессных моделей потоков работ (workflow patterns). Образцы разбиваются на три группы, соответствующие трем основным перспективам модели бизнес-процесса: поток управления, данные, ресурсы. Каждая группа (включающая по несколько десятков образцов), в свою очередь, разбивается на категории. Например, группа потока управления включает восемь категорий, соответствующих различным аспектам моделирования управления потоками работ:

- образцы ветвления;
- образцы синхронизации;
- образцы повторения;
- образцы, включающие множественные экземпляры;
- образцы одновременности исполнения;
- образцы триггеров;
- образцы отмены;
- образцы завершения.

1.1.5 Графовые модели данных

Раздел основан на работах автора [60] [58].

Исследования графовых моделей начались в середине 1980-х годов. Математическими основаниями для них послужила теория графов, а наибольшее влияние оказали так называемые *семантические модели* (например, модель «сущность-связь»). Основными отличительными чертами графовых моделей данных являются следующие [165]:

- данные и/или схема данных представляются в виде графов или структур данных, обобщающих понятие графа (гиперграфы или гипервершины);
- манипулирование данными выражается в виде трансформаций графов или при помощи операций, основными параметрами которых являются такие характерные графовые структуры и свойства, как пути, подграфы, связность и т.д.;
- ограничения целостности тесно связаны с графами как структурой данных. Так, ограничениями могут быть уникальность меток ребер и вершин, типизация ребер и вершин, ограничения на область определения и область значений свойств ребер и вершин.

Графовые модели данных применяются в тех случаях, когда информация о взаимосвязях между данными или их топологии является более важной (или настолько

же важной), как сами данные. Поводом к использованию графовых моделей может быть также недостаточная выразительная сила языков запросов традиционных моделей. Наиболее распространенными примерами применения графовых моделей являются системы управления и анализа сложных сетей – социальных, биологических, информационных, транспортных, телекоммуникационных и других [166].

Социальные приложения графовых баз данных (БД) позволяют оперировать информацией о связях между людьми, поддерживать совместную работу и потоки информации, предсказывать поведение. Социальные сети помогают выделять прямые и непрямые отношения между людьми и группами, позволяя пользователям оценивать, рецензировать, находить интересующие вещи и друг друга. Понимание того, как люди связаны и как они взаимодействуют друг с другом, как представитель группы действует или осуществляет свой выбор на основании агрегированного поведения группы, позволяет выявить скрытые силы, влияющие на поведение индивидуумов.

Рекомендательные алгоритмы устанавливают связи между людьми и вещами (продуктами, сервисами, медиа-контентом) – всем тем, что релевантно предметной области, в которой используется рекомендательная система. Связи устанавливаются на основании поведения пользователей (покупки, оценки и т.д.). Как и в случае социальных сетей, эффективная рекомендация зависит от понимания связей между вещами, а также качества и силы связей. Такие структуры наилучшим образом представляются в виде атрибутированных графов. Запросы в графах являются преимущественно локальными, т.к. их отправной точкой являются один или несколько идентифицируемых объектов, а дальнейший поиск производится в окрестности этих объектов.

Геопространственные приложения графовых БД варьируются от вычисления маршрутов в транспортных или логистических сетях до пространственных операций, таких, как поиск всех интересующих точек в ограниченной области, нахождение центра региона, вычисление пересечения регионов. Геопространственные операции зависят от используемых структур данных, начиная с простых взвешенных и направленных связей и до пространственных индексов, таких, как R-деревья. Такие индексы естественным образом представляются в виде графов. Геопространственные данные могут существовать в графовой БД совместно с другими видами данных, например, социальными. При этом становятся возможными сложные многомерные запросы над несколькими предметными областями одновременно. Геопространственные приложения особенно актуальны в областях телекоммуникаций, логистики, путешествий, планирования маршрутов.

Основными данными (master data) называются данные, являющиеся критическими для бизнес-операций. Основные данные включают данные о пользователях, покупателях, продуктах, поставщиках, отделах, сайтах и т.д. В больших организациях такие данные сильно распределены и неоднородны по форматам, качеству и средствам доступа. *Управление основными данными* включает идентификацию, очистку, хранение и собственно, управление данными (governance). Управление основными данными должно адаптироваться к изменению организационной структуры, слиянию организаций, изменению бизнес-правил, включению новых источников данных и т.д. Графовые БД хорошо подходят для моделирования, хранения и запросов к основным метаданным и моделям основных данных.

Графовые БД предлагают решение для создания нового поколения систем *обнаружения мошенничества* (fraud detection). В таких системах применяются интеллектуальные рассуждения, в отличие от алгоритмов статистического анализа или распознавания образцов. Из сведений о транзакциях и пользователях обычно выводятся данные о связях между ними. Эти связи обходят с целью обнаружения потенциальных образцов мошенничества. Дополнительная информация о пользователях (принадлежность к организации; персональные, профессиональные и семейные связи; история адресов и т.д.) существенно способствует обнаружению образцов. Транзакция связывается с известным мошенником, либо образец транзакции связывается с мошенническими образцами. Графовые БД хорошо подходят для быстрого обхода связей (миллионы связей в секунду), обнаружения образцов и их визуализации.

Наибольшего разнообразия в своем развитии графовые модели достигли в 1990-х годах. Наряду с обычными графами, представляющими собой множества вершин (помеченных или непомеченных), соединенных ребрами (направленными или ненаправленными, помеченными или непомеченными), развитие в графовых моделях получили такие структуры, как *гипервершины* и *гиперграфы*. Гипервершина представляет собой вложенный граф, а ребра гиперграфа могут соединять не две, а произвольное количество вершин [167].

В настоящее время в мире наибольший интерес вызывают графовые СУБД, ориентированные на обработку и анализ больших графов. Масштаб «больших» графов можно представить на примере сети Facebook – в 2023 г. она насчитывает 2.94 миллиарда активных пользователей в месяц и около 300 миллиардов связей «дружбы» между профилями. Весь Веб можно также рассматривать как граф страниц (вершины), соединенных гиперссылками (ребрами). В этом случае речь также идет о графе с

миллиардами вершин и ребер. Ведущим интернет-компаниям, таким, как Google, приходится осуществлять вычисления именно на таком графе.

Разрабатываются и исследуются различные инфраструктуры масштабируемой распределенной обработки и анализа графов.

Графовая модель системы *Pregel* [168], разработанная компанией Google, обладает следующими особенностями:

– входными данными Pregel-программы является ориентированный граф, каждая вершина которого идентифицируется уникальным идентификатором строкового типа;

– с вершиной ассоциируется изменяемое значение типа, задаваемого пользователем;

– направленные ребра ассоциируются с исходной вершиной;

– с ребром ассоциируется изменяемое значение типа, задаваемого пользователем, и идентификатор целевой вершины.

Основой модели вычислений Pregel является модель BSP (Bulk Synchronous Parallel). Вычисление состоит из последовательности итераций (супершагов). Перед началом вычисления все вершины являются активными. На каждом шаге для каждой активной вершины вызывается определенная пользователем функция, вызовы для всех вершин могут быть параллельными. Функция определяет поведение в отдельной вершине *V* на шаге *S*. Функция может читать сообщения, посланные вершине *V* другими вершинами на шаге *S-1*, посылать сообщения другим вершинам, которые будут получены на шаге *S+1*, изменять состояние *V* и исходящих ребер, добавлять и удалять ребра и вершины. Функция может перевести вершину в неактивное состояние. Вершина вновь становится активной, если ей придут новые сообщения. Вычисление завершается, когда все вершины становятся неактивными. Результатом вычисления является преобразованный граф. Такая модель вычислений называется *вершинно-центрированной*, поскольку основными объектами манипулирования в ней являются вершины.

Система Pregel является проприетарной, однако, корпорацией Apache разработан *Giraph* [169] – реализация Pregel с открытым кодом. Giraph основан на известной инфраструктуре распределенных масштабируемых вычислений *Hadoop* [30], запросы пользователя транслируются в задачи программной модели MapReduce. Именно Giraph используется в настоящее время для анализа социального графа Facebook.

Еще один вариант вершинно-центрированной вычислительной модели предлагается в системе *Trinity* [170], разработанной Microsoft Research. Отличие от модели Pregel состоит в том, что на каждом супершаге вершина может получать или

принимать сообщения, а также изменять значения только у *ограниченного* множества вершин (обычно, своих соседей). Это позволяет значительно оптимизировать алгоритмы на графах, укладывающиеся в такую модель (например, PageRank или поиск кратчайших путей). Однако, по утверждению разработчиков, Trinity не ограничивается данной моделью. Благодаря тому, что для хранения графов используется так называемое *распределенное облако памяти,* в системе может быть реализована любая вычислительная модель, в том числе модель Pregel или модель асинхронных вычислений [171]. Графовая модель данных, используемая в системе Trinity [13], обладает следующими особенностями:

– в модели изначально не выделяются вершины и ребра, эти понятия обобщены и объединены в понятие *ячейка* (cell);

– ячейка определяется уникальным идентификатором;

– с ячейкой могут быть связаны различные множества других ячеек, например:

1) для моделирования ненаправленного графа – множество соседних ячеек (соединенных ребром с данной ячейкой);

2) для моделирования направленного графа – множество ячеек, соединенных с данной ячейкой исходящим ребром, и множество ячеек, соединенных с данной ячейкой входящим ребром;

– с ячейкой может быть связана дополнительная информация (имя, описание и т.д.).

Если для решения задачи ребра графа необходимо аннотировать информацией, то ребра также могут быть представлены в виде ячеек. В этом случае каждая ячейка-вершина связывается с множеством инцидентных ячеек-ребер, а каждая ячейка-ребро – с множеством инцидентных ячеек-вершин. Очевидным образом модель позволяет представлять гиперграфы – с ячейкой-ребром можно связывать не две вершины (которые соединяет ребро), а произвольное количество вершин.

По утверждению разработчиков, система Trinity, в отличие от Pregel, позволяет решать задачи из обоих основных классов задач над графами [172]:

– запросы в режиме реального времени, основывающиеся на обходе графа;

– автономная (offline) аналитика на графах.

Эффективная аналитика обеспечивается использованием ограниченной вершинно-центрированной модели вычислений. Эффективный обход графа в режиме реального времени возможен за счет хранения графа в памяти распределенной системы (распределенное облако памяти) и распараллеливания запросов. В настоящее время ведутся работы по поиску и реализации эффективных алгоритмов решения задач

реального времени на больших графах. Например, новый метод поиска подграфов, изоморфных данному, с использованием инфраструктуры Trinity, предложен в работе [173].

Однако, несмотря на важность работ в области вершинно-центрированных моделей вычислений и обобщенных графовых моделей данных, обзоры состояния современных графовых СУБД показывают, что большинство существующих баз данных основаны на простых или атрибутированных графах (*attributed graph* или *property graph*), в которых атрибуты (свойства) приписываются ребрам и/или вершинам графа. Именно такие модели и были выбраны в данной статье в качестве исходных, подлежащих унификации моделей. К числу систем, основанных на простых или атрибутированных графах, относятся Neo4j [174], Sparksee [175], InfiniteGraph, OrientDB, VertexDB, Filament, OQGraph, Horton [176], InfoGrid и другие.

Такие системы также ориентируются в настоящее время на распределенную обработку больших графов. В качестве основы системы используют различные распределенные инфраструктуры.

Например, система *Horton* [176], разрабатываемая в Microsoft Research, использует распределенную инфраструктуру облачных вычислений *Orleans* [177].

В системе *Titan* [178] в качестве графового хранилища могут использоваться различные базы данных: Apache Cassandra, Apache HBase, Oracle BerkleyDB, которые, в свою очередь, предназначены для работы на вычислительных кластерах (БД Cassandra и HBase в качестве инфраструктуры хранения и вычислений используют Hadoop).

Широко распространенная графовая БД Neo4j [174], являющаяся наиболее эффективной среди аналогов [179], использует собственную кластерную архитектуру вида «главный-подчиненный» (master-slave). Ее отличительной особенностью является репликация графа на каждый узел кластера. В настоящее время система поддерживает графы размером в сотни миллиардов вершин и ребер[21]. Это сравнимо с потребностями самых крупных компаний, таких, как Google или Facebook.

Отметим, что системы, использующие сегментирование (sharding) графов в кластере вместо репликации на каждом узле, критикуются за то, что в них практически невозможно поддерживать ссылочную целостность между данными на разных узлах. Запрос к распределенному графу фактически превращается в набор локальных запросов

[21] https://neo4j.com/blog/achieve-unrivaled-speed-and-scalability-neo4j/

к отдельным узлам. К тому же, задача оптимального разбиения графа по узлам кластера является NP-полной, что делает ее практически нерешаемой на больших графах.

Системы, использующие сегментирование, могут быть эффективно использованы для решения задач, данные в которых являются сравнительно «плоскими», когда известно, что характерные запросы к графам будут сопровождаться прохождением по путям небольшой длины. Neo4j напротив, позволяет оперировать с «глубокими» структурами данных и очень быстро осуществлять глубокий поиск в графе. Тем самым обеспечивается практически неограниченная гибкость в моделировании графов предметной области.

1.2 Каноническая модель данных и ее роль в интеграции данных

Основные идеи *унификации моделей данных* в модельно неоднородной инфраструктуре данных заключаются в следующем. Для инфраструктуры данных выбирается *ядро* унифицирующей (канонической) модели данных. Каноническая модель данных в инфраструктуре служит языком представления знаний, упоминаемым в принципе FAIR I1 (*(мета) данные используют формальный, доступный, разделяемый и широко применимый язык для представления знаний*). Каждая *исходная* модель данных, используемая для представления некоторого набора источников данных в инфраструктуре, должна быть отображена в каноническую модель. При необходимости отображение может сопровождаться *расширением* ядра канонической модели, для того, чтобы покрыть все особенности исходных моделей данных. Примерами расширений могут служить структуры (типы) данных, ограничения (зависимости), сложные операции над данными. Отображение должно быть формализовано и верифицировано: должно быть предоставлено формальное доказательство того, что отображение сохраняет семантику структур данных и операций манипулирования данными исходной модели данных. Каноническая модель для инфраструктуры данных конструируется как объединение ядра и всех расширений.

При интеграции неоднородных баз данных в инфраструктуре данных для установления семантических соотношений между схемами источников и федеративной схемой необходимо отображение ЯОД исходной модели в каноническую. ЯМД канонической модели, напротив, необходимо отображать в ЯМД исходной модели. Это связано с тем, что запросы к посреднику в канонической модели необходимо отображать в запросы к источникам.

Отметим отличие виртуальной и материализованной интеграции. При виртуальной интеграции отображение ЯМД обеспечивает возможность трансляции программ на языке посредника в запросы на языке источников.

В случае материализованной интеграции данные извлекаются из источника и представляются в хранилище в канонической модели. При этом программы на языке канонической модели исполняются непосредственно на данных. Отображение ЯМД нужно лишь для того, чтобы убедиться, что отображение моделей сохраняет информацию и семантику операций. Семантически правильное отображение служит базой для процесса «извлечения – преобразования – загрузки» (ETL), формирующего из данных источника данные хранилища: ETL-процесс может быть выражен только в терминах канонической модели.

В рамках данной работы в качестве *ядра канонической модели данных* в основном рассматривается язык СИНТЕЗ [42], разработанный основателем направления сохраняющих семантику отображений моделей данных Л. А. Калиниченко[22].

Язык СИНТЕЗ представляет собой комбинированную слабоструктурированную и объектную модель данных. Слабоструктурированные (самоописываемые) данные представлены *фреймами* ([42], раздел 4). Фреймы используются для описания любого объекта, включая объекты самого языка, такие как спецификации типов, классов, функций, утверждений. Фрейм можно рассматривать как множество атрибутов, называемых *слотами.* Каждый слот может иметь множество значений. Информация, характеризующая слот как целое, может быть представлена в *метаслоте*. Слоты разделяются знаком точка с запятой, сам фрейм заключается в фигурные скобки. Имя слота и множество его значений разделяются двоеточием, значения одного слота разделяются запятыми.

Единицей определения канонической модели является *модуль*. Каждый модуль задает обобщенное представление источников данных и сервисов, либо является модулем спецификации предметной области или концептуального проекта информационной системы.

Типизированные данные должны соответствовать *абстрактным типам данных* (АТД), описывающих состояние и поведение своих экземпляров в терминах *атрибутов* и методов типа. В определения типов могут быть включены *инварианты* типов (ограничения, выраженные в виде замкнутой логической формулы).

---

[22] Монография с редакцией языка СИНТЕЗ 2007 г. вышла в соавторстве с автором данной работы.

По умолчанию АТД считается *объектным* – экземпляры такого типа (объекты) имеют встроенный атрибут *self*, значением которого является уникальный идентификатор, остающийся неизменным в течении всего времени жизни объекта. При модификации объекты сохраняют идентифицируемость. АТД также может быть объявлен необъектным: в этом случае экземпляры типа представляют собой неизменчивые значения. Модификация значения необъектного типа приводит к образованию нового значения, старое как бы перестает существовать.

Отношение подтипа над множеством абстрактных типов данных формирует решётку с *Taval* (общий супертип) в корне и типом *Tnone* (общий подтип) внизу решётки. Значение подтипа может использоваться всюду, где ожидается значение супертипа; подтип наследует элементы спецификации супертипа; допускается множественное наследование спецификаций подтипом.

Для решения задач композиционного проектирования информационных систем [180] было разработано исчисление спецификаций типов. На основе частично упорядоченного отношением тип-подтип множества спецификаций типов определена решетка типов. Операциями решетки типов являются соединение (*join*) и пересечение (*meet*) типов. Неформально, соединение $T_1 \sqcup T_2$ спецификаций типов T_1 и T_2 включает всю информацию, содержащуюся в спецификациях T_1 и T_2; пересечение $T_1 \sqcap T_2$ включает лишь общую информацию из спецификаций T_1 и T_2.

Помимо АТД, язык содержит также обширную коллекцию встроенных типов данных.

Методы и инварианты АТД описываются при помощи встроенного типа *функции.* Спецификация типа функции включает описание параметров функции и предикативную спецификацию функции. Для задания предикативных спецификаций функций в канонической модели используется язык логических формул многосортного объектного исчисления – типизированный вариант логики предикатов первого порядка. Предикативная спецификация позволяет задавать смешанные пред и постусловия, определяющие действия, реализуемые функцией. Для обращения к источникам данных 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 представляются в канонической модели *классами*. Явно разграничивая объявление типа и объявление коллекции (класса), язык СИНТЕЗ подчеркивает роль коллекции как представителя

множества объектов и роль типа как спецификации структуры и поведения объектов, связанных с коллекцией. Определения классов также могут включать инварианты.

Коллекции могут связываться друг с другом отношением обобщения-специализации (класс-подкласс). Суперкласс включает в себя подкласс как множество; тип экземпляра суперкласса является супертипом типа экземпляра подкласса.

Для классов предусмотрена также дополнительная многоуровневая система классификации на основе отношения класс-экземпляр. Поскольку класс сам является объектом, то он может стать экземпляром более общего класса, называемого *метаклассом.* Метакласс, в свою очередь, может стать экземпляром метакласса третьего уровня и т.д.

Атрибуты объектов в языке рассматриваются, в свою очередь, как объекты метаклассов ассоциаций, устанавливающие соответствие между набором объектов в области определения ассоциации и набором объектов в области значений ассоциации.

Для описания длительных составных деятельностей используются классы *скриптов,* основанных на сетях Петри.

Для расширения языка СИНТЕЗ в качестве канонической модели используются метаклассы, метафреймы, параметризованные конструкции, включая утверждения и родовые типы данных.

Следует отметить, что в качестве ядра канонической модели могут рассматриваться и другие широко применяемые модели данных, такие, как SQL (стандарт ISO/ANSI SQL 2011 или более поздний) или RDF/RDF Schema со SPARQL в качестве языка запросов. Однако, даже выбор между SQL и RDF достаточно сложен. С одной стороны, SQL поддерживается промышленными стандартами, методами и технологиями, которые развивались десятилетиями. С другой стороны, язык RDF – это рекомендация консорциума W3C, поддерживаемая крупными вендорами масштабируемых хранилищ триплетов, он тесно связан с онтологическим каркасом OWL, позволяет гибко изменять схему данных, естественным образом обеспечивает логический вывод над данными, что очень важно для организации баз знаний. В некоторых системах в качестве канонической используется документная модель JSON [181].

В качестве ядра канонической модели данных потенциально можно также использовать классические концептуальные модели данных – IDEF1X [118], ER [117], EXPRESS (язык, определенный в стандарте ISO 10303 [182]). При этом при интеграции данных можно использовать существующие отображения и трансформации различных моделей данных в IDEF1X, ER, EXPRESS.

Метод унификации моделей, рассматриваемый в главе 2, может применяться с различными моделями данных в качестве ядра канонической модели.

1.3 Методы интеграции схем данных, подлежащие верификации

Интеграция схем данных играет существенную роль при конструировании систем виртуальной интеграции данных [26].

Когда унификация моделей данных в модельно неоднородной инфраструктуре проведена, и построены адаптеры моделей данных [183], реализующие трансформации исходных моделей данных в каноническую, все разномодельные схемы источников данных автоматически приводятся к канонической модели. Следующим этапом интеграции является преодоление неоднородности на уровне схем (интеграция схем). Этот этап называется *регистрацией источников* в системе виртуальной интеграции (предметном посреднике).

Для элементов схемы посредника (типов, классов, отношений, атрибутов, методов) производится поиск релевантных им элементов схем источников данных. Например, этот поиск может быть основан на аннотировании элементов схем понятиями общей онтологии. После этого производится конструирование представлений (views), связывающих элементы схем источников и схемы посредника [184]. Представления могут иметь вид Global-As-View (GAV), когда элемент схемы посредника выражается в виде представления над совокупностью схем источников; Local-As-View (LAV), когда элемент схемы источника выражается в виде представления над схемой посредника; или Global-And-Local-As-View (GLAV), когда композиция элементов схемы посредника выражается как композиция элементов схем источников. Задачей верификации интеграции схем при этом является формальное доказательство того, что виртуальная схема посредника корректным образом реализуется посредством схем источников. Методы верификации интеграции схем, сводящие корректность интеграции к уточнению формальных семантических спецификаций, рассматриваются в главе 3 работы.

1.4 Методы интеграции данных, подлежащие верификации

Раздел основан на работе автора [185].

В процессе интеграции *собственно данных* традиционно выделяются три этапа [186]:

- трансформация данных;
- разрешение сущностей (Entity Resolution) [187];

– слияние сущностей (Data Fusion) [188].

Под сущностями понимаются цифровые представления объектов реального мира. При извлечении сущностей из источников данные *трансформируются*: преобразуются их структура и форматы. Из разных источников могут быть извлечены данные об одних и тех же объектах реального мира. Под *разрешением сущностей* понимают группирование, сопоставление, кластеризация сущностей из разных источников. *Слияние сущностей* – это образование интегрированного представления о сущности реального мира из разных описаний, полученных из разных источников, с разрешением конфликтов для каждого из атрибутов.

Трансформация данных основывается на результатах предыдущего этапа интеграции – интеграции схем. Обычно трансформация данных включает нормализацию схем и нормализацию данных. К операциям *нормализации схем* относятся, например, операции преобразования атрибута (изменение имени и/или преобразование значения из одного типа в другой), слияние значений разных исходных атрибутов в значение целевого атрибута, слияние множественных значений. *Нормализация данных* может включать приведение строк к одному регистру, удаление разделителей, исправление опечаток, замена сокращений и аббревиатур на полные стандартные формы и т.д.

Группирование (blocking) как начальный этап *разрешения сущностей*, предполагает разбиение объектов из исходных коллекций на группы (блоки). Группирование происходит на основе сходства описаний сущностей (равенство или сходство значений некоторых атрибутов).

На этапе *сопоставления* (matching) для каждой пары описаний сущностей, находящихся в одном блоке, происходит оценка того, принадлежат ли описания одному и тому же объекту реального мира. Оценка выполняется при помощи вычисления некоторой *функции сопоставления*. В простейшем случае функция сопоставления может представлять собой линейную комбинацию атомарных мер сходства значений нескольких атрибутов. Более сложные функции сопоставления могут комбинировать несколько условий сопоставления. Функции сопоставления могут формироваться также при помощи различных методов машинного обучения.

На этапе *кластеризации* на основании графа сопоставлений описаний в блоках, формируются *кластеры сущностей*, содержащие описания, относящиеся к одной и той же сущности реального мира. Кластеризация нацелена на установление прямых сопоставлений из непрямых (с использованием транзитивности отношения сопоставления) с одновременным отбрасыванием некоторых сопоставлений в пользу других сопоставлений с высокими весами.

Даже когда источники данных предоставляют информацию об одном и том же атрибуте одной и той же сущности, значения атрибута в разных источниках могут не совпадать. Эти конфликты возникают из-за неправильного ввода, некорректных вычислений, устаревшей информации, противоречивой интерпретации семантики и т.д. Цель *слияния данных* – определить, какое значение отражает положение дел в реальном мире. Значение может быть атомарным (дата рождения), множеством (телефонные номера), списком (авторы статьи) [186].

В настоящее время разработано множество методов решения задачи слияния данных. Разрешение конфликтов и удаление ошибочных значений опирается на коллективный «разум» источников данных, определение надежных источников, определение копирования данных между источниками.

При разрешении конфликтов возможно использование различных высокоуровневых стратегий [189]: *игнорирование конфликтов* (сохранение всех вариантов значений или создание всех возможных комбинаций значений, когда разрешение конфликта, фактически, остается за пользователем), *избегание конфликтов* (когда значения берутся из предпочитаемого источника, либо отбираются лишь непротиворечивые данные), *разрешение конфликтов* (выбор наиболее часто встречающегося, случайного, среднего, наиболее актуального значения и т.д.).

Реализация стратегий возможна при помощи различных реляционных операций [189]. Использование различных вариантов *соединения* (join) отношений опирается на комбинацию кортежей из отношений при обращении в истину некоторых предикатов над атрибутами. Варианты операции *объединения* (union) предполагают добавление кортежей из исходных отношений в заранее определенное отношение унифицирующей схемы.

Современным *базовым методом определения истинного значения* элемента данных обычно является голосование: значение, представляемое наибольшим количеством источников, объявляется истинным. *Надежность источника* определяется степенью корректности предоставляемых им значений. Более надежный источник обладает большим весом при голосовании. При обнаружении *копирования значения* из другого источника, вес этого значения при голосовании может быть уменьшен. К подходам, сочетающим вычисление вероятности копирования, вычисление надежности источников и голосование, относится, например, алгоритм *АссиСору* и его модификации [186]. Все большее значение в области слияния данных приобретают методы машинного обучения [190].

1.5 Языки и инструментальные средства трансформации моделей

Среди языков и инструментов конструирования трансформаций моделей данных можно выделить две группы: *декларативные* и *декларативно-императивные*. Данный раздел ограничивается кратким описанием языков и инструментов, применяемых при программной реализации метода унификации моделей данных (раздел 2.9).

К первой декларативной группе относятся языки метакомпиляции SDF (Syntax Definition Formalism) и ASF (Algebraic Specification Formalism), поддерживаемые инструментарием Meta-Environment [191], основанным на переписывании термов.

Формальный синтаксис моделей данных определяется в виде модулей языка SDF, включающих раздел сортов (куда попадают все нетерминальные символы синтаксиса) и раздел контекстно-свободного синтаксиса (включающий правила). Разделителем в правилах SDF является символ ->, голова правила располагается справа от разделителя, тело — слева. Опциональные части правил помечаются знаком вопроса ?, повторяющиеся части — знаком *.

Трансформации моделей данных определяются как модули языка ASF, представляющие собой наборы функций. Каждая функция определяет трансформацию синтаксического элемента исходной модели в синтаксический элемент целевой модели. Разрешены рекурсивные вызовы функций трансформации. На основе ASF-трансформаций, автоматически генерируется (компилируется) программный код трансформации на языке C. Полученная трансформация применяется для преобразования спецификаций исходной модели в спецификации целевой модели.

Термы хранятся в специализированной древовидной структуре с опциональными аннотациями для каждой вершины. Аннотации используются для хранения информации, специфической для различных инструментов обработки термов, например, координат текста на диаграмме или цветов атрибутов. Термы хранятся с максимальным совместным использованием подтермов, что сокращает объем необходимой памяти и повышает эффективность проверки равенства термов. Пересылка термов между компонентами инструментария осуществляется в бинарном формате, обеспечивающем высокую плотность и поддерживающем совместные использование подтермов. Это позволяет обрабатывать достаточно большие термы (более миллиона вершин).

Инструментарий Meta-Environment включает следующие компоненты, соединенные программной шиной, основанной на процессной алгебре [191]:

– интерфейс пользователя, включающий браузер для просмотра графа импорта текущей спецификации;

– текстовый редактор для редактирования спецификаций;

– структурный редактор с поддержкой синтаксиса;

– синтаксический анализатор вида SGLR (обобщенный LR), параметризованный таблицей синтаксического разбора;

– генератор таблиц синтаксического разбора на основе SDF-спецификаций;

– репозиторий термов, хранящий представления всех термов;

– компилятор языка ASF в язык C;

– интерпретатор спецификаций.

Ко второй группе языков трансформации моделей, сочетающих как декларативные, так и императивные средства описания трансформаций, относится ATL (ATLAS Transformation Language) [192], разработанный научной группой ATLAS INRIA & LINA. Язык развивается в рамках движимой моделями инженерии (Model-driven Engineering) [193][194], поддерживаемой Object Management Group. Базовыми составляющими MDE являются модели. Они рассматриваются как первичные сущности, и наиболее важными операциями манипулирования моделями становятся трансформации моделей из одной в другую.

Язык ATL позволяет описывать способ порождения моделей (называемых *целевыми*) на основании моделей, называемых *исходными*. Система типов и операций над типами языка ATL очень близка (но не тождественна) системе типов языка OCL [195]. Для языка ATL на базе платформы Eclipse реализована интегрированная среда разработки, называемая ATL Development Tools (ADT). В качестве модели уровня M3 языком ATL поддерживается модель *Ecore* [196].

Технически трансформация представляет собой модуль языка ATL. Модуль состоит из заголовка (включающего имя модуля и имена переменных, соответствующих исходной и целевой моделям), опциональной секции импорта (позволяющей импортировать библиотеки ATL), множества вспомогательных методов (называемых helper, они могут рассматриваться как эквивалент методов языка Java и могут вызываться из различных мест трансформации) и множества правил, определяющих способ построения элементов целевой модели на основе элементов исходной модели. Правила 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.

Следует отметить, что ATL – не единственный язык трансформации моделей, развиваемый в рамках MDE. Например, семейство языков QVT (Query/View/Transformation) [197] стандартизировано OMG, поэтому его также можно применять для унификации моделей данных.

1.6 Методы и средства уточнения спецификаций

Раздел основан на работах автора [74] [50].

Одним из наиболее важных понятий в области формальных методов является *уточнение*. Понятие уточнения интенсивно используется в методах разработки информационных систем, основанных на формальном доказательстве (proof-based methods). Изначально эти методы предназначались для разработки систем «сверху-вниз» методом «пошагового уточнения». Главная идея пошагового уточнения заключается в следующем: разработка системы начинается со спецификации, обладающей высокой степенью абстракции, затем спецификация последовательно детализируется. При этом образуется последовательность все более конкретных спецификаций. Отношение между двумя последовательными спецификациями и называется отношением *уточнения*. Важнейшим свойством уточнения является сохранение уже доказанных свойств системы. Свойства при этом могут быть различного рода: например, завершаемость работы системы или свойства, связанные с безопасностью.

В процессе разработки формулируются теоремы специального вида (*proof obligations*), истинность которых гарантирует корректность перехода от более абстрактной модели к более конкретной. Теоремы доказываются при помощи инструментальных средств доказательства автоматическим и (или) интерактивным образом.

На самом абстрактном уровне необходимым является описание статических свойств системы в виде инварианта. На основе этого инварианта формулируются теоремы, истинность которых означает непротиворечивость системы (сохранение инварианта операциями системы). Каждый из последующих шагов уточнения связан с новым инвариантом, устанавливающим соответствие между состояниями более конкретной и более абстрактной систем. Новый инвариант может также накладывать дополнительные ограничения на состояние конкретной системы, содержащее, возможно, большее количество деталей. Инварианты, связанные с шагами уточнения (*склеивающие инварианты*) используются при формулировке теорем уточнения.

Существующие методы формализации уточнения различаются языками спецификации систем, технологиями доказательства уточнения и инструментальными средствами доказательства уточнения.

В подходе Hehner [198] описание системы начинается с выделения величин, характеризующих систему. Каждой из таких величин сопоставляется переменная. *Спецификацией* называется предикат, описывающий связь между пред и постсостоянием системы в терминах этих переменных (поведение системы). Спецификация S_2 уточняет спецификацию S_1 для любого предсостояния s и постсостояния s' спецификация S_2 логически эквивалентна или сильнее спецификации S_1: $\forall\, s, s'\, (S_2 \Rightarrow S_1)$.

В подходе Morgan [199] спецификация имеет вид $x: [S, D]$ где x - набор переменных, S - предусловие, D - постусловие. Данная спецификация означает следующее: в случае выполнения предусловия S изменить переменные из x так, чтобы выполнялось постусловие D. Семантика спецификации задается ее *слабейшим предусловием* $wp(x: [S, D], P)$, гарантирующее завершение программы, удовлетворяющей спецификации S, в состоянии, удовлетворяющем P. Уточнение задается в терминах слабейшего предусловия: спецификация S_2 уточняет спецификацию S_1 если для любого постусловия P слабейшее предусловие S_1 сильнее слабейшего предусловия S_2.

Исчисление уточнения Back [200], основано на теории решеток и логике высоких порядков. Программы и спецификации рассматриваются как частные случаи *контрактов* между независимыми агентами. Контракт определяет поведение агента. Язык описания контрактов включает операцию обновления (изменения состояния системы в соответствии с некоторой функцией), выбор между контрактами, последовательную композицию контрактов, конъюнкцию контрактов, утверждение $\{g\}$ (требование к агенту, чтобы состояние системы удовлетворяло условию g), предположение $[g]$ (агент обязан исполнить контракт только в случае выполнения условия g). Тот факт, что агент может выполнить контракт C из начального состояния s и при этом удовлетворить постусловию q, обозначается $s\{|S|\}q$. Контракт S *уточняется* контрактом S' если для любого состояния s и постусловия q выполнено $s\{|S|\}q \Rightarrow s\{|S'|\}q$.

Формальный язык Z [201] основан на теории множеств и предоставляет средства структуризации спецификаций - *схемы*. Схема содержит набор именованных переменных, а также предикат, связывающий пред и постсостояния переменных. Язык включает *исчисление схем*: схемы могут соединяться операциями конъюнкции, дизъюнкции, композиции. Уточнение в Z определяется следующим образом: схема с предикатом P_2 *уточняет* схему с предикатом P_1 если

$$\forall\ s,\ s'\ ((\text{pre}\ P_1 \Rightarrow \text{pre}\ P_2) \wedge (\text{pre}\ P_1 \wedge P_2 \Rightarrow P_1))$$

Здесь s есть совокупное предсостояние, s' - постсостояние, pre P означает предусловие, выделяемое из предиката P. Язык Z поддержан инструментальными средствами Community Z Tools[23], Z-Eves[24], Hol-Z [202].

Подход VDM (Vienna Development Method) [203] берет свое начало из исследований по разработке и анализу информационных систем, проводимых в середине 70х годов прошлого века в лаборатории IBM в Вене. Язык спецификаций VDM основан на трехзначной логике, что помогает оперировать с понятием неопределенности: данные в большинстве случаев моделируются частичными функциями, и неопределенность возникает всякий раз, когда функция применяется к значению, лежащему вне ее области определения. Состояние системы в VDM задается набором типизированных переменных. Операции, определяемые при помощи пред и постусловий, могут экспортировать эти переменные и тем самым получить к ним разделяемый доступ. *Уточнение операции* O_A абстрактной системы с пространством состояний A операцией O_R более конкретной системы с пространством состояний R формализуется в VDM в виде двух теорем: *правила области определения*

$$\forall\ r \in R\ (pre_{OA}(retr(r)) \Rightarrow pre_{OR}(r))$$

и *правила результата*

$$\forall\ pre_r,\ r \in R\ (pre_{OA}(retr(pre_r)) \wedge post_{OR}(pre_r,\ r) \Rightarrow post_{OA}(retr(pre_r),\ retr(r))$$

Здесь pre_{OA}, $post_{OA}$ – это предусловие и постусловия операции O_A, pre_{OR}, $post_{OR}$ – это предусловие и постусловия операции O_R; pre_r – предсостояние системы, используемое в постусловии $post_{OR}$; *retr* – функция извлечения, сопоставляющая элементы пространств состояний абстрактной и конкретной систем. Метод VDM поддержан инструментальными средствами анализа спецификаций и средствами доказательства теорем VDMTools[25].

В подходе RAISE, включающем язык спецификаций RSL [204] и метод проектирования и разработки информационных систем [205], сочетаются логический (модельно-ориентированный) и алгебраический стили спецификации систем. Алгебраический стиль предназначен для определения спецификаций высокого уровня

---

[23] https://czt.sourceforge.net/

[24] https://github.com/arandilopez/z-eves

[25] http://fmvdm.org/

абстракции, логический - для определения более конкретных спецификаций. Модельно-ориентированные черты языка RSL являются развитием соответствующих черт языков VDM и Z. Основной конструкцией языка является *класс*, содержащий определения типов, значений, переменных, каналов и аксиом. Типы могут быть определены абстрактно или с использованием конструкторов сортов, свойственных теории множеств. Функции могут быть определены при помощи сигнатур и аксиом; при помощи пред и постусловий; а также явным указанием выражения, являющегося значением функции. Понятие уточнения в языке RSL формализуется следующим образом: класс *B′ уточняет* класс *B*, если сигнатура класса *B′* включает сигнатуру класса *B* и все *свойства* класса *B* имеют место и для класса *B′*. Сигнатура класса включает имена определенных в нем типов, значений, переменных, каналов и объектов (с сигнатурами их классов). Свойствами класса являются все аксиомы, полученные из определений компонентов класса. Подход RAISE поддержан инструментальными средствами RSLTC, RSL*[26].

В данной работе в качестве языка формализации уточнения используется AMN (Abstract Machine Notation) [35], основанный на логике первого порядка и теории множеств Цермело-Френкеля. AMN выбран как язык, поддержанный индустриальной технологией и средствами доказательства Atelier B [206]. Накоплен более чем двадцатилетний мировой опыт применения языка AMN и средств его инструментальной поддержки при разработке промышленных программных систем [36]. Язык AMN позволяет моделировать программные системы в виде абстрактных машин - спецификаций пространства состояний и поведения (определенного операциями на состояниях) систем. Спецификация состояния абстрактной машины задается *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. Неформально, спецификация *B* уточняет спецификацию *A*, если пользователь может использовать *B* вместо *A*, не замечая факта замены *A* на *B*. Уточняющая спецификация может быть значительно более детальной, чем уточняемая спецификация. Уточнение формализуется в AMN путем формулировки ряда теорем специального вида, так

[26] https://raisetools.github.io/

называемых *proof obligations*. Вообще говоря, в терминах пред- и постусловий операций, уточнение спецификаций в AMN означает ослабление предусловий и усиление постусловий соответствующих операций в этих спецификациях.

Теоремы уточнения формулируются автоматически при помощи инструментальных средств поддержки доказательства уточнения на основании склеивающих инвариантов, соотносящих состояния уточняемой и уточняющей систем. Теоремы могут быть доказаны при помощи инструментальных средств поддержки автоматического и (или) интерактивного доказательства.

Структурно спецификация AMN включает разделы определения множеств (SETS), определения констант (ABSTRACT_CONSTANTS), свойств констант (PROPERTIES), определения переменных состояния (ABSTRACT_VARIABLES), определения ограничений на значения переменных (INVARIANT), инициализации переменных (INITIALISATION), определения операций (OPERATIONS).

Любая операция в AMN имеет следующий вид:

$$r_1, \ldots, r_n \leftarrow \mathrm{op}(p_1, \ldots, p_m) = S;$$

где op — имя операции, $r_1, \ldots, r_n$ — выходные параметры операции, $p_1, \ldots, p_m$ — входные параметры операции, S — подстановка, определяющая действие операции на пространстве состояний.

Язык обобщённых подстановок [35] позволяет описывать переходы между состояниями системы. Каждая обобщённая подстановка S определяет преобразователь предиката, связывающий некоторое постусловие R со своим *слабейшим* предусловием $[S]R$, что гарантирует сохранение R после выполнения операции. В таком случае говорят, что *S устанавливает R*. Слабейшим предусловием называется такое предусловие, что предикат «начального состояния», связанный с некоторым предикатом «заключительного состояния», разрешает максимально большое число состояний. В таблице 1.1 рассматриваются основные виды обобщённых подстановок и соответствующие им слабейшие предусловия. В таблице S, T, T_1, T_2 — это подстановки, x, y, t — переменные, E, F — выражения, G, G_1, G_2, P — предикаты, $P\{x \to E\}$ — предикат P, в котором все свободные вхождения переменной x заменены на выражение E.

Таблица 1.1 – Обобщенные подстановки и их семантика

Обобщённая подстановка S	$[S]P$
Присваивание переменной $x := E$	$P\{x \to E\}$
Пустая подстановка *skip*	P
Параллельная подстановка $x := E \parallel y := F$	$[x, y := E, F]P$
Охраняемая подстановка SELECT G_1 THEN T_1 WHEN G_2 THEN T_2 END	$(G_1 \Rightarrow [T_1]P) \wedge (G_2 \Rightarrow [T_2]P)$
Предусловие PRE G THEN T END	$G \wedge [T]P$
Обобщенная охраняемая подстановка ANY t WHERE G THEN T END	$\forall\, t\, (G \Rightarrow [T]P)$
Последовательная подстановка S ; T	$[S][T]P$
Условная подстановка IF G THEN S ELSE T END	$(G \Rightarrow [S]P) \wedge (\neg G \Rightarrow [T]P)$

Теоремы корректности для спецификации M общего вида выглядят следующим образом:

```
REFINEMENT M
ABSTRACT_VARIABLES
  v
INVARIANT
  Inv
INITIALISATION
  Init
OPERATIONS
y ← op(x) =
PRE P
THEN
  S
END
```

I. *Теорема непустоты состояния спецификации*

$\exists$ v. Inv

II. *Теорема установления инварианта после инициализации*

[Init]Inv

III. *Теорема сохранения инварианта операциями*

Inv $\wedge$ P $\Rightarrow$ [S]Inv

Теорема I утверждает, что спецификация непротиворечива: существует состояние спецификации, удовлетворяющее инварианту. Теорема II утверждает, что инициализация переменных корректна: она устанавливает инвариант. Теорема III утверждает, что операции корректны: они переводят систему из одного состояния, удовлетворяющего инварианту, в другое состояние, удовлетворяющее инварианту. Каждая из теорем при генерации разделяется на несколько более простых частей, которые могут быть доказаны независимо.

В AMN существует три вида спецификаций[27]: абстрактная машина (*abstract machine*), уточнение (*refinement*), реализация (*implementation*). Абстрактная машина может быть только уточняемой спецификацией, и при описании операций абстрактной машины не разрешается использовать последовательную и циклическую подстановки. Реализация может быть только уточняющей спецификацией, при описании операций реализации не разрешается использовать недетерминированные и параллельные конструкции и предусловия. Реализации также не разрешается иметь собственных переменных. Уточнение является более универсальной спецификацией, так как может использоваться как в качестве уточняемой, так и в качестве уточняющей конструкции, язык описания операций уточнения не имеет таких ограничений как в абстрактной машине, так и в реализации. Поэтому конструкция уточнения является наиболее предпочтительной для однородного представления в AMN схем и запросов различных моделей данных.

1.7 Связь используемой терминологии с государственными стандартами РФ

Понятия программы, программного компонента, программного комплекса, используемые в работе, соответствуют определениям ГОСТ 19.101-2024 [207]:

– 3.1 *Программа* – совокупность команд и данных, обеспечивающая выполнение заданной последовательности действий средствами вычислительной техники.

– 3.3 *Программный компонент* – программа, рассматриваемая как единое целое, выполняющая законченную функцию. *Программный комплекс* — программа, состоящая из двух или более программных компонентов (или других комплексов), выполняющих взаимосвязанные функции, и применяемая самостоятельно или в составе другого комплекса.

Понятия информации, данных соответствуют определениям ГОСТ Р ИСО 15926-1 [208]:

– 3.1.5 *данные*: представление информации в формальном виде, подходящем для коммуникации, интерпретации или обработки людьми или на ЭВМ.

– 3.1.7 *информация*: факты, концепции или инструкции.

Понятие метаданных соответствует определению ГОСТ Р 58539-2019 [209]:

– 4.1.7 метаданные: данные, которые определяют и описывают другие данные.

---

[27] В тексте работы спецификации AMN называются также *конструкциями* AMN или *машинами* AMN.

– 4.1.10 регистр метаданных: хранилище метаданных или база данных, поддерживаемая регистром метаданных.

Понятия интероперабельности и семантической интероперабельности соответствуют определениям ГОСТ Р 550062–2021 [210] и 59797–2021 [211]:

– 3.1.8 интероперабельность: способность двух или более информационных систем или компонентов к обмену информацией и к использованию информации, полученной в результате обмена.

– 3.1.16 семантическая интероперабельность: свойство (способность) взаимодействующих информационных систем одинаковым образом интерпретировать смысл информации, которой они обмениваются.

Понятие интеграции данных соответствует определениям ГОСТ ИСО 15926-1 [208], пункт 4.1, Примечание 2: *интеграция данных* означает объединение информации, полученной из нескольких независимых источников, в один логически последовательный набор данных.

Понятия модели и метамодели соответствуют определениям ГОСТ Р 58539–2019 [209]:

– 4.1.11 *метамодель*: модель, описывающая совокупность моделей. Формируется с помощью специального языка, отражающего особенности рассматриваемых моделей.

– 4.1.12 *модель*: представление некоторых аспектов рассматриваемой предметной области с помощью нормативных инструментов моделирования и конструктивных элементов моделей.

– 4.1.13 *конструктивный элемент модели*: единица описания при представлении модели.

– 4.1.17 *репозиторий модели*: репозиторий для хранения моделей.

Понятие хранилища данных соответствует определению ГОСТ ИСО 15926-2 [212], пункт 3.1.5, *хранилище данных*: хранение данных, при котором родственные данные объединяются с целью предоставления интегрированного набора данных, не содержащего дублирующей или избыточной информации и поддерживающего различные прикладные варианты.

В рамках рассматриваемых систем интеграции данных (предметных посредников и хранилищ данных) достигается *уровень зрелости интероперабельности* 3 – Организационный (Среда взаимодействия – унифицированная: использует метамодели, что позволяет сопоставлять между собой различные гетерогенные системы) в соответствии с ГОСТ Р ИСО 11354-2–2016 [213].

Понятие масштабируемости соответствует определению ГОСТ Р 550622-2021 [210], пункт 3.1.12, *масштабируемость*: способность обеспечивать функциональные возможности вверх и вниз по упорядоченному ряду прикладных платформ, отличающихся по быстродействию и ресурсам.

Понятие верификации соответствует определению ГОСТ Р 59352–2021 [214], пункт 3.1.2, *верификация*: подтверждение (на основе предоставления объективных подтверждений) того, что заданные требования полностью выполнены. Особенность работы состоит в том, что она сосредоточена на *формальной верификации* с использованием средств формального логического доказательства, когда требования могут быть выражены на формальном логическом языке.

Схемы, конформные рассматриваемым в работе моделям данных, соответствуют определению ГОСТ Р ИСО 15926-2 [212], пункт 3.1.2, *концептуальная модель данных*: модель данных[28], определенная стандартом ИСО/ТО 9007, в котором структура данных представляется в форме, не зависящей от какого-либо способа физического хранения или формата внешнего представления.

Семантика основных элементов канонической модели данных (языка СИНТЕЗ) – типов, отношения тип-подтип, классов, атрибутов, отношения класс-экземпляр, отношения класс-подкласс, объектов – соответствует концепциям модели данных (подраздел 4.6 ГОСТ Р ИСО 15926-2 [212]).

Термины «онтология» и ее основные элементы «понятия» соответствуют определению ГОСТ Р 58539–2019 [209]:

– 4.1.20 *онтология*: спецификация конкретных (абстрактных) сущностей и отношений между ними в рассматриваемой предметной области знаний.

– 4.1.2 *понятие*: единица знаний, создаваемая уникальной комбинацией характеристик.

[28] Следует отметить, что в данном определении под *моделью данных* понимается модель уровня M1 по классификации MOF, то есть *схема данных*. В стандарте ИСО/ТО 9007 также используется понятие *концептуальной схемы*.

2 Метод конструирования сохраняющих семантику отображений моделей данных

Глава основана на работах автора [50] [85] [97] [102] [78] [82] [45] [46] [101].

2.1 Определение унификации моделей данных и ее этапы

Инфраструктура данных объединяет источники данных, которые представлены с использованием различных моделей данных в модельно неоднородную среду. Для обеспечения возможности совместного использования данных из различных источников и их интеграции эти модели данных и соответствующие им языки манипулирования данными должны быть *унифицированы* в рамках некоторой канонической модели данных *C*.

Основным принципом конструирования (синтеза) канонической модели для инфраструктуры данных является *расширяемость ядра* канонической модели в неоднородной среде, чтобы охватить особенности моделей исходных данных [41]. Ядро $Kernel_C$ канонической модели фиксируется. Конкретная исходная модель данных *R* среды *унифицирована*, если создано такое расширение *E* ядра канонической модели (это расширение может быть пустым) и такое отображение *M* исходной модели в расширенную каноническую модель, что исходная модель *уточняет* расширенную каноническую модель. Уточнение модели *C* моделью *R* означает, что для любой допустимой спецификации (схемы) *S*, определенной с использованием модели *R*, ее образ *M(S)* в модели *C уточняется* спецификацией *S* (рис. 2.1). Горизонтальные стрелки показывают направление уточнения: схема *S уточняет M(S)*, или, другими словами, *S* – это *уточнение M(S)*.

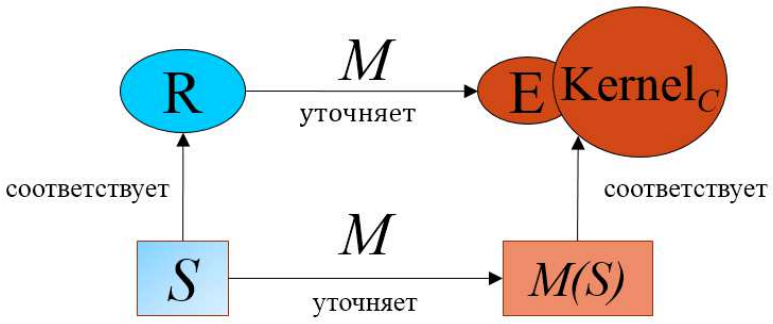

Рисунок 2.1 – Унификация исходной модели данных

Такое уточняющее отображение моделей означает сохранение информации и операций исходной модели (т.е. ее семантику) после преобразования ее схем в каноническую модель. В результате унификации также должна быть получена

возможность формально доказывать уточнение произвольной конкретной схемой *S* модели *R* ее образа *M(S)*[29].

Каноническая модель для инфраструктуры данных синтезируется как объединение расширений, построенных для всех моделей данных используемых источников (рис. 2.2).

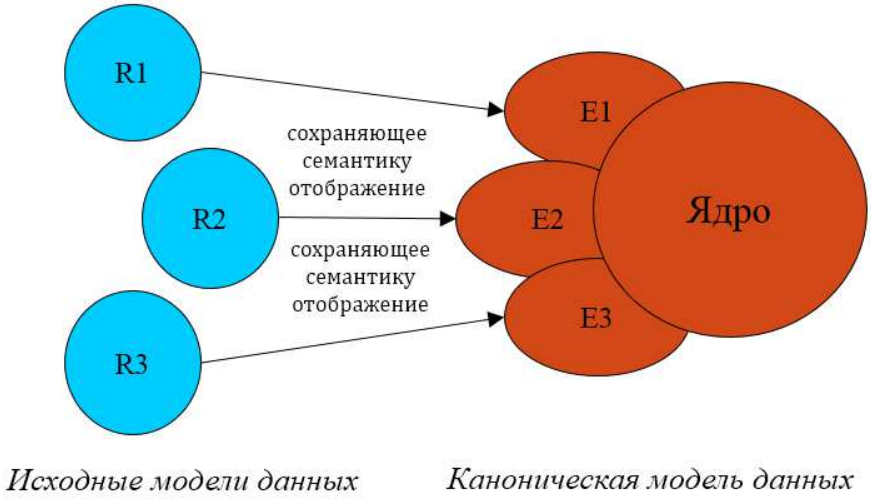

Рисунок 2.2 – Конструирование расширяемой унифицирующей модели данных

Для обеспечения деятельности по унификации моделей данных необходимы следующие основные языки и формализмы:

– ядро канонической информационной модели;

– формализм, позволяющий описывать синтаксис моделей данных;

– формализм, позволяющий описывать трансформации из одной модели данных в другую;

– формализм, поддерживающий доказательство уточнения.

В качестве ядра канонической информационной модели в настоящей работе в большинстве случаев рассматривается язык СИНТЕЗ [42], ориентированный на семантическую интероперабельность и композиционное проектирование информационных систем в широком диапазоне существующих неоднородных информационных компонентов. Однако, метод унификации моделей может быть

[29] Построение отображений моделей данных – это сложная задача. Для одной и той же пары моделей данных можно построить совершенно разные и даже некорректные отображения. Цель метода унификации, рассматриваемого в данной главе – построение доказуемо корректных, сохраняющих информацию и операции отображений.

применен также в случаях когда в качестве ядра выбрана какая-либо другая модель данных, например, SQL или RDF.

В качестве примеров формализмов описания синтаксиса в данной работе рассматриваются языки SDF (Syntax Definition Formalism) и CS (Concrete Syntax Specification Language). В качестве примеров формализмов описания трансформаций моделей – языки ASF (Algebraic Specification Formalism) и ATL (ATLAS Transformation Language). Языки SDF и ASF поддержаны программной средой Meta-Environment[30] [191], язык CS – плагином EMFText[31] среды Eclipse, язык ATL – средой разработки[32], встроенной в Eclipse.

Для формализации семантики моделей данных и верификации уточнения используется язык AMN, основанный на логике предикатов первого порядка и теории множеств и поддержанный технологией и инструментальными средствами доказательства уточнения.

Деятельность по унификации исходной модели *R* в канонической модели *C*, осуществляемая экспертом при поддержке инструментальных средств унификации моделей (Унификатора моделей), разбивается на следующие этапы:

- формализация синтаксиса моделей *R* и *C*;
- интеграция абстрактных синтаксисов моделей *R* и *C*;
- создание необходимого расширения *E* канонической модели *C;*
- построение трансформации модели *R* в расширенную каноническую модель;
- формализация семантики моделей *R* и *C*;
- верификация уточнения моделью *R* расширенной канонической модели.

Формализация синтаксиса и семантики канонической модели осуществляется один раз и используется затем при унификации всех исходных моделей данных неоднородной среды.

Основные идеи этапов унификации состоят в следующем:

1. *Формализация синтаксиса моделей данных.* Входными данными этого этапа является синтаксис моделей, изложенный в некоторых нормативных документах

---

[30] https://github.com/cwi-swat/meta-environment

[31] https://devboost.github.io/EMFText/

[32] https://eclipse.dev/atl/

(описаниях соответствующих языков). Синтаксис моделей чаще всего представляется в каком-либо варианте формы Бэкуса–Наура.

а) По нормативному описанию синтаксиса создается *абстрактный синтаксис* модели, представляющий собой модель уровня MOF M2, соответствующую некоторой выбранной метамодели уровня MOF M3, например, Ecore [196], используемая в проекте Eclipse Modeling Project[33].

б) Также с использованием выбранного языка определения конкретного синтаксиса (например, SDF или CS) создается *грамматика конкретного синтаксиса* модели (соответствующая абстрактному синтаксису), и это новое определение считается формальным описанием конкретного синтаксиса.

в) На основании грамматики конкретного синтаксиса инструментарий *текстового определения синтаксиса* автоматически генерирует *синтаксический анализатор* (парсер) схем модели данных. Синтаксический анализатор может принимать на вход схемы модели данных, проверять их соответствие синтаксису и автоматически генерировать по схемам модели уровня MOF M1, соответствующие модели абстрактного синтаксиса (рисунок 2.3).

Детали и примеры применения метода на данном этапе приведены ниже в разделе 2.2.

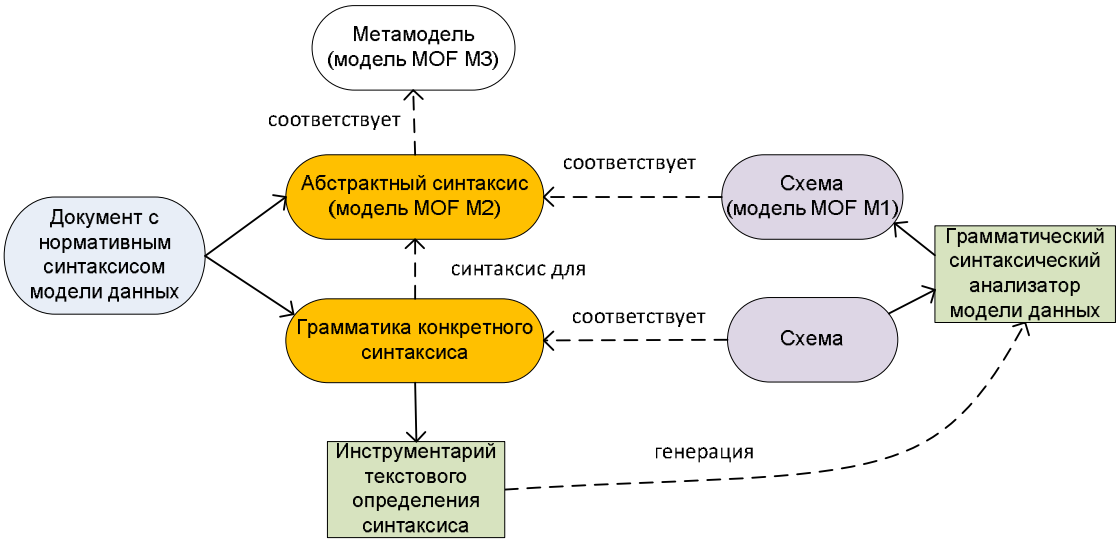

Рисунок 2.3 – Формализация синтаксиса моделей данных

[33] https://projects.eclipse.org/projects/modeling

2. *Интеграция абстрактных синтаксисов моделей*. Абстрактный синтаксис модели данных – это описание, содержащее значимые связи между понятиями, соответствующими конструкциям (элементам) модели. Понятия и связи, составляющие абстрактный синтаксис, могут быть аннотированы вербальными определениями. Целью интеграции абстрактных синтаксисов является выделение релевантных конструкций исходной и канонической моделей. Входными данными этапа являются синтаксисы моделей и их вербальная семантика, описанная в нормативных документах.
    а) Соответствия между релевантными элементами исходной и канонической моделей могут быть установлены экспертом вручную с использованием, например, инструментов, подобных простому редактору *Ecore2Ecore*, входящему в проект Eclipse Modeling Framework[34].
    б) Классический подход к автоматизированному установлению соответствий – автоматическое порождение векторов дескрипторов элементов синтаксиса на основе вербальных определений и поиск элементов синтаксиса канонической модели, близких к элементам синтаксиса исходной модели, с использованием вычисления меры близости векторов.
    в) Также для установления соответствий могут быть применены различные современные инструменты сопоставления схем [215] [216], вплоть до использующих большие языковые модели [217].
    г) После применения автоматизированных инструментов эксперт подтверждает либо отвергает найденные соответствия, а также добавляет соответствия, не найденные автоматически.
    д) Результатом интеграции является список соответствий элементов исходной и канонической моделей. Одному элементу канонической модели может соответствовать несколько конструкций исходной модели (и наоборот).

    Детали и примеры применения метода на данном этапе приведены ниже в разделе 2.3.
3. *Создание необходимого расширения канонической модели.* К необходимости создания расширения эксперт приходит в процессе интеграции эталонных схем исходной и канонической моделей: обнаруживается, что средств ядра (или ядра с

[34] https://eclipse.dev/emf/

уже построенными расширениями) канонической модели недостаточно для выражения некоторых конструкций исходной модели. К этому выводу эксперт приходит, если:

а) между конструкциями исходной модели и конструкциями канонической модели алгоритмы интеграции не установили связей;
б) на основе анализа документов, описывающих синтаксис и семантику исходной и канонической модели сделано экспертное заключение о невыразимости конструкций исходной модели в канонической.

При создании расширения для канонической модели предполагаются определенными:

а) нормативные документы, описывающие синтаксис и семантику модели;
б) формальный синтаксис;
в) AMN-семантика (т.е. вербальные правила и инструментальные средства отображения в язык AMN).

Для исходной модели необходимо обеспечить:

а) нормативные документы, описывающие синтаксис и семантику;
б) формальный синтаксис;
в) AMN-семантику;
г) установленные автоматически или с помощью эксперта и подтвержденные экспертом соответствия элементов исходной и канонической моделей;
д) неформальное определение отображения тех конструкций исходной модели в конструкции канонической, для которых установлено соответствие.

Создание расширения включает следующие шаги (рисунок 2.4):

а) Эксперт определяет название расширения и расширяемую версию канонической модели.
б) Дальнейшая деятельность эксперта заключается в итеративном редактировании синтаксиса расширения и вербальной семантики (включая правила отображения AMN). При этом эксперт обращается к описаниям исходной и канонической моделей, к интерфейсу интеграции абстрактных синтаксисов моделей. Одновременно эксперт осуществляет дополнение вербального описания отображения исходной модели в каноническую для недостающих конструкций.
в) Далее уточняется картина интеграции абстрактных синтаксисов исходной модели и расширенной канонической модели: устанавливаются и

фиксируются соответствия между конструкциями исходной модели и конструкциями расширения.

г) Относящаяся к расширению метаинформация фиксируется в *реестре моделей* (специализированной базе данных).

Детали и примеры применения метода на данном этапе приведены ниже в разделе 2.4.

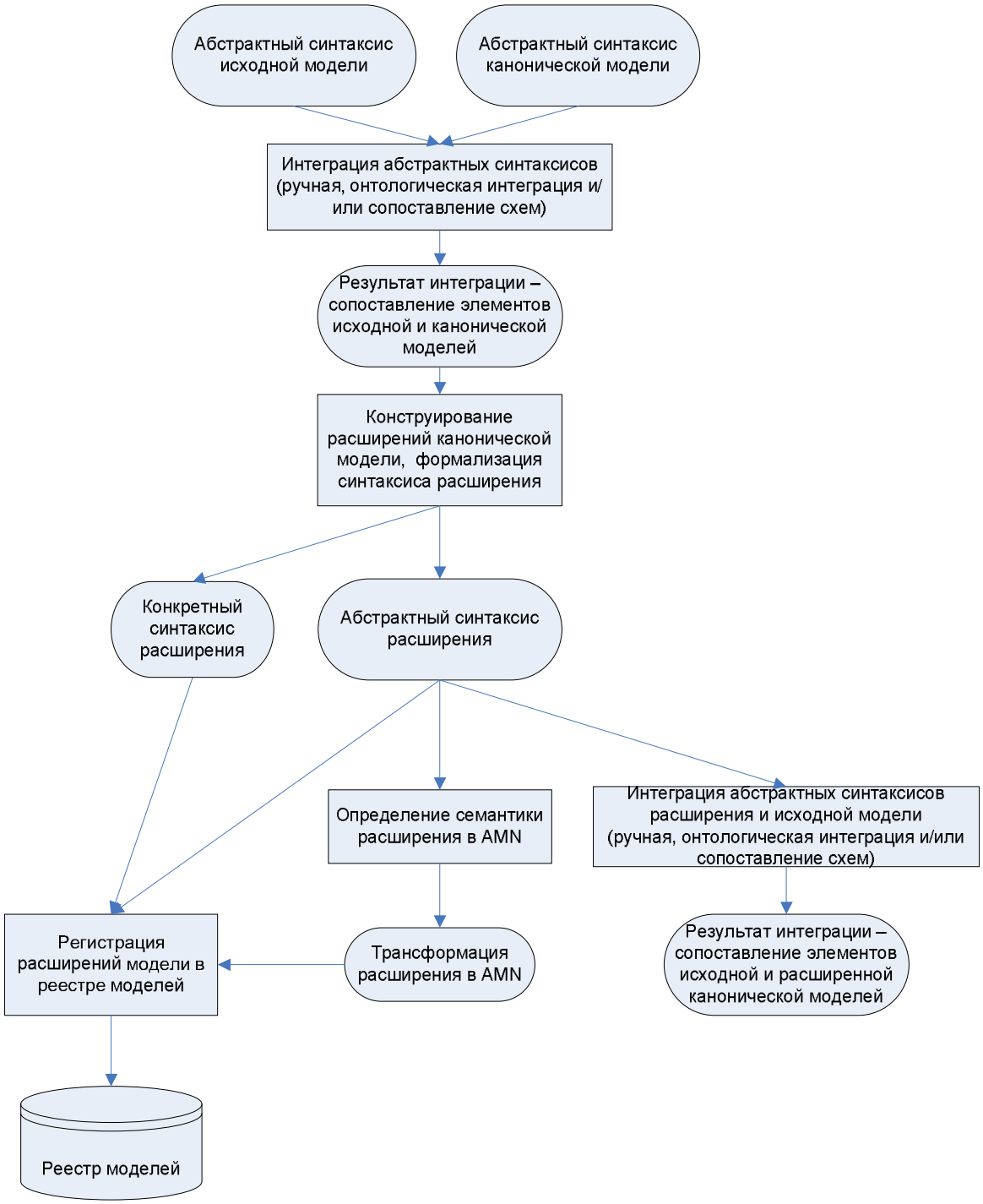

Рисунок 2.4 – Создание расширения канонической модели

4. *Построение трансформации исходной модели в расширенную каноническую модель.* Входными данными этапа являются список соответствий конструкций исходной и канонической моделей, синтаксис и вербальная семантика исходной и канонической моделей (рис. 2.5).
    а) Эксперт разрабатывает неформальные правила отображения исходной модели в расширенную каноническую.
    б) Автоматически генерируется заготовка трансформации исходной модели в расширенную каноническую. Заготовка строится на основе информации, содержащейся в списке соответствий элементов исходной и канонической моделей и синтаксисах моделей.
    в) Эксперт формализует вербальные правила отображения, превращая автоматически порожденную заготовку трансформации в полноценную трансформацию исходной модели в каноническую.

    Детали и примеры применения метода на данном этапе приведены ниже в разделах 2.5, 2.6.

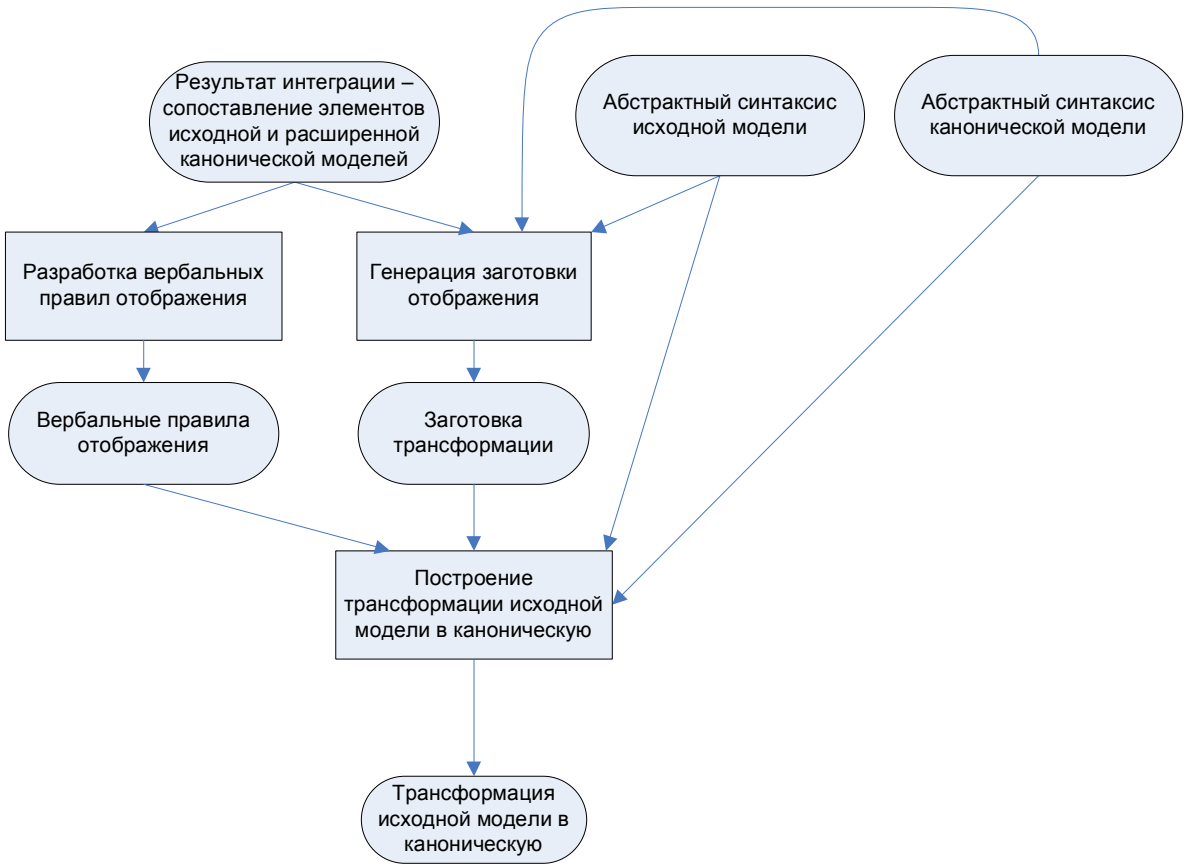

Рисунок 2.5 – Построение трансформации исходной модели в расширенную каноническую модель

5. *Формализация семантики моделей.* Входными данными этого этапа являются описания семантики моделей, изложенные в описаниях соответствующих языков.

Семантика модели может быть как вербальной, представленной в виде комментария к синтаксису, так и частично формализованной (например, с использованием логики и теории множеств). Семантика модели формализуется экспертом в виде определяемой на выбранном языке трансформаций (например, ASF или ATL) трансформации схем модели в язык AMN. При построении семантических трансформаций моделей используется их формальный синтаксис (рисунок 2.6). Такая формализация семантики необходима для последующей верификации уточнения моделей. Детали и примеры применения метода на данном этапе приведены ниже в разделах 2.7, 2.8.

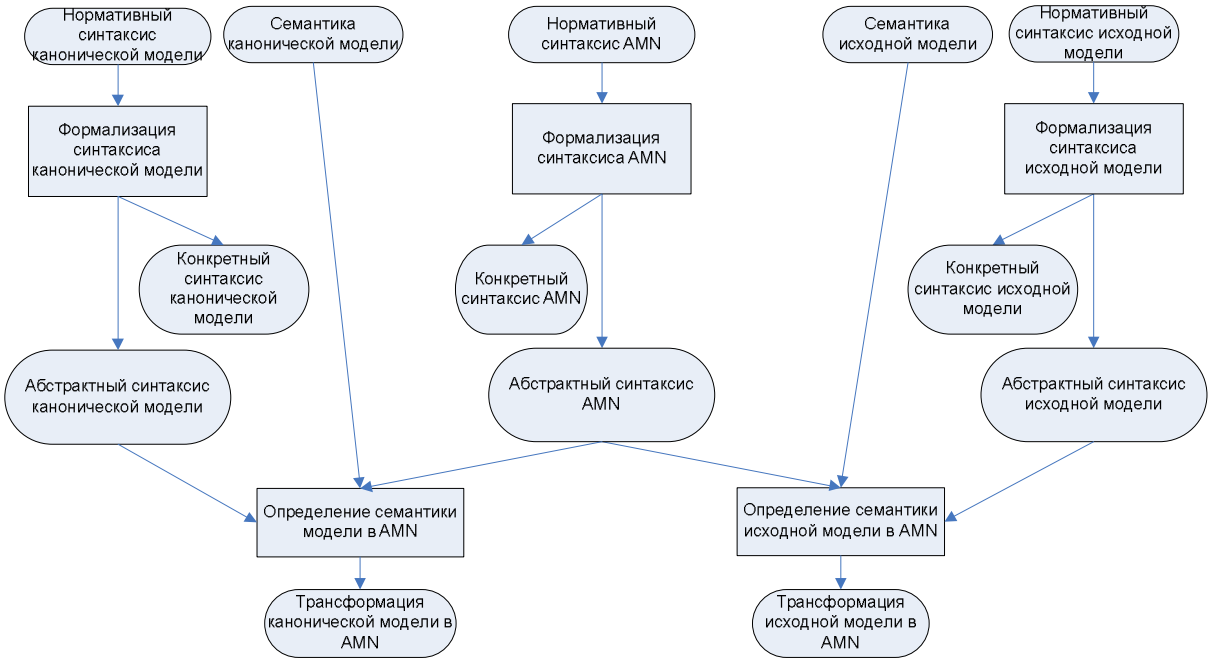

Рисунок 2.6 – Формализация семантики моделей данных

6. *Верификация уточнения исходной моделью расширенной канонической модели.* Верификация уточнения осуществляется на наборе образцов спецификаций исходной модели. Желательно, чтобы образец являлся достаточно характерной спецификацией исходной модели (типовой конструкцией). Примерами могут служить образцы потоков работ (workflow patterns) [218], использованные для синтеза канонической процессной модели (приложение А.2). В случае, когда для модели затруднительно выделить образцы, верификация может быть проведена для используемых при интеграции данных конкретных схем источников, представленных в исходной модели. Идея верификации уточнения образцом *S* исходной модели его образа *CS* при отображении в каноническую модель состоит в следующем: образец *S* и его образ *CS* отображаются в язык AMN, и уточнение доказывается средствами B-технологии для соответствующих AMN-

спецификаций. Уточнение AMN-спецификаций будет свидетельствовать об уточнении образцом *S* его образа *CS* (рисунок 2.7):

а) Экспертом выделяются образцы исходной модели, строятся их спецификации S_i и проверяются на соответствие формальному синтаксису.

б) С использованием трансформации исходной модели в AMN (построенной на этапе 5) автоматически порождаются AMN-спецификации S_i^{AMN} для образцов.

в) С использованием трансформации исходной модели в каноническую (построенной на этапе 4в) автоматически порождаются канонические спецификации CS_i образцов.

г) С использованием трансформации канонической модели в AMN (построенной на этапе 5), автоматически порождаются канонические AMN-спецификации CS_i^{AMN} для образцов.

д) Экспертом с использованием средств B-технологии доказывается уточнение спецификациями S_i^{AMN} спецификаций CS_i^{AMN}. Сначала применяются автоматические средства доказательства. Теоремы, которые не удалось доказать автоматически, доказываются при помощи интерактивных средств доказательства и участием эксперта. Текст доказательств фиксируется. Если какие-то теоремы не удается доказать – это говорит об ошибке в отображении моделей. Необходимо проверить и скорректировать отображение конструкций, свойства которых верифицируются недоказанными теоремами, а затем повторить доказательство.

Детали и примеры применения метода на данном этапе приведены ниже в разделе 2.8.1.

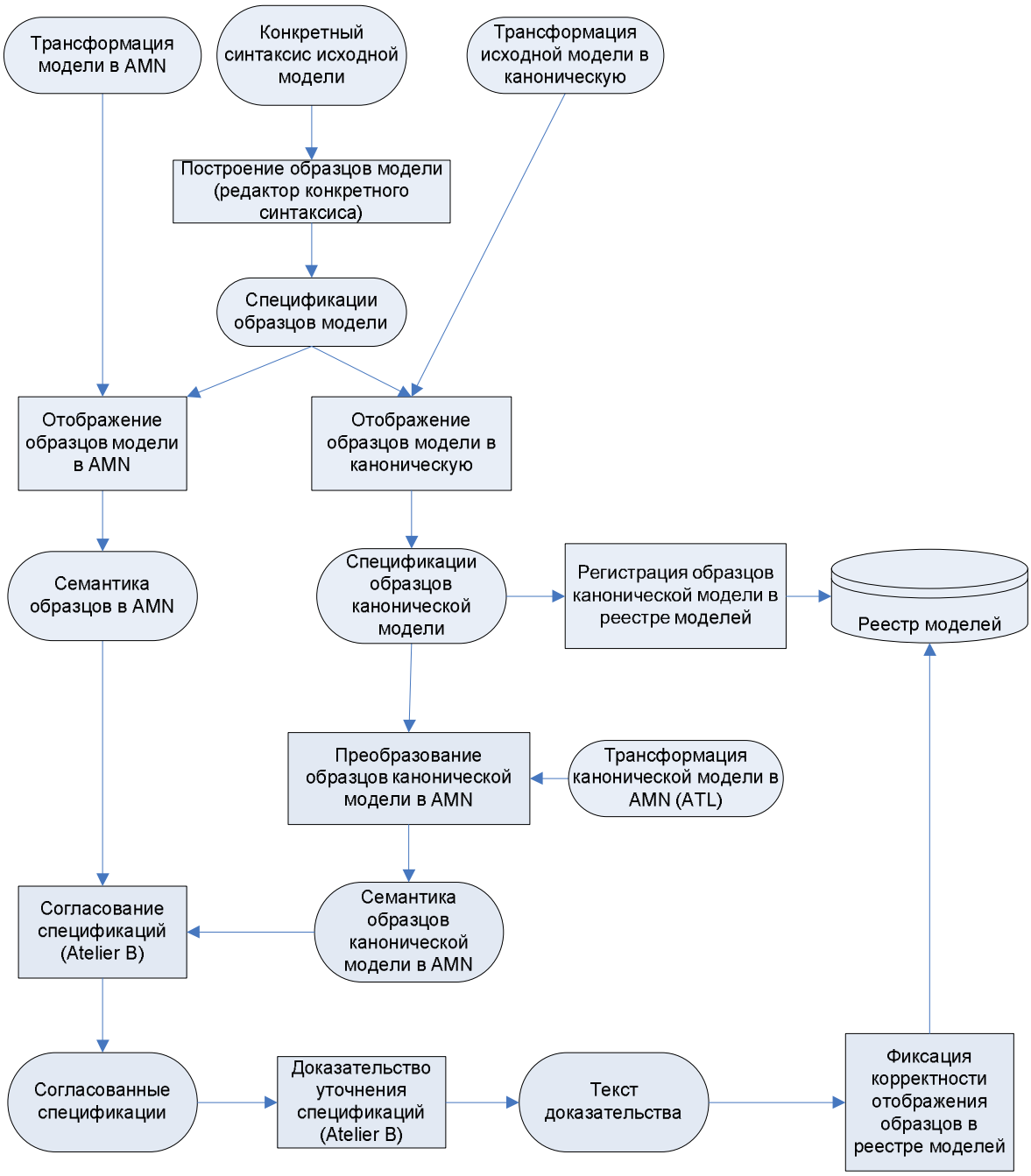

Рисунок 2.7 – Верификация уточнения исходной моделью расширенной канонической модели

Заметим, что метод верификации уточнения моделей на наборе образцов спецификаций исходной модели не является единственно возможным. Семантика модели данных R может быть представлена в виде единой AMN-спецификации R_{AMN}. При этом структуры данных модели – например, классы, массивы и т.д. – представляются переменными спецификации, различные свойства структур данных представляются инвариантами, характерные операции моделей данных представляются операциями AMN. Уточнение моделей данных при этом сводится к уточнению их семантических AMN-спецификаций (раздел 2.8.2).

Также в случаях, когда одна из моделей данных имеет строго определенную нормативную семантику в некоторой семантической структуре, а другая модель (ее подмножество) допускает определение своей семантики в той же семантической структуре, возможно доказательство корректности *взаимного отображения* моделей данных. Взаимным называется такое отображение моделей данных, которое определено как множество взаимооднозначных соответствий между элементами моделей. При этом доказательство эквивалентности семантики элементов моделей, связанных отображением, проводится индукцией по синтаксису моделей (раздел 2.8.3).

Каждый из методов верификации имеет свои преимущества и недостатки. Так, верификация на наборе образцов позволяет разделить доказательства уточнения моделей на непротиворечивые фрагменты, соответствующие образцам. Однако, покрытие всего множества возможных схем исходной модели образцами подобно покрытию программного кода тестами – невозможно строго доказать его полноту заранее. Однако, для каждой конкретной схемы источника данных доказательство уточнения возможно.

Верификация на основе единых спецификаций моделей более сложна, поскольку предполагает, что все структуры модели и характерные операции выражены в одной спецификации, а не разделены на множество образцов. Однако, при этом доказательство обладает естественным единством и комплексностью.

Доказательство корректности взаимного отображения моделей данных с математической точки зрения даже сильнее, чем уточнение моделей (фактически, это взаимное уточнение моделей). Однако, такое отображение не всегда возможно. Кроме того, для такого рода верификации нужна семантика обеих моделей в единой семантической структуре, что тоже не всегда возможно и удобно. Также, для доказательства корректности уже неприменима B-технология. В разделе 2.8.3 доказательства проводятся вручную. Вопросы применения для доказательства других автоматизированных систем, подобных Isabelle/HOL[35] или Lean[36], выходят за рамки данной работы.

[35] https://isabelle.in.tum.de/

[36] https://lean-lang.org/

Два варианта программной архитектуры и реализации инструментальных средств унификации моделей данных, а также их сравнение, рассмотрены в разделе 2.9 данной главы.

Детальные примеры применения метода унификации моделей данных для различных видов моделей (сетевой, процессной, онтологической, графовой, тензорной, на логических правилах) приведены в приложении А.

2.2 Формализация синтаксиса моделей данных

Формализация синтаксиса моделей данных проиллюстрирована в данном разделе на примерах подмножеств канонической модели данных (языка СИНТЕЗ) и онтологической модели данных OWL.

2.2.1 Формализация абстрактного синтаксиса

Формализация синтаксиса модели данных производится на основании некоторого нормативного документа, описывающего модель данных. Для языка СИНТЕЗ таким описанием является работа [42]. Фрагмент нормативного синтаксиса для абстрактного типа данных в расширенной форме Бэкуса-Наура выглядит следующим образом ([42], раздел 6.4):

```
<abstract type> ::= {<type identifier>;
  in: type;
  [supertype: <supertype list>]
  [<attribute specification list>]}
<supertype list> ::=
  <type identifier> | <type identifier>, <supertype list>
```

Элементы синтаксиса, заключенные в угловые скобки, обозначают нетерминальные символы, квадратные скобки означают опциональность элемента, вертикальная черта – альтернативные варианты.

На основании нормативного описания синтаксиса создается *абстрактный синтаксис* модели данных. Например, абстрактный синтаксис может быть представлен в виде модели уровня M2, конформной метамодели *Ecore* (реализация OMG Essential Meta-Object Facility [23], используемая в программном каркасе Eclipse Modeling Framework [196]). Элементами модели, конформной *Ecore*, могут быть классы (*EClass*), атрибуты (*EAttribute*), ссылки (*EReference*), типы данных (*EDataType*).

Общие принципы создания абстрактного синтаксиса на основе нормативной грамматики состоят в следующем:

– для нетерминального символа грамматики (например, <*abstract type*>) создается класс модели синтаксиса (например, *ADTDef*). Исключения, для которых отдельный класс не создается, составляют, в частности:

1) нетерминальные символы-списки (например, <*supertype list*>);

2) идентификаторы (<*identifier*>), представляемые в модели строковыми значениями;

– если нетерминальный символ (например, <*formula*>) задается набором альтернатив (например, <*quantified formula*> | *(*<*formula*>*)* | <*formula*> *&* <*formula*>), то класс, соответствующий альтернативе, становится подклассом класса, соответствующего этому нетерминальному символу (например, классы *QuantifiedFormula, BracketFormula, Conjunction* становятся подклассами класса *Formula*);

– если нетерминальный символ (например, <*abstract type*>) задается только одной альтернативой, то его определение служит основой для выделения атрибутов и ссылок соответствующего класса. Например, вхождение нетерминального символа <*type identifier*> в определение <*abstract type*> порождает включение атрибута *name* в класс *ADTDef*, а вхождение опционального нетерминального символа *[*<*attribute specification list*>*]* порождает включение ссылки *attributes* на класс *AttributeDef* с кардинальностью 0..* (у АТД может не быть собственных атрибутов, а может быть несколько атрибутов).

Заметим, что применение этих принципов позволяет создать заготовку модели абстрактного синтаксиса, превращение же заготовки в полноценную модель синтаксиса, непротиворечиво отражающую все аспекты нормативного синтаксиса, остается за экспертом.

Так, нормативный синтаксис языка СИНТЕЗ формализован в виде модели *Synthesis* (код модели опубликован в репозитории GitHub[37]). Например, синтаксис АТД формализуется следующим классом *ADTDef* метамодели *Ecore*:

```
<eClassifiers xsi:type="ecore:EClass" name="ADTDef"
  eSuperTypes="#//TypeDef">
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="attributes" upperBound="-1"
    eType="#//AttributeDef"
    eOpposite="#//AttributeDef/attributeOf"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="invariants" upperBound="-1"
    eType="#//InvariantDef"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
   name="subtypes" lowerBound="1" upperBound="-1"
   eType="#//ADTDef" eOpposite="#//ADTDef/supertypes"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="supertypes" upperBound="-1" eType="#//ADTDef"
```

37 https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis/metamodel/Synthesis.ecore

```
    eOpposite="#//ADTDef/subtypes"/>
</eClassifiers>
```

Из описания можно видеть, что класс *ADTDef* наследуется от класса *TypeDef* и обладает атрибутами (*attributes*), инвариантами (*invariants*), подтипами (*subtypes*) и супертипами (*supertypes*). Инварианты, как специальный вид атрибутов, не выделены явно в нормативной грамматике АТД, однако, ввиду их существенного семантического отличия от атрибутов состояния и функциональных атрибутов, они были выделены в абстрактном синтаксисе в отдельный класс. Без «синтаксического сахара» XML это описание может быть представлено в следующем виде:

```
ADTDef(attributes: AttributeDef*, invariants: InvariantDef*,
  subtypes: ADTDef*, supertypes: ADTDef*): TypeDef
```

Знак * здесь означает кардинальность свойства: например, АТД может обладать несколькими атрибутами или инвариантами.

Нормативным документом для языка OWL является рекомендация W3C [11]. Синтаксис OWL определен в с использованием варианта формы Бэкуса-Наура. Пример правил определения аксиомы объектного свойства выглядит следующим образом:

```
ObjectPropertyAxiom :=
  InverseObjectProperties | ObjectPropertyDomain | ObjectPropertyRange |
    SymmetricObjectProperty | TransitiveObjectProperty
ObjectPropertyDomain := 'ObjectPropertyDomain'
  '(' ObjectPropertyExpression  ClassExpression ')'
```

Подмножество абстрактного синтаксиса OWL[38], представленное в метамодели Ecore и визуализированное в редакторе Ecore, изображено на рисунке 2.8. Так, класс *OWLObjectProperty*, представляющий объектные свойства, принадлежит пакету *OWL*. Класс *OWLObjectProperty* – это подкласс класса *RDFSProperty,* включающий несколько ссылок: *domain, range, superProperty*, *inverseProperty*. Классы, представляющие специальные виды объектных свойств (например, транзитивные свойства *TransitiveProperty*) определены как подклассы *OWLObjectProperty*.

38 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis/metamodel/OWL.ecore

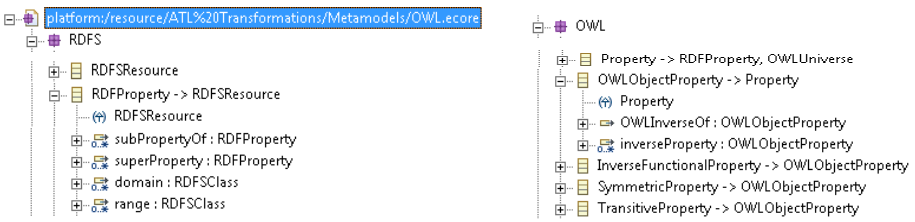

Рисунок 2.8 – Абстрактный синтаксис OWL в редакторе метамодели Ecore

Сериализация в XML элементов модели OWL из правого столбца рисунка 2.8 выглядит следующим образом:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:ecore="http://www.eclipse.org/emf/Ecore">
<ecore:EPackage name="OWL">
  <eClassifiers xsi:type="ecore:EClass" name="OWLObjectProperty"
    eSuperTypes="/1/Property">
    <eStructuralFeatures xsi:type="ecore:EReference" name="OWLInverseOf"
        eType="/1/OWLObjectProperty"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TransitiveProperty"
    eSuperTypes="/1/OWLObjectProperty"/>
</ecore:EPackage>
</xmi:XMI>
```

2.2.2 Формализация конкретного синтаксиса

Конкретный синтаксис модели данных определяется на языке формализации конкретного синтаксиса (например, CS или SDF), и сочетает модель абстрактного синтаксиса и «синтаксический сахар» (используемый для повышения читаемости текстового представления), соответствующий нормативному синтаксису. Для каждого класса модели абстрактного синтаксиса определяется грамматическое правило. Правило определяет грамматическую структуру элемента синтаксиса (в соответствии с нормативным синтаксисом) с использованием

– терминальных символов, заключенных в кавычки (например *“supertypes”* или *“{”*);

– нетерминальных символов, соответствующих атрибутам или ссылкам класса модели синтаксиса (например, *name* или *supertypes*);

– грамматических конструкций, обозначающих множественность (*), опциональность (?), альтернативу (|).

Конкретный синтаксис фрагмента языка СИНТЕЗ, формализованный с использованием языка CS из каркаса EMFText [219] выглядит следующим образом (полный синтаксис опубликован в репозитории GitHub[39]):

```
SYNTAXDEF syn FOR http://synthesis.ipi.ac.ru/Synthesis/
RULES
ADTDef ::=
  "{" name[]  ";"  "in" ":" "type"  ";"
  ("supertypes" ":"  supertypes[] ("," supertypes[])* ";")?    "}" ;
ClassDef ::=
  "{" name[]  ";"  "in" ":" "class"  ";"
  ("superclass" ":"  superclasses[] ("," superclasses[])* ";")?   "}" ;
```

Аналогично, правило конкретного синтаксиса OWL для объектного свойства выглядит следующим образом (полный синтаксис опубликован в репозитории GitHub[40]):

```
SYNTAXDEF owl
FOR <http://org.emftext/owl.ecore> <owl.genmodel>
RULES{
OWLObjectProperty ::= "ObjectProperty:" iri[IRI] (
  | "Domain:" domain ("," domain)*
  | "Range:" range ("," range)*
  | "SubObjectPropertyOf:" subPropertyOf ("","subPropertyOf)*
  | "ObjectInverseOf:" OWLInverseOf
 )*;
}
```

Пример применения языка SDF для формализации конкретного синтаксиса объектных свойств OWL выглядит следующим образом:

```
module unifier/owl/OWL-Syntax

sorts
  Axiom ObjectPropertyAxiom

context-free syntax
  ObjectPropertyAxiom -> Axiom

  "ObjectProperty" "(" PropertyID Annotation*
  SuperProperty* InverseOf? ("Symmetric")? PropertyKind?
  ObjectPropertyDomain* ObjectPropertyRange* ")"
  -> ObjectPropertyAxiom
```

Как можно видеть, формальный синтаксис OWL образует SDF-модуль *OWL-Syntax*, включающий раздел сортов (куда попадают все нетерминальные символы синтаксиса) и раздел контекстно-свободного синтаксиса (включающий грамматические правила). Разделителем в правилах SDF является символ ->, голова правила

---

39 https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis/metamodel/Synthesis.cs

40 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis/metamodel/owl.cs

располагается справа от разделителя, тело – слева. Опциональные части правил помечаются знаком вопроса ?, повторяющиеся части – знаком *.

Инструментальные средства поддержки разработки конкретного синтаксиса (например, EMFText, Meta-Environment) предоставляют возможность автоматической генерации синтаксического анализатора (парсера) текстов, и редактора схем модели данных. Каждая схема модели данных, удовлетворяющая грамматике конкретного синтаксиса, может быть автоматически преобразована в модель уровня M1, конформную модели абстрактного синтаксиса.

2.3 Интеграция абстрактных синтаксисов моделей: установление соответствий между элементами моделей

Интеграция абстрактных синтаксисов направлена на идентификацию релевантных элементов исходной и канонической моделей. Эти пары релевантных элементов (*соответствия*) предназначены для того, чтобы служить отправной точкой построения трансформации исходной модели в каноническую.

Так, на рис. 2.9 изображены некоторые из элементов моделей абстрактного синтаксиса языков OWL и СИНТЕЗ (для лучшей читаемости они представлены в виде диаграммы классов UML).

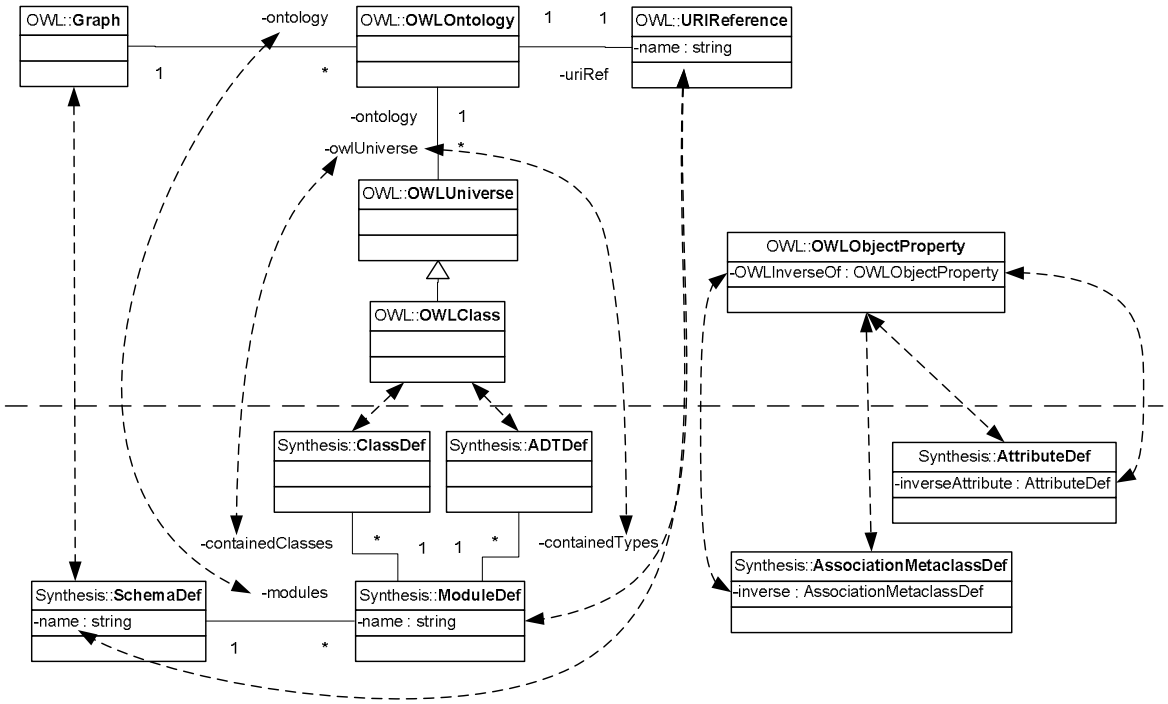

Рисунок 2.9 - Соответствия между элементами моделей OWL и Synthesis

В верхней части диаграммы изображены элементы модели *OWL*, в нижней части – элементы модели *Synthesis*.

Элементами моделей, участвующих в соответствиях, могут быть классы (например, *OWLOntology* – элемент типа *Eclass* метамодели *Ecore*), атрибуты (например, *URIReference.name* – элемент типа *EAttribute* в *Ecore*), связи между классами (например, *Graph.ontology* – элемент типа *EReference* в *Ecore*). Связи могут быть различной кардинальности: один-к-одному или один-ко-многим.

Основным элементом спецификации OWL является граф (*Graph*), состоящий из онтологий (*OWLOntology*). Основным элементом спецификации языка СИНТЕЗ является схема (*SchemaDef*), состоящая из модулей (*ModuleDef*). Модули содержат классы (*containedClasses*) и типы (*containedTypes*). Структуру типов составляют атрибуты (*AttributeDef*). Для описания специализированных свойств атрибутов, их категоризации, используются метаклассы ассоциаций (*AssociationMetaclassDef*). Атрибут может являться экземпляром некоторых метаклассов ассоциаций [42].

Соответствия между элементами моделей обозначены на рисунке пунктирными линиями, в тексте соответствие обозначается знаком ~. Соответствие элементов моделей может быть установлено между классами (*Graph ~ SchemaDef*; граф соответствует схеме), между связями (*OWLOntology.ontology ~ SchemaDef.modules*), между атрибутами (*OWLOntology.uriRef.name ~ ModuleDef.name*). Элементу одной модели может соответствовать несколько элементов другой модели. Если обозначить множество элементов исходной модели (*OWL*) через E_S, множество элементов целевой модели (СИНТЕЗ) – через E_T, то множество соответствий R между элементами моделей есть подмножество декартова произведения $E_S \times E_T$ (т.е. отношение на этих множествах). Список соответствий элементов моделей, изображенных на рис. 2.9, приведен в таблице 2.1.

В выражениях пути, задающих элементы моделей, точка означает навигацию по атрибуту или связи один-к-одному, стрелка (->) – навигацию по связи один-ко-многим.

Соответствия могут быть установлены экспертом вручную с помощью таких инструментов, как редактор *Ecore2Ecore* (который является компонентом основных библиотек Eclipse EMF), или с помощью более продвинутых инструментов сопоставления схем [220]. Например, интеграция может быть частично автоматизирована на основе вербальных определений, аннотирующих элементы модели абстрактного синтаксиса.

Таблица 2.1 - Список соответствий элементов моделей *OWL* и *Synthesis*

Элемент модели OWL	*Элемент модели Synthesis*
Graph	SchemaDef
OWLOntology	ModuleDef
OWLClass	ADTDef, ClassDef
OWLObjectProperty	AttributeDef, AssociationMetaclassDef
Graph.ontology->uriRef.name	SchemaDef.name
Graph.ontology	SchemaDef.modules
OWLOntology.uriRef.name	ModuleDef.name
OWLOntology.owlUniverse	ModuleDef.containedTypes, ModuleDef.containedClasses
OWLObjectProperty.OWLInverseOf	AttributeDef.inverseAttribute, AssociationMetaclass.inverse

По вербальным описаниям конструируются представления (embeddings) для элементов – в простейшем случае это могут быть представления на основе TF-IDF, а также представления, сгенерированные языковыми моделями (например, BERT). Однако, в большинстве случаев, соответствия должны быть дополнены и подтверждены экспертом.

Конечный список соответствий может быть представлен в различных форматах, например, в *ecore2ecore*, и сериализован в XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore2ecore:Ecore2EcoreMappingRoot xmi:version="2.0"
xmlns:ecore2ecore="http://www.eclipse.org/emf/2004/Ecore2Ecore">
<nested>
  <inputs href="OWL.ecore#/1/OWLObjectProperty"/>
  <outputs href="Synthesis.ecore#//AttributeDef"/>
</nested>
<nested>
  <inputs href="OWL.ecore#/0/RDFProperty/range"/>
  <outputs href="Synthesis.ecore#//AttributeDef/type"/>
</nested>
<inputs href="OWL.ecore#/"/>
<outputs href="Synthesis.ecore#/"/>
</ecore2ecore:Ecore2EcoreMappingRoot>
```

Данный XML-файл содержит соответствия между элементами исходной модели *OWL.ecore* целевой модели *Synthesis.ecore*: например, класс *OWLObjectProperty* соответствует классу *AttributeDef,* а сссылка *RDFProperty.range* – ссылке *AttributeDef.type*.

2.4 Создание расширения канонической модели

Расширения канонической модели могут принимать вид различных конструкций: это могут быть специализированные типы данных, ограничения (зависимости, аксиомы), сложные операции над структурами данных.

В данном разделе иллюстрируется создание двух видов расширений: в виде специализированных метаклассов ассоциаций для унификации языка OWL и в виде специализированных классов для унификации графовой модели.

2.4.1 Расширение при помощи метаклассов ассоциаций

Расширения в виде специализированных метаклассов ассоциаций ([42], подраздел *7.4 Association metaclasses*) конструируются для унификации в языке СИНТЕЗ различных видов объектных свойств языка OWL: транзитивных, симметричных, рефлексивных и т.д. [11]. В частности, для представления *рефлексивных* свойств в языке СИНТЕЗ создан следующий метакласс ассоциаций:

```
{ Reflexive; in: association, metaclass;
instance_section: {
  domain: type; range: type;
  reflexivity: { in:nvariant;
  {{ all x,y (is_in([x,y], this) -> is_in([y,x], this)) }}
} } }
```

Метакласс ассоциаций – это множество ассоциаций, которые в свою очередь представляют собой множества связанных пар объектов. Метакласс *Reflexive* включает инвариант *reflexivity*, указывающий на рефлексивность ассоциации. Семантика метакласса определяется следующим образом. Пусть *x, y* - объекты. Если объект *x* связан с объектом *y* ассоциацией *a*, то *y* также связан с *x* ассоциацией *a*. Каждая ассоциация, объявленная как экземпляр метакласса *Reflexive*, является рефлексивной. В приведенной выше спецификации метакласса *this* обозначает текущий экземпляр ассоциации, *is_in* обозначает предикат принадлежности объекта множеству, *[x, y]* обозначает пару связанных объектов. Спецификация инварианта определяется логической формулой первого порядка: *all* обозначает квантор всеобщности, -> обозначает логическую импликацию.

Для представления *транзитивных* свойств в языке СИНТЕЗ создан метакласс ассоциаций *Transitive*:

```
{ Transitive; in: association, metaclass;
instance_section: {
  domain: type; range: type;
  transitivity: { in: predicate, invariant;
  {{ all a,b,c(in([a,b], this) & in([b,c], this) -> in([a,c], this)) }}
} } }
```

Транзитивность ассоциации выражается инвариантом *transitivity* метакласса следующим образом. Пусть *a, b, c* - объекты. Если *a* связан с *b* ассоциацией *assoc*, а *b* связано с *c* той же ассоциацией, то и *a* связан с *c* ассоциацией *assoc*. Таким образом, любая ассоциация, определенная как экземпляр метакласса *Transitive*, является транзитивной.

Метаклассы ассоциаций, такие, как *Reflexive* или *Transitive*, являются расширениями канонической модели, предназначенными для представления свойств объектов специального вида. При этом синтаксического расширения ядра канонической модели не требуется, достаточно определить специальную структуру данных и соответствующую аксиому (инвариант). Расширение такого рода может быть представлено как модель уровня MOF M1, соответствующая канонической модели уровня MOF M2, формализующей абстрактный синтаксис языка СИНТЕЗ.

При этом метаклассы ассоциаций в целом можно рассматривать как *синтаксическое расширение* канонической модели. Синтаксис расширения формализуется так же, как и синтаксис модели данных в целом (раздел 2.2). Структуры абстрактного синтаксиса метаклассов ассоциаций показаны в правой части рисунка 2.10. Он рассматривается как расширение абстрактного синтаксиса ядра канонической модели. Соответствия между элементами синтаксиса исходной модели и элементами синтаксиса расширения устанавливаются так же, как для моделей данных в целом (раздел 2.3). Соответствия между элементами синтаксиса OWL и элементами расширения показаны на рисунке XX синими стрелками. Так, объектное свойство OWL (*OWLObjectProperty*) соответствует метаклассу ассоциаций (*AssociationMetaclassDef*). Область определения *domain* и область значений *range* объектного свойства соответствуют области определения и области значений метакласса ассоциаций соответственно. Ссылка на инверсное свойство (*OWLInverseOf*) соответствует инверсной ассоциации (*inverse*) метакласса ассоциаций.

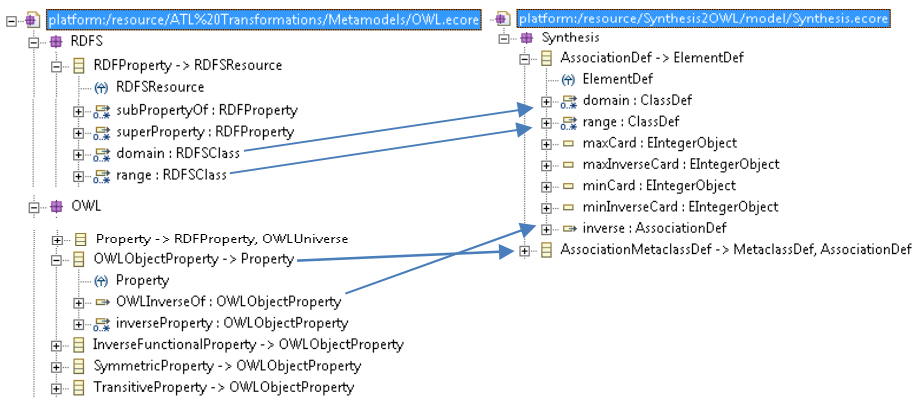

Рисунок 2.10 – Абстрактный синтаксис метаклассов ассоциаций и соответствия его элементов элементам абстрактного синтаксиса OWL

Абстрактный синтаксис расширения связывается с конкретным синтаксисом при помощи грамматических правил. Так, правило на языке CS метакласса ассоциаций выглядит следующим образом:

```
AssociationMetaclassDef ::=
"{" name[] ";" ("inverse" ":" inverse[] ";")?
  ("association_type" ":" "{" "{" minCard[DECIMAL_INTEGER_LITERAL] ","
      maxCard[DECIMAL_INTEGER_LITERAL] "}"  ","
    "{" minInverseCard[DECIMAL_INTEGER_LITERAL] ","
       maxInverseCard[DECIMAL_INTEGER_LITERAL] "}" "}" ";" )?
    ("domain" ":" domain[] ("," domain[])*  ";")?
    ("range" ":" range[] ("," range[])* ";" )?
"}";
```

Это правило рассматривается как расширение конкретного синтаксиса ядра канонической модели.

2.4.2 Расширение при помощи специализированных классов

Расширения в виде специализированных классов используются, в частности, при унификации в языке СИНТЕЗ графовой модели данных (подробно унификация графовой модели описана в разделе 2.8.2 с дополнением А.5). Расширение включает классы *vertices* и *edges,* предназначенные для представления вершин и ребер графа соответственно. Спецификация класса *edges* выглядит следующим образом (приведен фрагмент спецификации):

```
{ edges; in: class;
  instance_section: {
    startVertex: vertices.inst;
    endVertex: vertices.inst;
    isValidEdge: { in: predicate;
      params: {+stVtx/vertices.inst, +endVtx/vertices.inst,
               returns/Boolean };
      {{ (stVtx = this.startVertex &
            endVtx = this.endVertex -> returns = true) &
           (stVtx <> this.startVertex | endVtx <> this.endVertex) ->
              returns = false) }}
     };
}
```

Каждое ребро графа – это экземпляр класса *edge*. Каждое ребро имеет исходящую вершину (*startVertex*) и входящую вершину (*endVertex*). Предикат *isValidEdge* определен на парах вершин. Для ребра *e* предикат *e.isValidEdge(v1, v2)* обращается в истину тогда и только тогда, когда исходящая вершина *e* (*e.startVertex*) есть *v1* и входящая вершина *e* (*e.endVertex*) есть *v2.*

Расширение такого рода может быть представлено как модель уровня MOF M1, соответствующая канонической модели уровня MOF M2, формализующей абстрактный

синтаксис языка СИНТЕЗ, синтаксического расширения ядра канонической модели при этом не требуется.

2.5 Конструирование трансформации исходной модели в расширенную каноническую модель

Конструирование трансформаций моделей данных производится при помощи языков трансформации, примерами которых являются язык метакомпиляции ASF (Algebraic Specification Formalism) и язык трансформации моделей ATLAS Transformation Language (ATL). Конструирование трансформаций производится на основе установленных соответствий элементов абстрактного синтаксиса моделей (раздел 2.3) и неформальных правил отображения моделей.

Неформальные правила отображения исходной модели в расширенную каноническую, разрабатываемые экспертом, на примере языка OWL выглядят следующим образом (приведены несколько основных правил):

– *онтология* языка OWL отображаются в одноименный *модуль* языка СИНТЕЗ;

– *класс* языка OWL отображаются в одноименный *АТД* и *класс* языка СИНТЕЗ, типом экземпляров которого является этот АТД;

– *объектное свойство* языка OWL отображаются в одноименный *атрибут* соответствующего АТД языка СИНТЕЗ;

– *ограничение значений свойств* OWL отображается в *инвариант* соответствующего класса языка СИНТЕЗ.

Применение данных правил на примере фрагмента онтологии Глобальной почвенной информационной системы GloSIS приведено в таблице 2.2 (онтология *GLoSIS*, классы *FeatureOfInterest, GL_Plot, GL_Site, Observation, CropClass*, объектное свойство *hasFeatureOfInterest*). Более подробно данный пример рассмотрен в приложении А.3.

Далее эксперт формализует вербальные правила отображения, конструируя трансформацию исходной модели в каноническую на языке трансформации. Так, структура трансформации на языке ASF выглядит следующим образом:

```
module <имя модуля>
<список импортируемых модулей>
hiddens
  variables
  <список переменных>
exports
  context-free syntax
  <список сигнатур функций трансляции>
equations
  <список правил трансляции>
```

Таблица 2.2 – Пример отображения фрагмента онтологии OWL в язык СИНТЕЗ

Онтология OWL	Спецификация СИНТЕЗ

Ontology (GLoSIS
 	
Class(FeatureOfInterest)
Class(GL_Plot
  FeatureOfInterest)
Class(GL_Site
  FeatureOfInterest)
 	
Class(Observation
  restriction(
  hasFeatureOfInterest
  cardinality(1))
)
 	
ObjectProperty(
  hasFeatureOfInterest
  domain(Observation)
  range(FeatureOfInterest)
)
 	
Class(CropClass
  Observation
  restriction(
  hasFeatureOfInterest
  allValuesFrom(
  unionOf(GL_Plot
  GL_Site)))
)
)
	{ GLoSIS; **in: module, ontology;**
	type:
	{ GL_Plot; **in: type, owl;**
	  supertype: FeatureOfInterest; },
	{ GL_Site; **in: type, owl;**
	  supertype: FeatureOfInterest; },
	{ FeatureOfInterest; **in: type, owl;** },
	{ Observation; **in: type, owl;**
	  hasFeatureOfInterest: FeatureOfInterest;
	  metaslot in:
	  HasFeatureOfInterest;
	  min_card: 1;
	  max_card: 1;
	  end
	},
	{ CropClass; **in: type, owl;**
	  supertype: Observation;
	};
	
	class_specification:
	{ gl_plot; **in: class, owl;**
	  superclass: featureOfInterest;
	  instance_section: GL_Plot;
	},
	{ gl_site; **in: class, owl;**
	  superclass: featureOfInterest;
	  instance_section: GL_Site;
	},
	{ featureOfInterest;
	  in: class, owl;
	  instance_section: FeatureOfInterest;
	},
	{ observation; **in: class, owl;**
	  instance_section: {
	  in: type, owl;
	  supertype: Observation;
	  inv_hasFeatureOfInterest: {
	  in: predicate, invariant;
	  {{ **all** obs/Observation (
	  in(obs, observation) -> **in**(obs.hasFeatureOfInterest,
	  featureOfInterest)) }}
	  };
	},
	{ cropClass; **in: class, owl;**
	  instance_section: {
	  in: type, owl;
	  supertype: CropClass;
	  restriction_hasFeatureOfInterest: {
	  in: predicate, invariant;
	  {{ **all** cc/CropClass (
	  in(cc, cropClass) -> **in**(cc.hasFeatureOfInterest,
	  union(gl_plot, gl_site) }}
	  };
	  }; }; }

Фрагмент трансформации языка OWL в язык СИНТЕЗ, созданный на основе соответствий элементов моделей, приведенных в разделе 2.3, выглядит следующим образом (полный текст трансформации опубликован в репозитории GitHub[41]):

```
module unifier/owl2synthesis/owl-translator
imports unifier/owl/OWL-Syntax
imports unifier/synthesis/Synthesis-Syntax
exports
context-free syntax
t-Type-Specification-List(Directive*, Directive*) ->
  {Type-Specification ","}*
t-Class-Declarator-List(Directive*) -> {Class-Declarator ","}*
hiddens
variables
"Directive*"[0-9\']* -> Directive*
"Type-Specification*"[0-9\']* -> {Type-Specification ","}*
equations
Type-Specification* :=
  t-Type-Specification-List(Directive*, Directive*2),
Synthesis-Id := t-Synthesis-Id(OwlID)
====>
t-Type-Specification-List(
  Class(OwlID Description*)
  Directive*,
  Directive*2
) =
{ Synthesis-Id; in: type, owl;
  t-Type-Supertype-Section(Description*)
  t-Attribute-Specification-List(OwlID, Description*, Directive*2)
},
Type-Specification*

Synthesis-Id := t-Synthesis-Id(OwlID)
====>
t-Class-Declarator-List(
  Class(OwlID Description*)
  Directive*
) =
{ Synthesis-Id; in: class, owl;
  instance_section: Synthesis-Id;
},
t-Class-Declarator-List(Directive*)
```

Трансформация импортирует определения синтаксиса исходного модели (*OWL-Syntax*) и канонической модели (*Synthesis-Syntax*). Определены две функции трансформации: первая преобразует список спецификаций классов OWL в список спецификаций АТД языка СИНТЕЗ (*t-Type-Specification-List*), а вторая преобразует список спецификаций классов OWL в список спецификаций классов языка СИНТЕЗ (*t-*

41 https://github.com/sstupnikov/ModelTransformation/tree/master/OWL2Synthesis.Meta-Environment/unifier/owl2synthesis

Class-Declarator-List). Сигнатуры функций определены в разделе трансформации *context-free syntax*. Правила трансформации определены как правила переписывания термов в разделе *equations*. Оба правила содержат условия сопоставления вида *A := B* (фактически, это инициализации вспомогательных переменных в правиле) и равенство термов. Как условия сопоставления, так и равенства термов содержат рекурсивные применения функций трансформации. Дополнительные синтаксические переменные, используемые в правилах трансформации, объявлены в разделе *variables*.

Фрагмент трансформации на языке ATL, созданный на основе тех же соответствий элементов моделей, выглядит следующим образом:

```
module OWL2Synthesis;
create OUT : Synthesis from IN : OWL;
rule OWLClass{
  from c: OWL!OWLClass
  using{
    sup: Set(OWL!OWLClass) =
       c.subClassOf->select(e | e.oclIsTypeOf(OWL!OWLClass)); }
  to
    type: Synthesis!ADTDef(
      name <- c.resourceName(),
      supertypes <- sup),
    class: Synthesis!ClassDef(
      name <- c.resourceName().toLower(),
      instanceType <- type)
  do{
    class.superclasses <-
      sup->collect(e | thisModule.resolveTemp(e, 'class'));
  }
}
```

Для элемента исходной модели синтаксиса (*OWLClass*) определено одноименное правило трансформации. Объявлен и типизирован в секции *from исходный элемент* правила (*c*). Вспомогательные переменные правила объявлены и инициализированы в секции *using*. Например, переменная *sup* инициализирована как множество суперклассов исходного класса *c*. *Целевые элементы* правила (*type, class*) объявлены и типизированы в секции *to*. Для целевых элементов определены *множества связываний*. Связывания определяют, каким образом должны инициализироваться атрибуты или ссылки целевых элементов (*name, supertypes, instanceType, superclasses*). Связывания определяются декларативно в секции *to* section либо императивно в секции *do*.

Для трансформаций моделей на основе установленных соответствий элементов абстрактного синтаксиса моделей могут быть автоматически сгенерированы их *заготовки*. Пример автоматически сгенерированной заготовки правила трансформации

для объектного свойства OWL приведен в левом столбце таблицы XX (полный код заготовки опубликован в репозитории GitHub[42]). Объектное свойство OWL соответствует атрибуту и метаклассу ассоциаций языка СИНТЕЗ. Поэтому и правило для каждого объектного свойства *op* (типизированного как *OWLObjectProperty*) создает атрибут *at* (типизированный как *AttributeDef*) и метакласс ассоциаций *am* (типизированный как *AssociationMetaclassDef*). Свойства *name*, *attributeOf*, *type* элемента *AttributeDef* и свойство *name* элемента *AssociationMetaclassDef* декларативно инициализируются на основе их соответствий (например, *AttributeDef.attributeOf ~ OWLObjectProperty.domain*) в секции *to*. Свойство *inverseAttribute* элемента *AttributeDef* и свойства *inverse, domain, range* элемента *AssociationMetaclassDef* инициализируются в императивной секции *do* правила.

Метод автоматического построения заготовок трансформаций моделей детально изложен в следующем разделе 2.6.

Несмотря на возможность автоматической генерации заготовок трансформаций моделей, трансформация должна быть завершена (расширена, модифицирована) экспертом. Пример фрагмента модифицированной экспертом трансформации приведен в правой части таблицы 2.3 (полный текст трансформации опубликован в репозитории GitHub[43]). Соответствующие куски заготовки правила и модифицированного правила в таблице выровнены. С заготовкой проведены следующие модификации:

– в секции *using* введены дополнительные переменные *domain, range*, инициализированные некоторыми входными элементами для устранения избыточности и повышения читаемости кода;

– трансформация URI-ссылки в имя элемента выделена в отдельную вспомогательную (*helper*) функцию *resourceName*;

– если объектное свойство *op* рефлексивно, то метакласс ассоциаций *am* устанавливается подклассом метакласса ассоциаций *Reflexive*, рассмотренного в разделе 2.4.1.

---

42 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis/transformation/OWL2SynthesisTemplate.atl

43 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis/transfor-mation/OWL2Synthesis.atl

Таблица 2.3 – Примеры заготовки трансформации моделей данных и соответствующей завершенной трансформации

Заготовка трансформации	Завешенная трансформация

module OWL2Synthesis;
create OUT: Synthesis **from** IN: OWL;
 	
rule OWLObjectProperty {
from op: OWL!OWLObjectProperty
 	
to
at: Synthesis!AttributeDef(
name <- op.uriRef->any(e | true).
fragmentIdentifier.name,
attributeOf <-
op.domain->any(c | true),
type <- op.range->any(c | true);
),
 	
am:
Synthesis!AssociationMetaclassDef(
name <- op.uriRef->any(e | true).
fragmentIdentifier.name)
do {
at.inverseAttribute <- thisModule.
 	
resolveTemp(op.OWLInverseOf,'at');
am.inverse <- thisModule.
resolveTemp(op.OWLInverseOf,
'am');
am.domain <- thisModule.
resolveTemp(domain, 'class');
am.range <- thisModule.
resolveTemp(range, 'class');
} }
	module OWL2Synthesis;
	create OUT: Synthesis **from** IN: OWL;
	
	rule OWLObjectProperty{
	from op: OWL!OWLObjectProperty
	using{
	domain: OWL!OWLClass = op.domain->
	any(c|
	c.oclIsKindOf(OWL!OWLClass));
	range: OWL!OWLClass = op.range->
	any(c |
	c.oclIsKindOf(OWL!OWLClass));
	}
	to
	at: Synthesis!AttributeDef(
	name <- op.resourceName(),
	
	attributeOf <- domain,
	
	type <- range
	),
	
	am:
	Synthesis!AssociationMetaclassDef(
	name <- op.resourceName() +
	'Metaclass')
	do{
	at.inverseAttribute <- thisModule.
	resolveTemp(op.OWLInverseOf,'at');
	am.inverse <- thisModule.
	resolveTemp(p.OWLInverseOf, 'am');
	am.domain <- thisModule.
	resolveTemp(domain, 'class');
	am.range <- thisModule.
	resolveTemp(range, 'class');
	if (op.oclIsKindOf(
	OWL!ReflexiveProperty))
	am.superclasses <-
	assoc.superclasses->
	including(Synthesis!MetaclassDef.
	allInstances()->any(m |
	m.name = 'Reflexive'));
	} }
	
	helper context OWL!RDFSResource
	def: resourceName(): String =
	let r : OWL!LocalName =
	self.uriRef->any(e |
	e.oclIsKindOf(OWL!URIReference)).
	fragmentIdentifier
	in if not r.oclIsUndefined()
	then r.name **else** 'AResource' **endif**;

2.6 Автоматизация построения трансформаций моделей

Раздел основан на работах автора [64] [104].

В данном разделе рассматриваются методы автоматизированного построения трансформаций моделей данных на основе средств движимой моделями инженерии (MDE) [193]. Результатом применения методов являются декларативно-императивные трансформации, реализующие отображения произвольных исходных моделей уровня M1, соответствующих исходной метамодели уровня M2, в целевые модели уровня M1, конформные целевой метамодели уровня M2.

Задача автоматизации процесса построения трансформаций является весьма актуальной, принимая во внимание трудоемкость процесса и многообразие моделей данных.

Рассматриваются два аспекта автоматизации построения трансформаций:

– построение *прямых* трансформаций на основе установленных соответствий между элементами моделей;

– построение *обратных* трансформаций моделей на основе прямых трансформаций.

Соответствия между элементами моделей могут быть установлены различными способами. Например, элементы моделей могут быть аннотированы описаниями на естественном языке. При этом соответствие между элементами устанавливается (автоматическим или полуавтоматическим способом) на основании сходства этих описаний. Соответствие элементов моделей может быть установлено на основании анализа структуры моделей, с применением алгоритмов сопоставления схем (schema matching [221]). Элементы моделей также могут быть аннотированы понятиями некоторой онтологии [222]. При этом соответствие между элементами устанавливается, если понятия, которыми аннотированы элементы, связаны отношением понятие-подпонятие. Для рассматриваемых в данном разделе методов механизм установления соответствий между понятиями не существенен.

Первый метод нацелен на технику декларативно-императивного программирования отображения моделей (конструирование составляющих его правил) на основании установленных соответствий между элементами моделей в предположении, что проблемы семантики отображений разрешены.

Целью второго метода, является построение обратных трансформаций моделей на основе прямых трансформаций. Рассмотрим две модели данных - *S* и *T*. Предположим, что создана трансформация *S2T* модели *S* в модель *T* (с применением упомянутого выше

метода построения трансформаций на основе соответствий между элементами моделей или полностью вручную). Такую трансформацию назовем *прямой*, модель *S* – исходной, модель *T* – целевой. Задача состоит в автоматизированном построении трансформации *T2S* модели *T* в модель *S*, называемой *обратной*, на основе спецификации трансформации *S2T*. Метод построения обратных трансформаций, как и метод построения трансформаций на основе соответствий элементов моделей, позволяет лишь частично автоматизировать процесс построения трансформаций. Конструкции языков трансформации моделей различаются по сложности и семантике: для одних возможно автоматически построить соответствующую конструкцию обратной трансформации, для других разработать такую конструкцию должен эксперт на основании семантики моделей.

Методы построения обратных трансформаций могут быть использованы при интеграции источников данных в предметных посредниках в различных целях, например:

– построение реверсивных отображений онтологических моделей для регистрации источников данных в посредниках [223];

– доказательство эквивалентности информационных моделей (модель *S эквивалентна* модели *T,* если *S* уточняет *T* и *T* уточняет *S*; для доказательства двухстороннего уточнения моделей необходимы прямое и обратное отображения);

– построение отображения канонической модели посредника M_1 в каноническую модель посредника M_2 для регистрации M_2 как источника в M_1;

– построение отображения канонической модели посредника в некоторый прагматический язык, например, UML.

В качестве языка трансформации моделей рассматривается язык ATL [192]. В качестве модели уровня M3 рассматривается модель *Ecore* [196].

Представленные методы могут быть адаптированы для построения трансформаций, выраженных на сходных с ATL языках (например, QVT или его диалектах). Конкретная модель уровня M3 также не является существенной, могут быть использованы и другие метаметамодели, например, MOF [23] или KM3 [224]. Методы также могут быть использованы независимо от Унификатора для неверифицируемого построения прямых и обратных трансформаций моделей данных.

Методы иллюстрируются на примерах построения трансформаций между языками OWL [11] и СИНТЕЗ [42], а также между языками RIF-FLD [144] и СИНТЕЗ. Формальные синтаксисы этих языков были представлены в виде моделей уровня M2,

конформных метамодели *Ecore* (раздел 2.2). Ниже эти М2-модели обозначаются *OWL* и *Synthesis* соответственно.

2.6.1 Построение трансформаций моделей на основе соответствий между элементами моделей

Целью предлагаемого метода является автоматическое построение как можно более полной трансформации исходной модели в целевую на основе множества *R* соответствий между элементами моделей.

Технически 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 *OWLOntology2ModuleDef*

```
rule OWLOntology2ModuleDef{
  from o: OWL!OWLOntology
  to m: Synthesis!ModuleDef(
    name <- o.uriRef.name
  )
}
```

входным элементом является элемент типа *OWLOntology,* порождается один элемент типа *ModuleDef*, и имя модуля формируется из имени онтологии, указанного в ее уникальном идентификаторе ресурса (*uriRef*).

Основным содержанием метода является набор *порождающих правил*, обеспечивающих автоматическое построение ATL-модуля трансформации заданной исходной модели *S* в заданную целевую модель *T,* включающих*:*

– порождающее правило для построения заголовка трансформации;

– порождающие правила для построения сигнатуры сопоставляющих правил ATL, составляющих трансформацию (сигнатура включает имя, исходный и целевые элементы правила);

– порождающие правила для построения инициализации целевых элементов в правилах трансформации.

Одному элементу исходной модели может соответствовать несколько элементов целевой модели и наоборот. В разных случаях требуются разные порождающие правила построения сигнатур и инициализаций.

Рассмотрим правила построения трансформаций моделей данных на основании соответствий элементов моделей (на примере отображения языка OWL в язык СИНТЕЗ).

Построение заголовка трансформации. Трансформация OWL в язык СИНТЕЗ представляется модулем языка ATL и называется *OWL2Synthesis*. Заголовок трансформации выглядит следующим образом:

```
module OWL2Synthesis;
create OUT: Synthesis from IN: OWL;
```

Заголовок говорит о том, что трансформация получает на вход модель, конформную модели *OWL*, и порождает модель, конформную модели *Synthesis*.

Построение сигнатур правил трансформации. Для каждого элемента A типа *EClass* из области определения $domain(R) \subseteq E_S$ отношения соответствия элементов R создается правило установления соответствия трансформации. Если элементу A в E_F соответствует ровно один элемент B, то правило называется *A2B:*

```
rule A2B{
  from a: OWL!A
  to b: Synthesis!B
}
```

Если элементу A соответствуют несколько элементов $B_1, \ldots, B_n$, то правило называется *A:*

```
rule A{
  from a: OWL!A
  to b1: Synthesis!B1,
    b2: Fund!B2,
    …
    bn: Fund!Bn
}
```

Примером является правило *OWLClass*:

```
rule OWLClass{
  from c: OWL!OWLClass
  to
    type: Synthesis!ADTDef,
    class: Synthesis!ClassDef
}
```

Правило говорит о том, что на основании элемента типа *OWLClass* будут созданы два элемента - типов *ADTDef* и *ClassDef* соответственно.

Инициализация свойств целевого элемента в правилах. Для каждого построенного правила *A* (*A2B*), для каждого элемента B_i (соответствующего *A*), для каждого свойства (атрибута или связи) *f* элемента B_i производится поиск соответствующего свойства элемента *A*. Затем, на основании найденного соответствия, в правило добавляется инициализация свойства *f*.

Порождающее правило для инициализации свойств примитивных типов. В случае, если свойство имеет примитивный тип (Boolean, Integer, String и т.д.), инициализация свойства производится в разделе *to* правила (называемом целевым образцом – *target pattern* – правила).

Примером такого свойства является *SchemaDef.name,* участвующее в соответствии *Graph.ontology->uriRef.name ~ SchemaDef.name.* Инициализация этого свойства в правиле *Graph2SchemaDef* выглядит следующим образом:

```
rule Graph2SchemaDef{
  from g: OWL!Graph
  to s: Synthesis!SchemaDef(
    name <- g.ontology->any().uriRef.name
  )
}
```

Свойство *name* элемента *SchemaDef* инициализируется выражением, построенным на основании пути в структуре элемента *Graph*, указанном в соответствии. Точечная нотация в структуре пути остается без изменений, для навигации по связи один-ко-многим необходимо применить операцию *any* выбора произвольного экземпляра из коллекции. В данном случае имя схемы инициализируется одним из имен онтологий графа.

Часто подобная инициализация не является окончательной, и эксперт способен найти семантически более корректный способ инициализации. Например, в данном случае, имя схемы можно образовать, соединив все имена онтологий в одну строку и добавив к ней слово *Schema:*

```
name <- g.ontology->collect(e| e.uriRef->any().name->sum()+'Schema'
```

Имена собираются из онтологий при помощи операции *collect*, и затем соединяются при помощи операции *sum* (семантика этих операций в ATL совпадает с семантикой одноименных операций OCL).

Тем не менее, автоматически построенная инициализация является основой для дальнейшей работы эксперта.

Порождающее правило для инициализации свойства, тип которого является единственным образом своего прообраза в отношении соответствия R. Инициализация свойства в целевом образце возможна также, если тип свойства является единственным образом своего прообраза в отношении соответствия *R*. Примером такого свойства является *SchemaDef.modules,* участвующее в соответствии *Graph.ontology ~ SchemaDef.modules*. Свойство *modules* является связью один-ко-многим между элементами *SchemaDef* и *ModuleDef*, т.е. тип свойства *modules – ModuleDef*. Прообразом элемента *ModuleDef* в отношении *R* является элемент *OWLOntology*. Единственным образом в отношении *R* у этого элемента является *ModuleDef*. Инициализация этого свойства в правиле *Graph2Schema* выглядит следующим образом:

```
rule Graph2Schema{
  from g: OWL!OWLGraph
  to s: Synthesis!SchemaDef(
    modules <- g.ontology
  )
}
```

Такая инициализация предполагает, что для экземпляров коллекции *g.ontology* (которые являются экземплярами класса *OWLOntology*) существует сопоставляющее правило, которое преобразует их в экземпляры коллекции *s.modules* (которые являются экземплярами класса *ModuleDef*). Такое правило будет неявно вызываться для каждого экземпляра коллекции *g.ontology* (в соответствии с семантикой языка ATL). Действительно, согласно приведенным выше правилам, транслятор включает необходимое правило *OWLOntology2ModuleDef*.

Порождающие правила для инициализации свойства, тип которого является не единственным образом своего прообраза в отношении соответствия R. Рассмотрим теперь свойства, типы которых являются не единственными образами своих прообразов в *R*. Правила преобразования типов этих свойств порождают несколько целевых элементов. Поэтому, в общем случае, необходимо явно указать, какой же именно целевой элемент будет использован для инициализации свойства.

Для такого рода инициализаций используется встроенная функция *thisModule.resolveTemp(var, target-pattern-name)*. Идентификатор *thisModule* говорит о том, что *resolveTemp* является функцией модуля трансформации. Первым параметром функции является переменная, содержащая элемент исходной модели, из которого

должен быть получен искомый целевой элемент. Вторым параметром является строка с именем искомого целевого элемента. В языке ATL использование функции *thisModule.resolveTemp()* возможно только в императивном разделе *do* правила. Этот раздел располагается в правилах после раздела *to*; в нем при помощи императивных операторов инициализируются те свойства, которые не инициализированы декларативно в разделе *to*.

Примером свойства, тип которого есть не единственный образ своего прообраза в *R,* является *ModuleDef.containedTypes*. Свойство является связью один-ко-многим. Его тип *ADTDef* участвует в сопоставлении *ADTDef ~ OWLClass*. Однако, *OWLClass* участвует также в сопоставлении *OWLClass ~ ClassDef*. Поэтому инициализация свойства *containedTypes* в правиле *OWLOntology2ModuleDef* выглядит следующим образом:

```
rule OWLOntology2ModuleDef{
  from o: OWL!OWLOntology
  to m: Synthesis!ModuleDef
  do{
    m.containedTypes <- o.owlUniverse->
     select(e|e.oclIsTypeOf(OWL!OWLClass))->
     collect(e|thisModule.resolveTemp(e, 'type'));
  }
}
```

Преобразование экземпляров коллекции *o.owlUniverse* в экземпляры коллекции *m.containedTypes* производится при помощи операции *collect,* вызывающей для каждого экземпляра функцию *resolveTemp*. Второй аргумент функции указывает, что для преобразования следует использовать целевой элемент *type* правила *OWLClass* (раздел 2.2.2).

В инициализации используется также операция *select* (семантика которой совпадает с семантикой одноименной операции OCL), выбирающая из коллекции для последующего преобразования только элементы класса *OWLClass*. Это делается потому, что тип коллекции *o.owlUniverse* является суперклассом *OWLClass*, и в коллекции могут содержаться элементы, имеющие отличный от *OWLClass* тип. Такое появление операции *select* при инициализации является примером применения вспомогательного *Порождающего правила выборки подходящих элементов коллекции при инициализации свойства*.

Для свойств, являющихся связью один-к-одному, порождающее правило инициализации при помощи *resolveTemp* выглядит более просто. Так, для свойства *AssociationMetaclass.inverse* инициализация выглядит следующим образом:

```
rule OWLObjectProperty{
```

```
  from p: OWL!OWLObjectProperty
  to
    assoc: Synthesis!AssociationMetaclassDef
  do{
    assoc.inverse <-
      thisModule.resolveTemp(
        p.OWLInverseOf, 'assoc');
  }
}
```

Применения операций *select* и *collect* в данном случае не требуется, достаточно вызова функции *resolveTemp*, обращающейся к правилу преобразования элемента *OWLObjectProperty*:

```
rule OWLObjectProperty{
  from p: OWL!OWLObjectProperty
  to a: Synthesis!AttributeDef(…),
  assoc: Synthesis!AssociationMetaclassDef(…)
  do{…}
}
```

Следующие два правила иллюстрируются на примере трансформации модели Synthesis в каркас логических диалектов RIF-FLD[44].

Порождающее правило для инициализации путей целевой модели. Рассмотрим соответствие свойств, в правой части которого находится свойство, выраженное путем в структуре целевой модели:

```
ModuleDef.containedPrograms ~ Document.group->contents
```

Для реализации этого соответствия в ATL-правиле необходимо включить в целевой образец, кроме элемента типа Document, элемент типа Group (тип промежуточного свойства group). В результате, ATL-правило выглядит следующим образом:

```
rule Module2Document{
  from m: Synthesis!ModuleDef
  to d: RIFFLD!Document(group <- grp),
  grp: RIFFLD!Group(contents <- m.containedPrograms)
}
```

[44] Модель абстрактного синтаксиса каркаса логических диалектов RIF-FLD , включающая используемые в примерах сущности Document, Group, размещена в репозитории GitHub: https://github.com/sstupnikov/ModelTransformation/blob/master/RIF_FLD/metamodel/RIF-FLD.ecore. Принципы взаимного отображения между канонической моделью (языком СИНТЕЗ) и языком RIF-FLD изложены в приложении.

Порождающее правило для построения условия на исходный элемент. Рассмотрим соответствие свойств, в левой части которого находится свойство, выраженное путем в структуре исходной модели:

```
LogicProgram.program->rules ~ Group->content
```

Заметим, что тип *Program* свойства *program* не имеет свойства *rules*, но свойство *rules* имеет его подтип *RuleList*. Поэтому на исходный элемент типа *LogicProgram* необходимо наложить ограничение:

```
rule LogicProgram2Group{
from lp: Synthesis!LogicProgram(
not lp.program.oclIsUndefined()
and lp.program.oclIsKindOf(Synthesis!RuleList) )
  to g: RIFFLD!Group( content <- lp.program.rules )
}
```

Вышеизложенные правила были реализованы в виде высокоуровневой трансформации[45] TTL (Transformation Template Constructor) на языке ATL. Трансформация принимает на вход Ecore-модель, конформную метамодели *Similarities*[46] (отвечающей соответствиям между элементами моделей) и генерирует Ecore-модель, конформную метамодели ATL[47], которая, в свою очередь, сериализуется в виде текстового файла на языке ATL. Таким образом, обеспечивается автоматическая генерация шаблонов трансформаций моделей данных.

Заметим, что соответствия между элементами моделей всегда содержат лишь часть семантики отображения моделей. Например, элементы могут находиться в соответствии, а отображение при этом зависит от некоторого сложного условия. Недостающая семантика должна быть добавлена в трансформацию вручную экспертом. Таким образом, сценарий построения трансформации состоит из двух шагов:

– автоматического построения шаблона трансформации на основе соответствий между элементами моделей;

---

[45] https://github.com/sstupnikov/ModelUnifier.ATL/blob/master/transformations/TTC.atl

[46] https://github.com/sstupnikov/ModelUnifier.ATL/blob/master/model/Similarities.ecore

[47] https://github.com/sstupnikov/ModelUnifier.ATL/blob/master/model/ATL.ecore

– превращения экспертом образца в полноценную трансформацию на основании семантики, не содержащейся во множестве соответствий.

Заметим также, что в процессе унификации моделей может потребоваться расширение целевой (канонической) модели. Список соответствий между элементами моделей при этом пополняется соответствиями между элементами исходной модели и элементами расширения. Предлагаемый метод построения трансформаций при этом должен быть применен к полному списку соответствий.

2.6.2 Построение обратных трансформаций моделей на основе прямых трансформаций моделей

Рассмотрим трансформацию t модели U в модель V и трансформацию r модели V в модель U. Обозначим через $M(U)$ множество моделей, конформных модели U.

Определение. Назовем *r обратной* трансформации t на множестве $N \subseteq M(U)$, если для любой модели s из N выполнено $r(t(s)) = s$. Назовем *r всюду обратной* для t, если r является обратной для t на $M(U)$.

Определение. Назовем *r слабо обратной* трансформации t на множестве $N \subseteq M(U)$, если для любой модели s из N выполнено $t(r(t(s))) = t(s)$. Назовем *r всюду слабо обратной* для t, если r является слабо обратной для t на $M(U)$.

Обратную трансформацию r возможно построить только для такой прямой трансформации t, которая передает всю информацию, содержащуюся в исходной модели. Слабо обратная трансформация восстанавливает всю информацию, передаваемую прямой трансформацией, при этом прямая трансформация может передавать только часть информации, содержащейся в исходной модели. Очевидно, что обратная трансформация на множестве является слабо обратной на том же множестве. Обратную трансформацию возможно построить в том случае, когда *в каждой инициализации каждого правила атрибут целевой модели связывается ровно с одним атрибутом исходной модели.* В некоторых случаях возможно ослабление данного условия.

Обратная трансформация является полной и правильной, не требующей ручной модификации. Слабо обратная трансформация является образцом, который предлагается эксперту для проверки и модификации в случае необходимости.

Определение. Назовем подмножество L трансформаций, выраженных на некотором языке трансформации моделей (например, ATL), *(слабо) обратимым*, если существует алгоритм, конструирующий для любой трансформации t из L всюду (слабо) обратную ей трансформацию r.

Определение. Назовем подмножество *K* конструкций (операторов, функций, операций) языка трансформации моделей (например, ATL) *обратимым (слабо обратимым)*, если множество трансформаций, построенных на основе этих конструкций, является обратимым (слабо обратимым).

Возможно, что слабо обратимым является каждый язык трансформации моделей (и ATL, в частности) целиком. Однако, слабое обращение имеет смысл не для всех конструкций языка.

Задача автоматизации построения обратных трансформаций сводится к следующим подзадачам:

– нахождение максимального (слабо) обратимого подмножества K_R конструкций языка трансформации моделей;

– разработка алгоритма построения обратного отображения для прямого отображения, построенного на основе K_R.

Обозначим семантическую функцию, преобразующую конструкции прямой трансформации в конструкции обратной трансформации, через *r[]*. Будем также называть преобразование конструкций прямой трансформации в конструкции обратной трансформации *обращением*.

Метод построения обратных трансформаций включает *правила обращения* следующих конструкций языка ATL:

– заголовок отображения;

– вспомогательные правила (helpers), являющиеся ATL-эквивалентом методов языка Java;

– сопоставляющие правила;

– локальные переменные правил;

– инициализация целевых элементов в декларативной и императивной частях правил;

– условный оператор *if-then* и оператор присваивания в инициализациях;

– выражения путей в элементах исходной модели;

– OCL-выражения *if-then* и *let*;

– операции примитивных типов OCL: логическое отрицание; сложение, вычитание, операция *toString* для числовых типов; умножение, деление, операции *sqrt, exp, log, toDegrees, toRadians* типа *Real*; операции *toInteger, toReal, toSequence, split, concat* (в частном случае) типа *String*;

– операция *toSequence* OCL-типов *Set, OrderedSet, Bag*; операция *toSet* OCL-типов *OrderedSet, Bag*;

– операции *append, prepend, insertAt* OCL-типа *Sequence*;

– операции над коллекциями OCL *any* и *select*.

Для некоторых конструкций возможно лишь слабое обращение (условный оператор, операции *any* и *select*).

В целом, построение обратной трансформации состоит из двух последовательных шагов:

– автоматическое построение обратной трансформации на основе прямой трансформации. Каждая слабо обратимая конструкция прямой трансформации обращается, остальные конструкции помечаются;

– ручная модификация автоматически построенной трансформации экспертом. Обратимые конструкции не требуют модификации. Обращение слабо обратимых конструкций проверяется и модифицируется, если это необходимо. Помеченные конструкции обращаются вручную.

Ниже рассмотрены и проиллюстрированы правила обращения.

Обращение заголовка трансформации. Заголовок прямой трансформации исходной модели *S* в целевую модель *T* преобразуется в заголовок обратной трансформации следующим образом:

```
r[  module S2T
    create OUT: T from IN: S
] =
module T2S;
create OUT: S from IN: T;
```

Обращение правил трансформации. Рассмотрим правило установления соответствия *U2V* общего вида, преобразующее элемент *U* модели *S* в элемент *V* модели *T*:

```
rule U2V{
  from u: S!U
  to v: T!V
  do{ statements }
}
```

Заметим, что в таком правиле инициализация свойств целевого элемента производится не в разделе *to* образца целевого элемента, а в императивном разделе *do.* Этот раздел состоит из последовательности *statements* императивных операторов,

соединенных секвенциальным оператором (;). Без ограничения общности можно рассуждать об обращении правил именно такого вида. Действительно, правило, содержащее инициализацию свойства в разделе *to* можно преобразовать в эквивалентное правило, содержащее инициализацию только в разделе *do.* Так, правило

```
rule OWLOntology2ModuleDef{
  from o: OWL!OWLOntology
  to s: Synthesis!ModuleDef(
    name <- o.uriRef.name
  )
}
```

эквивалентным образом преобразуется в правило

```
rule OWLOntology2ModuleDef{
  from o: OWL!OWLOntology
  to m: Synthesis!ModuleDef
  do{ m.name <- o. uriRef.name; }
}
```

Итак, сопоставляющее правило обращается следующим образом:

```
r[  rule U2V{
      from u: S!U
      to v: T!V
      do{ statements }
}
] =
rule V2U{
  from v: T!V
  to u: S!U
  do{ r[statements] }
}
```

Каждый оператор из секции *do* прямого правила преобразуется в один или несколько операторов секции *do* обратного правила.

Обращение оператора присваивания. Оператор присваивания в правилах имеет следующий вид:

```
rule U2V{
  from u: S!U
  to v: T!V
  do{ t(v) <- s(u) }
}
```

Здесь *t(v)* – некоторое свойство элемента целевой модели, *s(u)* – выражение языка ATL (являющееся декларативным выражением OCL). Рассматриваются такие присваивания, в которых *s* зависит от исходной переменной *u* (именно в них происходит связывание элементов исходной и целевой моделей).

Обращение присваивания (действие функции *r* на присваивании) задается следующим образом. К присваиванию применяются *правила обращения присваиваний.*

Каждое правило обращения преобразует присваивание таким образом, что правая часть присваивания упрощается, а левая – усложняется. Правило имеет вид $b_1 \Rightarrow b_2$, где b_1 и b_2 – присваивания. Правила применяются до тех пор, пока правая часть не превратится в выражение вида *u.a*, где *u* – имя целевой переменной, *a* – атрибут целевого элемента. Если в процессе обращения обнаружилось, что к присваиванию не применимо ни одно правило, а правая часть представляет собой выражение, отличное от *u.a*, то присваивание считается *необратимым* (автоматически необратимым). Обращение такого присваивания производится экспертом вручную, если обращение вообще возможно. Правило обращения может разбить присваивание на несколько присваиваний. При этом в дальнейшем правила обращения применяются к каждому из полученных присваиваний до максимального упрощения их правых частей.

Пусть присваивание *t(v) <- s(u)* превратилось в процессе применения правил обращения в набор присваиваний

t_1 <- u.a_1; … , t_n <- u.a_n;

где t_i – выражения, a_i – имена свойств. Тогда действие функции *r* на присваивании *t(v) <- s(u)* определяется следующим образом:

r[t(v) <- s(u)] = u.a_1 <- t_1, … , u.a_n <- t_n

Рассмотрим различные правила обращения присваиваний в контексте правила *U2V*.

В случае, если присваивание представляет собой *простое соответствие свойств*, не требуется применение никаких правил обращения присваиваний, преобразование следует из определения функции *r*:

```
r[v.b <- u.a] = u.a <- v.b
```

Обращение путей в структуре исходного элемента. Рассмотрим правило обращения *путей* в структуре исходного элемента. Рассмотрим присваивание

```
t <- s.d.b
```

где *s* – выражение типа T_s; *d* – атрибут типа T_s , имеющий тип T_d; *b* – атрибут типа T_d, имеющий тип T_b. Заметим, что при использовании таких выражений пути происходит уплощение структуры исходного типа. Отсюда следует, что в обратном преобразовании следует восстановить структуру. В данном случае это означает создание экземпляра типа T_d:

```
t <- s.d.b ⇒ t <- w.b; w <- s.d
```

Здесь *w* – новая переменная типа T_d, которая становится дополнительным элементом целевого образца (target pattern) обратного правила. Тем самым правило, кроме экземпляра типа *U*, будет порождать экземпляр типа T_d:

```
rule V2U{
  from v: T!V
  to u: S!U, w: T!Td
  do{ … }
}
```

Первое из полученных присваиваний (*t* <- *w.b*) относится к элементу *w* целевого образца, второе – к элементу *u* целевого образца.

Рассмотрим пример использования обращения путей. Прямое правило *OWLOntology2ModuleDef* преобразуется в обратное правило *ModuleDef2OWLOntology*:

```
rule OWLOntology2ModuleDef{
  from o: OWL!OWLOntology
  to m: Synthesis!ModuleDef(
    name <- o.uriRef.name
  )
}
rule ModuleDef2OWLOntology{
  from m: Synthesis!ModuleDef
  to o: OWL!OWLOntology,
    u: OWL!URIReference
  do{ o.uriRef <- u; u.name <- m.name; }
}
```

Условие обратимости присваиваний. Вообще, обращение (автоматическое) присваивания возможно, если в *правой части присваивания используется ровно одно свойство (атрибут или связь) исходной модели*. Если свойству целевой модели присваивается выражение, которое включает несколько свойств исходной модели, то обращение такого связывания невозможно (из-за неоднозначности). При таких отображениях в общем случае теряется информация о том, какие значения имели исходные свойства. В целевом свойстве сохраняется лишь некоторая агрегатная информация об этих значениях.

Обращение операций примитивных типов. В ряде случаев возможно автоматическое обращение присваиваний, в которых OCL-выражения содержат операции над примитивными типами. Ниже рассматриваются основные обратимые операции. Пусть *c* обозначает константу некоторого примитивного типа (*Boolean,*

Integer, Real), *t* и *s* – выражения, включающие атрибуты целевого и исходного элемента соответственно.

Возможно, например, обращение логического отрицания. Правило обращения присваивания для операции отрицания выглядит следующим образом:

```
t <- not s ⇒ not t <- s
```

При использовании остальных логических операций (*and, or, implies*) в присваиваниях возможна потеря информации, а потому обращение невозможно. Слабое обращение логических операций возможно, но семантически некорректно.

Для числовых типов (*Integer, Real*) возможно обращение операций сложения и вычитания:

```
t <- s - c ⇒ t + c <- s
t <- c - s ⇒ c - t <- s
t <- s + c ⇒ t - c <- s
```

Для типа *Real* возможно обращение операций деления и умножения:

```
t <- s/c ⇒ t*c <- s
t <- c/s ⇒ c/t <- s
t <- s*c ⇒ t/c <- s
```

Для типа *String* возможно обращение операции преобразования к типу коллекции:

```
t <- s->toSequence() ⇒ t->sum() <- s
```

Строка преобразуется в последовательность строк, состоящих из одного символа, в обратном преобразовании все строки из последовательности объединяются в одну.

Обращение операций над коллекциями. В языке OCL определены различные типы коллекций: *Set, OrderedSet, Bag, Sequence*. В данном разделе рассматриваются правила обращения присваиваний, содержащих основные операция, определенные над типами коллекций. Семантика всех операций над коллекциями, используемых в данном разделе, и оператора *let* совпадает с соответствующей семантикой в языке OCL.

Пусть *e, elm, n, cond* – выражения, не зависящие от атрибутов исходной модели, *t* и *s* – выражения, включающие атрибуты целевого и исходного элемента соответственно.

Для типа *Sequence* возможно обращение операции *append* присоединения элемента в конец последовательности:

```
t <- s->append(elm) ⇒
   let seq: Sequence = t in seq->subSequence(1, seq->size()-1) <- s
t <- e->append(s) ⇒ t->last() <- s
```

операции *prepend* присоединения элемента в начало последовательности:

```
t <- s->prepend(elm) ⇒
  let seq: Sequence = t in seq->subSequence(2, seq->size()) <- s
t <- e->prepend(s) ⇒ t->first() <- s
```

операции *insertAt* вставки элемента в коллекцию на место с некоторым номером *n*:

```
t <- e->insertAt(n, s) ⇒ t->at(n) <- s
t <- s->insertAt(n, e) ⇒
  let seq: Sequence = t in
  let head: Sequence = seq->subSequence(1, n-1) in
  let tail: Sequence = seq->subsequence(n+1, seq->size()) in
  tail.iterate(elm, acc: Sequence = head |  acc.append(elm))
  <- s
```

Операции *any* выбора произвольного элемента коллекции и *select* выбора подмножества элементов коллекции по условию не являются обратимыми, однако допустимым является вариант слабого их обращения:

```
r[t <- u.c->any(e | cond)] = u.c <- u.c->including(t)
r[t <- u.c->select(e | cond)] = u.c <- u.c->union(t)
```

Здесь *u* – исходная переменная, *c* – атрибут исходного элемента. Тип атрибута *c* является типом коллекции.

Вышеизложенные правила были реализованы в виде высокоуровневой трансформации ATLReverse на языке ATL. Трансформация принимает на вход Ecore-модель (конформную метамодели ATL), рассматриваемую как прямая трансформация, и автоматически генерирует Ecore-модель, конформную метамодели ATL, которая, в свою очередь, сериализуется в виде текстового файла на языке ATL и рассматривается как шаблон обратной трансформации.

2.7 Формализация семантики канонической объектной модели данных

Раздел основан на работах автора [46] [69] [110] [107] [101].

В данном разделе рассматривается метод формализации семантики канонической модели данных в языке AMN. Фактически, производится определение отображения Θ абстрактного синтаксиса подмножества ядра канонической модели (языка СИНТЕЗ) в спецификации AMN. Отображение задается при помощи семантических функций, определенных на элементах синтаксиса канонической модели, а также при помощи алгоритмов.

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 $M \in Module$ канонической модели, $\Theta(M) = \{M_{ctxt}\} \cup M_{types}$, где M_{ctxt} – контекстная машина, M_{types} – множество машин, соответствующих АТД. В нижеследующих разделах излагается, каким образом по спецификации M строятся M_{ctxt} и M_{types}.

2.7.1 Основные принципы определения семантики канонических моделей данных

Определение семантики АТД в AMN базируется на экстенсиональном принципе: тип моделируется константным множеством своих экземпляров. Такое множество экземпляров называется *экстенсионалом* типа. 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.

Потенциальные экземпляры всех абстрактных типов моделируются предопределенным множеством $AVAL$ и для каждого типа T с экстенсионалом E_T выполнено $E_T \subseteq AVAL$. Экстенсионалы типов соотносятся в соответствии с отношением тип-подтип; для каждой пары типов T_1, T_2, в которой T_2 является подтипом T_1, выполнено отношение включения на экстенсионалах $E_{T2} \subset E_{T1}$. Таким образом моделируется иерархия типов.

Представление типа множеством своих потенциальных экземпляров позволяет естественным образом определить семантику спецификации типа набором функций AMN: атрибут a типа T моделируется функцией $a \in E_T \rightarrow E_{Ta}$, где T_a – 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 $Obj \subseteq AVAL$. Для обеспечения уникальной идентификации объектов заводится множество объектных идентификаторов OID. Биекция $self \in Obj \rightarrow OID$ моделирует неявный атрибут *self* каждого объектного типа.

Спецификации типов канонической модели могут ссылаться друг на друга в спецификациях атрибутов, методов, инвариантов, образуя *структуру спецификаций типов*. Определение семантики структуры спецификаций типов осуществляется при помощи средств композиции абстрактных машин AMN. Основным принципом отображения структуры спецификаций типов канонической модели в AMN является стремление отобразить каждый АТД отдельной абстрактной машиной. Проблема состоит в том, что средства композиции абстрактных машин являются гораздо менее гибкими, чем средства организации структуры спецификаций типов канонической модели. Данное противоречие разрешается при помощи процедуры преобразования ориентированного графа структуры модуля *M*, представляющего композиционную структуру спецификаций типов модуля, — *DSG*(*M*), в ориентированный граф, представляющий композиционную структуру набора абстрактных машин, — *DAMSG*(*M*).

Определение. Ориентированный композиционный граф *DSG*(*M*) **модуля** *M, есть пара* ⟨*N, E*⟩*, где множеством вершин графа N является множество Types*(*M*) *АТД, определенных в секции* type *модуле M; множеством направленных ребер графа E является множество упорядоченных пар* ⟨T_1, T_2⟩*, где* T_1, $T_1 \in Types\ (M)$*, и спецификации инвариантов и (или) методов* T_1 *содержат ссылки на атрибуты, методы, инварианты типа* T_2*, либо ссылки на коллекции с типом экземпляров* T_2.

Определение. Ориентированный граф структуры абстрактных машин *DAMSG*(*M*) **для модуля** *M, есть пара* ⟨*N, E*⟩*, где множеством вершин графа N является множество подмножеств Types*(*M*) *такое, что*

$$\forall\ s_1,\ s_2 \bullet (s_1 \in N \wedge s_2 \in N \Rightarrow s_1 \cap s_2 = \emptyset) \wedge \cup_{s \in N}\ s = Types\ (M)$$

Это означает, что множества спецификаций типов из N являются непересекающимися и объединение их образует множество всех спецификаций типов Types(*M*). *Множеством направленных раскрашенных ребер графа E является множество упорядоченных троек* ⟨s_1, s_2, *c*⟩*, где* s_1, $s_2 \in N$,

$$c \in \{sees,\ uses,\ includes,\ extends\}.$$

Каждая из вершин s графа соответствует абстрактной машине, в которую отображается весь набор типов из s. Ребро между вершинами графа означает отношение композиции абстрактных машин, цвет ребра означает вид композиции.

DAMSG строится из *DSG* следующими последовательными преобразованиями:

– *Раскраска.* Ребра графа раскрашиваются по следующим правилам:

1) ребро $\langle s_i, s_j \rangle$ раскрашивается в цвет *sees*, если методы типов из s_i ссылаются на атрибуты, методы типов из s_j , либо на коллекции с типами экземпляров из s_j ;

2) ребро $\langle s_i , s_j \rangle$ раскрашивается в цвет *uses*, если инварианты типов из s_i ссылаются на атрибуты, методы типов из s_j , либо на коллекции с типами экземпляров из s_j ;

3) ребро $\langle s_i, s_j \rangle$ раскрашивается в цвет *includes*, если методы типов из s_i изменяют атрибуты типов из s_j ;

4) ребро $\langle s_i , s_j \rangle$ раскрашивается в цвет *extends*, если среди типов из s_j есть непосредственные супертипы типов из s_i.

– *Элиминация циклов.* В языке AMN не разрешаются циклы в графе композиционной иерархии абстрактных машин. Поэтому циклы в графе элиминируются при помощи следующей процедуры. Последовательно перебираются все циклы *DCG* , множество вершин цикла сливается в одну вершину. При слиянии двух ребер e_1 и e_2, результирующее ребро e красится в более яркий цвет из цветов ребер e_1 и e_2. Шкала яркости цветов от менее яркого к более яркому выглядит так: *sees, uses, includes, extends*.

– *Элиминация дублирующих includes/extends путей.* В языке AMN не разрешается одной машине быть дважды включенной в другую машину. Ввиду этого граф композиционной структуры абстрактных машин должен обладать следующим свойством: между любыми двумя вершинами не может быть более одного пути, ребра которого раскрашены в цвет *includes* либо в цвет *extends*. Процедура элиминации дублирующих *includes*-путей состоит в следующем. Для любой дублирующей пары *includes*-путей

$$P_1 = s_0 \to s_{i1} \to \cdots \to s_{in} \to s_1,\ P_2 = s_0 \to s_{j1} \to \cdots \to s_{jm} \to s_1$$

вершины $s_0, s_{i1}, \ldots s_{in}, s_{j1}, \ldots s_{jm}$ сливаются в одну вершину.

Утверждение 3.1. *Ациклический граф после применения процедуры элиминации дублирующих includes/extends путей, остается ациклическим.*

Процедура элиминации дублирующих путей состоит из конечного числа слияния вершин, составляющих пару дублирующих путей, в одну вершину. Докажем, что после каждого слияния ацикличность сохраняется. Рассмотрим множество вершин

$$V = \{v_1, \ldots v_n, w_1, \ldots w_m\}$$

составляющих пару дублирующих путей $v_1 \ldots v_n$ и $w_1 \ldots w_m$. Обозначим через v вершину, полученную путем слияния вершин из V.

Пусть в графе, содержащем v, есть цикл, не проходящий через v. Тогда точно такой же цикл был бы в графе, содержащем V. Противоречие.

Пусть в графе, содержащем v, есть цикл $u_1,\dots, u_k$, проходящий через v, т.е. $v = u_i$. Тогда в графе, содержащем V , существовал цикл

$$u_1 \dots u_{i-1} v_1 \dots v_n u_{i+1} \dots u_k$$

Противоречие.

□

Граф *DAMSG*, не содержащий дублирующих *includes/extends* путей является основанием для формирования множества машин M_{types}. Вершина s_i графа *ADCG* отображается в отдельную абстрактную машину $M_i \in M_{types}$. Раскраска ребра графа индуцирует отношения композиции между машинами.

2.7.2 Структура контекстной машины

Контекстная машина M_{ctxt} 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 $T \in Types$ (M) вводится константное множество ext_T, моделирующее полный экстенсионал и константное множество $extp\_T$, моделирующее собственный экстенсионал. Иерархия типов выстраивается при помощи соотношений $ext\_T_2 \subset ext\_T_1$, записываемых в раздел PROPERTIES для каждой пары типов T_1, T_2, что T_2 является подтипом T_1.

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

$$extp\_T_1 \cap extp\_T_2$$

Здесь T_1, T_2 обозначают различные 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 $Obj \in \mathrm{POW}(AVAL)$, обозначающую множество, моделирующее совокупность экстенсионалов объектных типов;

– множество *OID* объектных идентификаторов;

– константу *self*, устанавливающую взаимооднозначное соответствие между экземплярами объектных типов и объектными идентификаторами и типизированную в разделе PROPERTIES как полная биекция $self \in Obj \rightarrow OID$;

– константы *ext_T, extp_T* для каждого объектного типа T, типизированные в разделе PROPERTIES как

$$ext\_T \in \mathrm{POW}(Obj) \wedge extp\_T \in \mathrm{POW}(Obj) \wedge extp\_T \subseteq ext\_T$$

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

2.7.3 Структура абстрактной машины, соответствующей типу: семантика атрибутов, методов, инвариантов

Ориентированный граф структуры абстрактных машин для модуля, *DAMSG*, отражает композиционную структуру абстрактных машин множества M_{types}. Каждая абстрактная машина из M_{types} формируется на основании спецификаций АТД, собранных в одну вершину графа *DAMSG*.

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

Для каждого *атрибута a*: T_a типа T машина M_T содержит переменную a в разделе VARIABLES. Переменная a типизируется в разделе INVARIANT выражением, формируемым семантической функцией *mAttributeInv* на основании имени типа и спецификации атрибута. Префикс *m* отличает все семантические функции отображения элементов синтаксиса канонической модели в AMN. В определении семантических функций в квадратные скобки заключены параметры – элементы синтаксиса, а в круглые скобки – вспомогательные параметры, U – идентификатор абстрактного типа данных:

```
mAttributeInv(T: IDENTIFIER)[a: Ta] = a ∈ ext_T →  mAttrType(a)[Ta]
mAttrType(A: IDENTIFIER)[Ta] = mType(A)[Ta]
mType(A: IDENTIFIER)[U] = ext_U
mType(A: IDENTIFIER)[integer] = INT
mBuiltInType(A: IDENTIFIER)[Boolean] = BOOL
mBuiltInType(A: IDENTIFIER)[string] = STRING_TYPE
mBuiltInType(A: IDENTIFIER)[{set; type_of_element: ST;}] = POW(mType[[ST]])
mBuiltInType(A: IDENTIFIER)[{enum; enum_list: { e1; … en }] = A_enum
```

Определение перечисления, формируемое функцией *mEnum*, определенной на домене *EnumDef*, включается в раздел SETS машины M_T :

```
mEnum(A: IDENTIFIER)[{enum; enum_list: { e1; … en }] =
  A_enum = { e1, … en }
```

Приведём примеры типизации атрибутов абстрактного и строкового типов:

```
mAttributeInv(Cultivar)[crop: Crop] = crop ∈ ext_Cultivar →  ext_Crop
mAttributeInv(Entity)[name: string] = name ∈ ext_Entity →  STRING_TYPE
```

Методы модификации атрибутов моделируются операциями *set* машины M_T в разделе OPERATIONS. Тела операций *set* формируются семантической функцией *mSetOperation*:

```
mSetOperation(T : IDENTIFIER)[a: Ta] =
set_a(s)=
PRE s ∈ ext_T → mAttrType[Ta]
THEN a := a <+ s
END
```

Параметр *s* представляет собой множество пар $o \mapsto v$, где o – значение типа T, v – значение, которое должно быть сообщено соответствующему атрибуту значения o функцией *set*.

Пример применения функции *mSetOperation* выглядит следующим образом:

```
mSetOperation(Cultivar)[crop: Crop] =
set_crop(s)=
PRE s ∈ ext_Cultivar → ext_Crop
THEN crop := crop <+ s
END
```

Инициализация атрибутов осуществляется при помощи подстановки ANY в разделе INITIALISATION. Конкретная подстановка инициализации атрибута формируется семантической функцией *mAttrInit*:

```
mAttrInit(T: IDENTIFIER)[a: Ta] =
ANY aa WHERE
  aa ∈ ext_T → mAttrType[Ta]
THEN
  a := aa
END
```

Пример применения функции *mAttrInit* выглядит следующим образом:

```
mAttrInit(Cultivar)[crop: Crop] =
ANY acrop WHERE
  acrop ∈ ext_Cultivar → ext_Crop
THEN
  crop := acrop
END
```

Инициализирующие подстановки атрибутов соединяются между собой оператором параллельной подстановки ||.

Инвариант, предикативная спецификация которого задается формулой f канонической модели, отображается конъюнктивным компонентом f_m инварианта машины M_T. Формула f_m есть формула AMN, формируемая семантической функцией *mPredicate*: f_m = *mPredicate*[f]. Данная функция формирует представление формул канонической модели предикатами AMN и описывается в следующем разделе.

Метод meth типа *T* представляется в AMN операцией *meth_op* машины M_T . Операцию машины формирует функция *mMethodSubst* на основании имени типа и спецификации метода. Рассмотрим действие функции *mMethodSubst* на канонической спецификации *F* метода общего вида *meth* типа *T* с входными параметрами p_i ($i = 1..n$), выходными параметрами p_j ($j = n+1..m$), предикативной спецификацией *f*:

```
meth: {in: function;
  params:{+p1/T1, ... , +pn/Tn, −pn+1/Tn+1, ... , −pm /Tm};
  {predicative: { f }};
}
```

Функция *mMethodSubst* действует на *F* следующим образом:

```
mMethodSubst(T)[F] =
  pn+1, ... , pm ← meth_op(av, p1,... , pn) =
  PRE av ∈ ext_T ∧ p1 ∈ mType[T1] ∧ ... ∧ pn ∈ mType[Tn]
  THEN
    mSubstitution[f]
  END
```

Первым входным параметром операции *meth_op* является объект, для которого вызывается метод *meth*. Семантическая функция *mSubstitution*, формирующая представление формул канонической модели подстановками AMN, описана в разделе определения семантики формул при помощи обобщенных подстановок.

Коллекции, описанные в модуле, представляются в AMN переменными-множествами. Переменная *C*, соответствующая коллекции *C*, тип экземпляров которой есть *T*, описывается в машине M_T , инкапсулирующей отображение спецификации типа *T* в AMN, и типизируется в разделе INVARIANT машины M_T формулой, порождаемой функцией *mCollectionInv* на основании имени типа и спецификации коллекции (синтаксический домен *ClassSpec*):

```
mCollectionInv[{C; in: class; superclass: SC1, SC2; instance_section: T;}]=
  C ∈ POW(mType[T]) ∧ mSuperclassInv(C)[SC1] ∧ mSuperclassInv(C)[SC2]
mSuperclassInv(C)[SC] = C ⊆ SC
```

Ниже приводится пример действия функции *mCollectionInv* на спецификации класса *cultivars*:

```
mCollectionInv[
  { cultivars; in:class; superclass: entities;
    instance_section: Cultivar;}  ] =
  cultivars ∈ POW(ext_Cultivars) ∧ cultivars ⊆entities
```

2.7.4 Определение семантики формул канонической модели предикатами AMN

Данный раздел посвящен описанию действия функции mPredicate на формулах канонической модели. Описание строится индуктивно, начиная с атомарных предикатов до составных формул.

Составные формулы канонической модели могут быть образованы из атомарных предикатов и условий при помощи логических операций (*&, |, ^, −>*). Семантика логических операций в формулах канонической модели определяется естественным для логики предикатов первого порядка образом:

```
mPredicate[^F] = ¬ mPredicate[F]
mPredicate[F1 & F2] = mPredicate[F1] ∧ mPredicate[F2]
mPredicate[F1 | F2] = mPredicate[F1] ∨ mPredicate[F2]
mPredicate[F1 -> F2] = mPredicate[F1] ⇒ mPredicate[F2]
```

Семантика разрешенных в канонической модели формул с кванторами определяется следующим образом:

```
mPredicate[ex v1/T1, ... , vn/Tn(F)]] =
 ∃ v1,... , vn.(mPredicate[F] ∧mTypedVar[v1/T1] ∧ ... ∧ mTypedVar[vn/Tn])
mPredicate[all v1/T1,... , vn/Tn(F)] =
 ∀ v1,... , vn.(mTypedVar[v1/T1] ∧ ... ∧ mTypedVar[vn/Tn] ⇒mPredicate[F])
```

Атомарные предикаты. К атомарным относятся предикаты-классы, предикаты-методы и встроенные предикаты канонической модели. Семантика предиката-класса определяется следующим образом:

```
mPredicate[C(x/T)] = x ∈ C ∧ x ∈ ext_T
```

Атомарным предикатом может быть вызов метода *M:*

```
v1. ··· .vn.M(p1,... , pm)
```

при условии, что *M* не изменяет состояния ИС. Семантика вызова метода определяется следующим образом:

```
mPredicate[v1. ··· .vn.M(t1,... , tm)] =
  mPredicate[f{this →v1. ··· .vn , p1 →t1,... , pm →tm}]
```

Здесь *f* – предикативная спецификация метода *M*, p_i – имена параметров метода, конструкция $f\{p \to t\}$ означает формулу *f,* в которой вхождения переменной *p* заменены на терм *t*.

Предикаты равенства, арифметические предикаты, предикаты принадлежности множеств имеют естественную семантику в AMN:

```
mPredicate[t = s] =
  mTerm[t] = mTerm[s]
mPredicate[t gt s] =
  mTerm[t] > mTerm[s]
mPredicate[t ge s] =
  mTerm[t] >= mTerm[s]
mPredicate[t lt s] =
  mTerm[t] < mTerm[s]
mPredicate[t le s] =
  mTerm[t] <= mTerm[s]
mPredicate[t ne s] =
```

```
  mTerm[t] <> mTerm[s]
mPredicate[t < s] =
  mTerm[t] ⊂ mTerm[s]
mPredicate[t <= s] =
  mTerm[t] ⊆ mTerm[s]
```

Определение семантики встроенных предикатов *is_in*, *is_empty* осуществляется следующим образом:

```
mPredicate[is_in(t, s)] =
  mTerm[t] ∈ mTerm[s]
mPredicate[is_empty(t)]] =
  mTerm[t] = {}
```

Термы. За определение семантики термов отвечает семантическая функция *mTerm*.

Арифметические выражения формируются при помощи арифметических операций (+, −, ∗, /) из других термов. Арифметические операции имеют аналоги, определенные на числовых типах AMN, и потому обладают естественной семантикой:

```
mTerm[(t + s)] =
  mTerm[t] + mTerm[s]
mTerm[(t - s)] =
  mTerm[t] - mTerm[s]
mTerm[(t * s)] =
  mTerm[t] * mTerm[s]
mTerm[(t / s)] =
  mTerm[t] / mTerm[s]
```

Операции на множествах также обладают естественной семантикой:

```
mTerm[uni(t, s)] =
  mTerm[t] ∪ mTerm[s]
mTerm[differ(t, s)] =
  mTerm[t] - mTerm[s]
mTerm[intersect(t, s)] =
  mTerm[t] ∩ mTerm[s]
mTerm[cardibal(t)] =
  card(mTerm[t])
```

Рассмотрим теперь действие функции *mTerm* на атомарных термах. Атомарным термом может быть идентификатор класса, имя переменной или параметра метода, значение встроенного типа (например, целое число или логическое значение), путь в композиционной иерархии АТД, терм-выделение множества.

Типы значений канонической модели имеют аналоги в AMN, и потому значения интерпретируются естественным образом:

```
mTerm[val] = val
```

На имени переменной *v* или имени класса *C* функция *mTerm* действует как тождественное отображение:

```
mTerm[v] = v
```

$mTerm[C] = C$

Если началом пути v_1 является указатель текущего экземпляра типа *this*, определение семантики пути осуществляется следующим образом:

$$mPath[this.v_2. \cdots .v_n] = v_n(v_{n-1}(v_2(av)))$$

Переменная *av* обозначает объект, для которого вызывается метод. Если началом пути является имя переменной, то определение семантики пути осуществляется следующим образом:

$$mPath[v_1.v_2. \cdots .v_n] = v_n(v_{n-1}(\cdots v_2(v_1)))$$

Семантика терма-выделения множества определяется следующим образом:

$$mTerm[\{v_1/T_1, \ldots , v_n/T_n \mid F\}] = \{v_1, \ldots , v_n \mid mPredicate[F] \wedge mTypedVar[v_1/T_1] \wedge \ldots \wedge mTypedVar[v_n/T_n]\}$$
$$mTypedVar[v/T] = v \in ext\_T$$

2.7.5 Определение семантики формул канонической модели обобщенными подстановками AMN

В данном разделе описывается определение семантики формул-спецификаций методов канонической модели обобщенными подстановками языка AMN.

Метод может изменять состояние базы данных. Для выражения условий, связанных с изменением состояния базы данных, ссылки на значения переменных при завершении метода изображаются идентификаторами переменных, выделенными следующими после них знаком постсостояния «~». Кроме того, метод может вызывать другие методы, в свою очередь изменяющие состояние базы данных.

Определение. Метод, изменяющий состояние базы данных, называется *методом с побочными эффектами.* Побочный эффект называется *явным*, если производится использованием в качестве терма переменной со знаком постсостояния (например, *this.cropPerYear~*). Побочный эффект называется *неявным*, если производится в спецификации метода вызовом другого метода с эффектами.

Пример. Рассмотрим пример типа, содержащего методы с побочными эффектами.

```
{ Crop; in: type;
  validPredecessor: {set; type_of_element: Crop; };
},
{ CropRotation; in: type;
  beginYear: integer;
  endYear: integer;
  cropPerYear: {array;
    index: {range; l_bound: 1; u_bound: 10; type_of_element: integer;};
    type_of_element: Crop;
  };

  insertCrop:
  { in: function; params: {+ayear/integer, +acrop/Crop};
```

```
    { predicative:
      {
        (this.cropPerYear~ = update(this.cropPerYear, acrop, ayear))
      }
    };
  };

  checkedInsertCrop:
  { in: function; params: {+ayear/integer, +acrop/Crop};
    { predicative:
      {
        (ayear ne this.beginYear &
         elem(this.cropPerYear, (ayear - 1)) ne none ->
         is_in(acrop.validPredecessors,
              elem(this.cropPerYear, (ayear - 1)))) &
        (ayear ne this.endYear &
         elem(this.cropPerYear, (ayear + 1)) ne none ->
         is_in(elem(this.cropPerYear, (ayear + 1)).validPredecessors,
                acrop)) &
         ->
         insertCrop(this, ayear, acrop)
      }
    };
  };
}
```

Метод *insertCrop* типа *CropRotation* (севооборот) устанавливает выращивание культуры *acrop* в год *ayear* севооборота. При этом происходит изменение состояния атрибута *cropPerYear* типа массив, отвечающего структуре севооборота. Эффект в методе выражен явно.

С другой стороны, метод *checkedInsertCrop* устанавливает значение *acrop* в год *ayear* только в том случае, если *acrop* является корректным *предшественником* (*validPredecessor*) для культуры следующего года *acrop+1* (при ее наличии) и культура предыдущего года *acrop-1* (при ее наличии) является корректным предшественником для *acrop*. Эффект при этом производится при помощи вызова метода *insertCrop*, то есть неявно.

Для представления спецификации метода, использующего вызовы методов с эффектами, обобщенной подстановкой, необходимо сначала преобразовать его спецификацию эквивалентным образом, заменив каждый вызов метода с эффектами на формулу-спецификацию соответствующего метода, в которой параметры метода заменены на их значения, указанные при вызове. Спецификация, преобразованная таким образом, будет обладать тем свойством, что все ее эффекты являются явными. Необходимым ограничением является отсутствие рекурсии или взаимной рекурсии в спецификациях методов с эффектами.

Нижеприведенный алгоритм *SideEffectExposure* преобразует формулу канонической модели *f* эквивалентным образом, элиминируя неявные эффекты.

Алгоритм *SideEffectExposure(f)*

начало

$f^* := f$

если существует путь $p = v_1. \cdots .v_n$, что

p - подтерм f^*

и

существует i_o, $1 < i \le n$ что $v_{i0} = m(t_1,\ldots, t_r)$ где m - метод с эффектами, и

спецификация m выглядит следующим образом

```
m: {in : function;
   params :{+p1/T1,... , +pk /Tk , -pk+1/Tk+1,... , -p1 /T1 };
   { predicative: { fm }};
}
```

и

для всех j, $1 < j < i_0$; v_j не является вызовом метода с эффектами

то

если $l = k$ **или** $l > k+1$ **то**

$f^* := f^*\{v_1. \cdots .v_{i0} \rightarrow f_m\{ this \rightarrow v_1. \ldots .v_{i0}-1, p_1 \rightarrow t_1, \ldots , p_r \rightarrow t_r \}\}$

иначе

$f^* :=$ **ex** $c_m/T_{k+1}(f_m\{ this \rightarrow v_1. \ldots .v_{i0}-1, p_1 \rightarrow t_1,\ldots , p_r \rightarrow t_r , p_{k+1} \rightarrow c_m\}$ &

$f^*\{v_1. \ldots .v_{i0} \rightarrow c_m\})$

возвратить *SideEffectExposure(f*)*

иначе возвратить f^*

конец

Алгоритм последовательно элиминирует вызовы методов с эффектами, преобразуя формулу f с использованием формул-спецификаций методов с эффектами, а потому, при условии отсутствия рекурсии, завершается. В алгоритме различаются два случая.

В первом случае происходит вызов метода без выходных параметров ($l = k = r$) или с несколькими выходными параметрами ($l > k +1$, $r = k +l$). Такой вызов может быть использован лишь как предикат-метод, а потому $i_o = n$. Вместо вызова метода в этом случае подставляется формула – предикативная спецификация f_m метода m, в которой имена параметров заменены соответствующими термами - параметрами конкретного вызова метода. Во втором случае происходит вызов метода с одним выходным параметром, при этом $l = k+1$, $r = k$. Преобразование в этом случае основано на эквивалентности формул

$$F(x, y) \Leftrightarrow \exists\, v/T_y \bullet (v = y \wedge F(x, v))$$

Здесь y - свободная переменная F типизированная типом T_y; y вместе с вектором переменных x составляют множество всех свободных переменных F. Если в качестве F рассмотреть формулу канонической модели f, в качестве y - вызов

$$w = v_1. \cdots .v_{i0-1}.m(t_1, \ldots , t_k)$$

метода m, выходной параметр p_{k+1} которого имеет тип T; получим

$$f \Leftrightarrow \exists\ v/T \bullet (v = w \wedge f\{ w \rightarrow v \}))$$

Формула же $v = w$ семантически эквивалентна формуле

$$f_m\{ this \rightarrow v_1. \cdots .v_{io-1}, p_1 \rightarrow t_1, \ldots , p_k \rightarrow t_k, p_{k+1} \rightarrow v \}$$

где f_m – формула-спецификация m, $p_1,\ldots , p_k$ - входные параметры m.

Пример. Предикативную спецификацию метода *checkedInsertCrop* из АТД *Crop* алгоритм преобразует следующим образом:

SideEffectExposure(

```
(ayear ne this.beginYear &
 elem(this.cropPerYear, (ayear - 1)) ne none ->
  is_in(acrop.validPredecessors, elem(this.cropPerYear, (ayear - 1)))) &
(ayear ne this.endYear &
 elem(this.cropPerYear, (ayear + 1)) ne none ->
  is_in(elem(this.cropPerYear, (ayear + 1)).validPredecessors, acrop)) &
 ->
 insertCrop(this, ayear, acrop)
```

) =

```
(ayear ne this.beginYear &
 elem(this.cropPerYear, (ayear - 1)) ne none ->
  is_in(acrop.validPredecessors, elem(this.cropPerYear, (ayear - 1)))) &
(ayear ne this.endYear &
 elem(this.cropPerYear, (ayear + 1)) ne none ->
  is_in(elem(this.cropPerYear, (ayear + 1)).validPredecessors, acrop)) &
 ->
 this.cropPerYear~ = update(this.cropPerYear, acrop, ayear)
```

Рассмотрим теперь алгоритм работы функции *mSubstitution*, формирующей представление формул канонической модели подстановками AMN. Пусть f_m есть формула – спецификация метода m типа T:

```
m: {in : function;
  params: { +p1/T1,... , +pk/Tk , -pk+1/Tk+1,... , -p1/T1 };
  { predicative: {fm}}; }
}
```

Алгоритм *mSubstitution*[f_m]

начало

пусть $t_1, \ldots, t_n$ - все такие различные подтермы формулы *SideEffectExposure*(f_m), являющиеся путями знаком постсостояния «~», и типы T_{ti} есть типы термов t_i

пусть $v_1, \ldots, v_n, w_{k+1}, \ldots, w_l$ - имена новых переменных, не имеющих вхождений в f_m

возвратить

ANY $v_1, \ldots, v_n, w_{k+1}, \ldots, w_l$ WHERE

$v_1 \in ext\_T_{t1} \wedge \ldots v_n \in ext\_T_{tn} \wedge w_{k+1} \in ext\_T_{k+1} \wedge \ldots \wedge w_l \in ext\_T_l \wedge$

mPredicate[*SideEffectExposure*(f_m){$t_1 \rightarrow v_1, \ldots, t_n \rightarrow v_n, p_{k+1} \rightarrow w_{k+1}, \ldots, p_l \rightarrow w_l$}]]

THEN

Assignment(t_1, v_1); ... ; *Assignment*(t_n, v_n);

$p_{k+1} := w_{k+1}$; ... ; $p_l := w_l$

END

конец

Алгоритм *Assignment* формирует подстановку, присваивающую атрибуту, на который ссылается путь t , значение v.

Алгоритм *Assignment*(t, v)

начало

пусть $t = a_1. \cdots .a_n$

если функциональная переменная, соответствующая атрибуту a_n, инкапсулирована в той же машине, что и семантика типа T

то возвратить *sPath*[t] := v

иначе возвратить *set_a*$_n$({*sPath*[t] ↦ v})

конец

Пример. Спецификация метода *checkedInsertCrop* отображается в операцию AMN *checkedInsertCrop* при помощи семантических функций *mMethodSubst*, *mSubstitution, mPredicate* следующим образом:

```
checkedInsertCrop(av, ayear, acrop)=
PRE av: ext_CropRotation ∧ ayear ∈ INT ∧ acrop ∈ ext_Crop
THEN
  ANY acropPerYear WHERE
    acropPerYear ∈ ext_CropRotation → ((1..10) → ext_Crop) ∧
    (ayear <> beginYear(av) ∧ cropPerYear(av)(ayear - 1) <> none ⇒
      cropPerYear(av)(ayear - 1) ∈ validPredecessors(acrop) )) ∧
    (ayear <> endYear(av) ∧ cropPerYear(av)(ayear + 1) <> none ⇒
      acrop ∈ validPredecessors(cropPerYear(av)(ayear + 1))
     ⇒
     acropPerYear = cropPerYear(av) <+ (ayear ↦ acrop)
  THEN
```

```
    cropPerYear(av):= acropPerYear
  END
END
```

2.7.6 Реализация семантического отображения канонической модели данных в язык AMN

В данном разделе описываются и иллюстрируются основные принципы реализации семантического отображения семантического отображения канонической модели данных в язык AMN в виде трансформации на языке ATL. Трансформация размещена в репозитории GitHub[48]. Абстрактный синтаксис языков СИНТЕЗ и AMN представляется в виде моделей уровня M2 (называемых *Synthesis*[49] и *Amn*[50] соответственно), конформных метамодели *Ecore.* Трансформация автоматически порождает схему (модель уровня M1), удовлетворяющую модели *Amn,* на основании схемы, удовлетворяющей модели *Synthesis.*

Фрагмент модели *Synthesis*, необходимый для описания и иллюстрации принципов построения трансформации, очищенный от синтаксического сахара XML, выглядит следующим образом:

```
ElementDef(name: String): ValueDef
FrameDef(parameters: ParameterDef*): ElementDef
ParameterDef(parameterKind: String, parameterOf: FrameDef,
  type: TypeDef): ElementDef
TypeDef(typeInModule: ModuleDef): FrameDef
ModuleDef(containedTypes: TypeDef*): FrameDef
ADTDef(attributes: AttributeDef*, invariants: InvariantDef*,
  subtypes: ADTDef*, supertypes: ADTDef*): TypeDef
AttributeDef(attributeOf: ADTDef, type: TypeDef): ElementDef
InvariantDef(predicativeSpec: Formula): TypeDef
FunctionDef(functionInModule: ModuleDef, predicativeSpec: Formula): TypeDef
IntegerDef(unsigned: Boolean): TypeDef
SetDef(ofType: TypeDef): TypeDef

Formula()
Variable(): ValueDef
Atom(symbol: String, terms: ValueDef): Formula
BuiltInPredicate(): Atom
ArithmeticPredicate(): Atom
Conjunction(formula: Formula*):Formula
FunctionCall(terms: ValueDef): ElementDef
BuiltInFunction(): FunctionCall
IntegerValueDef(value: EInt): ValueDef
```

---

48 https://github.com/sstupnikov/ModelTransformation/tree/master/Synthesis2AMN

49 https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis2AMN/Metamodels/Synthesis.ecore

50 https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis2AMN/Metamodels/AMN.ecore

Фрагмент модели *AMN*, необходимый для описания и иллюстрации принципов построения трансформации, очищенный от синтаксического сахара XML, выглядит следующим образом:

```
Element(name: String)
Machine(sees: AbstractMachine*,
  extendsClause: AbstractMachine*, sets: Set*,
  properties: Predicate, invariant: Predicate,
  operations: Operations*): Element
Refinement(abstractConstants: Element*,
  abstractVariables: Element*):Machine
Operation(inputParams: Variable*, outputParams: Variable*,
  Substitution: Substitution): Element
Set(): Element
Deferred(): Set

Predicate(stringRepr: String)
AtomicPredicate(sign: String, expression: Expression*): Predicate
Conjunction(predicate: Predicate*): Predicate
Expression()
OperationalExpression(sign: String): Expression
BinaryOperator(expression: Expression*): OperationalExpression
FunctionalExpression(expression: Expression*): OperationalExpression
Variable(name: String): Expression
NamedConstant (name: String): Expression
IntegerValue(value: EInt): Expression
Substitution()
Precondition(pre: Predicate, thenPart: Substitution): Substitution
Any(any: Variable*, where: Predicate, thenPart: Substitution): Substitution
SequenceSubstitution(substitution: Substitution*): Substitution
```

Ниже в разделе приводятся основные принципы формирования трансформации. В качестве иллюстрации принципов приводятся фрагменты ATL-правил. Полные правила, составляющие трансформацию, опубликованы в репозитории GitHub по вышеприведенной ссылке.

Формирование заголовка трансформации. Трансформация, преобразующая спецификации языка СИНТЕЗ в спецификации языка AMN, представляет собой модуль языка ATL, получающий на вход модель уровня M1, конформную модели *Synthesis* и порождающий модель уровня M1, конформную модели *AMN*. Заголовок модуля выглядит следующим образом:

```
module Synthesis2AMN;
create OUT:AMN from IN:Synthesis;
```

Порождение контекстной машины. Правило, порождающее контекстную машину *am* на основании модуля *m* языка СИНТЕЗ, выглядит следующим образом:

```
rule Module2Context{
  from m: Synthesis!ModuleDef
  using{
    adts0: Set(Synthesis!ADTDef) =
      m.containedTypes->select(t | t.oclIsKindOf(Synthesis!ADTDef));
  }
  to ctxt: AMN!AbstractMachine(
      name <- m.contextMachineName,
      sets <- Sequence{avalSet, oidSet},
      abstractConstants <- Sequence{'Obj', 'self'},
      properties <- Sequence{objProp, selfProp}
    ),
    avalSet: AMN!SetDecl(name <- 'AVAL'),
    oidSet: AMN!SetDecl(name <- 'OID'),
    objProp : AMN!AtomicPredicate (
      sign <- ':',
      expression <- Sequence{objConst, objType}
    ),
    objConst : AMN!NamedConstant(name <- 'Obj'),
    objType : AMN!FunctionalExpression (
      sign <- 'POW',
      expression <- aval
    ),
    aval: AMN!NamedConstant(name <- 'AVAL'),
    selfProp: AMN!AtomicPredicate(
      sign <- ':',
      expression <- Sequence{selfConst, selfType}
    ),
    selfConst: AMN!NamedConstant(name <- 'self'),
    selfType: AMN!BinaryOperator (
      sign <- '>->>',
      expression <- Sequence{objConst1, oid}
    ),
    objConst1: AMN!NamedConstant(name <- 'Obj'),
    oid: AMN!NamedConstant(name <- 'OID')
  do{
    for(adt in adts0){
      ctxt.abstractConstants <-
        Sequence{'extp_' + adt.name.amnId, 'ext_' + adt.name.amnId};
      ctxt.properties <-
        Sequence{ thisModule.resolveTemp(adt, 'ownExtTyp') };
    }
  }
}

helper context Synthesis!ModuleDef def: contextMachineName: String =
  if not (self.name.trim() = '')
  then self.name.amnId + '_Context'
  else 'Undefined_Context'
  endif
;
```

Контекстной машине присваивается имя (при помощи вспомогательной функции *contextMachineName*), добавляются множества *AVAL*, *OID,* константы *Obj, self* и предикаты *objProp, selfProp*, описывающие их свойства. В секции *from* правила описывается имя (*m*) и тип (*Synthesis!ModuleDef*) входного элемента, в секции *to* – имена (*ctxt, avalSet, oidSet, objProp, objConst, objType, aval, selfProp, selfConst, selfType,*

objConst1, oid) и типы выходных элементов. Для каждого выходного элемента устанавливаются его свойства. В секции *using* определяется вспомогательное множество *adts0* всех АТД модуля. Константы и предикаты, отражающие экстенсиональный принцип представления АТД в AMN, определяются в секции императивного кода *do* (выполняющегося после создания элементов, указанных в секции *to*). Специальная функция *resolveTemp* применяется для того, чтобы получить доступ к элементам, созданным другими правилами. Так, выражение *thisModule.resolveTemp(adt, 'ownExtTyp')* возвращает атомарный предикат типизации *ownExtTyp*, созданный на основе АТД *adt* правилом *ADT*.

Порождение конструкций AMN, выражающих семантику АТД. Правило, порождающее отдельную конструкцию REFINEMENT для каждого АТД, выглядит следующим образом:

```
rule ADT{
  from adt: Synthesis!ADTDef
  using{
    nonMethods: Set(Synthesis!AttributeDef) = adt.attributes->
      select(a | not a.type.oclIsKindOf(Synthesis!Predicative));
    methods: Set(Synthesis!AttributeDef) = adt.attributes->
      select(a | a.type.oclIsKindOf(Synthesis!FunctionDef));
    invs: Set(Synthesis!AttributeDef) = adt.attributes->
      select(a | a.type.oclIsKindOf(Synthesis!InvariantDef));
  }
  to
    ref: AMN!Refinement(
      name <- adt.name.amnId + 'Ref',
      sees <- thisModule.resolveTemp(adt.typeInModule, 'ctxt'),
      abstractVariables <- nonMethods->collect(a | a.name.amnId),
      invariant <- nonMethods->
        collect(attr | thisModule.Attribute2Invariant(attr)),
      invariant <- invs->collect(attr | attr.type.predicativeSpec),
      operations <- methods
    ),
    ownExtTyp: AMN!AtomicPredicate (
      sign <- ':',
      expression <- extp0,
      expression <- powObj0
    ),
    extp0: AMN!NamedConstant(name <- 'extp_' + adt.name.amnId),
    powObj0 : AMN!FunctionalExpression (
      sign <- 'POW',
      expression <- objConst0
    ),
    objConst0: AMN!NamedConstant(name <- 'Obj')
  do{
    if(adt.supertypes->notEmpty()){
      for(e in adt.supertypes){
        thisModule.SubtypeRelation(e, adt);
      }
    }
  }
}
```

В секции *using* вышеприведенного правила атрибуты АТД разделяются на обычные атрибуты (*nonMethods*), методы (*methods*), инварианты (*invs*). В секции *to* для конструкции *ref* определяется имя (*name*), в секцию SEES добавляется контекстная машина, набор переменных (*abstractVariables*) формируется на основании обычных атрибутов АТД, инвариант формируется на основании инвариантов АТД, операции (*operations*) формируются на основании методов АТД. Приведены константы и выражения, необходимые для формирования выходного элемента *ownExtTyp*, использующегося при формировании контекстной машины. В секции *do* для каждого супертипа АТД происходит вызов правила *SubtypeRelation*, формирующего в контекстной машине предикат, выражающий семантику отношения тип-подтип на экстенсионалах АТД:

```
rule SubtypeRelation(sup: Synthesis!ADTDef, sub: Synthesis!ADTDef){
  to
    rel: AMN!AtomicPredicate(
      sign <- '<<:',
      expression <- Sequence{subConst, supConst}
    ),
    supConst: AMN!NamedConstant(name <- 'ext_' + sup.name.amnId),
    subConst: AMN!NamedConstant(name <- 'ext_' + sub.name.amnId)
  do {
    thisModule.resolveTemp(sup.typeInModule, 'ctxt').properties <- rel;
  }
}
```

Порождение предикатов типизации переменных, соответствующих атрибутам АТД. Правило, порождающее на основании спецификации атрибута АТД предикаты типизации в машине типа, выглядит следующим образом:

```
lazy rule Attribute2Invariant{
  from attr: Synthesis!AttributeDef(
          not attr.type.oclIsKindOf(Synthesis!Predicative))
  to  inv: AMN!Conjunction(
      predicate <- attrTyping
    ),
    attrTyping: AMN!AtomicPredicate(
      sign <- ':',
      expression <- attrVar,
      expression <- extFunc
    ),
    attrVar: AMN!NamedConstant(name <- attr.name.amnId),
    extFunc: AMN!BinaryOperator (
      sign <- '-->',
      expression <- extConst
    ),
    extConst: AMN!NamedConstant(name <- 'ext_' +
attr.attributeOf.name.amnId)
  do{
    thisModule.PutAttributeTypeToBinaryOperator(attr, extFunc);
  }
}
```

Правило формирует инвариант типа – конъюнктивную формулу (*inv*), включающую предикат типизации (*attrTyping*). Формирование выражения, соответствующего типу атрибута происходит при помощи правила *PutAttributeTypeToBinaryOperator*:

```
rule PutAttributeTypeToBinaryOperator(
    attr: Synthesis!AttributeDef, bop: AMN!BinaryOperator){
  do{
    if(attr.type.oclIsKindOf(Synthesis!ADTDef))
      bop.expression <- thisModule.ADT_Typing(attr.type);
    else
    if(attr.type.oclIsKindOf(Synthesis!IntegerDef))
      bop.expression <- thisModule.Integer_Typing(attr.type);
  }
}
```

Это правило, в свою очередь, вызывает отдельные правила формирования выражений в зависимости от типа атрибута: для АТД – правило *ADT_Typing,* для целочисленного типа – правило *Integer_Typing*:

```
lazy rule ADT_Typing{
  from adt: Synthesis!ADTDef
  to  const: AMN!NamedConstant(name <- 'ext_' + adt.name.amnId)
}

lazy rule Integer_Typing{
  from int: Synthesis!IntegerDef
  to  const: AMN!NamedConstant(name <- 'INTEGER')
}
```

Порождение конструкций AMN, выражающих семантику методов АТД. Правило, порождающее операции AMN на основании спецификаций методов АТД, выглядит следующим образом:

```
rule FunctionalAttribute2Operation{
  from
    attr: Synthesis!AttributeDef(
              attr.type.oclIsKindOf(Synthesis!FunctionDef))
  using{
    inputPars: Sequence(Synthesis!ParameterDef) =
        attr.type.parameters->select(p | p.parameterKind = '+');
    outputPars: Sequence(Synthesis!ParameterDef) =
        attr.type.parameters->select(p | p.parameterKind = '-');
  }
  to
    operation: AMN!Operation (
      name <- attr.name.amnId,
      inputParams <- thisModule.firstParamOfObjectMethod,
      inputParams <- inputPars->collect(p | p.name.amnId),
      outputParams <- outputPars->collect(p | p.name.amnId),
      substitution <- subst
    ),
    subst: AMN!Precondition(
```

```
      pre <- objParamTyping,
      pre <- inputPars,
      thenPart <- anySubst
    ),

    objParamTyping: AMN!AtomicPredicate (
      sign <- ':',
      expression <- Sequence{objParam, ext}
    ),
    objParam: AMN!Variable(name <- thisModule.firstParamOfObjectMethod),
    ext: AMN!NamedConstant(name <- 'ext_' + attr.attributeOf.name.amnId),
    anySubst: AMN!Any(
      any <- outputPars->collect(p | p.outParamTempVarName),
      where <- whr,
      thenPart <- simSubst
    ),
    whr: AMN!Conjunction(
      predicate <- outputPars,
      predicate <- attr.type.predicativeSpec
    ),
    simSubst: AMN!Simultaneous(
      substitution <-
          outputPars->collect(p | thisModule.resolveTemp(p, 'assign'))
    )
}
```

Правило применяется для атрибутов, являющихся функциями (*FunctionDef*). Вспомогательные переменные *inputPars, outputPars* группируют входные и выходные параметры метода соответственно. Формируется операция AMN (*operation*) с именем, соответствующим имени атрибута. Имена входных (*inputParams*) и выходных (*outputParams*) параметров операции соответствуют именам параметров метода. Тело операции (*substitition*) образует подстановка с предусловием типизации входных параметров (*subst*). Ее тело (*thenPart*) образует подстановка ANY (*anySubst*), накладывающая ограничения на значения выходных параметров (формирующиеся на основании типов выходных параметров и предикативной спецификации метода *predicativeSpec*) и содержащая параллельную подстановку присваивания значений выходным парамерам (*simSubst*).

Правило *Parameter* формирует предикаты типизации (*paramTyping*) и подстановки присваивания значений (*assign*) параметрам функций, использующиеся в вышеприведенном правиле *FunctionalAttribute2Operation*:

```
rule Parameter{
  from p: Synthesis!ParameterDef
  to
    paramTyping: AMN!AtomicPredicate (
      sign <- ':',
      expression <- param
    ),
    param: AMN!Variable(
      name <- if p.parameterKind = '+'
          then p.name.amnId
```

```
          else p.outParamTempVarName
          endif
    ),
    assign: AMN!BecomesEqual(
      leftExpression <- paramVar,
      rightExpression <- newParamVar
    ),
    paramVar: AMN!Variable(name <- p.name.amnId),
    newParamVar: AMN!Variable(name <- p.outParamTempVarName)
}
```

Порождение конструкций AMN, выражающих семантику формул канониччеcкой модели. Каждая из конструкций, составляющих формулы языка СИНТЕЗ, преобразуется в формулы AMN при помощи отдельного правила. Так, существуют правило для преобразования логических связок (например, конъюнкции):

```
rule Conjunction {
  from fSyn: Synthesis!Conjunction
  to
    fAMN: AMN!Conjunction(
      predicate <- fSyn.formula
    )
}
```

формул с кванторами (например, с квантором всеобщности):

```
rule UniversallyQuantified{
  from
    fSyn: Synthesis!UniversallyQuantifiedFormula
  to
    fAMN: AMN!UniversalPredicate(
      predicate <- body,
      variable <- fSyn.variables->collect(v | v.name.amnId)
    ),
    body: AMN!Implication(
      predicate <- varTyping,
      predicate <- fSyn.formula
    ),
    varTyping: AMN!Conjunction(
      predicate <-
          fSyn.variables->collect(v | thisModule.TypedVar2Predicate(v))
    )
}
```

встроенных предикатов (например, *is_in*):

```
rule BuiltinPredicate{
  from fSyn: Synthesis!Atom(fSyn.oclIsTypeOf(Synthesis!Atom) and
       fSyn.isBuiltin)
  to
    fAMN: AMN!AtomicPredicate(
      expression <-
        if fSyn.symbol = 'is_in'
        then Sequence{ fSyn.terms->asSequence()->at(2),
                       fSyn.terms->asSequence()->at(1)}
        else fSyn.terms
```

```
        endif
    )
  do{
    if(fSyn.symbol = 'is_in') fAMN.sign <- ':';
  }
}
```

арифметических предикатов:

```
rule RelationPredicate{
  from
    fSyn: Synthesis!RelationPredicate
  to
    fAMN: AMN!AtomicPredicate(
      expression <- fSyn.terms
    )
  do{
    if(fSyn.symbol = 'lt') fAMN.sign <- '<';
    if(fSyn.symbol = 'le') fAMN.sign <- '<=';
    if(fSyn.symbol = '>=' or fSyn.symbol = 'ge') fAMN.sign <- '>=';
    if(fSyn.symbol = '>' or fSyn.symbol = 'gt') fAMN.sign <- '>';
    if(fSyn.symbol = '<>' or fSyn.symbol = 'ne') fAMN.sign <- '/=';
    if(fSyn.symbol = '=' or fSyn.symbol = 'eq') fAMN.sign <- '=';
  }
}
```

встроенных функций (например, *intersect*):

```
rule BinaryOperator{
  from valSyn: Synthesis!FunctionCall(valSyn.isBinaryOperator)
  to  valAMN: AMN!BinaryOperator(
      expression <- valSyn.terms
    )
  do{
    if(valSyn.name = 'intersect') valAMN.sign <- '/\\';
  }
}
```

значений (например, целочисленных):

```
rule IntValue{
  from valSyn: Synthesis!IntValueDef
  to valAMN: AMN!IntegerValue(value <- valSyn.value)
}
```

переменных:

```
rule Variable{
  from valSyn: Synthesis!Variable
  using{
    varName: String = valSyn.name.amnId;
  }
  to  valAMN: AMN!Variable(
      name <- varName
    )
}
```

Полные определения остальных правил, в том числе правила формирования конструкций AMN, соответствующих путям (*NavigationPath*) и эффектам методов

(*Effect*), содержатся в трансформации, опубликованной в репозитории GitHub по сылке, приведенной в начале раздела.

2.8 Формализация семантики исходных моделей данных и верификация корректности отображения моделей

В данном разделе рассматриваются три метода формализации семантики исходных моделей данных и верификации корректности отображения моделей данных:

– на основе уточнения образцов моделей данных;

– на основе уточнения AMN-спецификаций моделей данных;

– на основе эквивалентного представления моделей данных в единой семантической структуре.

Вопросы применимости этих методов обсуждались в разделе 2.1 при изложении общей структуры метода унификации моделей.

2.8.1 Верификация на основе уточнения образцов

Формализация семантики модели данных означает построение трансформации схем исходной модели данных в AMN-спецификации. Эта семантическая трансформация создаются вручную экспертом с использованием языков трансформаций (подобных ASF или ATL). Концептуально процесс формализации семантики исходных моделей данных аналогичен процессу формализации семантики канонической модели данных, описанному в разделе 2.7. Так, основные идеи формализации семантики языка OWL как исходной модели в языке AMN состоят в следующем:

– онтология OWL представляется в AMN одноименной спецификацией вида REFINEMENT;

– множество всех индивидуализированных объектов (individuals) онтологии представляется в AMN предопределенным множеством *Ind*;

– классы онтологии OWL представляются в AMN одноименными переменными, типизированными в инварианте как подмножества *Ind*. Также классу ставится в соответствие операция AMN, пополняющая класс новым элементом;

– объектные свойства OWL представляются в AMN одноименными переменными, типизированными в инварианте как отношения между множествами, представляющими области определения и области значения свойств. Также свойству ставится в соответствие операция AMN, изменяющая значение этого свойства у некоторого индивидуализированного объекта. Операция изменяет значение свойства у индивидуализированного объекта *ind* на некоторое значение *val*, но только в том случае,

когда выполняется предусловие операции, гарантирующее выполнение инварианта после выполнения операции;

– различные ограничения на классы и свойства OWL (ограничения значений и кардинальности свойств, аксиомы эквивалентности классов и т. д.) представляются в AMN соответствующими частями инварианта.

Изложенные идеи формализованы в виде трансформации на языках ASF и SDF[51]. На языке SDF определены сигнатуры правил трансформации. Например, сигнатуры основных правил, используемых для создания конструкции REFINEMENT из онтологии выглядят следующим образом:

```
module unifier/owl2synthesis/owl-translator
imports unifier/owl/OWL-Syntax
imports unifier/synthesis/Synthesis-Syntax
imports unifier/Uni-Common

exports
context-free syntax

  create-Refinement(OwlSpecification) -> Refinement
  create-Variables(Directive*) -> {Amn-Id ","}*
  simplify-Predicate(Predicate) -> Predicate
  create-Initialisation(Directive*) -> Substitution
  create-Operations(Directive*) -> {Operation ";"}*
```

В ASF-файле определяются спецификации правил, соответствующие сигнатурам:

```
[Refinement]
Amn-Id := create-Refinement-Name(OntologyID?)
====>
create-Refinement (NamespaceDef* Ontology( OntologyID? Directive*)) =
REFINEMENT Amn-Id
VARIABLES
create-Variables(Directive*)
INVARIANT
simplify-Predicate(simplify-Predicate(create-Invariant(Directive*)))
INITIALISATION
create-Initialisation(Directive*)
OPERATIONS
create-Operations(Directive*)
END
```

Трансформация исходной модели данных в AMN позволяет автоматически преобразовывать любой синтаксически корректный *образец* модели данных (например, онтологию OWL) в спецификацию AMN, выражающую ее семантику. Например,

[51] https://github.com/sstupnikov/ModelTransformation/tree/master/OWL2Synthesis.Meta-Environment/unifier/owl2amn

семантика фрагмента онтологии GLoSIS, рассмотренного в разделе 2.5, приведена в таблице 2.4.

Таблица 2.4 – Образец OWL и его семантика в AMN

Онтология OWL	Семантическая спецификация AMN
Ontology (GLoSIS **Class**(FeatureOfInterest) **Class**(GL_Plot FeatureOfInterest) **Class**(GL_Site FeatureOfInterest) **Class**(Observation **restriction**(hasFeatureOfInterest **cardinality**(1))) **ObjectProperty**(hasFeatureOfInterest **domain**(Observation) **range**(FeatureOfInterest)) **Class**(CropClass Observation **restriction**(hasFeatureOfInterest **allValuesFrom**(**unionOf**(GL_Plot GL_Site)))))	**REFINEMENT** GLoSIS **SETS** Ind **VARIABLES** GL_Plot, GL_Site, FeatureOfInterest, Observation, CropClass, hasFeatureOfInterest **INVARIANT** GL_Plot: **POW**(Ind) & GL_Site: **POW**(Ind) & FeatureOfInterest: **POW**(Ind) & GL_Plot <: FeatureOfInterest & GL_Site <: FeatureOfInterest & Observation: POW(Ind) & CropClass <: Observation & hasFeatureOfInterest: Observation <-> FeatureOfInterest & !(obs).(obs: Observation => card(hasFeatureOfInterest[{obs}]) = 1) & CropClass <: { xx | xx: Ind & !(yy).(yy: Ind & (xx|->yy): hasFeatureOfInterest => yy: GL_Plot \/ GL_Site) } **INITIALISATION** GL_Plot:= {} || GL_Site:= {} || FeatureOfInterest:= {} || Observation:= {} || CropClass:= {} **OPERATIONS** include_Observation(ind, foi)= **PRE** ind: Ind & foi: FeatureOfInterest **THEN** hasFeatureOfInterest(ind) := foi || Observation := Observation \/ {ind} **END;** include_CropClass(ind, foi)= **PRE** ind: Ind & foi: FeatureOfInterest & foi: GL_Plot \/ GL_Site **THEN** hasFeatureOfInterest(ind) := foi || CropClass := CropClass \/ {ind} || Observation := Observation \/ {ind} **END;** set_featureOfInterest(ind, val)= **PRE** ind: Observation & val: FeatureOfInterest & (ind: CropClass => val: GL_Plot \/ GL_Site) **THEN** hasFeatureOfInterest:= hasFeatureOfInterest \/ {ind|->val} **END** **END**

Далее к спецификации образца модели данных применяется *трансформация исходной модели в каноническую* (конструирование такой трансформации на примере формализации отображения языка OWL в язык СИНТЕЗ и пример трансформации приведенного выше образца рассмотрены в разделе 2.5) и автоматически генерируется *каноническая спецификация образца*. Затем к полученной канонической спецификации образца применяется семантическая трансформация в AMN (такая семантическая трансформация языка СИНТЕЗ как канонической модели рассмотрена в разделе 2.7), и автоматически генерируется *каноническая семантическая спецификация образца*. Например, семантика канонического представления фрагмента онтологии GLoSIS, рассмотренного в разделе 2.5, приведена в таблице 2.5.

Таблица 2.5 – Пример канонической и семантической спецификации образца OWL

Каноническая спецификация образца	**Каноническая семантическая спецификация образца**
{ GLoSIS; **in: module, ontology;** **type:** { GL_Plot; **in: type, owl;** **supertype:** FeatureOfInterest; }, { GL_Site; **in: type, owl;** **supertype:** FeatureOfInterest; }, { FeatureOfInterest; **in: type, owl;** }, { Observation; **in: type, owl;** hasFeatureOfInterest: FeatureOfInterest; **metaslot in:** HasFeatureOfInterest; **min_card:** 1; **max_card:** 1; **end** }, { CropClass; **in: type, owl;** **supertype:** Observation; }; **class_specification:** { gl_plot; **in: class, owl;** **superclass:** featureOfInterest; **instance_section:** GL_Plot; }, { gl_site; **in: class, owl;** **superclass:** featureOfInterest; **instance_section:** GL_Site; }, { featureOfInterest; **in: class, owl;**	**REFINEMENT** CanonicalGLoSIS **ABSTRACT_CONSTANTS** ext_GL_Plot, ext_GL_Site, ext_FeatureOfInterest, ext_Observation, ext_CropClass **PROPERTIES** ext_GL_Plot: **POW**(Obj) & ext_GL_Site: **POW**(Obj) & ext_FeatureOfInterest: **POW**(Obj) & ext_Observation: **POW**(Obj) & ext_CropClass: **POW**(Obj) & ext_GL_Plot<:ext_FeatureOfInterest & ext_GL_Site<:ext_FeatureOfInterest & ext_CropClass <: ext_Observation **VARIABLES** gl_plot, gl_site, featureOfInterest, observation, cropClass **INVARIANT** gl_plot: **POW**(ext_GL_Plot) & gl_site: **POW**(ext_GL_Site) & featureOfInterest: **POW**(ext_FeatureOfInterest) & observation: **POW**(ext_Observation) & cropClass: **POW**(ext_CropClass) & gl_plot <: featureOfInterest & gl_site <: featureOfInterest & hasFeatureOfInterestCan: ext_Observation +-> ext_FeatureOfInterest & !(obs).(obs: ext_Observation & obs: observation => obs: **dom**(hasFeatureOfInterestCan)) & !(obs).(obs: ext_Observation & obs: observation => hasFeatureOfInterestCan(obs): featureOfInterest) &

```
  instance_section:
FeatureOfInterest;
},
{ observation; in: class, owl;
  instance_section: {
    in: type, owl;
    supertype: Observation;
    inv_hasFeatureOfInterest: {
     in: predicate, invariant;
     {{ all obs/Observation (
     in(obs, observation) ->
     in(obs.hasFeatureOfInterest,
          featureOfInterest)  )
     }}
  };
},
{ cropClass; in: class, owl;
  instance_section: {
   in: type, owl;
   supertype: CropClass;
   restriction_hasFeatureOfInterest:
   { in: predicate, invariant;
     {{ all cc/CropClass (
      in(cc, cropClass) ->
      in(cc.hasFeatureOfInterest,
        union(gl_plot, gl_site)
     }}
   };
  };
};
}
```

```
  !(cc).(cc: ext_CropClass & cc:
cropClass =>
      hasFeatureOfInterestCan(cc):
(gl_plot \/ gl_site) )
INITIALISATION
  gl_plot := {} || gl_site := {} ||
featureOfInterest := {} ||
  observation := {} || cropClass := {}
hasFeatureOfInterestCan := {}
OPERATIONS
include_Observation(ind, foi)=
PRE ind: ext_Observation &
    foi: ext_FeatureOfInterest & foi:
featureOfInterest
THEN
hasFeatureOfInterestCan(ind) := foi ||
observation := observation \/ {ind}
END;

include_CropClass(ind, foi)=
PRE ind: ext_Observation &
    foi: ext_FeatureOfInterest & foi:
featureOfInterest &
    foi: gl_plot \/ gl_site &
THEN
hasFeatureOfInterestCan(ind) := foi ||
 cropClass := cropClass \/ {ind} ||
 observation := observation \/ {ind}
END;

set_featureOfInterest(ind, val)=
PRE ind: ext_Observation & ind:
observation &
    val: ext_FeatureOfInterest & val:
featureOfInterest &
    (ind: ext_CropClass & ind:
cropClass => val: gl_plot \/ gl_site)
THEN
 hasFeatureOfInterestCan(ind):= val
END;

END
```

Финальным этапом верификации является доказательство того факта, что семантическая спецификация образца *уточняет* каноническую семантическую спецификацию образца. Семантические спецификации загружаются в проект инструментария Atelier B. Эксперт создает *склеивающий инвариант*, связывающий состояния переменных уточняющей и уточняемой спецификаций. Этот инвариант конъюнктивно добавляется к инварианту уточняющей спецификации. После этого для доказательства уточнения применяются автоматизированные и интерактивные средства доказательства.

Так, для связи состояния спецификаций *GLoSIS* и *CanonicalGLoSIS,* приведенных выше, был определен и добавлен в спецификацию *GLoSIS* склеивающий инвариант (приведен фрагмент инварианта):

```
GL_Plot = gl_plot & GL_Site = gl_site &
FeatureOfInterest = featureOfInterest &
Observation = observation & CropClass = cropClass  &
!(ind, val).(ind: Ind & val: FeatureOfInterest =>
    (((ind|->val): hasFeatureOfInterest) <=>
         (hasFeatureOfInterestCan(ind) = val)))
```

Полные семантические спецификации *образца* были загружены в проект Atelier B 4.7.2. Далее автоматически были сгенерированы 88 теорем, в совокупности утверждающие факт уточнения спецификаций. Большое количество теорем объясняется тем, что сложные теоремы автоматически подразделяются инструментальным средством на более простые, которые можно доказывать независимо. С использованием автоматических средств доказательства было доказано 52 теоремы. Технические детали верификации унификации языка OWL (в том числе, касающиеся данного образца) более полно изложены в приложении А.3.

2.8.2 Верификация на основе уточнения AMN-спецификаций моделей

Раздел основан на работах автора [89][60][93][58].

Метод доказательства сохранения информации и семантики операций при отображении моделей данных, рассматриваемый в данном разделе, основывается на представлении семантики моделей целиком как спецификаций в формальном языке спецификаций AMN.

Идея метода заключается в следующем. Рассмотрим исходную модель S и целевую модель T. Построим отображение θ модели S в модель T. Выразим семантику моделей в виде спецификаций AMN, построив при этом машины M_S и M_T соответственно. При этом структуры данных моделей – например, классы, массивы и т.д. – представляются переменными машин, различные свойства структур данных представляются инвариантами машин, характерные операции моделей данных представляются операциями машин. Рассматриваемые операции исходной и целевой модели должны быть связаны отображением ЯМД. Отображение ЯОД представляется в виде специального *склеивающего инварианта* – замкнутой формулы, связывающей состояния машин M_S и M_T.

Будем считать отображение θ *сохраняющим информацию и семантику операций*, если машина M_S, соответствующая исходной модели, уточняет машину M_T,

соответствующую целевой модели. Уточнение доказывается автоматизированно при помощи специальных программных средств [206].

Ниже в данном разделе метод иллюстрируется на примере доказательства сохранения информации и семантики операций при отображении графовой модели данных в каноническую.

2.8.2.1 Синтетическая модель данных атрибутированных графов

В настоящее время существует большое количество СУБД, модели данных которых основаны на простых или атрибутированных графах. Языки определения данных (ЯОД), языки манипулирования данными (ЯМД), прикладные интерфейсы пользователя (API) различаются в этих системах весьма существенно. Для того, чтобы обеспечить общность подхода по унификации графовых моделей, в данном разделе рассматривается синтетическая модель данных атрибутированных графов. Структуры данных модели покрывают возможности моделей таких известных систем, как, например, Neo4j [174], Sparksee [175], InfiniteGraph, OrientDB, VertexDB, Filament, OQGraph, Horton [176], InfoGrid.

В качестве ЯМД синтетической модели рассматривается декларативный язык Cypher [225], развиваемый в системе Neo4j. Поэтому, фактически, рассматриваемая модель является расширением графовой модели Neo4j. С точки зрения общности по отношению к другим графовым языкам запросов, язык Cypher покрывает такие классы возможностей, как смежность (adjacency) вершин и ребер, достижимость по путям фиксированной длины (fixed-length paths reachability), достижимость по простым регулярным путям (regular simple paths), поиск кратчайших путей, поиск подграфов по образцу (pattern matching). Это также означает, что язык Cypher покрывает возможности языков запросов основных современных графовых баз данных (в том числе, перечисленных в предыдущем параграфе).

Итак, синтетическая графовая модель выбрана таким образом, чтобы рассматриваемый метод отображения ее в каноническую можно было применить для унификации различных реальных графовых моделей систем, упомянутых выше.

Рассмотрим сначала вопросы определения данных в модели данных атрибутированных графов.

База данных в модели есть граф, вершины и ребра которого *типизированы*. Тип вершины или ребра представляет собой, фактически, совокупность атрибутов (свойств),

приписываемых вершине или ребру. Определим формально множество всевозможных типов вершин *VertexTypes* и множество типов ребер *EdgeTypes*.

Так, *VertexTypes* представляет собой множество троек вида <*id, name, A*>, где *id* – идентификатор типа (например, целое число), *name* – имя типа (строка символов), *A* – подмножество множества всевозможных атрибутов *Attributes*. Идентификатор необходим для описания атрибута, поскольку его имя может быть не уникальным.

Множество атрибутов *Attributes* есть множество кортежей вида <*id, name, type*>, где *id* и *name* – идентификатор и имя атрибута соответственно, *type* $\in B$ – тип атрибута, *B* – множество встроенных типов (например, *boolean, int, float, string,*типы массивов и т.д.)

Множество *EdgeTypes* есть набор кортежей вида <*id, name, A, directed, restricted, head, tail*>, где *id* – идентификатор типа; *name* – имя типа; $A \subseteq Attributes$; *directed* $\in$ *{true, false}* – флаг направленности ребра; *restricted* $\in$ *{true, false}* – булевский флаг определенности типов вершин, которые связывает ребро; *head* $\in$ *VertexTypes* – тип исходящей вершины ребра; *tail* $\in$ *VertexTypes* – тип входящей вершины ребра.

Для любого типа $T \in EdgeTypes$ если *restricted(T) = true*, то значения *head(T), tail(T)* определены; если же *restricted(T) = false*, то значения *head(T), tail(T)* не определены.

Произвольная *схема S* = <*VT(S), ET(S)*> обобщенной графовой модели включает два множества: множество типов вершин $VT(S) \subseteq VertexTypes$ и множество типов ребер $ET(S) \subseteq EdgeTypes$.

База данных (граф) *G*, удовлетворяющий схеме *S* имеет вид *G* = <*V, E*>, где

– $V = \{v \mid \exists T.(T \in VT(S) \ \& \ v: T)\}$ – множество вершин графа такое, что любая вершина имеет тип из *VT(S)*;

– $E = \{<e, t, h> \mid \exists T.(T \in ET(S) \ \& \ e: T \ \& \ t \in V \ \& \ t: tail(T) \ \& \ h \in V \ \& \ h: head(T))\}$ – множество ребер графа такое, что любое ребро имеет тип из *ET(S)* и соединяет вершины из *V*.

Типизация *x: T* означает, что для вершины (ребра) *x* могут быть определены атрибуты из *A(T)* (атрибуты типа *T*).

Рассмотрим пример схемы *SoilTaxonomy* базы данных таксономии почв из информационно-справочной системы по классификации почв России[52] в обобщенной

[52] http://infosoil.ru/

графовой модели. Для упрощения будем опускать в примерах идентификаторы типов и атрибутов, считая, что в данном примере имена однозначно идентифицируют типы:

```
VT(SoilTaxonomy) = { soilType, soilBranch }
ET(SoilTaxonomy) = { childOf }
A(soilType) = {<id, long>, <title, string>, <formula, string>}
A(soilBranch) = {<id, long>, <title, string>}
childOf = <"childOf", ∅, true, true, soilType, soilBranch>
```

Фрагмент схемы включает два типа вершин (*soilType, soilBranch*) и тип ребер (*childOf*). Тип *soilType* включает три атрибута (*id, title, formula*), *soilBranch*– два (*id, title*), *directs* – ни одного. Ребра типа *childOf* направлены от вершины типа *soilType* к вершине типа *soilBranch*.

Пример простого графа *g*, удовлетворяющего схеме *SoilTaxonomy* выглядит следующим образом:

```
g = <{t, b}, {c}>
t: soilType = <id: 1, title: "Буроземы", formula: "AY-BM-C">
b: soilBranch = <id: 2, title: "Структурно-метаморфические почвы">
c: childOf = <<>, t, b>
```

Вопрос выбора ЯМД для обобщенной графовой модели достаточно сложен. Графовые языки запросов развивались в течение многих лет вместе с самими графовыми моделями [226]. Существуют работы по анализу и сравнению выразительной силы и вычислительной сложности графовых ЯМД [227].

Однако, в современных популярных графовых СУБД языки манипулирования представлены в большинстве случаев просто прикладным интерфейсом пользователя (API), предоставляющим доступ к структуре графа, методы обхода графа и инкапсулирующим основные алгоритмы на графах. Не существует стандарта графового языка запросов.

В данной работе в качестве ЯМД модели атрибутированных графов выбран язык Cypher [225]. С одной стороны, этот язык поддерживается и развивается в одной из самых популярных графовых СУБД с открытым кодом – Neo4j. С другой стороны, язык является декларативным, в отличие от существующих графовых API или скриптовых языков (как Gremlin). Основные конструкции и ключевые слова имеют сходство с такими широко распространенными языками, как SQL и SPARQL.

2.8.2.2 Отображение языка определения данных графовой модели в каноническую

Схема в обобщенной графовой модели представляется в языке СИНТЕЗ в виде одноименного модуля (например, *SoilTaxonomy* из предыдущего пункта):

```
{ SoilTaxonomy; in: module;
  imports: PropertyGraph;
   …
}
```

Модуль *SoilTaxonomy* импортирует модуль *PropertyGraph*, представляющий собой расширение канонической модели данных, необходимое для унификации графовой модели, и включающий классы, содержащие вершины и ребра графов (*vertices* и *edges* соответственно):

```
{ PropertyGraph; in: module;
  { vertices; in: class; … },
  { edges: in: class; … };
}
```

Тип вершины (например, *soilType*) представляется в языке СИНТЕЗ одноименным классом (который объявляется подклассом класса всех вершин *vertices*), также входящим в модуль, соответствующий схеме:

```
{ soilType; in: class; superclass: vertices;
  instance_type: {
    id: long;
    title: string;
    formula: string;
  };
}
```

Атрибуты типа вершины, представляются в языке СИНТЕЗ атрибутами типа экземпляров (*instance_type*) соответствующего класса. Между встроенными типами графовой модели (*long, string, int* и т.д.) и встроенными типами языка СИНТЕЗ (*long, string, integer*) устанавливается взаимно-однозначное соответствие.

Тип ребра (например, *childOf*) также представляется одноименным классом (который объявляется подклассом класса всех вершин *edges*):

```
{ childOf; in: class; superclass: edges;
  instance_type: {
    metaframe
      directed: true;
      restricted: true;
      startVertexType: soilType;
      endVertexType: soilBranch;
    end
    edgeConstr: {in: invariant;
      {{ all e/childOf.inst (childOf(e) ->
            soilType(e.startVertex) & soilBranch(e.endVertex)) }}
     }; };
}
```

Атрибуты типа ребра, аналогично типу вершины, представляются в языке СИНТЕЗ атрибутами типа экземпляров соответствующего класса.

Заметим, что информация о направленности (*directed*), определенности ребра (*restricted*), типах его исходящей (*startVertexType*) и входящей (*endVertexType*) вершин представляется специальной конструкцией - метафреймом [42], связанным с типом экземпляра класса. Метафреймы в языке СИНТЕЗ предназначены для выражения дополнительной метаинформации, связанной с такими сущностями, как модули, типы, классы, функции. Этот метафрейм входит в расширение канонической модели данных, необходимое для унификации графовой модели.

Кроме того, ограничение на типы исходящей и входящей вершин представляется инвариантом *edgeConstr*, заданным формулой в типизированной логике первого порядка. Знак *all* означает квантор всеобщности, знак -> - логическую импликацию, & - конъюнкцию, выражение *x/T* – типизацию переменной *x* типом *T*, *C.inst* – тип экземпляров (instance) класса *C*. Предикат *C(x)*, где *C* – имя класса, обращается в истину на экземплярах класса *C*.

Заметим также, что на переменной *e* типа *childOf.inst* определены атрибуты исходящей вершины ребра *startVertex* и входящей вершины ребра *endVertex,* хотя их нет непосредственно в типе *childOf.inst.* Эти атрибуты являются общими для всех типов вершин и принадлежат типу экземпляров класса *edges*:

```
{ edges; in: class;
  instance_section: {
    startVertex: vertices.inst;
    endVertex: vertices.inst;
    isValidEdge: { in: predicate;
      params: {+stVtx/vertices.inst, +endVtx/vertices.inst
               returns/Boolean };
      {{ (stVtx = this.startVertex &
            endVtx = this.endVertex -> returns = true) &
           (stVtx <> this.startVertex |
            endVtx <> this.endVertex -> returns = false) }}
     };
}
```

Кроме упомянутых атрибутов, тип *edges.inst* включает метод-предикат *isValidEdge*. Предикат *e.isValidEdge(v1, v2)* обращается в истину, если исходящая вершина ребра *e* (*e.startVertex*) совпадает с *v1,* и входящая вершина ребра *e* (*e.endVertex*) совпадает с *v2*. Спецификация метода задается формулой первого порядка, связывающей входные и выходные параметры метода. Знак | означает дизъюнкцию, <> - неравенство, *this* – объект, для которого вызывается метод.

2.8.2.3 Отображение языка манипулирования данными

Язык запросов (программ) модели СИНТЕЗ представляет собой Datalog-подобный язык в объектной среде. Программа представляет собой набор конъюнктивных запросов (правил) вида

$q(x/T) :- C_1(x_1/T_1) \ \& \ \ldots \ \& \ C_n(x_n/T_n) \ \& \ F_1(X_1, Y_1) \ \& \ \ldots \ \& \ F_m(X_m, Y_m) \ \& \ B.$

Тело запроса представляет собой конъюнкцию предикатов-коллекций, функциональных предикатов и ограничения. Здесь C_i - имена коллекций (классов), F_j – имена функций, x_i – имена переменных, значения которых пробегают по классам, T_i – типы переменных, X_j и Y_j – входные и выходные параметры функций, B – ограничение, налагаемое на x_i, X_j, Y_j.

В дальнейшем будет использоваться запись предиката-коллекции вида *soilType([title, formula])*. Неформально это означает, что нас не интересуют объекты класса *soilType* целиком, а лишь их атрибуты *title, formula.* Формально запись означает сокращение от *soilType(_/soilType.inst[title, formula]).* Здесь знак _ обозначает анонимную переменную, *soilType.inst* – анонимный тип экземпляров (instance) класса *soilType*; *title, formula* – необходимые атрибуты типа экземпляров класса.

Рассмотрим отображение основных конструкций ЯМД на нескольких примерах.

Пример 1 (Конъюнктивный запрос с использованием предиката смежности вершин и ребер). Рассмотрим запрос, возвращающий типы почв, в названии которых содержится подстрока «Бурозем», которые относятся к одному отделу с типом почв «Буроземы» (иными словами, выясним, есть ли еще какие-то буроземы, кроме обычных):

```
return([res_title]) :-
soilType(burozem/[title]) &
soilBranch(br) &
soilType(res/[res_title: title]) &
childOf(c1) & childOf(c2) &
c1.isValidEdge(burozem, br) &
c2.isValidEdge(res, br) &
title = “Буроземы” &
res_title.like(“*Бурозем*”).
```

Запрос вернет непустой результат, если в графе базы данных существует такая вершина-отдел *br*, и такие ребра *c1, c2* типа *childOf,* что *c1* соединяет вершину *burozem* с вершиной *br*, и *c2* соединяет вершину *res* с вершиной *br.*

В языке Cypher такой запрос имеет вид

```
MATCH (burozem:soilType{title: 'Буроземы'})-[c1:childOf]->br,
      res-[c2:childOf]->br
WHERE res.title =~ /*Бурозем*/
RETURN res.title AS res_title
```

Каждый запрос языка Cypher представляет собой образец (pattern), по которому производится поиск в графе базы данных. В секции MATCH указывается образец поиска в графе. В данном случае это указание, что следует искать отдел, к которому относится (*childOf*) тип почв *Буроземы* (вершина *burozem*), а также другие типы почв, относящиеся к этому же отделу. В секции WHERE указывается фильтр поиска. В данном случае это подстрока «Бурозем». В секции RESULT указываются возвращаемые значения. В данном случае это полное имя родственного типа почв.

Основные принципы отображения конъюнктивных запросов объектной модели в язык Cypher, проиллюстрированные на данном примере, состоят в следующем:

– конъюнктивный запрос представляется в языке Cypher запросом, возвращающим результат (секция RETURN);

– предикаты-коллекции и предикат смежности вершин и ребер представляются образцами секции MATCH. Каждому предикату смежности соответствует свой образец. Переменные, типизированные в предикатах-коллекциях, представляются одноименными переменными, использующимися в образцах;

– предикаты-условия представляются соответствующими предикатами секции WHERE;

– атрибуты результирующего предиката конъюнктивного запроса представляются одноименными атрибутами в секции RETURN.

Пример 2 (Удаление вершин). Рассмотрим запрос, удаляющий из базы данных тип почв «Солончаки тёмные» (этот тип был удален из классификации в 2008 г.):

```
-soilType(sol) :- soilType(sol) & sol.title = "Солончаки тёмные".
```

В правилах со знаком «–» в голове осуществляется удаление объектов из коллекции.

В языке Cypher такой запрос представляется запросом с секцией DELETE:

```
MATCH (sol:soilType{title: 'Солончаки тёмные'})
DELETE sol
```

Пример 3 (Обновление значения атрибута). Рассмотрим запрос, устанавливающий формулу типа почв «Подбуры»:

```
return(podbur/[title, formula]):-
soilType(podbur) &
podbur.title = "Подбуры" &
podbur.set_formula("O-BHF-C").
```

В языке Cypher такой запрос представляется запросом с секцией SET:

```
MATCH (podbur:soilType{title: 'Подбуры'})
SET podbur.formula = 'O-BHF-C'
```

```
RETURN podbur.title, podbur.formula
```

2.8.2.4 Представление семантики подмножества объектной канонической модели в AMN

Спецификация, выражающая семантику *подмножества объектной канонической модели*, представляется в языке AMN конструкцией REFINEMENT:

```
REFINEMENT ObjectDM
```

Переменные, составляющие пространство состояний объектной модели, объявлены в разделе ABSTRACT_VARIABLES машины *ObjectDM* и типизируются в разделе INVARIANT:

```
ABSTRACT_VARIABLES
    typeNames, classNames, attributeNames,
    instanceType, typeAttributes, attributeType,
    unique, obligatory,
    intAttributeLowerBound, intAttributeHigherBound,
    objectIDs, objectType, objectsOfClass,
    integerAttributeValue,
    adtAttributeValue
INVARIANT …
```

Раздел INVARIANT содержит формулу, которая состоит из предикатов, типизирующих переменные состояния и налагающих различные совместные ограничения на переменные. Предикаты соединяются операцией конъюнкции.

Так, имена типов и классов представлены переменными *typeNames* и *classNames,* тип которых – подмножество множества строк (*STRING_Type*):

```
typeNames: POW(STRING_Type) &
classNames: POW(STRING_Type)
```

Здесь *POW* – конструктор множества подмножеств.

Атрибуты (переменная *attributeNames*) представлены частичной функцией (знак +->), ставящей в соответствие уникальному идентификатору атрибута (натуральному числу из множества *NAT*) имя атрибута (строку):

```
attributeNames: NAT +-> STRING_Type
```

Типы экземпляров классов (переменная *instanceType*) представлены тотальной функцией (знак -->) из множества имен классов в множество имен типов:

```
instanceType: classNames --> typeNames
```

Принадлежность атрибутов типам (переменная *typeAttributes*) выражена тотальной функцией из множества имен типов в множество подмножеств атрибутов:

```
typeAttributes: typeNames --> POW(dom(attributeNames))
```

Здесь *dom* – операция, возвращающая область определения функции, а *dom(attributeNames)* – множество имен атрибутов.

Типы значений атрибутов (переменная *attributeType*) представлены функцией из множества атрибутов в множество идентификаторов встроенных типов данных *BuiltInTypes*:

```
attributeType: dom(attributeNames) +-> BuiltInTypes
```

Множества уникальных атрибутов типов *unique* и множества определенных атрибутов типов *obligatory* представлены тотальными функциями из множества имен типов в множество подмножеств атрибутов:

```
unique: typeNames --> POW(dom(attributeNames)) &
obligatory: typeNames --> POW(dom(attributeNames))
```

Нижние границы целочисленных атрибутов (переменная *intAttributeLowerBound*) представлены частичной функцией из множества атрибутов в множество целых чисел:

```
intAttributeLowerBound: dom(attributeNames) +-> INT
```

Аналогично представляются верхние границы.

Идентификаторы объектов (переменная *objectIDs*) представлены подмножеством натуральных чисел:

```
objectIDs: POW(NAT)
```

Типы объектов (переменная *objectType*) представлены тотальной функцией из множества объектных идентификаторов в множество имен типов:

```
objectType: objectIDs --> typeNames
```

Состав классов (переменная *objectsOfClass*) представлен тотальной функцией из множества имен классов в множество подмножеств идентификаторов объектов:

```
objectsOfClass: classNames --> POW(objectIDs)
```

Значения атрибутов объектов (переменные *integerAttributeValue, adtAttributeValue* и т. д.) представлены функциями из множества атрибутов в функции из множества идентификаторов объектов в множества значений атрибутов. Для простоты рассмотрены лишь функции для целочисленных атрибутов и атрибутов со значениями АТД (объектами):

```
integerAttributeValue:
    dom(attributeNames) +-> (objectIDs +-> INT) &
adtAttributeValue:
    dom(attributeNames) +-> (objectIDs +-> NAT)
```

Дополнительные необходимые свойства переменных состояния представлены конъюнктивными компонентами инварианта. Так, каждый атрибут является атрибутом некоторого типа, и никакой атрибут не принадлежит двум типам одновременно:

```
UNION(tt).(tt: typeNames | typeAttributes(tt)) = dom(attributeNames)
   !(t1, t2).(t1: typeNames & t2: typeNames =>
      (typeAttributes(t1) /\ typeAttributes(t2) = {}))
```

Здесь *UNION* – родовая операция объединения, в данном случае объединяются множества атрибутов *typeAttributes(tt)* по всем именам типов *tt* из множества *typeNames*; «!» – знак квантора всеобщности, «=>» – логическая импликация, «/\» – символ пересечения множеств, «{}» – пустое множество.

Уникальные и определенные атрибуты типа выбираются из множества атрибутов типа:

```
!(tt).(tt: dom(unique) => unique(tt) <: typeAttributes(tt)) &
!(tt).(tt: dom(obligatory) => obligatory(tt) <: typeAttributes(tt))
```

Здесь «<:» – символ отношения множество – подмножество.

Нижние и верхние границы могут быть определены только для целочисленных атрибутов:

```
!(attr).(attr: dom(intAttributeLowerBound) =>
    attributeType(attr) = Integer)
```

Тип объекта – экземпляра класса есть тип экземпляров этого класса:

```
!(cc).(cc: classNames =>
    !(oo).(oo: objectsOfClass(cc) =>
       objectType(oo) = instanceType(cc)))
```

Для каждого атрибута определена ровно одна функция значений:

```
dom(adtAttributeValue) /\ dom(integerAttributeValue) = {} &
dom(adtAttributeValue) \/ dom(integerAttributeValue) = dom(attributeNames)
```

Здесь «\/» – символ объединения множеств.

Для любого объекта и любого определенного атрибута типа этого объекта функция значений атрибута определена на объекте:

```
!(oo, aa).(oo: dom(objectType) &
    aa: typeAttributes(objectType(oo)) &
    aa: obligatory(objectType(oo)) =>
        (attributeType(aa) = Integer =>
            oo: dom(integerAttributeValue(aa))) &
        (attributeType(aa) = ADT =>
            oo: dom(adtAttributeValue(aa)))  )
```

Для любого объекта и любого целочисленного атрибута типа объекта, определенного на объекте и для которого определена нижняя (верхняя) граница, значение атрибута не меньше (не больше) нижней (верхней) границы:

```
!(oo, aa).(oo: objectIDs &
    aa: typeAttributes(objectType(oo)) &
    oo: dom(integerAttributeValue(aa) =>
    (aa: dom(intAttributeLowerBound) =>
        (integerAttributeValue(aa)(oo) >=
         intAttributeLowerBound(aa))) )
```

Объект однозначно идентифицируется набором своих уникальных атрибутов:

```
!(oo1, oo2).(oo1: objectIDs & oo2: objectIDs &
    objectType(oo1) = objectType(oo2) &
    unique(objectType(oo1)) /= {} &
    !(aa).(aa: unique(objectType(oo1)) =>
        (attributeType(aa) = Integer =>
              integerAttributeValue(aa)(oo1) =
                   integerAttributeValue(aa)(oo2)) &
        (attributeType(aa) = ADT =>
               adtAttributeValue(aa)(oo1) =
                    adtAttributeValue(aa)(oo2)) ) =>
    oo1 = oo2 )
```

Семантика конструкций расширения канонической модели данных, необходимого для унификации графовой модели, представляется в AMN следующим образом.

Константы, необходимые для унификации графовой модели, объявлены в разделе CONSTANTS машины *ObjectDM* и типизируются в разделе PROPERTIES:

```
CONSTANTS
  c_edges, c_vertices,
  a_startVertex, a_endVertex,
  c_edges_instance_type
PROPERTIES …
```

Раздел PROPERTIES содержит формулу, которая состоит из предикатов, типизирующих константы. Предикаты соединяются операцией конъюнкции. Так, имена классов ребер и вершин представлены константами *c_edges* и *c_vertices,* тип которых – подмножество множества строк (*STRING_Type*):

```
c_edges: STRING_Type &
c_vertices: STRING_Type
```

Знак типизации «:» формально означает принадлежность элемента множеству.

Имя типа экземпляров класса ребер представлено константой *c_edges_instance_type*:

```
c_edges_instance_type: STRING_Type
```

Идентификаторы атрибутов этого типа, соответствующих исходящей и входящей вершинам ребра, представляются константами *a_startVertex, a_endVertex*, тип которых – натуральное число (*NAT*):

```
a_startVertex: NAT &
a_endVertex: NAT
```

Переменные, составляющие пространство состояний объектной модели, объявлены в разделе ABSTRACT_VARIABLES машины *ObjectDM* и типизируются в разделе INVARIANT:

```
ABSTRACT_VARIABLES
  m_directed, m_restricted,
  m_startVertexType,   m_endVertexType,
  isValidEdge
INVARIANT …
```

Раздел INVARIANT содержит формулу, которая состоит из предикатов, типизирующих переменные состояния, и налагающих различные совместные ограничения на переменные и константы. Предикаты соединяются операцией конъюнкции.

Так, декларируется, что *c_edges* и *c_vertices* действительно являются именами классов:

```
c_edges: classNames &
c_vertices: classNames
```

Здесь *classNames* – множество, содержащее имена всех классов базы данных.

Метаинформация, связанная с типом экземпляров класса ребер, представлена переменными *m_directed* (направленность ребра), *m_restricted* (определенность типов вершин ребра), *m_startVertexType* (тип исходящей вершины), *m_endVertexType* (тип входящей вершины):

```
m_directed: subclasses(c_edges) --> BOOL &
m_restricted: subclasses(c_edges) --> BOOL &
m_startVertexType:
  subclasses(c_edges) --> subclasses(c_vertices) &
m_endVertexType:
  subclasses(c_edges) --> subclasses(c_vertices)
```

Переменные типизированы полными функциями (знак -->), определенными на множестве всех классов ребер (которые являются подклассами класса всех вершин *c_edges*). Функция *subclasses* ставит в соответствие классу множество его подклассов.

Декларируется, что *c_edges_instance_type* действительно является именем типа, а атрибуты *a_startVertex* и *a_endVertex* являются атрибутами этого типа. Декларируется также, что тип значений данных атрибутов – абстрактный тип данных (*ADT*):

```
c_edges_instance_type: typeNames &
a_startVertex: typeAttributes(c_edges_instance_type) &
a_endVertex: typeAttributes(c_edges_instance_type) &
attributeType(a_startVertex) = ADT &
attributeType(a_endVertex) = ADT
```

Здесь функция *typeAttributes* возвращает множество атрибутов типа, функция *attributeType* – тип значений атрибута.

Предикат смежности вершин и ребер представляется функцией *isValidEdge*, сопоставляющей ребру *edg* и двум вершинам v_1, v_2 значение *истина*, если вершины v_1, v_2 соединены ребром *edg*:

```
isValidEdge: objectsOfClass(c_edges)*
  objectsOfClass(c_vertices) * objectsOfClass(c_vertices) --> BOOL
!(edg, v1, v2).(edg: objectsOfClass(c_edges) &
  v1: objectsOfClass(c_vertices) &
  v2: objectsOfClass(c_vertices) =>
  ((isValidEdge(edg, v1, v2) = TRUE) <=>
  (adtAttributeValue(a_startVertex)(edg) = v1 &
  adtAttributeValue(a_endVertex)(edg) = v2) ) )
```

Здесь * - знак декартова произведения множеств. Функция *adtAttributeValue(a)(o)* возвращает значение атрибута *a* объекта *o*.

Дополнительные необходимые свойства переменных состояния представлены конъюнктивными компонентами инварианта. Так, ребро обязательно связывает два объекта из класса *c_vertices* (вершины):

```
!edg.(edg: objectsOfClass(c_edges) =>
  adtAttributeValue(a_startVertex)(edg): objectsOfClass(c_vertices) &
  adtAttributeValue(a_endVertex)(edg): objectsOfClass(c_vertices) )
```

Здесь «!» – знак квантора всеобщности, «=>» – логическая импликация. Функция *objectsOfClass* возвращает множество объектов – экземпляров класса.

Если типы вершин, соединяемых ребром, определены, то они должны принадлежать классам, задаваемым метаатрибутами *startVertexType* и *endVertexType* класса ребра:

```
!(cls, edg).(cls: subclasses(c_edges) &
  edg: objectsOfClass(cls) =>
  (m_restricted(cls) = TRUE =>
  adtAttributeValue(a_startVertex)(edg):
      objectsOfClass(m_startVertexType(cls)) &
  adtAttributeValue(a_endVertex)(edg):
      objectsOfClass(m_endVertexType(cls)) ) )
```

Чтобы не загромождать рассуждения, из всего ЯМД рассмотрена единственная операция *deleteVertex* удаления вершины:

```
OPERATIONS
deleteVertex(attr, cond) =
PRE attr : dom(attributeNames) &
```

```
  cond : INT --> BOOL &
  attributeType(attr) = Integer
THEN
  objectsOfClass(c_vertices) :=
  objectsOfClass(c_vertices) -
  { vert | vert: objectsOfClass(c_vertices) &
    vert: dom(adtAttributeValue(attr)) &
    cond(integerAttributeValue(attr)(vert)) = TRUE }
END
```

Параметрами операции являются идентификатор целочисленного атрибута *attr* и функция *cond*, отвечающая условию на значение атрибута. Операция *deleteVertex* удаляет из класса *c_vertices* все такие вершины *vert*, что на *vert* определен атрибут *attr*, и для значения этого атрибута выполнено условие *cond*. Здесь знак «:=» означает присваивание, знак «-» - разность множеств; конструкция *{v | F(v)}* – выделение множества таких значений *v*, что предикат *F(v)* обращается в истину, функция *integerAttributeValue(a)(o)* возвращает значение целочисленного атрибута *a* объекта *o*.

Спецификация REFINEMENT, образованная расширением спецификации *ObjectDM* конструкциями, необходимыми для представления семантики графовой модели, получила название *ObjectDM_GraphExt.*

2.8.2.5 Представление семантики графовой модели в AMN

Спецификация, выражающая семантику синтетической графовой модели, представляется в языке AMN конструкцией

```
REFINEMENT GraphDM
```

Переменные, составляющие пространство состояний объектной модели, объявлены в разделе ABSTRACT_VARIABLES машины *GraphDM*:

```
ABSTRACT_VARIABLES
  vertexTypeIDs, edgeTypeIDs, attributeIDs,
  typeName, attributes, attributeName, attributeTyping,
  directed, restricted, headType, tailType,
  vertices, vertixType, edges, edgeType,
  headVertix, tailVertix,
  g_integerAttributeValue
```

Идентификаторы типов вершин представлены переменной *vertexTypeIDs*; идентификаторы типов ребер - переменной *edgeTypeIDs*; идентификаторы атрибутов – переменной *attributeIDs*; имена типов - переменной *typeName*; принадлежность атрибутов типам – переменной *attributes*; имена атрибутов – переменной *attributeName*; типы значений атрибутов – переменной *attributeTyping*; направленность ребер - переменной *directed*; определенность типов исходящей и входящей вершиной ребра - переменными *restricted*, *headType, tailType*; вершины и ребра, составляющие базу данных - переменными *vertices, edges*; типы конкретных вершин и ребер - переменными

vertixType, edgeType; исходящая и входящая вершины конкретных ребер - переменными *headVertix, tailVertix;* значения целочисленных атрибутов - переменной *g_integerAttributeValue.* Функции, представляющие значения атрибутов других типов (например, *BOOL* или *STRING*), определяются аналогично.

Переменные типизируются в разделе INVARIANT при помощи частичных (знак «+->») и тотальных функций аналогично переменным, использующимся для придания семантики объектной модели:

```
INVARIANT
  vertexTypeIDs: POW(NAT) &
  edgeTypeIDs: POW(NAT) &
  attributeIDs: POW(NAT) &
  typeName: vertexTypeIDs \/ edgeTypeIDs --> STRING_Type &
  attributes: vertexTypeIDs \/ edgeTypeIDs --> POW(attributeIDs) &
  directed: edgeTypeIDs --> BOOL &
  restricted: edgeTypeIDs --> BOOL &
  headType: edgeTypeIDs +-> vertexTypeIDs &
  tailType: edgeTypeIDs +-> vertexTypeIDs &
  attributeName: attributeIDs --> STRING_Type &
  attributeTyping: attributeIDs --> BuiltInTypes &
  vertices: POW(NAT) &
  vertixType: vertices --> vertexTypeIDs &
  edges: POW(NAT) &
  edgeType: edges --> edgeTypeIDs &
  headVertix: edges --> vertices &
  tailVertix: edges --> vertices &
  g_integerAttributeValue: (vertices \/ edges)*attributeIDs +-> INT
```

Здесь знак «\/» означает объединение множеств.

Дополнительные необходимые свойства переменных состояния представлены конъюнктивными компонентами инварианта. Так, для тех типов, метаатрибут *restricted* которых принимает значение *TRUE*, заданы типы исходящей и входящей вершин:

```
!(type).(type: edgeTypeIDs =>
  (restricted(type) = TRUE => type: dom(headType) & type: dom(tailType)) &
  (restricted(type) = FALSE => type /: dom(headType) & type/: dom(tailType)))
```

Здесь функция *dom* возвращает область определения функции, знак «/:» означает непринадлежность элемента множеству.

Функция *g_integerAttributeValue* определена только для целочисленных атрибутов. Атрибут, для которого определена эта функция, принадлежит типу соответствующей вершины или ребра:

```
!(vert, attr).(vert: vertices & attr: attributeIDs =>
  ((vert |-> attr) : dom(g_integerAttributeValue) =>
      attributeTyping(attr) = Integer) &
      attr: attributes(vertixType(vert))  ) &
!(edg, attr).(edg: edges & attr: attributeIDs =>
  ((edg |-> attr) : dom(g_integerAttributeValue) =>
    attributeTyping(attr) = Integer) &
```

```
    attr: attributes(edgeType(edg))  )
```

Если типы вершин, соединяемых ребром, определены (значение метаатрибута *restricted* типа этого ребра принимает значение *TRUE*), то они должны принадлежать типам, задаваемым метаатрибутами *headType* и *tailType* типа ребра:

```
!edg.(edg: edges =>
  (restricted(edg) = TRUE =>
      vertixType(headVertix(edg)) = headType(edgeType(edg)) &
      vertixType(tailVertix(edg)) = tailType(edgeType(edg)) ) )
```

Аналогично объектной модели рассмотрена единственная операция ЯМД – операция удаления вершины *deleteVertex*:

```
OPERATIONS
  deleteVertex(attr, cond) =
PRE attr: attributeIDs & cond: INT --> BOOL &
    attributeTyping(attr) = Integer
THEN
  vertices := vertices -
  {vert | vert: vertices & attr: attributes(vertixType(vert)) &
          cond(g_integerAttributeValue(vert, attr)) = TRUE }
END
```

Сигнатура операции совпадает с сигнатурой операции объектной модели. Семантика операции также аналогична: вершина *vert* удаляется из базы данных (множества *vertices*), если на *vert* определен атрибут *attr*, и для значения этого атрибута выполнено условие *cond.*

2.8.2.6 Инвариант уточнения и доказательство уточнения спецификаций

Для формального доказательства того, что спецификация *GraphDM* уточняет спецификацию *ObjectDM,* необходимо построить *инвариант уточнения*, связывающий переменные машин и добавить его к инварианту уточняющей машины.

Инвариант формализует принципы отображения ЯОД, и объединяет их в одну конъюнкцию.

Множество имен типов графовой модели совпадает с множеством имен классов объектной модели (за исключением предопределенных классов *c_edges, c_vertices*):

```
ran(typeName) = classNames - {c_edges, c_vertices}
```

Множество атрибутов типов графовой модели соответствует множеству атрибутов объектной модели (за исключением предопределенных атрибутов *a_startVertex, a_endVertex*):

```
attributeIDs = dom(attributeNames) - {a_startVertex, a_endVertex}
```

Имена и типы атрибутов графовой и объектной модели совпадают:

```
!attr.(attr: attributeIDs =>
  attributeName(attr) = attributeNames(attr) &
  attributeTyping(attr) = attributeType(attr)  )
```

Вершины и ребра графовой базы данных соответствуют объектам классов *c_vertices* и c*_edges*:

```
vertices = objectsOfClass(c_vertices) &
edges = objectsOfClass(c_edges)
!vert.(vert: vertices =>
 ((vert: objectsOfClass(typeName(vertixType(vert)))) <=>
 (vert: vertices))  ) &
!edg.(edg: edges =>
  ((edg: objectsOfClass(typeName(edgeType(edg)))) <=>
  (edg: edges))  )
```

Значения атрибутов вершин и ребер графовой модели совпадают со значениями соответствующих атрибутов соответствующих объектов:

```
!(vert, attr).(vert: vertices & attr: attributeIDs =>
  ((vert |-> attr) : dom(g_integerAttributeValue) =>
      g_integerAttributeValue(vert, attr) =
          integerAttributeValue(attr)(vert))    )
!(edg, attr).(edg: edges & attr: attributeIDs =>
  ((edg |-> attr) : dom(g_integerAttributeValue) =>
      g_integerAttributeValue(edg, attr) =
          integerAttributeValue(attr)(edg))   )
```

Для указания того, что машина *GraphDM* уточняет машину *ObjectDM*, в машину *GraphDM* была добавлена директива

```
REFINES ObjectDM_GraphExt
```

Спецификации *ObjectDM_GraphExt* и *GraphDM* вместе с инвариантом уточнения[53] были загружены в инструментальное средство Atelier B 4.7.1. Автоматически были сгенерированы теоремы, выражающие уточнение спецификаций (таблица 2.5). В частности, для спецификации *GraphDM* были сгенерированы 173 теоремы, 171 из них была доказана автоматически.

Таблица 2.5 - Количество сгенерированных и автоматически доказанных теорем непротиворечивости и уточнения спецификаций

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic		
		Force Fast	Force 1	Force 3
ObjectDM_GraphExt	222	49	188	197
GraphDM	173	47	171	171

[53] AMN-спецификации размещены в репозитории GitHub: https://github.com/sstupnikov/ModelTransformation/tree/master/GraphDM2ObjectModel/semantics

2.8.3 Верификация на основе эквивалентного представления в единой семантической структуре

Метод, рассматриваемый в данном разделе, предназначен для верификации корректности *взаимного* отображения моделей данных (подмножеств моделей данных), в том случае, когда одна из моделей имеет строго определенную нормативную семантику в некоторой семантической структуре, а другая модель допускает определение своей семантики в той же семантической структуре. *Взаимным* называется такое отображение моделей данных, которое определено как множество взаимооднозначных соответствий между элементами моделей.

Метод состоит из следующих этапов:

– определение отображения моделей данных (подмножеств моделей данных) как множества взаимооднозначных соответствий между элементами моделей;

– выбор нормативного документа с описанием семантики одной из моделей в некоторой семантической структуре;

– определение семантики второй модели в той же семантической структуре;

– доказательство эквивалентности семантики элементов моделей, связанных отображением.

В случае, если доказательство определено для всех элементов моделей данных (подмножеств моделей данных), связанных отображением, взаимное отображение моделей считается корректным.

Применение метода рассматривается ниже на примере языка определения данных - подмножества модели OWL 2 QL, а также на примере языка манипулирования данными – подмножества языка на правилах RIF-BLD.

2.8.3.1 Верификация взаимного отображения канонической модели данных и языка OWL 2

Раздел основан на работах автора [86] [106].

Определение отображения между канонической информационной моделью и языком OWL 2 демонстрируется ниже на конкретных примерах элементов моделей. В данном разделе приведены соответствия лишь для некоторых основных элементов моделей, остальные соответствия приведены в приложении.

Отображение онтологий. Онтология, представляемая в языке СИНТЕЗ онтологическим модулем (например, *SemanticSensorNetwork* - описание сенсоров и их наблюдений [228], понятия этой онтологии используются в онтологии Глобальной

почвенной информационной системы GloSIS [229]), отображается в одноименную онтологию OWL:

СИНТЕЗ	OWL

`{ SemanticSensorNetwork;`
` in: module, ontology;`
` …`
`}`
	`Ontology( SemanticSensorNetwork`
	` …`
	`)`

Заметим, что конструкции OWL здесь представлены в *функциональном синтаксисе* [11], позволяющем компактно описывать структуру онтологий.

Отображение классов. Класс (например, *observation*) и тип его экземпляров (*Observation*) языка СИНТЕЗ представляются в OWL единой конструкцией класса:

СИНТЕЗ	OWL

`{ observation; in: class;`
` instance_section: Observation;`
` …`
`}`
`{ Observation; in: type; … }`
	`Class( observation )`

Отображение свойств объектов. Свойства объектов представляются в языке СИНТЕЗ при помощи метаклассов ассоциаций (например, *madeBySensor*), задающих связь между объектами из области определения ассоциации (класс *observation*) и объектами из области значения ассоциации (класс *sensor*):

СИНТЕЗ	OWL

`{ MadeBySensor;`
` in: association, metaclass;`
` instance_section: {`
` domain: observation;`
` range: sensor;`
` };`
`}`
`{ Observation; in: type;`
`  madeBySensor: Sensor;`
`  metaslot`
`  in: MadeBySensor;`
`  end`
 	
` resultTime: time;`
`}`
	`ObjectProperty(madeBySensor)`
	`ObjectPropertyDomain(`
	`  madeBySensor  observation)`
	`ObjectPropertyRange(`
	`  madeBySensor  sensor)`
	`SubClassOf(observation`
	`  ObjectExactCardinality(1  resultTime))`
	
	`DataProperty(resultTime)`
	`DataPropertyDomain(`
	`  resultTime  observation)`
	`DataPropertyRange(`
	`  resultTime  xsd:dateTime)`
	`SubClassOf(observation`
	`  DataExactCardinality (1  resultTime) )`

Атрибут АТД (например, *worksOn* типа *Employee*) может быть объявлен принадлежащим некоторому метаклассу ассоциаций. Такие свойства представляются в OWL конструкцией *ObjectProperty* (если область значения ассоциации – коллекция объектов) или *DataProperty* (если область значения ассоциации – множество значений встроенного типа *real, string* и т.д., например, атрибут *salary* типа *Empolyee*). Атрибуты, тип которых не является типом множества (например, *salary*), имеют в языке СИНТЕЗ кардинальность *1* по умолчанию. В OWL кардинальность атрибута указывается явно (в случае атрибута *salary* при помощи конструкции *DataExactCardinality*):

Нормативный документ с описанием семантики для языка OWL 2 – это рекомендация W3C “OWL 2 Web Ontology Language Direct Semantics” [230]. Семантика модели определяется при помощи семантической структуры - интерпретации $I = (\Delta_I, \Delta_D, \cdot^C, \cdot^{OP}, \cdot^{DP}, \cdot^I, \cdot^{DT}, \cdot^{LT})$, которая представляет собой кортеж из следующих компонентов:

– Δ_I – объектный домен;

– Δ_D – домен типов данных;

– $\cdot^C$ – функция интерпретации классов; для каждого класса Cl выполнено $(Cl)^C \subseteq \Delta_I$;

– $\cdot^{OP}$ – функция интерпретации свойств объектов, область значения которых – коллекция объектов; для каждого свойства op выполнено $(op)^{OP} \subseteq \Delta_I \times \Delta_I$;

– $\cdot^{DP}$ – функция интерпретации свойств объектов, область значения которых – множество значений встроенного типа данных; для каждого свойства dp выполнено $(dp)^{DP} \subseteq \Delta_I \times \Delta_D$;

– $\cdot^I$ – функция интерпретации объектов; для каждого объекта o выполнено $(o)^I \in \Delta_I$;

– $\cdot^{DT}$ – функция интерпретации встроенных типов данных; для каждого типа dt выполнено $(dt)^{DT} \subseteq \Delta_D$;

– $\cdot^{LT}$ – функция интерпретации литералов; для каждого литерала l типа данных dt выполнено $(l)^{LT} \in (dt)^{DT}$.

Определим, что интерпретация I *удовлетворяет* онтологическому модулю O (является *моделью* O), если все конструкции онтологии удовлетворяют ряду условий (выраженных в виде формул логики предикатов первого порядка), соответствующих различным элементам модели. Тем самым будет определена семантика языка СИНТЕЗ при помощи семантической структуры I.

Так, *условия для свойств объектов* состоят в следующем. Когда класс (*employee*) связывается со своим типом экземпляров (*Employee*), на область определения, область значения и кардинальность свойств, составляющих тип, накладываются соответствующие ограничения:

Конструкция СИНТЕЗ	Условие

`{ observation; in: class;`
`instance_section: Observation;`
`}`
 	
`{ Observation; in: type;`
`madeBySensor: Sensor;`
`metaslot`
`in: MadeBySensor;`
`end`
`resultTime: time;`
`}`
`{ MadeBySensor;`
`in: association, metaclass;`
`instance_section: {`
`domain: observation;`
`range: sensor;`
`};`
`}`
	$\forall e, s$ (
	$(e, s) \in (resultTime)^{DP} \rightarrow$
	$e \in (observation)^{C} \wedge$
	$s \in (time)^{DT}) \wedge$
	$\forall e, s1, s2$ (
	$(e, s1) \in (resultTime)^{DP} \wedge$
	$(e, s2) \in (resultTime)^{DP} \rightarrow$
	$s1 = s2$)
	$\forall e, s1, s2$ (
	$(e, s1) \in (madeBySensor)^{OP} \wedge$
	$(e, s2) \in (madeBySensor)^{OP} \rightarrow$
	$s1 = s2$)
	
	$\forall e, p$ (
	$(e, p) \in (madeBySensor)^{OP} \rightarrow$
	$e \in (observation)^{C} \wedge$
	$p \in (sensor)^{C}$)

Остальные условия приведены в приложении.

Определив семантику онтологий канонической модели при помощи интерпретации I, можно также убедиться, что основные задачи вывода (непротиворечивость онтологии, поглощение понятий и т.д.), сводимые друг к другу в дескриптивных логиках, сводимы друг к другу и в канонической модели. Действительно, доказательства сводимости для дескриптивных логик легко преобразуются в доказательства сводимости для канонической модели. В качестве примера рассмотрим доказательство сводимости задачи поглощения к задаче отсутствия модели (unsatisfiability).

Утверждение. В канонической модели с интерпретацией I задача поглощения классов сводится к задаче отсутствия модели.

Доказательство. Рассмотрим онтологию O и задачу поглощения для классов $C_1 \subseteq C_2$ из O. По определению поглощения, $C_1 \subseteq C_2$ iff для любой модели $I = \langle \Delta_I, \Delta_D, \bullet^C, \bullet^{OP}, \bullet^{DP}, \bullet^I, \bullet^{DT}, \bullet^{LT} \rangle$ онтологии O выполнено $(C_1)^C \subseteq (C_2)^C$. Рассмотрим задачу отрицания поглощения: пусть существует такая модель I и такой объект a, что $a \in (C_1)^C$ и $a \notin (C_2)^C$. Рассмотрим онтологию O_1, полученную из O добавлением класса A и утверждения, что объект с именем id_a входит в A:

```
{ A; in: class;
 superclass: C1;
 instance_section: {
  inv: { in: invariant;
   { A <= differ(thing, C2) }
  };
}
{ ida; in: A; }
```

Тогда очевидно, что интерпретация I, расширенная утверждением $(id_a)^I = a$, является моделью O_1, т.е. из отрицания поглощения $C_1 \subseteq C_2$ следует существование модели для O_1. Следовательно, из отсутствия модели для O_1 следует поглощение $C_1 \subseteq C_2$. Таким образом, задача поглощения сведена к задаче отсутствия модели. ■

Аналогичным образом для канонической модели адаптируются и другие сводимости.

Теперь *определим формально корректность отображения* между СИНТЕЗ и OWL. Обозначим чрез Σ: $S_{SYN} \times S_{OWL}$ некоторое взаимное отображение (являющееся формально отношением) множества S_{SYN} схем, выраженных в канонической модели и S_{OWL} – множества онтологий, выраженных в языке OWL 2 (примером такого отображения является отображение σ, определенное выше). Онтологии $s \in S_{SYN}$ и $o \in S_{OWL}$ отображаются друг в друга тогда и только тогда, когда $\langle s, o \rangle \in \Sigma$.

Назовем отображение Σ *корректным*, если для любых двух онтологий $s \in S_{SYN}$ и $o \in S_{OWL}$ таких, что $\langle s, o \rangle \in \Sigma$ следующие условия эквивалентны:

- существует модель M_S для онтологии s;
- существует модель M_O для онтологии o.

Выше показано, как онтологии языка СИНТЕЗ интерпретируются при помощи семантических структур, используемых при интерпретации языка OWL 2. Таким образом, если рассмотреть для онтологий $s \in S_{SYN}$ и $o \in S_{OWL}$, связанных отображением σ, одну и ту же интерпретацию I, то для доказательства корректности отображения σ остается показать лишь, что I удовлетворяет s тогда и только тогда, когда I удовлетворяет o. Для этого нужно показать, что условия удовлетворения интерпретации онтологии канонической модели, эквивалентны условиям удовлетворения интерпретации онтологии OWL [230]. В общем случае доказательство осуществляется индукцией по построению онтологии (по классам, свойствам, фактам, инвариантам и т.д.). Ниже в данном разделе (и приложении) эквивалентность условий продемонстрирована на примерах, иллюстрирующих основные закономерности доказательства. В левой части таблиц приводятся условия для конструкций СИНТЕЗ, в правой части таблиц приводятся условия для соответствующих конструкций OWL. В условиях для OWL сохранены обозначения, используемые в [230].

Условия для *свойств объектов* выглядят следующим образом:

Условия для СИНТЕЗ	Условия для OWL
$\forall e, s\ (\ (e, s) \in (resultTime)^{DP} \rightarrow$ $e \in (observation)^{C} \wedge s \in (time)^{DT})$	$\forall x, y\ (\ (x, y) \in (resultTime)^{DP} \rightarrow\ x \in (observation)^{C}\ )$ $\forall x, y\ (\ (x, y) \in (resultTime)^{DP} \rightarrow y \in (xsd{:}dateTime)^{DT})$
$\forall e, s1, s2\ (\ (e, s1) \in (resultTime)^{DP} \wedge$ $(e, s2) \in (resultTime)^{DP} \rightarrow\ s1 = s2\ )$	$(observation)^{C} \subseteq \{\ x \mid \#\{\ y \mid (x\ ,\ y) \in (resultTime)^{DP}\ \} = 1\ \}$
$\forall e, s1, s2\ (\ (e, s1) \in (madeBySensor)^{OP} \wedge$ $(e, s2) \in (madeBySensor)^{OP} \rightarrow s1 = s2\ )$	$(observation)^{C} \subseteq \{\ x \mid \#\{\ y \mid (x\ ,\ y) \in (madeBySensor)^{OP}\ \} = 1\ \}$
$\forall e, p\ (\ (e, p) \in (madeBySensor)^{OP} \rightarrow$ $e \in (observation)^{C} \wedge p \in (sensor)^{C}\ )$	$\forall x, y\ (\ (x, y) \in (madeBySensor)^{OP} \rightarrow\ x \in (employee)^{C})$ $\forall x, y\ (\ (x, y) \in (madeBySensor)^{OP} \rightarrow\ y \in (sensor)^{C}\ )$

Нетрудно видеть, что конъюнкции условий для СИНТЕЗ и условий для OWL эквивалентны. В простейших случаях условия просто совпадают или совпадают с точностью до переменных.

Таким образом, построенное отображение σ корректно.

Утверждение. Задачи вывода в канонической модели эквивалентны задачам вывода в дескриптивной логике SROIQ [231], отвечающей языку OWL 2 DL.

Доказательство. Основные задачи вывода в SROIQ сводимы к задаче существования модели в SROIQ [231]. Существование модели в SROIQ эквивалентно существованию модели в языке СИНТЕЗ (поскольку отображение σ между канонической моделью и OWL корректно), а задачи вывода в языке СИНТЕЗ сводимы к задаче существования модели в языке СИНТЕЗ. ■

2.8.3.2 Верификация взаимного отображения канонической модели данных и языка RIF-BLD

Определение отображения между канонической информационной моделью и языком RIF-BLD [141] приведено в таблице 2.6 (рассмотрено отображение основных конструкций).

Пример отображения конкретного правила из предметной области звездной астрономии выглядит следующим образом. Для правила языка СИНТЕЗ

```
r(_/[ra, de, name, magnitudes]) :-
astronomicalObject(x/[ra: spatialCoord.ra, de: spatialCoord.de, name,
objectType, properMotion, quality, magnitudes]) &
ra < queryRA + radius & ra > queryRA – radius &
de < queryDE + radius & de > queryDE – radius &
checkType(ra, de, "G", nType) & nType = false &
objectType = Star &
properMotion < 0.01 &
quality < 0.01.
```

соответствующее правило на языке RIF-BLD выглядит следующим образом:

```
r(ra->?ra  de->?de  name->?name  magnitudes->?magnitudes]) :-
And(astronomicalObject(?x)
?x[spatialCoord->?spatialCoord  name->?name  magnitudes->?magnitudes
objectType->?objectType  properMotion->?properMotion  quality->?quality]
?spatialCoord[ra->?ra  de->?de]
?ra < queryRA + radius & ?ra > queryRA - radius
?de < queryDE + radius & ?de > queryDE – radius)
checkType(?ra  ?de  "G"  nType)
nType = false
objectType = Star
pred:numeric-lesser-than(?properMotion  0.01)
pred:numeric-lesser-than(?quality  0.01) ).
```

Таблица 2.6 – Отображение конструкций RIF-BLD и языка СИНТЕЗ

Примечание	Конструкция RIF-BLD	Конструкция СИНТЕЗ
Переменная	?v	v
Числовая, строковая константа	"k"^^xs:string	k
Имя функции или коллекции	n	n
Равенство	t = s	m[t] = m[s]
Предикат-коллекция, переименование в редукте	And(?v # C ?v[a->?a b->?c])	C(v/[a, c: b])
Функциональные термы и предикаты f – имя функции (не встроенной функции)	f(t_1 … t_n)	f(m[t_1], … , m[t_n])
Конъюнкция	And(φ_1 ... φ_n)	(m[φ_1] & ... & m[φ_n])
Правила v_1, ... ,v_n – все свободные переменные ψ, в φ нет свободных переменных, отличных от v_1, ... ,v_n	Forall ?v_1 ... ?v_n (φ:- ψ)	m[φ]:- m[ψ].
Списки правил	Group(φ_1 ... φ_n)	m[φ_1]. ... m[φ_n].
Модули	Document(Group(φ_1 ... φ_n))	{ Document; in: module; function: { Group; in: function; {{ m[φ_1] ... m[φ_n] }} } }
Встроенные функции	External(func:numeric-add(x y))	x+y
Встроенные предикаты	External(pred:numeric-less-than(x y)	x < y

Нормативный документ с описанием семантики для языка RIF-BLD – это рекомендация W3C «RIF Basic Logic Dialect» [141]. *Определение семантики языка СИНТЕЗ в семантической структуре языка RIF-BLD*

$I = <TV, DTS, D, D_{ind}, D_{func}, I_C, I_V, I_F, I_{NF}, I_{list}, I_{frame}, I_{isa}, I_{=}, I_{truth}>$

приведенной в этом документе, выглядит следующим образом.

Укажем необходимые для задания семантики непересекающиеся множества алфавита: *Const* (константы), *Var* (переменнные), *ArgNames* (имена аргументов), множество истинностных значений $TV = \{\boldsymbol{t}, \boldsymbol{f}\}$.

Используется следующее множество встроенных типов данных языка СИНТЕЗ *DTS = { Boolean, double, float, octet, integer, long, short, unsigned_long, unsigned_short, string, {time; from: {yy}; to: {dd};}, {time; from: {yy}; to: {ss};}, {interval; from: {dd}; to: {ss};}, {interval; from: {yy}; to: {mm};} }* [42]. Типы данных налагают следующее ограничение на семантические структуры (аналогично [141]). Пусть $dt \in DTS$, LS_{dt} – лексическое пространство *dt*, VS_{dt} – символьное пространство *dt*, $L_{dt}: LS_{dt} \rightarrow VS_{dt}$ – отображение из лексического пространства в символьное, тогда $VS_{dt} \subseteq D_{ind}$ и для любой константы *lit* типа *dt* ($lit \in LS_{dt}$) выполнено $I_C(lit) = L_{dt}(lit)$.

Отображение *I* термов в *D* также определяется в соответствии с [141] с необходимыми изменениями ввиду специфики синтаксиса языка СИНТЕЗ:

– *Функциональный терм*. $I(f(t_1, \dots, t_n)) = I_F(I(f))(I(t_1),\dots,I(t_n))$

– *Терм – вызов метода*. $I(v.f(t_1, \dots, t_n)) = I_F(I(f))(I(v),I(t_1),\dots,I(t_n))$

– *Предикат-коллекция с анонимной переменной*. $I(C(\_/T[a, b])) = I_{NF}(I(C))(\{<arg\_a, I(?a)>, <arg\_b, I(?b)>\})$. Функция I_{NF} интерпретирует предикаты-коллекции с анонимными переменными. Предикат $C(\_/T[a, b])$ интерпретируется как функция $SetOfFiniteSets(\{arg\_a, arg\_b\} \times D_{ind}) \rightarrow D_{ind}$, где $arg\_a, arg\_b \in ArgNames$ (новые имена аргументов).

– *Последовательности*.

1) Пустая последовательность: $I([]) = I_{list}(<>)$

2) $I([t_1, \dots, t_n]) = I_{list}(I(t_1), \dots, I(t_n))$, при $n>0$

– *Фреймы*. $I(\{id; a: t_a; b: t_b; c: t_{c1}, t_{c2};\}) = I_{frame}(I(id))(\{<I(a),I(t_a)>, <I(b),I(t_b), <I(c),I(t_{c1}), <I(c),I(t_{c2})>\})$

Семантическая структура *I* задает истинностное означивание формул при помощи функции $TVal_I(\circ)$. Функция *TVal* определяется аналогично [141] с необходимыми изменениями ввиду специфики синтаксиса СИНТЕЗ:

- *Предикат-коллекция*. $TVal_I(C(v/T[a.b/c,\ d])) = \boldsymbol{t}$ iff $I_{truth}(I_{isa}(I(v),\ I(C))) = \boldsymbol{t} \wedge I_{truth}(I_{frame}(I(v))(\{<I(a),I(?a)>,\ <I(d),I(?d)>\})) = \boldsymbol{t} \wedge I_{truth}(I_{frame}(I(?a))(\{<I(b),I(?c)>\})) = \boldsymbol{t}$
- *Кнонъюнкция*. $TVal_I(\varphi_1\ \&\ ...\ \&\ \varphi_n) = \boldsymbol{t}$ iff $TVal_I(\varphi_1) = ... = TVal_I(\varphi_n) = \boldsymbol{t}$, иначе, $TVal_I(\varphi_1\ \&\ ...\ \&\ \varphi_n) = \boldsymbol{f}$
- *Дизъюнкция*. $TVal_I(\varphi_1 \mid ... \mid \varphi_n) = \boldsymbol{f}$ *iff* $TVal_I(\varphi_1) = ... = TVal_I(\varphi_n) = \boldsymbol{f}$, иначе, $TVal_I(\varphi_1 \mid ... \mid \varphi_n) = \boldsymbol{t}$
- *Квантор существования*. $TVal_I(ex\ v_1, ...\ , v_n\ (\varphi)) = \boldsymbol{t}$ *iff* для некоторого I^* (I^* совпадает с I во всем, кроме, может быть, означивания $v_1, ... , v_n$) $TVal_{I^*}(\varphi) = \boldsymbol{t}$.
- *Правила*. $TVal_I(\varphi{:-}\ \psi.) = \boldsymbol{t}$ iff для любого I^* (I^* совпадает с I во всем, кроме, может быть, означивания $v_1, ... , v_n$), $TVal_{I^*}(\varphi) = \boldsymbol{f}$ либо $TVal_{I^*}(\psi) = \boldsymbol{t}$, иначе $TVal_I(\varphi{:-}\ \psi.) = \boldsymbol{f}$.
- *Факты*. $TVal_I(\varphi.) = \boldsymbol{t}$ iff для любого I^* (I^* совпадает с I во всем, кроме, может быть, означивания $v_1, ... , v_n$), $TVal_{I^*}(\varphi) = \boldsymbol{t}$.
- *Списки правил*. $TVal_I(\varphi_1. ... \varphi_n.) = \boldsymbol{t}$ iff $TVal_I(\varphi_1) = ... = TVal_I(\varphi_n) = \boldsymbol{t}$, иначе $TVal_I(\varphi_1. ... \varphi_n.) = \boldsymbol{f}$.
- *Модули*. Пусть M – модуль, N – импортированный в M модуль, G – множество программ из модуля N, I_N – структура, интерпретирующая N. Тогда мультиструктура, интерпретирующая M - $\hat{I} = \{I, I_N\}$ и $TVal_{\hat{I}}(M) = \boldsymbol{t}$ iff $TVal_I(\varphi_1. ... \varphi_n.) = TVal_{IN}(G) = \boldsymbol{t}$.

Полное сопоставление семантики конструкций в семантической структуре I приведено в приложении А.7. Из сопоставления видно, что семантика конструкций СИНТЕЗ и соответствующих конструкций RIF-BLD в точности совпадает, а значит, отображение корректно.

2.9 Схема функциональной структуры и реализации инструментальных средств унификации моделей данных

В данном разделе рассмотрены два варианта схемы функциональной структуры и реализации инструментальных средств унификации моделей данных:

– реализация с использованием декларативного языка трансформации и средств переписывания термов;

– реализация с использованием декларативно-императивного языка трансформации моделей (на основе технологии движимой моделями инженерии MDE – Model-Driven Engineering).

Проведено сравнение реализаций для всех этапов метода унификации моделей данных.

2.9.1 Реализация Унификатора моделей с использованием декларативного языка трансформации и средств переписывания термов

В данной реализации в качестве формализмов для описания синтаксиса и трансформаций моделей данных используются языки SDF (Syntax Definition Formalism) и ASF (Algebraic Specification Formalism), поддержанные программной средой Meta-Environment [191]. Язык ASF декларативен и исполнение описанных на нем трансформаций основывается на переписывании термов.

Схема функциональной структуры средств унификации изображена на рисунке 2.11 (пунктирные стрелки обозначают доступ из одного компонента к интерфейсу другого компонента).

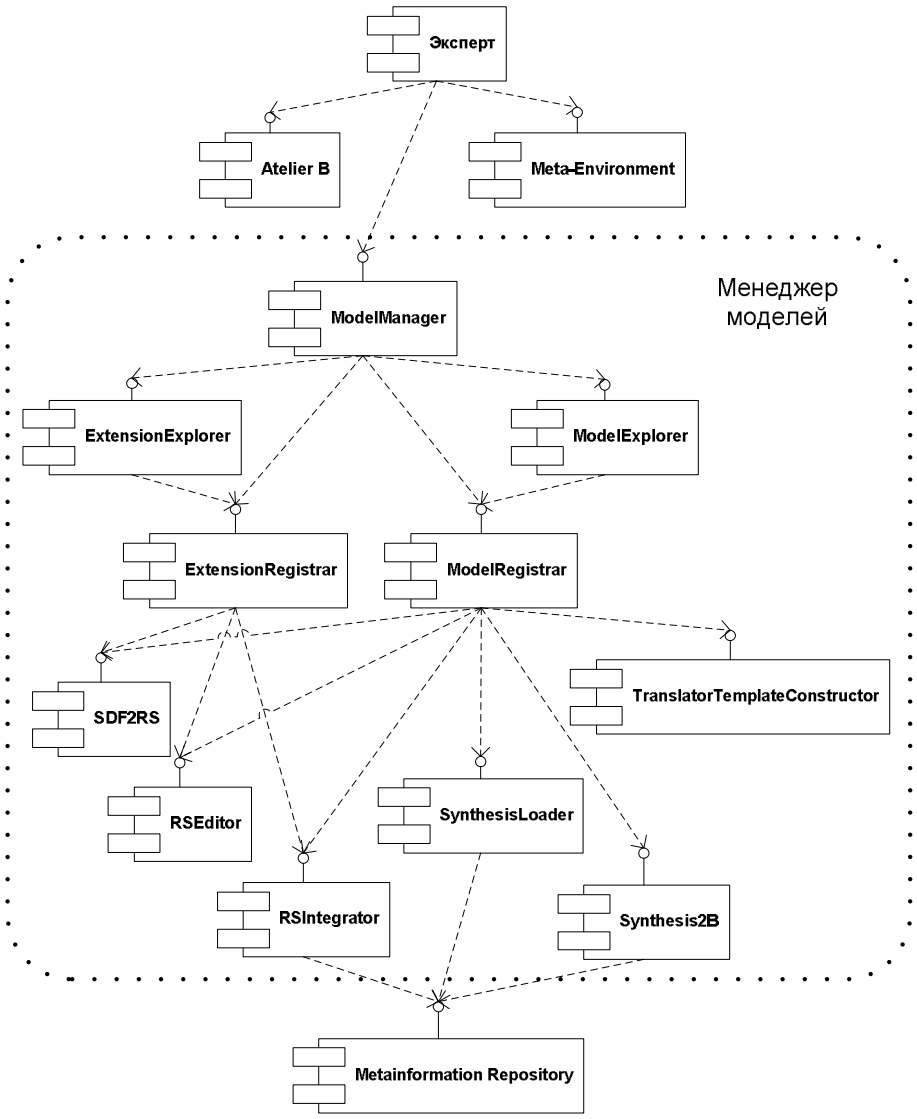

Рисунок 2.11 – Схема функциональной структуры Унификатора моделей с использованием декларативного языка трансформации и средств переписывания термов

Унификатор состоит из следующих крупных компонентов (групп компонентов):

– среда метакомпиляции Meta-Environment (для определения и проверки корректности синтаксиса моделей и трансформаций моделей);

– инструментарий Atelier B (поддерживающий язык AMN и средства доказательства уточнения спецификаций);

– репозиторий метаинформации;

– менеджер моделей.

Meta-Environment и Atelier B представляют собой самостоятельные продукты. Meta-Environment поддерживает следующие этапы унификации моделей: формализация синтаксиса (этапы 1б, 1в) и семантики исходной модели (этап 5), формализация синтаксиса расширения канонической модели (этапы 1б, 1в), формализация вербальных правил отображения исходной модели в каноническую и построение трансформаций моделей на основе автоматически сгенерированной заготовки (этап 4в), проверка образцов исходной модели на соответствие формальному синтаксису (этап 6а). Atelier B поддерживает этап 6д автоматического и интерактивного доказательства уточнения спецификаций.

Репозиторий метаинформации представляет собой объектно-реляционную базу данных и предназначен для реализации *реестра моделей* и хранения спецификаций. Реестр моделей (или просто *реестр*) содержит регистрационные карты моделей, расширений канонической модели, образцов. В регистрационных картах сохраняется вся информация, порождаемая в ходе унификации моделей (в том числе и информация, порождаемая в ходе взаимодействия эксперта с компонентами Meta-Environment и Atelier B). Например, список информационных объектов, составляющих регистрационную карту модели данных, выглядит следующим образом:

– название модели (краткое, полное);

– синтаксис модели:

  1) название документа, описывающего синтаксис (стандарт, заявка и т.д.);
  2) дата документа;
  3) ссылка на документ (URL);
  4) формальный конкретный синтаксис модели;
  5) спецификация абстрактного синтаксиса модели;

– семантика модели:

  1) название документа, описывающего семантику;
  2) дата документа;
  3) ссылка на документ (URL);

4) вербальные правила отображения модели в AMN (текстовый файл);

5) спецификация трансформации модели в AMN;

- список регистрационных карт образцов модели;
- ссылка на расширение канонической модели, с которым интегрируется модель;
- соответствие элементов исходной и канонической моделей;
- вербальные правила отображения модели в каноническую;
- спецификация трансформации модели в каноническую.

Все остальные компоненты Унификатора объединены в группу, называемую *менеджер моделей* (программная реализация на языке Java опубликована в репозитории GitHub[54]).

Компонент *ModelManager* предоставляет графический интерфейс, позволяющий эксперту подключиться к конкретному репозиторию метаинформации, а также вызвать компоненты обеспечивающие:

- поиск информационной модели в реестре;
- регистрацию новой модели в реестре;
- просмотр расширений, зарегистрированных в реестре;
- регистрацию нового расширения канонической модели в реестре.

Компонент *ModelExplorer* предоставляет графический интерфейс, позволяющий производить поиск модели в реестре по полному или краткому названию модели, названиям или ссылкам на документы, описывающие синтаксис или семантику модели.

Компонент *ExtensionExplorer* предоставляет графический интерфейс, позволяющий просматривать названия и вербальные описания зарегистрированных расширений канонической модели.

Компонент *ModelRegistrar* предоставляет графический интерфейс, позволяющий просматривать и редактировать регистрационные карты моделей и их образцов, содержащихся в реестре.

Компонент *ExtensionRegistrar* предоставляет графический интерфейс, позволяющий просматривать и редактировать регистрационные карты расширений канонической модели, содержащихся в реестре.

---

[54] https://github.com/sstupnikov/ModelUnifier.Meta-Environment

Компонент *SDF2RS* (SDF to Reference Schema) обеспечивает автоматическое построение заготовки исходной модели или расширения на основе их формального конкретного синтаксиса.

Компонент *RSEditor* (Reference Schema Editor) предоставляет графический интерфейс, позволяющий превратить заготовку схемы абстрактного синтаксиса модели или расширения в собственно схему абстрактного синтаксиса (этап 1а метода унификации).

Компонент *RSIntegrator* (Reference Schema Integrator) обеспечивает поддержку интеграции эталонных схем исходной модели и канонической модели (этап 2б) или расширения канонической модели (этап 3в).

Компонент *SynthesisLoader* обеспечивает помещение схем абстрактного синтаксиса моделей (этапы 1а, 2б) и спецификаций образцов моделей (представленных в канонической модели) в репозиторий метаинформации (этап 5в).

Компонент *TranslatorTemplateConstructor* обеспечивает автоматическую генерацию заготовки трансформации исходной модели в каноническую (этап 4б).

Компонент *Synthesis2B* обеспечивает автоматическое отображение спецификаций канонической модели в язык AMN (этап 5г).

2.9.2 Реализация Унификатора моделей с использованием декларативно-императивного языка трансформации моделей

В данной реализации в качестве формализма для описания абстрактного синтаксиса моделей данных используется метамодель Ecore [196] уровня MOF M3, используемая в проекте Eclipse Modeling Project. Для описания конкретного синтаксиса используется язык CS, реализованный в рамках каркаса EMFText [219]. Для конструирования трансформаций моделей данных используется язык ATL [192], сочетающий декларативные и императивные возможности.

Схема функциональной структуры средств унификации изображена на рисунке 2.12 (стрелки обозначают доступ из одного компонента к интерфейсу другого компонента).

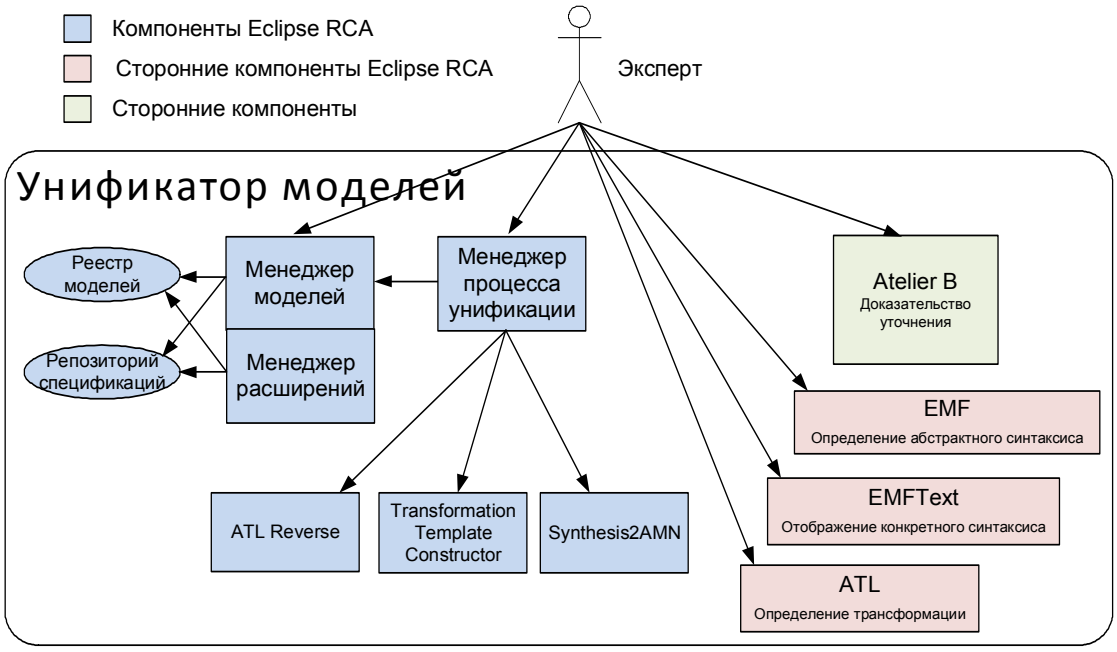

Рисунок 2.12 – Схема функциональной структурыУнификатора моделей с использованием декларативно-императивного языка трансформации моделей

Унификатор состоит из следующих крупных компонентов (групп компонентов):

– ядро Унификатора моделей, реализованное на платформе Eclipse в виде расширяемого клиентского приложения (Rich Client Application, RCA) на языке Java;

– средства платформы Eclipse для формализации синтаксиса и трансформаций моделей:

1) каркас Eclipse Modeling Framework для разработки абстрактного синтаксиса;

2) инструментарий EMFText разработки конкретного синтаксиса;

3) инструментарий поддержки ATL для разработки трансформаций моделей данных;

– инструментарий Atelier B (поддерживающий язык AMN и средства доказательства уточнения спецификаций).

EMF, EMFText, ATL, Atelier B представляют собой самостоятельные продукты. EMF поддерживает следующие этапы унификации моделей: формализация абстрактного синтаксиса моделей данных и их расширений (этап 1а), интеграция абстрактных синтаксисов моделей (этап 2а). EMFText поддерживает формализацию конкретного синтаксиса моделей данных и их расширений (этапы 1б, 1в), проверку образцов исходной модели на соответствие формальному синтаксису (этап 6а). ATL поддерживает формализацию вербальных правил отображения исходной модели в каноническую и построение трансформаций моделей на основе автоматически сгенерированной заготовки (этап 4в), формализацию семантики исходной модели (этап 5). Atelier B

поддерживает этап 6д автоматического и интерактивного доказательства уточнения спецификаций.

Остальные компоненты составляют ядро Унификатора моделей, представляющее собой расширение (плагин *ModelUnifierPlugin*) платформы Eclipse.

Реестр моделей представляет собой совокупность регистрационных карт моделей данных, расширений канонической модели, образцов, хранящихся в виде xml-файлов, объединяемых в проекты (подобные проектам Eclipse). В регистрационных картах сохраняется вся информация, порождаемая в ходе унификации моделей (в том числе и информация, порождаемая в ходе взаимодействия эксперта с компонентами EMF, EMFText, ATL, Atelier B). Структура регистрационных карт определяется соответствующими Ecore-моделями. Доступ к регистрационным картам осуществляется через интерфейсы, автоматически порождаемые для Ecore-моделей инструментарием EMF. Совокупность проектов различных регистрационных карт образует *рабочее пространство* унификатора. Структуры регистрационных карт включают следующие информационные объекты:

– Регистрационная карта процесса унификации:
    1) ссылка на регистрационную карту исходной модели;
    2) ссылка на регистрационную карту целевой модели;
    3) список регистрационных карт примеров исходной модели;
    4) ссылка на список соответствий элементов исходной и целевой моделей;
    5) вербальные правила отображения исходной модели в целевую;
    6) спецификация ATL-трансформации исходной модели в целевую.
– Регистрационная карта модели данных:
    1) название модели (краткое, [полное]);
    2) синтаксис модели:
        а) нормативный документ, описывающий синтаксис (стандарт, заявка и т.д.);
        б) абстрактный синтаксис модели (Ecore);
    3) семантика модели:
        а) документы, описывающие семантику;
        б) вербальные правила отображения модели в AMN;
        в) спецификация ATL-трансформации модели в AMN.
– Регистрационная карта расширения модели (наследуется от регистрационной карты модели):
    1) ссылка на регистрационную карту расширяемой модели.

– Регистрационная карта примера исходной модели:

1) название примера;

2) спецификация примера;

3) семантические спецификации примера модели в AMN;

4) спецификация примера в целевой модели;

5) семантические спецификации примера целевой модели в AMN;

6) текст доказательства уточнения примера канонической модели посредством примера исходной модели;

7) флаг доказанности уточнения.

Структура папок проекта унификации выглядит следующим образом:

– папка модели:

1) *.ecore файл регистрационной карты модели;

2) *.ecore файл абстрактного синтаксиса;

3) *.cs файл конкретного синтаксиса;

4) файлы синтаксиса;

5) файлы семантики;

6) файлы отображения в AMN;

7) *.atl файлы трансформации в AMN;

– папка расширения (наследует структуру папки модели);

– папка унификации:

1) *.ecore файл регистрационной карты процесса унификации;

2) *.ecore файл соответствий между элементами моделей;

3) папки образцов (репозиторий спецификаций):

а) *.ecore файл регистрационной карты образца;

б) *.ecore файл спецификации образца;

в) *.ecore файл AMN-спецификации образца;

г) *.ecore файл согласованной AMN-спецификации образца;

д) *.ecore файл целевой спецификации образца;

е) *.ecore файл целевой AMN-спецификации образца;

ж) *.ecore файл согласованной целевой AMN-спецификации образца;

з) файл с текстом доказательства.

Компонент *менеджер моделей* (ModelRegistrar) представляет собой графический интерфейс унификатора для работы с регистрационными картами моделей. С помощью графического интерфейса эксперт создает регистрационные карты моделей, просматривает и изменяет информацию, содержащуюся в регистрационных картах.

Компонент *менеджер расширений* (ExtensionRegistrar) представляет собой графический интерфейс унификатора для работы с регистрационными картами расширений. С помощью графического интерфейса эксперт создает регистрационные карты расширений, просматривает и изменяет информацию, содержащуюся в регистрационных картах.

Компонент *менеджер процесса унификации* (UnificationRegistrar) представляет собой графический интерфейс унификатора для поддержки процесса унификации исходной модели в канонической. С помощью графического интерфейса эксперт создает регистрационные карты образцов исходной модели, просматривает и изменяет информацию, содержащуюся в регистрационных картах.

Компонент *TemplateTransformationConstructor (TTC)* предназначен для генерации заготовки ATL-трансформации исходной модели в целевую на основе описаний их абстрактного синтаксиса и соответствия элементов моделей (раздел 2.6.1).

Компонент *ATLReverse* предназначен для генерации заготовки обратной ATL-трансформации на основе прямой ATL-трансформации (раздел 2.6.2).

Компонент *Synthesis2AMN* предназначен для автоматического преобразования канонических образцов моделей в их семантические спецификации на языке AMN (раздел 2.7.6).

2.9.3 Сравнение реализаций

Раздел основан на работе автора [82].

В данном разделе проведено сравнение подходов к реализации метода унификации моделей данных на основе средств метакомпиляции SDF-ASF и средств движимой моделями инженерии MDE, рассмотренных в предыдущих разделах. Сводка сравнения по нескольким основным критериям приведена в таблице 2.7. Каждый критерий более подробно объяснен в соответствующем пункте ниже.

2.9.3.1 Формализация синтаксиса моделей данных

В SDF синтаксис модели данных представляется с использованием варианта расширенной формы Бэкуса-Наура. Этот подход имеет недостатки, связанные с отсутствием встроенного абстрактного синтаксиса. Язык SDF предоставляет средства для определения грамматик конкретного синтаксиса, только в виде синтаксических структур, склеенных с «синтаксическим сахаром» (ключевыми словами или вспомогательными символами).

Таблица 2.7 – Сводка сравнения подходов к реализации метода унификации моделей данных

Критерий	Подход SDF-ASF	Подход MDE
Определение конкретного синтаксиса	Язык Syntax Definition Formalism, контекстно-свободные грамматики	Конкретный синтаксис определяется с использованием инструментариев EMFText или XText, интегрированных с Eclipse Modeling Framework (EMF), связывается с предопределенным абстрактным синтаксисом
Определение абстрактного синтаксиса	Заготовка абстрактного синтаксиса автоматически извлекается из конкретного синтаксиса специальным программным средством, затем редактируется экспертом.	Определяется независимо от конкретного синтаксиса с использованием метамодели Ecore уровня MOF M3
Интеграция абстрактных синтаксисов	Специальное программное средство используется для установления соответствий между элементами синтаксисов	Соответствия могут быть вручную установлены с использованием редактора Ecore2Ecore или автоматизированно с использованием сторонних средств сопоставления схем
Конструирование трансформаций	Язык Algebraic Specification Formalism, техника переписывание термов	Языки трансформации моделей (ATL, QVT)
Генерация заготовок трансформаций	Специальная программная реализация на языке Java	Генератор заготовок также реализован на ATL

ASF-трансформации моделей данных, в свою очередь, оперируют SDF-грамматиками и таким образом существенно зависят от конкретного синтаксиса. Однако, модели данных могут иметь несколько вариантов конкретного синтаксиса (форматов). Например, язык OWL может быть представлен с использованием XML или функционального синтаксиса. Таким образом, для каждой пары конкретного синтаксиса исходной модели и конкретного синтаксиса канонической модели необходимо разработать отдельную трансформацию. Таким образом, можно выделить следующие преимущества подхода MDE к формализации синтаксиса моделей данных (SYntax Formalization, SYF).

Преимущество SYF1. Абстрактные синтаксисы моделей данных (исходных, канонических или семантических) определяются как модели уровня MOF M2, конформные одной и той же модели уровня MOF M3 (Ecore) и совместимые со всей архитектурой MOF. Абстрактный синтаксис может быть легко сериализован в XML.

Преимущество SYF2. Определения абстрактного синтаксиса и конкретного синтаксиса для модели данных разделены. Абстрактный синтаксис может быть связан с любым видом конкретного синтаксиса. Редакторы и синтаксические анализаторы конкретного синтаксиса создаются автоматически на основе грамматик конкретного синтаксиса.

Преимущество SYF3. Абстрактный синтаксис модели данных, определенный один раз, может быть в дальнейшем применен в различных трансформациях либо в качестве входной, либо в качестве выходной модели.

2.9.3.2 Установление соответствий элементов моделей

Согласно подходу SDF-ASF, сначала из определения конкретного синтаксиса должна быть извлечена заготовка абстрактного синтаксиса. Затем эта заготовка должна быть доработана экспертом с использованием редактора абстрактного синтаксиса. После этого с помощью специальных инструментов должны быть установлены соответствия между элементами исходной и канонической модели. Очевидно, что соответствующий этап в подходе MDE намного проще благодаря разделению абстрактного и конкретного синтаксиса, рассмотренному в предыдущем подразделе. Можно выделить следующее преимущество подхода MDE к интеграции абстрактных синтаксисов (Integration of Abstract Syntaxes, IAS).

Преимущество IAS1. С использованием программных каркасов, поддерживающих MDE, реализованы и могут быть применены для интеграции абстрактных синтаксисов исходной и канонической моделей различные существующие методы сопоставления схем. Список соответствий между элементами абстрактных синтаксисов представляется в виде модели уровня MOF M1 и может быть сериализован в XML.

2.9.3.3 Создание расширения канонической модели

Можно выделить следующие преимущества подхода MDE к созданию расширений канонической модели (Extension Construction, EC).

Преимущество EC1. Абстрактный синтаксис синтаксического расширения канонической модели определяется как расширение абстрактного синтаксиса канонической модели уровня MOF M2. Конкретный синтаксис расширения определяется как расширение конкретного синтаксиса канонической модели.

Преимущество EC2. Расширение канонической модели, использующее синтаксические и семантические особенности ядра канонической модели (специальные структуры данных, такие, как типы и классы, зависимости, аксиомы и т.д.), определяется как модель уровня MOF M1, соответствующая канонической модели абстрактного синтаксиса уровня MOF M2.

Преимущество EC3. Элементы синтаксиса расширения сопоставляются с элементами синтаксиса исходной модели с помощью существующих инструментов сопоставления схем, разработанных с использованием программных каркасов MDE.

2.9.3.4 Конструирование трансформации исходной модели в расширенную каноническую модель

Трансформации моделей данных определяются в языке ASF как модули, которые представляют собой наборы функций. Функция определяет трансформацию синтаксического элемента исходной модели в синтаксический элемент канонической модели как процедуру переписывание термов. Функции оперируют конкретными синтаксическими термами, поэтому рекурсивные вызовы функций смешиваются с символами синтаксического сахара (включая пробелы, табуляции и новые строки) в выражениях правил переписывания. Функция трансформации обычно определяется как множество правил переписывания термов, а не как единое праивло, учитывающее различные варианты структуры входного терма. Отладка сложных трансформаций, состоящих из множества функций, может быть очень сложной. Таким образом, можно выделить следующие преимущества подхода MDE к построению трансформаций моделей данных (Construction of Transformations, CT).

Преимущество CT1. Трансформации не зависят от конкретного синтаксиса модели данных. Это делает программный код трансформации более читаемым. Кроме того, использование различных форматов спецификаций входной и выходной моделей данных не требует изменения трансформации.

Преимущество CT2. Языки трансформации моделей нацелены на *декларативное* определение трансформаций. Правила трансформации определяют:

– для каких типов элементов исходной модели должны быть созданы элементы целевой модели и

– способ инициализации создаваемых целевых элементов.

При этом не требуется переписывать вложенные термы. Декларативный стиль упрощает расширение преобразований. Тем не менее, в специальном разделе правила трансформации разрешены императивные инструкции, что повышает гибкость определения трансформации.

2.9.3.5 Формализация семантики моделей данных

Можно выделить следующие преимущества подхода MDE к формализации семантики моделей данных (SEmantics Formalization).

Преимущество SEF1. Семантические трансформации моделей данных в AMN реализуются с использованием того же инструментария MDE (ATL), что и трансформации исходных моделей данных в каноническую модель. Применяется

смешанный декларативно-императивный стиль конструирования трансформаций, обеспечивается расширяемость трансформаций.

Преимущество SEF2. Семантическая трансформация расширения ядра канонической модели сама расширяет семантическую трансформацию ядра канонической модели в AMN.

2.9.3.6 Верификация корректности отображения моделей

Можно выделить следующие преимущества подхода MDE к верификации корректности отображения моделей (Refinement Verification).

Преимущество RV1. На этапе верификации применяются все модели и трансформации, разработанные с использованием инструментария MDE на предыдущих этапах унификации моделей.

Преимущество RV1.1. Для определения образцов моделей данных и автоматического создания их представлений в абстрактном синтаксисе применяются автоматически сгенерированные редакторы конкретного синтаксиса.

Преимущество RV1.2. ATL-трансформации исходных моделей данных в каноническую модель применяются для генерации канонических представлений образцов исходных моделей данных.

Преимущество RV1.3. Семантические трансформации используются для генерации AMN-спецификаций, которые загружаются в средство доказательства теорем уточнения.

2.10 Родственные работы

Методы и инструменты для преобразования схем. Подход Model Independent Data and Schema Translation (MIDST) [232], предназначен для преобразования схем из одной модели данных в другую “эквивалентным” способом. Преобразования выражаются в виде логических правил на языке. Основная идея MIDST заключается в использовании метамодели – набора универсальных метаконструкций, применяемых для определения моделей данных, которые являются экземплярами метамодели. Каждая модель определяется ее конструкциями и метаконструкциями, на которые они ссылаются. *Супермодель* - это модель данных, содержащая конструкции, соответствующие всем метаконструкциям, известным системе. Каждая модель данных является специализацией супермодели, поэтому схема, выраженная в любой модели данных, является схемой в супермодели. Отображение схемы, выраженной в одной модели данных, в другую определяется в терминах преобразования метаконструкций.

Произвольное преобразование схем создается автоматически как композиция базовых преобразований, относящихся к отдельным конструкциям. Базовые преобразования определены для нескольких семейств моделей данных (сущность-связь, объектно-ориентированные, реляционные). Модель может сопровождаться набором утверждений (инвариантов модели). Вводится понятие *поглощения моделей*. Модель M_1 поглощается моделью M_2 тогда и только тогда, когда M_2 включает, по крайней мере, все конструкции M_1 и для каждого инварианта M_1, она включает соответствующий ослабленный инвариант. Этот подход применим для структурированных моделей, которые могут быть выражены в виде композиции метаконструкций. Для определения базовых преобразований требуется значительная предварительная работа.

Подход Model Independent Schema Management (MISM) [233] – это расширение MIDST, включающее операторы манипулирования схемами: соединение, разность и базовая версия сопоставления. Платформа MIDST-RT [234] еще больше расширяет подход MIDST с помощью средств преобразования данных во время выполнения (runtime). Основная идея заключается в том, чтобы на основе правил преобразования схемы, создаваемых в рамках MIDST, получить представления над исходной базой данных для сущностей целевой схемы. Такие представления служат для преобразования данных из исходной схемы в целевую.

В работе [235] из Университета Дижона в качестве канонической модели для полихранилища[55] рассматривается тензорная модель данных (TDM). Определены двунаправленные отображения схем TDM со схемами реляционной, графовой моделей и модели ключ-значение. Утверждается, что схемы TDM «структурно эквивалентны» схемам исходных моделей. В тензорной модели определены операции проекции, селекции, объединения, пересечения и соединения тензоров; для операций указаны эквивалентные операции реляционной алгебры. Однако, доказательство корректности отображений схем и языков запросов не приводится.

В работе [236] из Университета Мурсии предложена унифицированная метамодель (U-Schema) для представления логических схем реляционной, документной, графовой моделей и модели ключ-значение. Разработаны прямые и обратные отображения между U-Schema и исходными моделями. Утверждается реверсивность прямых и обратных отображений. Не рассматриваются отображения языков запросов.

---

[55] Архитектуры полихранилищ более подробно обсуждаются в разделе родственных работ 5.1.3.

По сравнению с вышеприведенными работами метод унификации модели данных, предложенный в данной работе, является более общим и семантически ориентированным. Он не ограничивается выделенными семействами структурированных моделей (например, модели данных могут включать функции и процессы). В качестве ядра канонической модели могут быть выбраны различные модели данных. Произвольные отображения моделей данных, не ограниченные композициями базовых преобразований, могут быть реализованы с использованием языка трансформаций моделей (например, ATL). Использование AMN для определения семантики моделей данных позволяет формально доказывать сохранение информации и операций моделей данных при отображениях.

Работы М. Манукяна по конструированию канонических моделей данных и интеграции данных, как указано в работе [237], основаны на работах Л.А. Калиниченко и автора данного диссертационного исследования и следуют методу унификации моделей, рассматриваемому в данной главе. В качестве ядра канонической модели эти работы используют модель XCM (eXtensible Conceptual Model, называемую в более ранних работах XDM – an XML Data Model), представляющую собой расширение XML элементами λ-исчисления для описания составных объектов [237] [238]. В более поздних работах [239] [240] [241] модель расширяется *контентными словарями* стандарта представления математических объектов OpenMath[56]. Словари содержат описания понятий, расширяющих модель. Например, определяются описания предметного посредника, хранилища данных, отображений элементов исходных моделей данных в каноническую модель. Для определения формальной семантики, в соответствии с методом унификации моделей, используется язык AMN.

Создание трансформаций моделей. В работе [242] предложен подход к автоматизированной разработке трансформаций моделей. Для пары исходной и целевой моделей автоматизированно создается *сплетающая* (weaving) модель, фиксирующая возможные связи между элементами исходной и целевой моделей. Отдельные элементы сплетающей модели модифицируются вручную. Затем для каждой связи элементов вычисляется ее значение сходства с использованием сходства строк, словаря синонимов и структурного сходства, основанного на внутренних свойствах элементов модели (типах, мощности и связях между элементами моделей). Выбираются связи с

[56] https://openmath.org/

наибольшим значением сходства и используются для генерации правил преобразования. Этот подход, по-видимому, хорошо применим для создания преобразований схем. Однако, преобразования моделей данных (языков) разработать гораздо сложнее, а полезность методов сопоставления схем сомнительна из-за семантических различий между моделями данных.

В работе [221] предлагается онтологически-базированный подход к отображению моделей (ontMT). Технологические пространства MDA и онтологий объединяются для автоматизации построения и эволюции отображений моделей. Целевая и исходная модели связываются с некоторой эталонной онтологией. Вычисляются отношения между элементами моделей и строится двунаправленное отображение моделей, выраженное в языке отношений QVT Relations.

В работе [243] рассматривается подход, основанный на примерах (by-example). Отображение определяется сначала на конкретных моделях уровня M1, иллюстрирующих нотацию языков, которые необходимо отобразить друг в друга. Информация, содержащаяся в отображениях конкретных моделей, объединяется и на ее основании строится двунаправленное ATL-отображение моделей уровня M2.

Предложены алгебраическая теория двунаправленных отображений моделей [244], а также двунаправленные языки запросов для обновляемых взглядов [245].

В работе [246] предложен методологический и технический каркас для управляемой моделями разработки модельных преобразований (MeTAGeM). Разработка начинается с модели независимой от платформы трансформации (PIT), которая представляет собой набор взаимосвязей между элементами исходной и целевой моделей, определенных экспертом. PIT-модель автоматически преобразуется в гибридную платформо-зависимую модель трансформации (PST-H). PST-H-модель содержит правила и операции для трансформации моделей. После этого PST-H-модель автоматически преобразуется в языки ATL или RubyTL.

В работе [247] рассматривается подход к автоматизации построения трансформаций моделей на основе требований, примеров и метамоделей.

По сравнению с родственными работами, метод автоматизированного построения трансформаций, рассматриваемый в данной работе, оставляет абстрактное сопоставление синтаксиса любым сторонним инструментам и концентрируется на порождающих правилах автоматического построения трансформаций моделей на основе класса соответствий между элементами моделей. Порождающие правила специализированы для построения трансформаций моделей уровня MOF M2. Для построения трансформаций используются некоторые специальные конструкции языка

ATL. Предлагаемый метод построения обратных трансформаций особенно полезен, когда прямая трансформация не является двунаправленной изначально.

Верификация трансформаций моделей. В ряде работ применяются различные формальные методы для верификации трансформаций моделей. В [248] формальная спецификация, валидация и верификация преобразований моделей осуществляется с помощью языка Abstract State Machines (ASM), поддерживаемого средствами интерактивного и автоматического доказательством теорем, включая проверку моделей (model checking). В [249] формальное исчисление для операционного диалекта языка QVT реализовано при помощи средств интерактивного доказательства теорем KIV, что позволяет доказывать свойства QVT-трансформаций для произвольных метамоделей. В [250] ограниченное множество правил сопоставления языка ATL преобразуется в модель трансформации на языке UML. Ограничения модели трансформации, подлежащие верификации, выражаются на языке OCL. Инструмент UML2Alloy применяется для преобразования модели трансформации и OCL-ограничений в спецификацию для инструмента проверки моделей Alloy. Таким образом может быть проверено любое ограничение, сводимое к задаче логической выполнимости (SAT).

В [251] предложен каркас для верификации трансформаций моделей. Он предназначен для поддержки, с одной стороны, ряда языков трансформаций моделей, таких как QVT, ATL и языки преобразования графов, а с другой стороны, ряда формализмов верификации, таких, как AMN, Alloy и т.д. Предложена простая метамодель для языков моделирования, включающая тип сущности, примитивный тип данных, свойство данных, структуру и элементы ограничений. Ограничения выражаются с использованием подмножества языка OCL. Определена формальная семантика метамодели в теории множеств первого порядка. Ограничения представлены в виде аксиом. Также определена метамодель спецификации трансформаций. Отображения трансформаций определяются с использованием пред- и постусловий, выражаемых как ограничения на языке OCL. Формализованы несколько свойств трансформаций и предложены способы применения различных методов для их верификации. Этот подход был реализован с использованием языка и инструментария UML-RSDS. Спецификации трансформаций определяются при помощи диаграмм примеров применения (use case) языка UML и могут быть автоматически преобразованы в языки Z3 и AMN для дальнейшего семантического анализа.

По сравнению с рассмотренными выше подходами, метод унификации моделей данных использует язык AMN для представления семантики моделей данных. Он предназначен не для доказательства произвольных свойств преобразований моделей с использованием средств верификации общего назначения, а для применения специальных интерактивных и автоматических средств для верификации уточнения моделей данных. Верификация при помощи инструментальных средств B-технологии выполняется на наборе образцов исходной модели данных (которые являются ее типичными конструкциями), или для конкретных схем, используемых при интеграции данных; либо на спецификациях самих моделей данных. Возможна также верификация на основе эквивалентного представления моделей данных в единой семантической структуре.

Таким образом, главные отличительные черты разработанного метода унификации моделей данных состоят в следующем:

– метод сочетает разработку унифицирующих трансформаций моделей и верификацию уточнения моделей данных, чтобы устранить модельную неоднородность современных СУБД, обеспечить формальную основу для отображения схем и виртуальной или материализованной интеграции в инфраструктурах неоднородных данных;

– для трансформаций моделей данных обеспечивается формальное доказательство сохранения информации и семантики операций ЯМД;

– для обеспечения унификации разнообразных моделей данных в неоднородных инфраструктурах метод предлагает механизм расширения и синтеза унифицирующей канонической модели данных;

– в качестве ядра канонической модели данных применена объектная модель с механизмами расширения в виде специализированных типов данных, ограничений (зависимостей, аксиом), сложных операций над структурами данных;

– метод применен для унификации широкого спектра моделей данных: сетевой модели данных, процессной модели, онтологической модели, графовой модели, тензорной модели, модели на логических правилах.

3 Метод верификации интеграции схем данных

В данной главе рассмотрены два варианта метода верификации интеграции схем данных.

Первый вариант метода нацелен на верификацию интеграции схем в предметных посредниках, использующих в качестве ядра унифицирующей канонической модели данных язык СИНТЕЗ. Вариант метода основан на уточнении абстрактных типов данных с использованием отображения канонической модели в AMN. Предполагается, что унификация моделей данных источников в канонической модели данных уже проведена.

Второй вариант метода нацелен на верификацию интеграции схем в системах виртуальной интеграции данных, использующих в качестве унифицирующей модели данных RDF. Корректность интеграции проверяется на наборе запросов пользователя к системе интеграции, выражаемых на языке SPARQL. Корректность доказывается путем отображения схем предметной области, схем источников данных и запросов в AMN, и дальнейшим применением автоматизированных средств доказательства уточнения для языка AMN.

3.1 Верификация интеграции схем данных на основе уточнения типов

Раздел основан на работах автора [45] [75].

В данном разделе рассматривается метод верификации корректности интеграции схем данных, основанный на уточнении типов данных с использованием отображения канонической модели в AMN.

Целью метода является формальное доказательство того, что некоторый *предметный посредник,* использующий в качестве ядра унифицирующей канонической модели данных язык СИНТЕЗ, осуществляет корректную интеграцию данных из некоторого набора источников данных. Корректность доказывается путем отображения схем посредника и схем источников данных в формальный язык AMN, и дальнейшим применением автоматизированных средств доказательства уточнения для языка AMN.

Общая схема метода изображена на рис. 3.1.

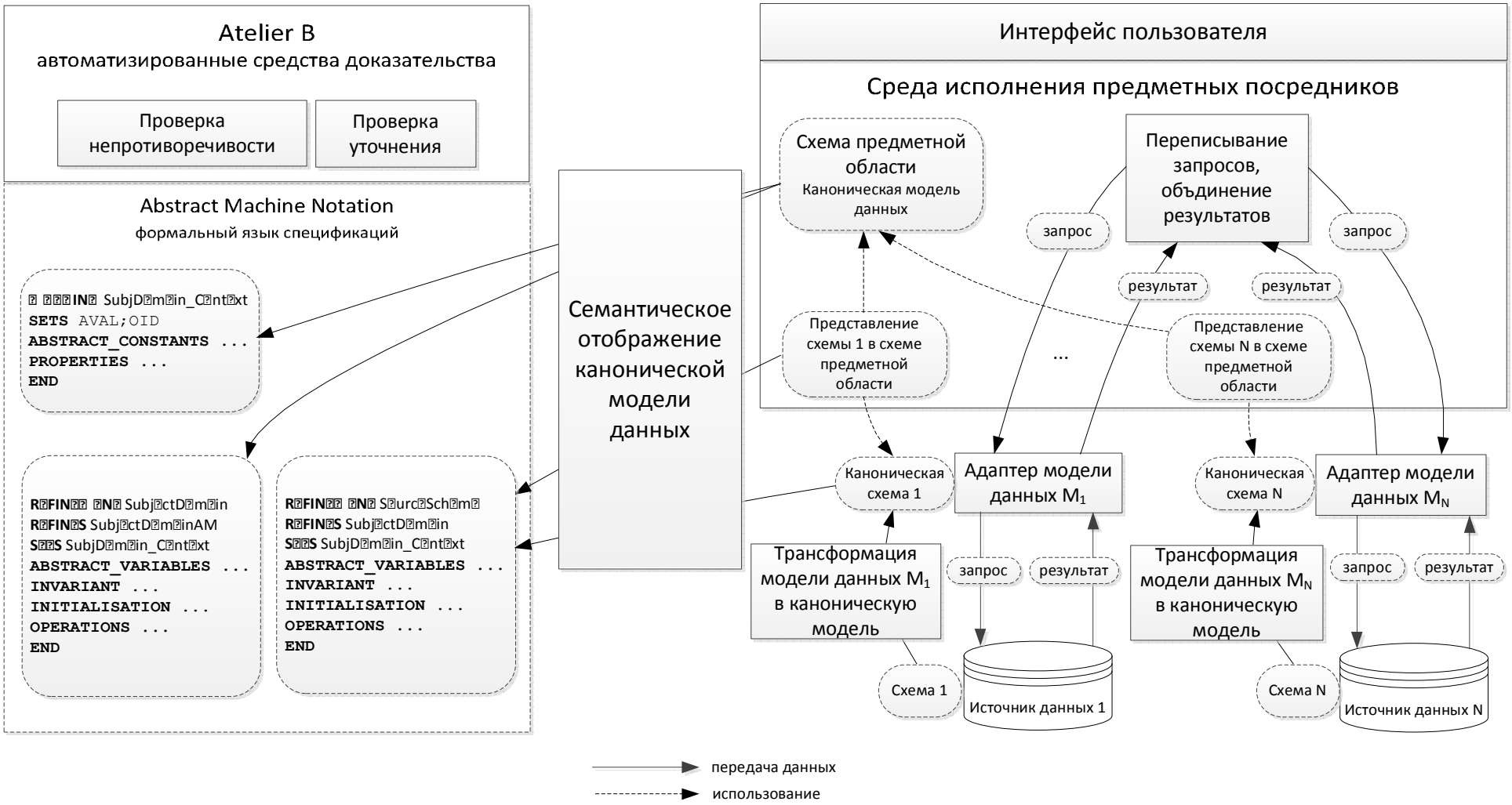

Рисунок 3.1 - Общая схема метода верификации интеграции схем данных на основе уточнения типов

В *предметном посреднике* задана *схема предметной области* (*глобальная схема*), представленная в канонической модели данных. В посреднике интегрированы *N источников данных.* Каждый из источников, в общем случае, использует собственную модель данных: в этой модели представляются *схема* источника и *запросы* к нему. Предполагается, что унификация моделей данных источников в канонической модели данных уже проведена в соответствии с методом, рассмотренным во второй главе. При этом построены *трансформации*, преобразующие схемы, представленные в моделях данных источников M_1, ... M_N в схемы канонической модели. Трансформации применены для преобразования схем источников в каноническую модель данных, получены *канонические схемы*.

Источник данных связывается с посредником при помощи *адаптера*. Для каждой модели данных источника конструируется собственный адаптер. Для каждого конкретного источника настраивается экземпляр адаптера, соответствующего модели данных источника: в экземпляре адаптера сохраняется IP-адрес источника и его схема.

Установлены соответствия между АТД глобальной схемы и релевантными им АТД схем источников. Для каждого источника в посреднике определен набор *представлений* (взглядов), связывающий сущности схемы источника и сущности глобальной схемы.

На вход от пользователя посредник принимает запросы, выраженные в терминах глобальной схемы в канонической модели. Входящий в среду исполнения предметных посредников компонент *переписывания запросов* преобразует эти запросы в запросы на языках источников с использованием представлений. Запросы отсылаются на источники, исполняются на них, а затем результаты возвращаются в систему, объединяются, и предоставляются пользователю.

В родственных работах сотрудников отдела «Методы и программные средства накопления и обработки данных» ФИЦ ИУ РАН подробно рассмотрены методы и средства разработки среды исполнения предметных посредников [26][252], регистрации источников данных в посредниках [101][136], переписывания запросов [73], создания адаптеров [183].

3.1.1 Основной алгоритм метода верификации интеграции схем

Основной алгоритм метода верификации интеграции схем, принимающий на вход глобальную схему S_{MED}, схемы источников $Sources = \{S_1, \dots, S_N\}$ в канонической модели, множество соответствий типов глобальной схемы и релевантных им АТД из схем источников *Sims*, множество взглядов $V = \{V_1, \dots, V_L\}$ приведен ниже.

Каждый элемент $s \in Sims$ имеет вид $T \mapsto U$, где АТД $T \in S_{MED}$, АТД $U \in S_{i0}$, $i_0 \in \{1, \dots, N\}$.

Каждый элемент V имеет вид Global and Local asView (GLAV) [253]:

$$G_1(v_1/T_{v1}) , \ldots , G_p(v_p/T_{vp}), B_G \supseteq H_1(w_1/T_{w1}) , \ldots , H_q(w_q/T_{wq}), B_H$$

где $G_1, \ldots , G_p$ – классы глобальной схемы, $T_{v1}, \ldots , T_{vp}$ – типы их экземпляров; $H_1, \ldots , H_q$ – классы схем источников, $T_{w1}, \ldots , T_{wq}$ – типы их экземпляров; B_G, B_H – конъюнкции предикатов-условий на значения атрибутов экземпляров классов, обозначенных переменными $v_1, \ldots , v_p$ и $w_1, \ldots , w_q$ соответственно; либо вид функционального взгляда:

$$T_M.g(u_{1, \ldots,} u_r) \supseteq T_S.h(u_{1, \ldots,} u_r)$$

где T_M – АТД глобальной схемы, g – один из его методов, T_S – АТД схемы некоторого источника, h – один из его методов.

Алгоритм *Верификация интеграции схем* (S_{MED}, *Sources, Sims, V*)

начало

- *proved* ← ∅
- Применить семантическую трансформацию Tr_{AMN} схем канонической модели в AMN (раздел 2.7.6) к схемам $S_{MED}, S_1, \ldots , S_N$. В результате применения будет получена контекстная машина *Context* и набор конструкций AMN, соответствующих АТД из схем. Для АТД T соответствующая конструкция AMN обозначается $Tr_{AMN}(T)$
- Установить отношение экстенсионалов АДТ глобальной схемы и схем источников в разделе PROPERTIES контекстной машины на основании соответствий АТД из *Sims*
- **для** каждого GLAV-взгляда $V_0 \in V$ вида
- $G_1(v_1/T_{v1}) , \ldots , G_p(v_p/T_{vp}), B_G \supseteq H_1(w_1/T_{w1}) , \ldots , H_q(w_q/T_{wq}), B_H$
    - $M \leftarrow \varnothing$
    - **для** $i = 1$ **до** p
        - $M \leftarrow$ объединить конструкции AMN M и $Tr_{AMN}(T_{vi})$
    - **для** всех АДТ T, которые используются в выражении B_G
        - $M \leftarrow$ объединить конструкции AMN M и $Tr_{AMN}(T)$
    - $M_{REF} \leftarrow \varnothing$
    - **для** $i = 1$ **до** q
        - $M_{REF} \leftarrow$ объединить конструкции AMN M_{REF} и $Tr_{AMN}(T_{wi})$
    - **для** всех АДТ T, которые используются в выражении B_H
        - $M_{REF} \leftarrow$ объединить конструкции AMN M_{REF} и $Tr_{AMN}(T)$

- добавить в M_{REF} секцию, определяющую направление доказательства уточнения
  - REFINES M
- **для** всех операций AMN $g \in M$, что не существует взгляда из V вида
- $T_M.g(u_{1, ..., } u_r) \supseteq T_S.h(u_{1, ..., } u_r)$
  - $M \leftarrow M \setminus \{g\}$
- **для** всех операций AMN $h \in M_{REF}$, что не существует взгляд из V вида
- $T_M.g(u_{1, ..., } u_r) \supseteq T_S.h(u_{1, ..., } u_r)$
  - добавить в M операцию h с пустым телом *skip*
- **для** всех операций AMN $g \in M$, что существует взгляд вида
- $T_M.g(u_{1, ..., } u_r) \supseteq T_S.h(u_{1, ..., } u_r)$
  - переименовать параметры операции $h \in M_{REF}$ в соответствии с именами параметров операции g
  - переименовать операцию $h \in M_{REF}$ в g
- Определить склеивающий инвариант уточнения для всех переменных, используемых в операциях конструкций M, M_{REF} на основании взглядов из V и добавить его в раздел INVARIANT машины M_{REF}
- Загрузить конструкции M, M_{REF} в проект Atelier B типа software development (разработка программного обеспечения) инструментального средства доказательства уточнения Atelier B. Для каждой спецификации произвести проверку типов (*Type Check*) и автоматическую генерацию теорем корректности и уточнения спецификаций (*Generate POs*). Провести доказательство уточнения конструкции M конструкцией M_{REF}: для доказательства теорем применить автоматические средства *Automatic (Force Fast, Force 0, Force 1, Force 2, Force 3)*. Для доказательства теорем, оставшихся недоказанными автоматически, применить средства интерактивного доказательства *Interactive Proof*.
- **если** доказательство завершено успешно **то**
  - $proved \leftarrow proved \cup \{V_0 \mapsto \text{true}\}$
- **иначе если** доказательство не удалось завершить успешно **то**
  - $proved \leftarrow proved \cup \{V_0 \mapsto \text{false}\}$

59 • **если для всех** GLAV-взглядов $V_0 \in V$ выполнено $proved(V_0)$ = true **то**

60 ○ возвратить «Интеграция схем корректна»

61 • **иначе**

62 ○ возвратить «Не удалось доказать корректность интеграции»

63 **конец**

Основные шаги алгоритма состоят в следующем.

К глобальной схеме и схемам источников применяется семантическая трансформация схем канонической модели в AMN. В результате получается контекстная машина и набор машин, соответствующих АТД схем (строки алгоритма 3-7). На основании соответствий АТД устанавливается соотношение между их экстенсионалами, выражается в виде конъюкции предикатов равенства и добавляется в раздел PROPERTIES контекстной машины (8-9).

Далее для каждого взгляда, связывающего классы глобальной схемы и классы схем источников, производятся следующие действия.

Все конструкции AMN, соответствующие используемым во взгляде АТД глобальной схемы, объединяются в одну уточняемую конструкцию M (12-16). Все конструкции AMN, соответствующие используемым во взгляде АТД источников, объединяются в одну уточняющую конструкцию M_{REF} (17-22). Фиксируется направление уточнения (24-26).

Если для какого-то метода g глобальной схемы отсутствует взгляд, связывающий его с методом некоторого АТД источника (т.е. интегрируемые источники не реализуют этот метод), то операция g удаляется из уточняемой машины M, т.к. ее нечем будет уточнять (27-29).

Если для какого-то метода h схемы источника отсутствует взгляд, связывающий его с методом некоторого АТД глобальной схемы, т.е. h не реализует напрямую какой-то метод глобальной схемы (но, возможно, участвует в реализации опосредованно), то в машину M добавляется одноименная операция с пустым телом для формальной корректности уточнения (30-34).

Для каждого метода g глобальной схемы, связанного взглядом с методом h некоторого источника, операция h в машине M_{REF} переименовывается в g (для формальной корректности уточнения). Аналогично в соответствии с параметрами g переименовываются параметры h (35-38).

Определяется склеивающий инвариант уточнения и добавляется в раздел INVARIANT машины M_{REF} (39-41).

Машины M, M_{REF} загружаются в проект Atelier B, производится автоматизированное доказательство уточнения (42-56).

В случае, если для всех взглядов доказательство уточнения завершено успешно, то корректность интеграции схем считается доказанной. В противном случае считается, что корректность интеграции схем доказать не удалось (57-61).

Далее в данном разделе детали метода иллюстрируются на примере применения интеграции схем данных в предметной области *проектирования землепользования*. Рассматривается интеграция одного источника сервиса данных о подборе сортов сельскохозяйственных культур. Иллюстрируются следующие детали метода:

- глобальная схема;
- схема источника;
- соответствие АТД глобальной схемы и релевантных им АТД из схемы источника;
- взгляды, связывающих классы и типы глобальной схемы с классами и типами источника;
- результаты применения семантической трансформации схем канонической модели в AMN к глобальной схеме и схеме источника;
- отношение экстенсионалов АДТ глобальной схемы и АТД схемы источника;
- необходимость объединения конструкций AMN, соответствующих нескольким АТД, в одну конструкцию;
- добавление операций с пустым телом в уточняемую конструкцию;
- переименование операций и параметров уточняющей конструкции;
- определение инварианта уточнения;
- статистика автоматизированного доказательства уточнения.

3.1.2 Определение глобальной схемы и схемы источника данных

Глобальная схема предметной области (рис. 3.2), представленная в виде диаграммы классов UML, включает классы с данными и методами о земельных участках (*sites*), типах почвы (*soilTypes*), культурах (*crops*), сортах (*cultivars*), проектировании землепользования (*landuseDesigners*).

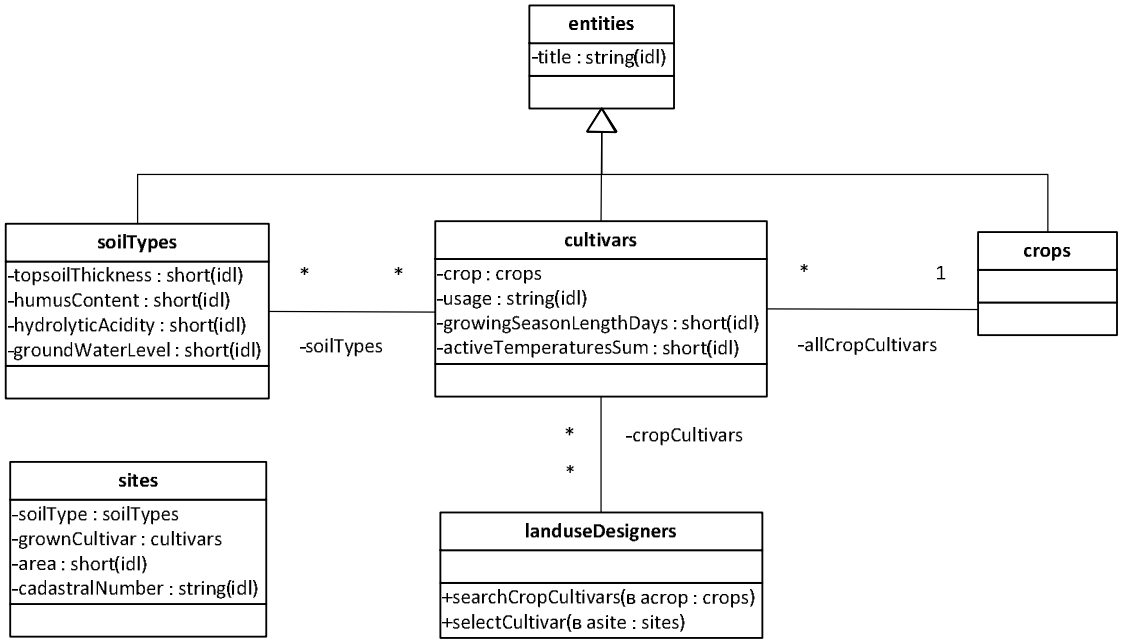

Рисунок 3.2 – Фрагмент концептуальной схемы проектирования землепользования

Каждому классу соответствует АТД его экземпляров. Спецификация АТД *LanduseDesigner,* соответствующего классу *landuseDesigners,* в канонической модели выглядит следующим образом:

```
{ LanduseDesigner; in: type;

  cropCultivars: {set; type_of_element: Cultivar; };
  metaslot init: {( )}; end

  searchCropCultivars:
  { in: function; params: {+acrop/Crop};
    { predicative:
      {
        (this.cropCultivars~ = acrop.allCropCultivars )
      }
    };
  };

  selectCultivar:
  { in: function; params: {+asite/Site};
    { predicative:
      {
        ^is_empty(this.cropCultivars) &
        ex cc/Cultivar ( is_in(cropCultivars, cc) &
                         is_in(cc.soilTypes, asite.soilType) &
                        (asite.grownCultivar~ = cc) )
      }
    };
  };
}
```

Метод *searchCropCultivars* выбирает сорта входной культуры *acrop*, метод *selectCultivar* подбирает для входного участка *asite* сорт, который может выращиваться на почве участка.

Полная спецификация фрагмента концептуальной схемы проектирования землепользования в канонической модели опубликована в репозитории GitHub[57].

Схема источника - сервиса данных о подборе сортов сельскохозяйственных культур (рис. 3.3), представленная в виде диаграммы классов UML, включает классы с данными и методами о земельных участках (*fields*), почвах (*soils*), культурах (*agricultures*), сортах (*varieties*), экономических параметрах сортов (*varietyParameters*), подборе сортов для участков (*varietySelectors*).

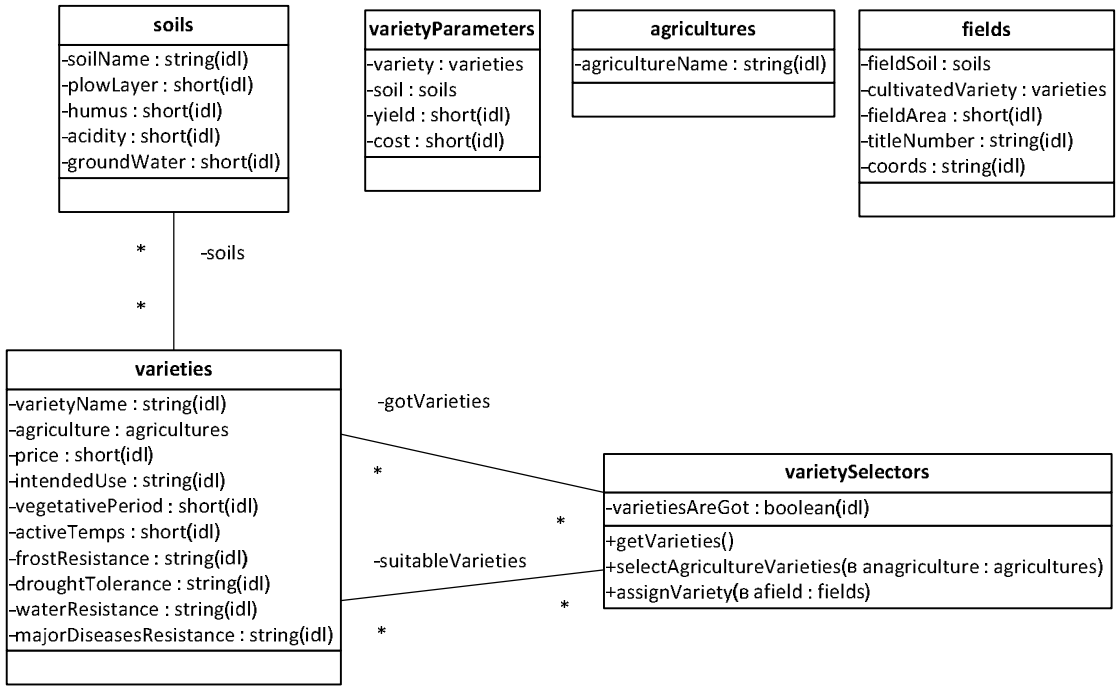

Рисунок 3.3 - Концептуальная схема сервиса данных о подборе сортов сельскохозяйственных культур

Каждому классу соответствует АТД его экземпляров. Спецификация АТД *VarietySelector,* соответствующего классу *varietySelectors,* в канонической модели выглядит следующим образом:

```
{ VarietySelector; in: type;
```

[57] https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis2AMN/Sample/landuse2.syn

```
  gotVarieties: {set; type_of_element: Variety; };
  metaslot init: {( )}; end

  varietiesAreGot: Boolean;
  metaslot init: false; end

  suitableVarieties: {set; type_of_element: Variety; };
  metaslot init: {( )}; end

  getVarieties:
  { in: function;
    { predicative:
      {
        (this.gotVarieties~ =  varieties) & (this.varietiesAreGot~ = true)
      }
    };
  };

  selectAgricultureVarieties:
  { in: function; params: {+anagriculture/Agriculture};
    { predicative:
      {
        (this.suitableVarieties~ =
         { vv/Variety | is_in(this.gotVarieties, vv) &
                        (vv.agriculture = anagriculture) })
      }
    };
  };

  assignVariety:
  { in: function; params: {+afield/Field};
    { predicative:
      {
        ^is_empty(this.suitableVarieties) &
        ex vp0/VarietyParameters (
          is_in(varietyParameters, vp0) &
          is_in(this.suitableVarieties, vp0.variety) &
          is_in(vp0.variety.soils, afield.fieldSoil) &
          (vp0.soil = afield.fieldSoil) &

          all vp1/VarietyParameters (
            is_in(varietyParameters, vp1) &
            is_in(this.suitableVarieties, vp1.variety) &
            is_in(vp1.variety.soils, afield.fieldSoil) &
            (vp1.soil = afield.fieldSoil) ->
            ( (vp0.yield)*((vp0.variety.price) - vp0.cost) ge
              (vp1.yield)*((vp1.variety.price) - vp1.cost) )
          ) &

          (afield.cultivatedVariety~ = v0)
        )
      }
    };
  };

}
```

Метод *getVarieties* получает сорта культур; метод *selectAgricultureVarieties* отбирает сорта входной культуры *anagriculture*; метод *assignVariety* подбирает для участка сорт с максимальной прибылью.

Полная спецификация концептуальной схемы сервиса данных о подборе сортов сельскохозяйственных культур в канонической модели опубликована в репозитории GitHub[58].

3.1.3 Соответствие типов глобальной схемы и схемы источника и взгляды, связывающие схемы

Множество соответствий типов глобальной схемы и релевантных им АТД из схемы источника выглядит следующим образом:

SoilType ↦ *Soil*

Crop ↦ *Agrculture*

Cultivar ↦ *Variety*

Cultivar ↦ *VarietyParameters*

Site ↦ *Field*

LanduseDesigner↦ *VarietySelector*

Множество взглядов, связывающих классы и типы глобальной схемы с классами и типами источника, выглядит следующим образом:

V_1
```
soilTypes(s/SoilType[name, topsoilThickness, humusContent,
hydrolyticAcidity, groundWaterLevel]) ⊇
soils(s/Soil[name: SoilName, topsoilThickness: plowLayer,
humusContent: humus, hydrolyticAcidity: acidity, groundWaterLevel:
groundWater]).
```

V_2
```
crops(c/Crop[name, allCropCultivars]) ⊇
agricultures(c/Agriculture[name: agricultureName]) &
allCropCultivars = { var/Variety |
    varieties(var/Variety[agriculture]) & var.agriculture = c }.
```

V_3
```
cultivars(clt/Cultivar[name, crop, soilTypes, usage,
growingSeasonLengthDays, activeTemperaturesSum]) ⊇
varieties(clt/Variety[name: varietyName, crop: agriculture,
soilTypes: soils, usage: intendedUse, growingSeasonLengthDays:
vegetativePeriod, activeTemperaturesSum: activeTemps]).
```

V_4
```
sites(s/Site[soilType, grownCultivar, area, cadastralNumber]) ⊇
fields(s/Field[soilType: fieldSoil, grownCultivar: cultivatedVariety,
area: fieldArea, cadastralNumber: titleNumber]).
```

[58] https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis2AMN/Sample/landuse2.syn

V_5 `landuseDesigners(ld/LanduseDesigner[cropCultivars]) ⊇ varietySelectors(ld/VarietySelector[cropCultivars: suitableVarieties]).`

V_6 `LanduseDesigner.searchCropCultivars(acrop/Crop) ⊇ VarietySelector.selectAgricultureVarieties(acrop/Agriculture).`

V_7 `LanduseDesigner.selectCultivar(asite/Site) ⊇ VarietySelector.assignVariety(asite/Field).`

3.1.4 Применение семантической трансформации схем канонической модели в AMN к глобальной схеме и схеме источника

Семантическая трансформация Tr_{AMN} схем канонической модели в AMN (раздел 2.7.6) была применена к *глобальной схеме проектирования землепользования* и *схеме источника - сервиса данных о подборе сортов сельскохозяйственных культур.* В результате применения была получена контекстная машина *LanduseDesign_Context* и набор конструкций AMN, соответствующих АТД из схем. Полный набор полученных AMN-спецификаций опубликован в репозитории GitHub[59]. Так, контекстная машина выглядит следующим образом (приведен фрагмент спецификации):

```
MACHINE
    LanduseDesign_Context
SETS
  AVAL;
  OID

ABSTRACT_CONSTANTS
  Obj,
  self,

  extp_Crop, ext_Crop,
  extp_Cultivar, ext_Cultivar,
  extp_Site, ext_Site,
  extp_LanduseDesigner, ext_LanduseDesigner,

  extp_Soil, ext_Soil,
  extp_Agriculture, ext_Agriculture,
  extp_Variety, ext_Variety,
  extp_Field, ext_Field,
  extp_VarietyParameters, ext_VarietyParameters,
  extp_VarietySelector, ext_VarietySelector

PROPERTIES
  Obj : POW(AVAL) &
  self : (Obj >->> OID) &
  extp_Crop : POW(Obj) & ext_Crop : POW(Obj) & extp_Crop <: ext_Crop &
  extp_Cultivar : POW(Obj) & ext_Cultivar : POW(Obj) &
```

[59] https://github.com/sstupnikov/ModelTransformation/tree/master/Synthesis2AMN/Sample/verification/Landuse

```
  extp_Cultivar <: ext_Cultivar &
  extp_Site : POW(Obj) & ext_Site : POW(Obj) & extp_Site <: ext_Site &
  extp_LanduseDesigner : POW(Obj) & ext_LanduseDesigner : POW(Obj) &
  extp_LanduseDesigner <: ext_LanduseDesigner &

  extp_Agriculture : POW(Obj) & ext_Agriculture : POW(Obj) &
  extp_Agriculture <: ext_Agriculture &
  extp_Variety : POW(Obj) & ext_Variety : POW(Obj) &
  extp_Variety <: ext_Variety &
  extp_Field : POW(Obj) & ext_Field : POW(Obj) & extp_Field <: ext_Field &
  extp_VarietyParameters : POW(Obj) & ext_VarietyParameters : POW(Obj) &
  extp_VarietyParameters <: ext_VarietyParameters &
  extp_VarietySelector : POW(Obj) & ext_VarietySelector : POW(Obj) &
  extp_VarietySelector <: ext_VarietySelector &

  extp_Crop = extp_Agriculture & ext_Crop = ext_Agriculture &
  extp_Cultivar = extp_Variety & ext_Cultivar = ext_Variety &
  extp_Site = extp_Field & ext_Site = ext_Field &
  extp_LanduseDesigner = extp_VarietySelector &
  ext_LanduseDesigner = ext_VarietySelector
END
```

В разделе констант ABSTRACT_CONSTANTS определены экстенсионалы АТД схем, в разделе PROPERTIES они типизированы.

Также в разделе PROPERTIES *установлено отношение экстенсионалов АДТ* на основании соответствий АТД. Например, на основании соответствия *Crop* ↦ *Agrculture* установлены отношения

```
extp_Crop = extp_Agriculture & ext_Crop = ext_Agriculture
```

Установление равенства экстенсионалов уточняемого и уточняющего типов основывается на следующем предложении.

Предложение. *При доказательстве уточнения пары типов* T_1, T_2 *в рамках B-технологии экстенсионалы уточняемого (*T_1) *и уточняющего* (T_2) *типов могут быть положены равными.*

Доказательство. Рассмотрим некоторое решение прикладной задачи P с использованием типа T_1. Пусть при решении задачи использовалось множество E_1 экземпляров типа T_1. Множество E_1 является подмножеством экстенсионала E_{T1}. Рассмотрим тип T_2, потенциально уточняющий тип T_1.

Рассмотрим множество E_2 потенциальных экземпляров типа T_2 такое, что

$$card(E_1) = card(E_2)$$

Пусть ϕ — некоторая биекция E_2 на E_1. Пусть потенциальный инвариант уточнения, связывающий атрибуты a типа T_1 и атрибуты b типа T_2, есть $I(a, b)$.

Рассмотрим инвариант J, связывающий атрибуты всех экземпляров типа T_1, используемых при решении P, с экземплярами типа T_2:

$$\forall x\ (x \in E_2 \Rightarrow I(\phi(x).a, x.b))$$

Соответствие между экземплярами устанавливается при помощи биекции ϕ.

Если ввиду данного инварианта уточнения справедливы все теоремы уточнения (раздел 1.7), то тип T_2 можно применять вместо типа T_1 при решении задачи P (экземпляр x используется вместо $\phi(x) \in E_1$).

Очевидно, что вместо того, чтобы при доказательстве уточнения использовать биекцию между E_1 и E_2, можно положить $E_1 = E_2$.

Заметим, что рассуждения никак не зависят от выбора задачи P, а значит, и от выбора подмножества E_1. Следовательно, при доказательстве уточнения типов с использованием B-технологии можно полагать экстенсионалы уточняемого и уточняющего типов равными. Предложение доказано.

□

В качестве примера конструкций AMN, полученных в результате применения трансформации Tr_{AMN} к спецификациям АТД, приведем спецификации, соответствующие АТД глобальной схемы *LanduseDesigner* и АТД схемы источника *VarietySelector*:

```
REFINEMENT LanduseDesigner
SEES  LanduseDesign_Context
INCLUDES
  Cultivar, Site
VARIABLES
  cropCultivars
INVARIANT
  cropCultivars : (ext_LanduseDesigner --> POW(ext_Cultivar))
INITIALISATION
  ANY acropCultivars WHERE
    acropCultivars : (ext_LanduseDesigner --> POW(ext_Cultivar))&
    !(av).((av : ext_LanduseDesigner => acropCultivars(av) = {}))
  THEN
    cropCultivars := acropCultivars
  END

OPERATIONS
  searchCropCultivars(av, acrop) =
  PRE
    av : ext_LanduseDesigner & acrop : ext_Crop
  THEN
    ANY eff1 WHERE
      eff1 : POW(ext_Cultivar) &
      eff1 = {cc | cc : ext_Cultivar & cc : cultivars & crop(cc) = acrop}
    THEN
      cropCultivars(av) := eff1
    END
  END;
```

```
  selectCultivar(av, asite) =
  PRE
    av : ext_LanduseDesigner & asite : ext_Site
  THEN
    ANY eff1 WHERE
      eff1 : ext_Cultivar &  not(cropCultivars(av) = {}) &
      #(cc).(cc : ext_Cultivar &   cc : cropCultivars(av) &
        soilType(asite) : soilTypes(cc) & eff1 = cc)
    THEN
      set_grownCultivar({(asite |-> eff1)})
    END
  END

END
```

Машина *LanduseDesigner* содержит переменную *cropCultivars*, соответствующую атрибуту АТД, а также две операции *searchCropCultivars, selectCultivar*, соответствующие методам АТД.

Машина, соответствующая АТД *VarietySelector* выглядит следующим образом:

```
REFINEMENT VarietySelector
SEES
  LanduseDesign_Context, VarietyParameters
INCLUDES
  Field, Variety
VARIABLES
  gotVarieties, varietiesAreGot, suitableVarieties
INVARIANT
  gotVarieties : (ext_VarietySelector --> POW(ext_Variety)) &
  varietiesAreGot : (ext_VarietySelector --> BOOL) &
  suitableVarieties : (ext_VarietySelector --> POW(ext_Variety))
OPERATIONS
  getVarieties(av) =
  PRE
    av : ext_VarietySelector
  THEN
    ANY eff1, eff2 WHERE
      eff1 : POW(ext_Variety) &  eff2 : BOOL & eff1 = varieties & eff2 = TRUE
    THEN
      gotVarieties(av) := eff1
      ||
      varietiesAreGot(av) := eff2
    END
  END;

  selectAgricultureVarieties(av, anagriculture) =
  PRE
    av : ext_VarietySelector & anagriculture : ext_Agriculture
  THEN
    ANY eff1 WHERE
      eff1 : POW(ext_Variety) &
      eff1 = {vv | vv : ext_Variety & vv : gotVarieties(av) &
                   agriculture(vv) = anagriculture}
    THEN
      suitableVarieties(av) := eff1
    END
  END;
```

```
  assignVariety(av, afield) =
  PRE
    av : ext_VarietySelector &
    afield: ext_Field
  THEN
    ANY eff1 WHERE
      eff1 : ext_Variety & not(suitableVarieties(av) = {}) &
      #(vp0).(vp0 : ext_VarietyParameters & vp0 : varietyParameters &
        variety(vp0) : suitableVarieties(av) &
        fieldSoil(afield) : soils(variety(vp0)) &
        soil(vp0) = fieldSoil(afield) &
        !(vp1).((vp1 : ext_VarietyParameters =>
          (vp1 : varietyParameters & variety(vp1) : suitableVarieties(av) &
              fieldSoil(afield) : soils(variety(vp1)) &
              soil(vp1) = fieldSoil(afield) =>
            ((yield(vp0)) * (((price(variety(vp0))) - cost(vp0)))) >=
             ((yield(vp1)) * (((price(variety(vp1))) - cost(vp1))))))) &
        eff1 = variety(vp0))
    THEN
      set_cultivatedVariety({(afield |-> eff1)})
    END
  END

END
```

Машина *VarietySelector* содержит переменные *gotVarieties, varietiesAreGot, suitableVarieties*, соответствующие атрибутам АТД, а также три операции *getVarieties, selectAgricultureVarieties*, *assignVariety* соответствующие методам АТД.

3.1.5 Формирование уточняемой и уточняющей спецификаций и доказательство уточнения

Конструкции AMN, подлежащие объединению, можно проиллюстрировать на примере взгляда V_2:

```
crops(c/Crop[name, allCropCultivars]) ⊇
agricultures(c/Agriculture[name: agricultureName]) &
allCropCultivars = { var/Variety |
    varieties(var/Variety[agriculture]) & var.agriculture = c }.
```

Правая часть взгляда ссылается на два АТД – *Agriculture, Variety*. Соответствующие им конструкции AMN нужно объединить в одну, а затем провести уточнение ею конструкции AMN, соответствующей АТД *Crop*.

Уточнение будет иллюстрироваться ниже на примере взгляда V_5 :

```
landuseDesigners(ld/LanduseDesigner[cropCultivars]) ⊇
varietySelectors(ld/VarietySelector[cropCultivars: suitableVarieties]).
```

Взгляд утверждает, что класс *landuseDesigners* и АТД *LanduseDesigner* глобальной схемы реализуются классом *varietySelectors* и АТД *VarietySelector* из схемы источника

соответственно. В этом случае в машине *VarietySelector* необходимо объявить, что она уточняет машину *LanduseDesigner*:

```
REFINES LanduseDesigner
```

Заметим, что АТД *VarietySelector* содержит метод *getVarieties*, на который не ссылается ни один взгляд. В этом случае в уточняемую машину LanduseDesigner необходимо *добавить соответствующую операцию с пустым телом:*

```
getVarieties(av) =
PRE
  av : ext_VarietySelector
THEN
    skip
END;
```

Переименование операций и параметров уточняющей машины можно проиллюстрировать на примере взгляда V_7:

```
LanduseDesigner.selectCultivar(asite/Site) ⊇
VarietySelector.assignVariety(asite/Field).
```

Метод *assignVariety* реализует метод *selectCultivar*. Поэтому в машине *VarietySelector* необходимо заменить сигнатуру операции

```
assignVariety(av, afield)
```

на сигнатуру

```
selectCultivar(av, asite)
```

При этом все вхождения параметра *afield* в теле операции необходимо заменить на *asite*.

Композиционная структура набора спецификаций AMN, необходимых для доказательства уточнения вышеприведенного взгляда, изображена на рис. 3.4 (соответствующие спецификации AMN опубликованы в репозитории GitHub[60]).

[60] https://github.com/sstupnikov/ModelTransformation/tree/master/Synthesis2AMN/Sample/verification/Landuse

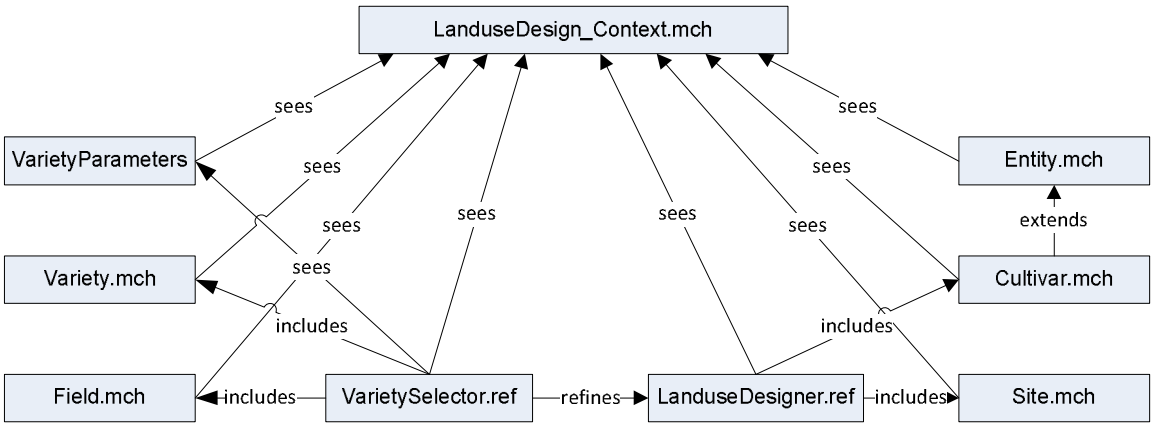

Рисунок 3.4 – Структура связей конструкций AMN

Инвариант уточнения машины *LanduseDesigner* машиной *VarietySelector* выглядит следующим образом:

```
varieties <: cultivars &
!(var).(var: ext_Variety =>
    crop(var) = agriculture(var) & soils(var) = soilTypes(var)) &
!(afield).(afield: ext_Field =>
    fieldSoil(afield) = soilType(afield) &
    cultivatedVariety(afield) = grownCultivar(afield) ) &
!(sel).(sel: ext_VarietySelector =>
      ( (cropCultivars(sel) = {}) <=> (suitableVarieties(sel) = {}) )&
      ( (cropCultivars(sel) /= {}) <=> (suitableVarieties(sel) /= {} &
         gotVarieties(sel) /= {} & varietiesAreGot(sel) = TRUE ) )     )
```

Инвариант связывает переменные, используемые операциями уточняемой и уточняющей машин.

Переменные *cultivars* и *varieties* определены в машинах *Cultivar* и *Variety* соответственно. Согласно взгляду V_3

```
cultivars(clt/Cultivar[name, crop, soilTypes, usage,
growingSeasonLengthDays, activeTemperaturesSum]) ⊇
varieties(clt/Variety[name: varietyName, crop: agriculture, soilTypes:
soils, usage: intendedUse, growingSeasonLengthDays: vegetativePeriod,
activeTemperaturesSum: activeTemps]).
```

каждому экземпляру класса *varieties* соответствует экземпляр класса *cultivars*. Поэтому при доказательстве уточнения можно использовать вложение классов как множеств:

```
varieties <: cultivars
```

Переменные *crop, soilTypes* определены в машине *Cultivar*; переменные *soils, agriculture* определены в машине *Variety*. Для всех экземпляров экстенсионала АТД *Variety* значения атрибутов *soils, agriculture* должны совпадать со значениями атрибутов *crop, soilTypes* соответственно:

```
!(var).(var: ext_Variety =>
      crop(var) = agriculture(var) & soils(var) = soilTypes(var))
```

Переменные *soilType, grownCultivar* определены в машине *Site*; переменные *fieldSoil, cultivatedVariety* определены в машине *Field*. Для всех экземпляров экстенсионала АТД *Field* значения атрибутов *fieldSoil, cultivatedVariety* должны совпадать со значениями атрибутов *soilType, grownCultivar* соответственно:

```
!(afield).(afield: ext_Field =>
    fieldSoil(afield) = soilType(afield) &
    cultivatedVariety(afield ) = grownCultivar(afield) )
```

Переменная *cropCultivars* определена в машине *LanduseDesigner*; переменные *gotVarieties, varietiesAreGot, suitableVarieties* определены в машине *VarietySelector*. Состояние экземпляра АТД *LanduseDesigner*, в котором значение атрибута *cropCultivars* – пустое множество, эквивалентно состоянию экземпляра АТД *VarietySelector*, в котором значение атрибута *suitableVarieties* – пустое множество. Состояние экземпляра АТД *LanduseDesigner*, в котором значение атрибута *cropCultivars* – непустое множество, эквивалентно состоянию экземпляра АТД *VarietySelector*, в котором значения атрибутов *suitableVarieties, gotVarieties* – непустые множества и значение атрибута *varietiesAreGot* – истина:

```
!(sel).(sel: ext_VarietySelector =>
        ( (cropCultivars(sel) = {}) <=> (suitableVarieties(sel) = {}) )&
        ( (cropCultivars(sel) /= {}) <=> (suitableVarieties(sel) /= {} &
           gotVarieties(sel) /= {} & varietiesAreGot(sel) = TRUE ) )
```

Машины *LanduseDesigner* и *VarietySelector* были загружены в проект инструментария Atelier B 4.7.1. Для доказательства уточнения были применены автоматические и интерактивные средства доказательства. Статистика доказательства уточнения приведена в таблице 3.1.

Таблица 3.1 – Количество сгенерированных и автоматически доказанных теорем уточнения спецификаций

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic	
		Force Fast	Force 3
VarietySelector	101	79	95

3.2 Основанная на запросах верификация интеграции схем данных

Раздел основан на работе автора [72].

В данном разделе рассматривается метод верификации корректности интеграции схем данных в модели RDF. Целью метода является формальное доказательство того, что некоторая *система интеграции данных,* использующая в качестве унифицирующей

модели данных язык RDF [10] и его расширения RDFS [129] и OWL [11], осуществляет корректную интеграцию данных из некоторого набора источников данных. Корректность проверяется на наборе запросов пользователя к системе интеграции, выражаемых на языке SPARQL [130] (именно поэтому верификация в данном разделе называется *основанной на запросах*). Корректность доказывается путем отображения схем предметной области, схем источников данных и запросов в формальный язык AMN, и дальнейшим применением автоматизированных средств доказательства уточнения для языка AMN.

Общая схема метода изображена на рис. 3.5.

В *системе интеграции данных* задана *схема предметной области* на языках RDF, RDFS, OWL. В системе интегрированы *N источников данных.* Каждый из источников, в общем случае, использует собственную модель данных: в этой модели представляются *схема* источника и *запросы* к нему. Источник связывается с системой при помощи *интерфейса прикладных программ* (ИПП). Для каждого источника определен набор *представлений* (взглядов), связывающий сущности схемы источника и сущности схемы предметной области. На вход от пользователя система принимает некоторый набор запросов $q_1, \ldots, q_M$, выраженных на языке SPARQL. Запросы выражены в терминах схемы предметной области. Входящий в систему компонент *переписывания запросов* преобразует эти запросы в запросы на языке источников с использованием *представлений*. Например, для *источника 1* формируются запросы $q'_1, \ldots, q'_M$. Запросы отсылаются на источники, исполняются на них, а затем результаты возвращаются в систему, объединяются, и предоставляются пользователю.

Набор запросов $q_1, \ldots, q_M$ должен включать основные типичные запросы пользователей в данной предметной области: именно на этом наборе запросов будет осуществляться верификация интеграции источников данных в системе. Для верификации необходимо определить ряд семантических отображений:

– семантическое отображение конструкций языков RDF, RDFS, OWL в AMN, сопоставляющее произвольной схеме и набору запросов на языке SPARQL, спецификацию вида REFINEMENT (обозначенную на рис. 1 как *SubjectDomain*). Каждому запросу q_i соответствует отдельная операция спецификации $op(q_i)$. Кроме того, общие конструкции языков RDF, RDFS, OWL, не зависящие от конкретной схемы, представляются отдельной спецификацией *ContextRDF* вида MACHINE;

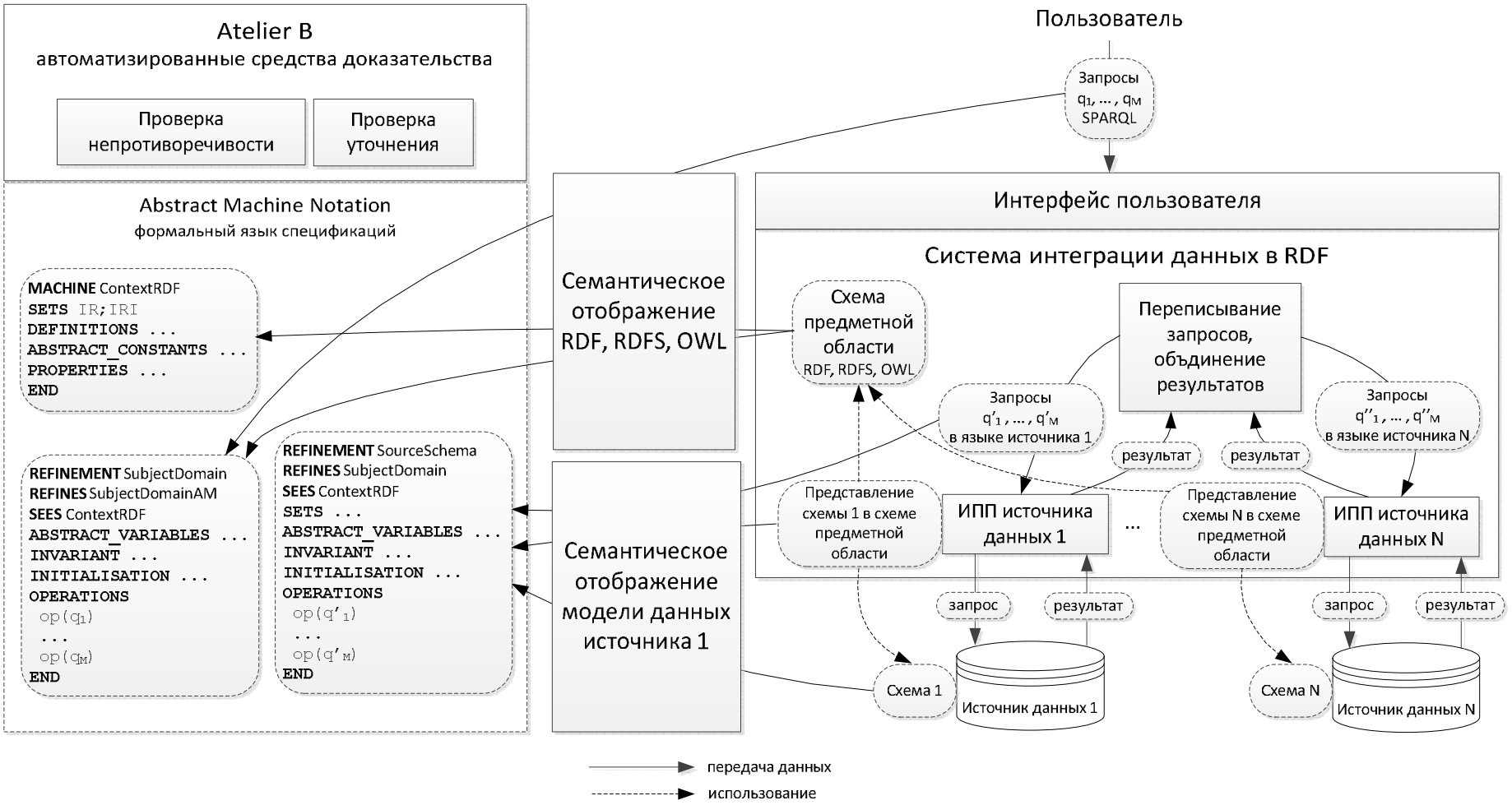

Рисунок 3.5 – Общая схема метода верификации отображений схем и запросов при интеграции данных в модели RDF

– семантические отображения моделей данных источников в AMN, сопоставляющее произвольной схеме и набору запросов в модели данных источника спецификацию вида REFINEMENT (для источника 1 обозначенная на рис. 1 как *SourceSchema*). Каждому запросу q'_j соответствует отдельная операция спецификации $op(q'_j)$. Указывается, что спецификация источника уточняет (REFINES) спецификацию предметной области;

– семантические отображения представлений, связывающих сущности схемы источника и сущности схемы предметной области, в формулы AMN.

Единожды определенное семантическое отображение RDF, RDFS, OWL в AMN затем может быть использовано для верификации интеграции произвольных поддерживаемых источников данных в произвольных системах интеграции, основанных на RDF и SPARQL. То же самое верно и для семантических отображений конкретных моделей данных источников (например, для реляционной модели).

Итак, с использованием семантических отображений схемы данных предметной области и источников, представления и наборы запросов отображаются в спецификации AMN. Далее, спецификации помещаются в проект Atelier B типа *software development* (разработка программного обеспечения). Для каждой спецификации производится проверка типов (*Type Check*), автоматическая генерация теорем корректности и уточнения спецификаций (*Generate POs*). Для доказательства теорем применяются автоматические средства *Automatic (Force Fast, Force 0, Force 1, Force 2, Force 3)*. Для доказательства теорем, оставшихся недоказанными автоматически, применяются средства интерактивного доказательства *Interactive Proof*.

В случае, если все теоремы доказаны, интеграция данных считается *корректной на основе запросов* $q_1, \ldots, q_M$.

Заметим, что на рис. 3.5 проиллюстрирована верификация лишь для *источника данных 1*. Аналогичным образом верификация должна быть проведена для *источников данных 2, … , N*.

Далее в данном разделе детали метода иллюстрируются на примере применения интеграции данных в системе *Ontop* (относящейся к классу основанных на онтологиях систем интеграции – Ontology Based Data Integration, OBDI), рассматриваемом в работе [134]. В качестве предметной области рассматривается *онкологическое медицинское обслуживание*. Схемы предметной области и источников включают структуры данных

пациентов и их заболеваний[61]. Рассматривается интеграция одного источника данных, представленного в реляционной модели. Семантические отображения в AMN иллюстрируются на примерах схем и запросов.

3.2.1 Семантическое отображение конструкций языков RDF, RDFS, OWL в AMN

Resource Definition Framework (RDF) [10] – это модель данных, рекомендованная консорциумом W3C для представления знаний в Веб. Эта модель, в частности, выбрана для эталонной реализации принципов FAIR [254] и представляется наиболее общепринятой [255]. С использованием RDF в качестве унифицирующей модели данных развивается большое количество систем интеграции данных, например, [134] и [256]. Интеграция данных в этих системах основана на переписывании запросов с использованием представлений (взглядов), связывающий сущности схемы источников данных и сущности схемы предметной области, задаваемой в системе интеграции.

Отображение строится в соответствии с нормативными документами – рекомендациями OMG, описывающими семантику RDF и RDFS [257], OWL [11] [230] [258]. Везде, где семантика OWL совпадает с семантикой RDF-RDFS, даются ссылки на семантику RDF-RDFS.

Отображение общих конструкций, не зависящих от конкретных схем иллюстрируется в таблице 3.2. Извлечения из нормативного документа [257] приводятся в левом столбце таблицы (нумерация пунктов взята из исходного документа), в правом столбце приведена спецификация AMN – абстрактная машина *ContextRDF*. Строки из нормативного документа в таблице расположены напротив строк спецификации AMN, реализующих соответствующую семантику.

В разделе множеств (SETS) определяются множество ресурсов *IR* и множество идентификаторов ресурсов *IRI*. Заметим, что к множеству ресурсов в RDF относятся значения встроенных типов (числа, строки и т.д.) и метаобъекты (свойства, классы, и т.д.). В AMN представить одним множеством набор столь разнородных объектов невозможно в силу строгих правил типизации. Поэтому к множеству *IR* в AMN не

[61] https://github.com/ontop/ontop-examples/tree/master/swj-2015

относятся значения встроенных типов и метаобъекты – они представляются отдельными множествами.

Таблица 3.2 - Отображение общих конструкций

Нормативное определение [257]	**Семантика в AMN**
A simple interpretation I is a structure consisting of: 1. A non-empty set IR of resources, called the domain or universe of I. 4. A mapping IS from IRIs into (IR union IP). 5. A partial mapping IL from literals into IR.	**MACHINE** ContextRDF **SETS** IR; IRI **DEFINITIONS** STRING_TYPE == **seq**(0..255) **ABSTRACT_CONSTANTS** IS_IR, IL_String **PROPERTIES** IR /= {} & IS_IR: IRI +-> IR & IS_IR: **FIN**(IS_IR) & IL_String: IRI +-> STRING_TYPE & IL_String: **FIN**(IL_String) **END**

Сопоставление IS идентификаторов и ресурсов задается константной функцией *IS_IR,* сопоставление строковых литералов значениям множества STRING_TYPE (представляющего строковый тип[62]) задается константной функцией *IL_String*.

Типизация функций *IS_IR, IL_String* производится в разделе PROPERTIES. Символ «/=» в AMN обозначает неравенство, «{}» - пустое множество, «:» - принадлежность множеству, «+->» - частичную функцию из одного множества в другое, «&» - логическую конъюнкцию. FIN обозначает множество конечных подмножеств, тем самым типизация *IS_IR: FIN(IS_IR)* задает конечность множества определения и множества значений функции *IS_IR*[63].

---

62 Множество STRING_TYPE определяется в разделе DEFINITIONS как множество последовательностей (seq) чисел от 0 до 255. Это связано с тем, что встроенный тип STRING в Atelier B может быть использован только для передачи параметра в операцию, но не для определения типов констант или переменных. Опыт автора по определению семантики различных языков в AMN показывает, что в большом количестве случаев такого определения достаточно. Для более тщательного определения строкового типа возможно использование, например, подхода, приведенного в [382], и основанного на использовании специальных токенов как ссылок на настоящие строки символов.

63 Задавая ограничения конечности функций, упрощающие представление семантики в AMN, мы опираемся на возможность конечной интерпретации RDF, обсуждаемой в [257], приложение B.

Отображение схемы предметной области. Определение схемы предметной области, представленной в примере онтологией[64] OWL, начинается с определения идентификатора онтологии и входящих в нее классов:

```
Ontology(<http://example.org/hospital>)
Class(:Person)  Class(:Patient)
Class(:Neoplasm)  Class(:NSCLC)  Class(:SCLC)
```

Базу данных госпиталя составляют пациенты (*Patient*) и виды их заболеваний (*Neoplasm, NSCLC, SCLC*). Семантика определения классов приведена в таблице 3.3.

Онтология представляется спецификацией *Hospital*[65] вида REFINEMENT. Из этой спецификации доступны (SEES) определения из спецификации *ContextRDF*. В разделе SETS определяется множество *IC* классов онтологии. В это множество, в частности, входит класс *Thing*, представляющий в OWL множество всех индивидов.

Таблица 3.3 - Семантика определения классов RDFS и OWL

Нормативное определение семантики [257] [11]	**Семантика в AMN**

RDF Schema extends RDF to a larger vocabulary … : … rdfs:Class … [257]
 	
Classes are defined to be things of type rdfs:Class, and the set of all classes in an interpretation will be called IC. [257]
 	
The semantic conditions are stated in terms of a mapping ICEXT from IC to the set of subsets of IR. [257]
The class with IRI owl:Thing represents the set of all individuals. [11]
	REFINEMENT Hospital
	SEES ContextRDF
	SETS
	IC = { Thing,
	Person, Patient,
	Neoplasm, NSCLC, SCLC
	};
	ABSTRACT_VARIABLES
	ICEXT
	INVARIANT
	ICEXT: IC --> **POW**(IR) &
	ICEXT(Thing) = IR
	INITIALIZATION
	ICEXT:= {
	Thing |-> IR,
	Patient |-> {}, Person |-> {},
	Neoplasm |-> {},
	NSCLC |-> {}, SCLC |-> {}
	}

В разделе ABSTRACT_VARIABLES декларируется и в разделе INVARIANT типизируется переменная *IEXT,* задающая интерпретацию классов как множеств ресурсов. Здесь «-->» обозначает тотальную функцию, *POW(X)* - множество подмножеств множества *X,* «|->» - упорядоченную пару. Язык AMN требует задания

64 https://github.com/ontop/ontop-examples/blob/master/swj-2015/PatientOnto.owl

65 Полные AMN-спецификации для рассматриваемого примера располагаются в репозитории https://github.com/sstupnikov/ModelTransformation/tree/master/RDF2AMN/example

начальных значений переменных в разделе INITIALIZATION, и переменная *IEXT* инициализируется функцией, сопоставляющей всем классам, кроме класса *Thing*, пустые множества. Классу *Thing* сопоставляется все множество ресурсов (индивидов) *IR*.

Следующая часть онтологии определяет иерархию классов:

```
SubClassOf(:NSCLC :Neoplasm)
SubClassOf(:SCLC :Neoplasm)
SubClassOf(:Patient :Person)
```

Здесь классы двух видов рака легких *NSCLC* (Non-Small Cell Lung Carcinoma) и *SCLC* (Small Cell Lung Carcinoma) – это подклассы класса *Neoplasm*. Класс пациентов (*Patient*) – подкласс класса людей. Семантика иерархии классов приведена в таблице 3.4.

Таблица 3.4 - Семантика иерархии классов.

Нормативное определение семантики [257]	**Семантика в AMN**
RDFS semantic conditions. IEXT(I(rdfs:subClassOf)) is transitive and reflexive on IC. If <x,y> is in IEXT(I(rdfs:subClassOf)) then x and y are in IC and ICEXT(x) is a subset of ICEXT(y).	**INVARIANT** ICEXT(NSCLC) <: ICEXT(Neoplasm) & ICEXT(SCLC) <: ICEXT(Neoplasm) & ICEXT(Patient) <: ICEXT(Person)

Отношение класс-подкласс, фактически, представляется отношением множество-подмножество (символ «<:» в AMN), обладающее необходимыми транзитивностью и рефлексивностью.

Далее в онтологии определяются свойства, их области определения и значений:

```
ObjectProperty(hasNeoplasm)
DataProperty(:hasName)
ObjectProperty(:hasStage)
ObjectPropertyDomain(:hasNeoplasm :Patient)
ObjectPropertyRange(:hasNeoplasm :Neoplasm)
ObjectPropertyDomain(:hasStage :Neoplasm)
DataPropertyRange(:hasName xsd:string)
```

Объектное свойство *hasNeoplasm* сопоставляет пациенту его вид заболевания, объектное свойство *hasStage* сопоставляет заболеванию его стадию, свойство *hasName* строкового типа данных задает имя индивида. Семантика свойств приведена в таблице 3.5 (нумерация пунктов в левом столбце взята из исходного документа [257]).

Таблица 3.5 - Семантика свойств.

Нормативное определение семантики [5]	**Семантика в AMN**
A simple interpretation I is a structure consisting of: 2. A set IP, called the set of properties of I. 4. A mapping IS from IRIs into (IR union IP) 5. A partial mapping IL from literals into IR. 3. A mapping IEXT from IP into the powerset of IR x IR i.e. the set of sets of pairs <x, y> with x and y in IR. RDFS semantic conditions. If <x, y> is in IEXT(I(rdfs:domain)) and <u, v> is in IEXT(x) then u is in ICEXT(y). If <x, y> is in IEXT(I(rdfs:range)) and <u, v> is in IEXT(x) then v is in ICEXT(y).	**SETS** IP = { hasNeoplasm }; IP_String_Range = { hasName }; IP_CancerStage_Range = { hasStage }; CancerStage = { stageI, stageII, stageIII, stageIIIa, stageIIIb, stageIV } **ABSTRACT_CONSTANTS** IS_IP, IS_IP, IL_CancerStage **PROPERTIES** IS_IP: IRI +-> IP & IS_IP: **FIN**(IS_IP) & IL_CancerStage: IRI +-> CancerStage & IL_CancerStage: **FIN**(IL_CancerStage) & **ABSTRACT_VARIABLES** IEXT, IEXT_String_Range, IEXT_CancerStage_Range **INVARIANT** IEXT: IP --> **POW**(IR*IR) IEXT_String_Range: IP_String_Range --> **POW**(IR*STRING_TYPE) & IEXT_CancerStage_Range: IP_CancerStage_Range --> **POW**(IR*CancerStage) & !(xx, yy).(xx: IR & yy: IR & (xx|->yy):IEXT(hasNeoplasm) => xx: ICEXT(Patient)) & !(xx, yy).(xx: IR & yy: IR & (xx|->yy):IEXT(hasNeoplasm) => yy: ICEXT(Neoplasm)) & !(xx, yy).(xx: IR & yy: CancerStage & (xx|->yy):IEXT_CancerStage_Range(hasStage) => xx: ICEXT(Neoplasm)) **INITIALISATION** IEXT:= { hasNeoplasm |-> {} } || IEXT_String_Range:= { hasName |-> {} } || IEXT_CancerStage_Range:= { hasStage |-> {} }

В силу невозможности объединять в AMN в одном множестве разнородные сущности, свойства представляются несколькими множествами. Во-первых, множество *IP* содержит объектные свойства, в область значений которых входят индивиды, не являющимися значениями встроенных типов (к таким свойствам относится *hasNeoplasm*). Во-вторых, для каждого встроенного или перечислимого типа определяется отдельное множество, содержащее свойства, областью значения которых является этот тип. Так, например, для свойств строкового типа определяется множество

IP_String_Range, для свойств перечислимого типа *CancerStage* – множество *IP_CancerStage_Range*.

Для сопоставления идентификаторов ресурсов и свойств определяется конечная константная функция *IS_IP,* для сопоставления идентификаторов ресурсов и множества значений типа *CancerStage* - конечная константная функция *IS_CancerStage*.

Функции интерпретации свойств определяются в разделе переменных. Так, функция *IEXT* сопоставляет каждому объектному свойству из *IP* множество пар элементов из *IR*. Аналогично определяются функции интерпретации свойств строкового типа (*IEXT_String_Range*) и типа *CancerStage* (*IEXT_CancerStage_Range*). Типизация этих функций производится в инварианте. Знак «*» обозначает декартово произведение множеств. Инициализация функций производится пустыми множествами.

Ограничения на область определения и область значений свойств представляются формулами, соответствующими семантике RDFS и становятся частью инварианта.

Отображение запросов. Каждому запросу в AMN-спецификации сопоставляется отдельная операция в разделе OPERATIONS. В качестве примера рассмотрим запрос на языке SPARQL, возвращающий имена пациентов со стадией заболевания IIIa (таблица 3.6). Строки запроса на SPARQL в таблице расположены напротив строк спецификации AMN, реализующих соответствующую семантику.

Таблица 3.6 - Семантика SPARQL-запроса

Запрос на SPARQL	Семантика запроса в AMN

SELECT ?name **WHERE** {
 	
?pp a :Patient ;
:hasName ?name ;
 	
:hasNeoplasm ?tumor .
 	
?tumor a :Neoplasm ;
:hasStage :stage-IIIa .
}
	OPERATIONS
	res <-- patientsWithTumorOfStageIIIa =
	BEGIN
	res:=
	{ rcrd |
	rcrd: struct(name: STRING_TYPE) &
	#(pp, tumor).(
	pp: ICEXT(Patient) &
	(pp |-> rcrd'name):
	IEXT_String_Range(hasName) &
	(pp |-> tumor):
	IEXT(hasNeoplasm) &
	tumor: ICEXT(Neoplasm) &
	(tumor |-> stageIIIa):
	IEXT_CancerStage_Range(hasStage))
	}
	END
	END

Результат операции *res* формируется при помощи конструкции выделения множества *{rcrd | F(rcrd)}*: множество включает те и только значения переменной *rcrd*, которые обращают формулу *F* в истину. Тип переменной *rcrd* – это кортеж (*struct*),

включающий единственный элемент *name* строкового типа. Доступ к элементам кортежа осуществляется с помощью операции «'»: *rcrd'name*. Переменные, используемые в теле запроса (*pp, tumor*) связываются в формуле квантором существования (символ «#»). Принадлежность переменной классу представляется с использованием функции *ICEXT,* например, *pp: ICEXT(Patient)*. Ограничения на значения свойств представляются с использованием функций интерпретации свойств *IEXT, IEXT_String_Range, IEXT_CancerStage_Range*.

3.2.1 Семантическое отображение конструкций реляционной модели данных в AMN

Схема источника данных представляется в AMN спецификацией *Patients* вида REFINEMENT. Схема включает единственную таблицу *tbl_patient*. Семантическое отображение ее определения в AMN приведено в таблице 3.7.

Таблица 3.7 - Семантика определения таблиц

Определение таблицы в SQL	Семантика в AMN
CREATE TABLE "tbl_patient" (patientid INT NOT NULL PRIMARY KEY, name VARCHAR NOT NULL, type BOOLEAN NOT NULL, stage INT NOT NULL)	**REFINEMENT** Patients **REFINES** CancerRef **SEES** ContextRDF **ABSTRACT_VARIABLES** tbl_patient **INVARIANT** tbl_patient: **POW**(struct(patientid: **INT**, name: STRING_TYPE, type: **BOOL**, stage: **INT**)) & !(pt1, pt2).(pt1: tbl_patient & pt2: tbl_patient & pt1 /= pt2 => pt1'patientid /= pt2'patientid) **INITIALISATION** tbl_patient:= {}

Таблица представляется одноименной переменной. Тип этой переменной – множество, содержащее кортежи (*struct*). Названия и типы атрибутов кортежей соответствуют названиям и типам атрибутов таблицы. Отдельной формулой в составе инварианта представляется ограничение первичного ключа. Переменная инициализируется пустым множеством.

Запрос на языке SPARQL, приведенный в предыдущем разделе, переписывается в системе интеграции с использованием представлений в запрос на языке SQL. Запрос и его семантика в AMN приведены в таблице 3.8.

Таблица 3.8 - Семантика SQL-запроса

Запрос на SQL	Семантика запроса в AMN

SELECT name
FROM "tbl_patient"
 	
WHERE stage = 4 AND
  type = false
	OPERATIONS
	res <-- patientsWithTumorOfStageIIIa =
	BEGIN
	  res:=
	  { rcrd |
	  rcrd: struct(name: STRING_TYPE) &
	  #ptnt.(ptnt: tbl_patient &
	  ptnt'name = rcrd'name
	  ptnt'stage = 4 &
	  ptnt'type = FALSE
	  )
	  }
	END END

Результат операции *res* формируется при помощи конструкции выделения множества. Тип элементов множества – кортежи (*struct*), включающие единственный элемент *name* строкового типа. Переменная *ptnt*, пробегающая по кортежам таблицы, связывается в формуле квантором существования.

3.2.2 Семантические отображения представлений в формулы AMN

Семантическое отображение одного из представлений приведено в таблице 3.8. Соответствующая формула AMN становится конъюнктивной частью инварианта уточняющей спецификации *Patients.ref* (в репозитории на Github выложена спецификация, содержащая семантическое отображение всех представлений).

Таблица 3.8 - Семантика представления (взгляда)

Представление	Семантика в AMN

mappingId Neop
 	
source
SELECT patientid **FROM**
"tbl_patient"
 	
target
inst:ds1/
{patientid}
 	
:hasNeoplasm
 	
inst:ds1/
neoplasm/
{patientid}
.
	!(patient).(
	  patient: tbl_patient
	  =>
	  #(patientInd, neoplasmInd).(
	  patientInd: ICEXT(Patient) &
	  IS_IR(personIRI(patient'patientid)) =
	  patientInd &
	  (patientInd |-> neoplasmInd):
	  IEXT(hasNeoplasm) &
	  neoplasmInd: ICEXT(Neoplasm) &
	  IS_IR(neoplasmIRI(patient'patientid)) =
	  neoplasmInd
	  )
	)

Исходная часть представления (*source*) – это SQL-запрос, а целевая (*target*) – набор RDF-триплетов. Неформальная семантика представления *Neop* состоит в том, что каждому пациенту с идентификатором *patientid* из таблицы *tbl_patient* в онтологии

отвечает экземпляр класса *Patient* с идентификатором *inst:ds1/patientid* и экземпляр класса *Neoplasm* с идентификатором *inst:ds1/neoplasm/patientid*, причем эти экземпляры связаны свойством *hasNeoplasm*.

Представление отображается в логическую импликацию, в посылке которой стоит конъюнкция предикатов, соответствующих исходной части представления, в заключении - конъюнкция предикатов под квантором существования, соответствующих целевой части. В данном случае посылка состоит из единственного предиката, типизирующего переменную *patient*, пробегающую по кортежам таблицы *tbl_patient*. Квантор существования в заключении связывает переменные, отвечающие экземплярам классов онтологии. Под квантором заключены предикаты принадлежности переменных классам, предикат связи экземпляров свойством *hasNeoplasm* и предикаты связи экземпляров с их идентификаторами, задаваемые при помощи функции *IS_IR* из спецификации *ContextRDF*. Вспомогательные константные функции *personIRI* и *neoplasmIRI,* используемые при этом, фактически, отвечают целевым выражениям *inst:ds1/{patientid}* и *inst:ds1/neoplasm/{patientid}* соответственно. Эти функции определяются в спецификации в разделе констант:

```
ABSTRACT_CONSTANTS
    personIRI,
    neoplasmIRI
PROPERTIES
    personIRI: 1..NN >-> IRI & ran(personIRI) <: dom(IS_IR) &
    neoplasmIRI: 1..NN >-> IRI & ran(neoplasmIRI) <: dom(IS_IR) &
    ran(personIRI) /\ ran(neoplasmIRI) = {}
```

Функции задают соответствие между целым числом – идентификатором пациента и идентификатором ресурса из множества IRI (например, числу *451* соответствует идентификатор *http://example.org/hospital/instances/ds1/451* согласно шаблону). Область определения функций ограничена достаточно большим числом *NN* из соображений конечности интерпретации.

3.2.3 Доказательство корректности интеграции

Для доказательства корректности интеграции в рассматриваемом примере, в инструментарии *Atelier B* был создан проект разработки ПО. Созданные в результате применения семантических отображений спецификации *ContextRDF.mch, Hospital.ref, Patients.ref* были помещены в проект. Была проведена автоматическая проверка синтаксической корректности спецификаций, проверка типов, автоматически сгенерированы теоремы корректности и уточнения спецификаций. Для автоматического доказательства теорем были применены средства Automatic (Force Fast, Force 0, Force 1). Статистика автоматического доказательства приведена в таблице 3.9.

Таблица 3.9 - Количество сгенерированных и автоматически доказанных теорем непротиворечивости и уточнения спецификаций

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic		
		Force Fast	Force 0	Force 1
ContextRDF.mch	0	0	0	0
Hospital.ref	86	25	81	81
Patients.ref	72	42	68	68

Как можно заметить, большая часть теорем доказана автоматически. Для доказательства остальных теорем необходимо применять интерактивные средства.

3.3 Родственные работы

Изложенные в данной главе методы опираются на определение формальной семантики канонических моделей систем интеграции данных.

Одними из самых широкоиспользуемых методов определения семантики моделей данных и языков запросов, обладающих сходными с канонической моделью чертами, являются методы денотационной и аксиоматической семантик [259][260][261][262][263].

Формальная семантика с использованием логики предикатов первого порядка с равенством для метамодели IDEF1X определена в стандарте FIPS PUB 184 [118]. Классам сущностей и связям ставятся в соответствие предикатные символы; определяется предикат принадлежности сущности классу (*exists*), предикат значения атрибута (*has*). С использованием предикатов определяются аксиомы теории. Определены правила порождения теории как совокупности аксиом по модели IDEF1X.

Достаточно давно ведутся исследования по формальному определению языка визуализации, спецификации, проектирования и документирования информационных систем Unified Modeling Language (UML), а также языка Object Constraint Language (OCL), предназначенного для спецификации дополнительных требований, которые невозможно или сложно выразить графическими средствами языка UML. В этих работах обычно индуктивно рассматривается представление конструкций языков UML/OCL в некотором формальном языке, основанном на математической логике, свойства конструкций представляются аксиомами языка.

В работах France [264][265] для определения семантики языка UML применяется формальный язык Z. Основное внимание уделено интерпретации в языке Z диаграмм классов UML. Разделяются понятия *интенсионала класса* как набора свойств, общих для всех объектов класса и *экстенсионала класса* как множества объектов класса. Для

экстенсионала и интенсионала вводятся различные *схемы* языка Z. Интенсиональная схема описывает класс в терминах его атрибутов, типы атрибутов интерпретируются как предопределенные множества. Экстенсиональная схема содержит переменную, интерпретирующую множество экземпляров класса, а также функцию, ставящую в соответствие объектам их атрибуты.

Работы Kim, Carrington [266][267] посвящены определению семантики диаграмм классов UML в языке Object-Z – объектно-ориентированном расширении Z. Язык Object-Z содержит понятия уникальной идентифицируемости объектов и наследования, что позволяет избавиться от экстенсиональных схем при интерпретации классов UML. Логическим продолжением данных работ является результат Roe, Broda, Russo [268] по представлению семантики подмножества OCL в Object-Z. Операции UML интерпретируются как отдельные *операционные схемы*, параметры операций интерпретируются как входные и выходные коммуникационные переменные схем. Инварианты OCL представляются либо как предикаты в схемах классов, либо как предикаты в схеме системы. Пред и постусловия операций UML представляются предикатами соответствующих операционных схем.

Работы Lano, Bicarregui [269][270][271] посвящены формальному определению языка UML в языке RAL (Real-time Action Logic) [272]. Спецификации RAL представляют собой структурированные теории, основанные на логике первого порядка и включающие средства описания временных свойств систем. Описана семантика конструкций UML, составляющих диаграммы классов, состояний, диаграммы последовательностей действий, активностей: классов, ассоциаций, атрибутов, операций, отношения обобщения, сообщений. Каждый UML-класс представляется отдельной теорией, содержащей одноименный предопределенный тип – множество всевозможных объектов класса. Операции UML представляются действиями (actions) соответствующих теорий RAL.

Семантика подмножества языка OCL в логике первого порядка рассмотрена в работах Beckert, Schmitt [273][274]. Ограничения в терминах OCL, налагаемые на диаграмму классов UML, представляются как теории логики первого порядка. Определены общие принципы построения формул теории по диаграмме классов. Наибольшее внимание уделено представлению в виде логических термов операций над коллекциями.

Существуют подходы к определению семантики диаграмм OMT и языков UML и OCL в языке AMN. Одной из первых работ, в которой были предложены некоторые принципы моделирования в AMN объектных диаграмм OMT, является работа Facon,

Laleau, Nguyen [275]. Meyer, Souquieres [276] на основе работы Lano [277] предложили схемы отображения в AMN структурных понятий UML – классов, атрибутов, ассоциаций и их свойств. Butler, Snook [278] [279] на основе алгоритмов, изложенных в работе [276], разработали инструментальное средство U2B для отображения ограниченного подмножества спецификаций классов UML в AMN. Предназначение U2B состоит в автоматизации процесса доказательства сохранения инвариантов классов операциями классов при помощи B-технологии.

В работе Ledang, Souquieres [280] предложен подход к моделированию операций классов UML в AMN. Во-первых, каждая UML-операция моделируется абстрактной операцией AMN, специфицирующий ожидаемые эффекты UML-операции. Во-вторых, если операция реализована в диаграмме взаимодействия или в диаграмме активностей, соответствующая абстрактная AMN-операция уточняется AMN-операцией реализации. Основная идея подхода состоит в группировке в одной машине представления UML-операции и представления тех данных, с которыми она оперирует. В работе Ledang, Souquieres [281] предложены схемы отображения конструкций языка OCL в AMN. Рассмотрено представление типов OCL типами AMN, представление операций над типами OCL выражениями и предикатами AMN, представление пред и постусловий операций UML, выраженных в OCL, обобщенными подстановками AMN. Схемы отображения UML, предложенные Ledang, Souquieres были частично реализованы [282] на платформе ArgoUML.

Известные подходы к моделированию диаграмм OMT, языков UML и OCL в AMN предназначены в основном для автоматизации доказательства непротиворечивости инвариантов классов и сохранения инвариантов классов операциями классов при помощи B-технологии.

Разработанный и реализованный в данной работе метод представления семантики канонической модели в AMN предназначен для обеспечения автоматизации доказательства уточнения канонических спецификаций с использованием B-технологии для доказательства корректности интеграции схем данных. При определении отображения использовались следующие особенности, характерные для рассмотренных подходов к моделированию языков OMT, UML, OCL в AMN: принцип моделирования абстрактных типов данных множествами своих потенциальных значений (экстенсионалами), моделирование атрибутов типов функциями, определенными на экстенсионалах типов, моделирование иерархии типов ограничениями на структуру экстенсионалов типов, моделирование методов операциями абстрактных машин AMN.

В отличие от существующих подходов к определению семантики моделей данных в AMN, построенное отображение обладает свойством *однородности*: AMN-спецификация, полученная в результате отображения канонической спецификации, может быть использована и как уточняющая спецификация, и как уточняемая. Для этого свойства был разработан способ моделирования структуры спецификаций типов канонической модели при помощи средств композиции абстрактных машин AMN. Способ нацелен на *модуляризацию* спецификации – разбиение ее на части, уточнение которых можно доказывать независимо. Для адекватного представления в AMN системы типов и коллекций канонической модели был усовершенствован *экстенсиональный принцип* определения семантики АТД (представление АТД *экстенсионалом* – предопределенным множеством возможных экземпляров типа). Разработано представление в AMN предикативных спецификаций функций канонической модели (формул, содержащих смешанные пред и постусловия функций). Предикативные спецификации функций могут содержать предикаты-коллекции, вызовы функций (в том числе с побочными эффектами), кванторы существования и всеобщности.

К родственным *методу отображения RDF в AMN* следует отнести работы, связанные с формальным представлением стека языков RDF.

В работе [283] конструкции RDF и RDFS отображаются в логические правила и факты, которые могут быть загружены в интерпретатор языка Даталог. Для валидации RDF-графов могут быть применены инструменты вывода, основанные на механизме SLD-резолюций.

В работе [284] представлена аксиоматизация RDF и RDFS в виде теории логики предикатов первого порядка. Аксиомы представлены в формате KIF (Knowledge Interchange Format).

Язык OWL поддерживается в инструментарии Hets (The Heterogeneous Tool Set) [285], обеспечивающем синтаксический разбор, статический анализ и управление доказательством для большого набора логик. Диалект OWL DL отображается в каркас Common Logic, стандартизованный ISO.

В работе [286] предлагается логическая интерпретация RDF и OWL-DL, нацеленная на унифицированное представление семантики обоих языков. Интерпретация основана на Эрбрановских моделях RDF-графов. Предлагается также семантика SPARQL-запросов, основанных на простых графовых шаблонах.

В работе [287] предлагается интерпретация RDF и RDFS с типами данных в языке F-Logic и дескриптивной логике DL-Lite$_R$.

В работе [288] представлено отображение большого фрагмента диалекта OWL 2 Full в логику первого порядка для решения задач логического вывода над OWL. На тестовом наборе для языка OWL 2 Full проведено сравнительное тестирование решения задач вывода с использованием ряда систем доказательства в логике первого порядка, систем поиска конечных моделей первого порядка и систем автоматического вывода в дескриптивных логиках.

В работе [289] предложена экстенсиональная семантика для представительного фрагмента RDFS, а также корректная и полная система правил вывода для этого фрагмента. Доказаны результаты по сложности задачи вывода над фрагментом и проведены эксперименты над рядом тестовых онтологий.

В отличие от упомянутых родственных работ, в данной главе для формального определения языков стека RDF используется язык AMN, нацеленный на доказательство уточнения спецификаций. Это накладывает ряд ограничений на выразительные средства языка, которые необходимо учитывать при построении отображения. Определение семантики строится таким образом, чтобы оставаться максимально терминологически близким к нормативной семантике RDF, описанной в рекомендациях W3C [230] [257] [258]. Рассматривается представление в AMN семантики всего стека RDF в совокупности с запросами языка SPARQL.

4 Метод верификации интеграции данных

Метод, рассматриваемый в данной главе, нацелен на определение формальной семантики и верификации на уровне интеграции собственно данных. При этом предполагается, что необходимое определение семантики и верификация на уровне моделей данных и схем уже проведена. Предполагается также, что процесс интеграции данных представляется в виде набора операций трансформации данных на SQL-подобном языке. При этом свойства процесса интеграции в целом не выражаются явно в программе интеграции, определяются лишь правила преобразования данных в конкретных операциях.

В процессе интеграции собственно данных обычно выделяют различные этапы: трансформация данных, разрешение сущностей (entity resolution) [290][291] и слияние данных (data fusion) [189][292]. *Трансформация данных* подразумевает их преобразование из исходной схемы (схемы коллекции - источника данных) в целевую (единую интегрированную схему). Вопросы разработки трансформо-центричных программных систем как многоуровневых структур, состоящих из отдельных трансформаций или множеств альтернативных трансформаций, рассматривались еще в классической книге Yourdon, Constantine [293]. Под *разрешением сущностей* обычно понимают выделение и связывание информации об одной и той же сущности реального мира из разных коллекций данных [290]. Под *слиянием сущностей* понимают комбинацию различных представлений одной и той же сущности реального мира в единое представление [189]. Этапы интеграции данных могут состоять из большого количества операций, и не обязательно идут в указанном порядке. Поэтому более правильно говорить, что процесс интеграции данных представляет собой поток работ, деятельности которого представляют собой отдельные операции трансформации структурированных (типизированных) данных, разрешения или слияния сущностей. Ряд обобщенных операций слияния данных выделен в работе [292].

Наряду с платформами распределенной параллельной обработки данных разрабатываются высокоуровневые языки программирования, которые могут быть использованы (или прямо предназначены) для трансформации данных, разрешения и слияния сущностей в рамках этих платформ. К таким языкам относятся, в частности, Jaql [294], Pig Latin [295], Highlevel Integration Language (HIL) [296]. Распределенное исполнение программ, написанных на таких языках, в среде Hadoop достигается путем компиляции высокоуровневых программ в программы на императивных языках (например, Java), которые, в свою очередь, исполняются с использованием средств поддержки в Hadoop распределенных вычислительных моделей [297].

Идея метода, предлагаемого в данной главе, состоит в том, чтобы сообщить высокоуровневым программам интеграции данных семантику в языке спецификаций, поддержанном средствами формального автоматического и/или интерактивного доказательства: для языка интеграции данных строится его отображение в язык спецификаций. Свойства программ, подлежащие проверке (корректность программ интеграции данных), связывающие состояния совокупности источников данных и состояние целевой интегрирующей базы данных, представляются экспертом в виде выражений выбранного языка спецификаций. Например, корректность может состоять в том, что для всех объектов из источников в целевой базе данных в результате применения программы интеграции появляются их правильные образы (отсутствуют потерянные данные). Затем, с использованием формальных средств доказательства, спецификация, выражающая семантику конкретного потока работ интеграции данных, проверяется на соответствие необходимым свойствам.

Для иллюстрации метода в качестве языка интеграции данных выбран HIL – язык, разработанный компанией IBM, поставляемый в составе решения IBM InfoSphere Master Data Management [298], и используемый, например, в проектах интеграции финансовых и социальных данных [296]. HIL – это язык высокого уровня для описания сложных потоков обработки данных, включающих операции трансформации данных, разрешения сущностей и слияния данных. Данные при этом могут поступать из больших коллекций данных, представленных в различных схемах. Язык был применен, в частности, в проекте интеграции данных из социальных сетей, была продемонстрирована масштабируемость применения языка как по разнообразию схем данных, так и по объему данных. По сравнению с традиционными средствами построения процессов извлечения-трансформации-загрузки (ETL) данных, основанных на реляционной модели и языке SQL, язык HIL предлагает значительно более гибкие декларативные средства интеграции данных, нацеленные на разрешение сущностей и слияние данных.

В качестве языка спецификаций выбран язык Нотация абстрактных машин (AMN) [35], поддержанный средствами формального доказательства [206]. Выбор AMN мотивирован возможностями этого языка по спецификации операций трансформации данных и опытом автора по использованию AMN для определения формальной семантики. Для формализации семантики языка HIL недостаточно лишь средств верификации процессов (таких, как конечные автоматы или сети Петри) – необходимы средства верификации сложных операций (деятельностей, составляющих процессы). Вместо AMN возможно использование и других языков, предоставляющих аналогичные возможности, например, RAISE или Z.

4.1 Формальная семантика языка разрешения сущностей и интеграции HIL

Раздел основан на работах автора [95] [76].

Подход к определению формальной семантики языка HIL демонстрируется в данном разделе на примере интеграции финансовых данных, публикуемых Комиссией по ценным бумагам и биржам (SEC) США. Пример рассматривался в работах, посвященных системе интеграции Midas [299] и языку HIL [296]. Целью рассматриваемого потока работ интеграции данных (являющегося частью общей задачи интеграции финансовых данных) является формирование коллекции сущностей *Person,* представляющей информацию о лицах, управляющих ведущими компаниями США (рис. 4.1). Исходными для потока являются две коллекции: *InsiderReportPerson* (для краткости, *IRP*) и *JobChange*. В первой коллекции содержатся данные, извлеченные из внутренних отчетов компаний о заработной плате сотрудников. Во второй коллекции содержатся данные, извлеченные из отчетов о принятии на работу или смене позиций сотрудников компаний.

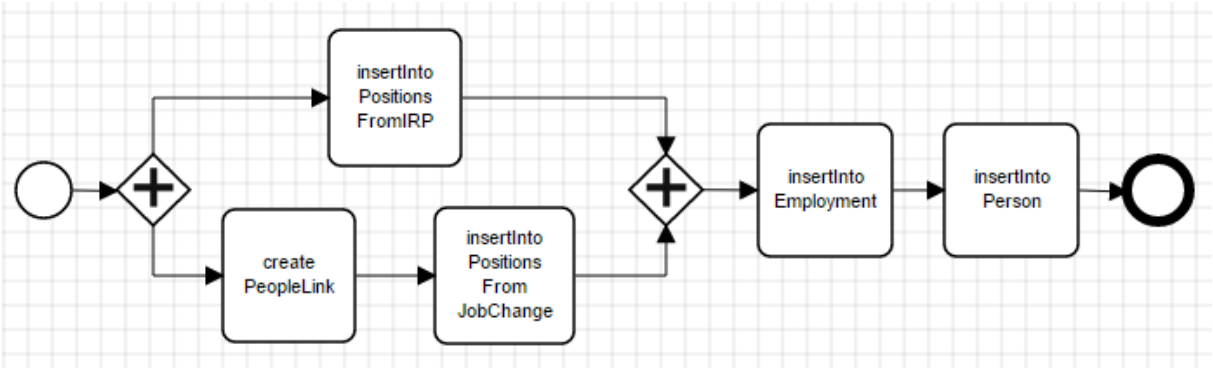

Рисунок 4.1 – Поток работ интеграции данных об управляющих компаниями лицах

Само извлечение информации из неструктурированных данных не является предметом настоящей работы. Поток включает пять операций:

– *insertIntoPositionsFromIRP* (извлечение данных о позициях управляющих лиц из коллекции *IRP* и включение их в промежуточную коллекцию *Positions*);

– *createPeopleLink* (создание коллекции *PeopleLink* связей между управляющими лицами, упомянутыми во внутренних отчетах, и записями о смене позиций);

– *insertIntoPositionsFromJobChange* (извлечение данных о позициях управляющих лиц из коллекции *JobChange* и включение их в промежуточную коллекцию *Positions*);

– *insertIntoEmployment* (включение данных о найме сотрудников компаниями в промежуточную коллекцию *Employment*);

– *insertIntoPerson* (включение данных об управляющих лицах и их местах работы в коллекцию *Person*).

Среди рассматриваемых операций *createPeopleLink* является характерным примером разрешения сущностей, а операции *insertIntoEmployment* и *insertIntoPositionsFromJobChange* – слияния сущностей.

Следует отметить, что в языке HIL отдельным операциям трансформации, разрешения или слияния сущностей не присваиваются имена, операции выражаются в виде правил в SQL-подобном синтаксисе. Приведенные выше имена присвоены соответствующим правилам для удобства рассуждений.

Также нужно отметить, что порядок исполнения операций в языке HIL напрямую не фиксируется [296]. Этот порядок определяется при компиляции с тем ограничением, что все промежуточные коллекции, используемые операциями, должны быть материализованы, т.е. исполнены операции, осуществляющие включение данных в эти коллекции. Поток работ, изображенный на рис. 4.1 в графическом представлении языка BPMN, отражает неявную семантику ограничений на последовательность исполнения операций в рассматриваемом примере. Ромб со знаком «+» означает разделение потока на параллельные ветви и слияние параллельных ветвей. Например, операция *insertIntoEmployment* должна быть исполнена после операций *insertIntoPositionsFromIRP, createPeopleLink, insertIntoPositionsFromJobChange,* поскольку она использует коллекции *PeopleLink* и *Positions,* формируемые соответствующими операциями.

Семантика основных конструкций языка HIL будет определена ниже в языке AMN и проиллюстрирована на примерах.

Семантическое отображение определяется с использованием набора семантических функций. В соответствии с принципами денотационной семантики, семантические функции отображают элементы *синтаксических доменов* языка HIL в конструкции языка AMN. Синтаксические домены HIL структурированы при помощи метамодели *Ecore* (являющейся реализацией OMG Essential Meta-Object Facility) применяемой в программном каркасе Eclipse Modeling Framework [196]. Разработана модель *HIL.ecore* [300], конформная метамодели *Ecore*. Разработанная модель формализует абстрактный синтаксис HIL игнорируя «синтаксический сахар» конкретного синтаксиса. Например, операция вставки формализована синтаксическим доменом *Insert* (фактически, это класс метамодели *Ecore*) (рисунок 4.2). Домен *Insert* – это поддомен домена *Statement*. Элементы домена *Insert* (операции вставки) включают секции *into, select from, where*, формализованные как синтаксические домены *Expression, LabeledExp, AliasedEntity, Predicate* соответственно.

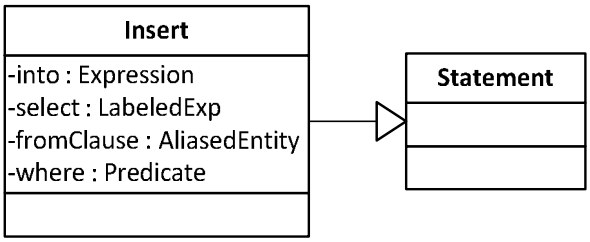

Рисунок 4.2 – Пример абстрактных синтаксических доменов языка HIL

Конкретный синтаксис[66] HIL, связывающий абстрактный синтаксис с необходимым для повышения читаемости кода «синтаксическим сахаром», формализован с использованием инструментария EMFText [219]. Например, синтаксический домен *Insert* связан с «синтаксическим сахаром» следующим образом:

```
Insert ::=
    "insert" "into" into
    "select" "[" select ("," select)* "]"
    "from" fromClause ("," fromClause)*
    ("where" where)? ";" ;
```

Нетерминальные символы конкретной грамматики соответствуют синтаксическим доменам, символ “*” означает повторение, символ “?” означает опциональность.

В данном разделе имена семантических функций начинаются с символа “s”, параметры семантических функций заключаются в квадратные скобки. Семантические функции, синтаксические домены, метапеременные выделены *курсивом*. Терминальные символы грамматик языков HIL и AMN выделены прямым шрифтом.

4.1.1 Семантика программ на языке HIL

Каждая программа на HIL представляется в AMN отдельной конструкцией вида REFINEMENT [35] (такого рода конструкции допускают наиболее широкие возможности определения спецификаций).

Семантическая функция для синтаксического домена *Program* определяется следующим образом:

[66] https://github.com/sstupnikov/ModelTransformation/blob/master/HIL2AMN/metamodel/HIL.cs

```
sProgram[Statement*] ≙
REFINEMENT HILProgramSemantics
ABSTRACT_CONSTANTS sAbstractConstants[Statement*]
PROPERTIES sProperties[Statement*]
VARIABLES sVariables[Statement*]
INVARIANT sInvariant[Statement*]
INITIALISATION sInitialisation[Statement*]
OPERATIONS sOperations[Statement*]
END
```

Эта функция формирует семантическую спецификацию для всей программы HIL. Отдельные секции спецификации (например, секции переменных или операций) формируются отдельными семантическими функциями, определенными ниже.

4.1.2 Семантика типов сущностей, типов индексов и функций

Типы сущностей и типы индексов представляются в AMN переменными абстрактных машин в разделе VARIABLES, и типизируются в разделе INVARIANT. Например, *тип сущностей* IRP (атрибут *name* отвечает имени персоны, *cik* – уникальному идентификатору персоны в документации SEC, *bdate* – дате рождения, *company* – имени компании, *companyCIK* – идентификатору компании, *title* – занимаемой должности, флаги *isOfficer* и *isDirector* – является ли персона служащим или управляющим лицом компании) представляется в конструкции *PersonFusion* следующим образом (Таблица 4.1).

Таблица 4.1 – Пример определения семантики типа сущности

Конструкция HIL	**Конструкция AMN**

`declare IRP: set[`
`  name: string,`
`  cik: int,`
`  bdate: string;`
`  company: string,`
`  title: string,`
`  isOfficer: Boolean,`
`  isDirector: Boolean];`
	`VARIABLES IRP, …`
	`INVARIANT`
	`  IRP: POW(struct(`
	`  name: STRING_TYPE, cik: INT,`
	`  bdate: STRING_TYPE, company: STRING_TYPE,`
	`  title: STRING_TYPE, isOfficer: BOOL,`
	`  isDirector: BOOL`
	`  )) & …`
	`INITIALISATION`
	`  IRP := {} || …`

Для типа *IRP* в разделе переменных объявляется одноименная переменная, которая типизируется в инварианте как подмножество (POW) множества всех структур (struct) – кортежей. Кортежи состоят из атрибутов, соответствующих атрибутам типа *IRP*. Между встроенными типами HIL и AMN установлено взаимооднозначное соответствие, используемое при типизации атрибутов структур в AMN. Например, тип *string* языка HIL соответствует типу STRING_TYPE в AMN, тип Boolean – типу BOOL и т.д.

Эти отображения формализуются следующими семантическими функциями:

```
sVariables[ declare name: type; ] ≙ name
sInitialisation[ declare name: type; ] ≙ name := {}
sInvariant[ declare name: type; ] ≙ name: sType[type]
sType[ set type ] ≙ POW(sType[type])
sType[ "[" element* "]" ] ≙ sRecordElement[element*]
sRecordElement[name: type] ≙ name: sType[type]
sType[string] ≙ STRING_TYPE
sType[int] ≙ INT
sType[Boolean] ≙ BOOL
```

Здесь семантические функции принимают обобщенные синтаксические определения HIL в качестве параметров. Например, функция *sVariables* принимает в качестве параметра синтаксическое выражение из синтаксического домена *Declaration* (объявление). Синтаксически, объявление выглядит как "declare *name: type*;". Слова, выделенные прямым шрифтом (например, "declare") – это ключевые слова языка HIL. Слова, выделенные курсивом (например, "*name*" и "*type*") - это метапеременные, принимающие значения из других синтаксических доменов (например, метапеременная *type* принимает значения из домена *Type*).

Тип индекса fmap в HIL представляет собой отображение из множества экземпляров типа ключа в экземпляры типа значения. Например, семантика типа *Positions* (в котором ключ задается парой ⟨идентификатор персоны *cik,* имя компании *company*⟩, а значение – множеством должностей *set[title]*) определяется в AMN следующим образом (Таблица 4.2).

Таблица 4.2 – Пример определения семантики типа индекса

Конструкция HIL	**Конструкция AMN**
declare Positions: fmap [cik: int, company: string] to set [title: string];	VARIABLES Positions, … INVARIANT Positions: struct(cik: INT, company: STRING_TYPE) +-> POW(struct(title: STRING_TYPE)) INITIALISATION Positions := {} || …

Тип *Positions* представляется в конструкции *PersonFusion* одноименной переменной, типизируемой в инварианте. Для представления отображения используется конструктор частичной функции AMN, обозначаемый символами «+->». Типы ключа и значения, как и в случае с типом сущностей *IRP*, также представляются при помощи конструкторов структур и множества подмножеств.

Для отображения конструкции *fmap* языка HIL в AMN используется конструктор частичной функции “+->”. Семантическая функция *sType* определена на конструкциях *fmap* следующим образом:

```
sType[fmap fromType to toType] ≙ sType[fromType] +-> sType[toType]
```

Функции HIL представляются в AMN одноименными *абстрактными константами*. Константы типизируются в секции PROPERTIES спецификации AMN. Например, представление в языке AMN функции *compareName* приведено в таблице 4.3.

Таблица 4.3 – Пример определения семантики функции

Конструкция HIL	**Конструкция AMN**

declare compareName:
  function
  string, string
  to boolean;
	ABSTRACT_CONSTANTS compareName
	PROPERTIES
	  compareName:
	  ((STRING_TYPE * STRING_TYPE) --> BOOL)

Это отображение формализуется при помощи семантических функций следующим образом:

```
sAbstractConstants[declare name: function fromType* to toType] ≙ name
sProperties[declare name: function fromType1, fromType2 to toType] ≙
  name: ((sType[fromType1]*sType[fromType2]) --> sType[toType])
```

4.1.3 Семантика операций трансформации, разрешения и слияния сущностей

Каждому правилу языка HIL ставится в соответствие операция абстрактной машины AMN.

4.1.3.1 Семантика операций разрешения сущностей

Примером операции такого рода является создание коллекции *PeopleLink* в HIL.Ее представление в AMN выглядит следующим образом (Таблица 4.4).

Сущности формируемой коллекции определяются в секции *select* и включают атрибуты *cik* (идентификатор персоны), *docID* (номер документа, где встречается упоминание о смене должности персоны), *span* (место в документе, где встречается упоминание о смене должности). Данные извлекаются (секция *from*) из коллекций *IRP* и *JobChange* (происходит соединение коллекций). В секции *match* происходит предварительный отсев кортежей, полученных в результате соединения коллекций: имена персон *name* должны в обеих коллекциях совпадать с точностью до нормализации (функция *normName*). В секции *check* происходит дополнительная проверка отобранных в *match* кортежей: в случае, если в обеих коллекциях имеются данные о дате рождения персоны, даты должны совпадать.

Таблица 4.4 – Пример определения семантики операции разрешения сущностей

Конструкция HIL	Конструкция AMN

create link PeopleLink as
select [cik: pp.cik,
  docID: jj.docID,
  span: jj.span]
from IRP pp, JobChange jj,
  pp.emp ee
match using
rule1:
  (ee.company = jj.company) and
  (com-pareName(pp.name,jj.name)=
  true),
rule2: (normName(pp.name) =
  normName(jj.name))
check
  if not(null(jj.bdate))
  and not(null(pp.bdate))
  then (jj.bdate = pp.bdate);
	createPeopleLink =
	SELECT TRUE = TRUE
	THEN PeopleLink :=
	{rr | #(pp, jj, ee).(pp : IRP &
	  jj : JobChange & ee : pp'emp &
	  rr = rec(cik: pp'cik,
	  docID: jj'docID, span: jj'span) &
	  (ee'company = jj'company &
	  compareName(pp'name, jj'name) =
	  TRUE or
	
	normName(pp'name)=normName(jj'name))&
	  (not((jj'bdate = null_string)) &
	  not((pp'bdate = null_string)) =>
	  jj'bdate = pp'bdate))};
	  state'PeopleLinkCreated := TRUE
	END

Основные идеи отображения состоят в следующем. Переменной *PeopleLink* присваивается множество, образованное при помощи конструкции выделения множества *{rr | F(rr)}* (здесь *rr* – имя переменной, отвечающей элементу множества, *F(rr)* – формула, которая должны обращаться в истину на элементах множества). Переменная *rr* типизируется типом записи *rec*, обладающим необходимыми для типа коллекции атрибутами. Для каждой из соединяемых коллекций заводится по переменной (*pp* и *jj*), переменные связываются квантором всеобщности # и типизируются как принадлежащие соответствующим коллекциям: *pp: IRP & jj: JobChange*. Условия из секций *match* и *check* представляются соответствующими условиями на переменные *pp* и *jj*. Необходимые функции (например, *normName*) определяются как константы в разделе ABSTRACT_CONSTANTS конструкции *PersonFusion* и типизируются в разделе PROPERTIES:

```
ABSTRACT_CONSTANTS normName, …
PROPERTIES
  normName: STRING_TYPE --> STRING_TYPE
```

Функция *normName* типизируется тотальной функцией из строк в строки.

Описанное выше отображение формализуется при помощи семантических функций следующим образом:

```
sOperations[ create link name as
  select select* from fromClause*
  match using match1, match2
  check check1 check2 ] ≙
sCreateLinkOperationName[name] =
SELECT sStateCheckPredicate[name]
THEN
  name := { rr | #(sExistentialVariables[fromClause*]).(
    sExistVarTyping[fromClause*] &
    rr = rec(sRecord[select*]) &
    (sMatch[match1] or sMatch[match2]) &
    sCheck[check1] & sCheck[check2]
  )};
  sStateSetSubstitution[name]
END

sCreateLinkOperationName[name] ≙ concatenateStrings("create", name)
sExistentialVariables[exp alias] ≙ alias
sExistVarTyping[exp alias] ≙ alias: sExpression[exp]
sRecord[label: exp] ≙ label: sExpression[exp]
sMatch[name: predicate] ≙ sPredicate[predicate]
sCheck[if ifPred then thenPred] ≙
    sPredicate[ifPred] => sPredicate[thenPred]
sPredicate[predicate1 and predicate2] ≙
    sPredicate[predicate1] & sPredicate[predicate2]
sPredicate[not(null(exp))] ≙ not(sExpression[exp] = null_string)
sPredicate[exp1 = exp2] ≙ sExpression[exp1] = sExpression[exp2]
sExpression[Identifier] ≙ Identifier
sExpression[recordID.elementId] ≙ recordID'elementId
sExpression[functionId(parameter1, parameter2)] ≙
    functionId(sExpression[parameter1], sExpression[parameter2])
```

Семантические функции манипулирования переменными состояния (*sStateCheckPredicate, sStateSetSubstitution*) определены в нижеследующих пунктах. Вспомогательная функция *concatenateStrings* – это конкатенация строк.

4.1.3.2 Семантика включения данных в коллекции с типами индексов

Рассмотрим отличительные особенности представления в AMN операций включения данных в коллекции с типами индексов на примере операций *insertIntoPositionsFromIRP* и *insertIntoEmploymentFromJobChangePeopleLink* (Таблица 4.5).

Знак «!» после идентификатора пополняемой коллекции *Positions* означает, что пара атрибутов *cik* и *company*, следующих после него, будет использоваться в коллекции в качестве ключа. В AMN такое пополнение коллекции *Positions* осуществляется в операции *insertIntoPositionsFromIRP*:

Множество *Positions* пополняется при помощи операции объединения множеств «\/». Пополняющее множество, как и в правиле из предыдущего пункта, образуется при помощи конструкции выделения множеств по переменной *rr*. Отличие состоит в том, что

переменная *rr* типизируется типом пары. Первым элементом пары является запись *rec(cik: ii'cik, company: ii'company),* соответствующая типу ключа коллекции Positions. Вторым элементом пары является множество названий должностей *title*, извлеченных из коллекции *IRP* с условием совпадения атрибутов ключа. Элементы пары соединяются знаком «|->».

Таблица 4.5 – Пример определения семантики операции включения данных в коллекции

Конструкция HIL	**Конструкция AMN**

insert into Positions
! [id: i.cik,
company: i.company]
select [title:
normTitle(i.title)]
from IRP i;
	insertIntoPositionsFromIRP =
	SELECT …
	THEN
	  Positions := Positions \/
	  { rr | #ii.(ii: IRP &
	  rr = rec(cik: ii'cik, company: ii'company) |->
	  { rr1 | #ii1.(ii1: IRP & ii1'cik = ii'cik &
	  ii1'company = ii'company &
	  rr1 = rec(title: normTitle(ii1'title))) })
	  };
	  …
	END;

insert into
  Employment!
  [cik: LL.cik]
select
[company:
  jj.company,
  positions:
  Positions!
  [cik: LL.cik,
  company:
  jj.company]
]
from
  JobChange jj,
  PeopleLink LL
where
  (jj.docID= LL.docID)
  and
  (jj.span = LL.span);
	insertIntoEmploymentFromJobChangePeopleLink =
	SELECT state'PeopleLinkCreated = TRUE &
	  state'
	  PositionsFromJobChangePeopleLinkInserted =
	  TRUE &
	  state'PositionsFromIRPInserted = TRUE
	THEN
	  Employment := (Employment \/
	  {rr1 | #(jj1, LL1).(jj1 : JobChange &
	  LL1 : PeopleLink &
	  rr1 = (rec(cik: LL1'cik) |->
	  {rr | #(jj, LL).(jj : JobChange &
	  LL : PeopleLink &
	  rr = rec(company: jj'company,
	  positions: Positions(
	  rec(cik: LL'cik,
	  company: jj'company))) &
	  LL'cik = LL1'cik & jj'docID = LL'do-cID &
	  jj'span = LL'span)})})});
	  state'
	  EmploymentFromJobChangePeopleLinkInserted :=
	  TRUE
	END;

Описанное выше отображение формализуется при помощи семантических функций следующим образом:

```
sOperations[ insert into name!exp as
  select select* from fromClause*
  where predicate ] ≙
concatenateStrings("insertInto", name, "From",
  sInsertOperationName[fromClause*]) =
SELECT sStateCheckPredicate[name fromClause*]
THEN
  name := name \/ { rr1 | #(sExistentialVariables1[fromClause*]).(
    sExistVarTyping1[fromClause*] &
    rr1 = rec(sRecord1[exp]) |->
    { rr | #(sExistentialVariables[fromClause*]).(
             sExistVarTyping[fromClause*] & rr = rec(sRecord[select*]) &
             sEqualityPredicate[exp] & sPredicate[predicate]) }
  )};
  sStateSetSubstitution[name fromClause*]
END

sInsertOperationName[Identifier1 alias1, Identifier2 alias2] ≙
  concatenateStrings(Identifier1, Identifier2)
sExistentialVariables1[exp alias] ≙ concatenateStrings(alias, "1")
sExistVarTyping1[exp alias] ≙
  concatenateStrings(alias, "1"): sExpression[exp]
sRecord1[label: exp] ≙ label: sExpression1[exp]
sExpression1[recordID.elementId] ≙
  concatenateStrings(recordID, "1")'elementId
sEqualityPredicate[exp] ≙ sExpression[exp] = sExpression1[exp]
```

4.1.4 Семантика порядка исполнения операций

Для отслеживания фактов исполнения операций в конструкции *HILProgramSemantics* заводится переменная *state*, отвечающая состоянию потока работ:

```
VARIABLES … state, …
INVARIANT
  state: struct(PeopleLinkCreated: BOOL,
         PositionsFromIRPInserted: BOOL,
         PositionsFromJobChangeInserted: BOOL,
         EmploymentInserted: BOOL,
         PersonInserted: BOOL
    ) & …
INITIALISATION
  state:= rec(PeopleLinkCreated: FALSE,
   PositionsFromIRPInserted: FALSE,
   PositionsFromJobChangePeopleLinkInserted: FALSE,
   EmploymentFromJobChangePeopleLinkInserted: FALSE,
   PersonFromIRPInserted: FALSE) || …
```

Переменная типизируется в инварианте типом структуры *struct*, атрибуты которого отражают завершение каждой из операций потока работ. Предусловием исполнения каждой из операций является завершение всех других необходимых операций. Также, завершающим действием каждой операции является установка флага завершения

операции переменной *state* в значение TRUE. Для операции *insertIntoEmployment* соответствующие предусловия и присваивания выглядят в AMN следующим образом:

```
OPERATIONS
…
insertIntoEmployment =
SELECT state'PositionsFromIRPInserted = TRUE &
  state'PositionsFromJobChangeInserted = TRUE
THEN
  …
  state'EmploymentInserted := TRUE
END;
```

Тело операции образовано конструкцией SELECT [35]. Это означает, что действия, указанные после ключевого слова THEN исполняются только в том случае, если предикат после ключевого слова SELECT обращается в истину. Операция *insertIntoEmployment* может быть исполнена только тогда, когда обе операции *insertIntoPositionsFromIRP* и *insertIntoPositionsFromJobChange* исполнены (т.е. флаги *state'PositionsFromIRPInserted* и *state'PositionsFromJobChangeInserted* имеют значение TRUE). После завершения операции флаг *state'EmploymentInserted* также должен быть установлен в значение TRUE.

Начальная инициализация переменной state производится в разделе INITIALISATION, флаги завершения всех операций устанавливаются в значение FALSE:

```
INITIALISATION
    state := rec(PeopleLinkCreated: FALSE,
        PositionsFromIRPInserted: FALSE,
        PositionsFromJobChangeInserted: FALSE,
        EmploymentInserted: FALSE,
        PersonInserted: FALSE)
  || …
```

Семантические функции, формирующие предикат типизации, инициализацию переменной состояния и подстановку присваивания переменной состояния определены следующим образом:

```
sStateTypingPredicate[Statement1 Statement2] ≙
  state: struct(sStateAttributeName[Statement1]: BOOL,
                sStateAttributeName[Statement2]: BOOL)
sStateAttributeName[create link name as
  select select* from fromClause* match using match* check check*] ≙
  concatenateStrings(name, "Created")
sStateAttributeName[insert into name!exp as
  select select* from fromClause* where predicate] ≙
  concatenateStrings(name, "From", sInsertOperationName[fromClause*],
    "Inserted")
sStateInitialisation[Statement1 Statement2] ≙
  state:= rec(sStateAttributeName[Statement1]: FALSE,
              sStateAttributeName[Statement2]: FALSE)
```

```
sStateSetSubstitution[name fromClause*] ≜
  state'sStateAttributeName[name fromClause*] := TRUE
```

Предикаты предусловий формируются с использованием семантической функции *precedingStatements* (предшествующие операции). Эта функция отображает каждую операцию HIL во множество предшествующих ей операций. Функция вычисляется следующим образом: для любой операции *s*, операция *t* принадлежит множеству *precedingStatements(s)* тогда и только тогда, когда существует коллекция *c* такая, что *c* создается или пополняется операцией *t* и *c* используется в секциях *from* или *select* операции *s*. Таким образом, коллекция *c* должна быть материализована перед исполнением операции *s*, и коллекция *c* пополняется операцией *t*. Это означает, что операция *t* должна быть исполнена перед операцией *s*.

Семантическая функция, формирующая предикат предусловия, выглядит следующим образом:

```
sStateCheckPredicate[Statement] ≜
  precedingStatements(Statement) = {s1, … , sn} ⇒
  state'sStateAttributeName[s1] = TRUE & … &
  state'sStateAttributeName[sn] = TRUE
```

Здесь $P \Rightarrow E$ – это вспомогательное выражение денотационной семантики, принимающее значение выражения *E,* когда предикат *P* обращается в истину.

Семантические функции, определенные в данном разделе, реализованы [300] в виде трансформации на языке ATLAS Transformation Language (ATL) [192] (программный код трансформации опубликован в репозитории GitHub[67]); разработана декларативно-императивная семантическая трансформация, преобразующая программы языка HIL в спецификации языка AMN. Большая часть семантического отображения реализована с использованием декларативных правил. Некоторые семантические функции (например, *precedingStatements*) реализованы при помощи императивных конструкций.

---

67 https://github.com/sstupnikov/ModelTransformation/tree/master/HIL2AMN

4.2 Применение формальной семантики для языка разрешения сущностей и слияния данных для верификации потоков работ интеграции данных

Программа интеграции данных на языке HIL, подлежащая верификации, автоматически преобразуется в семантическую спецификацию на языке AMN с использованием разработанной ATL-трансформации [300].

Затем формулы, отражающие свойства потока работ интеграции данных, подлежащие верификации, добавляются вручную в инвариант конструкции AMN, отражающей семантику потока работ интеграции данных (в рассматриваемом примере такой конструкцией является *PersonFusion*). Затем семантические спецификации помещаются в программное средство Atelier B [206], где осуществляется проверка синтаксиса спецификаций и корректности типизации переменных и термов. Затем осуществляется автоматическая генерация теорем, отражающих сохранение инварианта при выполнении операций спецификации. Теоремы доказываются с использованием автоматических и интерактивных средств доказательства Atelier B.

В качестве примера свойства потока работ интеграции данных рассмотрим одно из возможных свойств сохранения информации при извлечении данных из исходных коллекций *IRP* и *JobChange* и формировании коллекции *Person*. Свойство формулируется в виде формулы логики предикатов и добавляется в инвариант конструкции *PersonFusion*:

```
(state'PersonFromIRPInserted = TRUE =>
  !(nn, cc, tt).(
    (#(irp).(irp: IRP & nn = irp'name &
        cc = irp'company & tt = irp'title) or
    #(jc).(jc: JobChange & nn = jc'name &
        cc = jc'company & tt = jc'appointedAs)) =>
    #(pers).(pers: Person & pers'name = normName(nn) &
        #(ee).(ee: pers'emp & ee'company = cc &
          #(pos).(pos: ee'positions & pos'title = normTitle(tt)) ))))
```

В формуле утверждается, что по завершении потока работ (завершающая операция – *insertIntoPersonFromIRP*) выполняется следующее свойство: для любых наборов, состоящих из имени персоны, названия компании и должности, встречающихся совместно в записях коллекции *IRP* либо в записях коллекции *JobChange*, в коллекции *Person* найдется запись с такими же значениями имени персоны (с точностью до нормализации), названия компании и должности. Таким образом, из исходных коллекций извлекается вся возможная информация о местах работы и должностях персон.

Спецификация конструкции *PersonFusion* была помещена в средство Atelier B, была осуществлена синтаксическая проверка и проверка типов спецификации. Для

каждой из операций спецификации были автоматически сгенерированы по три теоремы, в совокупности утверждающие сохранение инварианта при завершении операций. Для доказательства теорем были применены автоматические и интерактивные средства доказательства.

4.3 Спецификация и реализация разномодельных правил интеграции данных

Раздел основан на работах автора [96] [77].

Метод определения формальной семантики конструкций языка HIL, рассмотренный выше в данной главе, позволяет использовать язык HIL для реализации верифицируемых трансформаций интеграции данных. Однако, для спецификации интеграции данных могут быть удобны языки более высокого уровня, например, основанные на логических правилах. В данном разделе рассматривается метод *спецификации правил интеграции данных с использованием рекомендации W3C - логического диалекта RIF-BLD [141] и их реализации на языке HIL*. Тем самым обеспечивается возможность верификации свойств спецификаций интеграции данных, выраженных на высокоуровневых логических правилах.

RIF-BLD включает достаточно широкий спектр возможностей спецификации, в частности, *позиционные термы* и *термы с именованными аргументами* (обобщающие понятие терма в логике первого порядка), *фреймовые термы* (выражающие утверждения о структуре объектов), *классификационные термы, термы равенства, внешние термы* (использующиеся для ссылок на сущности, рассматриваемые как «черные ящики» в пределах спецификации). Это позволяет использовать в одном правиле сущностей из разных коллекций, представленных в *разных моделях данных*. Рассматриваются правила, в голове которых могут присутствовать лишь предикаты, соответствующие сущностям целевой схемы, а в теле – предикаты, соответствующие сущностям исходных схем.

Ниже в данном разделе рассмотрены общие принципы спецификации и образцы реализации правил интеграции. Далее метод иллюстрируется на данных из предметной области организации поисковых и спасательных операций в Арктической зоне, представленных в различных моделях: XML и реляционная модель, документная модель, графовая модель.

4.3.1 Общие принципы спецификации правил интеграции данных с использованием RIF-BLD и их реализации в языке HIL

Основные принципы спецификации правил интеграции данных с использованием RIF-BLD состоят в следующем:

– правила интеграции имеют вид импликации, посылка которой называется *телом*, а заключение – *головой*;

– голова и тело правил представляют собой формулы, связывающие предикаты операциями конъюнкции или дизъюнкции;

– в теле правила допускаются предикаты, соответствующие сущностям только исходных схем, в голове – только целевой схеме;

– для описания свойств кортежей, принадлежащих отношениям реляционной модели, в правилах используются предикаты с именованными аргументами [141];

– для описания свойств структур данных произвольной степени вложенности (при помощи которых представляются данные различных нереляционных моделей – XML, документной, графовой и т.д.) используются фреймовые предикаты и предикаты членства [141];

– связь значений атрибутов сущностей исходных и целевой схем осуществляется при помощи переменных;

– свободные переменные, использующиеся в теле правила, связываются внешним квантором всеобщности;

– свободные переменные в голове правила, не встречающиеся в теле (фактически, они соответствуют данным, не определенным явно в исходных коллекциях), связываются квантором существования;

– нетривиальные предикаты-условия (отличные от равенства) и функции разрешения конфликтов определяются как внешние термы [141].

Основные принципы реализации правил интеграции данных на RIF-BLD в языке HIL состоят в следующем:

– сущности исходных и целевой схем, функции разрешения конфликтов и нетривиальные предикаты-условия объявляются при помощи директивы *declare*; для функций определяется их сигнатура;

– функции реализуются на языке Jaql в отдельных секциях программы, либо на языке Java во внешних файлах;

– для каждой сущности (отношения) в голове правила создается отдельный оператор *insert,* порождающий кортежи этих отношений:

1) в секции *from* объявляются исходные сущности, каждому предикату с именованными переменными или предикату членства в теле правила соответствует объявление с точностью до имени переменной;

2) в секции *select* указываются атрибуты целевых отношений (в соответствии с их названиями в предикатах головы правила) и выражения порождения их значений; выражения формируются в соответствии с термами в предикатах головы правила либо с термами во фреймовых предикатах тела правила;

3) в секции *where* указываются предикаты, отражающие способы соединения исходных сущностей (формируемые на основании совпадения имен переменных во фреймовых предикатах тела) и их отбора (соответствующие предикатам, налагающим условия на данные отдельных исходных коллекций в теле правила);

– если соединение сущностей исходных коллекций в теле правила осуществляется с использованием нетривиальных предикатов и функций (отличных от равенства атрибутов), предварительное соединение сущностей производится с использованием оператора разрешения сущностей *create link*:

1) в секции *from* указываются соединяемые коллекции;

2) в секции *select* указываются составные ключи, однозначно идентифицирующие исходные сущности;

3) в секции *match* указываются правила сопоставления сущностей;

4) коллекция пар сопоставленных сущностей затем используется в качестве входной в операторах *insert,* порождающих кортежи целевых отношений.

Описанные выше принципы реализации могут быть формализованы в виде следующих *образцов реализации*.

Таблица 4.6 - *Образец 1. Простое связывание значения атрибута.*

RIF-BLD	HIL
Forall ?sst ?v1 ?v2 ?v3 (AND(TargetRel1(a1->?v1 a3->?v3) TargetRel2(a2->External(f(?v2)))) :- And(SourceRel(b1->?v1) ?sst#SourceStruct ?sst[b2->?v2 b3->?v3]))	declare TargetRel1: ? declare TargetRel2: ? declare SourceRel: ? declare SourceStruct: ? declare f: function Type1 to Type2; insert into TargetRel1 select [a1: sr.b1, a3: sst.b3] from SourceRel sr, SourceStruct sst; insert into TargetRel2 select [a3: f(sr2.b2)] from SourceStruct sst;

В *образце 1* (таблица 4.6) применяются несколько общих принципов: исходные сущности (отношение *SourceRel* и структура *SourceStruct*), целевые сущности (отношения *TargetRel1* и *TargetRel2*) и функция *f* объявлены при помощи директивы

declare; операции *insert* определены для обоих целевых отношений; исходные сущности ассоциированы с переменными в секциях *from* операций *insert*. Значения исходных и целевых атрибутов, связанные переменными в формуле RIF-BLD, представлены выражениями в секциях *select*. Например, значения *SourceRel.b1* и *TargetRel1.a1,* связанные переменной *?v1* представлены выражением *a1: sr.b1*. Для связывания значений атрибутов могут использоваться внешние функции (например, *f*).

Таблица 4.7 - *Образец 2. Развертывание атрибута типа множества.*

RIF-BLD	HIL
Forall ?sst ?v1 ?v2 ?v3 (TargetRel(a->?v3) :- And(?sst#SourceStruct ?sst[b1->?v1] ?v2#?v1 ?v2[b2->?v3]))	insert into TargetRel select [a: v2.b2] from SourceStruct sst, sst.b1 v2;

Образец 2 (таблица 4.7): если исходная структура (например, *SourceStruct*) включает атрибут типа множества (например, *b1*) и внутренние подструктуры значений этого атрибута связаны с целевыми атрибутами (например, атрибут *b2* связан с атрибутом *TargetRel.a* переменной *?v3*), то множество значений атрибута объявляется отдельной переменной в секции *from* операции *insert* (например, объявление переменной *v2: sst.b1 v2*).

Таблица 4.8 – *Образец 3. Выражение пути в структуре.*

RIF-BLD	HIL
Forall ?sst ?v1 ?v2 ?v3 (TargetRel(a->?v3) :- And(?sst#SourceStruct1 ?sst[b1->?v1] ?v1#SourceStruct2 ?v1[b2->?v2] ?v2#SourceStruct3 ?v2[b3->?v3]))	insert into TargetRel select [a: sst.b1.b2.b3] from SourceStruct sst;

Образец 3 (таблица 4.8): исходные структуры могут содержать произвольное число вложенных структурных уровней. В RIF-BLD к этим уровням можно обращаться с использованием предикатов принадлежности множеству (например, *?v1#SourceStruct2*) и фреймовых предикатов (например, *?v1[b2->?v2]*). В языке HIL эти группы предикатов представляются выражениями путей в секциях *select* операций (например, путь *sst.b1.b2.b3*).

Таблица 4.9 – *Образец 4. Экзистенциальная переменная в голове правила.*

RIF-BLD	HIL
Forall ?v1 v2 (Exists ?v3 (TargetRel(a1->?v1 a2->?v3)) :- And(SourceRel(b1->?v1 b2->?v2)))	insert into TargetRel select [a1: sr.b1, a2: get_id(sr.b2)] from SourceRel sr;

Образец 4 (таблица 4.9): данные, необходимые для инициализации значения целевого атрибута (например, *a2*) могут быть не определены в исходных сущностях явно (например, в отношении *SourceRel*). В этом случае в правиле RIF-BLD переменная, обозначающая значение атрибута (например, *v3*) должна быть связана квантором существования. В языке HIL такое связывание квантором существования должно быть в явном виде реализовано в виде присваивания некоторого конкретного значения (генерируемого, например, функцией *get_id*).

Таблица 4.10 – *Образец 5. Простое соединение и селекция исходных сущностей.*

RIF-BLD	HIL
Forall ?sst ?vb1 ?vb2 ?vc2 (TargetRel(a1->?vb2 a2->?vc2) :- And(SourceRel(b1->?vb1 b2->?vb2) ?sst#SourceStruct ?sst[c1->?vb1 c2->?vc2] External(pred(?vc2))))	insert into TargetRel select [a1: sr.b2, a2: sst.b3] from SourceRel sr, SourceStruct sst where sr.b1 = sst.c1 and pred(sst.c2);

Образец 5 (таблица 4.10): предикаты селекции значений атрибутов исходных коллекций (например, *pred(?vc2)*) представляются в языке HIL соответствующими предикатами в секции *where.* Простые условия соединения исходных коллекций (например, атрибуты *SourceRel.b1* и *SourceStruct.c1* должны иметь одинаковое значение, обозначаемое переменной *?vb1*) также представляются в HIL предикатами в секции *where* (например, *sr.b1 = sst.c1*).

Образец 6 (таблица 4.11): исходные сущности могут сопоставляться в правиле RIF-BLD с использованием сложных условий. Например, сущности из коллекции *SourceStruct* и отношения *SourceRel* сопоставляются с использованием равенства атрибутов *SourceStruct.c1* и *SourceRel.c2,* а также с использованием дизъюнкции двух предикатов: *pred1(?vb1 ?vb2)* и *pred2(?vb1 ?vb2 ?vc).*

Таблица 4.11 – *Образец 6. Нетривиальное сопоставление исходных сущностей.*

RIF-BLD	HIL

Forall ?sst ?vb1 ?vb2 ?vc (
TargetRel(a1->?vb1 a2->?vc)
:-
And(
 ?sst#SourceStruct
 ?sst[b1->?vb1 c1->?vc]
 SourceRel(b2->?vb2 c2->?vc)
 Or(External(pred1(?vb1 ?vb2))
  External(pred2(?vb1 ?vb2 ?vc)))
))
	create link
	 SourceStruct_SourceRel_link as
	select[
	 b1_c1: [b1: sst.b1, c1: sst.c1]
	 b2_c2: [b2: sr.b2, c2: sr.c2]
	]
	from SourceStruct sst, SourceRel sr
	match using
	 rule1: sst.c1 = sr.c2 and
	  pred1(sst.b1, sr.b2),
	 rule2: sst.c1 = sr.c2 and
	  pred2(sst.b1, sr.b2, sst.c1)
	
	insert into TargetRel
	select [a1: sst.b1, a2: sst.b2]
	from SourceStruct sst,
	  SourceStruct_SourceRel_link lnk
	where sst.b1 = lnk.b1_c1.b1 and
	  sst.c1 = lnk.b1_c1.c1

Такое сопоставление реализуется при помощи операции *create link*. Атрибуты *SourceStruct.b1* и *SourceStruct.c1* здесь образуют составной ключ для коллекции *SourceStruct*; атрибуты *SourceRel.b2* и *SourceRel.c2* образуют составной ключ для отношения *SourceRel.* Предикаты сопоставления реализуются парой правил в секции *match* операции *create link*. Коллекция сопоставленных сущностей *SourceStruct_SourceRel_link* используется как исходная для операции *insert*, конструирующей кортежи отношения *TargetRel*.

4.3.2 Спецификация и реализация правил интеграции данных XML и реляционной модели с разрешением конфликтов

В данном разделе рассматриваются два примера правил интеграции данных, оперирующих сущностями XML и реляционной модели.

В первом примере рассматривается правило преобразования данных из XML к реляционной схеме хранилища (целевой схеме) с разрешением структурных конфликтов, конфликтов имен и значений.

Во втором примере рассматривается правило преобразования данных из соединения двух коллекций, одна из которых представлена в XML, другая – в реляционной модели.

4.3.2.1 Интеграция данных о маршрутах объектов

В левом столбце таблицы 4.12 приведен пример данных в формате XML о маршруте объекта, полученных из системы мониторинга, учета и классификации судов КИИС «МоРе» [301]. Маршрут (элемент *ISSKOI_Track*) состоит из маршрутных точек

(*ISSKOI_TrackPoint*). В каждой точке заданы значения координат (*pos*), времени (*Time*) и высоты (*BarAltitude*). Данные о других маршрутах имеют ту же структуру элементов и отличатся значениями элементов и атрибутов.

Таблица 4.12 - Данные о маршруте объекта и соответствующие элементы целевой схемы

Пример данных в исходной модели (XML)	Элементы целевой схемы (реляционная модель)
<ISSKOI_Track> <Id>56473</Id> <TrackName>copter-1</TrackName> <ISSKOI_TrackPoints> <ISSKOI_TrackPoint id="uuid-2b7ca14"> <Position> <Point id="uuid-859bef91"> <pos>33.8957 246.37</pos> </gml:Point> </Position> <Time>2016-12-12 T13:33:11</Time> <BarAltitude>533.89 </BarAltitude> <HSpeed>108.1</HSpeed> <VSpeed>2</VSpeed> </TrackPoint> </TrackPoints> </Track>	Track(PK trackId, name) TrackPoint(PK pointId, FK path, time, height, latitude, longitude)

В правом столбце таблицы приведены элементы целевой схемы, соответствующие исходным данным. Так, элемент *ISSKOI_Track* соответствует отношению целевой схемы *Track*, вложенный элемент *Id* соответствует первичному ключу (PK) *Track.trackId,* вложенный элемент *TrackName* – атрибуту *Track.name.* Элемент *ISSKOI_TrackPoint* соответствует отношению целевой схемы *TrackPoint,* атрибут элемента *id* соответствует первичному ключу *TrackPoint.pointId,* вложенные элементы *Time* и *BarAltitude* – атрибутам *TrackPoint.time* и *TrackPoint.height* соответственно, вложенный элемент *pos* – атрибутам *TrackPoint.logtitude* и *TrackPoint.latitude.*

В рассмотренном примере, как и в других примерах, приведенных ниже, приведена лишь часть элементов, составляющих исходные данные и целевую схему.

Очевидно, что при спецификации правила интеграции данных о маршрутах необходимо разрешить конфликты имен (например, элемент *BarAltitude* и атрибут *height* имеют одинаковую семантику, но разные имена), структурные конфликты и конфликты значений (элемент *pos*, содержащий широту и долготу, соответствует двум отдельным атрибутам *logtitude*, *latitude*).

С использованием диалекта RIF-BLD необходимое правило интеграции описывается следующим образом:

```
Forall ?Track ?Id ?TrackName ?TrackPoints ?TrackPoint ?Position ?Time
?Height ?Point ?pos
( AND( Track(trackId->?Id  name->?TrackName)
  TrackPoint(pointId->?pid  path->?Id  time->?Time  height->?Height
   latitude->External(get_latitude(?pos))
   longitude->External(get_longitude(?pos)) ))
:-
AND(?Track#ISSKOI_Track
    ?Track[Id->?Id  TrackName->?TrackName  TrackPoints->?TrackPoint]
  ?TPoint#ISSKOI_TrackPoint
  ?TPoint[id->?pid  Position->?Position  Time->?Time  BarAltitude->?Height]
  ?Position#Position  ?Position[Point->?Point]
  ?Point#Point  ?Point[pos->?pos]))
```

Forall в правиле обозначает квантор всеобщности, знак «:-» обозначает импликацию – логическое следование формулы-головы правила из формулы-тела правила, операция *AND* обозначает конъюнкцию. Идентификаторы вида *?X* обозначают переменные.

В правиле используется три вида предикатов. В голове правила используются *предикаты с именованными аргументами* [141] *Track* и *TrackPoint,* соответствующие отношениям целевой схемы.

В теле правила используются *предикаты членства* [141] (например, предикат *?Track#ISSKOI_Track,* обращающийся в истину, когда переменная *?Track* принимает значение произвольного элемента *ISSKOI_Track*) и *фреймовые предикаты* [141], отражающие структуру XML-элементов исходных данных. Так, предикат *?Point[pos->?pos]* обращается в истину на таких парах значений переменных *?Point* и *?pos,* что элемент – значение переменной *?Point* имеет вложенный элемент *pos* и его значение есть *?pos.* Конъюнкция предикатов в теле правила полностью задает структуру вложенных элементов и атрибутов произвольного элемента *ISSKOI_Track.*

Связь значений атрибутов отношений в голове правила и значений элементов и атрибутов в теле правила задается при помощи переменных, таким образом разрешаются конфликты имен и структурные конфликты.

Конфликты значений могут быть разрешены при помощи функций (например, *get_latitude*), представляемых в RIF-FLD как *внешние термы* (*External*). Их семантика в рамках RIF-FLD напрямую не определяется и уточняется при реализации (например, семантика функции *get_latitude* состоит в том, чтобы вернуть первое число, входящее в исходную строку).

Рассмотренное правило имеет естественную логическую семантику: для всех наборов значений подкванторных переменных, обращающих в истину все предикаты тела правила на исходной коллекции, отношения хранилища должны содержать кортежи, соответствующие предикатам головы правила с тем же набором значений

переменных. Для рассмотренного примера хранилище должно содержать кортежи *Track(trackId: 56473, name: “copter-1”), TrackPoint(pointId: “uuid-2b7ca14”, path: 56473, time: “2016-12-12T13:33:11”, height: 533.89, latitude: 33.8957, longitude: 246.37)*.

Вышеописанное правило может быть реализовано в языке HIL следующей программой:

```
declare ISSKOI_Track: ?;
declare Track: ?;
declare TrackPoint: ?;
declare get_latitude: function string to double;
declare get_longitude: function string to double;

@jaql{
get_latitude = fn($s) convert(substring(
  $s, 0, strPos($s, ' ')-1), schema double);
get_longitude = fn($s) convert(substring(
  $s, strPos($s, ' ')+1, strLen($s)), schema double);
}

insert into Track
select [ trackId: t."Id",name: t."TrackName"]
from ISSKOI_Track t;

insert into TrackPoint
select [ path: t."Id", time: p."Time", height: tpt."BarAltitude",
  hSpeed: tpt."HSpeed", vSpeed: tpt."VSpeed", course: tpt."Course",
  latitute: get_latitude(pt."pos"), longitude: get_longitude(pt."pos") ]
from ISSKOI_Track t, t."TrackPoints" tpt, tpt."Position" ps, ps."Point" pt;
```

В программе объявляются (*declare*) исходная сущность (внешний элемент *ISSKOI_Track*) и целевые сущности, соответствующие отношениям (*Track, TrackPoint*). Знак «?» в объявлении означает, что структура сущностей не задана при определении, а выводится из программы.

Объявляются функции разрешения конфликтов (*get_latitude, get_longitude*) и их сигнатуры. Реализация функций производится с использованием языка Jaql [294] (директива *@jaql*) и его функций работы со строками *substring, strPos*.

Для целевых отношений *Track, TrackPoint* определяются операторы *insert*, порождающие кортежи этих отношений. В секции *from* операторов *insert* объявляются исходные сущности, каждому предикату членства в теле правила RIF-BLD (например, *?Track#ISSKOI_Track*) соответствует объявление с точностью до имени переменной (например, *ISSKOI_Track t*).

Атрибуты целевых отношений и выражения порождения их значений определяются в секции *select* в соответствии с предикатами в голове правила и фреймовыми предикатами в теле правила. Например, предикату головы *Track(trackId->?Id)* и фреймовому предикату тела *?Track[Id->?Id]* соответствует определение *trackId: t."Id"* в секции *select* оператора *insert into Track*.

Заметим, что язык HIL оперирует данными в формате JSON [13], поэтому для применения рассмотренного правила на языке HIL для преобразования данных о маршрутах в целевую схему необходимо преобразовать исходные XML-документы в JSON при помощи встроенной функции *xmlToJson* языка Jaql. Полученные на выходе HIL-программы JSON-документы затем загружаются в реляционное хранилище над Hadoop (например, *Hive* [302]).

4.3.2.2 Интеграция данных о судах

В левом столбце таблицы 4.13 приведен пример данных о судах в формате XML, полученный из системы «Поиск-Море» (*ERRTableShips*) [303]; а также пример данных о судах в реляционной модели, полученный из системы ЕСИМО [304] (данные получены в формате CSV и преобразованы в JSON). В обеих исходных коллекциях могут быть найдены данные об одних и тех же судах (судно идентифицируется по названию и позывному). Кроме того, коллекции содержат различные взаимодополняющие данные о судах.

Таблица 4.13 - Данные о судах и соответствующие элементы целевой схемы

Пример данных в исходных моделях (XML, реляционная)	Элементы целевой схемы (реляционная модель)
<ERRTableShips> <Id>64694571</Id> <Name>мвс Ростов Великий</Name> <Callsing>UBZG5</Callsing> <Dates> <StartDate>2017-01-08</StartDate> <EndDate>2017-01-09</EndDate> <Dates> </ERRTableShips> [{ "Platforma_nazvanie": "мвс Ростов Велкий", "Strana_naimenovanie": "Россия", "Organizaciya_nazvanie": "ФГУП Балтийское БАСУ – Сахалинский филиал", "Pozyvnoj": "UBZG5", "nomer_IMO": "9586796" }]	SARUnit(beginDuty, endDuty, vehicle) Vessel(PK vehicleId, name, call, country, FK owner, imoNumber) LegalEntity(PK entityId, name)

В правом столбце таблицы приведены элементы целевой схемы, соответствующие исходным данным. Данные, соответствующие судну, как транспортному средству, сосредоточены в отношении *Vessel*; как спасательной единице – в отношении *SARUnit*; как юридической сущности - в отношении *LegalEntity*.

С использованием диалекта RIF-BLD необходимое правило интеграции описывается следующим образом:

```
Forall ?Id ?name ?call ?beginDuty ?endDuty ?country ?ownerName ?imoNumber
( Exists ?owner (
  AND( SARUnit(beginDuty->?beginDuty  endDuty->?endDuty  vehicle->?Id )
   Vessel(vehicleId->?Id  name->External(normalize(?nazv))
     call->?call  Country->?country  owner->?owner  imoNumber->?imoNumber))
   LegalEntity(entityId->?owner  name->?ownerName) ))
:-
AND( ?ERRTableShips#ERRTableShips
  ?ERRTableShips[Id->?Id  Name->?name Callsing->?call  Dates->?Dates]
  ?Dates#Dates  ?Dates[StartDate->?beginDuty  EndDate->?endDuty]
  Ship("Platforma_nazvanie"->?nazv "Pozyvnoj"->?call
   "Strana_naimenovanie"->?country "Organizaciya_nazvanie"->?ownerName
   "nomer_IMO"->?imoNumber)
  External(compareShipName(?name, ?nazv))  ))
```

Аналогично примеру, описанному в предыдущем подразделе, в голове правила конъюнкцией соединяются предикаты, соответствующие отношениям целевой схемы. Кроме того, конъюнкция в голове правила заключена под квантор существования (*Exists*) по переменной *?owner*[68], значение которой (не определенное в исходных данных) является первичным ключом *entityId* в кортеже отношения *LegalEntity* и внешним ключом в кортеже отношения *Vessel*.

В теле правила конъюнкцией соединяется предикат *Ship,* соответствующий отношению из системы ЕСИМО, предикаты членства и фреймовые предикаты, соответствующие структуре XML-документов из системы «Поиск-Море».

Отдельной особенностью правила является то, что при соединении сущностей из исходных коллекций происходит проверка на соответствие имен судов с некоторой точностью (возможны варианты записей и ошибки) при помощи функции *compareShipName*.

Вышеописанное правило может быть реализовано в языке HIL следующей программой (объявления сущностей и функций опущены):

```
create link ShipLink as
select [
 Callsing_Name: [Callsing: es.Callsing, Name: es.Name],
 "Pozyvnoj_Platforma_nazvanie": ["Pozyvnoj": s."Pozyvnoj",
 "Platforma_nazvanie": s."Platforma_nazvanie"]]
from ERRTableShips es, Ship s
match using
 rule1: es.Callsing = s."Pozyvnoj" and
   compareShipName(es.Name, s."Platforma_nazvanie");
```

[68] Строго говоря, квантор всеобщности в голове правила выходит за границы диалекта RIF-BLD, однако входит в каркас RIF-FLD.

```
insert into SARUnit
select [ vehicle: s.Id, beginDuty: d.StartDate, endDuty: d.EndDate ]
from ShipLink sl, ERRTableShips s, s.Dates d
where sl.Callsing_Name.Name = s.Name and
  sl.Callsing_Name.Callsing = s.Callsing;

insert into Vessel
select [ vehicleId: s.Id, name: normalize(s."Platforma_nazvanie"),
 call: s."Pozyvnoj", country: s."Strana_naimenovanie",
 owner: get_id(s."Organizaciya_nazvanie"), imoNumber: s."nomer_IMO"]
from ShipLink sl, Ship s
where sl.Callsing_Name.Callsing = s."Pozyvnoj"
  and  sl.Callsing_Name.Name = s."Platforma_nazvanie";

insert into LegalEntity
select [
 entityId: get_id(s."Organizaciya_nazvanie"),
 name: s."Organizaciya_nazvanie"]
from ShipLink sl, Ship s
where sl.Callsing_Name.Name = s."Platforma_nazvanie"  and
  sl.Callsing_Name.Callsing = s."Pozyvnoj";
```

Соединение сущностей исходных коллекций производится с использованием оператора разрешения сущностей *create link*. В секции *from* оператора указываются соединяемые коллекции (*ERRTableShips, Ship*), в секции *select* – составные ключи (*Callsing_Name, Pozyvnoj_Platforma_nazvanie*), однозначно идентифицирующие исходные сущности, правило сопоставления сущностей *rule1* (совпадение позывных и соответствие имен с точностью до функции *compareShipName*).

Как и в примере, описанном в предыдущем подразделе, для каждого отношения целевой схемы в программе определен оператор *insert*. Особенность данной программы состоит в том, что целевые отношения пополняются на основании только тех сущностей, которые связаны оператором *create link*.

Квантор всеобщности реализуется с использованием функции *get_id*, порождающей уникальный идентификатор, и, тем самым, означивающей подкванторную переменную *?owner*. Значение порожденного идентификатора присваивается первичному ключу *LegalEntity.entityId* и внешнему ключу *Vessel.owner*.

4.3.3 Спецификация и реализация правил интеграции данных документной модели

В данном разделе рассматривается пример правила интеграции данных, оперирующего сущностями документной модели. Исходная коллекция содержит сообщения о происшествиях в Арктической зоне из социальных сетей и сущности (персоны, суда, географические локации и т.д.), извлеченные средствами анализа текстов из сообщений [305]. Данные хранятся в документной СУБД MongoDB и экспортируются для дальнейшей интеграции в файлах в формате JSON.

В левом столбце таблицы 4.13 приведен пример данных о сообщении (*Messages*) и данных о сущностях, извлеченных из сообщения (*Entites*). Связь сообщений и сущностей, извлеченных из них, устанавливается на основании значения составного ключа *id*. В правом столбце таблицы приведены элементы целевой схемы, соответствующие исходным данным.

С использованием диалекта RIF-BLD правило интеграции данных о сообщениях описывается следующим образом:

```
Forall ?m ?ext ?doc ?coll ?src ?cont
  ?time ?lang ?auth ?mid ?mres ?mf ?eid ?eres
( Exists ?ent( AND(
 Document(documentId->?doc  collection->?coll source->?src  content->?cont
  time->?time  language->?lang  author->?auth)
 ExtractedEntity(entityId->?ent document->?doc
  token->?tok  tag->?tag  begin->?beg  end->?end) ))
:-
AND( ?m#Messages  ?ext#Entities
 ?m[id->?mid annotation->?cont metafields->?mf]
 ?mid[coll_id->?coll  res_id->?mres]
 ?mres[site_id->?src  doc_id->?doc]
 ?mf[mf203->?time  mf205->?lang  mf200->?auth]
 ?ext[id->?eid  entities->?ents]
 ?eid[coll_id->?coll  res_id->?eres]
 ?eres[site_id->?src  doc_id->?doc]
 ?ext#?ents
 ?ext[s_token->?tok  s_tag->?tag  s_begin->?beg  s_end->?end] ))
```

Таблица 4.13 - Данные о сообщениях и соответствующие элементы целевой схемы

Пример данных в исходной модели (документная)	Элементы целевой схемы (реляционная модель)

{"Messages": [{
"id": {"coll_id": "8002",
"res_id": {
"site_id": "9b290c9f3bda",
"doc_id": "3649a5559a62"}},
"annotation": "Chinese seismic vessel aimed for Russian
#Barents Sea oil at logistics port #Kirkenes",
"metafields": {
"mf203": "2016-4-30",
"mf205": "eng",
"mf200": "iceblogger"}
}]}
 	
{"Entities": [{
"id": {"coll_id": "8002",
"res_id": {
"site_id": "9b290c9f3bda",
"doc_id": "3649a5559a62"}},
"entities": [
{ "s_token": "Barents Sea",
"s_tag": "I-ALOC",
"s_end": 53,
"s_begin": 42},
{ "s_token": "Kirkenes",
"s_tag": "I-ALOC",
"s_end": 85,
"s_begin": 77}
]}] }
	Document(
	documentId,
	collection,
	source,
	content,
	time,
	language,
	author)
	
	ExtractedEntity(
	entityId,
	document,
	token,
	tag,
	begin
	end)

Аналогично примерам, приведенным в предыдущем разделе, в голове правила конъюнкцией соединяются предикаты, соответствующие отношениям целевой схемы. В теле правила при помощи предикатов членства и фреймовых предикатов отражена структура документов, соответствующих сообщениям и извлеченным сущностям. Правило может быть реализовано в языке HIL следующей программой:

```
insert into Document
select [ documentId: mres."doc_id",  collection: mid."coll_id",
  source: mres."site_id",  content: m."annotation",  time: mf."mf203",
  language: mf."mf205",  author: mf."mf200"]
from Message m, m."id" mid, mid."res_id" mres, m."metafields" mf;

insert into ExtractedEntity
select [ entityId: get_id(strcat(eres."site_id", eres."doc_id")),
  document: eres."doc_id",  token: e."s_token",  tag: e."s_tag",
  begin: e."s_begin",  end: e."s_end"]
from Entities ext, ext."id" eid,  eid."res_id" eres, ext.entities e;
```

Для обоих отношений целевой схемы в программе определен оператор *insert*.

4.3.4 Спецификация и реализация правил интеграции данных графовой модели

В данном разделе рассматривается пример правила интеграции данных, оперирующего сущностями графовой модели. Исходная коллекция содержит данные о спасательных операциях, данные хранятся в графовой СУБД Neo4j и экспортируются для дальнейшей интеграции в файлах в формате JSON.

В левом столбце таблицы 4.14 приведен пример данных о спасательных операциях. Данные содержат три вида вершин (*Nodes*): спасательная операция (*SarOperation*), координационный центр (*CoordCenter*), координатор операции (*Coordinator*); и два вида ребер (*Relationships*): *rCoordCenter* (ребра, связывающие операцию и центр), *rCoordinator* (ребра, связывающие операцию и ее координатора). В правом столбце таблицы приведены элементы целевой схемы, соответствующие исходным данным.

С использованием диалекта RIF-BLD интеграция данных о спасательных операциях описывается совокупностью следующих правил:

```
Forall ?pid ?name ?lbl ?prp(
Person(personId->?pid  name->?name) :-
AND( ?n#Nodes  ?n[labels->?lbl  properties->?prp]
  "Coordinator"#?lbl  ?prp[CoordinatorID->pid  Name->?name]))

Forall ?cid ?name ?lbl ?prp(
CoordinationCenter(centerId->?cid  name->?name) :-
AND( ?n#Nodes  ?n[labels->?lbl  properties->?prp]
  "CoordCenter"#?lbl  ?prp[CoordCenterID->cid  Name->?name]))
```

```
Forall (
SAROperation(operationId->?opid  creationTime->?time  name->?name
  country->?country  coordCenter->?cntr  coordinator->?coord)
:-
AND( ?n#Nodes  ?n[id->?id]  ?n[properties->?prp  labels->?lbl]
  ?r1#Relatioships  ?r2#Relatioships
  ?n1#Nodes  ?n1[id->?id1]  ?n2#Nodes  ?n2[id->?id2]
  "SarOperation"#lbl
  ?r1[type->"rCoordCenter"  startNode->?id  endNode->?id1]
  ?r2[type->"rCoordinator"  startNode->?id  endNode->?id2]
))
```

Таблица 4.14 - Данные о спасательных операциях и соответствующие элементы целевой схемы

Пример данных в исходной модели (графовая)	Элементы целевой схемы (реляционная модель)
{"Nodes": [{"id":"12", "labels":["SarOperation"], "properties":{ "OperationID":"5824", "OperationDate": "2016-09-15T00:00:00", "OperationName": "Arctic Sunrise Sinking", "countrycode":"643"} }, {"id":"13", "labels":["CoordCenter"], "properties":{ "CoordCenterID":"mrcc667340", "Name":"MRCC Dikson"} }, {"id":"14", "labels":["Coordinator"], "properties":{ "Name": "Schurov V. A.", "CoordinatorID": "5773"} }]} { "Relationships": [{"id":"4", "type":"rCoordCenter", "startNode":"12", "endNode":"13"}, {"id":"9", "type":"rCoordinator", "startNode":"12", "endNode":"14"}] }	SAROperation(PK operationId, FK coordCenter, FK coordinator, country, creationTime, name) CoordinationCenter(PK centerId, name) Person(PK personId, name)

Каждому отношению целевой схемы соответствует свой тип вершин, например, отношению *SAROperation* соответствуют вершины с меткой *SarOperation* (атрибут *labels*). Поэтому для каждого отношения удобно определить свое порождающее правило. Предикаты, соответствующие отношениям целевой схемы, составляют головы правил. В телах правил при помощи предикатов членства и фреймовых предикатов отражены структуры данных, соответствующие вершинам и ребрам графа.

Правила могут быть реализованы в языке HIL следующей программой:

```
insert into Person
select [ personId: prp."CoordinatorID", name: prp."Name"]
from Nodes n, n."properties" prp,  n."labels" lbl
where  array_includes(lbl, "Coordinator");
```

```
insert into CoordinationCenter
select [ centerId: prp."CoordCenterID",  name: prp."Name"]
from Nodes n, n."properties" prp n."labels" lbl
where  array_includes(lbl, "CoordCenter");

insert into SAROperation
select [ operationId: prp."OperationID",
  creationTime: prp."OperationDate",  name: prp."OperationName",
  country: prp."countrycode",
  coordCenter: prp1."CoordCenterID",  coordinator: prp2."CoordinatorID"]
from Nodes n, n."properties" prp,  n."labels" lbl,
  Relationships r1, Relationships r2, Nodes n1, Nodes n2
where
  array_includes(lbl, "SarOperation") and
  n."id" = r1."startNode" and  r1."type" = "rCoordCenter" and
  n1."id" = r1."endNode" and  n."id" = r2."startNode" and
  r2."type" = "rCoordinator" and  n2."id" = r2."endNode";
```

Каждое из правил представляется отдельным оператором *insert*. Предикаты проверки типа вершины (например, *"Coordinator"#?lbl*) реализуются с использованием функции *array_includes:*

```
@jaql{ array_includes = fn(a, e)
  if (not exists(a->filter $ == e)) false
  else true; }
```

4.4 Родственные работы

К *определению семантики языков программирования* (в том числе, императивных) обычно выделяют три подхода. *Операционная семантика* описывает исполнение конструкций языка программирования на некоторой абстрактной вычислительной машине. *Денотационная семантика* предполагает отображение синтаксических конструкций языка в математические объекты (чаще всего в терминах теории множеств и логики предикатов первого порядка). *Аксиоматическая семантика* выражает связи между предусловиями и постусловиями на переменные программы для конструкций языка при помощи индуктивного набора правил [306]. Аксиоматическая семантика наиболее абстрактна, она декларирует эффекты конструкций языка программирования. Денотационная семантика в явном виде придает значения конструкциям языка. При этом конструируется формальное доказательство того, что денотационная семантика языка удовлетворяет аксиоматическим свойствам. Далее, денотационная семантика может быть реализована при помощи операционных определений.

В классической работе Winskel [307] все три вида семантики описаны для модельного императивного языка IMP. Подробно различные аспекты денотационной семантики описаны в работе Schmidt [308], и денотационная семантика позиционируется

как подход, позволяющий наиболее полно и точно выразить значение конструкций языка программирования. Подвидом денотационной семантики можно назвать алгебраическую семантику, в которой состояние программы представляется в виде многосортной алгебры [309] [310]. Существуют работы, направленные на модификацию операционной семантики, а также на ее определение для определенных подклассов императивных языков или специализированных языков. Так, например, в работе [311] определена основанная на правилах операционная семантика минимального императивного языка (похожего на упомянутый выше IMP), приближающаяся по уровню абстракции к другим видам семантики. В работе [312] определена операционная семантика для модельного обратимого императивного языка RIMP. В работе [313] определена исполняемая семантика языка Structured Text (ST) для программируемых логических контроллеров (PLC). В работе [314] операционная и аксиоматическая (Логика Хоара) семантики для императивного языка Simpl определены в виде теории для средств автоматизированной верификации Isabelle/HOL.

Существуют примеры и более практически-ориентированных работ, направленных на определение семантики широко используемых язяков программирования. Так, в работе [315] определена операционная семантика языка Python 3.3 с использованием основанного на переписывании термов семантического каркаса K [316]. В работе [317] определена структурная операционная семантика SOS [318] для достаточно широкого подмножества языка Python на основе собственного метатеоретического каркаса, формализующего семантику SOS.

В работе [319] определена семантика базового класса SQL-запросов вида SELECT-FROM-WHERE с подзапросами. Определение семантики проведено с использованием теории множеств, семантика запроса – это функция, получающая на вход экземпляр базы данных и порождающая результат запроса – таблицу. Практической целью работы ставится доказательство эквивалентности рассматриваемого фрагмента SQL и реляционной алгебры, а также доказательство достаточности двузначной логики для представления неопределенных значений SQL.

В работе [320] предложена семантика фрагмента SQL, включающего *select [distinct] from where group by having* запросы, функции, агрегатные функции, ограниченные кванторы, подзапросы. Для определения семантики используется система управления формальными доказательствами Coq [321]. Также в Coq определен фрагмент реляционной алгебры и установлено отображение между представлением SQL и реляционной алгеброй, сохраняющее семантику.

В работе [322] рассматривается формальная семантика диаграмм потоков данных в языке VDM (Vienna Development Method). В работе [323] для представления семантики процессов извлечения-трансформации-загрузки (ETL) данных используется язык LDL (Logic-Based Data Language). Целью этих работ является обеспечение перехода от концептуальной модели процессов трансформации данных к логической. В системе Clio [33] реализован подход обмена данными, когда трансформация данных представлена в виде Datalog-подобной программы с логической семантикой.

Много работ известно в области *верификации трансформаций моделей* [324]. Эти работы проводятся в рамках Движимой моделями инженерии (MDE). Под моделями в MDE понимаются как модели данных (языки), так и концептуальные схемы. Трансформация моделей – это реализация их отображения, набор правил, в совокупности определяющих, каким образом сущности исходной модели должны быть преобразованы в сущности целевой модели. Стремление обобщить подходы к верификации на модели разных уровней часто приводит к существенным ограничениям, накладываемым на язык трансформации: трансформации должны быть достаточно просты и декларативны.

В одном из самых цитируемых обзоров этой области [325] подходы к верификации трансформаций моделей классифицируются по их техническим деталям (использованию тестирования, проверки моделей, доказательства теорем), а также по уровню формализации, языку трансформации и проверяемым свойствам. Использование тестов требует тщательного подхода к их генерации и обеспечения уверенности, что тесты в достаточной мере покрывают разнообразие экземпляров проверяемой модели. Подходы с использованием проверки моделей и доказательства теорем делят на прямые и непрямые. При прямых подходах верификации подлежат правила трансформации на некотором языке, а при непрямых – только результаты трансформации (когда корректность можно гарантировать лишь для ограниченного набора результатов, полученных применением трансформации).

Верификации подлежат такие свойства трансформаций, как завершаемость (процесс трансформации успешно завершается для любой правильно определенной исходной модели), определенность (уникальность целевой модели для заданной исходной модели и трансформации), синтаксическая корректность трансформации по отношению к языку трансформаций, сохранение семантики исполнения (трансформация исполняется в соответствии с ее спецификацией). Для формального доказательства свойства трансформаций представляются в формальных языках, различных вариантах логики первого порядка (например, Calculus of Inductive Constructions). В зависимости

от мощности используемого языка, для доказательства свойств могут быть использованы автоматические средства (SAT-решатели, средства проверки моделей) или автоматизированный логический вывод.

В работе [326] предложен подход к верификации трансформаций моделей с использованием языка AMN. Подход проиллюстрирован на примере простой декларативной трансформации, преобразующей данные из одной простой абстрактной объектной схемы (состоящей из двух классов) в другую. Каждое правило трансформации представляется операцией AMN. Проверяются свойства соответствия числа объектов исходного и целевого экземпляров схем, а также равенство значений атрибутов.

В работе [327] предложен обобщенный каркас для верификации трансформаций моделей. Каркас подходит для языков трансформации, подобных QVT или ATL, а также языков трансформации графов и предполагает использование различных методов и средств доказательства теорем (AMN, Z3 [328], Alloy) для верификации различных видов свойств. Предложена метамодель языков моделирования, включающая метаклассы языка, сущности, свойства сущности, структуры, ограничения. Определены правила представления семантики языков как экземпляров метамодели в логике первого порядка и теории множеств. Определена формальная семантика выражений языка OCL, использующихся для описания свойств (постусловий) трансформаций. Спецификации трансформаций представляют собой наборы декларативных правил, а реализации – описания на императивном языке программирования. Разработана реализация каркаса, в которой трансформации и их свойства представляются на языках UML и OCL, а их семантика определяется в языках Z3 и AMN.

В работе [329] предложен метод извлечения предопределенных видов инвариантов на языке OCL из декларативных трансформаций моделей для дальнейшего сведения к задаче удовлетворения ограничениям (CSP) и автоматической верификации.

В работе [330] предложен каркас представления семантики метамодели EMF, языка OCL, декларативных правил языка трансформаций ATL в промежуточный язык верификации Boogie [331] для дальнейшей верификации с использованием Z3.

Особенность метода, изложенного в данном разделе, состоит в том, что формальная денотационная семантика в языке AMN сообщается высокоуровневому языку разрешения сущностей и интеграции данных. Эта семантика позволяет применять автоматизированные средства доказательства промышленного уровня для верификации корректности потоков работ интеграции данных.

Родственное направление среди промышленных программных систем составляют системы *управления метаданными*, такие, как OpenMetadata[69] и Marquez[70]. Эти системы позволяют извлекать метаданные из различных СУБД, озер данных, брокеров сообщений (например, Apache Kafka[71]), систем визуализации данных (например, Grafana[72]), систем управления потоками работ (например, Apache Airflow[73]), систем управления моделями машинного обучения (например, MLflow[74]), поисковых систем (например, Elasticsearch[75]), систем хранения данных (например, Amazone S3[76]). Однако, системы управления метаданными сами не осуществляют интеграцию даных, а накапливают метаданные об ETL процессах, реализованных в других системах. Основная цель этих систем – отслеживание жизненного цикла данных (data lineage). В частности, реализованы средства *валидации данных* – проверки их свойств.

Так, в OpenMetadata поддерживаются механизмы контрактов для табличных данных (data contracts[77]). Свойства данных задаются в контрактах при помощи правил или предопределенных тестов качества.

В открытом каркасе отслеживания жизненного цикла данных OpenLineage[78] (эталонной реализацией которого является Marquez) для валидации данных предусмотрена интеграция с системой проверки качества данных Great Expectations[79]. Эта система реализует средства проверки статистического распределения данных, свежести данных, целостности данных (в том числе ссылочной целостности,

---

69 https://open-metadata.org/

70 https://marquezproject.ai/

71 https://kafka.apache.org/

72 https://grafana.com/

73 https://airflow.apache.org/

74 https://mlflow.org/

75 https://www.elastic.co/elasticsearch

76 https://aws.amazon.com/ru/s3/

77 https://docs.open-metadata.org/v1.13.x/how-to-guides/data-contracts

78 https://openlineage.io/

79 https://greatexpectations.io/

соответствия бизнес-правилам, непротиворечивости атрибутов из разных таблиц), пропущенных данных, соответствия данных схеме, уникальности данных, объема данных.

Валидация данных осуществляется также в системах управления основными данными (Master Data Management). Так, например, в системе АрхиГраф[80] ограничения целостности данных формулируются в виде логических правил над онтологией с использованием языка SHACL [332]. Аналогичный подход к валидации данных используется в системе PoolParty Semantic Suite, позиционирующейся как платформа семантического промежуточного (middleware) программного обеспечения.

Функции валидации данных могут быть запущены по расписанию, либо в произвольный момент времени по выбору пользователя. В любом случае, при этом требуется обращение к массивам данных, уже преобразованных программами интеграции данных. Метод же, рассматриваемый в данной главе, предполагает верификацию самих программ интеграции данных без их запуска и анализа преобразованных данных.

[80] https://arhigraph.ru/

5 Схемы функциональной структуры сред решения задач над неоднородными источниками данных с поддержкой верифицируемой интеграции данных

В данной главе описаны схемы функциональной структуры сред решения задач над неоднородными источниками данных, в которых применяются методы верифицируемой интеграции данных, описанные в предыдущих главах.

5.1 Схема функциональной структуры комбинированной виртуально-материализованной среды интеграции больших неоднородных коллекций данных

Раздел основан на работах автора [62][92][94][61].

5.1.1 Основные черты функциональной структуры комбинированной виртуально-материализованной среды интеграции

Схема функциональной структуры комбинированной виртуально-материализованной среды интеграции неоднородных коллекций данных различного вида (структурированных, слабоструктурированных и неструктурированных), рассматриваемая в данном разделе, нацелена на поддержку как виртуальной, так и материализованной интеграции коллекций данных, представленных как в традиционных, так и нетрадиционных моделях данных. Необходимость поддержки двух различных видов интеграции объясняется тем, что как виртуальный, так и материализованный подходы интеграции имеют свои достоинства и недостатки.

Виртуальная интеграция осуществляется с использованием технологии предметных посредников [26], образующих промежуточный слой между пользователем (приложением) и неоднородными источниками данных и сервисов. При этом данные из источников не материализуются в посреднике. К достоинствам виртуальной интеграции относится то, что развертывание и поддержка системы виртуальной интеграции значительно дешевле развертывания и поддержки системы материализованной интеграции (хранилища данных) исходя как из временных, так и из материальных затрат. Обычно системы виртуальной интеграции используются для интеграции источников, данные из которых трудно преобразовывать и (или) владельцами которых являются разные лица. Однако, данные в удаленных интегрируемых источниках не должны быть слишком большими, либо запросы к источникам должны обладать достаточной степенью селективности для того, чтобы объем данных, передаваемых из ресурса, не был

слишком велик. Ограничения на объем данных снимаются, если источники данных сосредоточены в одном *озере данных*.

При материализованной интеграции предполагается создание хранилища данных (warehouse), в которое загружаются коллекции данных, подлежащие интеграции. В процессе загрузки происходит преобразование данных из схемы коллекции в общую схему хранилища. При необходимости, хранилища могут масштабироваться на большие объемы данных (хотя это требует соответствующих материальных затрат). Хранилища предоставляют удобную и эффективную платформу преобразования и интеграции данных, а также решения сложных аналитических задач над данными.

Схема функциональной структуры комбинированной среды интеграции неоднородных коллекций данных (рис. 5.1) основана на существующей архитектуре предметных посредников [26].

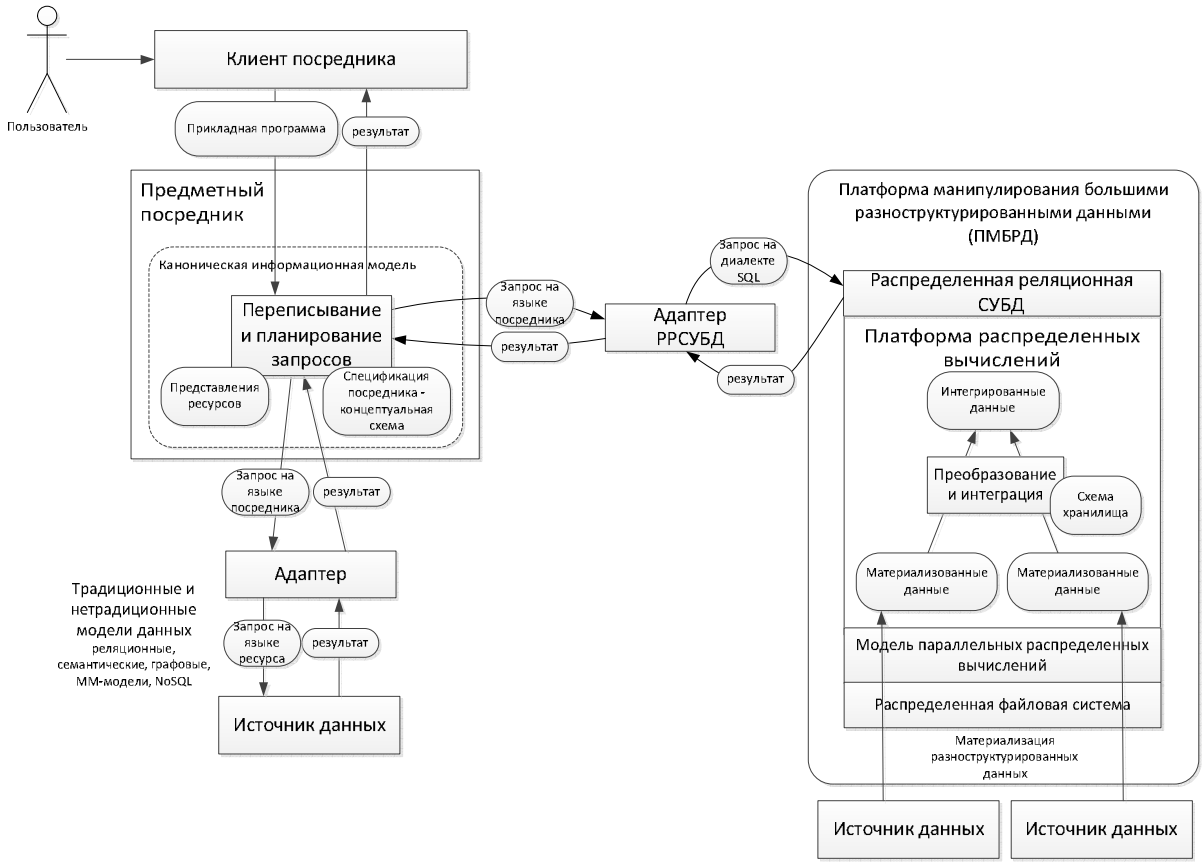

Рисунок 5.1 - Схема функциональной структуры комбинированной виртуально-материализованной среды интеграции больших неоднородных коллекций данных

Основные черты функциональной структуры выглядят следующим образом:

– концептуальная схема данных, предлагаемая пользователю системой интеграции (посредником), формируется независимо от интегрируемых источников;

– источники, релевантные концептуальной схеме, регистрируются в посреднике: их схемы связываются с концептуальной схемой при помощи представлений (взглядов). Определение взглядов осуществляется полуавтоматически и требует привлечения экспертов;

– пользователь задает программы (запросы) над концептуальной схемой. Эти запросы переписываются в посреднике с использованием взглядов в частичные подзапросы в терминах конкретных источников. Переписывание осуществляется в языке правил канонической модели посредников;

– взаимодействие источников и посредника реализуется при помощи адаптеров, располагающихся между посредником и источниками. Адаптеры преобразуют запросы из языка правил канонической модели в язык источника, а также преобразуют результат исполнения запроса из формата источника в формат посредника.

Таким образом, для виртуальной интеграции источников данных и сервисов, основанных на традиционных и нетрадиционных моделях, необходимо построение адаптеров соответствующих моделей данных. В области построения адаптеров для среды предметных посредников накоплен большой опыт, разработаны общая функциональная структура адаптеров и методы конструирования адаптеров [183], реализованы адаптеры реляционной и объектно-реляционной моделей, адаптер веб-сервисов, XML-адаптер и другие. Формальной базой для построения адаптеров служат отображения моделей данных источников в каноническую модель [58], разработанные в результате унификации моделей источников.

5.1.2 Функциональная структура платформы манипулирования большими разноструктурированными данными

Для *материализованной* интеграции источников в комбинированной среде необходима масштабируемая платформа манипулирования большими разноструктурированными данными (ПМБРД), основанная на комбинации платформы распределенных вычислений и распределенной реляционной СУБД. ПМБРД предназначена для реализации процесса извлечения и интеграции информации из данных, состоящего из следующих этапов: идентификация источников данных, извлечение данных из источников и их преобразование к единому формату, сопоставление схем источников данных и единой схемы хранилища, трансформация данных, разрешение сущностей, слияние данных.

Процесс начинается с идентификации источников данных, которые могут быть использованы при решении задач в выбранной предметной области. Эти источники могут предоставлять как структурированные данные об объектах или субъектах

предметной области[81] (например, журналы информационных систем, данные с сенсоров), так и неструктурированные данные, например, анкеты или тексты интервью экспертов, сообщения в СМИ или социальных сетях. Извлечение структурированной информации из разноструктурированных данных и ее интеграция становится одной из самых важных задач, стоящих перед разработчиками информационных систем. Это обусловлено тем, что существующие средства анализа данных, например, анализ многомерных кубов (OLAP) или интеллектуальный анализ данных (data mining), оперируют только структурированной информацией.

Особенно остро проблема извлечения информации встает при работе с неструктурированными текстовыми данными, содержащимися в публикациях электронных СМИ, новостных лентах, веб-страницах, записях в социальных сетях, коротких сообщениях микроблогов, электронных письмах, отчетах. Данные такого рода содержат полезную информацию – сущности, взаимосвязи между сущностями, факты, тональность текста (sentiments).

Получаемые от источников данные сохраняются в файловой системе в виде файлов, формат и схема которых определяется каждым конкретным источником данных. Например, файлы могут быть представлены в форматах XML, CSV, или в текстовом формате. Неструктурированная текстовая информация предварительно обрабатывается средствами анализа текстов [305].

Затем все загруженные данные преобразуются к единому формату. В качестве такового может быть выбран, например, формат JSON [13], позволяющий хранить как структурированные, так и слабоструктурированные данные.

Собранные данные интегрируются и загружаются в единое хранилище. Вопросы создания схемы единого хранилища на примере конкретной предметной области рассмотрены в работе [333]. Интеграция данных, в свою очередь, состоит из нескольких этапов.

Сопоставление схем (schema matching) является первым этапом интеграции данных. Для каждой сущности или атрибута схемы хранилища необходимо установить соответствующие им сущности или атрибуты схем источников данных. Эта задача достаточно сложна, поскольку ее решение должно опираться на семантику сущностей и атрибутов. Поэтому до сих пор часто используется спецификация соответствий

---

[81] Например, для предметной области поддержки *поисково-спасательных операций* это данные о морских и воздушных судах, данные о маршруте пропавшего объекта, последнее местоположение объекта, и т.д.

элементов схем вручную экспертом. Развиваются и методы автоматического или полуавтоматического поиска соответствий [334] [335] [336].

Трансформация данных представляет собой процесс преобразования данных, соответствующих схеме источника (исходной схеме) в информацию, соответствующую схеме хранилища (целевой схеме). Правила трансформации данных создаются на основании соответствий между элементами схем, найденных на этапе сопоставления схем. В некоторых случаях возможна автоматическая генерация правил или их шаблонов, подлежащих уточнению экспертом. Правила включают функции разрешения различных конфликтов между схемами: конфликтов имен (возникающих при использовании разных имен для семантически релевантных элементов или при использовании одинаковых имен для семантически различных элементов), конфликтов наборов атрибутов (например, при наличии вычислимых атрибутов), конфликтов типов атрибутов (например, *string* и *real*), конфликтов форматов представления значений атрибутов (например, разный формат дат), конфликтов уровней абстракции представления данных. Правила трансформации данных могут задаваться на различных языках трансформации.

К полученным данным, представленным в единой целевой схеме, применяются более тонкие методы интеграции, включающие методы *разрешения сущностей и методы слияния данных* [337]. Методы разрешения сущностей позволяют устанавливать сходство между одинаковыми сущностями их разных источников и находить дубликаты сущностей. Методы слияния сущностей позволяют задавать правила слияния данных об одной сущности из различных источников, разрешение возможных конфликтов, обнаружение и удаление ошибочных данных. Слияние данных реализуется при помощи специальных операций с использованием двух основных подходов. Первый подход основан на операции объединения (union): например, метеорологические данные сливаются нескольких источников в одну коллекцию целевой схемы. Второй подход основан на операции соединения (join): сущности из разных источников, описывающие один и тот же объект, сливаются в одну сущность целевой схемы. Полученная в результате интеграции информация загружается в единое хранилище.

Более подробно функциональная структура платформы манипулирования большими разноструктурированными данными (ПМБРД), нацеленная на реализацию процесса материализованной интеграции, изображена на рис. 5.2.

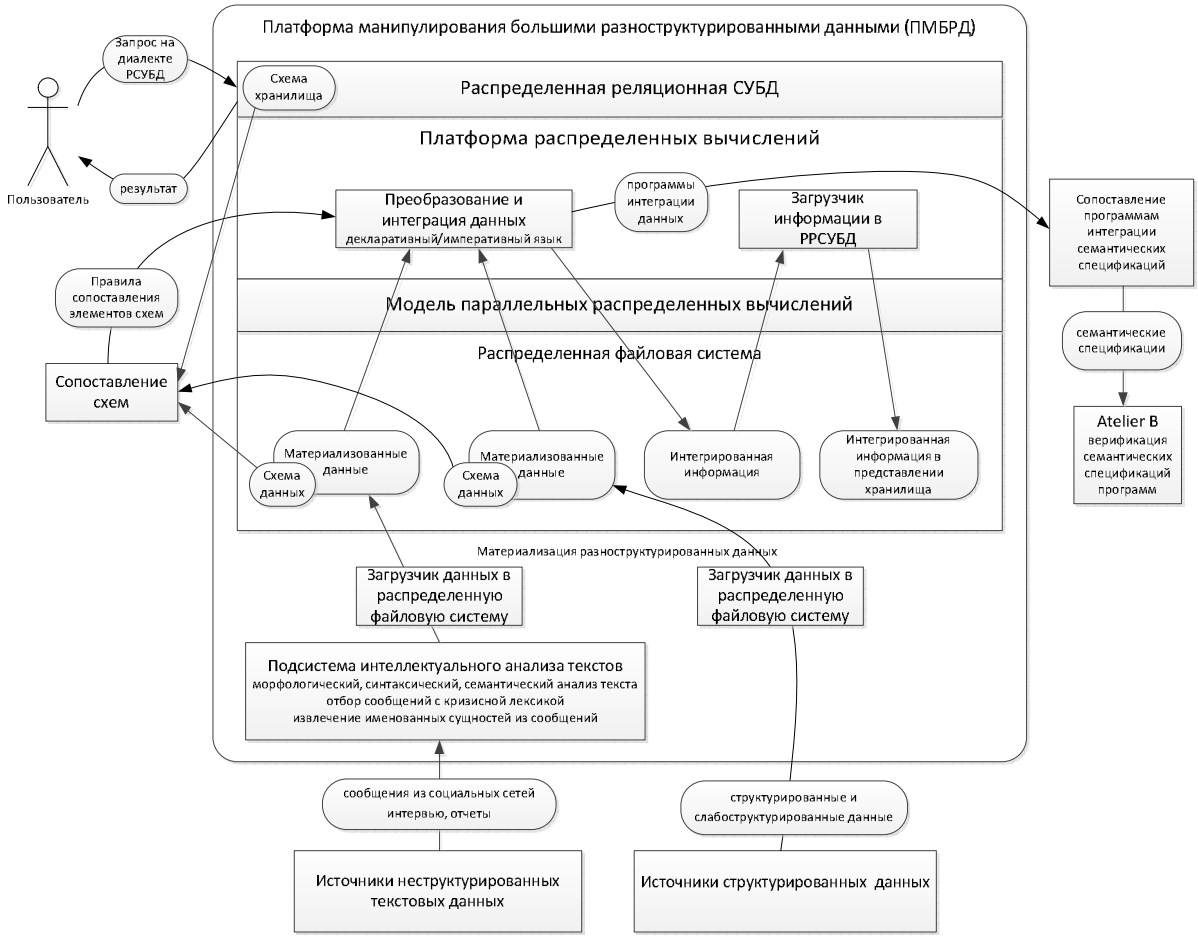

Рисунок 5.2 – Схема функциональной структуры платформы манипулирования большими разноструктурированными данными

Один из самых естественных вариантов для применения в качестве платформы распределенных вычислений – это система Apache Hadoop [30], впервые представленная в 2005 г. в составе проекта Apache Software Foundation и предназначенная для распределенного хранения и обработки больших объемов данных. Система включает, в частности, распределенную файловую систему HDFS (Hadoop Distributed File System) и реализации программных моделей параллельных распределенных вычислений (MapReduce [297], Spark [338], Tez [339]).

Платформа разворачивается на вычислительных кластерах, в состав которых входят узлы, хранящие фрагменты файлов. Теоретически могут быть сотни и тысячи таких узлов, основанных на недорогих вычислительных платформах (commodity hardware). Для обеспечения высокой надежности поддерживается избыточность путем создания копий фрагментов между узлами.

С точки зрения пользователя такая структура выглядит как обычная древовидная файловая система. Масштабируемость платформы на значительные объемы данных достигается за счет параллельной обработки фрагментов на узлах с использованием программной моделей параллельных распределенных вычислений. Модели распределенных вычислений позволяют разработчикам приложений абстрагироваться от сложностей работы распределенных программ, связанных с распределением данных, планированием нагрузки и обеспечением отказоустойчивости.

В качестве реализации реляционного хранилища данных над Hadoop может быть использован целый ряд систем: Apache Hive [302][340], Big SQL [341], Presto [342], Doris [343], Spark SQL.

В Hive реализованы известные структуры реляционных баз данных – таблицы, столбцы, разделы; поддерживается реляционный язык манипулирования данными HiveQL (стандарт SQL92 с некоторыми дополнениями). Система проецирует реляционную структуру на данные, хранящиеся в Hadoop и предоставляет возможность исполнения SQL-подобных запросов на больших наборах данных путем компиляции их в программы MapReduce, исполняемые в среде Hadoop.

Система *Db2 Big SQL* [22] – проприетарное решение IBM, в основном, поддерживающее стандарты ISO для SQL 2011 и 2016 г., включающее ядро системы управления базами данных и планировщик запросов для управляющего узла Hadoop, ядро системы управления базами данных и компоненты чтения и записи реляционных данных для рабочих узлов Hadoop.

Система *Presto* [342] разработана компанией Facebook как механизм исполнения SQL над Hive и HBase в Hadoop (а также над другими распределенными СУБД – Cassandra, MongoDB, Redis, PostgreSQL и т.д.) для масштабирования на сотни петабайт данных и повышения производительности. Структура Presto включает *менеджер ресурсов*, агрегирующий данные от *координаторов* и *рабочих компонентов*. Координаторы планируют и координируют исполнение запросов, управляют рабочими компонентами. Рабочие компоненты исполняют задачи координатора и обрабатывают данные. Для соединения с разными СУБД используются *коннекторы*.

Spark SQL – это модуль Apache Spark [338] для доступа к структурированным данным в программах на языках Java, Scala, Python, R с использованием текстовых SQL-запросов либо прикладного интерфейса DataFrame. Модуль обеспечивает доступ к данным различных систем и форматов: Hive, Avro, Parquet, ORC, JSON, JDBC.

Система *Doris* [343] нацелена на организацию хранилищ данных реального времени для быстрого исполнения запросов. К системе могут быть подключены как

реляционные СУБД (MySql, PostgreSQL, SQL Server, Oracle) и более низкоуровневые хранилища данных (Hive, Iceberg, Hudi), так и источники потоковых данных (например, Kafka). Структура Doris включает внешний (Frontend) и внутренний (Backend) компоненты. Внешние компоненты отвечают за планирование запросов, управление метаданными и узлами, а внутренние – за хранение данных и исполнение запросов. Система основана на колоночном хранении и высокой компрессии данных, а также различных индексах, нацеленных на минимизацию сканирования данных.

Таким образом, в функциональной структуре обеспечивается возможность распределенного хранения, преобразования и интеграции больших разноструктурированных данных, а также унифицированный взгляд на материализованные данные через реляционную модель. Ниже рассматриваются компоненты функциональной структуры, дополняющие связку платформы распределенных вычислений и распределенной реляционной СУБД.

Извлечение структурированных данных из неструктурированных текстовых осуществляется в рамках отдельной *подсистемы интеллектуального анализа текстов,* например [305][344][345][346]. К подсистеме подключаются источники неструктурированных текстовых данных (например, социальные сети). Производится морфологический, синтаксический, семантический анализ текста, отбор сообщений с кризисной лексикой, извлечение именованных сущностей из сообщений (имена персон, названия организаций, географических объектов, судов и т.д.). Фактически, для системы в целом подсистема сама является источником извлеченных именованных сущностей, представляющих собой структурированные данные.

Для каждого источника структурированных (например, реляционных) или слабоструктурированных (например, в формате XML) данных создается отдельный *загрузчик файлов данных в распределенную файловую систему*. Файлы могут быть экспортированы из источников в различных открытых форматах: JSON, XML, CSV, в виде текстовых файлов, а также в бинарном формате JSON (JSON binary).

Компонент *сопоставления схем* осуществляет автоматизированное сопоставление элементов схем (сущностей, атрибутов) источников и хранилища. В качестве такого компонента может использоваться, например, Harmony [336]. Простые правила сопоставления, представляемые в табличном виде могут быть порождены автоматически или полуавтоматически. Более сложные правила, связанные с разрешением конфликтов различного рода, необходимо описывать вручную с использованием декларативных языков на правилах (например, RIF-FLD [144]).

Компонент *преобразования и интеграции данных* представляет собой набор программ на высокоуровневых языках (SQL, Jaql [294], HIL [296]). Преобразование осуществляется из схем источников данных в единую схему хранилища с разрешением конфликтов между схемами и интеграцией в единые структуры информации хранилища. Основой для создания программ служат правила сопоставления элементов схем.

Язык SQL подходит для интеграции хорошо структурированных табличных данных.

Для работы же со слабоструктурированными данными необходимы другие языки, например, язык запросов и сценариев *Jaql,* использующий формат обмена данными JavaScript Object Notation (JSON [13]). Jaql поддерживает произвольную глубину вложенности структур данных, является в высокой степени функционально-ориентированным, чрезвычайно гибким. Основными структурами данных, подлежащими манипулированию при помощи Jaql, являются объекты (коллекции пар <имя, значение>) и массивы (упорядоченные списки значений). Поддерживается множество встроенных функций над массивами и объектами. Язык предоставляет широкий набор операций преобразования данных (фильтрация, группировка, сортировка, соединение, объединение и т.д.). Программы на Jaql автоматически компилируются в план исполнения задач над Hadoop.

Сложные же потоки обработки данных (очистки, устранения дублирования, слияния) и их интеграции могут быть реализованы с использованием комбинации языков Jaql и HIL. Декларативный язык *HIL* (High-level Integration Language) разработан IBM для программирования сложных потоков обработки данных (ETL), агрегирующих факты из больших коллекций разноструктурированной информации в целевые коллекции унифицированных сущностей. Язык позволяет реализовать методы извлечения, сопоставления и группирования, разбора, связывания, устранения дублирования (deduplication) различных разноструктурированных представлений информации об одних и тех же сущностях реального мира (entity resolution). HIL также позволяет реализовать методы и операции слияния (интеграции) данных об одних и тех же сущностях реального мира и их связей, представленных в разных коллекциях, образованных в процессе разрешения сущностей (включая реализации стратегий и устранения конфликтующих данных, операции поглощения и слияния данных). Программы на HIL компилируется в Jaql, что позволяет использовать HIL для преобразования и интеграции данных в Hadoop.

Языки Jaql и HIL поставляются в составе InfoSphere Master Data Management [298] – программной платформы управления основными данными.

Следует отметить, что методы материализованной интеграции, организации ETL-процессов, очистки данных, предлагаются в составах программных решений ведущих разработчиков баз данных. К таким решениям относится, например, IBM InfoSphere Information Server [347]. Этот программный продукт потенциально может быть использован в комбинированной среде для материализованной интеграции. При этом в качестве хранилища может быть использована платформа IBM InfoSphere Warehouse [348] (являющаяся частью СУБД DB2). Однако, Information Server предлагает лишь ограниченное количество относительно простых методов выделения, отображения и слияния сущностей. Недостаточность таких средств была осознана компанией IBM, в результате чего и появился язык HIL.

Для верификации (формального доказательства необходимых свойств) программ преобразования и интеграции данных, этим программам автоматически, с использованием компонента HIL2AMN сопоставляется формальная семантика в языке спецификаций AMN. Автоматизированное доказательство необходимых свойств осуществляется при помощи программного средства Atelier B [206]. Подробно метод верификации программ интеграции данных изложен и проиллюстрирован в главе 4.

Компонент *Загрузчик информации в РРСУБД* помещает интегрированную информацию в хранилище данных. Свои запросы пользователь задает именно к РРСУБД на языке SQL.

Функциональная структура допускает масштабирование во-первых, по количеству источников данных, из которых извлекается информация и во-вторых, по производительности извлечения, преобразования, интеграции информации и обработке запросов к ней.

Для *подключения нового источника данных* к системе необходимо разработать специализированный для источника загрузчик данных в распределенную файловую систему и настроить его, связав источник и систему извлечения информации. Далее, необходимо разработать программу (программы) для извлечения, преобразования и интеграции информации из данных источника в схему хранилища. При необходимости, программу следует верифицировать с использованием средств HIL2AMN и Atelier B.

Для *увеличения производительности* системы следует расширить вычислительный кластер, на котором развернута система, необходимым количеством узлов. Подобное масштабирование является неотъемлемым свойством платформ рапределенных вычислений.

Разработанная функциональная структура среды материализованной интеграции данных была реализована при создании прототипа системы извлечения и интеграции информации из данных по Арктической зоне [62][63]. Создание прототипа было обусловлено тем, что поисково-спасательные операции (ПСО) в Арктической зоне в современных условиях невозможно представить без всесторонней информационной поддержки. Существует множество источников информации, которые необходимо использовать при поисково-спасательных операциях. Источники содержат данные о маршруте движения судна, данные о гидрометеоусловиях в районе поиска, данные с бортовых систем объективного контроля, анкеты или тексты интервью экспертов, тексты интервью пилотов судов. При работе с этим многообразием информации естественным образом возникает задача по извлечению данных из разноструктурированных источников и их интеграции в единое хранилище. Извлеченная интегрированная информация в хранилище служит основой для планирования поисково-спасательных операций штабами поисковых работ.

Также предложенная функциональная структура была реализована при создании прототипа системы решения задач социально-политического мониторинга [94]. В рамках прототипа была реализована задача мониторинга тональности отношения населения к экономическим и политическим вопросам в конкретном регионе на основе извлечения информации из сообщений в социальных сетях и электронных СМИ. В качестве релевантных задаче были выбраны следующие коллекции:

– коллекция публикаций региональных электронных СМИ;

– коллекция сообщений из социальной сети ВКонтакте, авторы которых проживают в регионе;

– коллекция сообщений, загруженных из сервиса микроблогов Twitter, авторы которых проживают в регионе;

– коллекции профилей пользователей ВКонтакте и Twitter.

Для решения задачи была разработана целевая схема, включающая отношения для представления извлекаемых из текстов сущностей и отношения для представления информации о сообщениях и их авторах. В качестве средств анализа русскоязычных текстов для решения задачи мониторинга использовался семантико-ориентированный процессор Pullenti (Puller of Entities) [344]. Для установления соответствий между элементами схем использовался инструментарий Harmony Schema Matcher [336].

В комбинированной функциональной структуре ПМБРД рассматривается как еще один вид источников, подлежащий виртуальной интеграции. Интеграция становится двухслойной: материализованная интеграция осуществляется внутри ПМБРД, виртуальная – на уровне предметных посредников. Виртуальной интеграции при этом могут подлежать источники:

– данные из которых сложно или невозможно материализовать по разным причинам и (или)

– модель данных включает специфические операции и алгоритмы, адаптация которых в реляционной модели сложна и (или) неэффективна. К таким моделям относятся, например, графовые, тензорные модели и т.д.

Для унификации моделей данных источников в канонической модели предметного посредника и ее верификации применяется метод, подробно изложенный и проиллюстрированный в главе 2. Для верификации интеграции схем источников данных применяется метод, изложенный и проиллюстрированный в главе 3.

Для включения ПМБРД в среду предметных посредников необходимо разработать адаптер для соответствующей РРСУБД в соответствии с подходом, изложенным в работе [183]. При этом модель данных РРСУБД выступает в роли *модели сопряжения ПМБРД со средой предметных посредников*. Модель данных РРСУБД должна быть унифицирована (отображена в каноническую модель посредников). Концептуально унификация модели данных РРСУБД опирается на проведенную при конструировании реляционного адаптера [183] унификацию реляционной модели.

Материализации, как и виртуальной интеграции, могут подлежать данные, представленные в различных традиционных и нетрадиционных моделях. Решение о том, какой способ интеграции следует применить для конкретного источника, следует принимать исходя из целей системы интеграции (например, характерных запросов пользователя), эффективности, стоимости и т.д.

5.1.3 Родственные подходы и системы

Примером подхода, родственного рассмотренной выше комбинированной виртуально-материализованной функциональной структуре, является, например, *федерация* в системе BigSQL [349].

Функциональная структура средств федерации (рис. 5.3) включает федеративный сервер, федеративную базу данных, источники данных, клиентов (пользователей и приложения). Поддерживается федерация для реляционных СУБД Informix, DB2, Oracle, Teradata, SQL Server, Netezza, хранилищ Hive, HBase, S3.

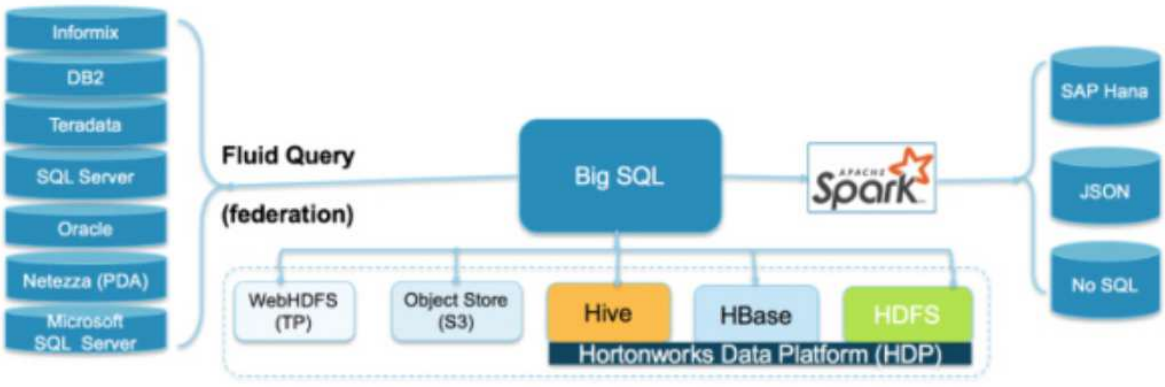

Рисунок 5.3 – Функциональная структура федеративных возможностей Big SQL[82]

Федерация является прямым аналогом виртуальной интеграции в предметных посредниках. Функциональная структура и подход к выполнению распределенных запросов в посредниках [183] и Big SQL обладают рядом сходных черт: слои выполнения программ, адаптеров и источников; планирование запросов; настраиваемые адаптеры. Преимуществом подхода предметных посредников является ориентация на разно-модельные источники. Посредники позволяют интегрировать модельно неоднородные источники: реляционные, объектно-реляционные, XML, веб-сервисы и др. Средства BigSQL ориентированы на виртуальную интеграцию преимущественно реляционных баз данных.

Новый взгляд на федерацию баз данных был предложен в системе BigDAWG [350], введен термин «полихранилища» (polystore). Архитектура полихранилищ направлена на унификацию запросов к источникам данных, основанных на разных моделях данных. В эталонной реализации системы СУБД SciDB используется для хранения временных рядов, хранилище ключ-значение Apache Accumulo – для хранения текстов, PostgreSQL – для хранения структурированных данных. Введено понятие «острова информации» (island of information), характеризующегося моделью данных и языком запросов, а также набором СУБД. Запросы формулируются на языке острова, и для каждой СУБД обеспечивается преобразование из языка острова в язык СУБД. В качестве примеров островов выступают острова реляционной модели и тензорной модели данных. Для соединения в запросе сущностей из разных островов используется конструкция преобразования структуры данных CAST, подзапросы к разным островам могут быть соединены в одном запросе при помощи конструкции SCOPE.

[82] https://www.ibm.com/docs/fr/db2-big-sql/6.0?topic=features-federation-capabilities

Таким образом, в полихранилищах интеграция моделей данных производится лишь внутри островов, интеграция данных между островами производится непосредственно в запросах (такую интеграцию можно назвать ad hoc интеграцией). Уровня интеграции данных, как в предметных посредниках при этом не достигается.

Подход ESTOCADA [351] направлен на усовершенствование архитектуры полихранилищ двумя путями. Во-первых, допускается хранение пересекающихся фрагментов одного набора данных в разнородных хранилищах (основанных на реляционной модели, JSON, ключ-значение, XML). Во-вторых, для повышения эффективности исполнения запросов используются материализованные представления на отдельных хранилищах и переписывания запросов на основе представлений. В выражениях, задающих представления, используются языки запросов различных моделей данных, однако целевой моделью для представлений является реляционная модель, и переписывание запросов происходит именно в ней. Фактически, в данной архитектуре происходит приближение от изначальной архитектуры полихранилищ к архитектуре реляционных предметных посредников.

Разработчики системы Polypheny-DB [352] [353] ставят своей целью превращение полихранилища в полноценную СУБД, вводя при этом термин *полиСУБД* (PolyDBMS). Как и предметные посредники, система соединяется с источниками данных (реляционными, документными, графовыми СУБД) при помощи адаптеров. В качестве языков запросов могут использоваться языки всех трех моделей данных: SQL, MongoQL, Cypher. Запросы преобразуются в логический план – дерево реляционных операторов расширенной реляционной алгебры. Логический план преобразуется в физический план с учетом исполнения подзапросов на источниках данных с передачей через адаптеры. Фактически, в системе осуществляется интеграция трех моделей данных в расширенной реляционной модели. Однако, в отличие от архитектуры предметных посредников, интеграции на уровне схем и переписывания запросов на основе представлений не происходит, и ad hoc интеграцию схем необходимо осуществлять непосредственно в запросах.

Архитектура системы HKPoly [354] представляет из себя еще один промежуточный вариант между классическим полихранилищем и предметным посредником. Система может интегрировать RDF-данные из хранилищ триплетов, данные из графовых и документных СУБД. В качестве языков запросов поддерживаются собственный язык HyQL и SPARQL. Как и в предметных посредниках, запросы осуществляются к глобальной схеме, связанной отображениями с локальными схемами. Кроме того, в системе допускается конструирование потоков работ трансформации данных с

отслеживанием происхождения данных. Однако, модель данных для определения глобальной схемы очень проста (включает лишь сущности и атрибуты) и отображение глобальной и локальной схем сводится к сопоставлению атрибутов.

В работе [355] из университета Токио рассматривается графоцентричная мультимодельная СУБД. В качестве канонической модели используется унифицированная метамодель из работы [236]. Схемы источников данных отображаются в каноническую модель и объединяются в интегрированную схему. Для привязки атрибутов и связей интегрированной схемы к сущностям из источников данных создается *словарь отображений*. В качестве языка запросов используется Cypher.

В работе [356] интеграция моделей данных осуществляется с использованием формализма теории категорий: определена категория унифицированных схем для представления схем реляционной, графовой, RDF, XML, JSON моделей данных. В качестве языка запросов предлагается собственный язык, инкапсулирующий функции и выражения функционального языка программирования Haskell. Запрос исполняется как Haskell-программа над неоднородными наборами данных. Как и в большинстве других полихранилищ, интеграция схем данных производится внутри запросов. К тому же, в системе не производится интеграция с другими СУБД на уровне запросов: данными оперирует Haskell-программа.

В работах [235] [357] из университета Дижона в качестве канонической модели для полихранилища рассматривается тензорная модель данных (TDM). Интеграция с тензорной моделью осуществляется для реляционной, графовой моделей и модели ключ-значение. В тензорной модели определены операции проекции, селекции, объединения, пересечения и соединения тензоров; для операций указаны эквивалентные операции реляционной алгебры. Интеграция схем предполагается при помощи виртуальных взглядов над источниками данных, композиции которых затем материализуются в системе. Переписывание запросов, в отличие от архитектуры предметных посредников, не предполагается. TDM реализована с использованием вычислительной модели Spark на языке Scala. Однако, о программной реализации полихранилища на основе TDM не известно.

В работе [358] из Университета Циньхуа предложен мультимодельный язык запросов ECQL (Equivalent Combinations of Query Languages), ориентированный на реализацию в полихранилищах. Язык оперирует реляционными структурами данных – записями как кортежами типизированных значений атрибутов и коллекциями как списками записей. Базовыми мультимодельными операциями над коллекциями выбраны

проекция, селекция, материализация и декартово произведение. Грамматика языка включает выражения SQL, Cypher и MQL (язык документной СУБД MongoDB). Работа является примером движения в направлении развития языков запросов для полихранилищ, а эклектичная природа языка подчеркивает уровень интеграции данных в хранилищах, не достигающий уровня предметных посредников.

Можно сделать вывод, что разработчики полихранилищ, развивая их архитектуры, стремятся снизить неоднородность и языков запросов в системе и повысить уровень встроенной интеграции данных, однако ввиду стремления к упрощению отображений моделей данных, схем и языков запросов, приблизиться к уровню предметных посредников удается редко.

Специализированный класс систем интеграции данных, часто используемых в промышленности, составляют системы управления основными данными (Master Data Management systems, MDM). *Основными данными* называется информационный объект, хранящийся в MDM-системе, и описывающий эталонным образом некоторый объект реального мира, отдельные аспекты которого представлены в виде информационных объектов в разных источниках данных. MDM-системы позволяют синхронизировать основные данные между различными бизнес-приложениями. Так, например, в системе АрхиГраф в качестве канонической модели основных данных используется онтологическая модель, основанная на стандартах RDF, RDFS, OWL. Над онтологией основных данных обеспечивается логический вывод. Возможна также организация онтологии как логической (виртуальной) витрины данных, при этом источниками данных могут быть СУБД Oracle, SQL Server, PostgreSQL, MongoDB, точки доступа SPARQL. Однако, фактически, средства интеграции данных в АрхиГраф либо слишком просты (требуется установление соответствия атрибута схемы источника данных атрибуту схемы основных данных), либо отдаются на откуп администратору (программная реализация нестандартной процедуры-обработчика). Отсутствуют продвинутые средства переписывания запросов, реализуемые в системах класса OBDI [132], а также встроенные средства реализации ETL-процессов.

5.2 Схема функциональной структуры основанной на правилах мультидиалектной среды

Раздел основан на работах автора [91][55][56][71][57].

5.2.1 Мотивация, основные свойства и базовые технологии функциональной структуры

Схема функциональной структуры, рассматриваемая в данном разделе, направлена на обеспечение средств поддержки разнообразия данных (разнородность типов данных, их представления и семантической интерпретации), семантическую интеграцию данных и поддержку декларативных спецификаций, необходимых для разработки сложных конвейеров с использованием языков высокого уровня для решения аналитических задач в областях с интенсивным использованием данных [3]. Функциональная структура обеспечивает согласованное совместное использование многообразия средств логического программирования, представления знаний, Семантического веба и предметных посредников для интеграции неоднородных баз данных.

Одна из основных идей *концептуального моделирования* состоит в том, что концептуальные спецификации решения задач должны быть выражены в терминах прикладной предметной области приложения независимо от их реализации. Обычно *спецификация предметной области*, понятная и приемлемая в соответствующем сообществе, включает набор последовательно связанных онтологий вместе с *концептуальной схемой* предметной области, определенной на основе таких онтологий. Концептуальные спецификации приложений в этой предметной области определяются в терминах концептуальной схемы для выражения абстрактных представлений структур данных и алгоритмов анализа данных, направленных на решение прикладных задач.

В течение многих лет для концептуального моделирования использовались различные языки, направленные на определение семантики вычислений в терминах решаемых прикладных задач. Преобладали структурированные и объектно-ориентированные спецификации графической (диаграммной) форме. Так, с использованием подхода «сущность-связь» (ER) предметная область моделируется в терминах сущностей, их атрибутов и связей между ними. Язык UML включает диаграммы для конкретных целей (диаграммы классов, диаграммы действий, диаграммы состояний, диаграммы взаимодействия объектов), используемые для проектирования программного обеспечения. При этом семантика спецификаций часто определяется неформально, в терминах естественного языка. Также для описания семантики данных были разработаны различные онтологические языки, основанные на дескриптивных

логиках. Языки онтологий позволяют определять формальные модели понятий предметной области, их таксономические отношения и ограничения над ними. При этом обеспечивается автоматический логический вывод над таксономическими структурами. Однако, одной семантики данных недостаточно, и для решения задач в областях с интенсивным использованием данных необходимо совместное определение семантики данных и поведения для описания алгоритмов решения задач.

Рассматриваемая функциональная структура направлена на определение средств концептуальной спецификации и реализации решения задач в областях с интенсивным использованием данных, обладающих следующими свойствами:

– декларативность используемых языковых средств;

– способность предоставлять полную и точную спецификацию абстрактной структуры и поведения объектов предметной области, их связей и взаимодействия;

– обоснованное разнообразие семантики средств моделирования, обеспечивающее надлежащую выразительность, компактность и точность определения поведения объектов предметной области и спецификаций алгоритмов решения задач;

– возможность расширения средств моделирования концептуальных спецификаций, удовлетворяющих меняющимся технологическим и практическим потребностям;

– обеспечение независимости спецификаций от языков и систем реализации;

– обеспечение независимости спецификаций от конкретных источников данных и сервисов в сочетании со средствами их семантической интеграции и интероперабельности, позволяющими справиться с растущим разнообразием методов реализации программ, баз данных, сервисов, онтологий и других источников;

– встроенная методология создания унифицирующих языков спецификаций, обеспечивающие построение сохраняющих семантику отображений концептуальных спецификаций в их реализации на конкретных языках и системах;

– достаточная эффективность и выразительность для публикации в сообществах, нацеленной на повторное использование спецификации.

Мотивацией к разработке функциональная структуры послужили исторические тенденции развития средств концептуального моделирования и появление Формата обмена правилами (Rule Interchange Format – RIF) [140] для Сематического веба как рекомендации консорциума W3C.

В рассматриваемой функциональной структуре функциональная совместимость декларативных программ, определенных на языках с различной семантикой, сочетается

со способностью интегрировать различные источники данных (базы данных и знаний, программные сервисы, онтологии) для решения задач.

Семантическая интеграция источников данных достигается с использованием предметных посредников (соответствующая общая функциональная структура рассматривается в предыдущем разделе), при этом применяется метод верифицируемой унификации моделей данных источников, рассмотренный в главе 2.

Другой мультидиалектный метод взаимодействия программ на основе правил, применяемый в функциональной структуре, основан на стандарте RIF (формат обмена правилами) [140] W3C. RIF представляет собой единое семейство языков (диалектов), основанных на правилах, и предлагает методологию построения сохраняющих семантику отображений конкретных языков, используемых в различных системах на правилах[83], в такие диалекты. В рамках RIF также разработана методология определения новых диалектов для расширения семейства.

Согласно идеологии RIF, любая конкретная система на правилах может выступать в двух независимых ролях – роли *поставщика* и роли *потребителя*. Роль поставщика означает, что система на правилах способна преобразовывать свои собственные программы на правилах, в программы на некотором диалекте RIF. Роль потребителя означает, что система способна принимать программу, определенную на некотором диалекте, и преобразовывать ее в программу на своем языке. Таким образом, для включения любого языка на правилах в семейство интероперабельных диалектов требуется обеспечить в поддерживающей его системе на правилах две сохраняющих семантику трансформации – из этого языка в соответствующий диалект и из диалекта на в язык. Каждый диалект может иметь свою собственную, отличную от других, семантику. Например, диалекты на логических правилах могут использовать классическую семантику первого порядка, хорошо обоснованную семантику (well-founded semantics, WFS) [359], семантику стабильных моделей (stable model semantics) [360] и т.д.

В стандарте RIF фиксированы базовые логический и продукционный диалекты. Базовый логический диалект RIF-BLD [141] соответствует Хорновским правилам с некоторыми синтаксическими и семантическими расширениями. Диалект продукционных правил RIF-PRD [142] отражает основные аспекты различных систем на

[83] К таким системам относятся, в частности, OntoBroker (https://www.semafora-systems.com/ontobroker-and-ontostudio-x), IBM Operational Decision Manager (https://www.ibm.com/products/operational-decision-manager), DLV (https://www.dlvsystem.it/dlvsite/dlv/).

продукционных правилах. Стандарт также включает каркас логических диалектов RIF-FLD [144]. RIF-FLD включает широкий набор синтаксических и семантических конструкций, часто используемых в логике. На основе этой структуры могут быть определены конкретные диалекты, имеющие специфическую семантику. На основе каркаса RIF-FLD в мире были разработаны диалекты RIF-CLPWD с хорошо обоснованной семантикой и RIF-CASPD с семантикой стабильных моделей (такие языки относятся к классу Answer Set Programming –ASP). WFS и ASP являются типичными примерами логических языков с различной семантикой и областями применения, и соответствуют двум философски различным подходам в логическом программировании. WFS придерживается идеи определения одной (хорошо обоснованной) модели для программы; ASP же обеспечивает вычисление набора предпочтительных моделей. ASP-системы, ориентированы на решение сложных комбинаторных (NP-полных) задач. WFS-системы используют трехзначную логику, являются вычислительно полными и могут использоваться в качестве универсальных средств логического программирования.

Стандарт RIF также определил необходимые концепции для обеспечения совместимости с RDF и OWL, несмотря на различия в их синтаксисе и семантике. Основная идея заключается в том, что программы на правилах и онтологии OWL рассматривают друг друга как «черные ящики» с четко определенными интерфейсами через экспортируемые предикаты. Онтологии экспортируют свои классы и свойства, а базы знаний на правилах, экспортируют определяемые ими предикаты. Каждая база знаний может ссылаться на предикаты, определенные в других базах знаний.

5.2.2 Основные принципы функциональной структуры

Функциональная структура мультидиалектной среды предусматривает совместное функционирование модулей предметных посредников, интегрирующих данные и сервисы, а также модулей, представляющих программы, основанные на знаниях и декларативных правилах, и основана на следующих принципах (Рис. 5.4):

– *Мультидиалектные концептуальные спецификации задач создаются на основе сочетания подходов предметных посредников и RIF.* Спецификации представляются в виде функциональной композиции декларативных модулей, каждый из которых основан на своем собственном языке (диалекте) с соответствующей семантикой. Языку посредников должен соответствовать один из существующих диалектов RIF (или новый, специфический диалект).

– *Модули спецификации рассматриваются как узлы одноранговой сети (peers).* Взаимодействие узлов основывается на методах обеспечения интероперабельности RIF.

– *Интеграция источников данных и сервисов обеспечивается в рамках отдельных модулей-посредников.*

– *Модули, обеспечивающие логический вывод на правилах, могут использоваться для декларативного программирования над посредниками*.

– *Мультидиалектные концептуальные спецификации задач реализуются путем пирингового взаимодействия* предметных посредников и систем на правилах, входящих в среду, при помощи техники *делегирования* (передачи фактов и правил от одного узла к другому) [361].

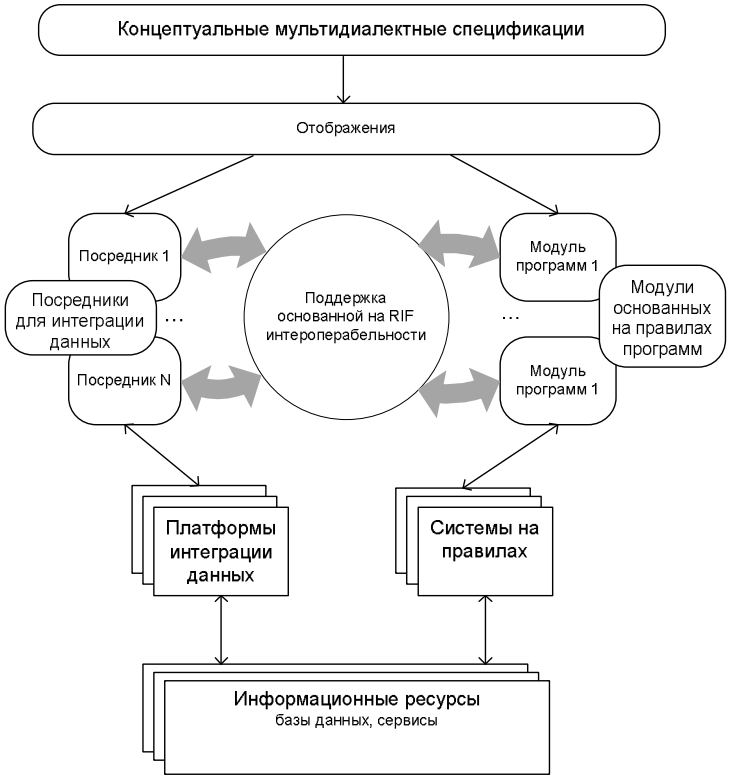

Рисунок 5.4 – Мультидиалектная среда концептуального решения задач

5.2.3 Концептуальное программирование над концептуальной схемой

Концептуальная спецификация задачи (класса задач) определяется в контексте предметной области и состоит из набора RIF-документов (*документ* является единицей спецификации RIF). Каждый документ содержит *группы правил*. Концептуальная спецификация задачи – это абстрактная программа решения задачи.

Концептуализация предметной области выполняется с использованием онтологий языка OWL [11], включающих понятия предметной области и связи между ними, и формирующих *концептуальную схему* (рис. 5.5, правая часть).

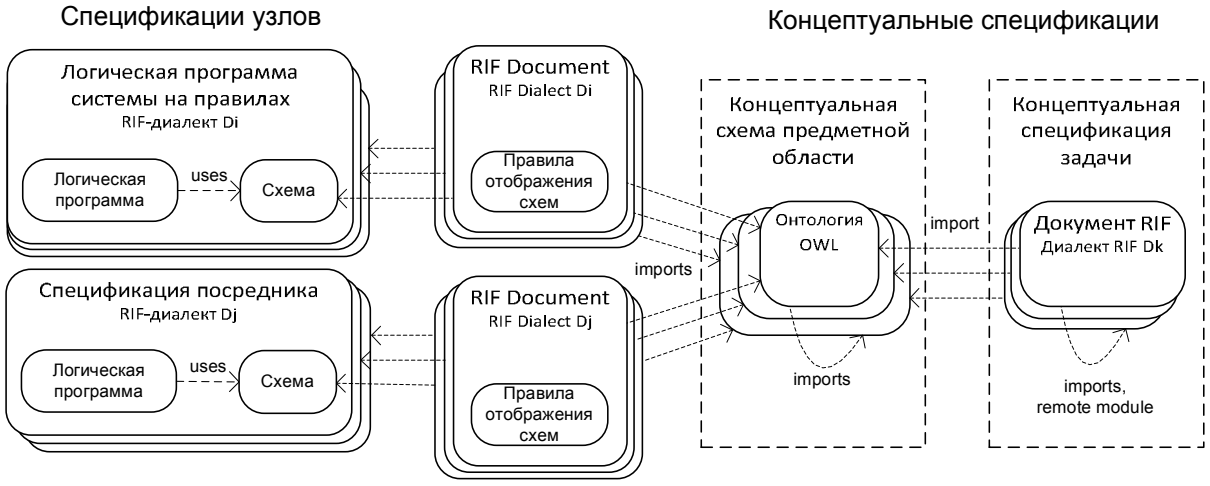

Рисунок 5.5 – Концептуальные спецификации и спецификации узлов

Концептуальная спецификация определяется *над* концептуальной схемой: имена сущностей (классов и атрибутов) концептуальной схемы и только они используются в правилах RIF-документов в качестве предикатов. Модульное построение концептуальной спецификации основано на методах *импорта* и *связывания* документов. Каждый документ RIF определяется с использованием диалекта, который является специализацией RIF Framework for Logic Dialects (RIF-FLD) [144]. Документы импортируют другие документы, имеющие ту же семантику (директива *Import*), или устанавливают связи с документами, определенными с использованием других диалектов и имеющими другую семантику (директива удаленного[84] модуля *Module*). Документы ссылаются на предикаты из импортированных документов (используя название импортируемого модуля *module:predicate*), а также на предикаты из удаленных модулей, используя *удаленные термы f@r* [144]. Здесь *f* означает терм, а *r* – ссылку на удаленный модуль.

Извлечение результатов решения задачи осуществляется путем запроса к виртуальной базе знаний, сформированной объединенной средой посредников и систем на правилах. Запросы формулируются в терминах концептуальной схемы и предикатов концептуальной спецификации задачи. Результатом является логическое значение (если формула замкнута) или набор кортежей значений свободных переменных формулы.

5.2.4 Отображение схем узлов в концептуальную схему

Схема S_R узла R мультидиалектной среды представляет собой набор сущностей – классов или отношений, предикатов логических программ и их атрибутов.

---

[84] В оригинале – remote module.

Система на правилах или посредник каждого узла *R* должны быть потребителями и поставщиками некоторого диалекта D_R (рис. 5.5, левая часть).

Узел *R релевантен* RIF-документу *d* концептуальной спецификации задачи (рис. 5.5, правая часть), если:

– D_R является поддиалектом диалекта документа *d* и

– сущности схемы S_R *онтологически релевантны*[85] сущностям концептуальной схемы, имена которых используются в *d*.

Схема релевантного узла *отображается* в концептуальную схему. Отображение устанавливает соответствие концептуальных сущностей, на которые ссылаются документ *d*, их выражениям в терминах сущностей схемы S_R с использованием логических правил диалекта D_R. Эти правила отображения схем представляют собой отдельный RIF-документ (рис. 5.5, средняя часть).

5.2.5 Реализация концептуальной спецификации

Программы, определенные в концептуальной спецификации, реализуются в пиринговой среде, сформированной релевантными узлами, схемы которых связаны с концептуальной спецификацией правилами сопоставления (рис. 5.5, средняя часть).

Узлы взаимодействуют на основе подхода к распределенному выполнению логических программ. Основная идея этого подхода заключается в *делегировании* – передаче фактов и правил от одного узла к другому. Узел представляет собой комбинацию адаптера и системы на правилах или посредника (рис. 5.6). Адаптер преобразует программы и факты из диалекта RIF в язык системы на правилах или язык посредника и наоборот. Адаптер также реализует механизм делегирования. Передача фактов и правил между узлами выполняется на диалектах RIF. Оболочки реализуют интерфейс RIFNodeWrapper (рис. 5.6).

Компонент функциональной структуры *Супервизор* хранит общую информацию о среде: концептуальные спецификации предметной области и задачи, список релевантных источников, RIF-документы отображения схем источников в концептуальную схему.

---

[85] Онтологическая релевантность устанавливается с использованием методов и средств сопоставления онтологий [220].

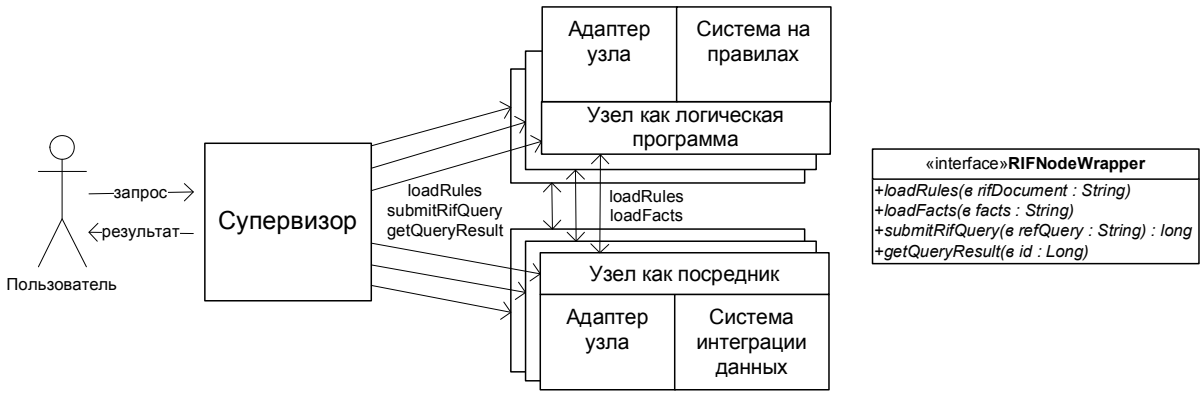

Рисунок 5.6 – Компоненты пиринговой функциональной структуры

Реализация концептуальной спецификации включает в себя следующие этапы:

– Переписывание Супервизором документов концептуальной спецификации в RIF-программы для узлов. Переписывание включает в себя (1) замену идентификаторов документов в именах предикатов на идентификаторы узлов и (2) добавление в программы правил отображения схем.

– Передача переписанных программ на узлы, релевантные документам концептуальной схемы. Передача выполняется Супервизором путем вызова метода *loadRules* адаптеров соответствующих узлов.

– Преобразование адаптерами RIF-программ в конкретные языки систем на правилах или посредников.

– Выполнение созданных программ в пиринговой среде.

В процессе переписывания концептуальной спецификации в программы для узлов связи между RIF-документами концептуальной спецификации, определяемыми удаленными или импортированными термами, заменяются связями между узлами, также определяемыми удаленными или импортированными термами. Для реализации удаленных и импортированных терминов используется механизм делегирования правил.

Основные идеи реализации механизма делегирования состоят в следующем. В общем случае программы, передаваемые какому-либо узлу, могут содержать *нелокальные* правила, содержащие удаленные или импортированные термы. Напротив, *локальные* правила не содержат таких термов. Чтобы сделать возможным выполнение программы на узле, программа должна быть *нормализована*, т.е. преобразована в эквивалентную программу, включающую только локальные правила и правила делегирования. Рассматриваются два вида правил делегирования для узла *n*:

– правило делегирования фактов вида *p@m(X):- q@n(X)*, где *p, q* – имена предикатов, *X* – список переменных, *m* – узел, отличный от *n*. Это правило означает, что все факты, обращающие предикат *q* в истину, должны быть переданы узлу *m* как факты,

обращающие предикат p в истину. *Фактом* называется такой терм $p(v_1, \ldots, v_n)$, что $p(v_1, \ldots, v_n) = true$, где $v_1, \ldots, v_n$ – значения переменных $X = x_1, \ldots, x_n$;

– правило делегирования вида $q@n(X) :- p@m(X)$. Правило должно быть передано узлу m, где оно становится правилом делегирования фактов.

Разработаны алгоритмы нормализации логических программ и исполнения программ в пиринговой среде [55].

5.2.6 Моделирование и реализация мультидиалектных потоков работ

Для моделирования процессов решения задач в виде потоков работ в функциональной структуре предлагается использование диалекта продукционных правил RIF PRD [142]. Документы, представленные при помощи данного диалекта, наряду с документами, представленными при помощи логических диалектов, становятся частью концептуальной спецификации задачи.

Поток работ состоит из набора деятельностей (task), организованных с помощью определенных конструкций – образцов потоков работ [218], определяющих порядок выполнения деятельностей. Характерные образцы – это, например, *последовательность* (sequence), *разделение* (split), *соединение* (join).

Спецификация такой организации называется *каркасом потока работ*. Каркас определяется с использованием продукционных правил RIF-PRD. Применяемый подход для представления потоков требует расширения диалекта RIF-PRD несколькими встроенными предикатами (они считаются частью пространства имен *wkfl*, которое определяется аналогично пространствам имен *func* и *pred*, определенным в [143] для встроенных функций и предикатов RIF):

– Предикат *wkfl:end-of-task(?arg)*, где *?arg* – идентификатор деятельности. Множество значений *?arg* – это тип данных *xsd:Name*, представляющий имена в языке XML Schema. Предикат становится истинным, когда деятельность завершена.

– Предикат *wkfl:variable_definition(?arg1 ?arg2)*, где *?arg1* – идентификатор переменной, *?arg2* – идентификатор типа переменной. Множество значений для обоих аргументов – тип *xsd:Name*. Истинность предиката означает, что переменная *?arg1* типа *?arg2* определена в контексте потока работ.

– Предикат *wkfl:variable_value(?arg1 ?arg2)*, где *?arg1* – идентификатор переменной, *?arg2* – значение переменной. Множество значений для первого аргумента – *xsd:Name*, множество значений для второго аргумента это объединение множеств значений всех встроенных типов данных RIF. Истинность предиката означает, что переменная *?arg1* принимает значение *?arg2*.

– Предикат *wkfl:parameter_definition(?arg1 ?arg2 ?arg3)*, где *?arg1* – идентификатор параметра потока работ, *?arg2* – идентификатор типа параметра, *?arg3* – вид параметра. Множество значений для первого и второго аргументов – *xsd:Name*. Множество значений для третьего аргумента – {IN, OUT, IN_OUT} (входной, выходной параметр или входной и выходной параметр одновременно). Истинность предиката означает, что для рабочего процесса определен параметр *?arg1* типа *?arg2* и вида *?arg3*.

– Предикат *wkfl:parameter_value(?arg1 ?arg2)* определяет значения параметров потоков работ таким же образом, как и *wkfl:variable_value* определяет значения переменных потоков работ.

Предикаты *wkfl:variable_definition* и *wkfl:variable_value* позволяют задавать переменные рабочего процесса и их значения и, таким образом, организовывать поток данных в рамках потока работ. Предикаты *wkfl:parameter_definition* и *wkfl:parameter_value* позволяют задавать параметры потока работ и их значения и, таким образом, определять интерфейс рабочего процесса в терминах входных и выходных параметров.

Предикат *wkfl:end-of-task(?arg)* позволяет управлять порядком выполнения деятельностей потоков работ, используя условия и действия (action) продукционных правил. Ниже представлены шаблоны правил, предназначенные для представления основных образцов потоков работ (рис. 5.7).

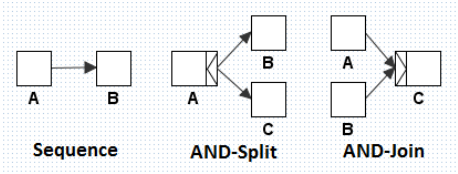

Рисунок 5.7 – Основные образцы потоков работ

Образец *AND-Split* (И-разделение) представляется в RIF PRD следующим шаблоном продукционных правил:

```
If Not(External(wkfl:end-of-task(A)))
Then Do (Act(A)  Assert(External(wkfl:end-of-task(A))))
If And(Not(External(wkfl:end-of-task(B))) External(wkfl:end-of-task(A)))
Then Do (Act(B)  Assert(External(wkfl:end-of-task(B))))
If And(Not(External(wkfl:end-of-task(C))) External(wkfl:end-of-task(A)))
Then Do (Act(C)  Assert(External(wkfl:end-of-task(C))))
```

Шаблон включает три правила для деятельностей *A, B, C* соответственно. *Act(A), Act(B), Act(C)* обозначают действия, ассоциируемые с деятельностями *A, B, C*. Порядок выполнения деятельностей (деятельности *B* и *C* исполняются одновременно после завершения деятельности *A*) описывается с использованием предиката *wkfl:end-of-task* в условиях правил и действия *Assert*, определенного в RIF PRD.

Аналогично, образец *AND-Join* (И-соединение) представляется в RIF PRD следующим шаблоном продукционных правил:

```
If Not(External(wkfl:end-of-task(A)))
Then Do (Act(A)  Assert(External(wkfl:end-of-task(A))))
If And(Not(External(wkfl:end-of-task(B))) External(wkfl:end-of-task(A)))
Then Do (Act(B)  Assert(External(wkfl:end-of-task(B))))
If And(Not(External(wkfl:end-of-task(C))) External(wkfl:end-of-task(A)))
Then Do (Act(C)  Assert(External(wkfl:end-of-task(C))))
```

Образец *Sequence* (последовательность) представляется в RIF PRD следующим шаблоном продукционных правил:

```
If Not(External(wkfl:end-of-task(A)))
Then Do (Act(A)  Assert(External(wkfl:end-of-task(A))))
If And(Not(External(wkfl:end-of-task(B))) External(wkfl:end-of-task(A)))
Then Do (Act(B)  Assert(External(wkfl:end-of-task(B))))
```

Аналогичным образом в RIF PRD представляются и другие образцы потоков работ: ИЛИ- и ИСКЛЮЧАЮЩЕЕ ИЛИ-разделения и соединения, структурированные циклы, подпотоки и т.д.

Деятельности потока работ могут быть заданы различными способами:

- в виде отдельных RIF-документов на различных логических RIF-диалектах;
- в виде отдельных RIF-документов на диалекте RIF-PRD;
- в виде набора продукционных правил, встроенных в каркас потока работ;
- в виде внешних функций, рассматриваемых как “черные ящики”.

Семантика деятельностей, определенных как мультидиалектные логические программы, определяется в соответствии со стандартом RIF-FLD и стандартами для соответствующих диалектов RIF (BLD, CASPD и других). Семантика деятельностей, определенных как программы на продукционных правилах, определяется в соответствии со стандартом RIF-PRD. Предполагается, что семантика внешних функций “задана извне в некотором документе” [144]. Все виды деятельностей (за исключением тех, которые встроены в каркас потока работ) указываются в каркасе рабочего процесса как *внешние термы* вида *External(t)*, где терм *t* определяется внешним ресурсом, идентифицируемым с помощью интернационализированного идентификатора ресурса (IRI) [144].

Потоки работ, определенные в концептуальной спецификации, реализуются в мультидиалектной среде (рис. 5.8). Системы на продукционных правилах (СПП), а также веб-сервисы, реализующие внешние функции, становятся компонентами среды.

Реализация концептуальной спецификации с потоками работ включает в себя следующие этапы:

– Передача концептуальных RIF-документов, составляющих каркас потока работ, в узел системы на продукционных правилах, выполняемая компонентом *Супервизор* (метод *loadWorkflow* Адаптера СПП).

– Преобразование концептуальных RIF-документов, составляющих каркас потока работ, в язык системы на продукционных правилах, выполняемое компонентом *Адаптер СПП*.

– Передача логических программ RIF, отвечающих деятельностям, в соответствующие узлы среды и преобразование RIF-программ в конкретные языки систем на правилах или посредников (метод *loadRules* адаптеров узлов).

– Выполнение потока работ.

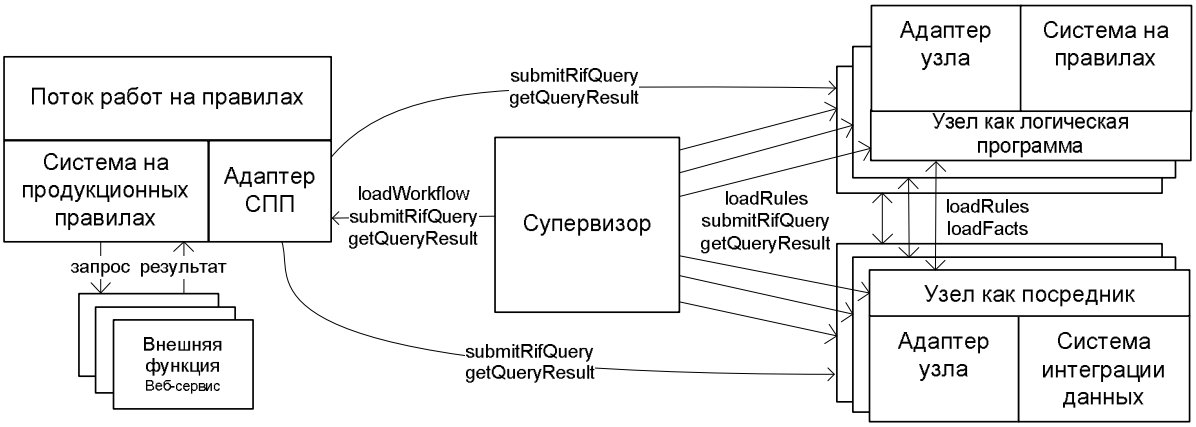

Рисунок 5.8 – Компоненты функциональной структуры с учетом потоков работ

Методы верификации интеграции моделей данных и схем данных, изложенные в главах 2 и 3 применяются в рассмотренной среде при создании предметных посредников (для унификации моделей данных источников и интеграции схем источников) и адаптеров узлов систем на правилах (для разработки отображений диалектов RIF в языки систем).

На основе функциональной структуры, рассмотренной в данном разделе, был реализован опытный образец инфраструктуры [55][56]. Подходы к концептуализации проблемной области на нескольких диалектах, делегированию правил, взаимодействию программ на основе правил и предметных посредников подробно объясняется и иллюстрируется на примере задачи информационной поддержки выбора диверсифицированного инвестиционного портфеля [362]. При концептуальном моделировании задачи язык OWL использовался для определения понятий предметной

области, диалекты RIF-BLD и RIF-CASPD – для определения логических программ, диалект RIF-PRD – для определения потока работ. Данные в виде временных рядов стоимости ценных бумаг, извлекались из источников Google Finance и Yahoo Finance. Для реализации и исполнения программы на диалекте RIF-CASPD использовалась система DLV[86]. Для реализации и исполнения программы на диалекте RIF-PRD использовался язык ILOG JRules и система IBM Operational Decision Manager.

[86] https://www.dlvsystem.it/dlvsite/dlv

ЗАКЛЮЧЕНИЕ

Основные результаты диссертационной работы заключаются в следующем.

1. Разработан новый метод конструирования сохраняющих семантику отображений моделей данных. Метод сочетает формальное определение синтаксиса и семантики моделей данных, интеграцию синтаксисов исходной и целевой моделей, создание расширений целевой модели, автоматизированное конструирование трансформации исходной модели в целевую, верификацию корректности отображения моделей. Разработаны три метода верификации корректности отображения моделей: на наборе образцов спецификаций исходной модели, на основе уточнения единых спецификаций исходной и целевой моделей, на основе эквивалентного представления исходной и целевой моделей в единой семантической структуре. Метод применен для унификации сетевой, процессной, онтологической, графовой, тензорной моделей данных, языка логических правил.

2. Разработан новый метод верификации интеграции схем данных в системах виртуальной интеграции. Первый вариант метода нацелен на верификацию интеграции схем в предметных посредниках, использующих объектную модель в качестве ядра унифицирующей канонической модели данных. Вариант метода основан на уточнении абстрактных типов данных с использованием отображения канонической модели в формальный язык спецификаций. Второй вариант метода нацелен на верификацию интеграции схем в системах виртуальной интеграции данных, использующих модель субъект-предикат-объект (RDF) в качестве унифицирующей канонической модели данных. Корректность интеграции проверяется на наборе запросов пользователя к системе интеграции, выражаемых на языке SPARQL. Корректность доказывается путем отображения схем предметной области, схем источников данных и запросов в формальный язык спецификаций, и дальнейшим применением автоматизированных средств доказательства уточнения.

3. Разработан новый метод верификации интеграции данных в системах материализованной интеграции данных. Программе интеграции данных, включающей операции трансформации данных, разрешения сущностей, слияния данных, сопоставляется формальная семантическая спецификация. Свойства программы, подлежащие проверке, представляются в виде выражений выбранного языка спецификаций. Затем, с использованием формальных средств доказательства, спецификация, выражающая семантику программы интеграции данных, проверяется на соответствие необходимым свойствам. Дополнительно разработан метод спецификации правил интеграции данных с использованием языка логических правил и их реализации

на языке интеграции данных. Тем самым обеспечивается возможность верификации свойств спецификаций интеграции данных, выраженных на высокоуровневых логических правилах.

4. Разработаны новые программные средства автоматизации методов интеграции данных, включающие средства поддержки процесса унификации моделей данных, генерации заготовок трансформаций моделей, семантической трансформации ядра канонической объектной модели данных и языка программ интеграции данных в формальный язык спецификаций. В рамках работы по диссертации получены 12 свидетельств о регистрации программы для ЭВМ.

5. Разработаны схемы функциональных структур сред решения задач над неоднородными источниками данных с поддержкой верифицируемой интеграции данных. Функциональная структура комбинированной среды интеграции неоднородных коллекций данных нацелена на поддержку как виртуальной, так и материализованной интеграции больших коллекций данных, представленных как в традиционных, так и нетрадиционных моделях данных на платформе распределенных вычислений. Функциональная структура была реализована при создании прототипа системы решения задач социально-политического мониторинга. Функциональная структура основанной на правилах мультидиалектной среды нацелена на согласованное совместное использование многообразия средств логического программирования, представления знаний, Семантического веба и предметных посредников для решения аналитических задач в областях с интенсивным использованием данных. Функциональная структура была реализована при создании прототипа системы решения задачи информационной поддержки выбора диверсифицированного инвестиционного портфеля.

6. Разработанные методы и средства применены для верификации интеграции данных в системе «Умное землепользование», разрабатываемой в Почвенном институте имени В. В. Докучаева; в интегрированной базе данных Института металлургии и материаловедения им. А.А. Байкова РАН; в базе данных двойных звезд, разрабатываемой в Институте астрономии РАН; в распределенном хранилище для поддержки поисково-спасательных операций в Арктической зоне.

Перспективы дальнейшей разработки темы состоят в применении больших языковых моделей для автоматической генерации заготовок трансформации моделей (в настоящее время корпус трансформаций моделей данных для обучения языковых моделей относительно мал), а также для автоматической генерации доказательств теорем верификации интеграции данных (для которых в настоящее время используются интерактивные средства доказательства).

Благодарности. Автор выражает глубокую благодарность своему учителю, руководителю дипломной и кандидатской работ – профессору Леониду Андреевичу Калиниченко. Леонид Андреевич направил научные устремления автора в его студенческие годы в область формальных методов и интеграции данных, в течение восемнадцати лет поддерживал и консультировал автора в его исследованиях.

Автор признателен научному консультанту работы – ученому секретарю ФИЦ ИУ РАН Виктору Николаевичу Захарову, профессору ВМК МГУ Владимиру Александровичу Сухомлину и сотрудникам отдела Методов и средств накопления и обработки данных ФИЦ ИУ РАН Д. О. Брюхову, Н. А. Скворцову, Д. Ю. Ковалеву за содержательные обсуждения результатов работы на научных конференциях, семинарах и совещаниях отдела.

ЛИТЕРАТУРА

1 Hey T., Tansley S., Tolle K. The Fourth Paradigm: Data-Intensive Scientific Discovery. — Redmond: Microsoft Research, 2009.

2 Проблемы доступа к данным в исследованиях с интенсивным использованием данных в России / Л. А. Калиниченко, А. А. Вольнова, Е. П. Гордов et al. // Информ. и её примен. — 2016. — Vol. 10. — P. 2–22.

3 Challenges and opportunities with big data. — A community white paper developed by leading researchers across the United States, 2012. — Access mode: http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf.

4 Salmi J. Study on open science: Impact, implications and policy options. — European Commission, Directorate-General for Research and Innovation, Publications Office, 2015. — Access mode: https://data.europa.eu/doi/10.2777/133494.

5 The fair guiding principles for scientific data management and stewardship / Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, et. al. // Scientific Data. — 2016. — Mar. — Vol. 3, no. 1. — P. 160018. — Access mode: https://doi.org/10.1038/sdata.2016.18.

6 ISO/IEC 9075-2:2023 Information technology — Database languages SQL. — ISO, 2023. — Access mode: https://www.iso.org/standard/76584.html.

7 Urban S. D., Dietrich S. W. Object data models // Encyclopedia of Database Systems. — Springer, 2018. — P. 2525–2532.

8 Baumann P. http://dx.doi.org/10.1007/978-1-4614-8265-9_2061Array Databases // Encyclopedia of Database Systems / Ed. by Ling Liu, M. Tamer Özsu. — New York, NY : Springer New York, 2018. — P. 165–177.

9 Wood P. T. Graph database // Encyclopedia of Database Systems. — Springer, 2018. — P. 1639–1643.

10 Klyne G., Carroll J. J., McBride B. Rdf 1.1 concepts and abstract syntax. w3c recommendation. — 2014. — Access mode: https://www.w3.org/TR/rdf11-concepts/.

11 Owl 2 web ontology language structural specification and functional-style syntax (second edition). w3c recommendation. — 2012. — Access mode: https://www.w3.org/TR/owl-syntax/.

12 Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation / T. Bray, J. Paoli, C. M. Sperberg-McQueen et al. — 2008. — Access mode: https://www.w3.org/TR/xml/.

13 ECMA-404 The JSON data interchange syntax. 2nd edition. — ECMA, 2017. — Access mode: https://ecma-international.org/publications-and-standards/standards/ecma-404/.

14 Palmer N. XML Process Definition Language // Encyclopedia of Database Systems / Ed. by Ling Liu, M. Tamer Özsu. — Springer US, 2009. — P. 3601–3601.

15 Web Services Business Process Execution Language Version 2.0. OASIS Standard. — 2007. — Access mode: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

16 Business Process Model and Notation (BPMN). — Object Management Group, 2013. — Access mode: http://www.omg.org/spec/BPMN.

17 Oberweis A., Reussner R. On the de-facto standard of event-driven process chains: How epc is defined in literature // Modellierung. — 2016. — Vol. 2, no. 4. — P. 61–76.

18 Tompa F. Document databases // Encyclopedia of Database Systems. — Springer, 2018. — P. 1237–1238.

19 Jaakkola H., Thalheim B. Sixty years –- and more –- of data modelling // Frontiers in Artificial Intelligence and Applications. — 2020. — Vol. 333. — P. 56–75. — Access mode: https://ebooks.iospress.nl/volumearticle/56435.

20 Putrama I. M., Martinek P. Heterogeneous data integration: Challenges and opportunities // Data in Brief. — 2024. — Vol. 56. — P. 110853.

21 Cheatham M., Pesquita C. Semantic data integration // Handbook of big data technologies. — 2017. — P. 263–305.

22 Semantic data integration and querying: a survey and challenges / Maroua Masmoudi, Sana Ben Abdallah Ben Lamine, Mohamed Hedi Karray et al. // ACM Computing Surveys. — 2024. — Vol. 56, no. 8. — P. 1–35.

23 Omg meta object facility (mof) core specification. version 2.5.1. — 2016. — Access mode: https://www.omg.org/spec/MOF/2.5.1/PDF.

24 Models versus model descriptions / Joachim Fischer, Birger Møller-Pedersen, Andreas Prinz, Bernhard Thalheim // Modelling to Program / Ed. by Ajantha Dahanayake, Oscar Pastor, Bernhard Thalheim. — Cham : Springer International Publishing, 2021. — P. 67–89.

25 Wiederhold G. Mediation in information systems // ACM Comput. Surv. — 1995. — Vol. 27, no. 2. — P. 265–267.

26 Mediation framework for enterprise information system infrastructures: Application-driven approach / Leonid A. Kalinichenko, Dmitry O. Briukhov, Dmitry O. Martynov et al. // ICEIS 2007 - Proceedings of the Ninth International Conference on Enterprise Information Systems, Volume DISI, Funchal, Madeira, Portugal, June 12-16, 2007 / Ed. by Jorge S. Cardoso, José Cordeiro, Joaquim Filipe. — 2007. — P. 246–251.

27 Diamantini C., Potena D., Storti E. Analytic processing in data lakes: A semantic query-driven discovery approach // Information Systems Frontiers. — 2024. — P. 1–19.

28 Harby A. A., Zulkernine F. Data lakehouse: A survey and experimental study // Information Systems. — 2025. — Vol. 127. — P. 102460.

29 Hai R., Quix C., Zhou C. Query rewriting for heterogeneous data lakes // Advances in Databases and Information Systems / Ed. by András Benczúr, Bernhard Thalheim, Tomáš Horváth. — Cham : Springer International Publishing, 2018. — P. 35–49.

30 Apache hadoop project. — 2024. — Access mode: http://hadoop.apache.org/.

31 Ibm infosphere information server deployment architectures / C. Ballard, T. Alon, N. Dronavalli et al. — ibm.com/redbooks, 2012.

32 Bar-Or A., Choudhary S. Transform xml using the datastage xml stage. — IBM developerWorks, 2011.

33 Clio: Schema mapping creation and data exchange / Ronald Fagin, Laura M. Haas, Mauricio A. Hernández et al. // Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos / Ed. by Alexander Borgida, Vinay K. Chaudhri, Paolo Giorgini, Eric S. K. Yu. — Vol. 5600 of Lecture Notes in Computer Science. — Springer, 2009. — P. 198–236. — Access mode: https://doi.org/10.1007/978-3-642-02463-4_12.

34 Data exchange: semantics and query answering / Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, Lucian Popa // Theor. Comput. Sci. — 2005. — Vol. 336, no. 1. — P. 89–124. — Access mode: https://doi.org/10.1016/j.tcs.2004.10.033.

35 Abrial J. R. The B-book: assigning programs to meanings. — New York : Cambridge University Press, 1996.

36 The first twenty-five years of industrial use of the b-method / Michael Butler, Philipp Körner, Sebastian Krings et al. // International Conference on Formal Methods for Industrial Critical Systems / Springer. — 2020. — P. 189–209.

37 N. White S. Matthews R. C. Formal verification: Will the seedling ever flower? // Philosophical Transactions A. — 2017. — Vol. 375. — P. 20150402. — Access mode: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2015.0402.

38 Kalinichenko L. A. Data models transformation method based on axiomatic data model extension // Fourth International Conference on Very Large Data Bases, September 13-15, 1978, West Berlin, Germany / Ed. by S. Bing Yao. — IEEE Computer Society, 1978. — P. 549–555.

39 Калиниченко Л. А. Методы и средства интеграции неоднородных баз данных. — М.: Наука, 1983.

40 Kalinichenko L. A. Methods and tools for equivalent data model mapping construction // Advances in Database Technology — EDBT ’90 / Ed. by François Bancilhon,

Constantino Thanos, Dennis Tsichritzis. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1990. — P. 92–119.

41 Kalinichenko L. A. Method for data models integration in the common paradigm // Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97), St.-Petersburg, Russia, September 2-5, 1997. Volume 1: Regular Papers. — Nevsky Dialect, 1997. — P. 275–284.

42 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. — М.: ИПИ РАН, 2007.

43 Калиниченко, Ступников С. А., Земцов Н. А. Синтез канонических моделей для интеграции неоднородных источников информации. — М.: ИПИ РАН, 2005.

44 Stupnikov S. A., Briukhov D. O., Skvortsov N. A. Анализ системного риска совместного кредитования над неоднородными коллекциями данных // Информатика и ее применения. — 2016. — Vol. 10, no. 1. — P. 23–33.

45 Ступников С. А. Автоматизация верификации уточнения при композиционном проектировании информационных систем и посредников // Системы и средства информатики. Спец. вып. Формальные методы и модели в композиционных инфраструктурах распределенных информационных систем. — 2005. — С. 96–119.

46 Ступников С. А. Отображение спецификаций, выраженных средствами ядра канонической модели, в язык amn // Системы и средства информатики. Спец. вып. Формальные методы и модели в композиционных инфраструктурах распределенных информационных систем. — 2005. — С. 69–95.

47 Калиниченко Л. А., Мартынов Д. О., Ступников С. А. Переписывание запросов на основе взглядов в типизированной среде посредников // Системы и средства информатики. Спец. вып. Формальные методы и модели в композиционных инфраструктурах распределенных информационных систем. — 2005. — С. 152–183.

48 Ступников С. А., Брюхов Д. О. Представление uml и ocl в канонической информационной модели // Системы и средства информатики. Спец. вып. Формальные методы и модели в композиционных инфраструктурах распределенных информационных систем. — 2005. — С. 120–129.

49 Ступников С. А. Формальная семантика ядра канонической объектной информационной модели // Системы и средства информатики. Спец. вып. Формальные методы и модели в композиционных инфраструктурах распределенных информационных систем. — 2005. — С. 40–68.

50 Конструирование канонических информационных моделей для интегрированных информационных систем / В. Н. Захаров, Л. А. Калиниченко, И. А. Соколов, С. А. Ступников // Информатика и ее применения. — 2007. — Т. 1, № 2. — С. 15–38.

51 Архитектура промежуточного слоя предметных посредников для решения задач над множеством интегрируемых неоднородных распределенных информационных ресурсов в гибридной грид-инфраструктуре виртуальных обсерваторий / Д. О. Брюхов, А. Е. Вовченко, В. Н. Захаров и др. // Информатика и ее применения. — 2008. — Т. 2, № 1. — С. 2–34.

52 Калиниченко Л. А., Ступников С. А. Унификация языков систем на правилах для обеспечения интероперабельности декларативных программ // Информатика и ее применения. — 2012. — Т. 6, № 2. — С. 59–76.

53 Kalinichenko L. A., Stupnikov S. A., Zakharov V. N. Extending information integration technologies for problem solving over heterogeneous information resources // Информатика и ее применения. — 2012. — Vol. 6, no. 1. — P. 70–77.

54 Скворцов Н. А., Ступников С. А., Калиниченко Л. А. Использование таксономий отношений класс-экземпляр в спецификациях предметных областей // Вестник ВГУ. Серия системный анализ и информационные технологии. — 2012. — Т. 1. — С. 157–167.

55 Conceptual declarative problem specification and solving in data intensive domains / Leonid Kalinichenko, Sergey Stupnikov, Alexey Vovchenko, Dmitry Kovalev // Информатика и ее применения. — 2013. — Vol. 7, no. 4. — P. 112–139.

56 Conceptual modeling of multi-dialect workflows / L. A. Kalinichenko, S. A. Stupnikov, A. E. Vovchenko, D. Y. Kovalev // Информатика и ее применения. — 2014. — Vol. 8, no. 4. — P. 110–124.

57 Ступников С. А., Вовченко А. Е. Методы построения отображений коллекций, представленных в нетрадиционных моделях данных, в интегрированное представление // Системы и средства информатики. — 2014. — Т. 24, № 4. — С. 29–44.

58 Методы унификации нетрадиционных моделей данных / С. А. Ступников, Н. А. Скворцов, В. И. Будзко и др. // Системы высокой доступности. — 2014. — Т. 10, № 1. — С. 18–39.

59 Модель метаданных для семантического поиска реализаций потоков работ, выраженных в виде правил / Н. А. Скворцов, А. Е. Вовченко, Л. А. Калиниченко и др. // Системы и средства информатики. — 2014. — Т. 24, № 4. — С. 4–28.

60 Ступников С. А. Отображение графовых моделей данных в каноническую модель в системах с интенсивным использованием данных // Системы высокой доступности. — 2014. — № 2. — С. 13–31.

61 Среда интеграции больших неоднородных коллекций данных / В. И. Будзко, С. А. Ступников, Л. А. Калиниченко и др. // Системы высокой доступности. — 2014. — № 3. — С. 3–19.

62 Брюхов Д. О., Скворцов Н. А., Ступников С. А. Методы интеграции разноструктурированных данных по арктической зоне для извлечения информации, нацеленной на поддержку поисково-спасательных операций // Системы высокой доступности. — 2017. — Т. 13, № 2. — С. 3–19.

63 Брюхов Д. О., Скворцов Н. А., Ступников С. А. Реализация методов интеграции данных в хранилище для поддержки поисково-спасательных операций в арктической зоне // Системы высокой доступности. — 2018. — Т. 14, № 4. — С. 36–54.

64 Ступников С. А., Калиниченко Л. А. Методы автоматизированного построения трансформаций информационных моделей // Системы и средства информатики. — 2009. — Т. 19. — С. 34–62.

65 Земцов Н. А., Ступников С. А. Формальное моделирование спецификаций процессов для композиционного проектирования потоков работ // Системы и средства информатики. — 2004. — Т. 14. — С. 186–198.

66 Брюхов Д. О., Ступников С. А. Логическая реляционная модель структур данных для решения задач в предметной области управления землепользованием // Информатика и её применения. — 2022. — Vol. 16, no. 4. — P. 93–98.

67 Skvortsov N. A., Stupnikov S. A. A semantic approach to workflow management and reuse for research problem solving // Data Intelligence. — 2022. — Vol. 4, no. 2. — P. 439 – 454.

68 Kalinichenko L. A., Stupnikov S. A., Zemtsov N. Extensible canonical process model synthesis applying formal interpretation // Advances in Databases and Information Systems, 9th East European Conference, ADBIS 2005, Tallinn, Estonia, September 12-15, 2005, Proceedings / Ed. by Johann Eder, Hele-Mai Haav, Ahto Kalja, Jaan Penjam. — Vol. 3631 of Lecture Notes in Computer Science. — Springer, 2005. — P. 183–198. — Access mode: https://doi.org/10.1007/11547686_14.

69 Stupnikov S. A., Kalinichenko L. A., Bressan S. Interactive discovery and composition of complex web services // Advances in Databases and Information Systems, 10th East European Conference, ADBIS 2006, Thessaloniki, Greece, September 3-7, 2006, Proceedings / Ed. by Yannis Manolopoulos, Jaroslav Pokorný, Timos K. Sellis. — Vol. 4152

of Lecture Notes in Computer Science. — Springer, 2006. — P. 216–231. — Access mode: https://doi.org/10.1007/11827252_18.

70 Kalinichenko L., Stupnikov S. Synthesis of the canonical models for database integration preserving semantics of the value inventive data models // Lecture Notes in Computer Science. — 2012. — Vol. 7503. — P. 223–239.

71 Multi-dialect workflows / L. Kalinichenko, S. Stupnikov, A. Vovchenko, D. Kovalev // Lecture Notes in Computer Science. — 2014. — Vol. 8716. — P. 352–365.

72 Stupnikov S. A. Query-driven verification of data integration in the rdf data model // Lobachevskii Journal of Mathematics. — 2023. — Vol. 44, no. 1. — P. 205–218.

73 Kalinichenko L. A., Martynov D. O., Stupnikov S. A. Query rewriting using views in a typed mediator environment // Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22-25, 2004, Proceeding / Ed. by Georg Gottlob, András A. Benczúr, János Demetrovics. — Vol. 3255 of Lecture Notes in Computer Science. — Springer, 2004. — P. 37–53. — Access mode: https://doi.org/10.1007/978-3-540-30204-9_3.

74 Stupnikov S. A. Formal semantics and verification of procedural sql programs implementing materialized data integration // Lobachevskii Journal of Mathematics. — 2025. — Vol. 46, no. 4. — P. 1511–1525.

75 Stupnikov S. A. Verification of global and local as view data integration in an object data model // Lobachevskii Journal of Mathematics. — 2025.

76 Stupnikov S. Semantics and verification of entity resolution and data fusion operations via transformation into a formal notation // Data Analytics and Management in Data Intensive Domains / Ed. by Leonid Kalinichenko, Sergei O. Kuznetsov, Yannis Manolopoulos. — Cham : Springer International Publishing, 2017. — P. 145–162.

77 Stupnikov S. Rule-based specification and implementation of multimodel data integration // Communications in Computer and Information Science. — 2018. — Vol. 822. — P. 198–212.

78 Stupnikov S., Kalinichenko L. Extensible unifying data model design for data integration in fair data infrastructures // Communications in Computer and Information Science. — 2019. — Vol. 1003. — P. 17–36.

79 Skvortsov N. A., Stupnikov S. A. Formalizing requirement specifications for problem solving in a research domain // Communications in Computer and Information Science. — 2019. — Vol. 1064. — P. 266–279.

80 Palagashvili A., Stupnikov S. Reversible mapping of relational and graph databases // Pattern Recognition and Image Analysis. — 2023. — Vol. 33, no. 2. — P. 113–121.

81 Sazontev V., Stupnikov S. An extensible approach to searching and selecting data sources for materialized big data integration in distributed computing environments // Pattern Recognition and Image Analysis. — 2023. — Vol. 33, no. 2. — P. 147–156.

82 Stupnikov S. Applying model-driven approach for data model unification // Modelling to Program / Ed. by Ajantha Dahanayake, Oscar Pastor, Bernhard Thalheim. — Cham : Springer International Publishing, 2021. — P. 212–232.

83 Tang W., Stupnikov S. A transformation of the rdf mapping language into a high-level data analysis language for execution in a distributed computing environment // Data Analytics and Management in Data Intensive Domains / Ed. by Alexander Sychev, Sergey Makhortov, Bernhard Thalheim. — Cham : Springer International Publishing, 2021. — P. 74–91.

84 Ступников С. А. Верификация интеграции данных в интегрированной системе баз данных по свойствам неорганических веществ и материалов // Ученые записки Казанского университета. Серия Физико-математические науки. — 2025. — Vol. 167, no. 2. — P. 367–383.

85 Kalinichenko L. A., Stupnikov S. A. Heterogeneous information model unification as a pre-requisite to resource schema mapping // Information Systems: People, Organizations, Institutions, and Technologies. Proceedings of the V Conference of the Italian Chapter of Association for Information Systems itAIS. — Physica-Verlag Heidelberg, 2010. — P. 373–380.

86 Kalinichenko L., Stupnikov S. Owl as yet another data model to be integrated // Proceedings II of the 15th East-European Conference on Advances in Databases and Information Systems. — Vol. 789 of CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2011. — P. 178–189.

87 Multilayer specifications in conceptual and ontological modeling / A. E. Vovchenko, V. N. Zakharov, L. A. Kalinichenko et al. // Proceedings of the 13th All-Russian Scientific Conference "Digital libraries: Advanced Methods and Technologies, Digital Collections". — Vol. 803 of CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2011. — P. 17–25.

88 Stupnikov S. Унификация модели данных, основанной на многомерных массивах, при интеграции неоднородных информационных ресурсов // Proceedings of the 14th All-Russian Scientific Conference "Digital libraries: Advanced Methods and Technologies, Digital Collections". — Т. 934 из CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2012. — С. 42–52.

89 Stupnikov S. Отображение графовой модели данных в каноническую объектно-фреймовую информационную модель при создании систем интеграции неоднородных

информационных ресурсов // Selected Papers of the 15th All-Russian Scientific Conference "Digital Libraries: Advanced Methods and Technologies, Digital Collections". — Т. 1108 из CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2013. — С. 85–94.

90 Метаданные о научных методах для обеспечения их повторного использования и воспроизводимости результатов / N. A. Skvortsov, D. O. Briukhov, L. A. Kalinichenko и др. // Selected Papers of the 15th All-Russian Scientific Conference "Digital Libraries: Advanced Methods and Technologies, Digital Collections". — Т. 1108 из CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2013. — С. 70–78.

91 Rule-based multi-dialect infrastructure for conceptual problem solving over heterogeneous distributed information resources / L. Kalinichenko, S. Stupnikov, A. Vovchenko, D. Kovalev // Advances in Intelligent Systems and Computing. — 2013. — Vol. 241. — P. 61–68.

92 Stupnikov S., Vovchenko A. Комбинированная виртуально-материализованная среда интеграции больших неоднородных коллекций данных // Selected Papers of XVI All-Russian Scientific Conference "Digital libraries: Advanced Methods and Technologies, Digital Collections". — Т. 1297 из CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2014. — С. 201–210.

93 Stupnikov S., Miloslavskaya N., Budzko V. Unification of graph data models for heterogeneous security information resources' integration // Proceedings of the International Conference on Open and Big Data OBD 2015 (joint with 3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015). — Institute of Electrical and Electronics Engineers Inc, 2015. — P. 457–464.

94 Извлечение информации из разноструктурированных данных и ее приведение к целевой схеме / Д. О. Брюхов, С. А. Ступников, Л. А. Калиниченко, А. Е. Вовченко // Selected Papers of the XVII International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2015). — Т. 1536 из CEUR Workshop Proceedings (CEUR-WS.org). — ceur-ws.org, 2015. — С. 81–90.

95 Stupnikov S. Formal semantics and verification of entity resolution and data fusion operations // Selected Papers of the XVII International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2016). — Vol. 1752 of CEUR Workshop Proceedings (CEUR-WS.org). — CEUR-WS.org, 2016. — P. 159–167.

96 Stupnikov S. Specification and implementation of multimodel data integration rules // Conference of 19th International Conference on Data Analytics and Management in Data Intensive Domains, DAMDID/RCDL 2017. — Vol. 2022. — CEUR-WS, 2017. — P. 197–205.

97 Stupnikov S., Kalinichenko L. Fair data based on extensible unifying data model development // Selected Papers of the XX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2018). — Vol. 2277 of CEUR Workshop Proceedings (CEUR-WS.org). — CEUR-WS, 2018. — P. 9–13.

98 Sazontev V., Stupnikov S. An extensible approach for materialized big data integration in distributed computation environments // 2019 Ivannikov Memorial Workshop (IVMEM) / Ed. by Сергей Петрович Прохоров. — IEEE, 2019. — P. 33–38.

99 Stupnikov S. A., Kalinichenko L. A., Dong J. S. Applying csp-like workflow process specifications for their refinement in AMN by pre-existing workflows // Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002, Bratislava, Slovakia, September 8-11, 2002, Proceedings, Volume 2: Research Communications / Ed. by Yannis Manolopoulos, Pavol Návrat. — Slovak University of Technology, Bratislava, 2002. — P. 206–216.

100 Ступников С. А., Калиниченко Л. А. Формирование базы метаинформации на основе спецификаций канонической модели // Проблемы программирования. — 2002. — № 1-2. — С. 301–308.

101 Briukhov D. O., Kalinichenko L. A., Stupnikov S. A. Compositional approach for heterogeneous sources registration at a subject mediator // Emerging Database Research In East Europe. Proceedings of the pre-conference workshop of VLDB 2003, Berlin, Germany. — 2003. — P. 5–11.

102 Kalinichenko L. A., Stupnikov S. A. Constructing of mappings of heterogeneous information models into the canonical models of integrated information systems // Advances in Databases and Information Systems, Proceedings of the 12th East European Conference, ADBIS 2008, September 5-9, 2008, Pori, Finland / Ed. by Paolo Atzeni, Albertas Caplinskas, Hannu Jaakkola. — Tampere University of Technology. Pori. Publication, 2008. — P. 106–122.

103 Скворцов Н. А., Ступников С. А. Использование онтологии верхнего уровня для отображения информационных моделей // Труды Десятой Всероссийской научной конференции Электронные библиотеки: перспективные методы и технологии, электронные коллекции RCDL. — 2008. — С. 122–127.

104 Stupnikov S., Kalinichenko L. Methods for semi-automatic construction of information models transformations // Proc. of the 13th East-European Conference Advances in Databases and Information Systems, workshop Model – Driven Architecture: Foundations, Practices and Implications (MDA). — Riga Technical University Riga, 2009. — P. 432–440.

105 Калиниченко Л. А., Ступников С. А. Каркас логических диалектов rif: анализ и демонстрация применения // Труды Второго симпозиума Онтологическое моделирование: состояние, направления исследований и применения (ассоциирован с XII Всероссийской научной конференцией RCDL'2010). — ИПИ РАН М, 2010. — С. 123–150.

106 Ступников С. А., Скворцов Н. А. Взаимное отображение канонической информационной модели и языка owl 2 // Труды 12-й Всероссийской научной конференции "Электронные библиотеки: перспективные методы и технологии, электронные коллекции" RCDL. — Казанский университет Казань, 2010. — С. 392–398.

107 Ступников С. А., Скворцов Н. А. Семантическая трансформация канонической информационной модели в формальный язык спецификаций для верификации уточнения // Труды 12-й Всероссийской научной конференции "Электронные библиотеки: перспективные методы и технологии, электронные коллекции" RCDL. — Казанский университет Казань, 2010. — С. 383–391.

108 Калиниченко Л. А., Ступников С. А. Анализ мотивации, целей и подходов проекта унификации языков на правилах // Труды Второго симпозиума Онтологическое моделирование: состояние, направления исследований и применения (ассоциирован с XII Всероссийской научной конференцией RCDL'2010). — ИПИ РАН М, 2011. — С. 79–122.

109 Ступников С. А., Калиниченко Л. А. Формальная семантика канонической информационной модели в композиционной инфраструктуре распределенных информационных систем // Проблемы и методы информатики. II Научная сессия ИПИ РАН: Тезисы докладов / Под ред. И.А. Соколова. — ИПИ РАН Москва, 2005. — С. 151–153.

110 Ступников С. А. Отображение канонической модели спецификаций в формальную нотацию для моделирования уточняющих спецификаций // Труды XXIV Конференции молодых учёных механико-математического факультета МГУ им. М.В. Ломоносова / Под ред. Дмитрий Владимирович Георгиевский. — Издательство Центра прикладных исследований механико-математического факультета МГУ им. М.В.Ломоносова, Москва, 2002. — С. 169–171.

111 Земцов, Ступников С. А., Калиниченко Л. А. Синтез канонической процессной модели для интероперабельных инфраструктур информационных систем // Проблемы и методы информатики. II Научная сессия ИПИ РАН: Тезисы докладов / Под ред. И.А. Соколова. — ИПИ РАН Москва, 2005. — С. 153–156.

112 Sundaresan S., Hu G. Schema integration of distributed databases using hyper-graph data model // IRI — 2005 IEEE International Conference on Information Reuse and Integration, Conf, 2005. — 2005. — P. 548–553.

113 Stonebraker M., Pavlo A. What goes around comes around... and around... // ACM Sigmod Record. — 2024. — Vol. 53, no. 2. — P. 21–37.

114 Hainaut J.-L. Hierarchical data model // Encyclopedia of Database Systems. — Springer, 2018. — P. 1689–1695.

115 Hainaut J.-L. Network data model // Encyclopedia of Database Systems. — Springer, 2018. — P. 2490–2497.

116 Embley D. W. Relational model // Encyclopedia of database systems. — Springer, 2018. — P. 3149–3154.

117 Song I.-Y., Chen P. P. Entity relationship model // Encyclopedia of Database Systems. — Springer, 2018. — P. 1315–1322.

118 Integration Definition for Information Modeling (IDEF1X). FIPS PUB 184. — National Institute of Standards and Technology, 1993.

119 Stonebraker M., Brown P., Moore D. Object-Relational Dbmss: Tracking the Next Great Wave. — Morgan Kaufmann Publishers, 1996.

120 Rys M. Xml schema // Encyclopedia of Database Systems. — Springer, 2018. — P. 4789–4791.

121 Fitzpatrick B. Distributed caching with memcached // Linux journal. — 2004. — Vol. 2004, no. 124. — P. 5.

122 Dynamo: Amazon's highly available key-value store / Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al. // ACM SIGOPS operating systems review. — 2007. — Vol. 41, no. 6. — P. 205–220.

123 Bigtable: A distributed storage system for structured data / Fay Chang, Jeffrey Dean, Sanjay Ghemawat et al. // ACM Transactions on Computer Systems (TOCS). — 2008. — Vol. 26, no. 2. — P. 1–26.

124 Gruber T. Ontology // Encyclopedia of Database Systems / Ed. by Ling Liu, M. Tamer Özsu. — Springer New York, 2018. — P. 2574–2576.

125 Borgida A. Description logics // Encyclopedia of Database Systems. — Springer, 2018. — P. 1–5.

126 Palmer N. Workflow model // Encyclopedia of Database Systems. — Springer, 2018. — P. 4716–4716.

127 Dumas M. Business process modeling // Encyclopedia of database systems. — Springer, 2018. — P. 374–382.

128 Knowledge Graphs - Methodology, Tools and Selected Use Cases / Dieter Fensel, Umutcan Simsek, Kevin Angele et al. — Springer, 2020.

129 Brickley D., Guha R. V. Rdf schema 1.1. w3c recommendation. — 2014. — Access mode: https://www.w3.org/TR/rdf-schema/.

130 Harris S., Seaborn A. Sparql 1.1 query language. w3c recommendation. — 2013. — Access mode: https://www.w3.org/TR/sparql11-query/.

131 Gruber T. R. Toward principles for the design of ontologies used for knowledge sharing // J. Human-Computer Studies. — 1995. — Vol. 43, no. 5-6. — P. 907–928.

132 Ontology-based data access and integration / Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo et al. // Encyclopedia of database systems. — Springer, 2018. — P. 2590–2596.

133 Ontologies and Databases: The DL-Lite Approach / Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo et al. // Reasoning Web. Semantic Technologies for Information Systems: 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures / Ed. by Sergio Tessaris, Enrico Franconi, Thomas Eiter et al. — Springer Berlin Heidelberg, 2009. — P. 255–356.

134 Ontop: Answering sparql queries over relational databases / Óscar Corcho, Diego Calvanese, Benjamin Cogrel et al. // Semant. Web. — 2017. — jan. — Vol. 8, no. 3. — P. 471–487. — Access mode: https://doi.org/10.3233/SW-160217.

135 Harris S., Seaborn A. Owl 2 web ontology language profiles. w3c recommendation. — 2012. — Access mode: https://www.w3.org/TR/owl2-profiles/.

136 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 / Ed. by Albertas Caplinskas, Johann Eder. — Springer Berlin Heidelberg, 2001. — P. 70–83.

137 Horrocks I., Sattler U., S. T. Practical reasoning for very expressive description logics // Logic Journal of the IGPL. — 2000. — Vol. 8. — P. 239–264.

138 Reasoners and rule engines: Jena inference support. — 2023. — Access mode: https://jena.apache.org/documentation/inference/index.html.

139 Kifer M. Rule interchange format: The framework // International Conference on Web Reasoning and Rule Systems / Springer. — 2008. — P. 1–11.

140 Kifer M., Boley H. Rif overview (second edition). w3c recommendation. — 2013. — Access mode: https://www.w3.org/TR/rif-overview/.

141 Boley H., Kifer M. Rif basic logic dialect. w3c recommendation. — 2013. — Access mode: https://www.w3.org/TR/rif-bld/.

142 De Sainte Marie C., Hallmark G., Paschke A. Rif production rule dialect (second edition). w3c recommendation. — 2013. — Access mode: http://www.w3.org/TR/rif-prd/.

143 Polleres A., Boley H., Kifer M. Rif datatypes and built-ins 1.0. w3c recommendation. — 2013. — Access mode: http://www.w3.org/TR/rif-dtb/.

144 Boley H., Kifer M. Rif framework for logic dialects. w3c recommendation. — 2013. — Access mode: http://www.w3.org/TR/rif-fld/.

145 Ступников С. А. Верифицируемое отображение модели данных, основанной на многомерных массивах, в объектную модель данных // Информатика и её применения. — 2013. — Vol. 7, no. 3. — P. 22–34.

146 Zalipynis R. A. R. Array dbms: Past, present, and (near) future // Proceedings of the VLDB Endowment. — 2021. — Vol. 14, no. 12. — P. 3186–3189.

147 Rodriges Zalipynis R. A. Array (Tensor) DBMS: Theoretical Foundations, Software, and Applications. — Dissertation for the purpose of obtaining academic degree Doctor of Science in Computer Science. HSE University, 2024.

148 Pedersen T., Jensen C. Multidimensional database technology // Computer. — 2001. — Vol. 34, no. 12. — P. 40–46.

149 Vassiliadis P., Sellis T. A survey of logical models for olap databases // SIGMOD Rec. — 1999. — Vol. 28, no. 4. — P. 64–69.

150 Libkin L., Machlin R., Wong L. A query language for multidimensional arrays: Design, implementation, and optimization techniques // Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. — New York, NY, USA : Association for Computing Machinery, 1996. — P. 228–239.

151 Baumann P. A database array algebra for spatio-temporal data and beyond // Next Generation Information Technologies and Systems / Ed. by Ron Y. Pinter, Shalom Tsur. — Springer Berlin Heidelberg, 1999. — P. 76–93.

152 Rasdaman (raster data manager) for multi-dimensional datacube management and analytics. — 2026. — Access mode: http://www.rasdaman.org/.

153 Brown P. G. Overview of scidb: Large scale array storage, processing and analysis // Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. — Association for Computing Machinery, 2010. — P. 963–968.

154 Large synoptic survey telescope. — 2023. — Access mode: http://www.lsst.org.

155 Becla J., Lim K. Report from the first workshop on extremely large databases // Data Science Journal. — 2008. — Vol. 7. — P. 1–13.

156 Scidb documentation 22.5. — 2022. — Access mode: https://paradigm4.atlassian.net/wiki/spaces/scidb/overview.

157 Sciql, a query language for science applications / M. Kersten, Y. Zhang, M. Ivanova, N. Nes. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 1–12.

158 Scidb: A database management system for applications with complex analytics / Michael Stonebraker, Paul Brown, Donghui Zhang, Jacek Becla // Computing in Science & Engineering. — 2013. — Vol. 15, no. 3. — P. 54–62.

159 Zalipynis R. A. R. Chronosdb: distributed, file based, geospatial array dbms // Proceedings of the VLDB Endowment. — 2018. — Vol. 11, no. 10. — P. 1247–1261.

160 Wegner P. Interaction as a basis for empirical computer science // ACM Comput. Surv. — 1995. — Vol. 27, no. 1. — P. 45–48.

161 Wegner P. Why interaction is more powerful than algorithms // Commun. ACM. — 1997. — Vol. 40, no. 5. — P. 80–91.

162 Хоар Ч. Взаимодействующие последовательные процессы. — М.: Мир, 1989.

163 Milner R. Communication and Concurrency. — L.: Prentice Hall, 1989.

164 Modern Business Process Automation: YAWL and its Support Environment / Arthur H. M. Hofstede, Wil M. P. Aalst, Michael Adams, Nick Russell. — Berlin-Heidelberg: Springer, 2010.

165 Angles R. A comparison of current graph database models // 2012 IEEE 28th International Conference on Data Engineering Workshops. — 2012. — P. 171–177.

166 Robinson I. Webber J. E. E. Graph Databases. — O'Reilly Media, 2013.

167 Iordanov B. Hypergraphdb: A generalized graph database // Web-Age Information Management / Ed. by Heng Tao Shen, Jian Pei, M. Tamer Özsu et al. — Springer Berlin Heidelberg, 2010. — P. 25–36.

168 Pregel: A system for large-scale graph processing / Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik et al. // Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. — SIGMOD '10. — New York, NY, USA : Association for Computing Machinery, 2010. — P. 135–146.

169 Apache giraph project. — 2020. — Access mode: http://giraph.apache.org/.

170 Shao B., Wang H., Li Y. Trinity: A distributed graph engine on a memory cloud // Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. — SIGMOD '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 505–516.

171 Kyrola A., Blelloch G., Guestrin C. Graphchi: Large-scale graph computation on just a pc // Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. — OSDI'12. — USA : USENIX Association, 2012. — P. 31–46.

172 Shao B., Wang H., Xiao Y. Managing and mining large graphs: Systems and implementations // Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. — SIGMOD ’12. — New York, NY, USA : Association for Computing Machinery, 2012. — P. 589–592.

173 Efficient subgraph matching on billion node graphs / Zhao Sun, Hongzhi Wang, Haixun Wang et al. // Proc. VLDB Endow. — 2012. — Vol. 5, no. 9. — P. 788–799.

174 Neo4j graph database. — 2023. — Access mode: http://www.neo4j.org/.

175 Sparksee: Out-of-core graph database for edge computing. — 2023. — Access mode: https://www.sparsity-technologies.com/#sparksee.

176 Horton+: A distributed system for processing declarative reachability queries over partitioned graphs / Mohamed Sarwat, Sameh Elnikety, Yuxiong He, Mohamed F. Mokbel. — 2013. — Vol. 6, no. 14.

177 Orleans: Cloud computing for everyone / Sergey Bykov, Alan Geller, Gabriel Kliot et al. — New York, NY, USA : Association for Computing Machinery, 2011.

178 Titan project. — 2022. — Access mode: https://github.com/thinkaurelius/titan/wiki.

179 Kolomicenko V., Svoboda M., Mlýnková I. H. Experimental comparison of graph databases // Proceedings of International Conference on Information Integration and Web-Based Applications & Services. — IIWAS ’13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 115–124.

180 Kalinichenko L. Compositional specification calculus for information systems development // Advances in Databases and Information Systems / Ed. by Johann Eder, Ivan Rozman, Tatjana Welzer. — Springer Berlin Heidelberg, 1999. — P. 317–331.

181 Neznanov A. A., Parinov A. A. Unified external data access implementation in formal concept analysis research toolbox. // CLA 2016: Proceedings of the Thirteenth International Conference on Concept Lattices and Their Applications. — 2016. — P. 285–297. — Access mode: https://ceur-ws.org/Vol-1624/paper22.pdf.

182 ISO 10303-1:2024 Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles. — International Organization for Standardization, 2024.

183 Вовченко А. Е. Рассредоточенная реализация приложений в среде предметных посредников. Диссертация на соискание ученой степени кандидата технических наук по специальности 05.13.11. — Москва: ИПИ РАН, 2012.

184 Katsis Y., Papakonstantinou Y. View-based data integration // Encyclopedia of database systems. — Springer, 2018. — P. 4452–4461.

185 Сазонтьев В. В., Ступников С. А., Захаров В. Н. Расширяемый подход к слиянию данных в распределенных вычислительных средах // Информатика и её применения. — 2023. — Vol. 17, no. 4. — P. 42–47.

186 Dong X. L., Srivastava D. Big data integration. — Morgan & Claypool Publishers, 2015.

187 An overview of end-to-end entity resolution for big data / Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas et al. // ACM Computing Surveys (CSUR). — 2020. — Vol. 53, no. 6. — P. 1–42.

188 Canalle G. K., Salgado A. C., Loscio B. F. A survey on data fusion: what for? in what form? what is next? // Journal of Intelligent Information Systems. — 2021. — Vol. 57, no. 1. — P. 25–50.

189 Bleiholder J., Naumann F. Data fusion // ACM Comput. Surv. — 2009. — jan. — Vol. 41, no. 1. — Access mode: https://doi.org/10.1145/1456650.1456651.

190 Dong X. L., Rekatsinas T. Data integration and machine learning: A natural synergy // Proceedings of the 2018 international conference on management of data. — 2018. — P. 1645–1650.

191 The ASF+SDF meta-environment: A component-based language development environment / Mark van den Brand, Arie van Deursen, Jan Heering, et. al. // Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings / Ed. by Reinhard Wilhelm. — Vol. 2027 of Lecture Notes in Computer Science. — Springer, 2001. — P. 365–370.

192 Atl — a model transformation technology. — 2025. — Access mode: https://eclipse.org/atl/.

193 Rodrigues da Silva A. Model-driven engineering: A survey supported by the unified conceptual model // Computer Languages, Systems & Structures. — 2015. — Vol. 43. — P. 139–155. — Access mode: https://www.sciencedirect.com/science/article/pii/S1477842415000408.

194 Object management group model driven architecture (mda). mda guide rev. 2.0. — OMG Document ormsc/2014-06-01, 2014.

195 Object constraint language specification version 2.4. — 2014. — Access mode: https://www.omg.org/spec/OCL/.

196 EMF: Eclipse Modeling Framework, 2nd Edition. / D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. — Addison-Wesley Professional, 2008.

197 Meta object facility (mof) 2.0 query/view/transformation specification. version 1.3. omg document number: formal/2016-06-03. — 2016. — Access mode: http://www.omg.org/spec/QVT/1.3.

198 Hehner E. C. R. A Practical Theory of Programming. — N.Y.: Springer-Verlag, 1993.

199 Morgan C. Programming from Specifications. — Hempstead: Prentice Hall, 1990.

200 Back J., Von Right J. Refinement Calculus. — N.Y.: Springer-Verlag, 1998.

201 Spivey J. The Z Notation: A Reference Manual. — L. Prentice Hall, 1992.

202 Brucker A. D., Rittinger F., Wolff B. Hol-z 2.0 // Journal of Universal Computer Science. — 2003. — Vol. 9, no. 2. — P. 152–172.

203 Validated Designs for Object-oriented Systems / J. Fitzgerald, P.G. Larsen, P. Mukherjee et al. — Springer-Verlag, 2005.

204 Group T. R. L. The RAISE specification language. — USA : Prentice-Hall, Inc., 1993.

205 Group R. L. Raise Method Manual. — USA : Prentice-Hall, Inc., 1993.

206 Atelier b, the industrial tool to efficiently deploy the b method. — 2022. — Access mode: http://www.atelierb.eu/.

207 ГОСТ 19.101—2024. Единая система программной документации. Виды программ и программных документов. — Российский институт стандартизации, 2024.

208 ГОСТ Р ИСО 15926-1—2008. Промышленные аватоматизированные системы и интеграция. Интеграция данных жизненного цикла для перерабатывающих предприятий, включая нефтяные и газовые производственные предприятия. Часть 1: Обзор и основополагающие принципы. — Москва: Стандартинформ, 2010.

209 ГОСТ Р 58539—2019. Информационные технологии. Концепция интероперабельности на основе метамоделей. Часть 1: Основные положения. — Москва: Стандартинформ, 2015.

210 ГОСТ Р 550062—2021. Информационные технологии. Интероперабельность. Основные положения. — Российский институт стандартизации, 2021.

211 ГОСТ Р 59797—2021. Информационные технологии. Сложные системы. Интероперабельность. Основные положения. — Российский институт стандартизации, 2021.

212 ГОСТ Р ИСО 15926-2—2010. Промышленные аватоматизированные системы и интеграция. Интеграция данных жизненного цикла для перерабатывающих предприятий, включая нефтяные и газовые производственные предприятия. Часть 2: Модель данных. — Москва: Стандартинформ, 2010.

213 ГОСТ Р ИСО 11354-2—2016. Усовершенствованные автоматизированные технологии и их применение. Требования к установлению интероперабельности процессов промышленных предприятий. Часть 2: Модель зрелости для оценки интероперабельности предприятий. — Москва: Стандартинформ, 2017.

214 ГОСТ Р 59352—2021. Системная инженерия. Защита информации в процессе верификации системы. — Москва: Стандартинформ, 2021.

215 Portisch J., Hladik M., Paulheim H. Background knowledge in schema matching: A survey // Semantic Web. — 2022.

216 Ionescu A. Reproducing state-of-the-art schema matching algorithms // Delft University of Technology. — 2020. — Access mode: https://repository.tudelft.nl/record/uuid:9f8056e6-cfdf-4240-99e3-5f45947d1fa7.

217 Chugh T., Zambre D. Astra: Automatic schema matching using machine translation // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track. — 2024. — P. 1237–1244.

218 Russell N., van der Aalst W. M. P., ter Hofstede A. H. M. Workflow Patterns: The Definitive Guide. — Cambridge, MA, USA : MIT Press, 2016.

219 Emftext concrete syntax mapper. — 2023. — Access mode: https://github.com/DevBoost/EMFText.

220 Survey on complex ontology matching / Elodie Thiéblin, Ollivier Haemmerlé, Nathalie Hernandez, Cassia Trojahn // Semantic Web. — 2020. — Vol. 11, no. 4. — P. 689–727.

221 Metamodel matching for automatic model transformation generation / Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, Clémentine Nebut // Model Driven Engineering Languages and Systems / Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel et al. — Springer Berlin Heidelberg, 2008. — P. 326–340.

222 Roser S., Bauer B. Automatic Generation and Evolution of Model Transformations Using Ontology Engineering Space // Journal on Data Semantics XI / Ed. by Stefano Spaccapietra, Jeff Z. Pan, Philippe Thiran et al. — Springer Berlin Heidelberg, 2008. — P. 32–64. — Access mode: https://doi.org/10.1007/978-3-540-92148-6_2.

223 Калиниченко Л. А., Скворцов Н. А. Реверсивное онтологическое моделирование при унифицированном представлении различных онтологических моделей источников информации в предметном посреднике // Системы и средства информатики: Спец. вып. Формальные методы и модели в композиционных инфраструктурах распределенных информационных систем. — 2005. — Vol. 27, no. 2. — P. 184–212.

224 Jouault F., Bézivin J. Km3: A dsl for metamodel specification // Formal Methods for Open Object-Based Distributed Systems / Ed. by Roberto Gorrieri, Heike Wehrheim. — Springer Berlin Heidelberg, 2006. — P. 171–185.

225 Cypher: An evolving query language for property graphs / Nadime Francis, Alastair Green, Paolo Guagliardo et al. — New York, NY, USA : Association for Computing Machinery, 2018.

226 Angles R., Gutierrez C. Survey of graph database models // ACM Comput. Surv. — 2008. — Vol. 40, no. 1.

227 Wood P. T. Query languages for graph databases // SIGMOD Rec. — 2012. — Vol. 41, no. 1. — P. 50–60.

228 Semantic sensor network ontology. w3c recommendation / R. Atkinson, R. Garcia-Castro, J. Lieberman, C. Stadler. — 2017. — Access mode: https://www.w3.org/TR/vocab-ssn/.

229 Global soil information system (glosis) ontology. sieusoil project / Palma R., Janiak B., Reznik T. et al. — 2020. — Access mode: http://w3id.org/glosis/model.

230 Owl 2 web ontology language direct semantics (second edition). w3c recommendation / B. Motik, P. F. Patel-Schneider, B. Parsia, B. C. Grau. — 2012. — Access mode: https://www.w3.org/TR/owl2-direct-semantics/.

231 Horrocks I., Kutz O., Sattler U. The even more irresistible sroiq // Proc. of the Tenth International Conference Principles of Knowledge Representation and Reasoning. — Menlo Park, California: AAAI Press, 2006. — P. 57–67.

232 Model-independent schema translation / Paolo Atzeni, Paolo Cappellari, Riccardo Torlone et al. // The VLDB Journal. — 2008. — nov. — Vol. 17, no. 6. — P. 1347–1370. — Access mode: https://doi.org/10.1007/s00778-008-0105-2.

233 MISM: A Platform for Model-Independent Solutions to Model Management Problems / Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, Giorgio Gianforme // Journal on Data Semantics XIV / Ed. by Stefano Spaccapietra, Lois Delcambre. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. — P. 133–161.

234 A runtime approach to model-generic translation of schema and data / Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti et al. // Information Systems. — 2012. — Vol. 37, no. 3. — P. 269–287. — Access mode: https://www.sciencedirect.com/science/article/pii/S0306437911001542.

235 Polystore and Tensor Data Model for Logical Data Independence and Impedance Mismatch in Big Data Analytics / Éric Leclercq, Annabelle Gillet, Thierry Grison, Marinette Savonnet // Transactions on Large-Scale Data- and Knowledge-Centered Systems

XLII / Ed. by Abdelkader Hameurlain, Roland Wagner. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2019. — P. 51–90.

236 Candel C. J. F., Sevilla Ruiz D., Garcia-Molina J. J. A unified metamodel for nosql and relational databases // Information Systems. — 2022. — Vol. 104. — P. 101898.

237 Manukyan M., Gevorgyan G. An approach to information integration based on the amn formalism // First Workshop on Programming the Semantic Web, in conjunction with the 11th International Semantic Web Conference. — 2012.

238 Manukyan M. Canonical model: Construction principles // Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services. — 2014. — P. 320–329.

239 Manukyan M. G. An ontology approach to data integration. // DAMDID/RCDL (Supplementary Proceedings). — 2020. — P. 33–47. — Access mode: https://ceur-ws.org/Vol-2790/paper05.pdf.

240 Manukyan M. G. On a conceptual data model with orientation to data integration. // DAMDID/RCDL (Supplementary Proceedings). — 2021. — P. 39–53. — Access mode: https://ceur-ws.org/Vol-3036/paper03.pdf.

241 Manukyan M. G. On an extensible conceptual data model by a non-formal example // New Trends in Database and Information Systems / Ed. by Silvia Chiusano, Tania Cerquitelli, Robert Wrembel et al. — Cham : Springer International Publishing, 2022. — P. 3–13.

242 Del Fabro M. D., Valduriez P. Semi-automatic model integration using matching transformations and weaving models // Proceedings of the 2007 ACM Symposium on Applied Computing. — SAC '07. — New York, NY, USA : Association for Computing Machinery, 2007. — P. 963–970. — Access mode: https://doi.org/10.1145/1244002.1244215.

243 Towards model transformation generation by-example / Manuel Wimmer, Michael Strommer, Horst Kargl, Gerhard Kramler // 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07). — 2007. — P. 285b–285b.

244 Diskin Z. Algebraic models for bidirectional model synchronization // Model Driven Engineering Languages and Systems / Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel et al. — Springer Berlin Heidelberg, 2008. — P. 21–36.

245 Bohannon A., Pierce B. C., Vaughan J. A. Relational lenses: A language for updatable views // Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. — PODS '06. — New York, NY, USA : Association for Computing Machinery, 2006. — P. 338–347. — Access mode: https://doi.org/10.1145/1142351.1142399.

246 Applying mde to the (semi-)automatic development of model transformations / Verónica Andrea Bollati, Juan Manuel Vara, Álvaro Jiménez, Esperanza Marcos // Inf. Softw. Technol. — 2013. — apr. — Vol. 55, no. 4. — P. 699–718. — Access mode: https://doi.org/10.1016/j.infsof.2012.11.004.

247 Lano K., Kolahdouz-Rahimi S., Fang S. Model transformation development using automated requirements analysis, metamodel matching, and transformation by example // ACM Transactions on Software Engineering and Methodology (TOSEM). — 2021. — Vol. 31, no. 2. — P. 1–71.

248 Di Ruscio D. Specification of Model Transformation and Weaving in Model Driven Engineering. PhD Thesis in Computer Science. — Dipartimento di Informatica Universit‘a di L’Aquila, 2007.

249 Stenzel K., Moebius N., Reif W. Formal verification of qvt transformations for code generation // Software & Systems Modeling. — 2015. — May. — Vol. 14, no. 2. — P. 981–1002. — Access mode: https://doi.org/10.1007/s10270-013-0351-7.

250 Verification of atl transformations using transformation models and model finders / Fabian Büttner, Marina Egea, Jordi Cabot, Martin Gogolla // Formal Methods and Software Engineering / Ed. by Toshiaki Aoki, Kenji Taguchi. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. — P. 198–213.

251 Lano K., Rahimi S. K., Clark T. Language-independent model transformation verification // Proceedings of the Third International Workshop on Verification of Model Transformations co-located with Software Technologies: Applications and Foundations, VOLT@STAF 2014, York, UK, July 21, 2014 / Ed. by Moussa Amrani, Eugene Syriani, Manuel Wimmer. — Vol. 1325 of CEUR Workshop Proceedings. — CEUR-WS.org, 2014. — P. 36–45. — Access mode: https://ceur-ws.org/Vol-1325/paper5.pdf.

252 The middleware architecture of the subject mediators for problem solving over a set of integrated heterogeneous distributed information resources in the hybrid grid-infrastructure of virtual observatories / D. O. Bryukhov, A. E. Vovchenko, V. N. Zakharov et al. // Inform. Primen. — 2008. — Vol. 2. — P. 2–34.

253 Briukhov D. O., Kalinichenko L. A., Martynov D. O. Source registration and query rewriting applying lav/glav techniques in a typed subject mediator // Proc. of the Ninth Russian Conference on Digital Libraries RCDL’2007. — 2007. — P. 253–262.

254 Interoperability and fairness through a novel combination of web technologies / Mark D. Wilkinson, Ruben Verborgh, Bonino da Silva Santos, et al. // PeerJ Computer Science. — 2017. — Vol. 3.

255 Jacobsen A., M. A. R., Juty N. Fair principles: Interpretations and implementation considerations // http://dx.doi.org/10.1162/dint_r_00024Data Intelligence. — 2017. — Vol. 2, no. 1-2. — P. 10–29.

256 Obi-wan: Ontology-based rdf integration of heterogeneous data / Maxime Buron, François Goasdoué, Ioana Manolescu, Marie-Laure Mugnier // Proc. VLDB Endow. — 2020. — Vol. 13, no. 12. — P. 2933–2936. — Access mode: https://doi.org/10.14778/3415478.3415512.

257 Hayes P. J., Patel-Schneider P. F. Rdf 1.1 semantics. w3c recommendation. — 2014. — Access mode: https://www.w3.org/TR/rdf11-mt/.

258 Schneider M. Owl 2 web ontology language rdf-based semantics (second edition). w3c recommendation. — 2012. — Access mode: https://www.w3.org/TR/owl2-rdf-based-semantics/.

259 Tennent R. The denotational semantics of programming languages // CACM. — 1976. — Vol. 19. — P. 437–453.

260 Allison L. A Practical Introduction to Denotational Semantics. — Cambridge: Cambridge University Press, 1986.

261 Schmidt D. Denotational Semantics: A Methodology for Language Development. — Dubuque: William C. Brown Pub., 1988.

262 Stoy J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. — Cambridge: MIT Press, 1977.

263 Winskel G. The Formal Semantics of Programming Languages. — Cambridge: MIT Press, 1993.

264 Bruel J., France R. Transforming uml models to formal specifications // UML'98: Beyond the Notation: Proc. of 1st International Workshop. — 1999.

265 France R., Grant E., Bruel J. UMLtranZ: An UML-based rigorous requirements modeling technique. — Fort Collins: Colorado State University, 2000.

266 Kim S., Carrington D. A formal mapping between uml models and object-z specifications // ZB2000 Formal Specification and Development in Z and B: Proc. First International Conference of B and Z users. — 2000. — P. 2–21.

267 Kim S., Carrington D. A formal denotational semantics of uml in object-z // L'OBJET. — 2001. — Vol. 7.

268 Roe D., Broda K., Russo A. Mapping uml models incorporating ocl constraints into object-z // Imperial College Technical Reports. — 2003. — Vol. 9.

269 Lano K., Bicarregui J. Formalising the uml in structured temporal theories // Technical Report TUM-I9813: Proceedings of the ECOOP 98 Workshop on Behavioural Semantics.

270 Lano K., Bicarregui J. Uml refinement and abstraction transformations // Proc. ROOM 2 Workshop.

271 Lano K., Bicarregui J., Evans A. Structured axiomatic semantics for uml models // Rigorous Object-Oriented Methods: Proc. of the Conference.

272 Lano K. Logical specification of reactive and real-time systems // Journal of Logic and Computation. — 1998. — Vol. 8. — P. 679–711. — Access mode: http://ewic.bcs.org/conferences/2000/objectmethods/papers/ paper5.htm.

273 Baar T., Beckert B., Schmitt P. An extension of dynamic logic for modeling ocl's pre operator // Perspectives of System Informatics: Proc. Fourth Andrei Ershov International Conference. — 2001.

274 Beckert B., Keller U., Schmitt P. Translating the object constraint language into first-order predicate logic // Proc. VERIFY: Workshop at Federated Logic Conferences. — 2002.

275 Facon P., Laleau R., Nguyen H. Mapping object diagrams into b specifications // In Proc. Methods Integration Workshop. — 1996. — Access mode: http://ewic.bcs.org/conferences/1996/methodsintegration/papers/ paper6.pdf.

276 Meyer E., Souquieres J. A systematic approach to transform omt diagrams to a b specification // Proc. FM'99. — 1999. — P. 875–895.

277 Lano K. The B Language and Method: A Guide to Practical Formal Development. — Springer-Verlag, 1996.

278 Butler M., Snook C. Verifying dynamic properties of uml models by translation to the b language and toolkit // Dynamic Behaviour in UML Models: Semantic Questions: Proc. of the UML 2000 Workshop. — 2000. — Access mode: http://www.disi.unige.it/person/ReggioG/UMLWORKSHOP/Snook.pdf.

279 Snook C., Butler M. Verifying dynamic properties of uml models by translation to the b language and toolkit // Tool-Supported Use of UML for Constructing B Specifications. Technical Report DSSE-TR-2002-3. — 2002.

280 Ledang H., Souquieres J. Modeling class operations in b: application to uml behavioral diagrams // Proc. of ASE 2001.

281 Ledang H., Souquieres J. Integration of uml and b specification techniques: Systematic transformation from ocl expressions into b // Proc. of APSEC 2002. — 2002. — P. 495–506.

282 Ledang H., Souquieres J., Charles S. Argouml+b : Un outil de transformation systematique de specifications uml vers b // Proc. of AFADL'2003. — 2003. — P. 3–18.

283 Conen W., Klapsing R. A logical interpretation of rdf. — 2000. — Vol. 5. — P. 013.

284 Fikes R., McGuinness D. An axiomatic semantics for rdf, rdf-s, and daml+oil. w3c note. — 2001. — Access mode: https://www.w3.org/TR/daml+oil-axioms/.

285 Mossakowski T., Maeder C., Lüttich K. The heterogeneous tool set, hets // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by Orna Grumberg, Michael Huth. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. — P. 519–522.

286 de Bruijn J., Franconi E., Tessaris S. Logical reconstruction of rdf and ontology languages // Principles and Practice of Semantic Web Reasoning / Ed. by François Fages, Sylvain Soliman. — Springer Berlin Heidelberg, 2005. — P. 65–71.

287 de Bruijn J., Heymans S. Logical foundations of rdf(s) with datatypes // J. Artif. Int. Res. — 2010. — may. — Vol. 38, no. 1. — P. 535–568.

288 Schneider M., Sutcliffe G. Reasoning in the owl 2 full ontology language using first-order automated theorem proving // Automated Deduction – CADE-23 / Ed. by Nikolaj Bjørner, Viorica Sofronie-Stokkermans. — Springer Berlin Heidelberg, 2011. — P. 461–475.

289 The logic of extensional rdfs / Enrico Franconi, Claudio Gutierrez, Alessandro Mosca et al. // The Semantic Web – ISWC 2013 / Ed. by Harith Alani, Lalana Kagal, Achille Fokoue et al. — Springer Berlin Heidelberg, 2013. — P. 101–116.

290 Köpcke H., Thor A., Rahm E. Evaluation of entity resolution approaches on real-world match problems // Proc. VLDB Endow. — 2010. — Vol. 3, no. 1. — P. 484–493.

291 An overview of end-to-end entity resolution for big data / Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas et al. // ACM Comput. Surv. — 2021. — Vol. 53, no. 6. — P. 127:1–127:42. — Access mode: https://doi.org/10.1145/3418896.

292 Bleiholder J. Data fusion and conflict resolution in integrated information systems. D.Sc. Diss. — Potsdam : Hasso-Plattner-Institut, 2010.

293 Yourdon E., Constantine L. L. Structured design: fundamentals of a discipline of computer program and systems design. — Prentice-Hall, Inc., 1979.

294 Jaql: A scripting language for large scale semistructured data analysis. / Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla et al. // PVLDB. — 2011. — Vol. 4, no. 12. — P. 1272–1283.

295 Pig latin: a not-so-foreign language for data processing / Christopher Olston, Benjamin C. Reed, Utkarsh Srivastava et al. // Proceedings of the ACM SIGMOD International

Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008 / Ed. by Jason Tsong-Li Wang. — ACM, 2008. — P. 1099–1110.

296 HIL: a high-level scripting language for entity integration / Mauricio A. Hernández, Georgia Koutrika, Rajasekar Krishnamurthy et al. // Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013 / Ed. by Giovanna Guerrini, Norman W. Paton. — ACM, 2013. — P. 549–560. — Access mode: https://doi.org/10.1145/2452376.2452440.

297 Miner D. MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems. — O'Reilly Media, 2012.

298 Ibm infosphere master data management documentation. — 2023. — Access mode: https://www.ibm.com/docs/en/imdm.

299 Extracting, linking and integrating data from public sources: A financial case study / Douglas Burdick, Mauricio A. Hernández, Howard Ho et al. // IEEE Data Eng. Bull. — 2011. — Vol. 34, no. 3. — P. 60–67. — Access mode: http://sites.computer.org/debull/A11sept/midas-deb1.pdf.

300 Hil2amn project. github repository. — 2023. — Access mode: https://goo.gl/IK1JzU.

301 Комплексная интегрированная информационная система «море». — 2024. — Access mode: https://portal.shipsea.ru/.

302 The apache hive data warehouse software. — 2024. — Access mode: http://hive.apache.org/.

303 Программный комплекс «поиск-море». — 2024. — Access mode: http://map.geopallada.ru/.

304 Единая государственная система информации об обстановке в мировом океане. — 2023. — Access mode: http://portal.esimo.ru/portal.

305 Devyatkin D., Shelmanov A. Text processing framework for emergency event detection in the arctic zone // Data Analytics and Management in Data Intensive Domains / Ed. by Leonid Kalinichenko, Sergei O. Kuznetsov, Yannis Manolopoulos. — Springer International Publishing, 2017. — P. 74–88.

306 Mosses P. D. Formal semantics of programming languages:—an overview— // Electronic Notes in Theoretical Computer Science. — 2006. — Vol. 148, no. 1. — P. 41–73.

307 Winskel G. The formal semantics of programming languages: an introduction. — MIT press, 1993.

308 Schmidt D. A. A Methodology for Language Development. — Allyn and Bacon, Boston, MA, 1986.

309 Goguen J., Malcolm G. Algebraic semantics of imperative programs. — MIT press, 1996.

310 Zamulin A. V. Algebraic semantics of an imperative programming language // Programming and Computer Software. — 2003. — Vol. 29, no. 6. — P. 328–337.

311 Kirchner F., Sinot F.-R. Rule-based operational semantics for an imperative language // Electronic Notes in Theoretical Computer Science. — 2007. — Vol. 174, no. 1. — P. 35–47.

312 Fernández M., Mackie I. A reversible operational semantics for imperative programming languages // International Conference on Formal Engineering Methods / Springer. — 2020. — P. 91–106.

313 K-st: A formal executable semantics of the structured text language for plcs / Kun Wang, Jingyi Wang, Christopher M Poskitt et al. // IEEE Transactions on Software Engineering. — 2023. — Vol. 49, no. 10. — P. 4796–4813.

314 Schirmer N. A sequential imperative programming language syntax, semantics, hoare logics and verification environment // Archive of Formal Proofs. — 2008. — Vol. 2008.

315 Guth D. A formal semantics of Python 3.3 : Ph.D. thesis / Dwight Guth ; University of Illinois at Urbana-Champaign. — 2013.

316 Rosu G., Serbănuță T. F. An overview of the k semantic framework // The Journal of Logic and Algebraic Programming. — 2010. — Vol. 79, no. 6. — P. 397–434.

317 Köhl M. A. An executable structural operational formal semantics for python // Master Thesis. Saarland University. — 2020. — Access mode: https://embedded.cs.uni-saarland.de/publications/theses/thesis_cs_msc_Koehl_Maximilian.pdf.

318 Plotkin G. D. The origins of structural operational semantics // The Journal of Logic and Algebraic Programming. — 2004. — Vol. 60. — P. 3–15.

319 Guagliardo P., Libkin L. A formal semantics of sql queries, its validation, and applications // Proceedings of the VLDB Endowment (PVLDB). — 2017. — Vol. 11, no. 1. — P. 27–39.

320 Benzaken V., Contejean E. A coq mechanised formal semantics for realistic sql queries: formally reconciling sql and bag relational algebra // Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. — 2019. — P. 249–261.

321 The rocq prover. — 2025. — Access mode: https://rocq-prover.org/.

322 Larsen P. G., Plat N., Toetenel H. A formal semantics of data flow diagrams // Formal Aspects Comput. — 1994. — Vol. 6, no. 6. — P. 586–606.

323 A generic and customizable framework for the design of ETL scenarios / Panos Vassiliadis, Alkis Simitsis, Panos Georgantas et al. // Inf. Syst. — 2005. — Vol. 30, no. 7. — P. 492–525.

324 Calegari D., Szasz N. Verification of model transformations: A survey of the state-of-the-art // XXXVIII Latin American Computer Conference - Selected Papers, CLEI 2012 Selected Papers, Medellin, Columbia, October 1-5, 2012 / Ed. by Yezid Donoso, Rodrigo M. Santos. — Vol. 292 of Electronic Notes in Theoretical Computer Science. — Elsevier, 2012. — P. 5–25. — Access mode: https://doi.org/10.1016/j.entcs.2013.02.002.

325 Ab. Rahim L., Whittle J. A survey of approaches for verifying model transformations // Software & Systems Modeling. — 2015. — Vol. 14, no. 2. — P. 1003–1028.

326 Ledang H., Dubois H. Proving model transformations // 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering / IEEE. — 2010. — P. 35–44.

327 Lano K., Clark T., Kolahdouz-Rahimi S. A framework for model transformation verification // Formal Aspects of Computing. — 2015. — Jan. — Vol. 27, no. 1. — P. 193–235. — Access mode: https://doi.org/10.1007/s00165-014-0313-z.

328 Z3 theorem prover. — 2025. — Access mode: https://github.com/Z3Prover.

329 Verification and validation of declarative model-to-model transformations through invariants / Jordi Cabot, Robert Clarisó, Esther Guerra, Juan de Lara // Journal of Systems and Software. — 2010. — Vol. 83, no. 2. — P. 283–302.

330 Cheng Z. Formal verification of relational model transformations using an intermediate verification language. — National University of Ireland, Maynooth (Ireland), 2016.

331 Boogie: A modular reusable verifier for object-oriented programs / Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine et al. // International Symposium on Formal Methods for Components and Objects / Springer. — 2005. — P. 364–387.

332 Knublauch H., Kontokostas D. Shapes constraint language (shacl). w3c recommendation. — 2017. — Access mode: https://www.w3.org/TR/shacl/.

333 Скворцов Н. А., Брюхов Д. О. Разработка схемы хранилища информации для поддержки поисковых действий в арктической зоне // Системы высокой доступности. — 2017. — Vol. 13, no. 2. — P. 20–44.

334 Shvaiko P., Euzenat J. Ontology matching: state of the art and future challenges // IEEE Transactions on knowledge and data engineering. — 2011. — Vol. 25, no. 1. — P. 158–176.

335 Do H.-H., Rahm E. Coma—a system for flexible combination of schema matching approaches // VLDB'02: Proceedings of the 28th International Conference on Very Large Databases / Elsevier. — 2002. — P. 610–621.

336 The harmony integration workbench / Peter Mork, Len Seligman, Arnon Rosenthal et al. // Journal on Data Semantics XI. — 2008. — P. 65–93.

337 Вовченко А. Е., Калиниченко Л. А., Ковалев Д. Ю. Методы разрешения сущностей и слияния данных в etl-процессе и их реализация в среде hadoop // Информатика и её применения. — 2014. — Vol. 8, no. 4. — P. 94–109.

338 Apache spark project. — 2024. — Access mode: http://spark.apache.org/.

339 Apache tez project. — 2024. — Access mode: http://tez.apache.org/.

340 Capriolo E., Wampler D., Rutherglen J. Programming Hive: Data warehouse and query language for Hadoop. — O'Reilly Media, Inc., 2012.

341 Ibm db2 big sql. — 2024. — Access mode: https://www.ibm.com/products/db2/big-sql.

342 Presto: Fast and reliable sql engine for data analytics and the open lakehouse. — 2024. — Access mode: https://prestodb.io/.

343 Apache doris: Open source, real-time data warehouse. — 2024. — Access mode: https://doris.apache.org/.

344 Система извлечения информации из текстов pullenti. — 2024. — Access mode: http://pullenti.ru/.

345 Systemt: a system for declarative information extraction / Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan et al. // ACM SIGMOD Record. — 2009. — Vol. 37, no. 4. — P. 7–13.

346 Брюхов Д., Скворцов Н. Извлечение информации из больших коллекций русскоязычных текстовых документов в среде hadoop // Электронные библиотеки: перспективные методы и технологии, электроные коллекции (RCDL 2014): Тр. — 2016. — P. 391–398.

347 Ibm infosphere information server documentation. — 2024. — Access mode: https://www.ibm.com/docs/en/iis.

348 Ibm db2 warehouse. — 2024. — Access mode: https://www.ibm.com/products/db2/warehouse.

349 Ibm db2 big sql federation capabilities. — 2022. — Access mode: https://www.ibm.com/docs/fr/db2-big-sql/6.0?topic=features-federation-capabilities.

350 The bigdawg polystore system / Jennie Duggan, Aaron J. Elmore, Michael Stonebraker et al. // SIGMOD Rec. — 2015. — Aug. — Vol. 44, no. 2. — P. 11–16.

351 Towards scalable hybrid stores: Constraint-based rewriting to the rescue / Rana Alotaibi, Damian Bursztyn, Alin Deutsch et al. — SIGMOD '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 1660–1677.

352 Polystore systems and dbmss: Love marriage or marriage of convenience? / Marco Vogt, David Lengweiler, Isabel Geissmann et al. // Heterogeneous Data Management, Polystores, and Analytics for Healthcare: VLDB Workshops, Poly 2021 and DMAH 2021, Virtual Event, August 20, 2021, Revised Selected Papers. — Berlin, Heidelberg : Springer-Verlag, 2021. — P. 65–69.

353 Polypheny-db: Towards bridging the gap between polystores and htap systems / Marco Vogt, Nils Hansen, Jan Schönholz et al. // Heterogeneous Data Management, Polystores, and Analytics for Healthcare / Ed. by Vijay Gadepally, Timothy Mattson, Michael Stonebraker et al. — Cham : Springer International Publishing, 2021. — P. 25–36.

354 Hkpoly: A polystore architecture to support data linkage and queries on distributed and heterogeneous data / Leonardo Guerreiro Azevedo, Renan Souza, Elton Soares et al. // Proceedings of the 20th Brazilian Symposium on Information Systems. — SBSI '24. — New York, NY, USA : Association for Computing Machinery, 2024.

355 Yamashita F., Chang Q., Miyazaki J. Unified schema-driven graph polystore: Achieving transparency in multi-model integration and migration // Database and Expert Systems Applications / Ed. by Robert Wrembel, Gabriele Kotsis, A. Min Tjoa, Ismail Khalil. — Cham : Springer Nature Switzerland, 2026. — P. 136–143.

356 Multicategory: multi-model query processing meets category theory and functional programming / Valter Uotila, Jiaheng Lu, Dieter Gawlick et al. // Proc. VLDB Endow. — 2021. — Jul. — Vol. 14, no. 12. — P. 2663–2666.

357 Empowering big data analytics with polystore and strongly typed functional queries / Annabelle Gillet, Éric Leclercq, Marinette Savonnet, Nadine Cullot. — IDEAS '20. — New York, NY, USA : Association for Computing Machinery, 2020.

358 Shi G., Wang C., Liu Y. Ecql: Towards succinct and extensible modeling of multi-model query results // Conceptual Modeling / Ed. by Wolfgang Maass, Hyoil Han, Hasan Yasar, Nick Multari. — Cham : Springer Nature Switzerland, 2025. — P. 112–130.

359 Van Gelder A., Ross K. A., Schlipf J. S. The well-founded semantics for general logic programs // Journal of the ACM (JACM). — 1991. — Vol. 38, no. 3. — P. 619–649.

360 Gelfond M., Lifschitz V. The stable model semantics for logic programming // Proc. 5'h International Conference and Symposium on Logic Programming. — 1988. — P. 1070–1080.

361 A rule-based language for web data management / Serge Abiteboul, Meghyn Bienvenu, Alban Galland, Émilien Antoine // Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. — 2011. — P. 293–304.

362 Sharpe W. F., Alexander G. J., Bailey J. W. Investments. — Prentice Hall, 1999.

363 Jensen K. Coloured Petri Nets: Basic Concepts, analysis methods and practical use. — Springer, 1991.

364 Butler M. csp2b: A practical approach to combining csp and b // Form. Aspects Comput. — 2000. — Vol. 12. — P. 182—198.

365 Schneider S., Davies J. A brief history of timed csp // Theoretical Computer Science. — 1995. — Vol. 138. — P. 243–271.

366 Treharne H., Schneider S. How to drive a b machine // ZB 2000: Formal Specification and Development in Z and B. — Springer Berlin Heidelberg, 2000. — P. 188–208.

367 Ledang H., Souquières J. Contributions for modelling uml state-charts in b // Integrated Formal Methods / Ed. by Michael Butler, Luigia Petre, Kaisa Sere. — Springer Berlin Heidelberg, 2002. — P. 109–127.

368 W3c xml schema definition language (xsd) 1.1 part 2: Datatypes. w3c recommendation / D. Peterson, S. Gao, A. Malhotra et al. — 2012. — Access mode: http://www.w3.org/TR/xmlschema11-2/.

369 Система мониторинга судов «виктория». — 2024. — Access mode: http://victoria.marsat.ru/.

370 Система коспас-сарсат. — 2024. — Access mode: https://www.cospas-sarsat.int/ru/.

371 Kiselyova N. N., Dudarev V. A., Zemskov V. S. Computer information resources of inorganic chemistry and materials science // Russian Chemical Reviews. — 2010. — Vol. 79, no. 2. — P. 145.

372 Kiselyova N. N., Dudarev V. A., Stolyarenko A. V. Integrated system of databases on the properties of inorganic substances and materials // High Temperature. — 2016. — Vol. 54, no. 2. — P. 215–222.

373 Kiselyova N. N., Dudarev V. A., Korzhuyev M. A. Database on the bandgap of inorganic substances and materials // Inorganic Materials: Applied Research. — 2016. — Vol. 7. — P. 34–39.

374 The binary star database. — 2025. — Access mode: https://bdb.inasan.ru/.

375 Binary star database bdb development: Structure, algorithms, and vo standards implementation / Dana Kovaleva, Pavel Kaygorodov, Oleg Malkov et al. // Astronomy and Computing. — 2015. — Vol. 11. — P. 119–125.

376 The binary star database bdb v3. 0 / Dana Kovaleva, Oleg Malkov, Pavel Kaygorodov, Bernard Debray // Astronomical Data Analysis Software and Systems XXVI / Astronomical Society of the Pacific. — Vol. 521. — 2016. — P. 217.

377 A new version of the binary star database bdb: Challenges and directions / Pavel Kaygorodov, Nikolay Skvortsov, Dana Kovaleva, Oleg Malkov // Open Astronomy. — 2023. — Vol. 32, no. 1. — P. 20220215.

378 The washington visual double star catalog. — 2020. — Access mode: https://vizier.cds.unistra.fr/viz-bin/VizieR-3?-source=B/wds/wds.

379 The 2001 us naval observatory double star cd-rom. i. the washington double star catalog / Brian D Mason, Gary L Wycoff, William I Hartkopf et al. // The Astronomical Journal. — 2001. — Vol. 122, no. 6. — P. 3466.

380 ГОСТ Р ИСО/МЭК 12207—2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. — Москва: Стандартинформ, 2011.

381 ГОСТ Р 59352—2021. Системная инженерия. Системный анализ процесса управления рисками для системы. — Москва: Стандартинформ, 2022.

382 Buchi M. The b bank // Program Development by Refinement. Formal Approaches to Computing and Information Technology FACIT / Ed. by E. Sekerinski, K. Sere. — Springer, 1999.

ПРИЛОЖЕНИЕ А

Применения метода унификации моделей данных

А.1 Унификация сетевой модели данных CODASYL

В данном разделе приведено краткое описание этапов унификации сетевой модели данных CODASYL от формального определения синтаксиса до конструирования трансформации.

Формализация синтаксиса модели CODASYL. Синтаксис модели формализован с использованием языка SDF. Пример определения синтаксического правила для *записи* (record) выглядит следующим образом:

```
module unifier/codasyl/DDL-Syntax
exports
  sorts
    Record-Entry Calcs Data-Subentry-List
  context-free syntax
    "RECORD" "NAME" "IS" DDL-Id ";"
    "LOCATION" "MODE" "IS" "CALC" "USING" Calcs
    "DUPLICATES" "NOT" "ALLOWED" ";"
     Data-Subentry-List
     -> Record-Entry
```

Полное определение конкретного синтаксиса используемого подмножества модели CODASYL опубликовано в репозитории GitHub[87].

Формализация абстрактного синтаксиса модели CODASYL в виде диаграммы классов UML приведена на рис. А.1.

Формализация синтаксиса языка СИНТЕЗ. Синтаксис модели формализован с использованием языка SDF. Пример определения синтаксических правил для секции типов и абстрактного типа данных выглядит следующим образом:

```
module unifier/synthesis/Synthesis-Syntax
exports
sorts
  Type-Specification-List  Type-Specification
  Abstract-Type  Abstract-Type-Metaframe
  Attribute-Specification-List  Attribute-Specification
  Assertion-List
context-free syntax
  "type" ":" Type-Specification-List ";" -> Type-Section
```

[87] https://github.com/sstupnikov/ModelTransformation/blob/master/Codasyl2Synthesis/unifier/codasyl/DDL-Syntax.sdf

```
{Type-Specification ","}* -> Type-Specification-List
Abstract-Type -> Type-Specification
"{" Synthesis-Id ";"  "in" ":" "type" ";"
    Abstract-Type-Metaframe?
    Attribute-Specification-List
"}"
 -> Abstract-Type
"metaframe" Assertion-List "end" -> Abstract-Type-Metaframe
```

Полное определение синтаксиса используемого подмножества языка СИНТЕЗ опубликовано в репозитории GitHub[88]. Формализация абстрактного синтаксиса подмножества в виде диаграммы классов UML приведена на рис. А.2.

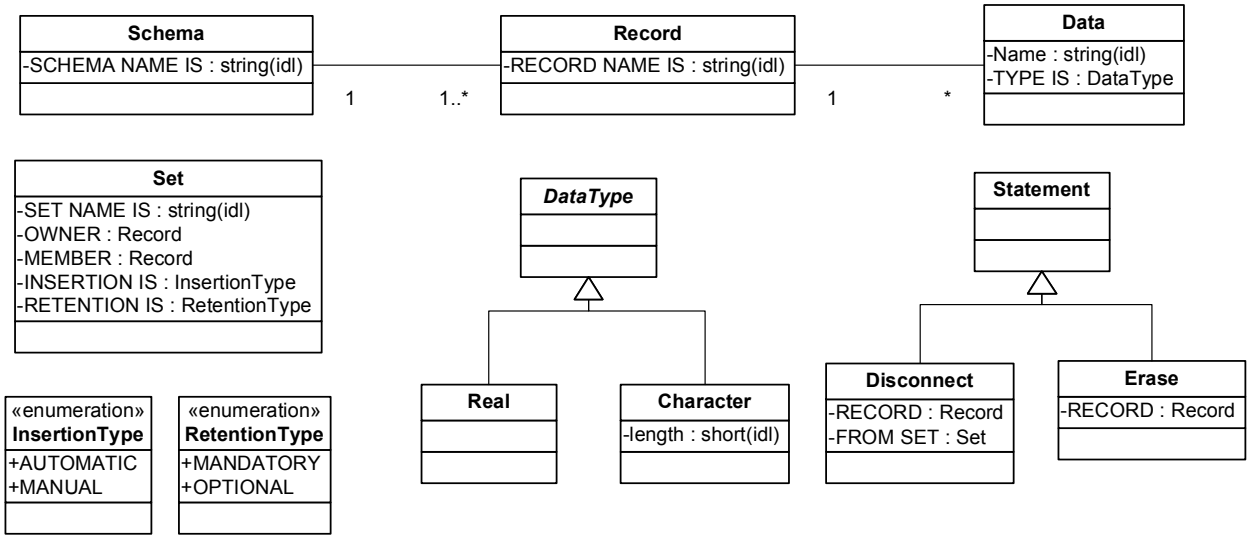

Рисунок А.1 – Абстрактный синтаксис модели CODASYL в виде диаграммы классов UML

На диаграмме представлены связи двух видов: отношение обобщения — специализации (например, тип *Real* является подтипом *DataType*) и ассоциации различных кардинальностей (например, со схемой *Schema* могут быть связаны записи — *Record*).

Установление соответствий элементов моделей. Соответствия между элементами моделей установлены следующим образом (Таблица А.1). Названия элементов соответствуют сортам синтаксисов.

88 https://github.com/sstupnikov/ModelTransformation/blob/master/Codasyl2Synthesis/unifier/synthesis/Synthesis-Syntax.sdf

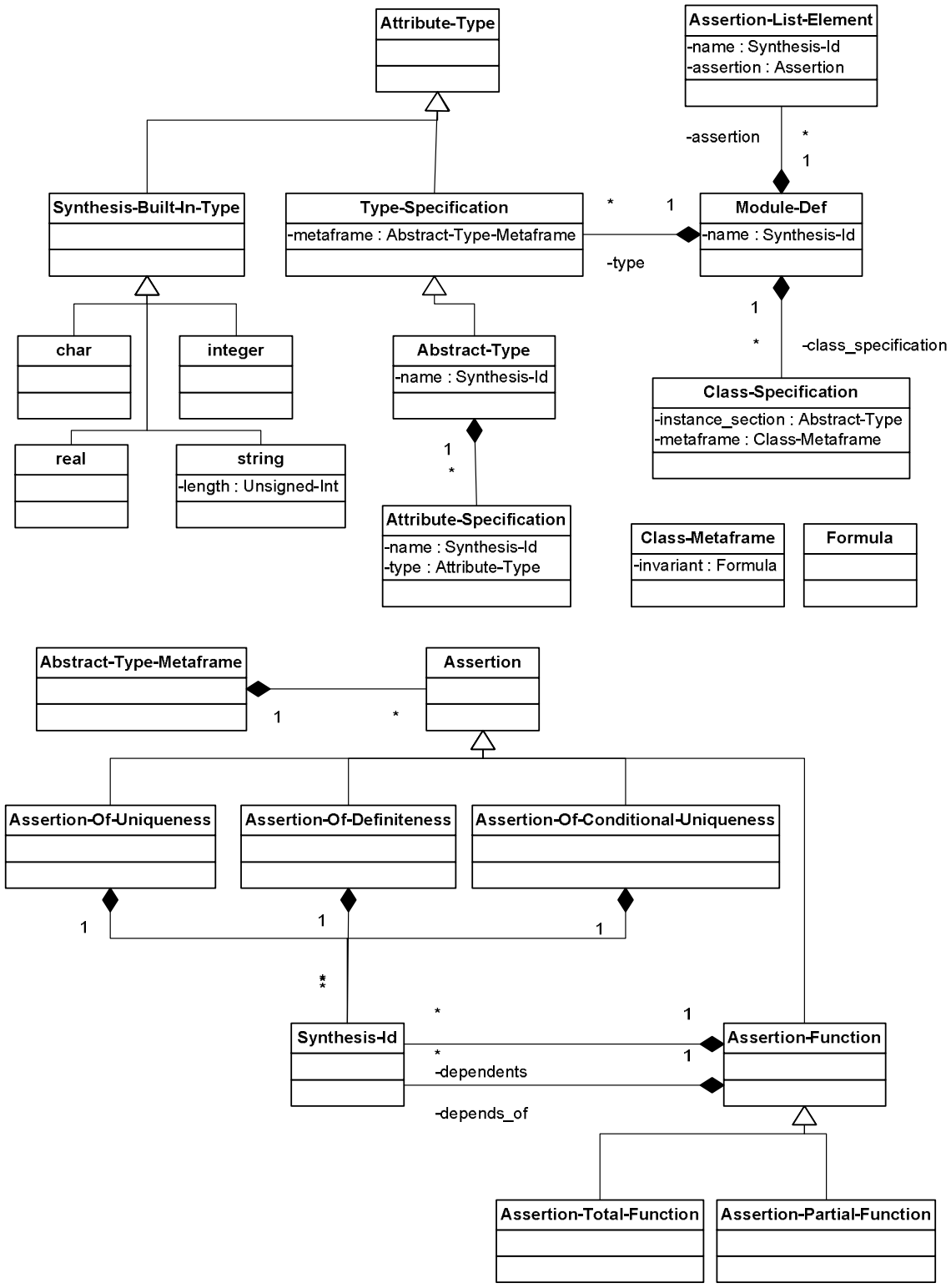

Рисунок А.2 - Абстрактный синтаксис подмножества языка СИНТЕЗ в виде диаграммы классов UML

Таблица А.1 - Соответствия элементов модели CODASYL элементам языка СИНТЕЗ

Элемент CODASYL	**Элемент канонической модели**
Schema-Def	Module-Def
Schema-Def.SCHEMA NAME IS	Module-Def.name
DDL-Id	Synthesis-Id
Record-Entry	Abstract-Type
Record-Entry.RECORD NAME IS	Type-Specification.name
Record-Entry	Class-Specification
Record-Entry.RECORD NAME IS	Class-Specification.name
Schema-Def.records	Module-Def.type
Schema-Def.records	Module-Def.class_specification
Data-Subentry	Attribute-Specification
Record-Entry.Calcs	Type-Specification.metaframe
Record-Entry.Calcs.RANGE	Assertion-Of-Uniqueness
Set-Entry.OWNER IS	Assertion-Function
Set-Entry.MEMBER IS	Assertion-Function
Membership-Type.MANDATORY AUTOMATIC	Assertion-Total-Function
Membership-Type.OPTIONAL MANUAL	Assertion-Partial-Function

Создание расширения канонической модели. Необходимое расширение языка СИНТЕЗ включает:

– утверждения о полной (*func*) и частичной (*pfunc*) функциональной зависимости атрибутов типа экземпляров одного класса (например, агроэкологической группы земель AE_GROUP) от атрибутов типа экземпляров другого класса (например, агроэкологического вида земель AE_TYPE):

```
{func; {AE_TYPE; ae_group_id}, {AE_GROUP; ae_group_id}};
```

– аксиома уникальности экземпляра классов (например, TREC01, TREC02) по набору значений атрибутов (например, DE01 и DE05):

```
all x/TREC01 ( TREC01(x/TREC01[DE01]) & TREC02(x/TREC02[DE01, DE05]) ->
count(x) = 1)
```

Конструирование трансформации моделей. Вербальные правила отображения модели CODASYL в язык СИНТЕЗ состоят в следующем:

– схема CODASYL отображается в одноименный модуль СИНТЕЗ;

– тип записи *R* отображается в некоторый АТД *T* и класс *C* с типом экземпляров *T*;

– атрибуты типа записи отображаются в атрибуты АТД;

– встроенный тип FLOAT отображается в тип *real*, FIXED – в *integer*, CHARACTER – в *string*;

– список атрибутов LOCATION MODE типа записи отображается в утверждение уникальности списка атрибутов АТД;

– если запись R_1 является владельцем набора S, запись R_2 - членом набора S, то:

1) ключевые атрибуты R_1 становятся атрибутами АТД, соответствующего R_2;

2) в раздел утверждений модуля добавляется утверждение о полной функциональной зависимости (в случае типа связи набора MANDATORY AUTOMATIC) или частичной функциональной зависимости (в случае OPTIONAL MANUAL) класса члена набора C_2 от класса владельца набора C_1 по ключевым атрибутам владельца;

– селектирующий путь набора S отображается в аксиому уникальности класса C_1 владельца R_1 набора:

– например, для пути

```
SET SELECTION IS THROUGH TSET01 OWNER IDENTIFIED BY KEY
THEN THROUGH TSET02 OWNER IDENTIFIED BY DE05, DE06, DE07
```

аксиома выглядит так:

```
all x/TREC01 ( TREC01(x/TREC01[DE01, DE02, DE03]) &
TREC02(x/TREC02[DE01, DE02, DE03, DE05, DE06, DE07]) -> count(x) = 1)
```

– аксиома включает ключи всех записей-владельцев наборов, включенных в путь;

– в случае, если селектирующий путь состоит только из ключа владельца R_1 набора S, то аксиома не создается, поскольку утверждение об уникальности ключа содержится в описании АТД;

– в случае, если запись R_1 является владельцем двух или более наборов с одним и тем же селектирующим путем, аксиома для всех таких наборов порождается только одна (поскольку аксиомы должны быть порождены одинаковые).

Вербальные правила отображения формализованы с использованием языков SDF и ASF. В SDF-файле определяются сигнатуры правил переписывания термов. Например, сигнатуры правил, необходимых для переписывания типа записи в АТД выглядят следующим образом:

```
module unifier/codasyl2synthesis/translator
imports unifier/codasyl/DDL-Syntax
imports unifier/synthesis/Synthesis-Syntax

exports
  context-free syntax
    t-DDL-Id(DDL-Id) -> Synthesis-Id
    t-Calcs(Calcs) -> {Synthesis-Id ","}*
    t-Data-Subentry-List(Data-Subentry-List) -> Attribute-Specification*
    t-Extract-Owners-Keys(DDL-Id, Set-Entry, Record-Entry-List) ->
Attribute-Specification*
    t-Create-Type-Specifications(Record-Entry-List, Set-Entry-List, Record-
Entry- List) -> {Type-Specification ","}*
```

В ASF-файле определяются спецификации правил, соответствующие сигнатурам:

```
equations
[Create-Type-Specifications]
Synthesis-Id := t-DDL-Id(DDL-Id),
Attribute-Name-List := t-Calcs(Calcs),
Attribute-Specification-List :=
t-Data-Subentry-List(Data-Subentry-List)
t-Extract-Owners-Keys(DDL-Id, Set-Entry-List, Record-Entry*)
====>
t-Create-Type-Specifications(
  RECORD NAME IS DDL-Id;
  LOCATION MODE IS CALC USING Calcs
  DUPLICATES NOT ALLOWED;
  Data-Subentry-List
,
Set-Entry-List,
Record-Entry*
)=
{ Synthesis-Id; in: type;
  metaframe {unique; { Attribute-Name-List }}; end
  Attribute-Specification-List
}
```

Полные спецификации правил для выбранного подмножества модели CODASYL опубликованы в репозитории GitHub[89].

Пример применения трансформации для преобразования подмножества схемы реестра агроэкологических групп земель, представленного в модели CODASYL, в спецификацию модуля языка СИНТЕЗ приведен в таблице А.2.

А.2 Унификация процессной модели данных

Раздел основан на работах автора [68] [65] [111] [43].

Анализ процессных моделей потоков работ разнообразных СУПР, проведенный группой Ван дер Аальста [164] показал, что предложенных образцов потоков работ достаточно для моделирования их конструкций. Более того, существующие СУПР реализуют лишь различные слабые подмножества полного набора образцов. С другой стороны, полнота этого набора образцов подтверждается тем, что введенного разнообразия достаточно для описания произвольных реальных потоков работ.

Эти результаты позволяют выбрать для синтеза канонической модели процессов в качестве исходных процессных моделей указанные образцы потоков работ. Тем самым удастся сочетать процессную полноту выбранного набора конструкций с возможностью

[89] https://github.com/sstupnikov/ModelTransformation/tree/master/Codasyl2Synthesis/unifier/codasyl2synthesis

моделирования этим набором произвольных процессов, выражаемых процессными моделями разнообразных СУПР.

Таблица А.2 - Пример преобразования схемы CODASYL в спецификацию модуля языка СИНТЕЗ

Схема CODASYL	**Модуль языка СИНТЕЗ**

SCHEMA NAME IS
AGROECOLOGICAL_GROUP_REGISTRY;
 	
RECORD NAME IS AE_GROUP;
LOCATION MODE IS CALC USING
AE_GROUP_ID
DUPLICATES NOT ALLOWED;
AE_GROUP_ID TYPE IS FIXED,
NAME TYPE IS CHARACTER 30;
 	
RECORD NAME IS AE_TYPE;
LOCATION MODE IS CALC USING
EMPNO
DUPLICATES NOT ALLOWED;
AE_TYPE_ID TYPE IS FIXED,
NAME TYPE IS CHARACTER 30,
RELIEF TYPE IS CHARACTER 30;
 	
SET NAME IS GROUP_TYPE;
OWNER IS AE_GROUP;
MEMBER IS AE_TYPE
MANDATORY AUTOMATIC
DUPLICATES ARE NOT ALLOWED FOR
AE_GROUP_ID
SET SELECTION IS THROUGH
GROUP_TYPE OWNER
IDENTIFIED BY KEY
	{ AGROECOLOGICAL_GROUP_REGISTRY;
	in: module;
	
	type:
	{ AE_GROUP; in: type;
	metaframe {unique; {AE_GROUP_ID}}; end
	AE_GROUP_ID : integer;
	NAME: {string; length: 30;};
	},
	{ AE_TYPE; in: type;
	metaframe {unique; {AE_TYPE_ID}}; end
	AE_TYPE_ID: integer;
	NAME: {string; length: 30;};
	RELIEF: {string; length: 30;};
	AE_GROUP: integer;
	},
	
	class_specification:
	{ CAE_GROUP; in: class;
	instance_section: AE_GROUP;
	},
	{ CAE_TYPE; in: class;
	instance_section: AE_TYPE;
	};
	
	assertion:
	{func; {AE_TYPE; AE_GROUP_ID},
	{AE_GROUP; AE_GROUP_ID}};
	}

Ядро канонической процессной модели и его синтаксис. В качестве ядра канонической процессной модели было выбрано подмножество языка скриптов СИНТЕЗ ([42], раздел 7.6), наиболее близкое к раскрашенным сетям Петри [363]. Ядро обладает следующими свойствами:

– основано на хорошо изученной и распространенной модели сетей Петри;

– вводит сети Петри в объектную среду. В результате потоки управления (control flow) и потоки данных (data flow) удается соединить, используя токены - объекты, принадлежащие определенному типу и обладающие уникальными идентификаторами;

– предоставляет средства связывания переходов сетей Петри с функциями, которые должны быть выполнены при срабатывании этих переходов. При этом также задаются правила связывания входных и выходных токенов перехода с входными и

выходными параметрами функции. Такое связывание необходимо для моделирования информационных систем в целом, что не принимается во внимание в более абстрактных моделях [164] [363].

Скрипты определяются с помощью родового типа скрипта в языке СИНТЕЗ и могут образовывать иерархию с отношением тип-подтип. Родовой тип – это наименее конкретизированный тип. Ниже приводится определение родового типа скрипта в форме Бэкуса-Наура. Каждый экземпляр типа скрипта отвечает некоторому исполнению потока работ, определяемому данным скриптом.

```
<script type identifier>::=
{ <type identifier>; in: script;
  [params: {<formal parameter list>};]
  [supertype: {<supertype name list>};]
  instance_section: {
    [<attribute specification list>;]
    [initial: <initial marking list>;]
    states: <state section>;
    [transitions: <transition section>;]
  }
}
```

Здесь *type identifier* – имя типа скрипта. В необязательном слоте *params* задаются формальные параметры типа. Супертипы данного типа (если они есть) перечисляются в слоте *supertype*. Экземпляр типа определяет слот *instance_section*. Для каждого экземпляра в слоте *initial* определяется начальная разметка – множество пар (имя состояния, константа). Константа задает значение токена, находящегося в начальный момент в указанном состоянии:

```
<element of initial marking list> ::= {<state identifier> , <constant>}
```

Слот *states* определяет множество мест конкретной сети. Каждое место характеризуется именем и типом токенов, которые могут в нем находиться:

```
<state section> ::=
  <state specification> | <state specification>, <state section>
<state specification> ::= { <state name>; token: <type>; }
```

В слоте *transitions* определяется множество переходов конкретной сети:

```
<transition section> ::=
  <transition specification> |
  <transition specification>, <transition section>
```

Каждый переход характеризуется именем (*transition name*), списком входных мест (*from*), списком выходных мест (*to*), списком условий (*conditions*), функцией, которая выполняется при срабатывании перехода (*activity*), а также списками связывания входных и выходных параметров этой функции (*bind_from, bind_to*).

```
<transition specification> ::=
{ <transition name>;
  from: <list of input state names>;
  [bind_from: <binding list>;]
  to: <list of output state names>;
  [bind_to: <binding list>;]
  [conditions: <assertion list>;]
  activity: <function declaration>;
}
```

В слоте *from* определяются места, из которых токены могут входить в данный переход при его срабатывании. В слоте *to* определяются места, в которых могут появляться токены при срабатывании данного перехода. Слот *conditions* определяет условия, при которых переход может сработать (помимо наличия необходимого числа токе- нов определенных типов в определенных местах).

Слот *activity* задает функцию, выполняемую при срабатывании перехода. При этом входными параметрами этой функции являются токены, входящие из мест, описанных в секции *bind_from*, а выходными – токены, помещаемые в места, описанные в секции *bind_to*. В этих двух секциях также задаются условия, в соответствии с которыми токены могут перемещаться.

```
<element of binding list> ::=
  { <state identifier>, <input/output parameter name> [, <condition>] }
```

Определение формальной семантики ядра канонической модели процессов. Скрипты позволяют описывать процессы как последовательности действий, каждое из которых может вызывать смену состояний базы данных, задаваемой спецификацией канонической модели. Основой для определения семантики скриптов в AMN является отображение абстрактных типов данных канонической модели в AMN [49], а также ряд исследований по отображению процессных моделей в AMN.

Батлером [364] был разработан язык, сочетающий ограниченное подмножество процессной алгебры CSP [162] и AMN, а также методы и средства отображения комбинированного языка в AMN. CSP при этом использовался как язык структуризации абстрактных машин. Естественным расширением данного подхода стала работа [99], расширяющая метод отображения подмножества CSP в AMN до отображения полного Timed CSP [365] (CSP, расширенный временными примитивами) в AMN. Идея комбинации CSP и AMN также разрабатывалась в работе Шнайдера [366], где подмножество CSP использовалось для управления абстрактными машинами AMN. В подходе к интеграции UML и AMN Леданг использует [367] идею охраняемых (guarded) подстановок для моделирования событий в диаграммах состояний (state-charts) UML.

Благодаря всем перечисленным работам сформировалось общее понимание, каким образом следует определять семантику процессных моделей в AMN. Это понимание, в свою очередь, позволило разработать метод отображения скриптов в AMN.

Основная идея отображения скриптов в AMN заключается в представлении состояния системы (т.е. мест скрипта) при помощи переменных AMN, и моделировании переходов при помощи операций AMN, тела которых представляют из себя охраняемые подстановки. Такая идея характерна для всех подходов, применяемых для представления процессных моделей в AMN.

Рассмотрим тип скрипта *S*, заданный следующей спецификацией:

```
{ S; in: script;
  params: { pf/function, PT/type };
  instance_section: {
    states: ... ;
    initial: ... ;
    transitions: ... ;
  }
}
```

Скрипт *S* представляется в AMN конструкцией REFINEMENT с именем *S*. Использование конструкции REFINEMENT, а не конструкций MACHINE или IMPLEMENTATION обусловлено необходимостью представления скриптов в AMN однородными структурами, поскольку любой скрипт может выступать как в роли уточняемой, так и в роли уточняющей спецификации.

Композиция машины *S* с машинами, представляющими другие абстрактные типы, производится следующим образом. Рассмотрим абстрактный тип S_F, спецификацию которого составляют методы, задаваемые функциями переходов скрипта. Машина *S* состоит ровно в тех отношениях композиции с машинами, представляющими другие абстрактные типы, в которых состояла бы с этими машинами машина S_F, представляющая тип S_F в AMN. Аналогично машинам, представляющим абстрактные типы, машина *S* состоит в отношении *SEES* с контекстной машиной *ContextM* модуля *M,* вввключающего тип скрипта.

В случае, если скрипт является родовым, и имеет параметр-тип *PT*, экстенсионал параметра определяется константами *ext_PT, extp_PT* в разделе PROPERTIES следующим образом:

$ext\_PT \in \mathrm{POW}(Obj) \wedge extp\_PT \in \mathrm{POW}(Obj) \wedge extp\_PT \subseteq ext\_PT$

Если скрипт имеет параметр-функцию *pf*, то считается, что параметр-функция имеет минимально необходимое количество входных и выходных параметров, определяемое сигнатурой функции слота *activity* перехода.

Каждое из *мест* скрипта, описанное в слоте *states*, представляется в AMN отдельной переменной. Место

```
{ s; token: T; }
```

где *T* - абстрактный тип данных, представляется переменной *s*, типизированной в разделе INVARIANT машины как

```
s ∈ POW(ext_T)
```

В случае, если начальная маркировка места *s* не описана в слоте *initial* скрипта, в разделе *INITIALISATION* машины переменная *s* инициализируется подстановкой

```
s := ∅
```

Если начальная маркировка места *s* описана элементом списка маркировки

```
{s, c}
```

где *c* - константа, то переменная *s* инициализируется подстановкой

```
s := mTerm[c]
```

где *mTerm* - семантическая функция отображения в AMN термов канонической модели.

Рассмотрим *переход*, задаваемый следующей спецификацией:

```
{ t;
  from: s11, s12, s21, s22;
  bind_from:
   {s11, p1, b11},  {s12, p1, b12},
   {s21, p2, b21},  {s22, p2, b22};
  to: s31, s32;
  bind_to:
   {s31, r1, b31}, {s32, r1, b32},
   {s41, r2, b41}, {s42, r2, b42};
  conditions: C1, ... , Cn;
  activity: { in: function;
    params: { +p1/T1, +p2/T2, -r1/T3, -r2/T4, -o/T5 };
    { predicative: { f } }
  }
}
```

Функция перехода имеет два входных параметра p_1, p_2 и три выходных параметра r_1, r_2, o. Места s_{11}, s_{12} с типом токенов T_1 связываются с параметром p_1 условиями b_{11}, b_{12} соответственно. Места s_{21}, s_{22} с типом токенов T_2 связываются с параметром p_2 условиями b_{21}, b_{22} соответственно. Места s_{31}, s_{32} с типом токенов T_3 связываются с параметром r_1 условиями b_{31}, b_{32} соответственно. Места s_{41}, s_{42} с типом токенов T_4 связываются с параметром r_2 условиями b_{41}, b_{42} соответственно. С параметром o не связываются никакие места.

Переход *t* представляется в AMN следующей операцией:

```
r1, r2, o ← t(p1, p2) =
SELECT
  mPredicate[C1] ∧ ... ∧ mPredicate[C1] ∧
  (∃ t1 (t1 ∈ s11 ∧ mPredicate[b11]) ∨ ∃ t1 (t1 ∈ s12 ∧ mPredicate[b12])) ∧
  (∃ t2 (t2 ∈ s21 ∧ mPredicate[b21]) ∨ ∃ t2 (t2 ∈ s22 ∧ mPredicate[b22]))
THEN
  ANY t1, t2 WHERE
    t1 ∈ Obj ∧ t2 ∈ Obj ∧
    (t1 ∈ s11 ∧ mPredicate[b11] ∨ t1 ∈ s12 ∧ mPredicate[b12]) ∧
    (t2 ∈ s21 ∧ mPredicate[b21] ∨ t2 ∈ s22 ∧ mPredicate[b22])
  THEN
    s11 := s11 − {t1} || s12 := s12 − {t1} ||
    s21 := s21 − {t2} || s22 := s22 − {t2} ||
    BEGIN
      mSubstitution[f]{p1 → t1, p2 → t2};
      BEGIN
        SELECT mPredicate[b31] THEN s31 := s31 ∪ {r1}
        WHEN mPredicate[b32] THEN s32 := s32 ∪ {r1}
        END ||
        SELECT mPredicate[b41] THEN s41 := s41 ∪ {r2}
        WHEN mPredicate[b42] THEN s42 := s42 ∪ {r2}
        END
      END
    END
  END
END
```

Тело операции представляет собой подстановку SELECT, условным предикатом которой является условие срабытывания перехода, извлекаемое из связываний слота *bind_from*. Представление условий связывания в AMN определяется семантической функцией *mPredicate*, преобразующей формулы канонической модели в предикаты AMN.

В случае выполнения условия срабатывания перехода, из мест, определенных связываниями слота *bind_from*, выбираются подходящие токены для передачи в качестве входных параметров в функцию перехода. Выбранные токены удаляются из своих мест.

Одновременно с этим выполняется подстановка, представляющая собой последовательную композицию двух подстановок, первая из которых представляет функцию перехода, а вторая представляет правильное размещение токена, полученного на выходе перехода.

Тело подстановки, представляющей функцию перехода, формирует семантическая функция *mSubstitution*; в полученном теле имена входных параметров p_1, p_2 заменяются на имена переменных t_1 и t_2, хранящих выбранные токены.

Токен, полученный в результате выполнения подстановки, представляющей функцию перехода, размещается в одном из мест, определенном связываниями слота *bind_to* при помощи охраняемой подстановки SELECT.

Действие перехода на пространстве состояний системы может задаваться функцией-параметром *pf* скрипта. При этом формула *f* имеет вид вызова функции-параметра *pf*(p_1, p_2, r_1, r_2, o) с соответствующими фактическими параметрами. Если функция-параметр в скрипте, семантику которого необходимо определить, не конкретизирована, то действие функции-параметра на пространстве состояний системы моделируется произвольным выбором выходных параметров:

```
ANY r'1, r'2, o'
WHERE r'1 ∈ ext_T3 ∧ r'2 ∈ ext_T4 ∧ o' ∈ ext_T5
  r1 := r'1 ||
  r2 := r'2 ||
  o := o'
END
```

В случае, если скрипт специфицирует лишь поток управления, т.е. не конкретизирует действие *activity* на пространстве состояний системы, вид операции, соответствующей переходу, упрощается:

```
t =
SELECT
  mPredicate[C1] ∧ ... ∧ mPredicate[C1] ∧
  (∃ t1 (t1 ∈ s11 ∧ mPredicate[b11]) ∨ ∃ t1 (t1 ∈ s12 ∧ mPredicate[b12])) ∧
  (∃ t2 (t2 ∈ s21 ∧ mPredicate[b21]) ∨ ∃ t2 (t2 ∈ s22 ∧ mPredicate[b22]))
THEN
  ANY t1, t2 WHERE
    t1 ∈ Obj ∧ t2 ∈ Obj ∧
    (t1 ∈ s11 ∧ mPredicate[b11] ∨ t1 ∈ s12 ∧ mPredicate[b12]) ∧
    (t2 ∈ s21 ∧ mPredicate[b21] ∨ t2 ∈ s22 ∧ mPredicate[b22])
  THEN
    s11 := s11 − {t1} || s12 := s12 − {t1} ||
    s21 := s21 − {t2} || s22 := s22 − {t2} ||
    BEGIN
      SELECT mPredicate[b31] THEN s31 := s31 ∪ {r1}
      WHEN mPredicate[b32] THEN s32 := s32 ∪ {r1}
      END ||
      SELECT mPredicate[b41] THEN s41 := s41 ∪ {r2}
      WHEN mPredicate[b42] THEN s42 := s42 ∪ {r2}
      END
    END
  END
END
```

Вышеприведенные рассуждения очевидным образом обобщаются на случай произвольного числа входных и выходных параметров функции перехода.

Построение расширений ядра канонической процессной модели. Расширения ядра канонической модели процессов будем строить для образцов потоков работ, определенных в работе [164]. Каждому образцу ставится в соответствие родовой тип скрипта. При этом ряд элементов скрипта (используемые функции и типы данных)

являются параметрами типа. Каждый переход параметризуется типом функции, которая вызывается при срабатывании этого перехода. Каждое место параметризуется типом токена, допустимым для данного места. Функция, подставляемая в качестве параметра скрипта, должна иметь определенное количество входных и выходных параметров определенных типов, сссоответствующее структуре скрипта.

Графическому изображению скриптов соответствует двудольный граф с двумя видами вершин – местами (изображаемыми кругами) и переходами (изображаемыми квадратами). Вершины разных видов соединяются отношением инцидентности (стрелки). В местах могут накапливаться токены различных типов. Переходы могут срабатывать при выполнении определенных условий, поглощая токены из входных мест перехода и производя токены в его выходных местах.

Входные и выходные места образца называются *entrance* и *exit* соответственно (с добавлением номера в случае необходимости). Дополнительные места носят названия *auxPlace, auxPlace1, auxPlace2* и т.д. Если образец связан с ветвлением, «стволовой» переход называется *Trunk*, а переходы в ветвях – *Branch*. В случае необходимости также вводится дополнительная нумерация. Такие нотационные соглашения не накладывают никаких ограничений на порядок использования и реализации образцов и введены для удобочитаемости.

В качестве примера рассмотрим расширение ядра канонической модели для образца *дискриминатора* (другие образцы группы потока управления рассмотрены в работе [43]). Данный образец описывает ситуацию, когда ожидается завершение одной из нескольких одновременно выполняемых (concurrent) ветвей процесса. В результате этого активизируется последующий переход, а все остальные параллельные ветви отменяются.

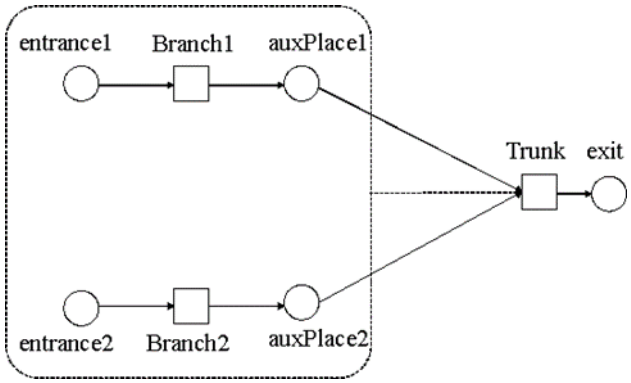

Рисунок А.3 – Образец дискриминатора

В канонической модели данному образцу соответствует родовой тип скрипта *discriminator* (рис. А.3). На рисунке с переходом *Trunk* пунктирной линией соединен

пунктирный прямоугольник со скругленными углами. Это означает, что при срабатывании перехода Trunk из отмеченной таким образом области удаляются все токены.

```
{ discriminator; in: script;
params: {
  branch1/function, branch2/function, trunk/function,
  entrance1TokenType/type, entrance2TokenType/type,
  auxPlaceTokenType/type, exitTokenType/type };

states:
  {entrance1; token: entrance1TokenType},
  {entrance2; token: entrance2TokenType},
  {auxPlace1; token: auxPlaceTokenType},
  {auxPlace2; token: auxPlaceTokenType},
  {exit; token: exitTokenType};

transitions:
{ Branch1;
  from: entrance1;
  bind_from: {entrance1, in};
  to: auxPlace1;
  bind_to: {auxPlace1, out};
  activity: {in: function;
    params: {+in/entrance1TokenType, -out/auxPlaceTokenType};
    {{branch1(in,out)}};
  }
},
{ Branch2;
  from: entrance2;
  bind_from: {entrance2, in};
  to: auxPlace2;
  bind_to: {auxPlace2, out};
  activity: {in: function;
    params: {+in/entrance1TokenType, -out/auxPlaceTokenType};
    {{branch1(in,out)}
  };
},
{ Trunk;
  from: auxPlace1, auxPlace2;
  bind_from: {auxPlace1, in}, {auxPlace2, in};
  to: exit;
  bind_to: {exit, out};
  activity: {in: function;
    params: {+in/auxPlaceTokenType, -out/exitTokenType };
    {{
        trunk(in, out) &
        isempty(entrance1') & isempty(entrance2') &
        isempty(auxPlace1') & isempty(auxPlace2')
     }};
  };
};
}
```

Исходная модель потоков работ. В качестве исходной модели потоков работ рассмотрим язык YAWL (Yet Another Workflow Language) [164], разработанный

разработанный с целью представления в нем всех образцов. Спецификации потоков работ представляются в YAWL при помощи *расширенных сетей потоков работ* (Extended Workflow Nets — EWF- сетей).

Определение. *Расширенная сеть потока работ* — это набор

$$N = (C , i, o, T, F, split , join, rem)$$

где

– C — множество мест;

– $i \in C$ — входное место;

– $o \in C$ — выходное место;

– T — множество задач,

– $F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\})$ — отношение инцидентности, каждая вершина в графе *(C, T, F)* содержится в направленном пути от i к o;

– $split$: $T \rightarrow \{AND, XOR, OR\}$ — функция, определяющая split-поведение каждой задачи;

– $join$: $T \rightarrow \{AND, XOR\}$ — функция, определяющая join-поведение каждой задачи;

– rem: $T \rightarrow$ POW$(T \cup C \setminus \{i, o\})$ — функция, определяющая дополнительные токены, которые необходимо убрать из части потока работ.

В языке YAWL под произвольной задачей $t \in T$ имеется ввиду сеть определенного вида. При определении образцов без ограничения общности разрешим только один тип подсети, а именно изображенный на рис. А.4. Произвольная задача t на данной диаграмме представлена прямоугольником со скругленными углами. Фактически, она заменяется на переход $enter_t$, место $exec_t$ и переход $exit_t$. Таким образом, мы будем говорить о задаче t или о сети с элементами $enter_t$, $exec_t$, $exit_t$, имея в виду одну и ту же сущность с разной степенью детализации.

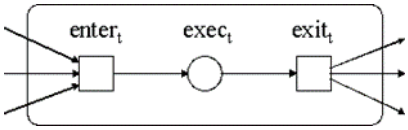

Рисунок А.4 - Структура перехода EWF-сети

Определение. *Состояние потока работ* спецификации

$$N = (C, i, o, T, F, split , join, rem)$$

это мультимножество s над Q, где $Q = C \cup (\cup_{t \in T} exec_t)$.

Состояние s потока работ — это, фактически, мультимножество токенов, находящихся в местах сети из множества C или местах $exec_t$, соответствующих задачам. Переходы на пространстве состояний производятся следующим образом. Когда

наступает время выполнить задачу t, срабатывает соответствующий ей переход $enter_t$. При этом необходимо выполнение определенных условий на наличие токенов во входных местах, в зависимости от того, является ли задача t XOR/AND-join. Например, для поведения AND-join все входные места задачи должны содержать токены. В результате срабатывания перехода $enter_t$ создается токен в месте $exec_t$. Нахождение токена в месте $exec_t$ означает, что задача t находится в стадии исполнения. Переход $exit_t$ срабатывает, когда токен находится в месте $exec_t$. При этом переход удаляет токены из выбранных частей сети в соответствии с функцией *rem*. Число производимых токенов зависит от split-поведения задачи t (AND, OR, XOR). Например, в случае XOR-split единственный токен производится ровно в одном ввыходном месте задачи.

Определение формальной семантики исходной модели потоков работ. Рассмотрим расширенную сеть потока работ $N = (C, i, o, T, F, split , join, rem)$. Такая сеть представляется в AMN конструкцией REFINEMENT с именем *N*.

Множество мест $C = c_1, \ldots , c_n$, вместе с входным и выходным местом сети, представляется в AMN переменной *states* машины *N*. Переменная типизируется в разделе INVARIANT следующим образом:

$$states \in States \rightarrow NAT$$

Здесь множество *States*, определяемое в разделе SETS как

$$States = \{state\_input, state\_output, state\_c_1, \ldots , state\_c_n \}$$

представляет собой множество строковых констант, соответствующих именам мест.

Натуральное число, хранимое в *states*(*state_c*$_i$) отражает количество токенов, содержащихся в месте c_i сети. Переменная *states* инициализируется в разделе INITIALISATION следующим образом.

```
ANY states1
WHERE
  states1 ∈ States → NAT
  ∀ st (st ∈ dom(states1) ⇒ states1(st) = 0)
THEN
  states := states1
END
```

Каждая задача t_i из T представляется в AMN двумя операциями *enter_t*$_i$ и *exit_t*$_i$, а также переменной *exec_t*$_i$ машины *N*. Переменная *exec_t*$_i$ типизируется в разделе INVARIANT следующим образом:

$$exec\_t_i \in NAT$$

Пусть множество входных мест задачи t_i:

$$\{c \mid (c, t_i) \in F\} = \{c_1^{in}, \ldots, c_m^{in}\}$$

Пусть множество выходных мест задачи t_i:

$$\{c \mid (t_i, c) \in F\} = \{c_1^{out}, \ldots, c_k^{out}\}$$

Пусть множество мест, из которых нужно убрать токены по исполнении задачи t_i:

$$rem(t_i) = \{c_1^{rem}, \ldots, c_l^{rem}\} \cup \{t_1^{rem}, \ldots, t_r^{rem}\}$$

где c_j^{rem} из C, t_j^{rem} из T.

Тогда операция *enter_t_i* машины N имеет следующий вид:

```
enter_t_i =
SELECT exec_t_i = 0 ∧ P_enter
THEN
  S_enter ||
  exec_t_i = exec_t_i + 1
END
```

Вид предиката P_{enter} зависит от join-поведения задачи t_i (значения $join(t_i)$):

– если $join(t_i)$ = AND, то P_{enter} есть

```
states(state_c1^in) > 0 ∧ ... ∧ states(state_c1^in) > 0
```

– если $join(t_i)$ = OR, то P_{enter} есть

```
states(state_c1^in) > 0 ∨ ... ∨ states(state_c1^in) > 0
```

– если $join(t_i)$ = XOR, то P_{enter} есть

```
states(state_c1^in) > 0 ∧
states(state_c2^in) = 0 ∧ ... ∧ states(state_cm^in) = 0
∨ ... ∨
states(state_c1^in) = 0 ∧ ... ∧ states(state_cj-1^in) = 0 ∧
states(state_cj^in) > 0 ∧
states(state_cj+1^in) = 0 ∧ ... ∧ states(state_cm^in) = 0 ∧
∨ ... ∨
states(state_c1^in) = 0 ∧ ... ∧ states(state_cm-1^in) = 0 ∧
states(state_cm^in) > 0
```

Вид подстановки S_{enter} также зависит от join-поведения задачи:

– если $join(t_i)$ = AND, то S_{enter} есть

```
states:= states <|-
  { state_c1^in ↦ state_c1^in - 1, … , state_cm^in ↦ state_cm^in - 1 }
```

– если $join(t_i)$ = XOR, то S_{enter} есть

```
IF states(state_c1^in) > 0
THEN states(state_c1^in) := states(state_c1^in) - 1
...
ELSEIF states(state_cm^in) > 0
THEN states(state_cm^in) := states(state_cm^in) - 1
END
```

– если *join*(t_i) = OR, то S_{enter} есть

```
states:= states <|-
  { st, num | st ∈ {state_c1^in, ... , state_cm^in} ∧ states(st) > 0 ∧
              num = states(st) - 1 }
```

Операция *enter_*t_i машины *N* имеет следующий вид:

```
exit_ti =
SELECT exec_ti > 0
THEN
  BEGIN
    Sexit ||
    exec_ti = exec_ti - 1
  END;
  Rexit
END
```

Вид подстановки S_{exit} зависит от split-поведения задачи t_i (значения *split*(t_i)):

– если *split*(t_i) = AND, то S_{exit} есть

```
states:= states <|-
  { state_c1^out ↦ state_c1^out − 1, … , state_ck^out ↦ state_ck^out − 1 }
```

– если *split*(t_i) = XOR, то S_{exit} есть

```
ANY st WHERE st ∈ States
THEN
  states(st) := states(st) − 1
END
```

– если *split*(t_i) = OR, то S_{exit} есть

```
ANY sts WHERE sts ∈ POW(States) ∧ card(sts) > 0
THEN
  states:= states <|- { st, num | st ∈ sts ∧ num = states(st) + 1 }
END
```

Подстановка R_{exit} имеет следующий вид:

```
states:= states <|-
  { st, num | st ∈ { state_c1^rem, ... , state_c1^rem} ∧ num = 0 } ||
exec_t1^rem := 0 || ... || exec_tr^rem := 0
```

Доказательство корректности уточнения расширений канонической процессной модели образцами исходной модели потоков работ. Рассмотрим процесс доказательства корректности уточнения на примере образца потоков работ – дискриминатора.

Родовой тип скрипта дискриминатора, в соответствии с расссмотренными выше принципами определения формальной семантики канонической процессной модели, представляется в AMN следующей конструкцией:

```
REFINEMENT DiscriminatorScript
SETS
    AVAL
CONSTANTS
    Obj,

    ext_entrance1TokenType, extp_entrance1TokenType,
    ext_entrance2TokenType, extp_entrance2TokenType,
    ext_auxState1TokenType, extp_auxState1TokenType,
    ext_auxState2TokenType, extp_auxState2TokenType,
    ext_exitTokenType, extp_exitTokenType

PROPERTIES
    Obj: POW(AVAL) &

    ext_entrance1TokenType: POW(Obj) &
    extp_entrance1TokenType <: ext_entrance1TokenType &
    ext_entrance2TokenType: POW(Obj) &
    extp_entrance2TokenType <: ext_entrance2TokenType &
    ext_auxState1TokenType: POW(Obj) &
    extp_auxState1TokenType <: ext_auxState1TokenType &
    ext_auxState2TokenType: POW(Obj) &
    extp_auxState2TokenType <: ext_auxState2TokenType &
    ext_exitTokenType: POW(Obj) &
    extp_exitTokenType <: ext_exitTokenType

ABSTRACT_VARIABLES
    entrance1, entrance2, auxState1, auxState2, exit

INVARIANT
    entrance1: POW(ext_entrance1TokenType) &
    entrance2: POW(ext_entrance2TokenType) &
    auxState1: POW(ext_auxState1TokenType) &
    auxState2: POW(ext_auxState2TokenType) &
    exit: POW(ext_exitTokenType)

INITIALISATION
    entrance1:= {} || entrance2:= {} ||
    auxState1:= {} || auxState2:= {} ||
    exit:= {}

OPERATIONS

branch1 =
SELECT #tt.(tt: entrance1)
THEN
    ANY tt WHERE tt: entrance1
    THEN
        entrance1:= entrance1 - {tt} ||
        ANY rr WHERE rr: ext_auxState1TokenType
        THEN
            auxState1 := auxState1 \/ {rr}
        END
    END
END;
```

```
branch2 =
SELECT #tt.(tt: entrance2)
THEN
    ANY tt WHERE tt: entrance2
    THEN
        entrance2:= entrance2 - {tt} ||
        ANY rr WHERE rr: ext_auxState2TokenType
        THEN
            auxState2 := auxState2 \/ {rr}
        END
    END
END;

trunk =
SELECT
    #tt.(tt: auxState1 & auxState2 = {}) or
    #tt.(tt: auxState2 & auxState1 = {})
THEN
    ANY tt WHERE
        tt: Obj &
        (tt: auxState1 or tt: auxState2)
    THEN
        auxState1:= auxState1 - {tt} ||
        auxState2:= auxState2 - {tt} ||
        ANY rr WHERE rr: ext_exitTokenType
        THEN
            exit := exit \/ {rr}
        END;
        BEGIN
            entrance1:= {} ||
            entrance2:= {} ||
            auxState1:= {} ||
            auxState2:= {}
        END
    END
END

END
```

В исходной модели образец дискриминатора представляется следующей EWF-сетью:

Discriminator = (C, i, o, T, F, split , join, rem)

C = {enter 1*, enter* 2*, auxState*1*, auxState*2*, exit}*

*T = {branch*1*, branch*2*, trunk}*

F = {enter 1 ∘⟶ *branch*1*, enter* 2 ∘⟶ *branch*2*, branch*1 ∘⟶ *auxSatate*1*,*

*branch*2 ⟶ *auxSatate*2*, auxSatate*1 ⟶ *trunck, auxSatate*2 ⟶ *trunck, trunk* ⟶ *exit }*

split = ∅

join = {trunk ∘⟶ *XOR}*

rem = {trunck ∘⟶ *{enter* 1*, enter* 2*, auxState*1*, auxState*2*, branch*1*, branch*2*}}*

Данная EWF-сеть, в соответствии с расссмотренными выше принципами определения формальной семантики исходной модели потоков работ, представляется в AMN следующей конструкцией:

```
REFINEMENT Discriminator

SETS
    States = { state_enter1, state_enter2, state_auxState1,
               state_auxState2, state_exit }

ABSTRACT_VARIABLES
   states, exec_branch1, exec_branch2, exec_trunk

INVARIANT
    states: States --> NAT &
    exec_branch1: NAT & exec_branch2: NAT & exec_trunk: NAT

INITIALISATION
    ANY states1
    WHERE
       states1: States --> NAT &
       !st.(st: States => states1(st)=0)
    THEN
        states:= states1
    END
    ||
    exec_branch1:= 0 ||
    exec_branch2:= 0 ||
    exec_trunk:= 0


OPERATIONS

enter_branch1 =
SELECT exec_branch1 = 0 & states(state_enter1) > 0
THEN
    states(state_enter1):= states(state_enter1) - 1  ||
    exec_branch1:= exec_branch1 + 1
END;

exit_branch1 =
SELECT exec_branch1 > 0
THEN
    states(state_auxState1):= states(state_auxState1) + 1  ||
    exec_branch1:= exec_branch1 - 1
END;

enter_branch2 =
SELECT exec_branch2 = 0 & states(state_enter2) > 0
THEN
    states(state_enter2):= states(state_enter2) - 1  ||
    exec_branch2:= exec_branch2 + 1
END
;

exit_branch2 =
SELECT exec_branch2 > 0
THEN
    states(state_auxState2):= states(state_auxState2) + 1  ||
    exec_branch2:= exec_branch2 - 1
END;
```

```
enter_trunk =
SELECT exec_trunk = 0 &
    states(state_auxState1) > 0 & states(state_auxState2) = 0  or
    states(state_auxState2) > 0 & states(state_auxState1) = 0
THEN
    IF states(state_auxState1) > 0
    THEN
        states(state_auxState1):= states(state_auxState1) - 1
    ELSE IF states(state_auxState2) > 0
    THEN
        states(state_auxState2):= states(state_auxState2) - 1
    END
    END
    ||
    exec_trunk:= exec_trunk + 1
END;

exit_trunk =
SELECT exec_trunk > 0
THEN
    states(state_exit):= states(state_exit) + 1 ||
    exec_trunk:= exec_trunk -1;
    ANY states1
    WHERE
        states1: States +-> NAT &
        dom(states1) = { state_enter1, state_enter2, state_auxState1,
state_auxState2 } &
        !st.(st: dom(states1) => states1(st) = 0)
    THEN
        states:= states <+ states1
    END
    ||
    exec_branch1:= 0 ||
    exec_branch2:= 0
END

END
```

Последний этап доказательства корректности расширения канонической процессной модели рассмотренным родовым типом скрипта заключается в использовании средств автоматизации (Atelier B) для доказательства уточнения машины *DiscriminatorScript* машиной *Discriminator*.

Для этого необходимо согласовать машины *DiscriminatorScript* и *Discriminator*. Процесс *согласования* состоит из следующих шагов:

– *Согласование направления уточнения.* Необходимо добавить раздел REFINES в машину *Discriminator:*

```
REFINES DiscriminatorScript
```

– *Согласование имен операций уточняемой и уточняющей машин*. Для каждого перехода скрипта *t* соответствующая операция *exit_t* машины *Discriminator* переименовывается в *t,* а в машину *DiscriminatorScript* добавляется пустая операция *enter_t: exit_branch1, exit_branch2, exit_trunk* переименовываются в *branch1, branch2,*

trunk соответственно, в машину *DiscriminatorScript* добавляется пустые операции *enter_branch1, enter_branch2, enter_trunk*:

```
enter_branch1 = skip;
enter_branch2 = skip;
enter_trunk = skip;
```

– *Добавление инварианта уточнения в раздел* INVARIANT *машины Discriminator.*

Инвариант описывает соотношение состояний уточняющей и уточняемой машин:

```
card(entrance1) = states(state_enter1) + exec_branch1 &
card(entrance2) = states(state_enter2) + exec_branch2 &
( card(auxState1) = states(state_auxState1) + exec_trunk &
  card(auxState2) = states(state_auxState2)  or
  card(auxState2) = states(state_auxState2) + exec_trunk &
  card(auxState1) = states(state_auxState1)
) &
card(exit) = states(state_exit)
```

Машины *DiscriminatorScript* и *Discriminator* в согласованном виде были введены в инструментальное средство автоматизации доказательства уточнения AMN (Atelier B 4.7.1). Далее автоматически были сформулированы 96 теорем, в совокупности утверждающие факт уточнения машины *DiscriminatorScript* машиной *Discriminator*. Большое количество теорем объясняется тем, что сложные теоремы автоматически разбиваются инструментальным средством на более простые теоремы, которые можно доказывать независимо. С использованием автоматических средств доказательства было доказано 79 теорем. В таблице А.3 указано общее количество теорем и количество автоматически доказанных теорем.

Таблица А.3 – Число теорем

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic	
		Force Fast	Force 3
Discriminator	96	36	79

А.3 Унификация онтологической модели данных OWL с использованием технологии переписывания термов с верификацией на основе уточнения образцов

В данном разделе приведено краткое описание этапов унификации онтологической модели данных OWL от формального определения синтаксиса до верификации.

Формализация синтаксиса модели OWL. Нормативное определение синтаксиса модели OWL определено в документе [11] в версии расширенной формы Бэкуса–Наура, правила которой имеют следующий вид:

```
axiom ::= ... |
'ObjectProperty(' propertyID
['Deprecated'] { annotation }
{'super(' propertyID ')'}
['inverseOf(' propertyID ')']
['Symmetric']
[ 'Functional' | 'InverseFunctional' |
'Functional' 'InverseFunctional' |
'Transitive' ]
{ 'domain(' classID ')' }
{ 'range(' classID ')' } ')'
```

Здесь символ ::= разделяет голову и тело правила, вертикальная черта (|) означает альтернативу, опциональные части правила заключаются в квадратные скобки [ ], повторяющиеся части заключаются в фигурные скобки { }.

Синтаксис модели формализован с использованием языка SDF. Пример определения синтаксического правила для *объектного свойства* выглядит следующим образом:

```
module unifier/owl/OWL-Syntax
sorts
  Axiom ObjectPropertyAxiom ...
context-free syntax
  ObjectPropertyAxiom -> Axiom
  "ObjectProperty" "(" PropertyID ("Deprecated")? Annotation*
  SuperProperty* InverseOf?  ("Symmetric")? PropertyKind?
  ObjectPropertyDomain*  ObjectPropertyRange* ")"
  -> ObjectPropertyAxiom
```

Полное определение синтаксиса используемого подмножества модели OWL опубликовано в репозитории GitHub[90]. Формализация абстрактного синтаксиса подмножества OWL в виде диаграммы классов UML приведена на рис. А.5.

Формализация семантики модели OWL. Формализация семантики заключается в построении отображения онтологий OWL в спецификации AMN. Ниже кратко проиллюстрированы принципы отображения OWL в AMN на примере части онтологии Глобальной почвенной информационной системы GloSIS:

---

90 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis.Meta-Environment/unifier/owl/OWL-Syntax.sdf

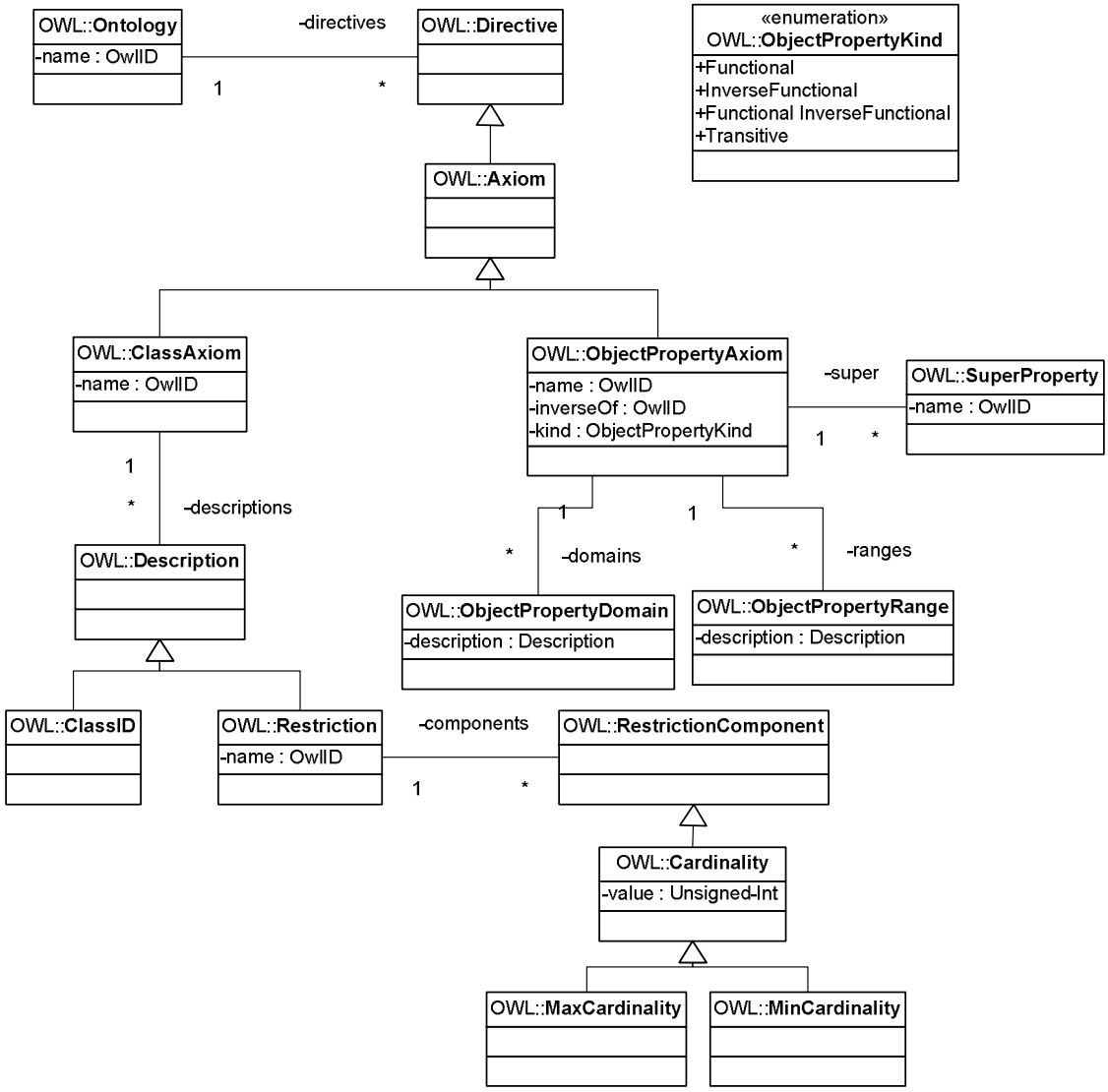

Рисунок А.5 - Абстрактный синтаксис подмножества языка OWL в виде диаграммы классов UML

```
Ontology ( GLoSIS

Class( FeatureOfInterest )
Class( GL_Plot  FeatureOfInterest )
Class( GL_Site  FeatureOfInterest )
Class( ObservableProperty )
Class( Result )
Class( CropClassValueCode  Result)

Individual( cropClassProperty  type(ObservableProperty) )

Individual( Barley  type(CropClassValueCode) )
Individual( Rye  type(CropClassValueCode) )

Class( Observation
restriction( hasResult  minCardinality(1) )
restriction( hasFeatureOfInterest cardinality(1) )
restriction( observedProperty  cardinality(1) )
)
```

```
ObjectProperty( hasResult  domain(Observation)  range(Result) )
ObjectProperty( hasFeatureOfInterest  domain(Observation)
range(FeatureOfInterest) )
ObjectProperty( observedProperty
  domain(Observation)  range(ObservableProperty) )

Class(CropClass
Observation
restriction( hasResult  someValuesFrom( CropClassValueCode ))
restriction( hasFeatureOfInterest
  allValuesFrom( unionOf(GL_Plot  GL_Site) ) )
restriction( observedProperty  value(cropClassProperty) )
)
)
```

Класс *Observation* отвечает наблюдениям – актам оценки или вычисления значения некоторого свойства (класс *ObservableProperty*) объекта интереса (класс *FeatureOfInterest*). Наблюдение приводит к достижению результата (класс *Result*). Примеры объектов интереса – это земельные участки (класс *GL_Site*), для которых производится исследование качества почвы и точки наблюдения (класс *GL_Plot*). Пример результата – код культуры (класс *CropClassValueCode*), в частности, ячмень (*Barley*) и рожь (*Rye*). Частный случай наблюдения – класс культуры (*CropClass*). Среди его результатов обязательно должен быть код культуры; его объектами интереса могут быть только участки и точки наблюдения.

Образом при отображении данной онтологии в AMN является пара спецификаций *SoilContext* и *OWLSoil*:

```
MACHINE  SoilContext
SETS
  Ind
END

REFINEMENT OWLSoil
SEES  SoilContext
ABSTRACT_CONSTANTS
    cropClassProperty, Barley, Rye
PROPERTIES
    cropClassProperty: Ind &
    Barley: Ind &
    Rye: Ind
VARIABLES
    GL_Plot, GL_Site, FeatureOfInterest, ObservableProperty,
    Result, CropClassValueCode, Observation, CropClass,
    hasResult, hasFeatureOfInterest, observedProperty
INVARIANT
    GL_Plot: POW(Ind) & GL_Site: POW(Ind) & FeatureOfInterest: POW(Ind) &
    GL_Plot <: FeatureOfInterest & GL_Site <: FeatureOfInterest &
    ObservableProperty: POW(Ind) & Result: POW(Ind) &
    CropClassValueCode <: Result &
    Observation: POW(Ind) & CropClass <: Observation &
```

```
    hasResult: Observation <-> Result &
    !(obs).(obs: Observation => card(hasResult[{obs}]) >= 1 ) &
    hasFeatureOfInterest: Observation <-> FeatureOfInterest &
    !(obs).(obs: Observation => card(hasFeatureOfInterest[{obs}]) = 1 ) &
    observedProperty: Observation <-> ObservableProperty &
    !(obs).(obs: Observation => card(observedProperty[{obs}]) = 1 ) &

    CropClass <: { xx | xx: Ind &
        #(yy).( (xx|->yy): hasResult  & yy: CropClassValueCode ) } &
    CropClass <: { xx | xx: Ind &
        !(yy).( yy: Ind & (xx|->yy): hasFeatureOfInterest =>
                yy: GL_Plot \/ GL_Site ) } &
    CropClass <: { xx | xx: Ind &
        (xx|->cropClassProperty): observedProperty }
INITIALISATION
    GL_Plot:= {} || GL_Site:= {} || FeatureOfInterest:= {} ||
    ObservableProperty:= { cropClassProperty } ||
    Result:= { Barley, Rye } || CropClassValueCode:= { Barley, Rye } ||
    Observation:= {} || CropClass:= {} || hasResult:= {} ||
    hasFeatureOfInterest:= {} || observedProperty:= {}

OPERATIONS
include_Observation(ind, results, foi, op)=
PRE ind: Ind &
    results: POW(Result) & card(results) >= 1 &
    foi: FeatureOfInterest &
    op: ObservableProperty
THEN
 hasResult:= (hasResult - ({ind}<|hasResult)) \/
             { pp | #(yy).(yy: results & pp = ind|->yy) } ||
 hasFeatureOfInterest(ind) := foi ||
 observedProperty(ind) := op ||
 Observation := Observation \/ {ind}
END;

include_CropClass(ind, results, foi, op)=
PRE ind: Ind &
    results: POW(Result) & card(results) >= 1 &
    #(res).(res: Result & res: CropClassValueCode) &
    foi: FeatureOfInterest & foi: GL_Plot \/ GL_Site &
    op: ObservableProperty & op = cropClassProperty
THEN
 hasResult:= (hasResult - ({ind}<|hasResult)) \/
             { pp | #(yy).(yy: results & pp = ind|->yy) } ||
 hasFeatureOfInterest(ind) := foi ||
 observedProperty(ind) := op ||
 CropClass := CropClass \/ {ind} ||
 Observation := Observation \/ {ind}
END;

set_hasResult(ind, val)=
PRE ind: Observation & val: POW(Result) & card(val) >= 1 &
    (ind: CropClass =>
        #(res).(res: Result & res: val & res: CropClassValueCode))
THEN
 hasResult:= (hasResult - ({ind}<|hasResult)) \/
             { pp | #(yy).(yy: val & pp = ind|->yy) }
END;
```

```
set_featureOfInterest(ind, val)=
PRE ind: Observation & val: FeatureOfInterest &
    (ind: CropClass => val: GL_Plot \/ GL_Site)
THEN
 hasFeatureOfInterest:= hasFeatureOfInterest \/ {ind|->val}
END;

set_observedProperty(ind, val)=
PRE ind: Observation &  val: ObservableProperty &
    (ind: CropClass => val = cropClassProperty)
THEN
 observedProperty:= observedProperty \/ {ind|->val}
END

END
```

Спецификация *OWLSoil* видит (SEES) спецификацию *SoilContext*. Множество всех индивидуализированных объектов (individuals) онтологии представляется в AMN множеством *Ind*. Классы (например, *Observation*) представляются одноименными переменными, типизированными в инварианте как подмножества *Ind*. Объектные свойства (например, *hasResult*) представляются одноименными переменными, типизированными в инварианте как отношения между множествами, представляющими области определения и области значения свойств. Различные ограничения на классы и свойства (ограничения значений и кардинальности свойств, аксиомы эквивалентности классов и т. д.) представляются соответствующими частями инварианта. Каждому классу также ставится в соответствие операция, пополняющая класс новым элементом, а каждому свойству — операция, изменяющая значение этого свойства у некоторого индивидуализированного объекта. Операция изменяет значение свойства у индивидуализированного объекта *ind* на значение *val*, но только в том случае, когда выполняется предусловие операции, гарантирующее выполнение инварианта после выполнения операции.

Для подмножества модели OWL с использованием языков SDF и ASF реализована трансформация в язык AMN.

На языке SDF формализован синтаксис AMN. Например, синтаксическое правило для конструкции REFINEMENT выглядит следующим образом:

```
module unifier/amn/AMN-Syntax
exports
sorts
  Refinement
context-free syntax
  ("REFINEMENT" | "MACHINE") Amn-Id
  ("REFINES" Amn-Id)?
  ("SEES" Amn-Id-List)?
  ("SETS" {Set-Declaration ";"}*)?
```

```
  ("CONSTANTS" Amn-Id-List)?
  ("PROPERTIES" Predicate)?
  ("VARIABLES" Amn-Id-List)?
  ("INVARIANT" Predicate)?
  ("INITIALISATION" Substitution)?
  ("OPERATIONS" {Operation ";"}*)?
  "END"
  -> Refinement
```

Полное определение синтаксиса используемого подмножества языка AMN опубликовано в репозитории GitHub[91].

Трансформация, реализующая отображение подмножества модели OWL в язык AMN, реализована с использованием языков SDF и ASF. На языке SDF определены сигнатуры правил трансформации. Например, сигнатуры основных правил, используемых для создания конструкции REFINEMENT из онтологии выглядят следующим образом:

```
module unifier/owl2synthesis/owl-translator
imports unifier/owl/OWL-Syntax
imports unifier/synthesis/Synthesis-Syntax
imports unifier/Uni-Common

exports
context-free syntax

  create-Refinement(OwlSpecification) -> Refinement
  create-Variables(Directive*) -> {Amn-Id ","}*
  simplify-Predicate(Predicate) -> Predicate
  create-Initialisation(Directive*) -> Substitution
  create-Operations(Directive*) -> {Operation ";"}*
```

В ASF-файле определяются спецификации правил, соответствующие сигнатурам:

```
[Refinement]
Amn-Id := create-Refinement-Name(OntologyID?)
====>
create-Refinement (NamespaceDef* Ontology( OntologyID? Directive*)) =
REFINEMENT Amn-Id
VARIABLES
create-Variables(Directive*)
INVARIANT
simplify-Predicate(simplify-Predicate(create-Invariant(Directive*)))
INITIALISATION
create-Initialisation(Directive*)
OPERATIONS
create-Operations(Directive*)
END
```

91 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis.Meta-Environment/unifier/amn/AMN-Syntax.sdf

Полные спецификации правил для трансформации выбранного подмножества модели OWL в AMN опубликованы в репозитории GitHub[92].

Формализация синтаксиса языка СИНТЕЗ. Синтаксис модели формализован с использованием языка SDF, пример определения синтаксических конструкций приведен в разделе А.1. Полное определение синтаксиса используемого подмножества языка СИНТЕЗ опубликовано в репозитории GitHub[93].

Установление соответствий элементов моделей. Соответствия между элементами моделей установлены следующим образом (Таблица А.4). Названия элементов соответствуют сортам синтаксисов.

Таблица А.4 - Соответствия элементов модели OWL элементам языка СИНТЕЗ

Элемент OWL	**Элемент канонической модели**
OwlID	Synthesis-Id
Ontology	ModuleDef
Ontology.name	ModuleDef.name
Ontology.directives	ModuleDef.type
Ontology.directives	ModuleDef.class_specification
ClassAxiom	Abstract-Type
ClassAxiom.name	Abstract-Type.name
ClassAxiom.descriptions	Abstract-Type.supertype
ClassAxiom.descriptions	Abstract-Type.attributes
Restriction	Attribute-Specification
Restriction.name	Attribute-Specification.name
ObjectPropertyAxiom	Attribute-Specification
ObjectPropertyAxiom.ranges	Attribute-Specification.type
Restriction.components	Attribute-Specification.metaslot
Cardinality	Metaframe-Slot
Cardinality.value	Metaframe-Slot.values
ObjectPropertyAxiom	Association-Metaclass
ObjectPropertyAxiom.name	Association-Metaclass.name
ObjectPropertyAxiom.super	Association-Metaclass.superclass
ObjectPropertyAxiom.inverseOf	Association-Metaclass.inverse
ObjectPropertyAxiom.kind	Association-Metaclass.association_type.metaslot
ObjectPropertyAxiom.domains	Association-Metaclass.domain
ObjectPropertyAxiom.ranges	Association-Metaclass.range

Создание расширения канонической модели. В процессе интеграции эталонных схем OWL и канонической модели было обнаружено, что атрибуту *kind* типа *ObjectPropertyAxiom*, определяющему вид объектного свойства (транзитивное,

---

92 https://github.com/sstupnikov/ModelTransformation/tree/master/OWL2Synthesis.Meta-Environment/unifier/owl2amn

93 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis.Meta-Environment/unifier/synthesis/Synthesis-Syntax.sdf

функциональное и т. д.), а также понятию *ObjectPropertyKind* не соответствуют никакие конструкции канонической модели. В канонической модели понятию объектного свойства (*ObjectPropertyAxiom*) соответствует понятие метакласса ассоциаций (*Association-Metaclass*). Было принято решение о расширении канонической модели новыми метатипами ассоциаций — *Transitive*, *Functional* и т. д. В качестве примера рассмотрим спецификацию метатипа Transitive:

```
{ Transitive;
in: association, metatype;
instance_section: {
domain: type;
range: type;
transitivity: {
in: predicate, invariant;
{{ all a/this.domain.inst, b/this.domain.inst, c/this.domain.inst(
    b/this.range.inst & in([a,b], this) & in([b,c], this) ->
    in([a,c], this)) }}
} } }
```

Транзитивность ассоциаций выражается инвариантом *transitivity* и состоит в следующем: если *a* состоит в ассоциации с *b* и *b* состоит в ассоциации с *c*, то а состоит в ассоциации с *c* (*a, b, c* — значения некоторого типа данных).

Семантику транзитивной ассоциации в AMN рассмотрим на следующем примере. Пусть тип *Region* содержит атрибут *subRegionOf*, принадлежащий метатипу ассоциаций *SubRegionOf* (который является подтипом метатипа транзитивных ассоциаций *Transitive*):

```
{ Region; in: type;
  subRegionOf: {set; type_of_element: Region;};
  metaslot in: SubRegionOf end
}
{ SubRegionOf; in: association, metatype;
  superclass: Transitive;
  instance_section: {
    domain: Region;
    range: Region;
}}
```

Транзитивность ассоциации означает, что если регион *A* является подрегионом *B* и регион *B* является подрегионом *C*, то *A* является подрегионом *C*. При отображении в AMN данная семантика будет выражаться следующим инвариантом:

```
!(a, b, c).(a: ext_Region & b: ext_Region & c: ext_Region &
            a: subRegionOf(b) & b: subRegionOf(c) => a: subRegionOf(c))
```

Здесь *subRegionOf*—переменная, представляющая в AMN соответствующий атрибут, *ext_Region* — множество, представляющее все допустимые экземпляры типа *Region*.

Подобным же образом осуществляется расширение семантики канонической модели для других видов ассоциаций. Список соответствий конструкций OWL и канонической модели расширяется следующим соответствием: *ObjectPropertyAxiom.kind ~ Association-Metaclass.superclass.*

Это соответствие означает, что вид объектного свойства OWL выражается в канонической модели отношением суперкласса между метатипом ассоциаций, выражающим объектное свойство (в примере — *SubRegionOf*), и некоторым встроенным метатипом ассоциаций (в примере — *Transitive*).

Синтаксис определения метаклассов ассоциаций был формализован в виде расширения синтаксиса языка СИНТЕЗ на SDF (ниже приведена часть определения, полное определение опубликовано в репозитории GitHub[94]):

```
module unifier/synthesis/AssociationMetaclassExtension
imports unifier/synthesis/Synthesis-Syntax
exports
sorts
  Association-Metaclass Superclass-List Domain Range Bounds

context-free syntax
  "{" Synthesis-Id ";" "in" ":" "association"
      ("," Metaclass-Name-List)? ";"
      ("superclass" ":"  Superclass-List ";")?
      ("inverse" ":" Metaclass-Name ";")?
      ("instance_section" ":" "{"
           ("association_type" ":" "{" Bounds "," Bounds "}" ";"
            (Attribute-Metaslot)? )?
           ("domain" ":"  Domain ";")?
           ("range" ":"  Range ";")?
       "}" ";" )?
  "}"
  -> Association-Metaclass
```

Конструирование трансформации моделей. Продемонстрируем вербальные правила отображения модели OWL в язык СИНТЕЗ на примере части онтологии GLoSIS, рассмотренной в начале раздела. Ее представление в языке СИНТЕЗ выглядит следующим образом:

```
{ GLoSIS; in: module, ontology;
frame:
{ cropClassProperty; in: ObservableProperty, observableProperty; },
{ Barley; in: CropClassValueCode, cropClassValueCode; },
{ Rye; in: CropClassValueCode, cropClassValueCode; };
```

94 https://github.com/sstupnikov/ModelTransformation/blob/master/OWL2Synthesis.Meta-Environment/unifier/synthesis/AssociationMetaclassExtension.sdf

```
type:
{ GL_Plot; in: type, owl; supertype: FeatureOfInterest; },
{ GL_Site; in: type, owl; supertype: FeatureOfInterest; },
{ FeatureOfInterest; in: type, owl; },
{ ObservableProperty; in: type, owl; },
{ Result; in: type, owl; },
{ CropClassValueCode; in: type, owl; supertype: Result},

{ Observation; in: type, owl;
  hasResult: {set; type_of_element: Result};
  metaslot in: HasResult; min_card: 1; end
  hasFeatureOfInterest: FeatureOfInterest;
  metaslot in: HasFeatureOfInterest; min_card: 1; max_card: 1; end
  observedProperty: ObservableProperty;
  metaslot in: ObservedProperty; min_card: 1; max_card: 1; end
},
{ CropClass; in: type, owl; supertype: Observation;};

class_specification:
{ gl_plot; in: class, owl; superclass: featureOfInterest;
  instance_section: GL_Plot;
},
{ gl_site; in: class, owl; superclass: featureOfInterest;
  instance_section: GL_Site;
},
{ featureOfInterest; in: class, owl;
  instance_section: FeatureOfInterest;
},
{ observableProperty; in: class, owl;
  instance_section: ObservableProperty;
},
{ result; in: class, owl; instance_section: Result;
},
{ cropClassValueCode; in: class, owl; superclass: result;
  instance_section: CropClassValueCode;
},
{ observation; in: class, owl;
  instance_section: { in: type, owl; supertype: Observation;
    inv_hasResult: {  in: predicate, invariant;
      {{ all obs/Observation (in(obs, observation) ->
          obs.hasResult <= result ) }}
    };
    inv_hasFeatureOfInterest: {  in: predicate, invariant;
      {{ all obs/Observation ( in(obs, observation) ->
           in(obs.hasFeatureOfInterest, featureOfInterest)  ) }}
    };
    inv_observedProperty: {  in: predicate, invariant;
      {{ all obs/Observation ( in(obs, observation) ->
           in(obs.observedProperty, observableProperty)  ) }}
    };
  };
},
{ cropClass; in: class, owl;
  instance_section: { in: type, owl; supertype: CropClass;
    restriction_hasResult: {  in: predicate, invariant;
      {{ all cc/CropClass (in(cc, cropClass) ->
           ex res/Result(in(res, cc.hasResult) &
                 in(res, cropClassValueCode)) ) }}
    };
```

```
    restriction_hasFeatureOfInterest: {  in: predicate, invariant;
      {{ all cc/CropClass (in(cc, cropClass) ->
          in(cc.hasFeatureOfInterest, union(gl_plot, gl_site) }}
    };
    restriction_observedProperty: {  in: predicate, invariant;
      {{ all cc/CropClass (in(cc, cropClass) ->
            cc.observedProperty = cropClassProperty }}
    };
  };
};
};
}
```

Классы OWL (например, *Observation*) представляются одноименными типами (*Observation*) и классами (*observation*) канонической модели. Объектные свойства (например, *hasResult*) представляются атрибутами типов. Ограничения значений свойств представляются инвариантами классов (например, *CropClass.restriction_hasResult*).

Вербальные правила отображения формализованы с использованием языков SDF и ASF. В SDF-файле определяются сигнатуры правил переписывания термов. Например, сигнатуры основных правил, необходимых для трансформации онтологии в модуль языка СИНТЕЗ выглядят следующим образом:

```
module unifier/owl2synthesis/owl-translator
imports unifier/owl/OWL-Syntax
imports unifier/synthesis/AssociationMetaclassExtension
exports
context-free syntax
  t-Module-Def(OwlSpecification) -> Module-Def
  t-Module-Name(OntologyID?) -> Synthesis-Id
  t-Type-Specification-List(Directive*, Directive*) ->
    {Type-Specification ","}*
  t-Class-Declarator-List(Directive*) -> {Class-Declarator ","}*
```

В данном примере функция *t-Module-Def* преобразует онтологию OWL в модуль канонической модели, функция *t-Type-Specification-List* преобразует список директив в список типов, функция *t-Class-Declarator-List* преобразует список директив в список классов.

В ASF-файле определяются спецификации правил, соответствующие сигнатурам:

```
[Module-Def]
Synthesis-Id := t-Module-Name(OntologyID?),
Type-Specification* := t-Type-Specification-List(Directive*, Directive*),
Class-Declarator* := t-Class-Declarator-List( Directive* )
====>
t-Module-Def( NamespaceDef* Ontology( OntologyID? Directive*) ) =
{ Synthesis-Id; in: module, ontology;
type:
Type-Specification*;
class_specification:
Class-Declarator*; }
```

Приведенное правило описывает действие функции *t-Module-Def*. В правиле использованы рекурсивные вызовы функций преобразования списка директив в списки типов и классов. Полные спецификации правил для выбранного подмножества модели OWL опубликованы в репозитории GitHub[95].

Верификация уточнения моделью OWL расширенной канонической модели. Ниже рассматривается верификация отображения OWL в язык СИНТЕЗ на примере одной схемы OWL, а именно — части онтологии GLoSIS, рассмотренной выше. Для данной онтологии уже рассмотрены спецификация на OWL, семантика в AMN и отображение в каноническую модель. Осталось рассмотреть отображение канонической спецификации в AMN и верификацию уточнения AMN-спецификаций. Результат отображения канонической спецификации в AMN выглядит следующим образом):

```
MACHINE  SoilContext
SETS
  AVAL
ABSTRACT_CONSTANTS
  Obj
PROPERTIES
  Obj: POW(AVAL)
END

REFINEMENT CanonicalSoil
SEES SoilContext
ABSTRACT_CONSTANTS
  ext_GL_Plot, ext_GL_Site, ext_FeatureOfInterest, ext_ObservableProperty,
  ext_Result, ext_CropClassValueCode, ext_Observation, ext_CropClass,
  cropClassPropertyCan, BarleyCan, RyeCan
PROPERTIES
  ext_GL_Plot: POW(Obj) & ext_GL_Site: POW(Obj) &
  ext_FeatureOfInterest: POW(Obj) & ext_ObservableProperty: POW(Obj) &
  ext_Result: POW(Obj) & ext_CropClassValueCode: POW(Obj) &
  ext_Observation: POW(Obj) & ext_CropClass: POW(Obj) &
  ext_GL_Plot <: ext_FeatureOfInterest &
  ext_GL_Site <: ext_FeatureOfInterest &
  ext_CropClassValueCode <: ext_Result &
  ext_CropClass <: ext_Observation &
  cropClassPropertyCan: ext_ObservableProperty &
  BarleyCan: ext_CropClassValueCode &
  RyeCan: ext_CropClassValueCode
VARIABLES
  gl_plot, gl_site, featureOfInterest, observableProperty,
  result, cropClassValueCode, observation, cropClass,
  hasResultCan, hasFeatureOfInterestCan, observedPropertyCan
INVARIANT
  gl_plot: POW(ext_GL_Plot) & gl_site: POW(ext_GL_Site) &
```

95 https://github.com/sstupnikov/ModelTransformation/tree/master/OWL2Synthesis.Meta-Environment/unifier/owl2synthesis

```
  featureOfInterest: POW(ext_FeatureOfInterest) &
  observableProperty: POW(ext_ObservableProperty) &
  result: POW(ext_Result) &
  cropClassValueCode: POW(ext_CropClassValueCode) &
  observation: POW(ext_Observation) &
  cropClass: POW(ext_CropClass) &
  gl_plot <: featureOfInterest & gl_site <: featureOfInterest &
  cropClassValueCode <: result & cropClass <: observation &
  hasResultCan: ext_Observation +-> POW(ext_Result) &
  hasFeatureOfInterestCan: ext_Observation +-> ext_FeatureOfInterest &
  observedPropertyCan: ext_Observation +-> ext_ObservableProperty &
  !(obs).(obs: dom(hasResultCan) => card(hasResultCan(obs)) >= 1) &
  !(obs).(obs: ext_Observation & obs: observation =>
      obs: dom(hasResultCan)) &
  !(obs).(obs: ext_Observation & obs: observation =>
      obs: dom(hasFeatureOfInterestCan)) &
  !(obs).(obs: ext_Observation & obs: observation =>
      obs: dom(observedPropertyCan)) &
  !(obs).(obs: ext_Observation & obs: observation =>
      hasResultCan(obs) <: result)  &
  !(obs).(obs: ext_Observation & obs: observation =>
      hasFeatureOfInterestCan(obs): featureOfInterest) &
  !(obs).(obs: ext_Observation & obs: observation =>
      observedPropertyCan(obs): observableProperty) &
  !(cc).(cc: ext_CropClass & cc: cropClass =>
      #(res).(res: ext_Result & res: hasResultCan(cc) &
            res: cropClassValueCode ) ) &
  !(cc).(cc: ext_CropClass & cc: cropClass =>
      hasFeatureOfInterestCan(cc): (gl_plot \/ gl_site) ) &
  !(cc).(cc: ext_CropClass & cc: cropClass =>
      observedPropertyCan(cc) = cropClassPropertyCan)

INITIALISATION
  gl_plot := {} || gl_site := {} || featureOfInterest := {} ||
  observableProperty := {cropClassPropertyCan} ||
  result := {BarleyCan, RyeCan} ||
  cropClassValueCode := {BarleyCan, RyeCan} ||
  observation := {} || cropClass := {} || hasResultCan := {} ||
  hasFeatureOfInterestCan := {} || observedPropertyCan := {}

OPERATIONS
include_Observation(ind, results, foi, op)=
PRE ind: ext_Observation &
    results: POW(ext_Result) & results <: result &
    results: FIN(results) & card(results) >= 1 &
    foi: ext_FeatureOfInterest & foi: featureOfInterest &
    op: ext_ObservableProperty & op: observableProperty
THEN
 hasResultCan(ind) := results ||
 hasFeatureOfInterestCan(ind) := foi ||
 observedPropertyCan(ind) := op ||
 observation := observation \/ {ind}
END;

include_CropClass(ind, results, foi, op)=
PRE ind: ext_Observation &
    results: POW(ext_Result) & results <: result &
    results: FIN(results) & card(results) >= 1 &
    #(res).(res: ext_Result & res: results & res: cropClassValueCode) &
    foi: ext_FeatureOfInterest & foi: featureOfInterest &
```

```
    foi: gl_plot \/ gl_site &
    op: ext_ObservableProperty & op: observableProperty &
    op = cropClassPropertyCan
THEN
 hasResultCan(ind) := results ||
 hasFeatureOfInterestCan(ind) := foi ||
 observedPropertyCan(ind) := op ||
 cropClass := cropClass \/ {ind} ||
 observation := observation \/ {ind}
END;

set_hasResult(ind, val)=
PRE ind: ext_Observation & ind: observation &
    val: POW(ext_Result) & val <: result & val: FIN(val) & card(val) >= 1 &
    (ind: ext_CropClass & ind: cropClass =>
        #(res).(res: ext_Result & res: val & res: cropClassValueCode))
THEN
 hasResultCan(ind) := val
END;

set_featureOfInterest(ind, val)=
PRE ind: ext_Observation & ind: observation &
    val: ext_FeatureOfInterest & val: featureOfInterest &
    (ind: ext_CropClass & ind: cropClass => val: gl_plot \/ gl_site)
THEN
 hasFeatureOfInterestCan(ind):= val
END;

set_observedProperty(ind, val)=
PRE ind: ext_Observation & ind: observation &
    val: ext_ObservableProperty & val: observableProperty &
    (ind: ext_CropClass & ind: cropClass => val = cropClassPropertyCan)
THEN
 observedPropertyCan(ind):= val
END

END
```

Множество всех значений абстрактных типов данных представляется в AMN множеством *AVAL*, множество всех значений объектных типов данных — его подмножеством *Obj*. Типы представляются множествами всех своих допустимых значений — экстенсионалами (например, тип *Observation* представляется экстенсионалом *ext_Observation*). Экстенсионалы объектных типов типизируются в секции PROPERTIES как подмножества *Obj*. Отношение тип–подтип представляется отношением множество–подмножество на экстенсионалах. Классы (например, *observation*) представляются одноименными переменными, типизированными в инварианте как подмножества экстенсионалов типов своих экземпляров. Отношение класс–подкласс представляется отношением множество–подмножество на соответствующих переменных.

Атрибуты типов (например, *hasResult*) представляются одноименными переменными (с постфиксом *Can*), типизированными в инварианте как функции,

определенные на экстенсионале типа. Различные ограничения на классы и свойства (ограничения значений и кардинальности свойств, аксиомы эквивалентности классов и т. д.) представляются соответствующими частями инварианта. Каждому классу также ставится в соответствие операция, пополняющая класс новым элементом, а каждому атрибуту—операция, изменяющая значение этого свойства у некоторого объекта.

Для связи состояния спецификаций *OWLSoil* и *CanonicalSoil* был определен и добавлен в спецификацию *OWLSoil* склеивающий инвариант:

```
GL_Plot = gl_plot & GL_Site = gl_site &
FeatureOfInterest = featureOfInterest &
ObservableProperty = observableProperty &
Result = result &  CropClassValueCode = cropClassValueCode &
Observation = observation & CropClass = cropClass  &

!(ind, val).(ind: Ind & val: POW(Result) =>
    ((hasResult[{ind}] = val) <=> (hasResultCan(ind) = val))) &
!(ind, val).(ind: Ind & val: FeatureOfInterest =>
    (((ind|->val): hasFeatureOfInterest) <=>
         (hasFeatureOfInterestCan(ind) = val))) &
!(ind, val).(ind: Ind & val: ObservableProperty =>
    (((ind|->val): observedProperty) <=> (observedPropertyCan(ind) = val)))
```

Спецификации AMN, отвечающие онтологии GLoSIS в OWL и ее образу при отображении в каноническую модель, были введены в инструментальное средство автоматизации доказательства уточнения AMN Atelier B 4.7.2 (спецификации размещены в репозитории GitHub[96]). Далее автоматически были сформулированы 88 теорем, в совокупности утверждающие факт уточнения спецификаций. Большое количество теорем объясняется тем, что сложные теоремы автоматически подразделяются инструментальным средством на более простые, которые можно доказывать независимо. С использованием автоматических средств доказательства было доказано 52 теоремы. В табл. А.5 указано общее количество теорем и количество автоматически доказанных теорем.

Таблица А.5 – Количество теорем

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic				
		Force Fast	Force 0	Force 1	Force 2	Force 3
OWLSoil.ref	88	22	42	46	50	52

96 https://github.com/sstupnikov/ModelTransformation/tree/master/OWL2Synthesis.Meta-Environment/unifier/owl-examples/Soil

А.4 Унификация онтологической модели данных OWL с верификацией на основе эквивалентного представления в единой семантической структуре (дополнение)

Определение *взаимооднозначных соответствий* между элементами моделей, не приведенных в основном тексте работы, выглядит следующим образом.

Отображение отношений между классами. Отношение класс-подкласс в простых случаях (например, *sensor* – подкласс *event*) представляется в языке СИНТЕЗ в слоте *superclass* соответствующего подкласса. Более сложные отношения между классами (например, *featureOfInterest* $\subseteq$ *event* $\cup$ *object* $\cup$ *informationEntity*) представляются инвариантами классов (*featureCompleteness*). В OWL отношения между классами представляются при помощи аксиом классов:

СИНТЕЗ	OWL

{ sensor; in: class;
superclass: event;
}
 	
{ featureOfInterest; in: class;
instance_section: {
featureCompleteness: {
in: invariant;
{ featureOfInterest <=
union(event,
union(object,
informationEntity))
}
};
}; }
	SubClassOf(sensor event)
	
	SubClassOf(
	featureOfInterest
	ObjectUnionOf(
	event
	object
	informationEntity)
	)

Отображение фактов. Факты (утверждения о значениях свойств объектов) представляются в языке СИНТЕЗ константами объектных типов. В OWL факты представляются утверждениями о существовании индивида (*NamedIndividual*), о принадлежности индивида классу (*ClassAssertion*), о значениях свойств (*DataPropertyAssertion, ObjectPropertyAssertion*) и т.д.:

СИНТЕЗ	OWL
{ "Observation/346344"; in: observation; resultTime: moment(2017#06#06 12:36:12); madeBySensor: "sensor/35-207306- 844818-0/BMP282"; }	NamedIndividual(Observation/346344) ClassAssertion(observation Observation346344) DataPropertyAssertion(resultTime Observation/346344 "2017-06-06T12:36:12Z"^^xsd:dateTime) ObjectPropertyAssertion(madeBySensor Observation346344 sensor/35-207306-844818-0/BMP282)

Отображение самоограничений. Пример инварианта класса языка СИНТЕЗ (*autoRegulatingProcess*), описывающего ситуацию, когда объект согласно некоторому свойству (*regulates*) соотносится сам с собой (авторегулирующийся процесс регулирует сам себя), приведен ниже. В OWL данная ситуация описывается конструкцией *ObjectHasSelf*:

СИНТЕЗ	OWL

`all p/AutoRegulatingProcess (`
`  is_in(autoRegulatingProcess, p) ->`
`  is_in(p.regulates, p) )`
	`SubClassOf(autoRegulatingProcess`
	`  ObjectHasSelf(regulates))`

Здесь *all* обозначает квантор всеобщности, знак / - типизацию переменной типом, предикат *is_in* – принадлежность элемента множеству, знак –> - логическую импликацию.

Отображение ограничений на размер области и класс значений свойств. Ниже приводится пример инварианта класса (*observation*), утверждающий, что среди результатов (свойство *hasResult*) любого наблюдения должен быть по крайней мере один образец (*sample*). В OWL данная ситуация представляется конструкцией *ObjectMinCardinality*:

СИНТЕЗ	OWL

`all o/Observation (`
`  is_in(observation, o) ->`
`  cardinal({r/Result |`
`  is_in(o.hasResult, r) &`
`  is_in(sample, r)})`
`  >= 1 )`
	`SubClassOf( observation`
	`  ObjectMinCardinality(`
	`  1 hasResult sample) )`

Здесь функция *cardinal* вычисляет мощность множества.

Отображение рефлексивных, иррефлексивных и асимметричных свойств. Для представления свойств специального вида в языке СИНТЕЗ определяются соответствующие метаклассы ассоциаций. Так, для представления рефлексивных свойств определяется метакласс ассоциаций *reflexive*:

```
{ reflexive; in: association, metaclass;
  instance_section: {
   reflexivity: { in: invariant;
     all x ( (is_in(this.domain, x) -> is_in(this, [x, x]))
            & (is_in(this.range, x) -> is_in(this, [x, x])) )
} } }
```

Здесь *this* обозначает класс ассоциаций, принадлежащий метаклассу *reflexive*, *domain* обозначает область определения класса ассоциаций, *range* – область значения, *[a, b]* обозначает пару объектов, участвующих в ассоциации. Метакласс каждого рефлексивного свойства становится подклассом *reflexive* (например, *PartOf,* свойство

«быть частью»). В OWL рефлексивность отражается при помощи конструкции *ReflexiveObjectProperty*:

СИНТЕЗ	OWL

`{ PartOf;`
` in: association, metaclass;`
` superclass: reflexive;`
`}`
	`ReflexiveObjectProperty(partOf)`

Аналогично отображаются иррефлексивные и асимметричные свойства.

Отображение непересекающихся свойств. Ниже рассмотрен пример инварианта класса (*process*), описывающего непересекающиеся свойства (*startTime* и *endTime*). В OWL непересекающиеся свойства представляются конструкциями *DisjointObjectProperties* и *DisjointDataProperties:*

СИНТЕЗ	OWL

`all p/Process(`
`  is_in(process, p) ->`
`  p.startTime <> p.endTime )`
	`DisjointDataProperties(`
	`  startTime  endTime`
	`)`

Отображение иерархии композиций свойств. Ниже рассмотрен пример композиции свойств (*locatedIn* и *partOf*), являющейся подсвойством некоторого свойства (*locatedIn*). Если хозяйство *f* находится в регионе *r* и регион *r* является частью региона *s*, то *f* находится в *s*. Иерархия композиций свойств выражается в OWL при помощи комбинации конструкций *SubPropertyOf* и *ObjectPropertyChain*:

СИНТЕЗ	OWL

`all f/FarmUnit, r/Region, s/Region (`
`  is_in(f, c) &`
`  is_in(region, r) &`
`  is_in(region, s) &`
`  is_in(f.locatedIn, r) &`
`  is_in(r.partOf, s) ->`
`  is_in(f.locatedIn, s )`
	`SubPropertyOf(`
	`  ObjectPropertyChain(`
	`  locatedIn  partOf )`
	`  locatedIn )`

Отображение ключей. Ключи представляются в языке СИНТЕЗ при помощи встроенных утверждений уникальности атрибутов (*unique*). В OWL ключи представляются при помощи конструкции *HasKey*:

СИНТЕЗ	OWL

`{ FarmUnit; in: type;`
`  unitCode: integer;`
`  keys: { unique; { unitCode } };`
`}`
	`HasKey( farmUnit  unitCode )`

Семантика конструкций языка СИНТЕЗ в семантической структуре языка OWL, т.е. условия того, что интерпретация $I = \langle \Delta_I, \Delta_D, \bullet^C, \bullet^{OP}, \bullet^{DP}, \bullet^I, \bullet^{DT}, \bullet^{LT} \rangle$ удовлетворяет

онтологическому модулю *O,* не приведенные в основном тексте работы, выглядят следующим образом.

Условие для свойств объектов в случае, когда метакласс ассоциаций свойства является подклассом некоторого специального метакласса (например, метакласс *PartOf* свойства *partOf* является подклассом метакласса *reflexive*), состоит в том, что свойство должно удовлетворять всем инвариантам суперкласса (свойство *partOf* должно быть рефлексивным):

Конструкция СИНТЕЗ	Условие

{ PartOf;
in: association, metaclass;
superclass: reflexive;
instance_section: {
domain: region;
range: region;
};
}
{ region; in: class;
instance_section: {
partOf: {set;
type_of_element: Region;};
metaslot in: PartOf end
};
}
}
	$\forall r$ (
	$r \in (region)^C \rightarrow$
	$(r, r) \in (partOf)^{OP}$)

Условия для фактов.

Конструкция СИНТЕЗ	Условие

{ "Observation/346344";
in: observation;
resultTime: moment(2017#06#06 12:36:12);
madeBySensor: "sensor/35-207306-844818-0/BMP282";
}
	$("Observation/346344")^I \in (observation)^C \wedge$
	$(("Observation/346344")^I, (2017\#06\#06\ 12:36:12)^{LT}) \in (resultTime)^{DP} \wedge$
	$(("Observation/346344")^I, ("sensor/35-207306-844818-0/BMP282")^I) \in (madeBySensor)^{OP}$

Условия для подклассов.

Конструкция СИНТЕЗ	Условие
{ sensor; in: class; superclass: event; }	$(sensor)^C \subseteq (event)^C$

Условия для утверждений уникальности.

СИНТЕЗ	OWL
{ FarmUnit; in: type; unitCode: integer; keys: { unique; { unitCode } }; }	$\forall x, y, k$ ($(x, k) \in (unitCode)^{DP} \wedge (y, k) \in (unitCode)^{DP} \rightarrow$ $x = y$)

Условия для инвариантов.

Конструкция СИНТЕЗ	Условие
`featureOfInterest <= union(event, union(object, informationEntity))`	$(featureOfInterest)^C \subseteq (event)^C \cup (object)^C \cup (informationEntity)^C$
`all p/AutoRegulatingProcess ( is_in(autoRegulatingProcess, p) -> is_in(p.regulates, p) )`	$\forall p\ (\ (p)^I \in (autoRegulatingProcess)^C \rightarrow ((p)^I, (p)^I) \in (regulates)^{OP})$
`cardinal({r/Result | is_in(o.hasResult, r) & is_in(sample, r)}) >= 1`	$\#\{r \mid (o, r) \in (hasResult)^{OP} \wedge r \in (sample)^C\} \geq 1$
`p.startTime <> p.endTime`	$\forall h, f\ ((p, h) \in (startTime)^{DP} \wedge (p, f) \in (endTime)^{DP} \rightarrow h \neq f)$

Сопоставление условий для удовлетворения интерпретации онтологии канонической модели и условий удовлетворения интерпретации онтологии OWL, не рассмотренные в основном тексте работы, выглядит следующим образом.

Условия для отношений между классами:

Условия для СИНТЕЗ	Условия для OWL
$(sensor)^C \subseteq (event)^C \wedge (featureOfInterest)^C \subseteq (event)^C \cup (object)^C \cup (informationEntity)^C$	$(sensor)^C \subseteq (event)^C$ $(featureOfInterest)^C \subseteq (event)^C \cup (object)^C \cup (informationEntity)^C$

Условия для фактов:

Условия для СИНТЕЗ	Условия для OWL
$(\text{"Observation/346344"})^I \in (observation)^C \wedge ((\text{"Observation/346344"})^I, (2017\#06\#06\ 12:36:12)^{LT}) \in (resultTime)^{DP} \wedge ((\text{"Observation/346344"})^I, (\text{"sensor/35-207306-844818-0/BMP282"})^I) \in (madeBySensor)^{OP}$	$(Observation/346344)^I \in (observation)^C \wedge ((Observation/346344)^I, (\text{"2017-06-06T12:36:12Z"}\text{\^\^}xsd:dateTime)^{LT}) \in (resultTime)^{DP} \wedge ((Observation/346344)^I, (sensor/35-207306-844818-0/BMP282)^I) \in (madeBySensor)^{OP}$

Условия для самоограничений:

Условия для СИНТЕЗ	Условия для OWL
$\forall p\ (\ (p)^I \in (autoRegulatingProcess)^C \rightarrow ((p)^I, (p)^I) \in (regulates)^{OP})$	$(autoRegulatingProcess)^C \subseteq \{ x \mid (x , x) \in (boss)^{OP}\}$

Условия для ограничений на размер области и класс значений свойств:

Условия для СИНТЕЗ	Условия для OWL
$\forall r\ (\ r \in (observation)^C \rightarrow \#\{r \mid (o, r) \in (hasResult)^{OP} \wedge r \in (sample)^C\} \geq 1)$	$(observation)^C \subseteq \{ x \mid \#\{ y \mid (x , y) \in (hasResult)^{OP} \wedge y \in (sample)^C\} \geq 1 \}$

Условия для рефлексивных свойств:

Условия для СИНТЕЗ	Условия для OWL
$\forall r\ (r \in (region)^C \rightarrow (r, r) \in (partOf)^{OP})$	$\forall x\ (x \in (region)^C \rightarrow (x, x) \in (partOf)^{OP})$

Условия для непересекающихся свойств:

Условия для СИНТЕЗ	Условия для OWL
$\forall h, f\ ((p, h) \in (startTime)^{DP} \wedge (p, f) \in (endTime)^{DP} \rightarrow h \neq f)$	$(startTime)^{DP} \cap (endTime)^{DP} = \varnothing$

Условия для иерархии композиций свойств:

Условия для СИНТЕЗ	Условия для OWL
$\forall f, r, s\ (f \in (farmUnit)^C \wedge r \in (region)^C \wedge s \in (region)^C \wedge (f, r) \in (locatedIn)^{OP} \wedge (r, s) \in (partOf)^{OP} \rightarrow (f, s) \in (locatedIn)^{OP})$	$\forall y_0, y_1, y_2\ ((y_0, y_1) \in (locatedIn)^{OP} \wedge (y_1, y_2) \in (partOf)^{OP} \rightarrow (y_0, y_2) \in (locatedIn)^{OP})$

Условия для утверждений уникальности:

Условия для СИНТЕЗ	Условия для OWL
$\forall x, y, k\ ((x, k) \in (unitCode)^{DP} \wedge (y, k) \in (unitCode)^{DP} \rightarrow x = y)$	$\forall x, y, w\ (x \in (farmUnit)^C \wedge y \in (farmUnit)^C \wedge (x, k) \in (unitCode)^{DP} \wedge (y, k) \in (unitCode)^{DP} \rightarrow x = y)$

Можно видеть, что конъюнкции условий для СИНТЕЗ и условий для OWL эквивалентны. В простейших случаях условия просто совпадают (например, отношения между классами) или совпадают с точностью до переменных.

А.5 Унификация графовой модели данных (дополнение)

Раздел основан на работах автора [89] [60] [93] [58].

В данном разделе (таблица А.6) приведено отображение между конструкциями подмножества ЯМД канонической модели данных (языка СИНТЕЗ) и конструкциями подмножества ЯМД графовой модели данных Cypher, опущенные в основном тексте работы. Числа, строки, булевские значения, идентификаторы отображаются без изменений. Символ *m[•]* обозначает семантическую функцию отображения конструкций графовой модели в конструкции канонической модели.

Таблица А.6 – Отображение ЯМД канонической и графовой моделей данных

Конструкция объектной модели данных	**Конструкция Property Graph**
Математические операции	
`+, -, *, /, mod`	`+, -, *, /, %`
Встроенные предикаты	
`=, <>, <, >, <=, >=`	`=, <>, <, >, <=, >=`
Выражения	

Вызов атрибутной функции переменной АТД `v.a`	`v.a`
Множества. `[ "a", "b" ]`	*Коллекции.* `["a", "b"]`
Образцы секции MATCH	

`propertyGraphVertices(a) &`
`propertyGraphVertices(b) &`
`propertyGraphEdges(e) &`
`e.isValidEdge(a, b)`
	`a-[e]->b`

`propertyGraphVertices(a) &`
`propertyGraphVertices(b) &`
`propertyGraphEdges(e) &`
`e.isValidUndirectedEdge(a, b)`
	`a-[e]-b`

`propertyGraphVertices(a) &`
`propertyGraphVertices(b) &`
`RelType(r) &`
`r.isValidEdge(a, b)`
	`a-[r: RelType]->b`

`propertyGraphVertices(a) &`
`propertyGraphVertices(b) &`
`(RT1(r) | RT2(r)) &`
`r.isValidEdge(a, b)`
	`a-[r: RT1 | RT2]->b`

`return(n) :-` `propertyGraphVertices(n).`	`MATCH (n)` `RETURN n`
WHERE	
`A & B | ^C`	`m[A] and m[B] or not m[C]`
`s1.like(s2)`	`s1 =~ /s2/`

`propertyGraphVertices(n) &`
`propertyGraphVertices(m) &`
`propertyGraphEdges(r) &`
`r.isValidEdge(n, m) &`
`r.type.name.like("K.*")`
	`MATCH (n)-[r]->(m)`
	`WHERE type(r) =~ '/K.*/'`

`n.belt <> none`	`has(n.belt)`
`r = none`	`r is null`
`is_in(["Peter", "Tobias"], a.name)`	`a.name IN ["Peter", "Tobias"]`
RETURN	
`q([anage: age]):-` `propertyGraphVertices(a/[age])`	`RETURN a.age AS anage`
Aggregation	

`q([c]) :-`
`propertyGraphVertices(n`
`metavalue queryStartVertex; end) &`
`propertyGraphVertices(x) &`
`propertyGraphEdges(e) &`
`n.id = 2 &`
	`MATCH (n{id: 2})-->(x)`
	`RETURN count(x)`

c = count(x).	
ORDER BY	
q(n) :- order_by({name}, propertyGraphVertices(n/[name])).	MATCH (n) RETURN n ORDER BY n.name
CREATE	
+propertyGraphVertices(n/[name, title]):- n.name = “Andres” & n.title = “Developer”.	CREATE n = {name: 'Andres', title: 'Developer'}
+R(r) :- propertyGraphVertices(a) & propertyGraphVertices(b) & r.isValidEdge(a, b) & r.name = a.name + '<->' + b.name.	CREATE a-[r: R {name: a.name + '<->' + b.name }]->b
DELETE	
-propertyGraphVertices(n) :- n.id = 4.	MATCH (n{id: 4}) DELETE n
-propertyGraphVertices(n) & -propertyGraphEdges(r) :- propertyGraphVertices(m) & r.isValidUndirectedEdge(n, m) & n.id = 3.	MATCH (n{id: 3}) MATCH n-[r]-m DELETE n, r
return(andres) :- propertyGraphEdges(andres) & andres.id = 3 & andres.set_age(none).	MATCH (andres{id: 3}) DELETE andres.age RETURN andres
SET	
return(n) :- propertyGraphEdges(n) & n.id = 2 & n.set_surname(“Taylor”).	MATCH (n{id: 2}) SET n.surname = 'Taylor' RETURN n
Functions	
all x/propertyGraphVertices.inst (is_in(p.nodes, x) -> x.age > 30)	ALL (x in nodes(p) WHERE x.age > 30)
ex x/string (is_in(a.array, x) & x = "one")	ANY(x in a.array WHERE x = "one")
all x/propertyGraphVertices.inst (is_in(p.nodes, x) -> not x.age = 25)	NONE(x in nodes(p) WHERE x.age = 25)
cardinal(p)	length(p)
a.id	ID(a)
first(a.array)	HEAD(a.array)
last(a.array)	LAST(a.array)
p.nodes()	NODES(p)
{ anage | exists n (is_in(p.nodes(), n) &	EXTRACT(n in nodes(p) : n.age)

`anage = n.age) }`	
`{ x | is_in(a.array, x) & cardinal(x) =3 }`	`FILTER(x in a.array : length(x) = 3)`
Mathematical function	
`abs(a.age - c.age)`	`ABS(a.age - c.age)`
`realint(3.141592)`	`ROUND(3.141592)`
`sqrt(256)`	`SQRT(256)`

A.6 Унификация тензорной модели данных ADM

Раздел основан на работах автора [88][145][58].

В данном разделе приведено краткое описание этапов унификации тензорной модели данных ADM, поддерживаемой СУБД SciDB, от формального определения синтаксиса до верификации.

Формализация синтаксиса тензорной модели. Абстрактный синтаксис модели ADM был формализован в метамодели Ecore[97]. Список элементов абстрактного синтаксиса приведен на рисунке А.7. Элементы ЯОД образуют пакет *AQLDDL*, элементы ЯМД образуют пакет *AQLDML*.

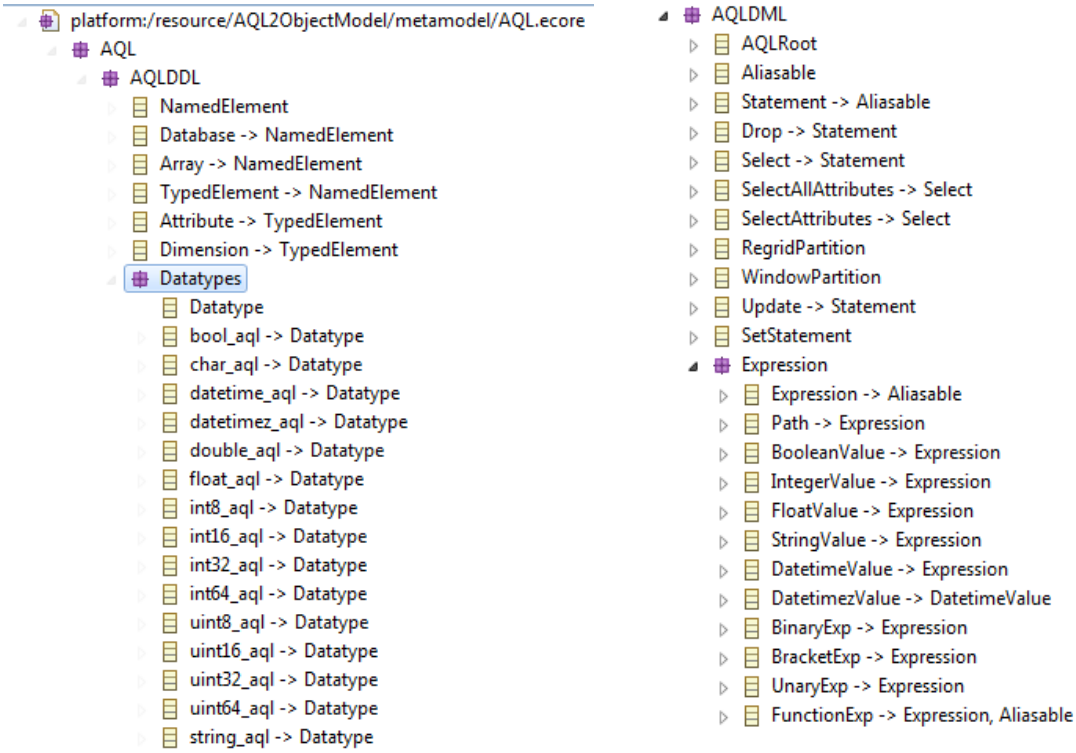

Рисунок А.7 – Элементы абстрактного синтаксиса модели ADM

[97] https://github.com/sstupnikov/ModelTransformation/blob/master/AQL2ObjectModel/metamodel/AQL.ecore

Установление соответствий элементов моделей. Соответствия элементов моделей приведены в таблицах А.7, А.8.

Таблица А.7 – Соответствия элементов ЯОД модели ADM и канонической объектной модели

Элемент QL DDL	Элемент объектной модели
D t b s	M dul D f
D t b s .n m	M dul D f.n m
D t b s . rr s	M dul D f.cl ss s
Arr	Cl ssD f, ADTD f
Arr .n m	Cl ssD f.n m
Arr . ttribut s	Cl ssD f.inst nc T p . ttribut s
Arr .dim nsi ns	Cl ssD f.inst nc T p . ttribut s
`Attribute`	Attribut D f
`Attribute.name`	Attribut D f.n m
`Attribute.type`	Attribut D f.t p
`Dimension`	Attribut D f
`Dimension.name`	Attribut D f.n m
`Dimension.type`	Attribut D f.t p
string_ ql	StringD f
b l_ ql	B l nD f
d t tim _ ql	Tim D f
d ubl _ ql	R lD f
fl t_ ql	R lD f
int8_ ql	Int g rD f
int16_ ql	Int g rD f
int32_ ql	Int g rD f
int64_ ql	Int g rD f
uint8_ ql	Int g rD f
uint16_ ql	Int g rD f
uint32_ ql	Int g rD f
uint64_ ql	Int g rD f

Основные принципы отображения модели данных ADM в каноническую объектную модель данных. СУБД SciDB поддерживает два языка для работы с массивами: AQL (Array Query Language) и AFL (Array Functional Language). AQL является SQL-подобным декларативным языком, включающим как операции ЯОД, так и операции ЯМД. AFL представляет собой функциональный язык манипулирования массивами, операции которого можно объединять в композиции. Допускается использование операций AFL в запросах AQL.

Операции языков и отображение будут иллюстрироваться на адаптированных примерах из сценария применения SciDB в области оптической астрономии, а также на простых примерах из документации SciDB [156].

Таблица А.8 – Соответствия элементов ЯМД модели ADM и канонической объектной модели

Элемент ЯМД объектной модели	Элемент ?QL D? L
Rul?, At?m	S?l?ctAttribut?s
Rul?.h??d.t?rms->t?p?.?l?m?nts, At?m.t?rms->t?p?.?l?m?nts	S?l?ctAttribut?s.?ttribut?s
Rul?.b?d?.f?rmul?	S?l?ctAttribut?s.wh?r?, S?l?ctAttribut?s.fr?mCl?us?, S?l?ctAttribut?s.j?in, S?l?ctAttribut?s.j?inOn
At?m.s?mb?l	S?l?ctAttribut?s.fr?mCl?us?(P?th)->c?mp?n?nts
R?ductEl?m?ntD?f, V?ri?bl?	P?th
R?ductEl?m?ntD?f.p?th->?ttribut?.n?m?, V?ri?bl?.n?m?	P?th.c?mp?n?nts
R?ductEl?m?ntD?f.n?m?	P?th.?li?s
Functi?nC?ll	Functi?nExp
Functi?nC?ll.n?m?	Functi?nExp.n?m?
Functi?nC?ll.t?rms	Functi?nExp.?rgum?nts
Multiplic?tiv?OpC?ll, Additiv?OpC?ll, R?l?ti?nPr?dic?t?	Bin?r?Exp
Multiplic?tiv?OpC?ll.n?m?, Additiv?OpC?ll.n?m?, R?l?ti?nPr?dic?t?.s?mb?l	Bin?r?Exp.s?mb?l
Multiplic?tiv?OpC?ll.t?rms, Additiv?OpC?ll.t?rms, R?l?ti?nPr?dic?t?.t?rms	Bin?r?Exp.l?ftExp, Bin?r?Exp.rightExp
Br?ck?tV?lu?	Br?ck?tExp
Br?ck?tV?lu?.t?rms	Br?ck?tExp.?xp
StringV?lu?D?f	StringV?lu?
StringV?lu?D?f.v?lu?	StringV?lu?.v?lu?
R??lV?lu?D?f	Fl??tV?lu?
R??lV?lu?D?f.v?lu?	Fl??tV?lu?.v?lu?
IntV?lu?D?f	Int?g?rV?lu?
IntV?lu?D?f.v?lu?	Int?g?rV?lu?.v?lu?
B??l??nV?lu?D?f	B??l??nV?lu?
B??l??nV?lu?D?f.v?lu?	B??l??nV?lu?.v?lu?

Пример основной операция ЯОД ADM – создание массива – выглядит следующим образом:

```
CREATE ARRAY source
< ampExposureId: int64 NULL, filterId: int8, apMag: double >
[ ra(double), de(double), objectId=0:*];
```

Создается массив оптических источников *source,* измерениями которого являются координаты *ra* и *de* типа *double* и целочисленный идентификатор объекта. Для целочисленного измерения указаны его нижняя (0) и верхняя («*», обозначающая бесконечность) границы. Ячейка массива состоит из трех атрибутов: *ampExposureId, filterId, apMag*. Указано, что атрибут *ampExposureId* может принимать неопределенное

значение NULL. В данном примере приведены только некоторые из реально используемых атрибутов и измерений.

В канонической модели (языке СИНТЕЗ) создание массива представляется определением одноименного класса:

```
{ source; in: class;
  instance_type:{
  double ra;
  ra2long: {in: function; params: {-ret/long}; };
  double de;
  de2long: {in: function; params: {-ret/long}; };
  long objectId; metaslot lower: 0; higher: inf; end
  objectIdBounds: {in: invariant;
    {{ all s (source(s) -> s.objectId >= 0) }}
  };
  long ampExposureId;
  short filterId;
  double apMag;
  key: { unique; { ra, de, objectId } };
  definiteness: {obligatory;
    { ra, de, objectId, filterId, apMag } };
  };
}
```

Как измерения, так и атрибуты, составляющие ячейку, представляются в языке СИНТЕЗ атрибутами типа экземпляров (*instance_type*) класса. Между встроенными типами ADM (*int8, int64, double* и т. д.) и встроенными типами языка СИНТЕЗ (*short, long, double*) устанавливается взаимно однозначное соответствие. Совокупность атрибутов, соответствующих измерениям, объявляется уникальной (инвариант *key,* выражаемый встроенным утверждением *unique*). Объявляется также, что атрибуты, соответствующие измерениям и не-NULL атрибутам ADM, должны быть определены у всех экземпляров класса (инвариант *definiteness,* выражаемый встроенным утверждением *obligatory*).

Таким образом обеспечивается сохранение отличительных свойств многомерных массивов («кубов данных»), существенным образом различающих измерения и атрибуты, составляющие ячейку:

– по набору значений измерений однозначно определяется набор значений атрибутов ячейки (уникальность измерений);

– ячейка массива всегда определяется полным набором значений измерений (определенность измерений).

Заметим также, что отсутствие в коллекции объекта с некоторым набором значений измерений означает *пустую ячейку* в массиве.

Для нецелочисленных измерений *ra* и *de* в языке СИНТЕЗ кроме атрибутов определяются функции *ra2long, de2long*, преобразующие нецелочисленные значения в

целочисленные. Необходимость привнесения этих функций вызвана следующим. При попытке описать операции, характерные для тензорных моделей, в объектной модели (в частности, в языке СИНТЕЗ) выясняется необходимость применения принципиально различных механизмов работы с целочисленными и нецелочисленными измерениями. Это вызвано различием типов измерений, возможной неравномерностью шага измерения и т. д. Для того чтобы обеспечить возможность единообразного описания операций над целочисленными и нецелочисленными измерениями и необходимы функции, приводящие нецелочисленные измерения к целочисленным.

Ограничения, связанные с нижними и верхними границами целочисленных измерений, представляются в языке СИНТЕЗ, во-первых, метаслотом соответствующего атрибута (например, *objectId*). В метаслоте хранится метаинформация, связанная с атрибутом как с отдельной сущностью языка. В данном случае метаслот включает два слота *lower* и *higher*, отвечающих соответственно верхней и нижней границе измерения. Во-вторых, создается инвариант (например, *objectIdBounds*), предикативная спецификация которого устанавливает ограничения на значения измерения для каждого из объектов класса, отвечающего массиву. Спецификация инварианта имеет вид формулы первого порядка, где *all* – квантор существования, «->» – импликация.

Необходимо отметить, что массив представляется в объектной модели множеством объектов класса (фактически, кортежей значений атрибутов). При этом наблюдается некоторое противоречие со стремлением создателей тензорных моделей отойти от моделей, основанных на кортежах. Однако в контексте интеграции источников данных тензорные модели – это лишь один класс из большого множества разнообразных классов моделей данных. Представление специфических тензорных моделей в объектной модели является методологически и технически гораздо более простым и естественным, нежели использование многомерных массивов в качестве канонической модели.

Изложенные принципы отображения ЯОД могут быть обобщены на случай, когда канонической является объектная или объектно-реляционная модель, отличная от языка СИНТЕЗ. Также не принципиален выбор модели данных, основанной на многомерных массивах. В общем виде принципы отображения ЯОД выглядят следующим образом:

– массив отображается в коллекцию типизированных объектов (класс) объектной модели;

– измерения и атрибуты, составляющие ячейку массива, отображаются в атрибуты типа экземпляров класса;

– между встроенными типами модели, основанной на многомерных массивах, и встроенными типами объектной модели устанавливается взаимно однозначное соответствие;

– совокупность атрибутов, соответствующих измерениям, объявляется уникальной (при помощи механизма ключей, утверждений или инвариантов);

– атрибуты, соответствующие измерениям и не-NULL атрибутам ячейки массива, объявляются определенными (при помощи утверждений или инвариантов);

– для нецелочисленных измерений в типе экземпляров дополнительно определяются методы, преобразующие нецелочисленные значения в целочисленные;

– ограничения, связанные с нижними и верхними границами целочисленных измерений, отображаются при помощи инвариантов или встроенных утверждений о кардинальности соответствующих атрибутов. В случае использования инвариантов при отображении, границы измерений представляются также метаданными атрибута.

При *отображении ЯМД* в данном разделе рассматривается подмножество языка запросов (программ) СИНТЕЗ. Формальная грамматика подмножества выглядит следующим образом:

```
<value> ::= <attribute identifier> | <typed variable> | <boolean value> |
<number> | <string>
<rule> ::= <atom> :- <body>.
<body> ::= <formula>
<formula> ::= <atomic formula> | <formula> & <formula>
<atomic formula> ::= <atom> | <term> <relation predicate> <term>
<atom> ::= <collection name>(<typed variable>)
<typed variable> ::= <variable identifier> / <type expression>
<type expression> ::= <type identifier> '[' <reduct element list> ']'
<reduct element> ::= [<rename identifier> :] <attribute identifier>
<term> ::= <arithmetic expression>
<arithmetic expression>::= <value> | <function designator> | (<arithmetic
expression>) |
<arithmetic expression> <arithmetic operation> <arithmetic expression>
<arithmetic operation> ::= + | - |* | /
<function designator> ::= <function name>(<term list>)
```

Программа рассматривается как набор конъюнктивных запросов (правил) вида:

$q(x/T) :- C_1(x_1/T_1) \ \& \ \ldots \ \& \ C_n(x_n/T_n) \ \& \ (X_1, Y_1) \ \& \ \ldots \ \& \ F_m(X_m, Y_m) \ \& \ B.$

Тело запроса представляет собой конъюнкцию предикатов-коллекций, функциональных предикатов и ограничения. Здесь C_i – имена коллекций (классов), F_j – имена функций, x_i – имена переменных, значения которых пробегают по классам, T_i – типы переменных, X_j и Y_j – входные и выходные параметры функций, B – ограничение,

налагаемое на x_i, X_j, Y_j. Предикаты, находящиеся в голове правил, могут быть использованы в телах других правил в качестве предикатов-коллекций.

Ниже будет использоваться запись предиката-коллекции вида *source([ra, de])*. Неформально это означает, что представляют интерес не объекты класса *source* целиком, а лишь их атрибуты *ra, de.* Формально запись означает сокращение от *source(_/source.inst[ra, de])*. Здесь знак _ обозначает анонимную переменную, *source.inst* – анонимный тип экземпляров (instance) класса *source*, а *ra, de* – необходимые атрибуты типа экземпляров класса.

Будет также использоваться запись *source([i, j, val1/val])*, означающая переименование атрибута *val* в *val1*.

При отображении ЯМД будут сначала рассмотрены основные конструкции языка программ СИНТЕЗ, соответствующие конструкциям языка AQL. Затем будут рассмотрены конструкции СИНТЕЗ, соответствующие конструкциям языка AFL.

Начнем рассмотрение с конструкций языка СИНТЕЗ, соответствующих конструкциям языка AQL, связанных с *извлечением* данных.

Предикаты-классы, условия, подзапросы. Рассмотрим программу, извлекающую координаты (*ra, de*) и апертурную звездную величину (*apMag*) астрономических источников из класса *source* с условием на фильтр (*filterId*) и апертурную звездную величину, причем запрос *q* использует результаты запроса *r*:

```
q([ra, de, apMag]) :- r([ra, de, apMag]) & filterId= #filterId.
r([ra, de, apMag]) :- source([ra, de, apMag]) & apMag >= #apMag.
```

Здесь *#filterId* и *#apMag* – некоторые константы соответствующих типов.

Такая программа представляется в AQL следующим запросом:

```
SELECT apMag FROM ( SELECT apMag FROM source WHERE apMag >= #apMag )
WHERE filterId = #filterId;
```

Простые условия отображаются в AQL без изменений, рекурсивные запросы представляются вложенными запросами. Заметим, что координаты *ra, de* не указываются в секции SELECT – они являются измерениями и извлекаются по умолчанию.

Соединение классов. Соединение по определенным атрибутам (например, *objectId*)

```
q2([ra, de, filterId, uMag]) :-
  source([ra, de, objectId, fliterId]) &
  objectSummary([objectId, uMag]).
```

представляется в AQL конструкцией JOIN-ON:

```
SELECT filterId, uMag INTO q2
FROM source
JOIN objectSummary
ON source.objectId = objectSummary.objectId;
```

где массив *objectSummary* имеет вид

```
CREATE ARRAY objectSummary
<uMag: float NULL,  gMag: float NULL>
[ objectId=0:* ];
```

Агрегация. Рассмотрим запрос, возвращающий объекты с минимальной звездной величиной *uMag*:

```
q([objectId, uMag]) :-
  objectSummary(obj/[objectId, uMag]) & uMag = min(obj.uMag).
```

Запрос представляется в AQL с использованием агрегирующей функции того же рода:

```
SELECT uMag
FROM source, (SELECT min(uMag) AS min FROM Source)
WHERE uMag = min;
```

Группирование. Рассмотрим запрос, возвращающий среднее значение звездной величины *uMag,* вычисленное на группе по идентификатору объекта *filterId:*

```
q([objectId, avgMag]) :-
  group_by({objectId}, q2(obj/[ra,de,filterId, uMag])) &
  avgMag = average(obj.uMag).
```

Здесь коллекция *q2,* на которой производится группирование по атрибуту *objectId* – результат соединения классов *source* и *objectSummary,* рассмотренных выше.

Очевидно, в AQL запрос представляется при помощи конструкции GROUP BY:

```
SELECT avg(uMag) AS avgMag
FROM q2 GROUP BY objectId;
```

Рассмотрим конструкции языка СИНТЕЗ, соответствующие конструкциям языка AQL и связанные с *изменением* данных.

Обновление. Рассмотрим запрос, изменяющий значения в квадратной матрице (см. предыдущий пример) на значения с обратным знаком в том случае, если модуль значения больше 5:

```
source(x/[i, j, val]) :-
  source(x/[i, j, val1/val]) & abs(val) > 5 & val = -val1.
```

В AQL данный запрос представляется следующим образом:

```
UPDATE source SET val = -val WHERE abs(val) > 5;
```

Удаление. Рассмотрим программу, удаляющую из базы данных класс и все его содержимое:

```
-source(x) :- source(x).;
delete_frame(source).
```

В правилах со знаком минус в голове осуществляется удаление объектов из коллекции. В данном случае из коллекции удаляются все объекты. Функция *delete_frame* удаляет коллекцию как объект из базы данных. Операция «;» обозначает последовательную композицию программ. В AQL данный запрос представляется при помощи операции *DROP*:

```
DROP ARRAY source;
```

Рассмотрим принципы отображения конструкций языка СИНТЕЗ, соответствующих конструкциям AFL, на примере *расширения элементов массива в подмассивы*. Каждый элемент массива расширяется в подмассив определенного размера. Значения всех ячеек подмассива дублируют значение оригинальной ячейки. Пример программы, расширяющей каждую ячейку матрицы 3×3 в подматрицу 2×2:

```
q([i, j, val]) :- {x/[i, j, val] | exists y (
  source(y/[i1/i, j1/j, val]) &
  ( i = i1*2 & j = j1*2 | i = i1*2 +1 & j = j1*2 |
  i= i1*2 & j= j1*2 + 1 | i= i1*2 +1 & j= j1*2 +1) ) }.
```

Здесь выражение *{x/T | F(x)}*, где *F* – формула со свободной переменной *x*, обозначает конструкцию выделения множества; *exists* обозначает квантор существования.

В ADM запрос представляется с использованием операции *xgrid*:

```
SELECT * FROM xgrid(source, 2, 2);
```

Можно заметить, что операция AFL *xgrid* имеет достаточно сложно устроенный прообраз в канонической модели (это справедливо и для многих других операций). Между тем эти операции являются естественными для массивов. Поэтому при интеграции источников данных, основанных на многомерных массивах, в канонической модели возможно использование специального класса *array*, инкапсулирующего специфические операции, характерные для многомерных массивов:

```
{ array; in: class;
  instance_type: {
  xgrid: { in: function;
    params: {
     +dimensions/{sequence; type_of_element: string;},
     +scales/{sequence; type_of_element: integer;}};
  };  };
}
```

В приведенном примере рассмотрена сигнатура единственной операции *xgrid*, параметрами которой являются последовательность имен измерений *dimensions* и последовательность масштабов увеличения по каждому из измерений *scales*. Параметром операции по умолчанию также считается класс *array* как коллекция

объектов. При отображении ЯОД каждый класс – образ массива (например, класс *source* из раздела 2.1) становится подклассом класса *array*:

```
{ source; in: class; superclass: array;
  instance_type: { … };
}
```

Аналогично *xgrid*, операциями класса *array* могут быть представлены такие операции AFL, как *substitute, sort, multiply* и т. д.

Заметим, что решение о представлении операций, характерных для многомерных массивов, в рамках специального класса канонической модели допускает обобщение на объектные канонические модели, отличные от языка СИНТЕЗ, и модели, основанные на многомерных массивах, отличные от ADM.

Конструирование трансформации моделей. Рассмотренное выше отображение было реализовано в виде трансформаций на языке ATL: *AQLDDL2ObjectModel* для ЯОД и *ObjectModelDML2AQL* для ЯМД. Трансформации размещены в репозитории GitHub[98]. Так, например, правило трансформации определения массива в определение АТД выглядит следующим образом:

```
rule Array{
  from arr: AQL!Array
  to cl: Synthesis!ClassDef(
    name <- arr.name,
    instanceType <- instType
  ),
  instType: Synthesis!ADTDef(
    name <- arr.name + 'InstanceType',
    attributes <- arr.attributes,
    attributes <- arr.dimensions
    )
}
```

Формализация семантики моделей. Семантика канонической модели представляется конструкцией REFINEMENT языка AMN:

```
REFINEMENT ObjectDM
```

как это показано в разделе «Верификация на основе уточнения AMN-спецификаций моделей» основной части работы. Здесь же приведем пример одной

[98] https://github.com/sstupnikov/ModelTransformation/tree/master/AQL2ObjectModel

операции для доказательства уточнения, а именно – операции *update* обновления значений атрибута в объектах класса:

```
OPERATIONS
update(cls, attr, exp, cond) =
PRE cls: classNames &
    attr: typeAttributes(instanceType(cls)) &
    attributeType(attr) = Integer &
    exp: INT --> INT & cond: NAT --> BOOL
THEN
  integerAttributeValue :=
  integerAttributeValue <+
  { xx | xx: (NAT*(NAT<->INT)) &
     #(oo, val).( oo: objectsOfClass(cls) & val: INT &
         xx = attr |-> ({oo |-> val}) &
     (cond(integerAttributeValue(attr)(oo)) = TRUE =>
         val = exp(integerAttributeValue(attr)(oo))) &
     (cond(integerAttributeValue(attr)(oo)) = FALSE =>
          val = integerAttributeValue(attr)(oo))  ) }
END
```

Параметрами операции являются имя класса *cls*, имя целочисленного атрибута *attr* типа экземпляров класса, функция *exp*, отвечающая за преобразование атрибута, и функция *cond*, отвечающая условию на значение атрибута. Пусть *o* – некоторый объект класса *cls*, для которого определено значение атрибута *attr* и это значение есть *v*. Тогда операция *update* изменяет значение атрибута на *exp(v)* в случае, если выражение *cond(v)* принимает значение «истина», и оставляет значение атрибута без изменений в противном случае. Очевидно, такая операция *update* есть обобщение примера обновления, рассмотренного выше при описании отображения моделей. Действительно, для рассмотренного примера *cls* есть *source, attr* есть *val*, *exp(v) = -v, cond(v) = abs(v).*

Спецификация, выражающая семантику модели ADM, представляется в языке AMN конструкцией

```
REFINEMENT ArrayDM
```

Переменные, составляющие пространство состояний объектной модели, объявлены в разделе ABSTRACT_VARIABLES машины *ArrayDM*:

```
ABSTRACT_VARIABLES
    arrayNames, dimensionNames, cellAttributeNames,
    arrayDimensions, arrayCellAttributes,
    cellAtrributeType, nullable,
    dimLowerBound, dimHigherBound,
    cells, dimensionValue, integerCellAttributeValue
```

Имена массивов представлены переменной *arrayNames*; имена измерений – переменной *dimensionNames*; имена атрибутов ячеек массива – переменной *cellAttributeNames*; принадлежность измерений массивам – переменной *arrayDimensions*;

принадлежность атрибутов ячеек – переменной *arrayCellAttributes*; тип атрибута ячейки – переменной *cellAtrributeType*; атрибуты ячеек массивов, которые могут принимать неопределенные значения, – переменной *nullable*; верхние (нижние) границы измерений – переменной *dimLowerBound* (*dimHigherBound*); множества идентификаторов ячеек массивов – переменной *cells*, значения измерений в ячейках – переменной *dimensionValue*; значения атрибутов ячеек – переменной *integerCellAttributeValue*. Переменные типизируются в разделе INVARIANT при помощи частичных и тотальных функций аналогично переменным, использующимся для придания семантики объектной модели:

```
INVARIANT
    arrayNames: POW(STRING_Type) &
    dimensionNames: NAT +-> STRING_Type &
    cellAttributeNames: NAT +-> STRING_Type &
    arrayDimensions: arrayNames --> POW(dom(dimensionNames)) &
    arrayCellAttributes: arrayNames -->
        POW(dom(cellAttributeNames)) &
    cellAtrributeType:
       dom(cellAttributeNames) --> BuiltInTypes &
    nullable: dom(cellAttributeNames) --> BOOL &
    dimLowerBound: dom(dimensionNames) --> INT &
    dimHigherBound: dom(dimensionNames) +-> INT &
    cells: arrayNames --> POW(NAT) &
    dimensionValue: NAT*dom(dimensionNames) +-> INT  &
    integerCellAttributeValue:
        NAT*dom(cellAttributeNames) +-> INT
```

Здесь «*» – знак декартова произведения множеств.

Дополнительные необходимые свойства переменных состояния представлены конъюнктивными компонентами инварианта. Так, любая ячейка любого массива однозначно идентифицируется набором значений измерений:

```
!(arr, cell1, cell2).(arr: arrayNames &
  cell1: cells(arr) &  cell2: cells(arr) &
  !(dim).(dim: arrayDimensions(arr) =>
    dimensionValue(cell1, dim) = dimensionValue(cell2, dim)) =>
    cell1 = cell2)
```

Для любой ячейки любого массива определены значения всех измерений и значение по крайней мере одного атрибута:

```
!(arr, cell).(arr: arrayNames & cell: cells(arr) =>
  !(dim).(dim: arrayDimensions(arr) =>
    (cell |-> dim): dom(dimensionValue)) &
    #(attr).(attr: arrayCellAttributes(arr) &
      cellAtrributeType(attr) = Integer &
      (cell, attr): dom(integerCellAttributeValue)) )
```

Аналогично объектной модели рассмотрена единственная операция ЯМД – операция обновления *update*:

```
OPERATIONS
update(cls, attr, exp, cond) =
PRE cls: arrayNames & attr: arrayCellAttributes(cls) &
  cellAtrributeType(attr) = Integer &
  exp: INT --> INT & cond: NAT --> BOOL
THEN
  integerCellAttributeValue :=
  integerCellAttributeValue <+
  { yy | yy: (NAT*NAT)*INT &
    #(cell, val).(cell: cells(cls) & val: INT &
    yy = ((cell |-> attr)|-> val) &
    (cond(integerCellAttributeValue(cell, attr)) = TRUE =>
     val = exp(integerCellAttributeValue(cell, attr))) &
    (cond(integerCellAttributeValue(cell, attr))= FALSE  =>
     val = integerCellAttributeValue(cell, attr)) ) }
END
END
```

Сигнатура операции совпадает с сигнатурой операции объектной модели. Семантика операции также аналогична: значение *v* атрибута *attr* массива *cls* заменяется на *exp(v)*, если значение *cond(v)* есть «истина», и не изменяется в противном случае.

Верификация уточнения моделью ADM канонической модели. Для формального доказательства того, что машина *ArrayDM* уточняет машину *ObjectDM*, необходимо построить *инвариант уточнения*, связывающий переменные машин, и добавить его к инварианту уточняющей машины.

Инвариант формализует принципы отображения ЯОД, изложенные в данном разделе выше, и объединяет их в одну конъюнкцию.

Так, множество имен массив совпадает с множеством имен классов:

```
classNames = arrayNames
```

Множество идентификаторов и имен измерений и атрибутов ячеек совпадает с множеством идентификаторов и имен атрибутов типов экземпляров классов:

```
attributeNames = dimensionNames \/ cellAttributeNames
```

Любому измерению любого массива соответствует атрибут типа экземпляра класса, соответствующего этому массиву:

```
!(arr, dim).(arr: arrayNames & dim: arrayDimensions(arr) =>
    #(attr).(attr: typeAttributes(instanceType(arr)) &
             attr = dim & attributeType(attr) = Integer) )
```

Любому атрибуту ячейки любого массива соответствует атрибут типа экземпляра класса, соответствующего этому массиву, и типы атрибутов совпадают:

```
!(arr, cattr).(arr: arrayNames &
   cattr: arrayCellAttributes(arr) =>
    #(attr).(attr: typeAttributes(instanceType(arr)) &
         attr = cattr & attributeType(attr) = attributeType(cattr)))
```

Атрибут ячейки массива, который может принимать неопределенные значения, соответствует определенному (*obligatory*) атрибуту типа:

```
!(arr, cattr).(arr: arrayNames & cattr /: dom(nullable) &
    cattr: arrayCellAttributes(arr) =>
    cattr: obligatory(instanceType(arr)) )
```

Здесь знак «/:» обозначает отношение непринадлежности элемента множеству.

Измерения соответствуют уникальным атрибутам типов:

```
!(arr, dim).(arr: arrayNames &
    dim: arrayDimensions(arr) => dim: unique(instanceType(arr)) )
```

Верхние (нижние) границы измерений равны верхним (нижним) границам соответствующих атрибутов типов:

```
!(dim).(dim: dom(dimLowerBound) =>
    dim: dom(intAttributeLowerBound) &
    dimLowerBound(dim) = intAttributeLowerBound(dim))
```

Непустые ячейки массивов соответствуют объектам классов:

```
cells = objectsOfClass
```

Для любой ячейки значения ее измерений и определенных атрибутов совпадают со значениями соответствующих атрибутов объекта, соответствующего ячейке:

```
!(cell, dim).(cell: NAT & dim: NAT &
  (cell |-> dim): dom(dimensionValue) =>
  cell: dom(integerAttributeValue(dim)) &
  dimensionValue(cell, dim) = integerAttributeValue(dim)(cell))
&
!(cell, cattr).(cell: NAT & cattr: NAT &
    (cell |-> cattr): dom(integerCellAttributeValue) =>
    cell: dom(integerAttributeValue(cattr)) &
    integerCellAttributeValue(cell, cattr) =
      integerAttributeValue(cattr)(cell) )
```

Для указания того, что машина *ArrayDM* уточняет машину *ObjectDM*, в машину *ArrayDM* была добавлена директива

```
REFINES ObjectDM
```

Спецификации *ObjectDM* и *ArrayDM* вместе с инвариантом уточнения[99] были загружены в инструментальное средство Atelier B 4.7.1. Автоматически были сгенерированы теоремы, выражающие уточнение спецификаций (таблица А.9). В частности, для спецификации *ArrayDM* были сгенерированы 112 теорем, 91 из них были

[99] AMN-спецификации размещены в репозитории GitHub: https://github.com/sstupnikov/ModelTransformation/tree/master/AQL2ObjectModel/semantics

доказаны автоматически, для доказательства остальных необходимо применять интерактивные средства доказательства.

Таблица А.9 - Количество сгенерированных и автоматически доказанных теорем непротиворечивости и уточнения спецификаций

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic		
		Force Fast	Force 0	Force 3
ObjectDM.ref	137	41	99	113
ArrayDM.ref	112	40	89	91

Результаты, изложенные в данном разделе, могут быть обобщены и использованы при интеграции в системах, использующих каноническую модель, отличную от языка СИНТЕЗ, например, другую объектную (ODMG) или объектно-реляционную модель (SQL:2003). Результаты также могут быть использованы для интеграции источников данных, представленных в модели, основанной на многомерных массивах, но отличной от ADM.

А.7 Унификация языка логических правил RIF-FLD (дополнение)

Подмножество *конкретного синтаксиса канонической модели данных* (языка СИНТЕЗ), допускающее взаимное отображение между канонической моделью и языком RIF-FLD, выглядит следующим образом:

```
<module> ::=
{ <identifier>; in: module, program;
[import:  <identifier> [, <identifier>]… ;]
[ frame: <frame> [, <frame>]… ; ]
}

<frame> ::= { <identifier>; [in: <identifier> [, <identifier>]… ;] [<slot>
;]… }
| { <identifier>; in: program; {{ <rule and fact list> }} }

<slot> ::= <identifier> : <value> [, <value>]… ;

<value> ::= <identifier> | <variable> | <boolean value> | <number> |
<string> | <constant of a moment> | <constant of interval> | <constant of
sequence>

<rule> ::= <head> :- <body>.
<fact> ::= <atomic formula>.

<head> ::= <atom> [& <atom>]…
<body> ::= <formula>
```

```
<formula> ::=
<atomic formula>
| (<formula>)
| <formula> & <formula>
| <formula> '|' <formula>
| ex <variable list> (<formula>)

<atomic formula> ::= <atom>
| <term> = <term>

<atom> ::= <predicate name>([<term list>])
<predicate name> ::= <collection name> | <function name>

<term> ::= <value> | <arithmetic expression>

<arithmetic expression>::= <variable> | <function designator> | <number> |
| (<arithmetic expression>) | <arithmetic expression> <arithmetic
operation> <arithmetic expression>
<arithmetic operation> ::= + | - |* | /

<function designator> ::= <function name>(<term list>)
```

Описание используемых нетерминальных символов грамматики приведено в [42].

Соответствия между элементами абстрактного синтаксиса языка СИНТЕЗ[100] и RIF-BLD[101] как подмножества RIF-FLD представлены в таблице А.10.

Таблица А.10 - Соответствия между элементами СИНТЕЗ и RIF-BLD

Элемент модели СИНТЕЗ	**Элемент модели RIF-BLD**

`ModuleDef`
`ModuleDef.name`
`ModuleDef.containedPrograms`
`ModuleDef.imports`
	`Document`
	`Document.annotation.frame->assignment.value`
	`Document.group->contents`
	`Document.import`

`LogicProgram` `LogicProgram.program(RuleList)->rules`	`Group` `Group->content`

`Rule`
`Rule.head`
`Rule.body(SimpleRuleBody).formula`
	`Implies`
	`Implies.left`
	`Implies.right`

`Atom`
`Atom.symbol`
`Atom->terms`
	`PositionalAtom, NamedArgumentAtom, Member, Frame`
	`PositionalAtom.termula, NamedArgumentAtom.termula`
	`PositionalAtom.arg, NamedArgumentAtom.arg`

[100] https://github.com/sstupnikov/ModelTransformation/blob/master/Synthesis/metamodel/Synthesis.ecore

[101] https://github.com/sstupnikov/ModelTransformation/blob/master/RIF_FLD/metamodel/RIF-FLD.ecore

Atom->terms(Variable).type(ReductDef)->elements	NamedArgumentAtom.arg
RelationPredicate RelationPredicate.symbol RelationPredicate.terms	Equal, External.termula(PositionalAtom) PositionalAtom.termula Equal.left, Equal.rught, PositionalAtom.args
Conjunction Conjunction.formula	Compound Compound.termula
Disjunction Disjunction.formula	Compound Compound.termula
ExsistentiallyQuantifiedFormula ExsistentiallyQuantifiedFormula.variables ExsistentiallyQuantifiedFormula.formula	Quantified Quantified.var Quantified.termula
Variable Variable.name	Var Var.name
FunctionCall FunctionCall.name FunctionCall.terms	PositionalAtom PositionalAtom.termula PositionalAtom.arg
NavigationPath NavigationPath.source NavigationPath.featureCall	PositionalAtom PositionalAtom.arg PositionalAtom.termula, PositionalAtom.arg
AdditiveOpCall AdditiveOpCall.name AdditiveOpCall->terms	External.termula(PositionalAtom) External.termula(PositionalAtom).termula(Constant).value External.termula(PositionalAtom)->arg
MultiplicativeOpCall MultiplicativeOpCall.name MultiplicativeOpCall.terms	External.termula(PositionalAtom) External.termula(PositionalAtom).termula(Constant).value External.termula(PositionalAtom)->arg
ReductElement ReductElement.path->attribute.name	NamedArgument NamedArgument.name.stringRepr, NamedArgument.value
ReductElement ReductElement.path->attribute.name	PropertyAssignment PropertyAssignment.property, PropertyAssignment.value
BooleanValueDef BooleanValueDef.value	Constant Constant.value
IntValueDef IntValueDef.value	Constant Constant.value
RealValueDef RealValueDef.value	Constant Constant.value
StringValueDef StringValueDef.value	Constant Constant.value

SequenceValueDef SequenceValueDef.containedContents	ClosedList ClosedList.termula
TimeConstantDef TimeConstantDef.year TimeConstantDef.month TimeConstantDef.day TimeConstantDef.hour TimeConstantDef.minute TimeConstantDef.second	Constant Constant.value Constant.value Constant.value Constant.value Constant.value Constant.value
IntervalConstantDef IntervalConstantDef.year IntervalConstantDef.month IntervalConstantDef.day IntervalConstantDef.hour IntervalConstantDef.minute IntervalConstantDef.second	Constant Constant.value Constant.value Constant.value Constant.value Constant.value Constant.value

Отображение между конструкциями языка СИНТЕЗ и конструкциями языка RIF-BLD с их семантикой в семантических структурах языка RIF-BLD [141] приведено в таблице А.11.

Таблица А.11 - Отображение между конструкциями языка СИНТЕЗ и конструкциями языка RIF-BLD с их семантикой

Примечание	Конструкция СИНТЕЗ	Семантика СИНТЕЗ	Конструкция RIF-BLD	Семантика RIF-BLD
Переменная	v	$v \in V$?r $I(v) = I_V(v)$	?v	?v $\in V$?r $I(?v) = I_V(?v)$
Числовая, строковая константа	k	k $\in C$?nst $I(k) = I_C(k)$	k	k $\in C$?nst $I(k) = I_C(k)$
Булевская константа	tru? f?ls?		“tru?”^^xs:B??l??n “f?ls?”^^xs:b??l??n	
Временная константа	m?m?nt(??#mm#dd) m?m?nt(??#mm#dd hh:min:ss) m?m?nt(hh:min:ss) int?rv?l(??#mm) int?rv?l(dd hh:min:ss)		“??-mm-dd”^^xs:d?t? “??-mm-ddThh:min:ss”^^xs:d?t?Tim? “hh:min:ss”^^xs:tim? “P??YmmM”^^xs:???rM?nthDur?ti?n “PddDThhHminMssS”^^xs:d??Tim?Dur?ti?n	
Константа-последовательность	$[t_1, \dots, t_n]$	$I([]) = I_{list}(<>)$ $I([t_1, \dots, t_n]) = I_{list}(I(t_1), \dots, I(t_n))$, при n>0	List(m[t_1] … m[t_n]) *Замечание. Внутри List не может быть переменных.* Ext?rn?l(func:m?k?-list(m[t_1] … m[t_n]))	$I(List()) = I_{list}(<>)$ $I(List(m[t_1] \dots m[t_n])) = I_{list}(I(m[t_1]), \dots, I(m[t_n]))$ $I(Ext?rn?l(func:m?k?-list(m[t_1] \dots m[t_n]))) = I_{?xt?rn?l}(?it?m_1 \dots ?it?m_n;$ func:m?k?-list(?it?m_1 ... ?it?m_n))($m[t_1], \dots, m[t_n]$) = $I_{list}(I(m[t_1]), \dots, I(m[t_n]))$
Фрейм	{id; ?: $t_?$; b: t_b; c: t_{c1}, t_{c2};	?, b, c $\in C$?nst I({id; ?: $t_?$; b: t_b; c: t_{c1}, t_{c2};}) = $I_{fr?m?}(I(id))$({<I(?),I($t_?$)>,	id[?->m[$t_?$] b->m[t_b] c->m[t_{c1}] c->m[t_{c2}])]	I(id[?->m[$t_?$] b->m[t_b] c->m[t_{c1}] c->m[t_{c2}])]) = $I_{fr?m?}(I(id))$({<I(?),I(m[$t_?$])>,

	} {J□hnD□nn; □g□: 33; s□l□r□: 10000; p□ssp□rt: 2345563, 3476263; }	<I(b),I(t_b), <I(c),I(t_{c1}), <I(c),I(t_{c2})>})	J□hnD□nn[□g□->33 s□l□r□->10000 p□ssp□rt->2345563 p□ssp□rt->3476263]	<I(b),I(m[t_b]), <I(c),I(m[t_{c1}]), <I(c),I(m[t_{c2}])>})
Равенство	t = s	I(t=s) = $I_{=}$(I(t), I(s))	m[t] = m[s]	I(m[t] = m[s]) = $I_{=}$(I(m[t]), I(m[s]))
Членство в коллекции	C(v)	I(C(v)) = $I_{is□}$(I(v), I(C))	?v # C	I(?v # C) = $I_{is□}$(I(?v), I(C))
Предикат-коллекция с анонимной переменной	C(_/T[□, b]) предполагается использование только в голове правила, без переменований и путей	□rg_□, □rg_b ∈ ArgN□m□s ?□, ?b ∈ V□r - *новые переменные* I(C(_/T[□, b])) = I_{NF}(I(C))({<□rg_□, I(?□)>, <□rg_b, I(?b)>})	C(□rg_□->?□ □rg_b->?b)	□rg_□, □rg_b ∈ ArgN□m□s ?□, ?b ∈ V□r I(C(□rg_□->?□ □rg_b->?b)) = I_{NF}(I(C))({<□rg_□, I(?□)>, <□rg_b, I(?b)>})
Переименование и пути в редукте	C(v/T[c: □.b, d])	□, b, c, d ∈ C□nst ?□, ?b, ?c, ?d ∈ V□r - *новые переменные* TV□I_I(C(v/T[□.b/c, d])) = **t** iff I_{truth}($I_{is□}$(I(v), I(C))) = **t** ∧ I_{truth}($I_{fr□m□}$(I(v))({<I(□),I(?□)>, <I(d),I(?d)>})) = **t** ∧ I_{truth}($I_{fr□m□}$(I(?□))({<I(b),I(?c)>})) = **t**	And(?v # C ?v[□->?□ d->?d] ?□[b->?c])	□, b, c, d ∈ C□nst ?□, ?b, ?c, ?d ∈ V□r TV□I_I(And(?v # C ?v[□->?□ d->?d] ?□[b->?c])) = **t** iff I_{truth}($I_{is□}$(I(?v), I(C))) = **t** ∧ I_{truth}($I_{fr□m□}$(I(?v))({<I(□),I(?□)>, <I(d),I(?d)>})) = **t** ∧ I_{truth}($I_{fr□m□}$(I(?□))({<I(b),I(?c)>})) = **t**
Термы - вызовы методов	v.f(t_1, ... , t_n)	f ∈ C□nst I(v.f(t_1, ... , t_n)) = I_F(I(f))(I(v),I(t_1),...,I(t_n))	f(?v m[t_1] … m[t_n])	f ∈ C□nst I(f(?v m[t_1] … m[t_n])) = I_F(I(f))(I(v),I(m[t_1]),...,I(m[t_n]))

Термы – пути как аргументы предикатов	P(v.□.b)	?v□, ?v□b ∈ V□r - *новые переменные* TV□I_I(P(v.□.b)) = **t** iff $I_{truth}(I_{is□}(I(?v□b), I(P))) = \mathbf{t} \wedge I_{truth}(I_{fr□m□}(I(v))(\{<I(□), I(?v□)>\})) = \mathbf{t} \wedge I_{truth}(I_{fr□m□}(I(?v□))(\{<I(b), I(?v□b)>\})) = \mathbf{t}$	And(?v[□->?v□] ?v□[b->?v□b] P(?v□b))	?v, ?v□, ?v□b ∈ V□r TV□I_I(And(?v[□->?v□] ?v□[b->?v□b] P(?v□b))) = **t** iff $I_{truth}(I_{is□}(I(?v□b), I(P))) = \mathbf{t} \wedge I_{truth}(I_{fr□m□}(I(?v))(\{<I(□), I(?v□)>\})) = \mathbf{t} \wedge I_{truth}(I_{fr□m□}(I(?v□))(\{<I(b), I(?v□b)>\})) = \mathbf{t}$
Функциональные термы и предикаты f – имя функции (не встроенной функции)	$f(t_1, \dots, t_n)$	f ∈ C□nst $I(f(t_1, \dots, t_n)) = I_F(I(f))(I(t_1),\dots,I(t_n))$	f(m[t_1] … m[t_n]) Ext□rn□I(m[t_1] … m[t_n] I□c) схема для такого терма (?X_1 … ?X_n; f(?X_1 … ?X_n); I□c) где I□c – местонахождение модуля	f ∈ C□nst $I(f(m[t_1] \dots m[t_n])) = I_F(I(f))(I(m[t_1]),\dots,I(m[t_n]))$
Конъюнкция	(ϕ_1 & ... & ϕ_n)	TV□I_I(ϕ_1 & ... & ϕ_n) = **t** iff TV□I_I(ϕ_1) = ... = TV□I_I(ϕ_n) = **t**, иначе, TV□I_I(ϕ_1 & ... & ϕ_n) = **f**	And(m[ϕ_1] ... m[ϕ_n])	TV□I_I(And(m[ϕ_1] ... m[ϕ_n])) = **t** iff TV□I_I(m[ϕ_1]) = ... = TV□I_I(m[ϕ_n]) = **t**, иначе, TV□I_I(And(m[ϕ_1] ... m[ϕ_n])) = **f**
Дизъюнкция	(ϕ_1 | ... | ϕ_n)	TV□I_I(ϕ_1 | ... | ϕ_n) = **f** iff TV□I_I(ϕ_1) = ... = TV□I_I(ϕ_n) = **f**, иначе, TV□I_I(ϕ_1 | ... | ϕ_n) = **t**	Or(m[ϕ_1] ... m[ϕ_n])	TV□I_I(Or(m[ϕ_1] ... m[ϕ_n])) = **f** iff TV□I_I(m[ϕ_1]) = ... = TV□I_I(m[ϕ_n]) = **f**, иначе, TV□I_I(Or(m[ϕ_1] ... m[ϕ_n])) = **t**
Квантор существования	□x v_1, ... ,v_n (ϕ)	TV□I_I(□x v_1, ... , v_n (ϕ)) = **t** iff для некоторого I* TV□I_{I*}(ϕ) = **t**. I* совпадает с I во всем, кроме, может быть, означивания v_1, ... , v_n.	Exists ?v_1 ... ?v_n(m[ϕ])	TV□I_I(Exists ?v_1 ... ?v_n(m[ϕ])) = **t** iff для некоторого I* TV□I_{I*}(ϕ) = **t**. I* совпадает с I во всем, кроме, может быть, означивания v_1, ... , v_n.
Правила v_1, ... ,v_n – все свободные переменные ψ, в φ нет	ϕ:- ψ.	TV□I_I(ϕ:- ψ.) = **t** iff для любого I*, TV□I_{I*}(ϕ) = **f** либо	F□r□ll ?v_1 ... ?v_n (m[ϕ]:- m[ψ])	TV□I_I(m[ϕ]:- m[ψ]) = **t** iff для любого I*, TV□I_{I*}(m[ϕ]) = **f** либо

свободных переменных, отличных от v_1, ... ,v_n		TV□I_{I^*}(ψ) = **t**, иначе TV□I_I(ϕ:-ψ.) = **f**. I* совпадает с I во всем, кроме, может быть, означивания v_1, ... , v_n.		TV□I_{I^*}(m[ψ]) = **t**, иначе TV□I_I(m[ϕ]:-m[ψ]) = **f**. I* совпадает с I во всем, кроме, может быть, означивания v1, ... , vn.
Факты	ϕ. v_1 ... v_n – свободные переменные	TV□I_I(ϕ.) = **t** iff для любого I*, TV□I_{I^*}(ϕ) = **t**. I* совпадает с I во всем, кроме, может быть, означивания v_1, ... , v_n.	F□r□ll ?v_1 ... ?v_n(m[ϕ])	TV□I_I(m[ϕ]) = **t** iff для любого I*, TV□I_{I^*}(m[ϕ]) = **t**. I* совпадает с I во всем, кроме, может быть, означивания v_1, ... , v_n.
Списки правил	$ϕ_1$. ... $ϕ_n$.	TV□I_I($ϕ_1$. ... $ϕ_n$.) = **t** iff TV□I_I($ϕ_1$) = ... = TV□I_I($ϕ_n$) = **t**, иначе TV□I_I($ϕ_1$. ... $ϕ_n$.) = **f**.	Gr□up(m[$ϕ_1$] ... m[$ϕ_n$])	TV□I_I(Gr□up(m[$ϕ_1$] ... m[$ϕ_n$])) = **t** iff TV□I_I(m[$ϕ_1$]) = ... = TV□I_I(m[$ϕ_n$]) = **t**, иначе TV□I_I(Gr□up(m[$ϕ_1$] ... m[$ϕ_n$])) = **f**.

Модули	{ m□dul□N□m□;
	in: m□dul□;
	imp□rt: impM□dul□N□m□;
	fr□m□:
	{ pr□gN□m□; in: pr□gr□m;
	{{ $ϕ_1$. ... $ϕ_n$. }};
	};
	}
	
	{ Sh□k□sp□□r□P□□tr□;
	in: m□dul□;
	imp□rt:
	Sh□k□sp□□r□Ont□l□g□;
	fr□m□:
		Пусть M – модуль m□dul□N□m□, N – модуль impM□dul□N□m□, G – множество программ из модуля N, I_N – структура, интерпретирующая N.
		$\hat{I}$ = {I, I_N}
		TV□$I_{\hat{I}}$(M) = **t** iff TV□I_I($ϕ_1$. ... $ϕ_n$.) = TV□I_{IN}(G) = **t**.
			(*<d□cum□nt> <d□cum□nt>[dc:titl□->"m□dul□N□m□"]*)
			D□cum□nt(
			Imp□rt(impM□dul□N□m□)
			Gr□up(
			(*<□ss□rti□ns>
			<□ss□rti□ns>[dc:titl□->"pr□gN□m□"]
			*)
			Gr□up(m[$ϕ_1$] ... m[$ϕ_n$])
			)
			
			(*<d□cum□nt> <d□cum□nt>[dc:titl□->"Sh□k□sp□□r□P□□tr□"]*)
			D□cum□nt(
			Imp□rt(Sh□k□sp□□r□Ont□l□g□)
				$\hat{I}$ = {I, $I_{m[N]}$}
				TV□$I_{\hat{I}}$(m[M]) = **t** iff TV□I_I(Gr□up(Gr□up(m[$ϕ_1$] ... m[$ϕ_n$]))) = TV□I_{IN}(m[G]) = **t**.

	{ H□ml□t; in: pr□gr□m;
	{{ ϕ_1. ... ϕ_n. }};
	},
	{ Oth□ll□; in: pr□gr□m;
	{{ ψ_1. ... ψ_m. }};
	};
	}
			Gr□up(
			(*<□ss□rti□ns>
			<□ss□rti□ns>[dc:titl□->"H□ml□t"]
			*)
			Gr□up(ϕ_1. ... ϕ_n.)
			
			(*<□ss□rti□ns>
			<□ss□rti□ns>[dc:titl□->"Oth□ll□"]
			*)
			Gr□up(ψ_1. ... ψ_m.)
			))	

Встроенные функции	x, □ – строки x+□	+ ∈ C□nst I_F(x+□) = z, z образована конкатенацией I(x) и I(□)	Ext□rn□l(func:c□nc□t(m[x] m[□]))	I(Ext□rn□l(func:c□nc□t(m[x] m[□]))) = $I_{□xt□rn□l}$(?□rg_1 ?□rg_2; func:c□nc□t(?□rg_1 ?□rg_2))(I(m[x]) I(m[□])) = r□s, r□s образована конкатенацией I(m[x]) и I(m[□])
Встроенные предикаты	x, □ – строки x < □	< ∈ C□nst TV□l_I(x < □) = I_{truth}(I_F(x < □)) = **t** iff I(x) < I(□) в лексикографическом смысле	Ext□rn□l(pr□d:num□ric-l□ss-th□n(Ext□rn□l(func:c□mp□r□(m[x] m[□])) 0))	TV□l_I(Ext□rn□l(pr□d:num□ric-l□ss-th□n(Ext□rn□l(func:c□mp□r□(m[x] m[□])) 0))) = I_{truth}($I_{□xt□rn□l}$(?□rg_1 ?□rg_2; pr□d:num□ric-l□ss-th□n(?□rg_1 ?□rg_2))(***I***$_{□xt□rn□l}$(?c□mp□r□nd$_1$?c□mp□r□nd$_2$; func:c□mp□r□(?c□mp□r□nd$_1$?c□mp□r□nd$_2$))(I(m[x]), I(m[□])), I(0))) = **t** iff I(m[x]) < I(m[□]) в лексикографическом

				смысле, т.к. именно в этом случае func:c□mp□r□ возвращает -1.

Замечание. Для следующих термов: фреймы, равенства, членство в коллекции, предикат-коллекция, функциональные термы и предикаты, вызовы методов функция истинностного означивания $TVal_I(\circ)$ для СИНТЕЗ и RIF-BLD определяется следующим образом: $TVal_I(t) = I_{truth}(I(t))$.

Строгое соответствие встроенных типов языка СИНТЕЗ и языка RIF-BLD приведено в таблице А.12.

Таблица А.12 - Соответствие встроенных типов СИНТЕЗ и RIF-BLD

Тип СИНТЕЗ	Тип RIF
Boolean	xs:Boolean
double	xs:double
float	xs:float
octet	xs:hexBinary
integer	xs:decimal
long	xs:int
short	xs:short
unsigned_long	xs:unsignedInt
unsigned_short	xs:unsignedShort
string	xs:string
{time; from: {yy}; to: {dd};}	xs:date
{time; from: {yy}; to: {ss};}	xs:dateTime
{time; from: {hh}; to: {ss};}	xs:time
{interval; from: {dd}; to: {ss};}	xs:dayTimeDuration
{interval; from: {yy}; to: {mm};}	xs:yearMonthDuration

Для числовых, булевского и строкового типов лексические пространства совпадают с соответствующими пространствами типов XSD [368]. Пространства значений типов СИНТЕЗ можно положить равными пространствам значений соответствующих типов XSD.

Отображения встроенных предикатов и функций приведены в таблицах А.13-А.16.

Таблица А.13 – Отображение числовых, булевых предикатов и функций

Конструкция СИНТЕЗ	Конструкция RIF-BLD
x + y	External(func:numeric-add(m[x] m[y]))
-	func:numeric-subtract
*	func:numeric-multiply
/	func:numeric-divide func:numeric-integer-divide
mod	func:numeric-mod

<	pred:numeric-less-than
>	pred:numeric-greater-than
<>	pred:numeric-not-equal
<=	pred:numeric-less-than-or-equal
>=	pred:numeric-greater-than-or-equal
not	func:not

Таблица А.14 – Отображение строковых предикатов и функций

Конструкция СИНТЕЗ	**Конструкция RIF-BLD**
+	func:concat
in	pred:contains
substring	func:substring
like	pred:matches
cardinal	func:string-length
=	func:compare
x != y	func:compare
x < y	func:compare
>	func:compare
<=	func:compare
>=	func:compare

Таблица А.15 – Отображение предикатов и функций над последовательностями

Конструкция СИНТЕЗ	**Конструкция RIF-BLD**
cardinal	func:count
is_empty	=
is_in	pred:list-contains
first	func:get
last	func:get
elem(s, i)	func:get
concatenation	func:concatenate
reverse	func:reverse
insert_after(s, e, i)	func:insert-before
insert_before	func:insert-before
insert_first	func:insert-before
insert_last	func:append
remove_at	func:remove

remove_first	func:remove
remove_last	func:remove
replace_at	func:remove, func:insert-before

Таблица А.16 – Отображение временных предикатов и функций над типами time и interval

Конструкция СИНТЕЗ	Конструкция RIF-BLD
add	func:add-yearMonthDurations func:add-dayTimeDurations func:add-yearMonthDuration-to-date func:add-dayTimeDuration-to-time
minus	func:subtract-yearMonthDurations func:subtract-dayTimeDurations
ti_minus	func:subtract-yearMonthDuration-from-dateTime func:subtract-yearMonthDuration-from-date func:subtract-dayTimeDuration-from-dateTime func:subtract-dayTimeDuration-from-date func:subtract-dayTimeDuration-from-time
tt_minus	func:subtract-dateTimes func:subtract-dates func:subtract-times
mult	func:multiply-yearMonthDuration func:multiply-dayTimeDuration
div	func:divide-yearMonthDuration func:divide-dayTimeDuration
ii_div	func:divide-yearMonthDuration-by-yearMonthDuration func:divide-dayTimeDuration-by-dayTimeDuration
year	func:year-from-dateTime func:year-from-date func:year-from-duration
month	func:month-from-dateTime func:month-from-date func:month-from-duration
day	func:day-from-dateTime func:day-from-date func:day-from-duration
hour	func:hours-from-dateTime func:hours-from-time func:hours-from-duration
minute	func:minutes-from-dateTime func:minutes-from-time func:minutes-from-duration
second	func:seconds-from-dateTime func:seconds-from-time func:seconds-from-duration

equals	pred:dateTime-equal pred:date-equal pred:time-equal
ge	pred:dateTime-greater-than-or-equal pred:date-greater-than-or-equal pred:time-greater-than-or-equal pred:dayTimeDuration-greater-than-or-equal pred:yearMonthDuration-greater-than-or-equal
le	pred:dateTime-less-than-or-equal pred:date-less-than-or-equal pred:time-less-than-or-equal pred:dayTimeDuration-less-than-or-equal pred:yearMonthDuration-less-than-or-equal
ne	func:not, pred:dateTime-equal func:not, pred:date-equal func:not, pred:time-equal
between	pred:dateTime-less-than-or-equal pred:date-less-than-or-equal pred:time-less-than-or-equal
eq	pred:duration-equal
gt	pred:dateTime-greater-than pred:date-greater-than pred:time-greater-than pred:dayTimeDuration-greater-than pred:yearMonthDuration-greater-than
lt	pred:dateTime-less-than pred:date-less-than pred:time-less-than pred:dayTimeDuration-less-than pred:yearMonthDuration-less-than

ПРИЛОЖЕНИЕ Б

Применения метода верификации интеграции данных

Б.1 Верификация интеграции данных в системе «Умное землепользование»

Раздел основан на работе автора [74].

В данном разделе приведен пример применения метода верификации интеграции данных в системе «Умное землепользование», разрабатываемой в Почвенном институте имени В. В. Докучаева.

Рассмотрена интеграция данных в реестре земель. SQL-скрипт загрузки и преобразования данных[102], а также схема данных реестра земель[103] опубликованы в репозитории GitHub.

Семантика целевой схемы данных и операций вставки данных в ее таблицы выражается в AMN конструкцией *SmartLanduseDB* (приведен фрагмент спецификации, полная спецификация опубликована в репозитории GitHub[104]):

```
MACHINE
    SmartLanduseDB
```

Каждой таблице схемы (например, таблице агроэкологических типов земель *ae_type*) соответствует определение структуры в разделе DEFINITIONS (*ae_type_struct*) и переменная в разделе ABSTRACT_VARIABLES (*ae_type*):

```
DEFINITIONS
    ae_type_struct == struct(
        ae_type_id: INT,
        ae_group_id: INT,
        name: seq(0..255),
        relief: seq(0..255),
        otlozheniya: seq(0..255),
        pochva: seq(0..255),
        climate: seq(0..255)
    );

ABSTRACT_VARIABLES
    ae_type,
```

[102] https://github.com/sstupnikov/DataTransformation/blob/main/SmartLanduse/V4__2_reg_ae_type_data.sql

[103] https://github.com/sstupnikov/DataTransformation/blob/main/SmartLanduse/V3__1_init_db.sql

[104] https://github.com/sstupnikov/DataTransformation/blob/main/SmartLanduse/SmartLanduseDB.mch

Переменные, соответствующие таблицам, типизируются в разделе INVARIANT. Там же в виде формул логики первого порядка выражаются свойства уникальности первичных ключей:

```
INVARIANT
    ae_type: FIN(ae_type_struct) &
    !(ae_type1, ae_type2).(ae_type1: ae_type & ae_type2: ae_type &
        ae_type1'ae_type_id = ae_type2'ae_type_id =>
        ae_type1 = ae_type2)
```

Переменные инициализируются пустыми множествами в разделе INITIALISATION:

```
INITIALISATION
    ae_type:= {}
```

Операции вставки данных в таблицы (например, в таблицу *ae_type*) представляются операциями AMN:

```
OPERATIONS

insert_into_ae_type (anae_group_id, aname, arelief, anotlozheniya, apochva,
aclimate) =
PRE
    anae_group_id: INT &
    aname: seq(0..255) &
    arelief: seq(0..255) &
    anotlozheniya: seq(0..255) &
    apochva: seq(0..255) &
    aclimate: seq(0..255)
THEN
    ANY pk WHERE pk: INT & not(#(aet).(aet: ae_type & aet'ae_type_id = pk))
    THEN
       ae_type:= ae_type \/
       {rec(ae_type_id: pk, ae_group_id: anae_group_id, name: aname,
               relief: arelief, otlozheniya: anotlozheniya,
               pochva: apochva, climate: aclimate)}
    END
END

END
```

Семантика SQL-скрипта интеграции исходных данных в целевую схему выражается в AMN конструкцией *AgroecologicalGroupRegistryRef* (приведен фрагмент спецификации, полная спецификация опубликована в репозитории GitHub[105]), включающей (INCLUDES) спецификацию целевой схемы *SmartLanduseDB*:

[105] https://github.com/sstupnikov/DataTransformation/blob/main/SmartLanduse/AgroecologicalGroupRegistryRef.ref

```
REFINEMENT AgroecologicalGroupRegistryRef
INCLUDES  SmartLanduseDB
```

Скрипт, фактически, представляет собой цикл, для каждой записи входных данных выполняющий последовательность операций вставки соответствующих элементов данных в таблицы целевой схемы. Операции вставки, относящиеся к различным элементам данных, представляются различными операциями AMN. Для управления последовательным исполнением операций определено множество состояний машины:

```
SETS
    QUERY_PERFORMED = {
        READY_TO_INIT,
        RECORDS_EXTRACTED,
        RECORD_SELECTED,
        ZONE_TRANSFORMED,
        PROVINCE_TRANSFORMED,
        REGION_TRANSFORMED,
        DISTRICT_TRANSFORMED,
        GROUP_TRANSFORMED,
        TYPE_TRANSFORMED,
        TOPSOIL_TRANSFORMED,
        HUMUS_TRANSFORMED,
        PHKCL_TRANSFORMED,
        ACIDITY_TRANSFORMED
    }
```

Так, состояние READY_TO_INIT отвечает готовности машины к загрузке исходных данных, а, например, состояние TYPE_TRANSFORMED означает, что элемент данных, соответствующий агроэкологическому типу земель, вставлен в целевую базу данных.

Структура исходных данных *tmp_ae_group_struct* определена в разделе DEFINITIONS:

```
DEFINITIONS
    tmp_ae_group_struct == struct(
      administrative_region: seq(0..255),
      zone: seq(0..255),
      province: seq(0..255),
      administrative_district: seq(0..255),
      ae_group: seq(0..255),
      description: seq(0..255),
      ae_type: seq(0..255),
      topsoil_thickness: seq(0..255),
      humus_content: seq(0..255),
      phkcl: seq(0..255),
      hydrolytic_acidity: seq(0..255),
      ground_water_level: seq(0..255),
      relief: seq(0..255),
      otlozheniya: seq(0..255),
      pochva: seq(0..255),
      climate: seq(0..255)
    )
```

В разделе ABSTRACT_VARIABLES определены переменные, содержащие исходные данные (*tmp_ae_group*), текущую обрабатываемую запись (*r_group*), необработанные записи (*records_to_transform*), состояние машины (*queries_performed*), а также вспомогательные переменные, используемые в скрипте (например, *v_ae_type_id*):

```
ABSTRACT_VARIABLES
    tmp_ae_group,
    r_group,
    v_province_id,
    v_ae_group_id,
    v_ae_type_id,

    records_to_transform,
    queries_performed
```

В разделе INVARIANT переменные типизируются:

```
INVARIANT
    tmp_ae_group: FIN(tmp_ae_group_struct) &
    r_group: tmp_ae_group_struct &
    v_province_id: INT &
    v_ae_group_id: INT &
    v_ae_type_id: INT &
    records_to_transform: FIN(tmp_ae_group_struct) &
    queries_performed: QUERY_PERFORMED &
```

а также в виде формулы представляется свойство корректности интеграции данных (приведен только фрагмент, относящийся к виду земель):

```
   (queries_performed = READY_TO_INIT &
    tmp_ae_group /= {} &
    records_to_transform = {} =>
       !(record).(record: tmp_ae_group =>
           #(aprovince, agroup, atype, paet).(
               aprovince: province & record'province = aprovince'name &
               atype: ae_type & record'ae_type = atype'name &
               atype'ae_group_id = agroup'ae_group_id &
               atype'relief = record'relief &
               atype'otlozheniya = record'otlozheniya &
               atype'pochva = record'pochva &
               atype'climate = record'climate &
               paet: province_ae_type &
               paet'province_id = aprovince'province_id &
               paet'ae_type_id = atype'ae_type_id
    ) ) )
```

Инвариант утверждает, что после обработки скриптом всех записей исходного набора данных для каждого элемента каждой записи исходных данных в целевой базе данных существует соответствующий ему элемент и все элементы данных связаны между собой необходимым непротиворечивым образом.

Переменные инициализируются в разделе INITIALISATION:

```
INITIALISATION
    tmp_ae_group:= {} ||
    ANY vv WHERE vv: tmp_ae_group_struct THEN r_group:= vv END ||
    ANY vv WHERE vv: INT THEN v_province_id:= vv END ||
    ANY vv WHERE vv: INT THEN v_ae_group_id:= vv END ||
    ANY vv WHERE vv: INT THEN v_ae_type_id:= vv END ||
    records_to_transform:=  {} ||
    queries_performed:= READY_TO_INIT
```

Операции в разделе OPERATIONS выражают семантику различных операций, составляющих SQL-скрипт. Так, загрузка исходных данных в переменную *tmp_ae_group* производится при помощи операции *extract*, выбор очередной записи для преобразования и загрузки в целевую базу данных – при помощи операции *select_record_to_transform*, преобразование элемента данных, отвечающего типу земель – при помощи операции *transform_type*:

```
OPERATIONS

extract(records) =
PRE records: FIN(tmp_ae_group_struct)
THEN
    SELECT queries_performed = READY_TO_INIT
    THEN
        tmp_ae_group:= records ||
        records_to_transform:= records ||
        queries_performed:= RECORDS_EXTRACTED
    END
END;

select_record_to_transform =
SELECT queries_performed = RECORDS_EXTRACTED
THEN
    IF records_to_transform /= {}
    THEN
        ANY vv WHERE vv: records_to_transform
        THEN
            r_group:= vv ||
            queries_performed:= RECORD_SELECTED
        END
    ELSE
        queries_performed:= READY_TO_INIT
    END
END;

transform_type =
SELECT queries_performed = GROUP_TRANSFORMED
THEN
    IF not(#(vv).(vv: ae_type & vv'name = r_group'ae_type &
                  vv'ae_group_id = v_ae_group_id))
    THEN
        insert_into_ae_type(v_ae_group_id, r_group'ae_type, r_group'relief,
            r_group'otlozheniya, r_group'pochva, r_group'climate);
        ANY vv WHERE
            vv: ae_type &
```

```
            vv'name = r_group'ae_type &
            vv'ae_group_id = v_ae_group_id
         THEN
            v_ae_type_id:= vv'ae_type_id
         END;
         insert_into_province_ae_type (v_province_id, v_ae_type_id)
      END
      ||
      queries_performed:= TYPE_TRANSFORMED
END;
```

Конструкции *SmartLanduseDB* и *AgroecologicalGroupRegistry* были загружены в проект инструментария Atelier B, автоматически были сгенерированы теоремы корректности спецификаций, применены средства автоматического и интерактивного доказательства. Статистика доказательства приведена в таблице Б.1.

Таблица Б.1 - Количество сгенерированных и автоматически доказанных теорем непротиворечивости спецификаций

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic		
		Force Fast	Force 0	Force 3
SmartLanduseDB.mch	40	10	30	30
AgroecologicalGroup RegistryRef.ref	55	1	30	31

Б.2 Верификация интеграции данных в хранилище для поддержки поисково-спасательных операций в Арктической зоне

Раздел основан на работах автора [62] [63].

В данном разделе приведен пример применения метода верификации интеграции данных в хранилище для поддержки поисково-спасательных операций в Арктической зоне [63]. Схема хранилища содержит набор сущностей, связанных с описанием поисковых операций, поисковых задач и инцидентов, сил и средств в составе координационных центров ПСО, объектов поиска, маршрутов, траекторий движения, а также инфраструктуры портов и аэродромов, средств коммуникации, локации и оповещения, погодных условий, упоминаний инцидентов и объектов поиска в сети Интернет и СМИ [333]. В качестве конкретных источников данных, подлежащих интеграции, выбраны система мониторинга судов «Виктория» [369], комплексная интегрированная информационная система «МоРе» [301], межведомственная информационная система ЕСИМО [304], международная спутниковая поисково-спасательная система КОСПАС-САРСАТ [370], программный комплекс «Поиск-Море» [303].

Правила интеграции данных реализованы с использованием языка HIL. Структуры данных из источников выглядят следующим образом (в качестве примера приведены только структуры данных о судах):

```
declare ShipPositions : set[ uid: string, callSign: string, name:  string,
nameLat: string, isoCountryCode: string, imo: string, mmsi: string];
declare Ship : set[uid: string, call_sign: string, name: string, country:
string, speed: string, max_range: string, max_crew: string, description:
string, imo_number: string, length: string, breadth: string, displacement:
string, deadweight: string];
declare VehicleVICT : set[uid: string, call: string, name: string, latName:
string, country: string, vehicleType: string];
declare VesselVICT : set[uid: string, imoNumber: string, mmsi: string,
vehicleId: string];
declare VehicleESIM : set[uid: string, call: string, name: string, country:
string, maxSpeed: string, maxRange: string, maxCrew: string, description:
string, vehicleType: string, ownerId: string];
declare VesselESIM : set[uid: string, imoNumber: string, length: string,
width: string, displacement: string, deadweight: string, vehicleId: string];
```

Фрагмент целевой схемы, включающий структуры данных о судах, выглядит следующим образом:

```
declare Vehicle : set[uid: string, call: string, name: string, latName:
string, country: string, vehicleType: string, description: string, maxCrew:
string, maxRange: string, maxSpeed: string, ownerId: string];
declare Vessel : set[uid: string, imoNumber: string, mmsi: string,
vehicleId: string, length: string, width: string, deadweight: string,
displacement: string];
declare VehicleLink : set[
  vehicleVict: [uid: string, call: string, name: string],
  vehicleEsim: [uid: string, call: string, name: string]];
```

Примеры правил трансформации данных выглядят следующим образом:

```
insert into VehicleVICT
select [
    uid: get_source_vehicle_id(sp.uid)
    , call: sp.callSign
    , name: sp.name
    , latName: sp.nameLat
    , country: get_country(sp.isoCountryCode)
    , vehicleType: "Vessel"
    ]
from ShipPositions sp;

create link VehicleLink as
select [
    vehicleVict: [
        uid: vv.uid
        , call: vv.call
        , name: vv.name
        ]
    , vehicleEsim: [
        uid: ve.uid
        , call: ve.call
        , name: ve.name
        ]    ]
```

```
from VehicleVICT vv, VehicleESIM ve
match using
    rule1: (vv.call = ve.call)
;

insert into Vehicle
select [
    uid: get_vehicle_id(vv.uid, ve.uid)
    , call: vv.call
    , name: vv.name
    , latName: vv.latName
    , country: vv.country
    , vehicleType: vv.vehicleType
    , description: ve.description
    , maxCrew: ve.maxCrew
    , maxRange: ve.maxRange
    , maxSpeed: ve.maxSpeed
    , ownerId: ve.ownerId
    ]
from VehicleVICT vv, VehicleESIM ve, VehicleLink vl
where (vv.uid = vl.vehicleVict.uid) and (ve.uid = vl.vehicleEsim.uid)
;
```

Семантические спецификации правил интеграции на языке AMN были получены автоматически применением разработанного программного средства [300]. Коллекции входных и выходных данных были представлены в виде переменных, типизированных в разделе INVARIANT и инициализированных в разделе IIINITIALISATION (приведен фрагмент определения). Переменная *state* отвечает за состояние машины для управления порядком исполнения операций:

```
REFINEMENT ArcticRef
VARIABLES
  state,
  ShipPositions,
  Ship,
  VehicleVICT,
  VesselVICT,
  VehicleESIM,
  VesselESIM,
  Vehicle,
  Vessel,
  VehicleLink

INVARIANT
  ShipPositions : POW(struct(
      uid: STRING_TYPE,
      callSign: STRING_TYPE,
      name: STRING_TYPE,
      nameLat: STRING_TYPE,
      isoCountryCode: STRING_TYPE,
      imo: STRING_TYPE,
      mmsi: STRING_TYPE
      )) &
  state : struct(
    VehicleLinkCreated: BOOL,
    VehicleESIMFromShipInserted: BOOL,
    VesselESIMFromShipInserted: BOOL,
```

```
    VehicleVICTFromShipPositionsInserted: BOOL,
    VesselVICTFromShipPositionsInserted: BOOL,
    VehicleFromVehicleVICTVehicleESIMVehicleLinkInserted: BOOL,
    VesselFromVesselVICTVesselESIMVehicleLinkInserted: BOOL
)

INITIALISATION
  ShipPositions := {} ||
  state :=
    rec(VehicleLinkCreated: FALSE,
      VehicleESIMFromShipInserted: FALSE,
      VesselESIMFromShipInserted: FALSE,
      VehicleVICTFromShipPositionsInserted: FALSE,
      VesselVICTFromShipPositionsInserted: FALSE,
      VehicleFromVehicleVICTVehicleESIMVehicleLinkInserted: FALSE,
      VesselFromVesselVICTVesselESIMVehicleLinkInserted: FALSE)
```

Каждое правило представлено отдельной операцией AMN (приведены операции для примеров правил, приведенных выше):

```
OPERATIONS

insertIntoVehicleVICTFromShipPositions =
SELECT
    TRUE = TRUE
  THEN
    VehicleVICT := (VehicleVICT \/
      {rr | #(sp).(sp : ShipPositions &
        rr = rec(uid: get_source_vehicle_id(sp'uid),
            call: sp'callSign, name: sp'name, latName: sp'nameLat,
            country: get_country(sp'isoCountryCode),
            vehicleType: VesselConst))});
    state'VehicleVICTFromShipPositionsInserted := TRUE
END;

createVehicleLink =
SELECT
    state'VehicleESIMFromShipInserted = TRUE &
    state'VehicleVICTFromShipPositionsInserted = TRUE
THEN
    VehicleLink := {rr | #(vv, ve).(vv : VehicleVICT &
        ve : VehicleESIM &
        rr = rec(
          vehicleVict: rec(uid: vv'uid, call: vv'call,  name: vv'name),
          vehicleEsim: rec(uid: ve'uid, call: ve'call, name: ve'name)) &
        (vv'call = ve'call))};
    state'VehicleLinkCreated := TRUE
  END;

insertIntoVehicleFromVehicleVICTVehicleESIMVehicleLink =
SELECT
    state'VehicleESIMFromShipInserted = TRUE &
    state'VehicleLinkCreated = TRUE &
    state'VehicleVICTFromShipPositionsInserted = TRUE
THEN
    Vehicle := (Vehicle \/ {rr | #(vv, ve, vl).(vv : VehicleVICT &
        ve : VehicleESIM &
        vl : VehicleLink &
        rr = rec(uid: get_vehicle_id(vv'uid, ve'uid), call: vv'call,
```

```
            name: vv'name, latName: vv'latName, country: vv'country,
            vehicleType: vv'vehicleType, description: ve'description,
             maxCrew: ve'maxCrew, maxRange: ve'maxRange,
              maxSpeed: ve'maxSpeed, ownerId: ve'ownerId) &
        vv'uid = vl'vehicleVict'uid &
        ve'uid = vl'vehicleEsim'uid)});
    state'VehicleFromVehicleVICTVehicleESIMVehicleLinkInserted := TRUE
  END;
```

В качестве свойства программы интеграции, подлежащего верификации, рассматривается сохранение информации о судах при интеграции, выражаемое следующей формулой:

```
state'VehicleFromVehicleVICTVehicleESIMVehicleLinkInserted = TRUE &
state'VesselFromVesselVICTVesselESIMVehicleLinkInserted= TRUE  =>
  !(pp, ss).(pp: ShipPositions & ss: Ship & pp'callSign = ss'call_sign =>
    #(vh, vs).(vh: Vehicle & vs: Vessel & vh'call = pp'callSign &
      vs'vehicleId = vh'uid & vs'mmsi = pp'mmsi &
      vs'deadweight = ss'deadweight )
```

В формуле утверждается, что для любой пары сущностей из исходных коллекций *ShipPositions* и *Ship* соответственно с совпадающим позывным (*callSign, call_sign*) по завершении процесса интеграции в результирующих коллекциях *Vehicle* и *Vessel* присутствуют сущности, значения позывного (*call*) и других существенных атрибутов (например, *mmsi, deadweight*) которых совпадают с значениями соответствующих атрибутов исходных сущностей. Эта формула добавляется в инвариант семантической спецификации *ArcticRef*.

Полная спецификация правил трансформации данных о судах и их формальной семантики в AMN опубликована в репозитории GitHub[106]. Семантические спецификации были помещены в проект инструментального средства Atelier B. Для спецификации *Arctic.ref* было автоматически сгенерировано 32 теоремы непротиворечивости, из них автоматически доказано 11.

Б.3 Верификация интеграции данных в интегрированной базе данных Института металлургии и материаловедения им. А.А. Байкова РАН

Раздел основан на работе автора [84].

В данном разделе приведен пример применения метода верификации интеграции данных в интегрированной базе данных Института металлургии и материаловедения им. А.А. Байкова РАН [371] [372].

---

[106] https://github.com/sstupnikov/DataTransformation/blob/main/Arctic/

Рассмотрена интеграция данных из базы данных Bandgap [373], содержащей данные о ширине запрещенной зоны основных классов неорганических веществ. «Ширина запрещенной зоны является фундаментальным параметром конденсированных фаз, который характеризует природу химической связи в материале. По величине запрещенной зоны можно судить о типе химической связи, доминирующей в соединении, устойчивости соединения в определенном интервале изменений состава и внешних параметров, типе электронной проводимости в образцах, склонности материала к ионной проводимости, а также основных термодинамических характеристиках соединения[107]».

Интеграция данных проводится в два этапа: на первом этапе данные из базы данных Bandgap, помеченные на удаление, изменение или добавление, преобразуются в промежуточное XML-представление; на втором этапе для элементов XML-представления вызываются процедуры целевой интегрированной базы данных, удаляющие, изменяющие или добавляющие в нее соответствующие записи.

Фрагменты схемы исходной базы данных (*Bandgap-ver.sql*), схемы промежуточного XML-представления (*MUService-ver.xsd*), схемы и процедур целевой базы данных (*Metabase-ver.sql*), программы преобразования данных к промежуточному представлению (*UpdateBandGapMeta-ver.vbs*), использованные при верификации, опубликованы в репозитории GitHub[108].

Семантика схемы исходной базы данных и необходимых запросов к ней выражается в AMN конструкцией *Bandgap.mch*[109]:

```
MACHINE
    Bandgap
```

Определяются и типизируются структуры данных и переменные, выражающие семантику отношений исходной схемы (*SubstancesConv* – отношение, включающее записи о химических системах, *PropertiesConv* - отношение, включающее записи о свойствах систем, *DBContentConv* - отношение, включающее записи о наличии в базе данных значений свойств систем):

---

[107] https://bg.imet-db.ru/default.asp?lang=ru

[108] https://github.com/sstupnikov/DataTransformation/tree/main/Bandgap

[109] https://github.com/sstupnikov/DataTransformation/blob/main/Bandgap/Bandgap.mch

```
DEFINITIONS
    SubstancesConv_struct == struct(
        SubstanceID: INT,
        UpdateStatus: INT,
        NumElements: INT,
        Compound: seq(0..255),
        Elements: seq(0..255)
    );

    PropertiesConv_struct == struct(
        NOMPROP: INT,
        UpdateStatus: INT,
        NAZVPROP: seq(0..255),
        HTML: seq(0..255)
    );

    DBContentConv_struct == struct(
        SubstanceID: INT,
        NOMPROP: INT,
        UpdateStatus: INT
    )
ABSTRACT_VARIABLES
    SubstancesConv,
    PropertiesConv,
    DBContentConv
INVARIANT
    SubstancesConv: FIN(SubstancesConv_struct) &
    !(subst1, subst2).(subst1: SubstancesConv & subst2: SubstancesConv &
        subst1'SubstanceID = subst2'SubstanceID =>
        subst1 = subst2) &

    PropertiesConv: FIN(PropertiesConv_struct) &
    !(prop1, prop2).(prop1: PropertiesConv & prop2: PropertiesConv &
        prop1'NOMPROP = prop2'NOMPROP =>
        prop1 = prop2) &

    DBContentConv: FIN(DBContentConv_struct) &
    !(cont1, cont2).(cont1: DBContentConv & cont2: DBContentConv &
        cont1'SubstanceID = cont2'SubstanceID &
        cont1'NOMPROP = cont2'NOMPROP =>
        cont1 = cont2)
INITIALISATION

    SubstancesConv:= {} ||
    PropertiesConv:= {} ||
    DBContentConv:= {}
```

Семантика запросов к базе данных выражается операциями AMN:

```
OPERATIONS
result <-- select_SubstancesConv =
result := { rcrd | rcrd: SubstancesConv & rcrd'UpdateStatus > 0 };

result <-- select_PropertiesConv =
result := { rcrd | rcrd: PropertiesConv & rcrd'UpdateStatus > 0 };

result <-- select_DBContentConv =
result := { rcrd | rcrd: DBContentConv & rcrd'UpdateStatus > 0 }

END
```

Семантика схемы промежуточного XML-представления и операций создания узлов XML-документа выражается в AMN конструкцией *MUService.mch*[110]*:*

```
MACHINE  MUService
```

Определяются и типизируются структуры данных, выражающие семантику элементов схемы:

```
DEFINITIONS
    SystemInfoItem == struct(
        SystemID: INT,
        Elements: seq(0..255),
        SystemInfo: seq(0..255),
        Description: seq(0..255),
        UpdateStatus: INT
    );

    PropertiesInfoItem == struct(
        PropID: INT,
        Name: seq(0..255),
        Description: seq(0..255),
        WWWTemplatePage: seq(0..255),
        UpdateStatus: INT
    );

    DBContentItem == struct(
        SystemID: INT,
        PropID: INT,
        UpdateStatus: INT
    )
ABSTRACT_VARIABLES
    MetaBase
INVARIANT
    MetaBase: struct(
       SystemInfo: FIN(SystemInfoItem),
       PropertiesInfo: FIN(PropertiesInfoItem),
       DBContent: FIN(DBContentItem)
    )
```

Семантика операций создания узлов XML-документа выражается операциями AMN:

```
OPERATIONS
createNodeSystemInfoItem(value) =
PRE value: SystemInfoItem
THEN
    MetaBase'SystemInfo := MetaBase'SystemInfo \/ {value}
END;
```

[110] https://github.com/sstupnikov/DataTransformation/blob/main/Bandgap/MUService.mch

```
createNodePropertiesInfoItem(value) =
PRE value: PropertiesInfoItem
THEN
    MetaBase'PropertiesInfo := MetaBase'PropertiesInfo \/ {value}
END;

createNodeDBContentInfoItem(value) =
PRE value: DBContentItem
THEN
    MetaBase'DBContent := MetaBase'DBContent \/ {value}
END

END
```

Семантика схемы целевой базы данных выражается в AMN конструкцией *Metabase.mch*[111]:

```
MACHINE
    Metabase
```

Определяются и типизируются структуры данных и переменные, выражающие семантику отношений целевой схемы:

```
DEFINITIONS
    SystemInfo_struct == struct(
        DBID: INT,
        SystemID: INT,
        ElemNumber: INT,
        UpdateStatus: INT,
        Elements: seq(0..255),
        SystemInfo: seq(0..255),
        Description: seq(0..255)
    );

    PropertiesInfo_struct == struct(
        DBID: INT,
        PropID: INT,
        Name: seq(0..255),
        Description: seq(0..255),
        WWWTemplatePage: seq(0..255),
        UpdateStatus: INT
    );

    DBContent_struct == struct(
        DBID: INT,
        SystemID: INT,
        PropID: INT,
        UpdateStatus: INT
    )
```

111 https://github.com/sstupnikov/DataTransformation/blob/main/Bandgap/Metabase.mch

```
ABSTRACT_VARIABLES
    SystemInfo,
    PropertiesInfo,
    DBContent

INVARIANT
    SystemInfo: FIN(SystemInfo_struct) &
    !(system1, system2).(system1: SystemInfo & system2: SystemInfo &
        system1'DBID = system2'DBID &
        system1'SystemID = system2'SystemID =>
        system1 = system2) &

    PropertiesInfo: FIN(PropertiesInfo_struct) &
    !(prop1, prop2).(prop1: PropertiesInfo & prop2: PropertiesInfo &
        prop1'DBID = prop2'DBID & prop1'PropID = prop2'PropID =>
        prop1 = prop2) &

    DBContent: FIN(DBContent_struct) &
    !(content1, content2).(content1: DBContent & content2: DBContent &
        content1'DBID = content2'DBID &
        content1'SystemID = content2'SystemID &
        content1'PropID = content2'PropID =>
        content1 = content2)

INITIALISATION
    SystemInfo:= {} ||
    PropertiesInfo:= {} ||
    DBContent:= {}
```

Семантика процедур целевой базы данных выражается операциями AMN (ниже приведен пример операции для одной из процедур):

```
UpdateDBContent(adbid, asystemid, apropid, anupdatestatus) =
PRE
    adbid: INT & asystemid: INT &
    apropid: INT &  anupdatestatus: INT
THEN
    IF anupdatestatus = 2
    THEN
       DBContent := DBContent -
        { rcrd | rcrd: DBContent  & rcrd'DBID = adbid &
                 rcrd'SystemID = asystemid & rcrd'PropID  = apropid}
    ELSE
       IF #(rcrd).(rcrd: DBContent & rcrd'DBID = adbid &
                   rcrd'SystemID = asystemid & rcrd'PropID = apropid)
       THEN // Update
           DBContent :=
           (DBContent - { rcrd | rcrd: DBContent & rcrd'DBID = adbid &
              rcrd'SystemID = asystemid & rcrd'PropID = apropid}) \/
           {rec(DBID: adbid, SystemID: asystemid, PropID: apropid,
                UpdateStatus: 1)}
       ELSE // Insert
           DBContent := DBContent \/
           {rec(DBID: adbid, SystemID: asystemid, PropID: apropid,
                UpdateStatus: 1)}
       END
    END
END
```

Семантика программ интеграции данных выражается в AMN конструкцией *Bandgap2MetabaseRef.ref*[112]:

```
REFINEMENT Bandgap2MetabaseRef
INCLUDES
    Bandgap,
    MUService,
    Metabase
```

Управление последовательностью исполнения операций трансформации данных осуществляется при помощи переменной *state*, принимающей значения из множества TRANSFORMATION_PERFORMED:

```
SETS
    TRANSFORMATION_PERFORMED = {
        READY_TO_TRANSFORM,
        TRANSFORM_SUBSTANCESCONV_RECORD,
        TRANSFORM_PROPERTIESCONV,
        TRANSFORM_PROPERTIESCONV_RECORD,
        TRANSFORM_DBCONTENTCONV,
        TRANSFORM_DBCONTENTCONV_RECORD,
        TRANSFORM_SYSTEM_INFO,
        TRANSFORM_SYSTEM_INFO_NODE,
        TRANSFORM_PROPERTIES_INFO,
        TRANSFORM_PROPERTIES_INFO_NODE,
        TRANSFORM_DBCONTENT,
        TRANSFORM_DBCONTENT_NODE
    }
ABSTRACT_VARIABLES
    state
INVARIANT
    state: TRANSFORMATION_PERFORMED
INITIALISATION
    state:= READY_TO_TRANSFORM
```

Так, состояние TRANSFORM_PROPERTIESCONV означает, что происходит трансформация записей таблицы *PropertiesConv*, а состояние TRANSFORM_SYSTEM_INFO – что происходит трансформация XML-элементов типа *SystemInfoItem*.

Определяются вспомогательные переменные, соответствующие множествам еще не обработанных записей или элементов:

```
ABSTRACT_VARIABLES
    RSN_SubstancesConv,
    RSN_PropertiesConv,
    RSN_DBContentConv,
    systemInfo,
    propertiesInfo,
```

---

[112] https://github.com/sstupnikov/DataTransformation/blob/main/Bandgap/Bandgap2MetabaseRef.ref

```
    dbContent
INVARIANT
    RSN_SubstancesConv: FIN(SubstancesConv_struct) &
    RSN_PropertiesConv: FIN(PropertiesConv_struct) &
    RSN_DBContentConv: FIN(DBContentConv_struct) &
    systemInfo: FIN(SystemInfoItem) &
    propertiesInfo: FIN(PropertiesInfoItem) &
    dbContent: FIN(DBContentItem)
INITIALISATION
    RSN_SubstancesConv:= {} ||
    RSN_PropertiesConv:= {} ||
    RSN_DBContentConv:= {} ||
    systemInfo:= {} ||
    propertiesInfo:= {} ||
    dbContent:= {}
```

Семантика операций трансформации отдельных записей или XML-элементов выражается операциями AMN (ниже приведен пример одной из операций, соответствующей преобразованию записи из таблицы *DBContentConv*):

```
ProcessDBContentInfoItem =
SELECT state = TRANSFORM_DBCONTENTCONV_RECORD
THEN
    VAR SubstanceID, NOMPROP, UpdateStatus, tmpLink IN
        IF RSN_DBContentConv /= {}
        THEN
            ANY rcrd WHERE rcrd: RSN_DBContentConv
            THEN
                SubstanceID:= rcrd'SubstanceID;
                NOMPROP := rcrd'NOMPROP;
                UpdateStatus := rcrd'UpdateStatus;

                tmpLink := rec(
                    SystemID: SubstanceID,
                    PropID: NOMPROP,
                    UpdateStatus: UpdateStatus
                );

                createNodeDBContentInfoItem(tmpLink);

                RSN_DBContentConv:= RSN_DBContentConv - {rcrd}
            END
        ELSE
            state:= TRANSFORM_SYSTEM_INFO
        END
    END
END;
```

В виде формулы – части инварианта - представляется свойство корректности интеграции данных (приведен фрагмент формулы, относящийся к химическим системам):

```
state = READY_TO_TRANSFORM &
MetaBase /= MetaBaseConst &
(MetaBase'SystemInfo /= {} or MetaBase'PropertiesInfo /= {} or
 MetaBase'DBContent /= {}) =>
```

```
!(record).(record: SubstancesConv & record'UpdateStatus = 2 =>
  not(#(system).(system: SystemInfo &
                 record'SubstanceID = system'SystemID)) ) &
!(record).(record: SubstancesConv & record'UpdateStatus > 0 &
           record'UpdateStatus /= 2 =>
#(system).(system: SystemInfo &
  record'SubstanceID = system'SystemID &
  record'Elements = system'Elements &
  record'Compound = system'SystemInfo))
```

Формула утверждает, что после выполнения всех операций трансформации данных в целевой базе данных нет записей о химических системах, помеченных на удаление в исходной базе данных, и есть записи о всех химических системах, помеченных на обновление или вставку в исходной базе данных.

Конструкции *Bandgap.mch, MUService.mch, Metabase.mch, Bandgap2Metabase.ref* были загружены в проект инструментария Atelier B, автоматически были сгенерированы теоремы корректности спецификаций, применены средства автоматического и интерактивного доказательства. Статистика доказательства приведена в таблице Б.2.

Таблица Б.2 - Количество сгенерированных и автоматически доказанных теорем непротиворечивости спецификаций

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic			
		Force Fast	Force 0	Force 1	Force 2
Bandgap.mch	6	3	6		
MUService.mch	8				2
Metabase.mch	33	5	22	23	
Bandgap2Metabase.ref	320	8	267	272	

Б.4 Верификация интеграции данных в базе данных двойных звезд ИНАСАН

В данном разделе приведен пример применения метода верификации интеграции данных в базе данных двойных звезд, разрабатываемой в Институте астрономии РАН [374] [375] [376] [377]. BDB нацелена на интеграцию данных о двойных и кратных звездных системах различных наблюдательных типов, содержит данные о физических и позиционных параметрах более чем 500 000 звездных систем, их компонентов и пар. Кратность систем варьируется от 2 до более чем 20. Система интегрирует данные из более чем 40 открытых астрономических каталогов и баз данных. Интеграция данных из источников в системе производится при помощи программ на императивном языке программирования (Python). В качестве целевой базы данных используется реляционная СУБД PostgreSQL. Преобразование исходных данных в кортежи целевой базы данных

осуществляется с использованием объектно-реляционного отображения (библиотека SQLAlchemy[113] и декларативная надстройка над ней – Elixir).

Для верификации интеграции данных в BDB, фактически, необходимо определить формальную семантику подмножества императивного языка (Python), используемого для определения программ интеграции, а также семантику подмножества языка SQL. В данном подразделе приведена структура схем данных и программ интеграции данных в базе данных двойных звезд, общая структура спецификаций языка AMN, выражающих формальную семантику программ интеграции данных. Определение формальной семантики схемы источника данных, схемы целевой базы данных, программ трансформации данных и загрузки в целевую базу данных иллюстрируется на примерах интеграции Вашингтонского каталога двойных звезд (WDS) [378] [379], поддерживаемого Морской обсерваторией США. Программа интеграции WDS содержит большинство императивных конструкций, применяемых для интеграции источников в BDB.

Структура схем данных и программ интеграции данных в базе данных двойных звезд [374] включает следующие компоненты:

- схемы источников данных;
- реляционная схема интегрированной базы данных;
- программы интеграции данных (преобразования данных из схем источников в целевую схему) на языке Python;
- операции вставки кортежей в таблицы целевой базы данных на языке SQL.

Источники данных для BDB – это файлы астрономических звездных каталогов. Схемы данных файлов каталогов описываются в файлах Readme[114], которые обычно можно получить при помощи сервиса астрономических каталогов Visier, разработанного астрономической обсерваторией Страсбурга [VIS]. Так, например, схема каталога WDS[115] содержит побайтовое описание строк файла каталога wds.dat[116]. Примеры строк из описания схемы WDS выглядит следующим образом:

---

113 https://www.sqlalchemy.org/

114 Также в файлах ReadMe указано общее число строк каталога (например, 131662 для WDS).

115 https://cdsarc.cds.unistra.fr/ftp/B/wds/ReadMe

116 https://cdsarc.cds.unistra.fr/ftp/B/wds/wds.dat.gz

```
--------------------------------------------------------------------------
   Bytes Format Units   Label   Explanations
--------------------------------------------------------------------------
   1- 10  A10   ---     WDS     WDS name (based on J2000 posi-tion)
  59- 63  F5.2  mag     mag1    ? Magnitude of First Component
```

Так, байты 1-10 каждой строки файла каталога содержат подстроку (тип обозначается «A») длины 10, обозначающую имя звездной системы в каталоге. Байты 59-63 содержат действительное число (тип обозначается «F»), содержащее максимум 5 значащих цифр и максимум 2 – после запятой, и обозначающее звездную величину первого компонента пары[117]. Всего строка каталога WDS содержит 27 атрибутов.

Для каждого каталога создана программа интеграции данных на языке Python, осуществляющая загрузку данных из файла каталога, для каждой строки файла вызывающая код преобразования данных, а затем использующая объектно-реляционное отображение (ORM) для вставки соответствующих кортежей в таблицы целевой базы данных.

Определение формальной семантики схем данных и программ интеграции данных, а также их дальнейшая верификация осуществляется для каждого источника данных независимо. Структура семантических спецификаций верификации интеграции источников данных в BDB включает следующие компоненты:

– AMN-спецификация вида MACHINE[118] [35], определяющая семантику реляционной схемы целевой базы данных и операций вставки данных в таблицы целевой базы данных (BDB.mch);

– AMN-спецификация вида REFINEMENT [35], определяющая семантику императивной программы интеграции данных, последовательно вызывающей операции преобразования данных и вставки данных в таблицы целевой базы данных. Отдельная такая спецификация создается для каждого источника данных;

– набор спецификаций вида MACHINE, определяющий семантику операций встроенных типов данных и вспомогательных функций.

---

[117] Вообще говоря, A10 и F5.2 – это типы форматных строк из языка Фортран.

[118] Такая спецификация называется «абстрактной машиной», и ее нужно отличать от более общего понятия «абстрактная вычислительная машина», используемого при определении операционной семантики.

Ниже в данном разделе формальная семантика программ интеграции данных иллюстрируется на примере каталога WDS. Все упомянутые файлы схем, программ, AMN-спецификаций опубликованы в репозитории GitHub[119].

Так, структура связей семантических спецификаций AMN, необходимых для верификации корректности интеграции данных каталога WDS в BDB, изображена на рисунке Б.1.

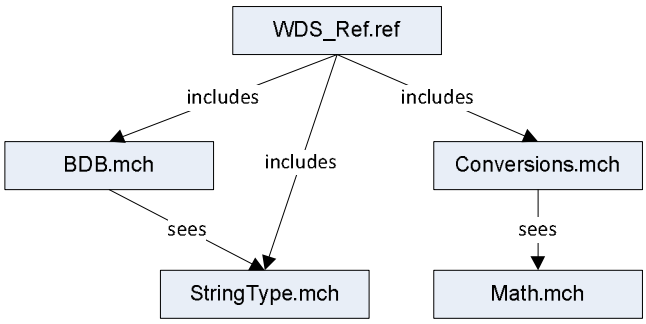

Рисунок Б.1 – Структура связей спецификаций AMN

Семантика программы интеграции данных WDS представляется при помощи спецификации *WDS_Ref.ref*. Семантика операций строкового типа данных представляется при помощи спецификации *StringType.mch*. Семантика операций числовых типов представляется при помощи спецификации *Math.mch*. Семантика вспомогательных функций преобразования данных[120] представляется при помощи спецификации *Conversions.mch*. Отношение *видимости* (sees) между спецификациями предназначено для доступа к множествам и константам «видимой» спецификации из «видящей» спецификации. Отношение *включения* (includes) спецификациями предназначено для доступа к множествам, константам, переменным, операциям «включаемой» спецификации из «включающей» спецификации.

Семантика реляционной схемы базы данных двойных звезд и операций вставки кортежей в ее таблицы. Фрагмент реляционной схемы базы данных двойных звезд приведен в файле *scheme-ver2.sql*. Схема включает следующие основные таблицы:

– таблица *bdb3_models_catline* содержит записи для каждой строки каждого файла каталога. Каждая запись включает исходную строку из файла каталога, номер этой

119 https://github.com/sstupnikov/DataTransformation/tree/main/BDB

120 https://github.com/sstupnikov/DataTransformation/blob/main/BDB/funcs-ver.py

строки, ссылку на запись о файле, а также внутренний идентификатор и идентификатор, на который как на внешний ключ ссылаются записи из таблиц пар, компонентов и систем, полученные из данной строки;

– таблица *bdb3_models_bdbsystem* содержит записи о звездных системах;

– таблица *bdb3_models_bdbpair* содержит записи о звездных парах;

– таблица *bdb3_models_bdbcomp* содержит записи о компонентах звездных систем.

Пример определения семантики таблицы компонентов звездных систем (*bdb3_models_bdbcomp*) приведен в таблице Б.3[121].

Таблица Б.3 – Семантика таблицы компонентов звездных систем

Конструкция SQL	Конструкция AMN

CREATE TABLE
public.bdb3_models_bdbcomp (
  catline_id integer NOT NULL,
  mag double precision,
  band character varying(40),
  "spType" character varying(40),
  "pmRA" double precision,
  "pmDE" double precision,
  "crdRA" double precision,
  "crdDE" double precision,
  sys_catline_id integer
);
	DEFINITIONS
	bdb3_models_bdbcomp_struct ==
	struct(
	  catline_id: INT,
	  mag: REAL,
	  spBand: seq(CHAR),
	  spType: seq(CHAR),
	  pmRA: REAL,
	  pmDE: REAL,
	  crdRA: REAL,
	  crdDE: REAL,
	  sys_catline_id: INT,
	);
	ABSTRACT_VARIABLES
	  bdb3_models_bdbcomp
	INVARIANT
	  bdb3_models_bdbcomp:
	  FIN(bdb3_models_bdbcomp_struct)
	INITIALISATION
	  bdb3_models_bdbcomp:= {}

Для реляционной таблицы в AMN определяется одноименная структура (*struct*) с соответствующими атрибутами. Строковому типу атрибута *character varying* в AMN соответствует тип последовательности символов (*seq(CHAR)*), целочисленному типу – тип INT, типу с плавающей точкой – тип REAL. Семантика таблицы как множества кортежей представляется переменной *bdb3_models_bdbcomp*, которая типизируется конечным множеством (FIN) упомянутых выше структур и инициализируется пустым множеством.

[121] Для экономии места приведены лишь те атрибуты таблицы, для которых есть соответствующие данные в каталоге WDS.

Семантика ограничений первичного ключа

```
ALTER TABLE ONLY public.bdb3_models_bdbcomp
ADD CONSTRAINT bdb3_models_bdbcomp_pkey
PRIMARY KEY (catline_id);
```

выражается формулой первого порядка, которая становится частью инварианта:

```
!(comp1, comp2).(comp1: bdb3_models_bdbcomp &
  comp2: bdb3_models_bdbcomp & comp1'catline_id = comp2'catline_id =>
  comp1 = comp2)
```

Аналогичным образом выражается семантика внешних ключей:

```
ALTER TABLE ONLY public.bdb3_models_bdbcomp
ADD CONSTRAINT bdb3_models_bdbcomp_sys_catline_id_fk
FOREIGN KEY (sys_catline_id)
REFERENCES public.bdb3_models_bdbsystem(catline_id);
```

Соответствующая формула выглядит следующим образом:

```
!comp.(comp: bdb3_models_bdbcomp =>
  #system.(system: bdb3_models_bdbsystem &
           system'catline_id = comp'sys_catline_id) )
```

Семантика операции вставки кортежа в таблицу компонентов выражается при помощи операции *insert_comp* спецификации BDB:

```
insert_comp(acatline_id, amag, aspBand, aspType, apmRA, apmDE, acrdRA,
acrdDE, asys_catline_id) =
PRE
acatline_id: INT & amag: REAL & aspBand: seq(CHAR) & aspType: seq(CHAR) &
apmDE: REAL & acrdRA: REAL & acrdDE: REAL & asys_catline_id: INT
THEN
  SELECT
    not(#comp.(comp: bdb3_models_bdbcomp &
                comp'catline_id = acatline_id)) &
    #catline.(catline: bdb3_models_catline &
              catline'cat_id = acatline_id) &
    #system.(system: bdb3_models_bdbsystem &
             system'catline_id = asys_catline_id)
  THEN
    bdb3_models_bdbcomp:= bdb3_models_bdbcomp \/
    {rec(catline_id: acatline_id, mag: amag, spBand: aspBand,
         spType: aspType, pmRA: apmRA, pmDE: apmDE, crdRA: acrdRA,
         crdDE: acrdDE, sys_catline_id: asys_catline_id)}
  END
END;
```

Предусловие операции (PRE) – это формула правильной типизации всех компонентов вставляемого кортежа. Дополнительные условия вставки кортежа (SELECT) – отсутствие во множестве компонентов *bdb3_models_bdbcomp* компонента с тем же идентификатором *acatline_id* (условие первичного ключа), наличие во множестве *bdb3_models_catline* кортежа с идентификатором *acatline_id* (условие внешнего ключа), наличие во множестве *bdb3_models_bdbsystem* (соответствующего таблице звездных

систем) системы с идентификатором *asys_catline_id*, частью которой является вставляемый компонент (условие внешнего ключа). При выполнение этих условий кортеж добавляется к множеству *bdb3_models_bdbcomp*.

Аналогичным образом определяется семантика операций вставки в другие таблицы целевой схемы.

Семантика операций встроенных типов данных и вспомогательных функций. В программах интеграции данных используются различные операции с числовыми типами, помимо обычных арифметических операций (модуль *Math* языка Python). Семантика таких операций выражается в одноименной AMN-спецификации *Math.mch* с использованием абстрактных констант и их свойств. Так, операции определения абсолютного значения действительного числа *abs* соответствует одноименная константа:

```
ABSTRACT_CONSTANTS
    abs
PROPERTIES
    abs: REAL --> REAL &
    !xx.(xx: REAL & xx >= 0.0 => abs(xx) = xx) &
    !xx.(xx: REAL & xx < 0.0 => abs(xx) = 0.0 - xx)
```

Константа *abs* типизируется как полная функция (-->) на множестве действительных чисел. Ее действие определяется при помощи конъюнкции двух формул с квантором всеобщности (!). Знак «=>» здесь обозначает логическую импликацию.

Спецификация *Conversions.mch* включает определения операций, выражающих семантику вспомогательных функций преобразования данных, определенных в файле *funcs-ver.py*. Так, например, для вычисления *прямого восхождения* (right ascension, RA) – одной из координат, принятой для задания позиции объектов на небесной сфере, используется функция *hms2deg*. Ее семантика выражается одноименной операцией спецификации *Conversions.mch* (таблица Б.4).

Таблица Б.4 – Семантика вспомогательной функции преобразования данных

Конструкция Python	Конструкция AMN

`def hms2deg(hh,mm,ss):`
`  return`
`  hh*15.0 +`
`  mm*15.0/60.0 +`
`  ss*15.0/3600.0`
	`MACHINE Conversions`
	`SEES Math`
	`OPERATIONS`
	`res <-- hms2deg(hh, mm, ss) =`
	`PRE hh: REAL & mm: REAL & ss: REAL`
	`THEN`
	`  res:= hh*15.0 + mm*15.0/60.0 + ss*15.0/3600.0`
	`END;`

Спецификация *StringType.mch* включает определения операций, выражающих семантику встроенных методов Python для обработки строк (*strip, split, find*). Так, например, семантика функции *find,* производящей поиск значения в строке и возвращающей позицию, на которой находится это значение, выражается одноименной операцией спецификации *StringType.mch:*

```
res <-- find(str, char) =
PRE str: seq(CHAR) & char: CHAR
THEN
  IF char /: ran(str)
  THEN
    res:= -1
  ELSE
    ANY ind WHERE
      ind: INT & ind >= 1 & ind <= size(str) & str(ind) = char &
      not(#ind1.(ind1: INT & ind1: dom(str) & ind >= 1 &
                 ind1 < ind & str(ind1) = char))
    THEN
      res:= ind
    END
  END
END
```

Здесь знак «/:» означает непринадлежность множеству, *ran* – область значений функции, *dom* – область определения функции, *size* – длина последовательности, *not* – логическое отрицание, «#» – квантор существования. Конструкция

```
ANY ind WHERE P THEN S END
```

означает выбор произвольного значения переменной *ind*, удовлетворяющего условию *P*, которое можно использовать во вложенной конструкции *S*.

Семантика программ интеграции данных в базе данных двойных звезд. Программа интеграции каждого источника данных в BDB обычно состоит из двух частей:

– предварительное преобразование данных и инициализация вспомогательных переменных (*Pre-iteration-code*);

– инициализация атрибутов кортежей таблиц целевой схемы (*Iteration-code*).

Этот код запускается последовательно для каждой строки из файла – источника данных, сначала – часть *Pre-iteration-code*, затем – часть *Iteration-code*.

Семантика программ интеграции данных будет проиллюстрирована ниже на примере фрагмента программы интеграции каталога WDS (файл *wds-ver3.dat_parsing*). Фрагмент включает основные конструкции языка Python, используемые при создании программ интеграции: последовательная композиция операторов, условный оператор, присваивание значения вспомогательной переменной, вызов библиотечной или

вспомогательной функции, присваивание атрибуту целевой схемы значения вспомогательной переменной или выражения (таблица Б.5).

Таблица Б.5 – Фрагмент программы интеграции каталога WDS

Pre-iteration-code	Iteration-code

`fld2=fld[2].strip()`
`if fld2!="":`
`  if len(fld2)==2:`
`  compA=fld2[0]`
`  compB=fld2[1]`
`  else:`
`  fs=fld2.split(",")`
`  if len(fs)==2:`
`  compA=fs[0]`
`  compB=fs[1]`
`  else:`
`  raise_error`
	`objA:id=fld[0]+compA`
	`objA:pmRA=pmC*fld[13]`
	`objA:crdDE=DE1`
	`objA:pmDE=pmC*fld[14]`
	`objA:crdRA=RA1`
	`objA:band=band`
	`objA:mag=fld[10]`
	`objA:spType=sptypeA`

Во фрагменте *Pre-iteration-code* в массиве *fld* хранятся значения атрибутов из текущей обрабатываемой строки каталога. Производится присваивание идентификаторов компонентов звездной пары вспомогательным переменным *compA, compB* на основании значения атрибута *Comp* из каталога, хранящегося как третий элемент массива *fld – fld[2]*. Например, если *fld[2] = «BD»*, то *compA = «B», compB = «D»*. Если же *fld[2] = «Aa,Ab»*, то *compA = «Aa», compB = «Ab».*

Во фрагменте *Iteration-code* переменная *objA* обозначает кортеж, вставляемый в таблицу компонентов звездных систем *bdb3_models_bdbcomp* целевой схемы. Кортеж соответствует первому компоненту пары. Производится присваивание значений атрибутам кортежа.

Семантика программы интеграции данных выражается в спецификации *WDS_Ref.ref* следующим образом. Для хранения исходных данных создаются переменные *fld0, … , fld26,* каждая из которых соответствует отдельному атрибуту входных данных. Тип каждой из переменных – это последовательность конечной длины (предполагается, что длина последовательности равна числу записей в исходных данных):

```
ABSTRACT_CONSTANTS
  RECORD_COUNT
PROPERTIES
  RECORD_COUNT: INT & RECORD_COUNT > 0
ABSTRACT_VARIABLES
  fld0, … fld26
```

```
INVARIANT
  fld0: seq(seq(CHAR)) & size(fld0) = RECORD_COUNT &
  !str.(str: ran(fld0) => size(str) <= 10) &
  fld10: seq(REAL) & size(fld10) = RECORD_COUNT  &
  fld13: seq(INT) & size(fld13) = RECORD_COUNT
```

Во фрагменте спецификации выше в качестве примера приведена типизация лишь нескольких переменных:

– переменная *fld0* соответствует атрибуту исходных данных *WDS* (идентификатор системы в каталоге), его формат – A10 (строка длины 10). Поэтому в инварианте длина любого элемента последовательности *fld0* ограничивается 10. Длина самой последовательности *fld0* задается константой RECORD_COUNT;

– переменная *fld10* соответствует атрибуту исходных данных *mag1* (звездная величина первого компонента системы), его формат – F5.2 (действительное число);

– переменная *fld13* соответствует атрибуту исходных данных *pmRA1* (первичное собственное движение по прямому восхождению), его формат – I4 (целое число).

Каждая вспомогательная переменная программы интеграции (например, *compA, compB, fld2* во фрагменте выше) представляется одноименной переменной спецификации *WDS_Ref.ref*:

```
ABSTRACT_VARIABLES
  compA, compB, fld2aux
INVARIANT
  compA: seq(CHAR) & compB: seq(CHAR) & fld2aux: seq(CHAR)
```

Итеративный процесс обработки записей исходного файла каталога моделируется при помощи целочисленной переменной *iteration_num,* которая хранит номер обрабатываемой записи:

```
ABSTRACT_VARIABLES
  iteration_num
INVARIANT
  iteration_num: INT
```

Последовательная композиция операторов программы интеграции данных из части *Pre-iteration-code* представляется в AMN при помощи набора операций. Для каждого небольшого логически связанного последовательного набора операторов создается своя операция [74]. Так, например, для фрагмента *Pre-iteration-code*, приведенного выше, созданы три операции спецификации *WDS_Ref.ref* (в таблице Б.6 строки кода Python приведены в левом столбце, соответствующие операции AMN – в правом).

Таблица Б.6 – Семантика последовательной композиции операторов

Конструкции Python	Операции AMN

`fld2=fld[2].strip()`	`pre_iteration_CompStrip =`
	`SELECT`
	`  iteration_num >= 1 &`
	`  iteration_num <= RECORD_COUNT &`
	`  iteration_state = COMP_AB_INIT`
	`THEN`
	`  fld2aux <-- strip(fld2(iteration_num));`
	`  iteration_state:= FLD2_STRIP`
	`END;`

`if fld2!="":`
`  if len(fld2)==2:`
`  compA=fld2[0]`
`  compB=fld2[1]`
	`pre_iteration_CompSimpleUpdate =`
	`SELECT`
	`  iteration_num >= 1 &`
	`  iteration_num <= RECORD_COUNT &`
	`  iteration_state = FLD2_STRIP &`
	`  fld2aux /= [] & size(fld2aux) = 2`
	`THEN`
	`  compA:= [fld2aux(1)];`
	`  compB:= [fld2aux(2)];`
	`  iteration_state:= COMP_AB_UPDATE`
	`END;`

`  else:`
`  fs=fld2.split(",")`
`  if len(fs)==2:`
`  compA=fs[0]`
`  compB=fs[1]`
`  else:`
`  raise_error`
	`pre_iteration_CompSplit =`
	`SELECT`
	`  iteration_num >= 1 &`
	`  iteration_num <= RECORD_COUNT &`
	`  iteration_state = FLD2_STRIP &`
	`  fld2aux /= [] & size(fld2aux) /= 2`
	`THEN`
	`  fs <-- binary_split(fld2aux,`
	`CHAR_COMMA);`
	`  compA:= fs(1);`
	`  compB:= fs(2);`
	`  iteration_state:= COMP_AB_UPDATE`
	`END;`

Предусловием каждой из операций является то, что значение переменной *iteration_num* не превышает RECORD_COUNT: это означает, что еще не все записи из входных данных обработаны. Для управления последовательностью вызова операций определена вспомогательная переменная *iteration_state*:

```
SETS
  ITERATION_STATE = { INITIAL, COMP_AB_INIT, FLD2_STRIP, COMP_AB_UPDATE,
                      ... COMPLETED }
ABSTRACT_VARIABLES
  iteration_state
INVARIANT
  iteration_state: ITERATION_STATE
```

Эта переменная принимает значения из множества состояний ITERATION_STATE (выше приведены лишь некоторые из состояний). Так, операция *pre_iteration_CompStrip* может быть выполнена, только если переменная *iteration_state* принимает значение COMP_AB_INIT. После выполнения операции переменная принимает значение

FLD2_STRIP. После этого может быть выполнена операция *pre_iteration_CompSimpleUpdate* и т.д.

В телах операций происходит присваивание значений вспомогательным переменным, вызов операций *strip, binary_split* спецификации *StringType.mch*, выражающих семантику методов *strip, split* языка Python соответственно. Ветви внешнего оператора *if-else* представлены отдельными операциями. Семантика последовательной композиции операторов Python представляется при помощи управления последовательностью вызова операций AMN и последовательной подстановки «;» языка AMN [35].

Полный граф состояний программы интеграции каталога WDS и переходов между ними при помощи операций AMN изображен на рисунке Б.2. Кратко предназначение операций описано в таблице Б.7.

Таблица Б.7 – Предназначение операций AMN

Операции AMN	Предназначение операций
pre_iteration_CompInit pre_iteration_CompStrip pre_iteration_CompSimpleUpdate pre_iteration_CompSplit	Операции выделения идентификаторов компонентов пары
pre_iteration_band_blue pre_iteration_band_ir pre_iteration_band_red	Операции определения спектральной полосы компонента пары
pre_iteration_pmC	Операция определения коэффициента собственного движения
pre_iteration_otype_S pre_iteration_otype_U pre_iteration_otype_Y	Операции определения оптического типа пары
pre_iteration_fld12_split pre_iteration_SpType	Операции определения спектрального типа компонентов
pre_iteration_RADE pre_iteration_Angles pre_iteration_rel2abs	Операция вычисления прямого восхождения и склонения компонентов пары
create_system	Операция вставки кортежа в таблицу звездных систем
create_compA create_compB	Операции вставки кортежей в таблицу компонентов
create_pair	Операция вставки кортежа в таблицу пар
iteration_last_wds finalize_iteration	Операции финализации обработки записи исходных данных

Семантика приведенного выше фрагмента *Iteration-code* выражается при помощи операции *create_compA*:

```
create_compA =
SELECT
    iteration_num >= 1 & iteration_num <= RECORD_COUNT &
    iteration_state = SYSTEM_CREATED
THEN
  VAR acatline_id IN
    acatline_id <-- insert_catline(fld0(iteration_num)^compA);
    insert_comp(acatline_id, fld10(iteration_num), spBand, sptypeA,
    pmC*real(fld13(iteration_num)), pmC*real(fld14(iteration_num)),
    RA1, DE1, last_sys_catline_id);
    last_compA_catline_id:= acatline_id
  END;
  iteration_state:= COMP_A_CREATED
END;
```

Рисунок Б.2 – Граф состояний и переходов программы интеграции

Операция *create_compA,* в свою очередь, вызывает операцию *insert_catline* вставки кортежа в сводную таблицу записей каталогов *bdb3_models_catline* и операцию *insert_comp* вставки кортежа в таблицу компонентов *bdb3_models_bdbcomp*. При этом атрибутам кортежей присваиваются соответствующие значения вспомогательных переменных или выражений.

Таким образом, основные принципы представления семантики императивных программ интеграции данных на языке Python в AMN состоят в следующем:

– семантика библиотечных модулей языка Python, групп методов встроенных типов, наборов вспомогательных функций выражается при помощи отдельных спецификаций вида MACHINE. Семантика простых методов выражается при помощи функциональных констант и их свойств. Семантика более сложных методов выражается при помощи операций AMN;

– семантика программы интеграции одного источника данных выражается отдельной спецификацией вида REFINEMENT;

– семантика последовательной композиции небольшого логически связанного последовательного набора операторов выражается отдельной операцией AMN;

– семантика последовательной композиции логических групп операторов выражается при помощи управления последовательностью вызова операций AMN с использованием переменной состояния программы;

– вызов простого библиотечного или вспомогательного метода моделируется термом с использованием соответствующей функциональной константы;

– вызов сложного библиотечного или вспомогательного метода моделируется вызовом соответствующей операции;

– семантика оператора присваивания значения переменной выражается подстановкой присваивания [35] либо вызовом операции;

– семантика относительно простого условного оператора выражается при помощи условной подстановки IF-THEN;

– семантика более сложного условного оператора выражается при помощи нескольких операций AMN, соответствующих отдельным ветвям оператора.

Верификация программ интеграции данных в базе данных двойных звезд. Свойства корректности программ интеграции данных, подлежащие верификации, определяются экспертом вручную в виде формул, конъюнктивно присоединяемых в раздел INVARIANT спецификации, выражающей семантику программы интеграции данных. Например, фрагмент свойства корректности интеграции каталога WDS в BDB выглядит следующим образом:

```
((iteration_state = COMPLETED) =>
!(ii).(ii: INT & ii >= 1 & ii <= RECORD_COUNT =>
#(asystem, apair, acompA, acompB, asystem_catline, apair_catline,
acompA_catline, acompB_catline).(
  asystem: bdb3_models_bdbsystem &
```

```
  apair: bdb3_models_bdbpair &
  acompA: bdb3_models_bdbcomp &
  acompB: bdb3_models_bdbcomp &

  apair'sys_catline_id = asystem'catline_id &
  acompA'sys_catline_id = asystem'catline_id &
  acompB'sys_catline_id = asystem'catline_id &
  apair'compA_catline_id = acompA'catline_id &
  apair'compB_catline_id = acompB'catline_id &

  asystem_catline: bdb3_models_catline &
  apair_catline: bdb3_models_catline &
  acompA_catline: bdb3_models_catline &
  acompB_catline: bdb3_models_catline  &

  asystem_catline'cat_id = asystem'catline_id &
  apair_catline'cat_id = apair'catline_id &
  acompA_catline'cat_id = acompA'catline_id &
  acompB_catline'cat_id = acompB'catline_id &

  asystem_catline'obj_id = fld0(ii) &
  acompA'mag = fld10(ii) &
  acompB'mag = fld11(ii) &

  #(str).(str: seq(CHAR) & apair_catline'obj_id = fld0(ii)^str) &
  #(str).(str: seq(CHAR) & acompA_catline'obj_id = fld0(ii)^str) &
  #(str).(str: seq(CHAR) & acompB_catline'obj_id = fld0(ii)^str)
)))
```

Формула утверждает, что если программа находится в состоянии COMPLETED (все записи из исходных данных обработаны), то для любой записи из исходных данных (номер которой задается переменной *ii*) в целевой базе данных существуют такие звездная система *asystem*, пара *apair*, два компонента (*acompA, acompB*) и соответствующие им кортежи в таблице записей каталогов *bdb3_models_catline*, что пара принадлежит системе, компоненты принадлежат паре, и значения их атрибутов соответствуют значениям атрибутов из исходных данных (например, значения *mag* звездных величин компонентов совпадают со значениями атрибутов *mag1* и *mag2* в исходных данных, идущих десятым и одиннадцатым по счету).

Полная формула, выражающая корректность интеграции каталога WDS в базе данных двойных звезд, была добавлена в раздел INVARIANT спецификации *WDS_Ref.ref*. Спецификации *StringType.mch, Math.mch, Conversions.mch, BDB.mch, WDS_Ref.ref* были загружены в проект инструментария Atelier B 4.7.1, автоматически были сгенерированы теоремы корректности спецификаций, применены средства автоматического и интерактивного доказательства с участием эксперта. Статистика доказательства приведена в таблице 8. Около 90% теорем удалось доказать автоматически.

Таблица Б.8 – Количество сгенерированных и автоматически доказанных теорем непротиворечивости спецификаций.

Спецификация	Количество сгенерированных теорем	Количество автоматически доказанных теорем при режиме доказательства Automatic	
		Force Fast	Force 0
StringType.mch	51	24	49
Math.mch	6	6	6
Conversions.mch	13	8	11
BDB.mch	37	4	19
WDS_Ref.ref	348	129	324

ПРИЛОЖЕНИЕ В

Список свидетельств о регистрации программ

1 Программа автоматического отображения спецификаций канонической информационной модели в Нотацию Абстрактных Машин: Свидетельство об официальной регистрации программы для ЭВМ № 2005611080 / С. А. Ступников. Зарегистрировано в Реестре программ для ЭВМ Федеральной службы по интеллектуальной собственности РФ 05.05.2005.

2 Программа загрузки спецификаций канонической информационной модели в репозиторий метаинформации: Свидетельство об официальной регистрации программы для ЭВМ №:2005611083 / С. А. Ступников. Зарегистрировано в Реестре программ для ЭВМ Федеральной службы по интеллектуальной собственности РФ 05.05.2005.

3 Программа загрузки спецификаций на языке UML XMI в репозиторий метаинформации с одновременным преобразованием в каноническую информационную модель: Свидетельство об официальной регистрации программы для ЭВМ N 2007613885 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 12.09.2007.

4 Программа загрузки схем реляционных баз данных в репозиторий метаинформации с одновременным преобразованием в каноническую информационную модель: Свидетельство об официальной регистрации программы для ЭВМ N 2007613884 / С.А. Ступников, А.Е. Вовченко. Зарегистрировано в Реестре программ для ЭВМ 12.09.2007.

5 Графический интерфейс Конструктора унифицирующих информационных моделей: Свидетельство об официальной регистрации программы для ЭВМ N 2008614240 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 05.09.2008.

6 Программа построения эталонных схем информационных моделей на основании формальных грамматик моделей: Свидетельство об официальной регистрации программы для ЭВМ N 2008614239 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 05.09.2008.

7 Генератор заготовок трансляторов информационных моделей: Свидетельство об официальной регистрации программы для ЭВМ N 2008614241 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 05.09.2008.

8 Генератор заготовок ATL-трансформаций информационных моделей. Свидетельство о государственной регистрации программы для ЭВМ № 2012610640 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 10.01.12.

9 Генератор заготовок обратных ATL-трансформаций информационных моделей. Свидетельство о государственной регистрации программы для ЭВМ № 2012610642 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 10.01.12.

10 Трансформация спецификаций языка СИНТЕЗ в Нотацию Абстрактных Машин. Свидетельство о государственной регистрации программы для ЭВМ № 2012611337 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 02.02.12.

11 Абстрактный и конкретный синтаксис языка СИНТЕЗ. Свидетельство о государственной регистрации программы для ЭВМ № 2012610641 / С.А. Ступников. Зарегистрировано в Реестре программ для ЭВМ 10.01.12.

12 С.А. Ступников. Трансформация программ языка High-level Integration Language (HIL) в Нотацию абстрактных машин. Свидетельство о государственной регистрации программы для ЭВМ № 2018612285 от 14.02.2018.

ПРИЛОЖЕНИЕ Г

Список научно-исследовательских работ, в рамках которых были использованы результаты диссертационного исследования

Проекты Российского научного фонда:

– 22-21-00692, Разработка и реализация расширяемого подхода к интеграции больших данных в распределенной вычислительной среде, 2022-2023.

Проекты Российского фонда фундаментальных исследований:

– 19-07-01198 «Моделирование на основе систем гипотез для решения обратной задачи - восстановления истории образования двойных звёздных систем в Галактике», ИНАСАН, 2019-2021 гг.

– 18-29-22096, «Методы и средства решения задач анализа данных в распределенных вычислительных инфраструктурах в области нейрофизиологии», ФИЦ ИУ РАН, 2019-2021 гг.

– 18-07-01434, Методы и средства организации экспериментов в движимых гипотезами исследованиях в областях с интенсивным использованием данных, ФИЦ ИУ РАН, 2018-2020.

– 15-29-06045, Извлечение информации из разноструктурированных данных для решения задач информационной поддержки поисковых действий в арктической зоне, НИЯУ МИФИ, 2015-2017 гг.

– 14-07-00548, Трансформация средствами Hadoop-образных инфраструктур контента больших разно-структурированных коллекций данных в структурированную, агрегированную информацию, ФИЦ ИУ РАН, 2014-2016 гг.,

– 13-07-0057, Методы и средства интеграции неоднородных (структурированных и слабоструктурированных) баз данных в системах с интенсивным использованием данных, ФИЦ ИУ РАН, 2013-2015.

– 11-07-00402-а, Интеграция декларативных программ и знаний на правилах, баз данных и сервисов для решения научных задач над множеством неоднородных распределенных информационных ресурсов, ФИЦ ИУ РАН, 2011-2013 гг.

– 10-07-00640-а, Создание грид-инфраструктуры и разработка программных комплексов для решения ресурсоемких вычислительных задач и задач обработки данных, МГУ им. М. В. Ломоносова, 2010-2012 гг.

– 10-07-00342-а, Создание методов и средств определения концептуальных моделей предметных областей в научных исследованиях и поддержки решения научных задач на основе предметных посредников в гибридной грид-инфраструктуре, ИПИ РАН, 2010-2012 гг.

– 08-07-00157-а, Исследование и разработка методов и средств семантической идентификации релевантных научной задаче спецификаций неоднородных распределенных информационных ресурсов и их интеграции в спецификациях задачи в информационных системах для научных исследований, ИПИ РАН, 2008-2010 гг.

– 06-07-08072-офи, Разработка и создание опытного образца конструктора унифицирующих моделей представления информации для интероперабельных систем интеграции неоднородных источников информации, ИПИ РАН, 2006-2007 гг.

– 06-07-89188-а, Методы и средства поддержки архитектуры предметных посредников в инфраструктуре AstroGrid для Российской Виртуальной Обсерватории, ИПИ РАН, 2006 – 2008;

– 05-07-90413-в, Методы организации решения задач над множественными распределенными неоднородными источниками информации, ИПИ РАН, 2005 -2006 гг.

– 04-07-90083-в, Разработка принципов и основ информационного взаимодействия в инфраструктуре Российской Виртуальной Обсерватории (РВО), 2004.

– 03-01-00821-а, Моделирование композиционных спецификаций для автоматизированного доказательства корректности уточнения спецификаций требований существующими компонентами при композиционном проектировании информационных систем, ИПИ РАН, 2003–2004 гг.

– 01-07-90084-в, Методы и средства проектирования предметно-ориентированных посредников неоднородных информационных коллекций в распределенных электронных библиотеках, ИПИ РАН, 2001–2003 гг.

Проекты Минобрнауки РФ:

– НИР в рамках соглашения № 075-15-2020-805 о Консорциуме по выполнению исследования по теме «Актуальные научные задачи стратегии адаптации потенциала землепользования России в современных условиях беспрецедентных вызовов (экономический кризис, изменения климата, кризис глобальных тенденций природопользования)», ФГБНУ ФИЦ «Почвенный институт им. В.В Докучаева», 2020-2023.

– Проект по Соглашению с Минобрнауки № 14.607.21.0176 «Разработка информационной системы поддержки технического обслуживания и предиктивного

ремонта объектов жилищно-коммунальной инфраструктуры в рамках концепции Интернета вещей», ФИЦ ИУ РАН, 2017-2018.

– Разработка комплекса программных средств, предназначенных для создания унифицированной среды, обеспечивающей интеграцию разнородных электронных коллекций в электронную библиотеку. Договор № 990-23-и от 01 июля 2001 г. по направлению «Создание программного обеспечения для информационных технологий». ИПИ РАН.

Проект 1-10 «Семантические модели неоднородной информации и их уточнения в композиционных архитектурах» программы «Фундаментальные основы информационных технологий и систем» Отделения информационных технологий и вычислительных систем (Отделения нанотехнологий и информационных технологий) РАН, ИПИ РАН, 2004-2008:

– Методы конструирования реверсивных отображений моделей представления информации в композиционных инфраструктурах информационных систем, этап 2007-2008 гг.

– Онтологическое моделирование для обеспечения семантической композиции и интеграции неоднородных источников информации, этап 2006 г.

– Семантические композиции неоднородной информации в информационных системах, этап 2005 г.

– Разработка методов синтеза канонической информационной модели на основе теории уточнения, этап 2004 г.

Проекты Программы фундаментальных исследований Президиума РАН № 16 «Фундаментальные проблемы системного программирования», раздел программы: 4. Системы управления базами данных, ИПИ РАН:

– Концептуализация потоков работ в мультидиалектной инфраструктуре интероперабельных декларативных спецификаций для их (потоков работ) повторного использования в областях с интенсивным использованием данных, 2014.

– Обеспечение повторного использования реализаций методов анализа информации и алгоритмов решения задач в научных областях с интенсивным использованием данных», 2013.

– Исследование методов и средств промежуточного слоя предметных посредников, обеспечивающего решение задач над множеством неоднородных распределенных информационных ресурсов, 2009-2012.

НИР, выполняемые в ФИЦ ИУ РАН:

– 0063-2019-0009, Методы и программные средства накопления и обработки данных, 2019-2025 гг.

– 0063-2016-0010, Методы и программные средства накопления и обработки данных, 2017-2019 гг.

– Спецификация и решение задач анализа данных в концептуальных терминах предметных областей с интенсивным использованием данных на основе Big Data - ориентированных ИТ, 2014-2016 гг.

– Исследование, разработка и практическая проверка на реальных данных разнообразной природы методов и средств анализа текстов на основе технологий Big Data, 2014 г.

– Изучение задач, архитектуры, компонентов систем анализа, обработки, хранения больших данных разнообразной природы и представления, 2013 г.

– Развитие средств спецификации, архитектуры и методов поддержки среды предметных посредников, 2011-2013 гг.

– Интеграция спецификаций неоднородных распределенных информационных ресурсов в спецификациях задач в информационных системах, 2008-2010 гг.

– Решение задач в архитектуре предметных посредников, сопряженных с инфраструктурой АстроГрид, 2008.

– Реализация гибридной инфраструктуры предметных посредников и AstroGrid для решения задач Российской Виртуальной Обсерватории (РВО) над множеством неоднородных, распределенных информационных ресурсов, 2006-2007 гг.

– Поиск и регистрация источников информации, релевантных спецификациям предметной области задач в научных исследованиях, 2006 – 2007 гг.

– Анализ архитектуры предметных посредников в инфраструктуре AstroGrid и ее реализация, 2005 – 2006.

– Композиционные методы решения задач в инфраструктурах интеграции информации для научных исследований, 2004 -2005 гг.

– Контекстуализация неоднородных информационных источников в предметном информационном посреднике, 2001 – 2003 гг.

ПРИЛОЖЕНИЕ Д

Акты внедрения и использования результатов диссертационного исследования

«УТВЕРЖДАЮ»
И.о. директора
ФГБНУ ФИЦ «Почвенный институт им. В.В. Докучаева»
Д.Н. Козлов
03 июня 2026 г.

АКТ
о внедрении результатов диссертационной работы
Ступникова Сергея Александровича
«Методы и средства верифицируемой интеграции неоднородных данных»,
представленной на соискание ученой степени доктора технических наук
по специальности 2.3.5 «Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей»

Рассмотрев основные результаты, выводы и рекомендации диссертационной работы ведущего научного сотрудника Федерального исследовательского центра «Информатика и управление» РАН (далее – ФИЦ ИУ РАН), комиссия в следующем составе:

председатель – и.о. первого заместителя директора ФГБНУ ФИЦ «Почвенный институт им. В.В. Докучаева», д.б.н. А.Г. Болотов;
члены комиссии:
– заведующий Ситуационным аналитическим центром ФГБНУ ФИЦ «Почвенный институт им. В.В. Докучаева», канд. биол. наук Д.С. Фомин;
– науч. сотр. ФГБНУ ФИЦ «Почвенный институт им. В.В. Докучаева» П.Р. Цымбарович

установила, что С.А. Ступников с 2020 г. по 2023 г. являлся ответственным исполнителем от ФИЦ ИУ РАН крупного научного проекта «Актуальные научные задачи стратегии адаптации потенциала землепользования России в

современных условиях беспрецедентных вызовов (экономический кризис, изменения климата, кризис глобальных тенденций природопользования)» подпрограммы «Фундаментальные научные исследования для долгосрочного развития и обеспечения конкурентоспособности общества и государства» государственной программы Российской Федерации «Научно-технологическое развитие Российской Федерации» (Соглашение с Минобрнауки России № 075-15-2020-805 от 02.10.2020 г.) и результаты диссертации С.А. Ступникова в виде:

– метода конструирования сохраняющих семантику отображений моделей данных;

– метода верификации интеграции схем данных в системах виртуальной интеграции;

– метода верификации интеграции данных в системах материализованной интеграции данных;

– программных средств автоматизации методов интеграции данных

внедрены ФГБНУ ФИЦ «Почвенный институт им. В.В. Докучаева» при создании Интеллектуальной системы локального планирования землепользования и агротехнологий «Умное землепользование».

Председатель комиссии _____ / А.Г. Болотов
03.06.2026г.

Члены комиссии: _____ 03.06.2026 / Д.С. Фомин

_____ 03.06.2026 / П.Р. Цымбарович

«УТВЕРЖДАЮ»
Директор
Федерального государственного бюджетного учреждения науки
Институт астрономии Российской академии наук

Профессор РАН
М.Е. Сачков
_____ 2026 г.

АКТ

о внедрении результатов диссертационной работы
Ступникова Сергея Александровича
«Методы и средства верифицируемой интеграции неоднородных данных»,
представленной на соискание ученой степени доктора технических наук
по специальности 2.3.5 «Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей»

Рассмотрев основные результаты, выводы и рекомендации диссертационной работы ведущего научного сотрудника Федерального исследовательского центра «Информатика и управление» РАН, комиссия в следующем составе:

председатель – заведующий Отделом физики звездных систем ИНАСАН, д. ф.-м.н. О.Ю. Малков;

члены комиссии:

– вед. науч. сотр. ИНАСАН к.ф.-м.н. Д.А. Ковалева;

– ст. науч. сотр. ИНАСАН к.ф.-м.н. П.В. Кайгородов –

установила, что результаты диссертации С.А. Ступникова в виде

– метода конструирования сохраняющих семантику отображений моделей данных;

– метода верификации интеграции данных в системах материализованной интеграции данных

внедрены в практику ИНАСАН, использованы при выполнении работ по проекту Российского фонда фундаментальных исследований 19-07-01198 «Моделирование на основе систем гипотез для решения обратной задачи - восстановления истории образования двойных звёздных систем в Галактике» и применены при верификации корректности интеграции данных в базе данных двойных звезд ИНАСАН.

Председатель комиссии /О.Ю. Малков/
Члены комиссии: _____ /Д.А. Ковалева/
_____ /П.В. Кайгородов/

«УТВЕРЖДАЮ»
Директор
Института металлургии и материаловедения
им. А. А. Байкова РАН
член-корреспондент РАН
А. А. Комлев
7 мая 2026 г.

АКТ

о внедрении результатов диссертационной работы
Ступникова Сергея Александровича
«Методы и средства верифицируемой интеграции неоднородных данных»,
представленной на соискание ученой степени доктора технических наук
по специальности 2.3.5 «Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей»

Рассмотрев основные результаты, выводы и рекомендации диссертационной работы ведущего научного сотрудника Федерального исследовательского центра «Информатика и управление» РАН, комиссия в следующем составе:

председатель – руководитель группы применения информационных технологий в материаловедении, зав. лабораторией полупроводниковых материалов ИМЕТ РАН, доктор химических наук Н.Н. Киселева;

члены комиссии:

– ст. науч. сотр. ИМЕТ РАН к.ф.-м.н. А.А. Докукин;

– науч. сотр. ИМЕТ РАН к.т.н. А.В. Столяренко -,

установила, что результаты диссертации С.А. Ступникова в виде *метода верификации интеграции данных в системах материализованной интеграции данных* внедрены в практику ИМЕТ РАН и применены для верификации корректности интеграции данных в интегрированной системе баз данных по свойствам неорганических веществ и материалов ИМЕТ РАН.

Председатель комиссии _____ /Н.Н. Киселева/

Члены комиссии _____ /А.А. Докукин/

_____ /А.В. Столяренко/

ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ «ПАЛЛАДА»
(ООО «ПАЛЛАДА»)
121357, г. Москва, Кутузовский пр. д 67, к.1
Тел.: + 7 (495) 440-83-05,
Факс: + 7 (495) 440-83-05,
email: info@geopallada.ru,
http: www.geopallada.ru

Исх. 76-нс № от 06.04.2026г.
На

Председателю
диссертационного совета
24.1.224.04

Академику РАН
И.А. Соколову

О внедрении результатов диссертации

Уважаемый Игорь Анатольевич!

Результаты диссертационной работы Ступникова Сергея Александровича «Методы и средства верифицируемой интеграции неоднородных данных» на соискание ученой степени доктора технических наук по специальности 2.3.5 «Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей» в виде

– метода верификации интеграции данных в системах материализованной интеграции данных; – семантической трансформации языка программ интеграции данных HIL в формальный язык спецификаций AMN

использованы при выполнении работ по проекту Российского фонда фундаментальных исследований 15-29-06045 «Извлечение информации из разноструктурированных данных для решения задач информационной поддержки поисковых действий в арктической зоне» и применены для верификации корректности интеграции данных в хранилище информации для поддержки поисковых действий в арктической зоне.

Генеральный директор _____/Н.Н. Сметанин/

«УТВЕРЖДАЮ»

Зам. декана по научной работе
факультета ВМК МГУ
имени М.В.Ломоносова

В.В. Фомичев

_____ 2026 г.

АКТ

об использовании результатов диссертационной работы
Ступникова Сергея Александровича
«Методы и средства верифицируемой интеграции неоднородных данных»,
представленной на соискание ученой степени доктора технических наук
по специальности 2.3.5 «Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей»

Рассмотрев основные результаты, выводы и рекомендации диссертационной работы ведущего научного сотрудника Федерального исследовательского центра «Информатика и управление» РАН, комиссия в следующем составе:

председатель:

– заведующий Лабораторией открытых информационных технологий кафедры информационной безопасности ВМК МГУ, д.т.н., профессор В.А. Сухомлин;

члены комиссии:

– ведущий научный сотрудник Лаборатории открытых информационных технологий кафедры информационной безопасности ВМК МГУ, д.т.н., Д.Е. Намиот;

– научный сотрудник Лаборатории открытых информационных технологий кафедры информационной безопасности ВМК МГУ, к.п.н., доцент А.В. Якушин

установила, что результаты диссертации С.А. Ступникова в виде *метода конструирования сохраняющих семантику отображений моделей данных* использованы в учебном процессе ВМК МГУ при чтении курса «Виртуальная интеграция неоднородных данных и унификация моделей данных» магистерской программы «Большие данные: инфраструктуры и методы решения задач».

Председатель комиссии _____/В.А. Сухомлин/

Члены комиссии: _____/Д.Е. Намиот/

_____/А.В. Якушин/