ТриниДата

Корпоративные автоматизированные системы на основе онтологических моделей: книга рецептов

ООО «ТриниДата», 2020

Оглавление

Все больше корпоративных автоматизированных систем строятся сегодня с использованием онтологий — формализованных, машинно-читаемых моделей знания. Набор необходимых для этого технологий содержится в спецификациях консорциума W3C: язык моделирования OWL, язык запросов SPARQL, средства создания правил контроля логической целостности и получения логического вывода SHACL. Созданы онтологии верхнего уровня, предоставляющие инструментарий для моделирования самых разных предметных областей, такие как BFO, GFO, Dolce. Программное обеспечение, поддерживающее использование таких технологий, создают вендоры со всего мира, среди которых компании TopQuadrant, Semantic Web Company, Ontotext и многие другие. На основе их продуктов строятся прикладные решения для крупнейших мировых корпораций — системы управления знаниями, системы сбора и анализа информации, ситуационные центры и многое другое.

Компания ТриниДата создала платформу АрхиГраф, позволяющую использовать всю мощь онтологий для создания корпоративных автоматизированных систем, используя мировые стандарты и распространенные методики моделирования. Наши инструменты обеспечивают интероперабельность созданных моделей с другими программными средствами,обеспечивая в то же время ряд дополнительных функций, необходимых в конкретных прикладных решениях. В течение шести лет мы накопили некоторый опыт эффективной разработки таких решений, и хотим поделиться им в форме «книги рецептов» — практического руководства, которое собирает воедино набор технологий, методик и инструментов, необходимых для создания современных и эффективных программных решений, и хотим поделиться им в форме «книги рецептов» — практического руководства, которое собирает воедино набор технологий, методик и инструментов, необходимых для создания современных и эффективных программных решений.

В этом документе мы рассмотрим только один из сценариев применения онтологий в корпоративных автоматизированных системах, но надеемся, что читатель самостоятельно применит их для проектирования решений в своей прикладной области.

Нашей задачей не является рассказ об онтологиях как таковых или погружение в детали технологий работы с ними. Для знакомства с их основами мы рекомендуем наше методическое пособие «Введение в онтологическое моделирование»[1] и монографию «Онтологическое моделирование предприятий: методы и технологии»[2] (в одной из глав которой описывается несколько сценариев встраивания онтологических инструментов в архитектуру автоматизированной системы). В этом документе мы постарались дать как обзор функциональных возможностей самих технологий и методов онтологического моделирования, так и конкретные прикладные рецепты их использования с помощью нашей платформы. Другими словами, мы старались ответить сразу на несколько вопросов читателя — архитектора ИТ-систем: зачем и как именно можно применить онтологии в бизнес-приложении.

1 https://trinidata.ru/files/SemanticIntro.pdf

2 https://trinidata.ru/files/EnterpriseModeling.pdf

Зачем нужны онтологии в корпоративных системах?

Онтологии в автоматизированных системах используются и как средство представления структуры данных (и самих данных), и как инструмент представления логики (алгоритма) выполнения программы. При этом онтология находится вне программного кода – а значит, структуру данных и логику выполнения приложения можно будет изменять без вмешательства в его код, ш и даже без перезапуска.

Скептически настроенный читатель возразит: существует множество платформ разработки приложений, которые позволяют конфигурировать и настраивать и структуру данных, и логику выполнения. Чем разработка с помощью онтологий принципиально отличается от использования какой-нибудь из таких платформ?

Отличий множество. Способ представления мира, реализованный в методиках и спецификациях онтологического моделирования, прочно связан с философскими, математическими и логическими основаниями – дескрипционной логикой, теорией множеств и др. Это дает возможность при построении моделей абстрагироваться от деталей программной реализации средств хранения данных, а ориентироваться на воспроизведение прежде всего концептуальных представлений бизнес-пользователей о предметной области (включая отражение разных точек зрения на одни и те же объекты и процессы). Модели, построенные таким образом, при грамотном подходе к моделированию содержат не так много искажений, отклонений структуры данных от представлений бизнес-пользователей, которые легко было бы внести архитектору данных при работе, например, с реляционными СУБД. Такие искажения, внесенные на стадии проектирования, дорого обходятся при сопровождении и модернизации программного продукта. Изначальная установка онтологий на гибкость и изменчивость модели, напротив, максимально снижает сроки истоимость любых возможных доработок.

Онтологические модели не связаны с какими-то определенными программными платформами – обрабатывать их может широкий спектр программных средств. Это значит, что созданные с использованием онтологий приложения не привязаны к тем или иным конкретным программным решениям, не зависят от их версий или политики вендоров: в любой момент можно переключить уже готовую программную систему на использование других, более современных средств хранения и обработки онтологий. И, конечно, разработчики приложений, использующих онтологии, не связаны ограничениями в части использования тех или иных языков программирования, операционных систем, средств разработки.

Функциональные преимущества создания автоматизированных систем на основе онтологий становятся очевидными, если при проектировании во главу угла ставится способность автоматизированной системы к изменениям. В современном мире постоянно изменяется среда, в которой происходит деятельность предприятий: меняются регуляторные требования, запросы потребителей, стремительно появляются и исчезают продукты и услуги, изменяются связанные с ними процессы. Другими словами – бизнес-объекты и процессы, в которых они участвуют, предельно нестабильны, и любой элемент бизнес-логики, жестко сформулированный в коде или структуре данных приложения, завтра может стать тем препятствием, которое не позволит предприятию адаптироваться к изменениям, завоевать новый рынок или просто остаться на плаву.

Для дальнейшего рассказа нам нужно выбрать пример, который позволит продемонстрировать описанные преимущества, но будет достаточно простым, чтобы не требовать многословных объяснений. Мы рассмотрим несложную автоматизированную систему, которая представляет собой «ситуационный центр» здания, способный собирать сигналы с датчиков температуры и влажности, а также тревоги от охранной и пожарной сигнализации. Предположим, что первоначально система создается для обработки только этих сигналов, но в дальнейшем планируется ее расширение для сбора и обработки множества других данных по зданию, включая состояние различного оборудования, сообщения от персонала и другие сведения. При проектировании нужно принимать во внимание, что в здании будут проводиться мероприятия, организаторы которых могут предъявлять собственные требования в части контроля инцидентов и реагирования на них. Это значит, что автоматизированная система не может ограничиваться ни жестко определенным набором видов оборудования, состояние которого отслеживается и сигналы с которого принимаются, ни определенными типами инцидентов, ни регламентами и участниками реагирования. Пользователями системы в будущем могут стать как сотрудники новых служб здания, так и представители организаторов мероприятий, функциональные требования которых не известны заранее. Могут появиться и новые регуляторные требования, связанные с обеспечением безопасности, которые потребуют обработки новых видов данных или создания новых отчетов.

Поскольку предприятие не располагает средствами и временем для реализации всех этих возможных изменений в будущем, оно максимально заинтересовано в создании как можно более гибкой системы, которую смогут настраивать по ходу эксплуатации аналитики, а не программисты. Это побуждает предприятие поставить задачу использования таких программных средств, которые обеспечат легкую интеграцию с новыми источниками данных, настройку пользовательских интерфейсов и отчетов, изменение набора типов обрабатываемых событий и регламентов их обработки без вмешательства в программный код. Все это мотивирует компанию, реализующую автоматизированную систему в соответствии с этими требованиями, использовать средства работы с онтологиями и вынести на уровень онтологической модели как можно большую часть логики работы программы.

Тому, как реализовать это на практике, посвящена оставшаяся часть этого документа.

Структура данных

В наших автоматизированных системах структура данных является частью онтологии, поэтому онтологическая модель можно рассматривать как «стержень», вокруг которого строится все прикладное решение. От того, насколько корректно будет создана такая модель, зависит легкость модификации приложения в будущем и те пределы, до которых будет возможно изменение функционала системы без вмешательства в ее код.

Не существует общепринятых жестких правил, по которым должны строиться онтологии. Есть несколько «школ», предлагающих те или иные методики моделирования, но каждая из них эффективна для решения только определенного диапазона задач; кроме того, следование слишком жестко формализованным методикам может лишить решение гибкости и привести к невозможности использовать те или иные преимущества средств работы с онтологиями.

Существует также широкий набор готовых онтологий, описывающих разные предметные области: FOAF описывает людей и их отношения, SSN и SOSA – сенсоры, сигналы и измерения, QUDT – единицы измерения, и др. При создании онтологии для автоматизированной системы можно импортировать эти онтологии или их фрагменты: это избавит от необходимости моделировать то, что уже было кем-то смоделировано, облегчит возможную будущую интеграцию с другими системами. С другой стороны, мы не считаем использование максимального набора готовых онтологий самоцелью, поскольку это может усложнить структуру модели, противоречить практическим целям создания системы, и в конечном счете заставить реализовывать в модели некие «компромиссные» решения, которые дорого обойдутся при модернизации системы.

Противоположным подходом является создание онтологии «с нуля» без всякой методики или каркаса, по принципу дескриптивного описания той части мира, которая должна быть смоделирована. Такой подход часто приводит к захламлению онтологии ненужными сущностями или недостаточно четкому их разделению, создает проблемы при расширении охвата онтологии, и в конечном счете мало чем отличается от критикуемых нами практик создания структур реляционных СУБД.

Аналитики нашей компании при создании онтологий обычно используют элементы стандартной онтологии BFO для описания верхнего уровня модели, импортируют части подходящих для решения задачи внешних онтологий, а оставшуюся часть модели достраивают самостоятельно. Такой подход не является единственно возможным и может быть изменен в зависимости от требований заказчика или практической целесообразности.

Онтология автоматизированной системы, решающей поставленную выше задачу создания ситуационного центра здания, должна включать следующие смысловые блоки:

- структура здания – этажи, помещения, ключевые конструктивные элементы;
- датчики и другие устройства, входящие в состав систем пожарной и охранной сигнализации;
- сигналы, которые могут генерироваться этими устройствами (значения измеряемых величин, тревоги и др.);
- события, о которых могут свидетельствовать такие сигналы (возгорания, нарушения режима безопасности объекта и др.);
- штатные должности, работники которых должны реагировать на такие события;
- регламенты реагирования, в соответствии с которыми они должны действовать.

Все перечисленные виды бизнес-объектов относятся прежде всего к реальному миру, то есть описывают происходящие в реальности события и их участников. Данные о конкретных объектах и событиях будут храниться в системе в соответствии со структурой, заданной соответствующими классами и свойствами онтологии. Кроме них, в онтологической модели должны быть представлены сущности, описывающие функционирование самой автоматизированной системы:

- экранные формы, состав входящих в них полей ввода и органов управления;
- правила преобразования сигналов, поступающих от устройств;
- действия, выполняемые автоматизированной системой в рамках регламентов реагирования на события (отправка уведомлений, передача управляющих сигналов на оборудование и др.);
- правила построения отчетных форм, экспорта данных во внешние системы;
- правила формирования аналитических дэшбордов, интерактивных карт и др.

Как мы увидим дальше, часть логики (алгоритмов) обработки данных будет представлена также в виде правил логического вывода. Таким образом, онтологии позволяют представлять логику выполнения автоматизированной системой каких-либо действий или преобразований как с помощью элементов самой онтологии, так и в виде правил логического вывода (правила, записанные с помощью формализма SHACL, технически сами также являются частью онтологии).

В платформе АрхиГраф создание онтологий выполняется с помощью редактора АрхиГраф.Мир. Мы рекомендуем познакомиться с обзором его функциональных возможностей, приведенным в соответствующем документе[1].

1 https://trinidata.ru/files/ArchiGraphMIRUserGuide.pdf

Чтобы начать работу с АрхиГраф.Мир, нужно выполнить минимально необходимый набор действий по конфигурированию АрхиГраф.MDM – платформы работы с данными, программный интерфейс которой АрхиГраф.Мир использует для чтения и записи данных онтологической модели и представленных в соответствии с ней данных. Отметим, что в составе демо-версии АрхиГраф.Мир и АрхиГраф MDM поставляются уже настроенными, и выполнять в демо-версии описанные дальше действия не нужно.

Прежде всего, нужно создать «точку доступа» – так в соответствии с протоколом SPARQL называется экземпляр графовой базы данных, доступ к которому обеспечивается через определенный URL программного интерфейса. Создание точки доступа выполняется с помощью следующей команды в консоли сервера, где установлена АрхиГраф.MDM:

```
mdmctl add endpoint Demo --prefix http://trinidata.ru/
demo/ --root http://trinidata.ru/demo
```

В этой команде Demo – название точки доступа, http://trinidata.ru/demo/ – префикс по умолчанию, http://trinidata.ru/demo – идентификатор корневого элемента онтологии, подклассами которого будут все остальные элементы.

Также должно быть создано основное хранилище, в которое АрхиГраф.MDM будет записывать структуру модели и представленные в соответствии с ней данные по умолчанию, пока не сконфигурированы другие хранилища (с этим мы познакомимся в следующей главе). При автоматическом конфигурировании создание хранилища заключается в развертывании экземпляра графовой СУБД с точкой доступа SPARQL, например Apache Fuseki, и его регистрации в конфигурационной базе данных АрхиГраф.MDM:

```
mdmctl add storage fuseki "Хранилище модели Demo" --host
127.0.0.1 --port 8080 --login mdm --password test
--query /fuseki/demo/query --update /fuseki/demo/update
```

В этой команде fuseki – тип создаваемого хранилища, «Хранилище модели Demo» – его название, 127.0.0.1 – IP-адрес хоста, 8080 – порт, mdm – логин для HTTP-авторизации, test – пароль для HTTP-авторизации, /fuseki/demo/query – путь к SPARQL-сервису чтения данных, /fuseki/demo/update – путь к SPARQL-сервису изменения данных.

Доступные в базовой поставке MDM типы хранилищ перечислены в Таблице 1.

Дополнительно поставляются также классы хранилищ для работы с Oracle, SQL Server, MySQL.

Затем нужно присоединить новое хранилище к ранее созданной точке доступа в качестве хранилища структуры модели (TBox) следующей командой:

```
mdmctl bind model_storage "Хранилище модели Demo" Demo
```

После выполнения перечисленных настроек редактор АрхиГраф.Мир, настроенный на работу с копией АрхиГраф.MDM, к которой подключено хранилище (см. главу «Развертывание»), готов к работе.

Таблица 1. Типы хранилищ, доступные в базовой поставке АрхиГраф.MDM

Идентификатор	Описание
Графовые СУБД	
fuseki	Apache Fuseki
allegrograph	AllegroGraph
blazegraph	BlazeGraph
Документ-ориентированные СУБД	
mongo	MongoDB через библиотеку MongoClient
mongodb	MongoDB через библиотеку MongoDB
mongodba	MongoDB через библиотеку MongoDB с поддержкой языковых версий значений литералов
Реляционные СУБД	
postgresql	Postgresql с хранением значений атрибутов в отдельных столбцах таблиц
postgresql_jsonb	Postgresql с хранением значений атрибутов в поле jsonb и отдельных столбцах таблиц для индексации
postgresql_multilang	Postgresql с хранением значений атрибутов в поле jsonb и отдельных столбцах таблиц для индексации, с поддержкой языковых версий значений литералов

Приведем примеры создания элементов онтологии, соответствующих перечисленным типам информационных объектов. Для этого прежде всего рассмотрим структуру онтологической модели, на основе которой будут приводиться примеры в этом документе.

В этом документе мы продемонстрируем создание фрагмента онтологической модели на основе онтологии верхнего уровня BFO[1], с использованием элементов стандартных онтологий SOSA, SSN и QUDT[2].

Онтологии верхнего уровня делят все сущее на классы, выделенные на основании фундаментальных, «жестких» (не изменяющихся на всем протяжении существования объекта) характеристик. На верхнем уровне BFO находятся классы с названиями «Continuant» и «Occurent». Этим терминам сложно подобрать адекватный перевод на русский язык, поэтому в русскоязычных моделях мы используем их транслитерации «Континуант» и «Оккурент». Общая фундаментальная характеристика для всех континуантов состоит в том, что они полностью явлены в каждый момент времени, могут стабильно существовать более или менее длительный период времени. Оккуренты, напротив, представляют собой изменения (процессы, события), которые разворачиваются во времени и не могут рассматриваться безотносительно него. Примерами континуантов в нашей модели будут являться приборы, входящие в состав пожарной и охранной сигнализаций, а примерами оккурентов – процессы измерений, выполняемых этими приборами, и сгенерированные ими тревоги.

Континуанты, в свою очередь, делятся на три подкласса – независимые, общезависимые и специфически зависимые; не вдаваясь здесь в философские основания такого деления, заметим, что оно связано с тем, существуют ли такие объекты независимо от других, либо способны проявляться (существовать) только относительно других объектов. Так, все материальные физические объекты являются независимыми континуантами, а различные информационные сущности – общезависимыми континуантами. Оккуренты также имеют внутреннее деление, на котором мы не будем здесь останавливаться; аналитикам, интересующимся этим вопросом, рекомендуем обратиться к статье Г. Ю. Лобанова «Basic Formal Ontology как средство построения онтологии в системной модели аргументации».

Процесс создания элементов онтологии в интерфейсе редактора АрхиГраф.Мир описан в документации пользователя на него[3], а также в документе «Знакомство с демо-версией АрхиГраф». Результат создания части дерева классов онтологии показан на рис. 1.

Классы онтологии могут иметь разные префиксы. При создании нового класса редактор предлагает использовать префикс по умолчанию, однако можно указать и URI класса полностью — эта возможность позволяет создавать элементы с идентификаторами, определенными во внешних онтологиях. Можно не создавать элементы внешней онтологии вручную, а импортировать файл онтологии в точку доступа Fuseki, где хранится TBox созда-

1 https://basic-formal-ontology.org/

2 https://www.w3.org/TR/vocab-ssn/

3 https://trinidata.ru/files/ArchiGraphMIRUserGuide.pdf

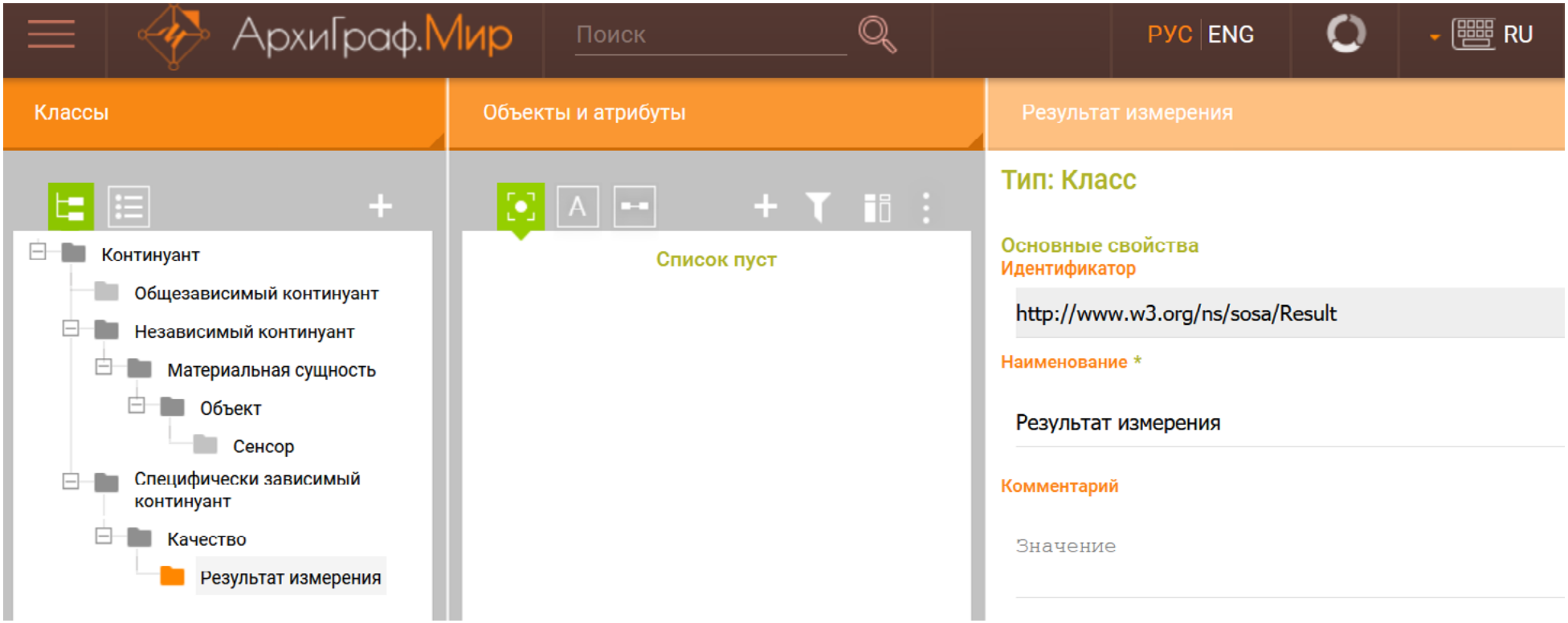
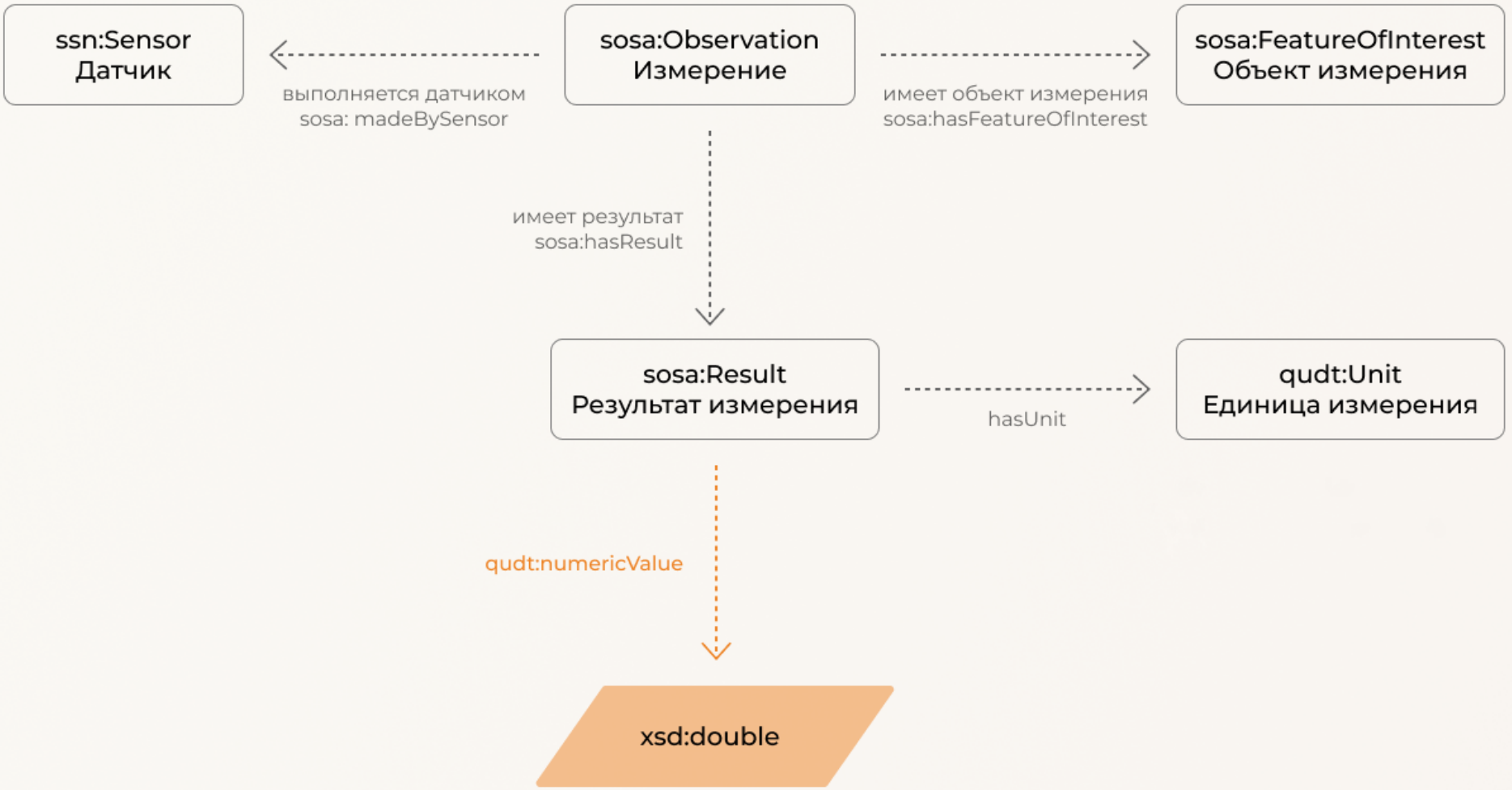


Рис. 1. Фрагмент дерева классов онтологии в интерфейсе редактора АрхиГраф.Мир

ваемой модели; в рассматриваемом примере мы не будем этого делать, потому что из всех используемых внешних онтологий нам нужно только небольшое число классов.

Кроме рассмотренных выше классов BFO, в этом фрагменте онтологии участвуют классы «Сенсор» (`ssn:Sensor`) и «Результат измерения» (`sosa:Result`), заимствованные из онтологий SSN и SOSA. Мы определили для этих классов место в онтологии BFO, поскольку сами онтологии SSN и SOSA не наследуют ни одну из онтологий верхнего уровня. Когда потребуется создать более специализированные подклассы в этих классах, например «Датчик температуры» или «Сигнализатор задымления» как подклассы класса «Сенсор», мы будем использовать префикс нашей онтологии по умолчанию. Использование префиксов разных онтологий в единой прикладной онтологической модели является нормальной практикой.

Рис. 2. Диаграмма фрагмента онтологической модели



Определив набор классов, необходимо перейти к созданию свойств, которые могут связывать между собой объекты этих классов. Чтобы легче ориентироваться в свойствах и связях классов, мы рекомендуем аналитикам строить диаграммы фрагментов онтологической модели. Общепринятой нотации для таких диаграмм не существует, но мы выработали, как нам кажется, интуитивно понятный визуальный синтаксис. Пример такой диаграммы приведен на рис. 2. Прямоугольники на нем означают классы модели, стрелки между ними – свойства-связи (`owl:ObjectProperty`), которыми могут быть связаны объекты этих классов. В виде параллелограмма показано свойство-литерал (`owl:DatatypeProperty`), соединенное с классом, объекты которого могут обладать значениями этого свойства. Внутри параллелограмма указан тип данных, к которому относятся значения свойства.

Кроме перечисленных выше классов на этой диаграмме появляется класс «Единица измерения» (`qudt:Unit`) из внешней онтологии QUDT, описывающей единицы измерения физических величин. Свойства (предикаты) также в основном заимствованы из QUDT и SOSA.

Опираясь на диаграмму, создадим в онтологии три свойства-связи и одно свойство-литерал. В результате набор свойств, значениями которых могут обладать объекты класса `sosa:Result`, будет выглядеть в редакторе АрхиГраф.Мир так (область «Атрибуты» на странице свойств класса `sosa:Result`):

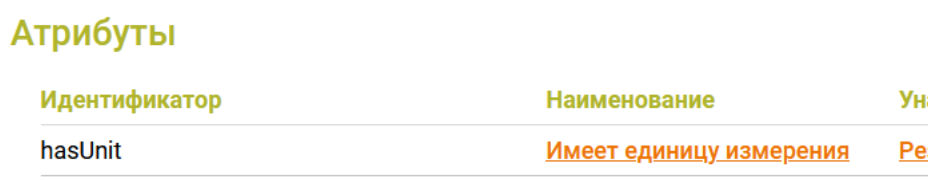


Рис. 3. Список атрибутов, значениями которых могут обладать объекты класса Result

Мы завершили создание TBox – набора классов и определений свойств нашей модели.

Онтологию, построенную в АрхиГраф.Мир, можно выгрузить в файл формата RDF/XML или Turtle для того, чтобы поработать с ней во внешних приложениях – таких, как редактор Protégé. Можно, наоборот, и загружать элементы TBox из файлов в RDF Triple store, в которой АрхиГраф.MDM хранит описание структуры модели. После этого нужно синхронизировать кэш MDM и кэш редактора.

Для очистки кэша MDM нужно выполнить запрос

```
<GetDataSchema
 Endpoint="Demo"
 Originator="[…]"
 ClearCache="1"
/>
```

Для очистки кэша редактора нужно вызвать URL [хост редактора]/agmir/api/cache/rebuild?endpoint=[код точки доступа]).

Следующая задача – создание индивидуальных объектов. Среди них можно условно выделить элементы справочников и перечислений, входящие в состав нормативно-справочной информации. В нашем примере к НСИ относятся единицы измерения и объекты измерений. Можно было бы импортировать из QUDT полный справочник единиц, но мы ограничимся созданием только тех из них, которые потребуются для обработки данных в нашем примере.

Для создания единиц измерения в системе воспользуемся инструментом импорта из Excel. Для этого в редакторе АрхиГраф.Мир нужно перейти на страницу «Импорт/экспорт из Excel», выбрав соответствующий пункт в главном меню. Затем надо выбрать слева в дереве один или несколько классов, объекты которых мы хотим создать в модели путем импорта. В средней части страницы нужно отметить переключатель «Объекты класса» и нажать кнопку «Выгрузить». Будет сформирован Excel-файл – шаблон для заполнения информации об объектах. Работа с файлами Excel подробно описана в документации пользователя АрхиГраф.Мир. Пример файла, заполненного информацией о двух объектах — единицах измерения, выглядит так:

	A	B	C	D	E
1	http://qudt.org/schema/qudt/Unit	Единица измерения			
2	uri	rdfs:label	rdfs:comment	archigraph:archive	type
3	Сущность	Название	Описание	Удаленный	Тип
4	http://qudt.org/vocab/unit/DEG_C	Градус Цельсия		нет	http://qudt.org/schema/qudt/Unit
5	http://qudt.org/vocab/unit/KiloGM	Килограмм на кубический метр		нет	http://qudt.org/schema/qudt/Unit

Рис. 4. Файл Excel, подготовленный для импорта единиц измерения в онтологию

Подготовленный файл нужно сохранить на локальном компьютере, выбрать его в области «Импорт» той же страницы редактора, и нажать кнопку «Отправить». После этого на экране отобразится протокол результатов импорта, как показано на рис. 5.

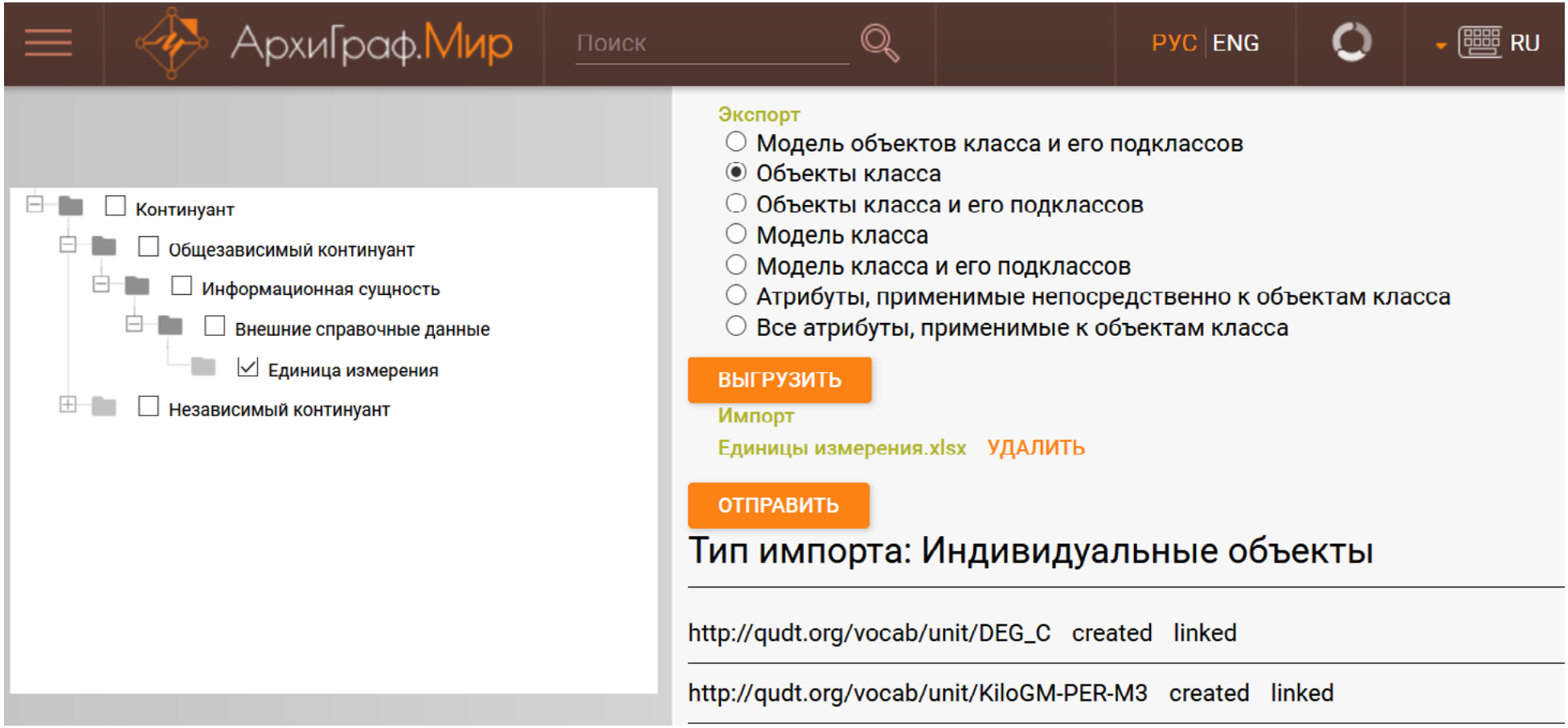


Рис. 5. Результат импорта единиц измерения из Excel

В интерфейсе редактора созданные единицы измерения будут выглядеть так, как показано на рис. 6.

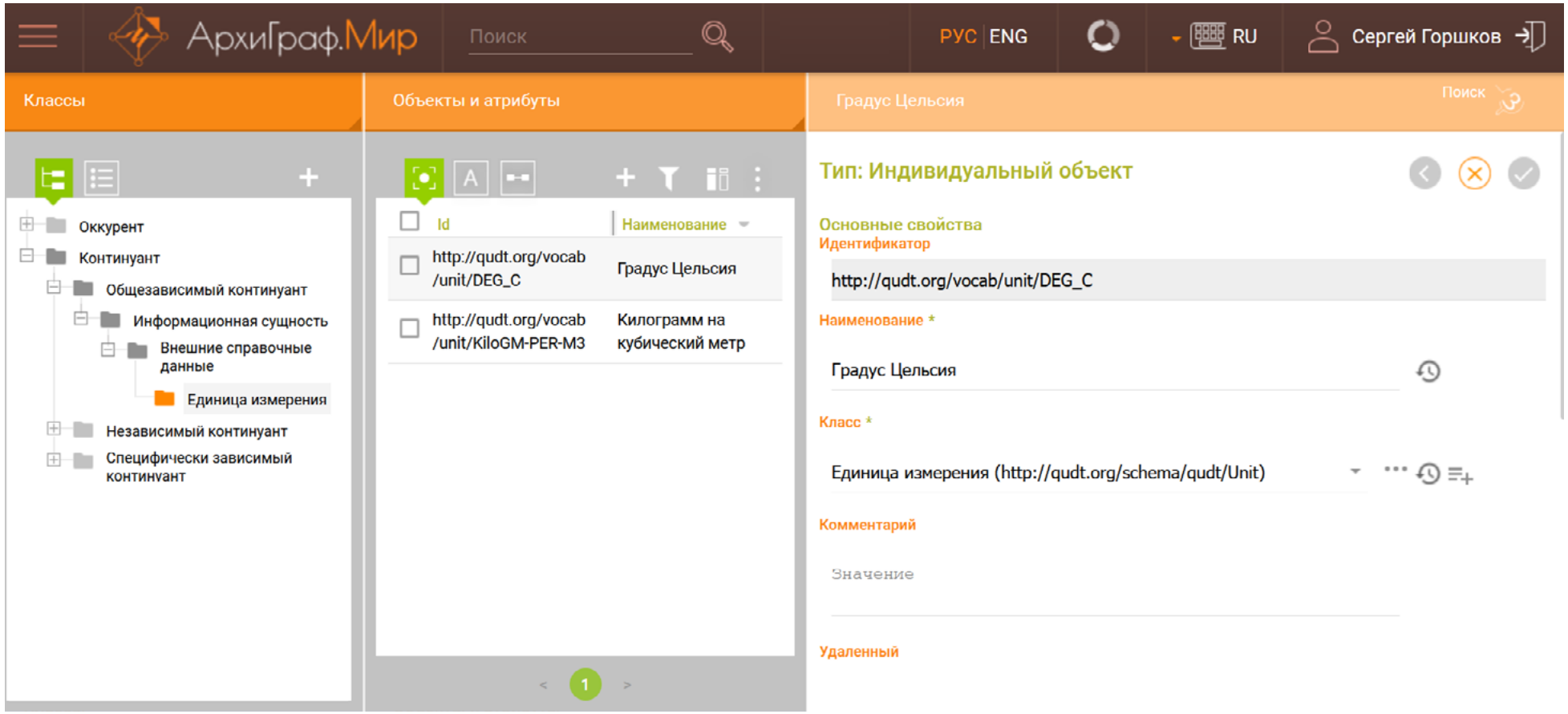


Рис. 6. Результат создания единиц измерения в редакторе АрхиГраф.Мир

Объекты измерения создадим через онтологию вручную. При ручном создании объектов система сгенерирует для них уникальные идентификаторы по предусмотренному в ней шаблону. Список созданных объектов измерения будет выглядеть так:

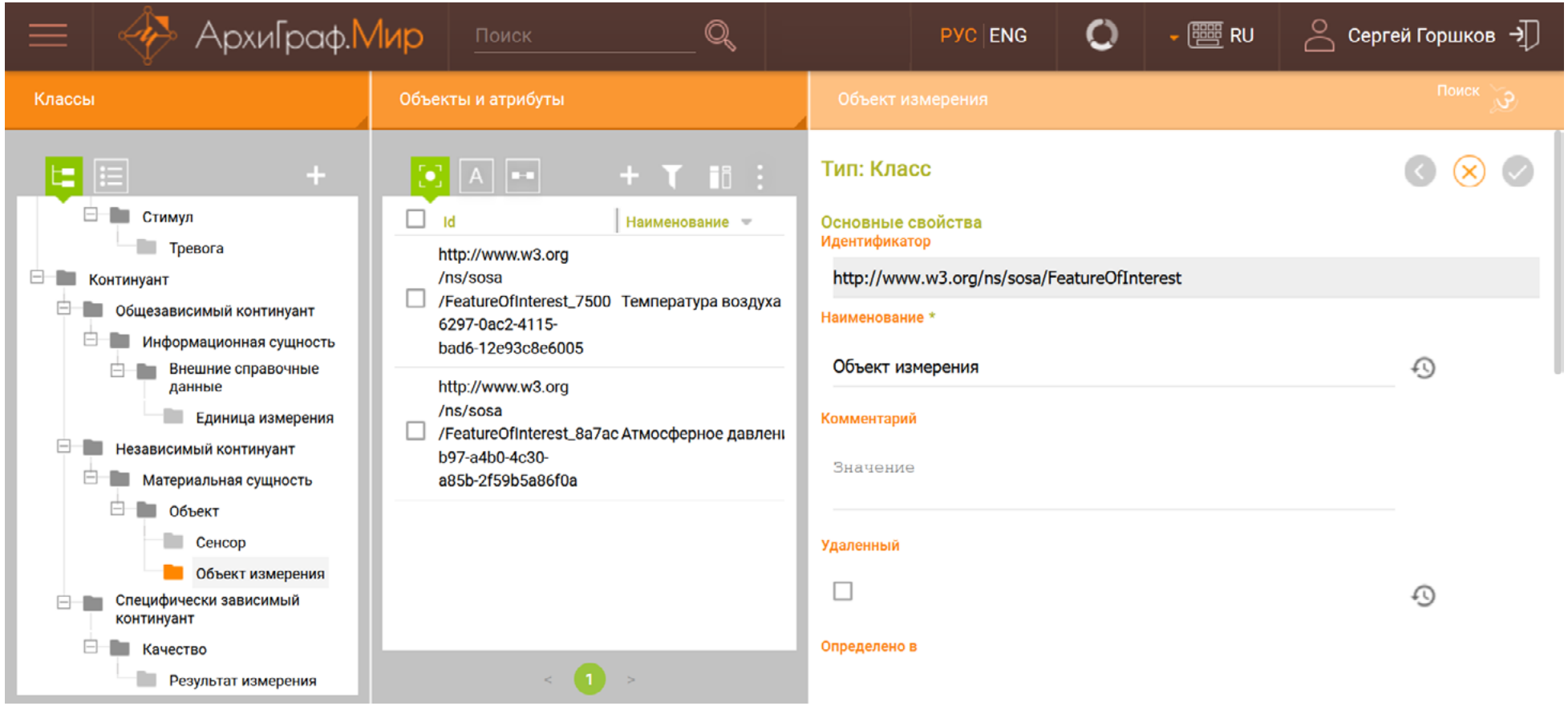


Рис. 7. Список объектов измерения, созданных вручную в редакторе АрхиГраф.Мир

Индивидуальные объекты, входящие в состав НСИ, хотя и относятся формально к ABox – части модели, описывающей конкретные факты о конкретных объектах – могут храниться вместе с TBox в графовой базе данных. Все остальные объекты, состав которых будет динамично меняться по ходу использования системы, а количество значительно – список датчиков, результатов измерения, тревог – лучше разместить в реляционной или документ-ориентированной базе данных. Чтобы разобраться, как сконфигурировать АрхиГраф.MDM для хранения объектов таким образом, необходимо познакомиться с принципами хранения данных в системе.

Хранение данных

Платформа АрхиГраф обеспечивает хранение конкретных объектов (в терминах онтологических спецификаций они называются «индивидуальными объектами») в соответствии со структурой, заданной в онтологии. При этом под управлением платформы может находиться несколько хранилищ разных типов.

«Родным» типом хранилищ для онтологий являются графовые СУБД, поддерживающие доступ к их содержимому по протоколу SPARQL. Существует немало коммерческих реализаций таких СУБД (Stardog, AllegroGraph, Virtuoso и др.), а также проекты с открытым исходным кодом – например, Apache Fuseki. Платформа АрхиГраф использует графовую СУБД (обычно Apache Fuseki) для хранения TBox – части модели, содержащей определения классов и свойств, а также индивидуальные объекты справочников / перечислений.

Хранение большого числа индивидуальных объектов в графовой СУБД неэффективно, потому что такие базы обычно не позволяют строить индексы, учитывающие специфику хранения данных разной природы (временных рядов, событий, связанных между собой каталогов объектов и др.). Fuseki, строго говоря, вообще не предназначена для хранения динамично меняющейся информации. Ключевой функциональной особенностью АрхиГраф является возможность перенести хранение индивидуальных объектов определенных ветвей онтологии в реляционную (обычно Postgres) или документ-ориентированную (например, MongoDB) СУБД. В таких базах данных можно построить индексы, учитывающие специфику структуры данных и наиболее актуальные виды запросов, можно сегментировать таблицы, использовать кластеризацию хранилищ.

Важно, что платформа АрхиГраф обеспечивает доступ по протоколу SPARQL к объектам, расположенным в любых хранилищах, «создавая иллюзию» нахождения всех данных в единой графовой СУБД для приложений-клиентов. Также сохраняется возможность применять к этим объектам правила логического вывода, о которых мы расскажем далее.

В нашем примере сигналы, поступающие от датчиков, а также события представляют собой наборы данных, объем которых растет со временем и может достичь миллионов объектов. Целесообразно перенести хранение таких данных в таблицы реляционной базы Postgres, организовав их индексирование и, возможно, сегментацию. Это значительно ускорит выборку сигналов и событий, в том числе для лент оповещений в приложении, а также обеспечит возможность масштабирования системы в части объема хранимых данных.

Что нужно сделать для реализации такой схемы?

Прежде всего, нужно развернуть и сконфигурировать хранилище Postgres. Создать в нем базу данных, а в ней – таблицу, которая будет хранить объекты выбранных классов. Можно как создать разные таблицы для объектов разных классов, так и использовать

единую таблицу – выбор зависит от того, совпадают ли атрибуты, по которым нужно построить индексы для объектов этих классов. В нашем случае сигналы (`ssn:Stimulus`) и события (`Event`) имеют как пересекающиеся, так и собственные атрибуты; кроме того, из-за существенно большего числа сигналов к ним может быть применена другая политика сегментирования таблицы по датам, в связи с чем более рациональным выглядит решение о создании разных таблиц для объектов этих двух классов.

После того, как таблица создана, в ней нужно создать следующие поля:

- uri – для хранения уникального идентификатора (URI) объекта. Это поле текстового типа character varying(255), которое хранит уникальный идентификатор, присваиваемый платформой АрхиГраф каждому индивидуальному объекту;
- type – поле для хранения принадлежности объекта классам. Поскольку каждый объект может принадлежать нескольким кассам одновременно, поле должно иметь тип «массив строк», character varying(255)[];
- data – поле типа jsonb для хранения значений всех свойств объекта. АрхиГраф записывает значения свойств в коллекцию JSON, которая будет помещена в это поле.

Можно создать также отдельные поля для хранения значений атрибутов, по которым будет проиндексирована и/или сегментирована таблица. В нашем случае это поля, хранящие дату и время поступления сигнала или возникновения события, а также связи сигнала/события с датчиком и помещением, тип события. Далее по этим полям создаются индексы средствами Postgres. Названия всех перечисленных полей могут быть выбраны произвольно.

Далее нужно выполнить настройки АрхиГраф.MDM для того, чтобы она могла работать с созданным хранилищем. Прежде всего нужно зарегистрировать само хранилище – создать запись в конфигурационной базе данных, где будут указаны реквизиты подключения к нему:

```
mdmctl add storage postgresql_jsonb "Хранилище данных
Demo Postgres" --host 127.0.0.1 --login mdm --password
test --dbname mdm_data --table test --history true
```

Все указанные здесь параметры команды нам уже встречались, кроме следующих:

- --dbname mdm_data – таким образом задается имя базы данных, которая используется как хранилище,
- --table test – указывает конкретную таблицу с данными,
- --history true – определяет, что хранилище сохраняет историю изменения свойств объектов.

Затем нужно привязать хранилище к точке доступа, с которой оно будет использоваться:

```
mdmctl bind storage "Хранилище данных Demo Postgres"
Demo
```

Чтобы увидеть список хранилищ, привязанных к точке доступа, выполните команду

```
mdmctl show storages Demo
```

Далее необходимо привязать класс(ы) онтологии к новому хранилищу в рамках определенной точки доступа. Для этого нужно выполнить команду

```
mdmctl bind class http://www.w3.org/sosa/Observation
"Хранилище данных Demo Postgres" Demo
```

Класс задается своим идентификатором: можно использовать краткую форму записи, если его префикс совпадает с префиксом по умолчанию для данной точки доступа, и полную форму (в виде URI) в остальных случаях.

Чтобы увидеть список классов, привязанных к хранилищам, выполните команду

```
mdmctl show class Demo
```

После создания таблицы с данными имена созданных в ней полей должны быть внесены в настройки мэппинга платформы АрхиГраф. Для каждого поля необходимо выполнить команду:

```
mdmctl bind property "http://www.w3.org/1999/02/22-rdf-
syntax-ns#type" test type
```

После bind property в синтаксисе этой команды идут последовательно идентификатор свойства в онтологической модели, название таблицы базы данных и имя соответствующего поля в этой таблице. Для привязки столбца с идентификатором сущности используется следующая команда:

```
mdmctl bind property uri test uri
```

Подобные команды нужно выполнить и для других столбцов, созданных в таблице с данными. Поле data также нужно привязать к псевдо-атрибуту data.

Увидеть мэппинг атрибутов для определенной таблицы позволяет следующая команда:

```
mdmctl show property test
```

Ее параметром является имя таблицы.

После того, как такая настройка выполнена, вы сможете работать с данными классов, привязанных к хранилищам, точно так же, как и раньше – но индивидуальные объекты этих классов будут расположены не в графовой базе данных, а в реляционной. Аналогичным образом можно подключать к АрхиГраф и другие хранилища, например MongoDB.

Какие же настройки нужно сделать в нашем примере? Взглянув на дерево онтологии, мы увидим, что целесообразно разделить объекты как минимум на три коллекции – физических объектов (сенсоров и др.), событий и результатов измерений. По каждой из коллекций можно будет построить индексы по атрибутам, ко-

торые обеспечат быструю выборку данных по наиболее типичным для них наборам условий (например, событий – по дате и времени). В таблице 2 приведены настройки соответствия классов хранилищам, которые мы рекомендуем применить в рассматриваемом примере.

Индексы необходимо создать в Postgres вручную после того, как созданы соответствующие таблицы. Если набор столбцов, по которым целесообразно построить индексы, заранее не известен – можно начать работу с не индексированными таблицами, а затем использовать логи АрхиГраф.MDM для анализа наиболее типичных / долго выполняющихся запросов к данным разных типов, чтобы определить оптимальную стратегию индексирования.

Таблица 2. Распределение классов онтологии по хранилищам

Название класса	Идентификатор класса	Хранилище	Индексы
Материальная сущность	http://purl.obolibrary.org/obo/BFO_0000040	postgresql_jsonb db_objects	
Сигнал	http://www.w3.org/ns/ssn/Stimulus	postgresql_jsonb db_signals	dateTime, sosa:madeBySensor
Событие	http://trinidata.ru/demo/Event	postgresql_jsonb db_events	dateTime, hasEventType
Результат измерения	http://www.w3.org/ns/sosa/Result	postgresql_jsonb db_results	

Свойства классов и свойств

Читатели, хорошо знакомые с концепциями онтологического моделирования, знают о том, что спецификации «семантического веба» RDF, RDFS, OWL определяют ряд стандартных предикатов, таких как `rdf:type`, `rdf:label` или `owl:sameAs`. Такие предикаты применимы не только к индивидуальным объектам, но и к классам или свойствам онтологии (а иногда и только к ним). Это значит, что классы и свойства могут обладать собственными значениями свойств: они могут иметь названия, описания, различные связи между собой. Некоторые из этих предикатов (`rdf:type`, `rdfs:label`, `rdfs:comment`, `rdfs:subClassOf`, `rdfs:domain`, `rdfs:range`, `rdfs:minCardinality`, `rdfs:maxCardinality`) «известны» платформе АрхиГраф по умолчанию.

Но что делать, если нужно воспользоваться предикатом, отсутствующим в этом списке – например, объявить одно свойство инверсным по отношению к другому при помощи предиката `owl:inverseOf`, или объявить два индивидуальных объекта соответствующими одному и тому же означаемому при помощи предиката `owl:sameAs`? Для этого нужно зарегистрировать в АрхиГраф.MDM так называемое «стандартное свойство» (т.е. определенное стандартом, а не явно объявленное в онтологии).

Для добавления стандартного свойства нужно выполнить команду следующего синтаксиса:

```
mdmctl add property owl:sameAs Demo --name "Эквивалентен"@ru --name "Equivalent"@en --domain owl:NamedIndividual --range owl:NamedIndividual --singular true --mandatory false --multilang true --readonly false
```

В этом вызове после команды add property последовательно идут:

- owl:sameAs – URI создаваемого свойства
- Demo – идентификатор точки доступа, в которой оно будет создано
- --name «Эквивалентен»@ru – название свойства на русском языке
- --name «Equivalent»@en – название свойства на английском языке
- --domain owl:NamedIndividual – область применения свойства
- --range owl:NamedIndividual – диапазон значений свойства
- --singular true – равно true, если свойство может принимать единственное значение для каждого объекта

- --mandatory false – равно true, если свойство должно обязательно иметь хотя бы одно значение для каждого объекта
- --multilang true – равно true, если значения свойства могут быть заданы на разных языках
- --readonly false – равно true, если значение свойства доступно только для просмотра и не может быть изменено.

В этом и других примерах команд все параметры, не предваряемые ключом --[название параметра], являются обязательными, а предваряемые ключом – могут быть пропущены.

Для просмотра стандартных свойств, объявленных в точке доступа Demo, используйте следующую команду:

```
mdmctl show field Demo
```

После того, как «стандартные свойства» зарегистрированы в АрхиГраф.MDM, они станут автоматически отображаться в формах редактирования свойств элементов онтологии в АрхиГраф.Мир. Однако их объявление не означает, что MDM будет автоматически осуществлять логический вывод (entailment), предусмотренный стандартами: например, если свойство A объявлено под-свойством (`subPropertyOf`) свойства B, и объект X обладает значением свойства A, MDM не сделает ожидаемый вывод о том, что X обладает и значением свойства B. Подобные выводы делаются только для свойств, перечисленных в списке, приведенном в начале этого раздела – `rdfs:subClassOf`, `rdfs:domain`, `rdfs:range`. Правила entailment'а для добавленных пользователем стандартных свойств можно создать вручную при помощи АрхиГраф.СУЗ, о которых мы расскажем далее.

Права доступа и безопасность

Большая часть операций с данными, представленными в соответствии с онтологией, будет выполняться не через редактор АрхиГраф.Мир, а при помощи программного интерфейса АрхиГраф.MDM. Наша платформа предоставляет несколько видов программных интерфейсов, среди которых мы рассмотрим REST, SPARQL и обмен сообщениями через очереди[1].

Отметим, что в АрхиГраф.MDM существует две версии ядра, осуществляющего ответы на запросы. Основное ядро реализовано на языке PHP и разворачивается в контейнерах в Kubernetes так, как это описано в главе «Развертывание». Дополнительное ядро в настоящее время находится в активной разработке, но уже доступно для использования (поставляется как дополнительный компонент платформы). Этот вариант ядра реализован на C++ и представляет собой Linux-приложение, реализующее функции веб-сервера и обработчика очередей Kafka и RabbitMQ точно таким же образом, каким их обрабатывает основной вариант ядра. Дополнительное ядро имеет производительность, в 2-3 раза превышающую производительность основного ядра, но обеспечивает в настоящее время только обработку запросов на чтение данных. Дополнительное ядро полностью функционально в части основных функций MDM и поддержки многоязычности значений, но пока не поддерживает работу с геоданными. Также оно поддерживает лишь ограниченный набор хранилищ: Fuseki и PostgreSQL (варианты с обычными таблицами и с хранением объектов в полях типа jsonb).

Для использования API АрхиГраф.MDM необходимо зарегистрировать в ней «отправителей запросов», или Originator'ов – идентификаторы клиентских программных компонентов, которые будут вызывать ее сервисы. Каждый компонент-клиент должен указывать в каждом запросе к MDM какой-либо Originator для того, чтобы MDM могла проконтролировать права доступа компонента к данным. В доверенных средах обращения к API могут осуществляться без авторизации, непосредственно по переданному идентификатору, а в средах с повышенными требованиями к безопасности можно дополнительно использовать механизм JWT (JSON Web Tokens) для авторизации программных компонентов.

Контроль безопасности в инфраструктуре, построенной вокруг платформы АрхиГраф, основан на использовании продукта KeyCloak. Этот свободно распространяемый продукт имеет широкие возможности интеграции с провайдерами авторизации, такими как Active Directory, а также может сам выступать в качестве инструмента создания учетных записей и ролей пользователей и программных компонентов. KeyCloak позволяет также настроить прозрачную авторизацию Kerberos для того, чтобы пользователь не вводил логин и пароль при входе в систему; он же обеспечивает механизм SSO, благодаря которому пользователь, авторизованный при входе в одно из приложений АрхиГраф, оказывается авторизованным и в других приложениях.

1 Также с MDM можно работать через GraphQL и по протоколу websocket

KeyCloak используется и для авторизации пользователей в интерактивном режиме, и для авторизации программных компонентов при работе с API АрхиГраф.MDM. В защищенной среде программный компонент, желающий использовать API MDM, должен сначала обратиться к KeyCloak, пройти авторизацию и получить токен, который затем использовать при обращении к API. MDM проверит валидность токена и в зависимости от результатов выполнит запрос или откажет в его выполнении.

Важно отметить, что АрхиГраф.MDM не работает с учетными записями пользователей и «ничего не знает» о них. Клиентами MDM являются программные компоненты. Работа же с пользователями происходит на уровне приложений, предоставляющих графический пользовательский интерфейс, таких как АрхиГраф.Мир. Наш редактор онтологии сам является клиентом MDM, поскольку использует ее API для выполнения всех операций с онтологией. Значит, при обращении к API MDM он должен передать ей определенный код программного компонента. Как же определяется, от имени какого компонента работает с MDM АрхиГраф.Мир?

Ответ на этот вопрос очень важен для понимания концепции безопасности и разделения прав доступа на платформе АрхиГраф. Мы рекомендуем для каждой роли (группы) пользователей создавать отдельный виртуальный «программный компонент», или Originator в АрхиГраф.MDM. Для таких компонентов, как мы покажем далее, можно настроить права доступа к структуре и данным онтологической модели. Чтобы этого достичь, нужно сопоставить группы пользователей KeyCloak и Originator'ы MDM с помощью файла perm.json, как это описано в главе «Развертывание».

Мы рекомендуем использовать ту же парадигму во всех приложениях, предоставляющих пользователям визуальный интерфейс для работы с онтологией под управлением АрхиГраф.MDM. Для программных компонентов, выполняющих операции трансформации данных без участия пользователей (импорт/экспорт и другая обработка) можно использовать единственный Originator для каждого компонента. Мы не рекомендуем использовать один и тот же Originator для нескольких компонентов, поскольку это, во-первых, затруднит анализ логов, где Originator фиксируется для каждого запроса, во-вторых, приведет к невозможности получать одним компонентом по подписке уведомления об изменениях модели и данных, сделанных другим компонентом. Работу механизма подписок мы рассмотрим далее.

Рассмотрим команды, с помощью которых осуществляется управление списком отправителей запросов (Originator'ов) MDM. Для добавления нового отправителя нужно выполнить следующую команду:

```
mdmctl add system "Демо-клиент" demo --token democlient
```

Здесь «Демо-клиент» – читаемое название отправителя, а demo – код Originator'а, который нужно будет передавать при каждом обращении к API MDM.

После создания отправителя нужно предоставить ему права работы с одной или несколькими точками доступа:

```
mdmctl bind system demo Demo
```

В синтаксисе этой команды после команды bind system следует код Originator'a (demo), а затем – код точки доступа (Demo).

По умолчанию Originator получает доступ ко всем операциям со всеми элементами структуры и данных точки доступа. Однако этот доступ можно ограничить на уровне конкретных классов, выполнив следующую команду:

```
mdmctl allow demo http://www.w3.org/ns/sosa/Observation
1 0 0
```

Здесь после команды allow последовательно указаны код Originator'a demo, идентификатор класса, к которому предоставляется доступ, и три флага, соответствующие операциям с объектами этого класса – чтение, запись, запись с подтверждением. В приведенном примере Originator'у demo разрешено чтение объектов класса `sosa:Observation` и запрещена их запись. Права, назначенные на надкласс, наследуются всеми его подклассами; при наследовании прав от нескольких надклассов выбирается наиболее строгий вариант. Чтобы установить права на работу с классами и свойствами модели, нужно в качестве идентификатора класса в команде указать `owl:Class`, `owl:ObjectProperty` или `owl:DatatypeProperty`.

Уровень доступа «запись с подтверждением» означает, что от имени данного Originator'a можно делать запросы на запись (создание, изменение, удаление) объектов, однако такие запросы не будут автоматически немедленно выполнены, а будут помещены в специальную внутреннюю очередь MDM. При помощи визуального административного интерфейса MDM пользователь с правами администратора может просмотреть очередь запросов на изменения объектов, и принять решение по каждому из них – выполнить или отклонить запрос.

Чтобы просмотреть список систем и назначенных им прав доступа, выполните команду:

```
mdmctl show system
```

В нашем примере необходимо создать Originator'ов для программных компонентов, которые будут работать с MDM – адаптеров приема данных от датчиков и для визуального интерфейса ситуационного центра. Если с интерфейсом будут работать пользователи разных групп – нужно создать отдельного Originator'a для каждой группы, и распределить права доступа этих Originator'ов к объектам разных классов в соответствии с функциональными задачами пользователей этих групп.

Архитектура приложений на платформе АрхиГраф

Теперь, когда мы вооружены знаниями об организации хранения данных в АрхиГраф.MDM и выполнили все необходимые настройки, можно перейти к созданию сценария обработки поступающих в ситуационный центр данных с помощью АрхиГраф.MDM. Платформа поддерживает несколько способов взаимодействия с ней со стороны пользователей и программных компонентов. Эти способы схематически показаны на рис. 8.

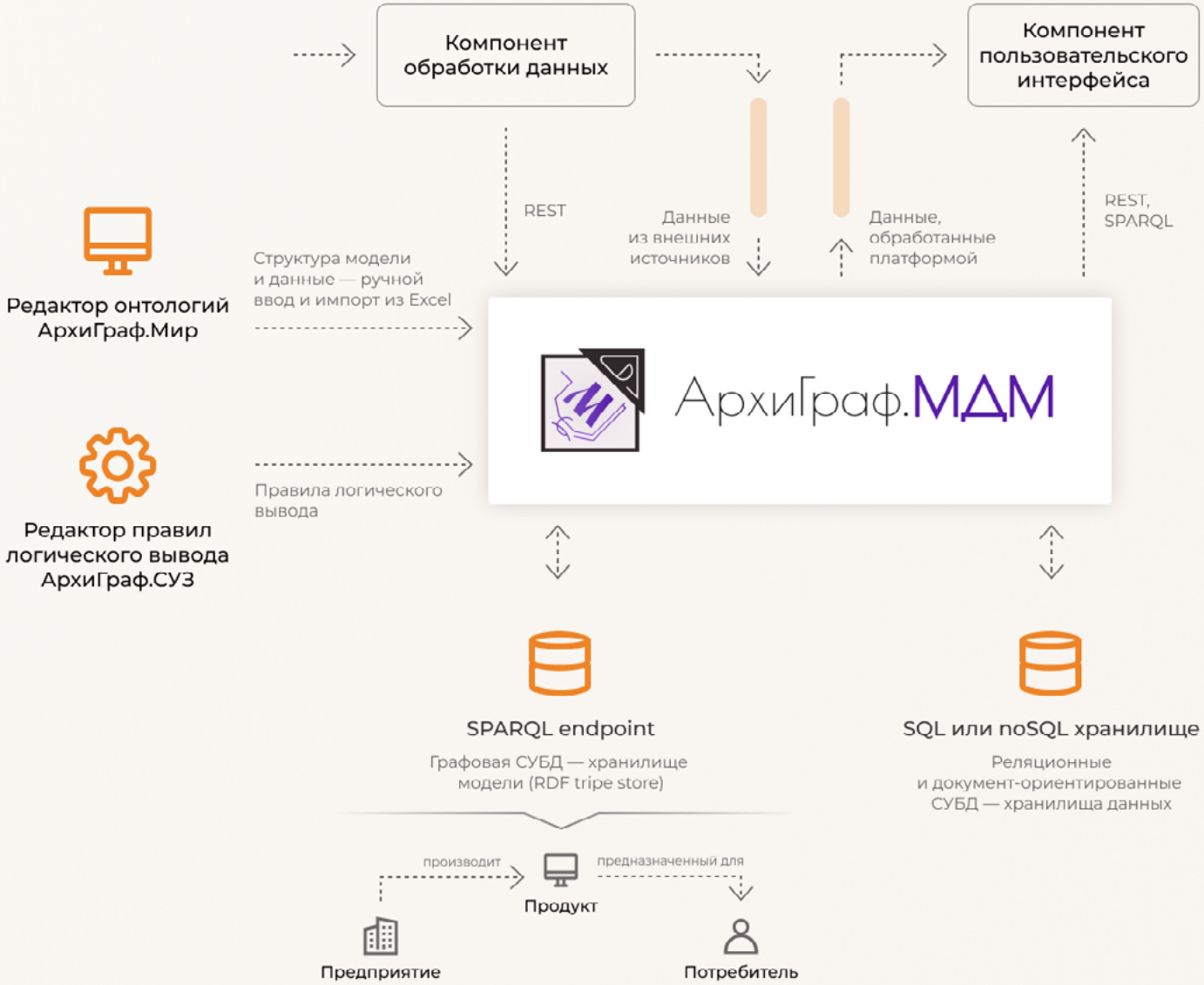
Стрелки на этой диаграмме показывают направление передачи данных, а не направление вызова программных интерфейсов.

Как видно из диаграммы, вся работа с данными происходит через компонент платформы АрхиГраф.MDM. Прикладные компоненты автоматизированной системы условно обозначены как «Компонент обработки данных» и «Компонент пользовательского интерфейса» (в реальных системах может быть множество компонентов и подобного, и иного назначения). Платформа предоставляет программным компонентам несколько способов получения и записи данных:

- В синхронном режиме – через REST-интерфейс или SPARQL точку доступа;
- В асинхронном режиме через очереди обмена сообщениями (запросы инициируются прикладными компонентами);
- В асинхронном режиме по подписке через очереди обмена сообщениями (запросы инициируются MDM).

Последний способ целесообразно применять в случае, когда приложение должно отображать изменения, происходящие в данных – например, показывать пользователю ленту событий. Работа в асинхронном режиме по подписке позволяет избежать постоянного опроса MDM для получения изменений. Далее мы рассмотрим сценарии использования всех способов обмена с MDM, но прежде необходимо рассказать о принципах развертывания платформы.

Рис. 8. Архитектура компонентов автоматизированной системы на платформе АрхиГраф



Развертывание платформы АрхиГраф

Для развертывания нужны перечисленные ниже программные компоненты, без которых функционирование платформы невозможно:

- PostgreSQL версии >= 9.6
- Apache Jena Fuseki версии = 3.12
- Keycloak версии >= 10.0.1
- RabbitMQ версии >=3.6.6 или Apache Kafka версии >= 2.4.0

Для кеширования данных онтологической модели потребуется также:

- Redis версии >=5.0.3

Для сбора и анализа файлов журналов приложений (логов) необходимы:

- ELK Stack[1] - Elasticsearch, Logstash (и Fluentd), Kibana
- Timber.io Vector

Для сбора метрик с приложений необходимы:

- Prometheus
- php-fpm-exporter

Установка и настройка всех перечисленных компонентов выходит за рамки данного руководства. Предполагаем, что установлены и развернуты базовые компоненты, необходимые для работы (внутри периметра Kubernetes или за его границами):

- PostgreSQL – обязательно в stateful варианте
- Fuseki – как stand-alone или как сервис Tomcat
- Keycloak – с сохранением данных в PostgreSQL
- RabbitMQ или Apache Kafka – любой из брокеров, обязательно в stateful варианте
- Redis – кластер (предпочтительнее) или stand-alone, можно без сохранения состояния, т.к. кэш всегда можно сгенерировать с нуля

1 https://www.digitalocean.com/community/tutorials/how-to-set-up-an-elastics-earch-fluentd-and-kibana-efk-logging-stack-on-kubernetes-ru#шаг-4-%E2%80%94-создание-набора-демонов-fluentd

Базы данных приложений

Компоненты платформы АрхиГраф используют несколько баз данных (далее – БД), без которых функционирование невозможно. Каждый компонент использует свою БД для хранения настроек и другой мета-информации. Для начала работы нужно развернуть следующие БД:

- db_mdm – здесь МДМ хранит все настройки и т.д.
- db_agmir – АрхиГраф.Мир хранит в ней информацию о сессиях
- db_suz – логические правила и настройки АрхиГраф.СУЗ

Создайте на сервере PostgreSQL отдельных пользователей (role):

```
CREATE ROLE mdm with LOGIN ENCRYPTED PASSWORD ‘mdm’;
CREATE ROLE agmir with LOGIN ENCRYPTED PASSWORD
‘agmir’;
CREATE ROLE suz with LOGIN ENCRYPTED PASSWORD ‘suz’;
```

Создайте на сервере PostgreSQL отдельные БД с указанием владельцев:

```
CREATE DATABASE db_mdm with owner mdm;
CREATE DATABASE db_agmir with owner agmir;
CREATE DATABASE db_suz with owner suz;
```

Разверните исходные дампы баз в соответствующие целевые БД:

```
psql -h [хост Postgres] -U mdm -d db_mdm < /path/to/
mdm_dump.psql
```

```
psql -h [хост Postgres] -U suz -d db_suz < /path/to/
suz_dump.psql
```

Обратите внимание, что БД для АрхиГраф.Мир наполняется автоматически при развертывании приложения и не требует развертывания из дампа.

Датасет Fuseki

Создайте датасет Fuseki, используя его веб-панель управления, для этого перейдите на вкладку Manage datasets и выберите Add new dataset:

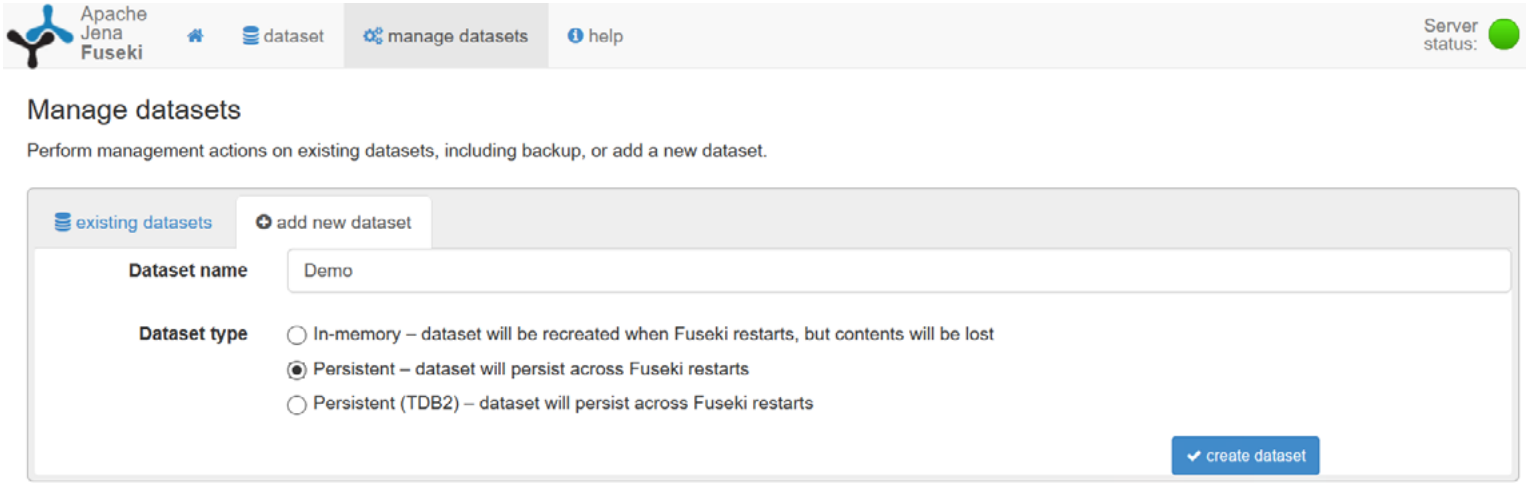


Рис. 9. Создание набора данных в Fuseki

Введите имя датасета, выберите тип Persistent и нажмите кнопку Create dataset.

Дождитесь создания датасета, затем нажмите кнопку Upload data:

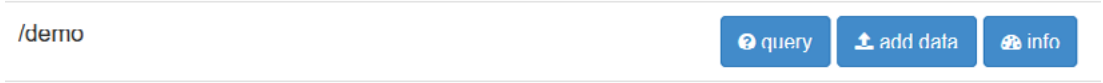


Рис. 10. Кнопка загрузки данных в списке наборов данных в Fuseki

В открывшемся окне нажмите кнопку Select files, выберите на компьютере исходный дамп датасета (в принципе, можно начать работу и с пустым набором данных) и нажмите кнопку Upload now:

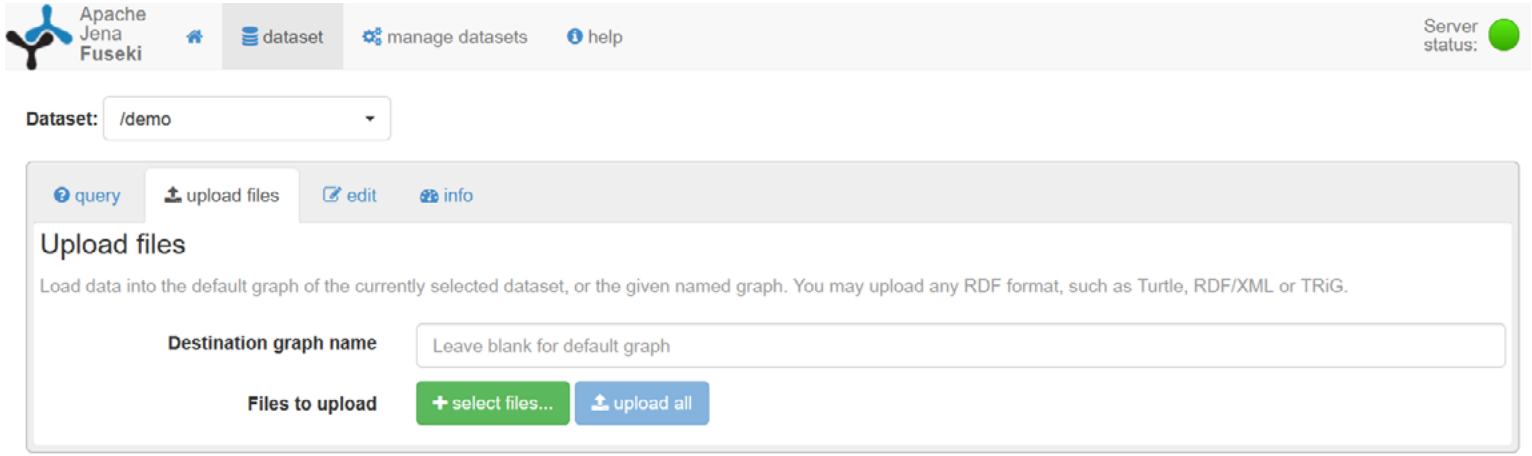


Рис. 11. Форма загрузки онтологии из файла в Fuseki

Дождитесь загрузки и развертывания датасета.

Развертывание АрхиГраф.MDM в Kubernetes

Для развертывания АрхиГраф.MDM вам потребуется доступ к консоли сервера мастер-узла Kubernetes и набор файлов развертывания (manifests) проекта/

По умолчанию все компоненты платформы АрхиГраф развертываются в пространстве с именем «trinidata», которое необходимо создать до развертывания. За дополнительной информацией обращайтесь к официальной документации на Kubernetes[1].

Загрузите файлы развертывания из репозитория и внесите исправления в файл mdm-config.ext, укажите имя или адрес сервера, имя пользователя и пароль, а также имя БД:

```
<?php
// Параметры подключения к базе данных
$host="postgres_host";
$uname="mdm";
$pass="mdm";
$dbname="db_mdm";
$dbserver="pgsql";
```

Внесите исправления в файл base/deploy.yml, указав корректные ссылки на образы (предполагается, что они уже загружены в локальные репозитории):

```
…
  - name: php
    image: &fpm gitlab.trinidata.ru:5005/php/mdm:master
…
```

Внесите исправления в файл mdm/kustomization.yml, указав корректные ссылки на образы (предполагается, что они уже загружены в локальные репозитории):

```
…
images:
  - name: gitlab.trinidata.ru:5005/php/mdm:master
    newTag: mdm
…
```

Внесите исправления в файл mdm/worker.yml, указав корректные ссылки на образы (предполагается, что они уже загружены в локальные репозитории):

```
…
  - name: php-worker
    image: &fpm gitlab.trinidata.ru:5005/php/mdm:master
…
```

Внесите исправления в файл mdm/docker-config.json, указав корректную ссылку на репозиторий Docker и реквизиты для авторизации в формате user:password, закодированные при помощи функции base64:

1 https://kubernetes.io/ru/docs/concepts/overview/working-with-objects/namespaces/

```
{
  "auths": {
    "gitlab.trinidata.ru:5005": {
      "auth": "base64encoded"[1]
    }
  },
  "HttpHeaders": {
    "User-Agent": "Docker-Client/19.03.3 (linux)"
  }
}
```

Внесите исправления в файл mdm/vector.toml, указав корректную ссылку на сервер Elasticsearch для отправки файлов логов на индексацию:

```
...
[sinks.mdm-es]
  inputs = ["mdm-js"]
  type = "elasticsearch"
  host = "http://10.0.0.1:9200"[2]
  index = "mdm_parser-%Y-%m-%d"
...
```

Сгенерируйте финальный файла развертывания, выполнив команду из каталога, в который были загружены манифесты:

```
kubectl kustomize mdm > ready.yml
```

1 Указать base64encode(имя:пароль)

2 Указать адрес или имя сервера Elasticsearch

Разверните АрхиГраф.MDM, выполнив команду из каталога с полученным файлом:

```
kubectl apply -f ready.yml
```

Дождитесь загрузки образов и развертывания контейнеров, это может занять некоторое время. Контролируйте процесс, отслеживая запуск сущностей при помощи команды:

```
kubectl -n trinidata get pod
```

При успешном развертывании должны быть запущены 6 контейнеров, входящие в состав двух подов:

```
NAME                    READY   STATUS    RESTARTS   AGE
mdm-dev-7bf577cc7c      4/4     Running   0          1m
46hbn
mdm-exec-req-dev-       2/2     Running   0          1m
5c4d975c84-t4w2n
```

Под с именем «mdm-dev» может запускаться в нескольких экземплярах для масштабирования и балансировки нагрузки путем равномерного распределения запросов к MDM среди всех копий. Для масштабирования и запуска 4 экземпляров выполните команду:

```
kubectl -n trinidata scale --replicas=4 deployment mdm-dev
```

Контролируйте процесс, отслеживая запуск сущностей при помощи команды:

```
kubectl -n trinidata get pod
```

Под с именем «mdm-exec-req» предназначен для обработки очередей синхронизации и ни в коем случае не должен запускаться более чем в одном экземпляре. Если синхронизация между площадками с MDM не используется, можно временно отключить указанный под и входящие в него контейнеры, выполнив масштабирование к нулю экземпляров:

```
kubectl -n trinidata scale --replicas=0 deployment mdm-exec-req
```

Для маршрутизации входящего трафика в рамках платформы Kubernetes могут быть использованы различные варианты.[1]

В данном развертывании предполагается, что входящий трафик маршрутизируется через прокси-сервер, размещенный на мастер-узле, при помощи встроенного в Kubernetes механизма DNS. Если мы развернули MDM по приведенной выше схеме с использованием пространства имен «trinidata», то его целевым адресом станет:

```
mdm-dev.trinidata.svc.cluster.local:80
```

После развертывания необходимо обратиться к конфигурации MDM и настроить ее на работу с источником данных для онтологической модели (см. главу «Структура данных»)

1 https://kubernetes.io/docs/concepts/services-networking/

Развертывание АрхиГраф.СУЗ

Для развертывания АрхиГраф.СУЗ вам потребуется доступ к консоли сервера мастер-узла Kubernetes и набор файлов развертывания (manifests) проекта.

По умолчанию все компоненты платформы АрхиГраф разворачиваются в пространстве с именем «trinidata», которое необходимо создать до развертывания.

Загрузите файлы развертывания из репозитория и внесите исправления в файл suz-config.ext, укажите имя или адрес сервера, имя пользователя и пароль, а также имя БД:

```
<?php
// Параметры подключения к базе данных
$host="postgres_host";
$uname="suz";
$pass="suz";
$dbname="db_suz";
$dbserver="pgsql";
```

Внесите исправления в файл base/deploy.yml, указав корректные ссылки на образы (предполагается, что они уже загружены в локальные репозитории):

```
...
  - name: php
    image: &fpm gitlab.trinidata.ru:5005/php/suz:mdm
...
```

Внесите исправления в файл mdm/docker-config.json, указав корректную ссылку на репозиторий Docker и реквизиты для авторизации в формате user:password, закодированные при помощи функции base64:

```
{
  "auths": {
    "gitlab.trinidata.ru:5005": {
      "auth": "base64encoded"[1]
    }
  },
  "HttpHeaders": {
    "User-Agent": "Docker-Client/19.03.3 (linux)"
  }
}
```

Сгенерируйте финальный файла развертывания, выполнив команду из каталога, в который были загружены манифесты:

```
kubectl kustomize mdm > ready.yml
```

Разверните АрхиГраф.СУЗ, выполнив команду из каталога с полученным файлом:

```
kubectl apply -f ready.yml
```

Дождитесь загрузки образов и развертывания контейнеров, это может занять некоторое время. Контролируйте процесс, отслеживая запуск сущностей при помощи команды:

```
kubectl -n trinidata get pod
```

При успешном развертывании должны быть запущены 2 контейнера, входящие в состав одного пода:

```
NAME                  READY   STATUS    RESTARTS   AGE
suz-dev-85769bc5fb    2/2     Running   0          1m
-s9qp7
```

Если мы развернули СУЗ по приведенной выше схеме с использованием пространства имен «trinidata», то его целевым адресом станет:

```
suz-dev.trinidata.svc.cluster.local:80
```

Обратите внимание, что для корректной работы СУЗ в рамках описанного развертывания, его контейнер должен получать путь /suz/ при маршрутизации трафика.

Развертывание АрхиГраф.Мир

Для развертывания редактора АрхиГраф.Мир вам потребуется доступ к консоли сервера мастер-узла Kubernetes и набор файлов развертывания (manifests) проекта.

По умолчанию, все компоненты платформы АрхиГраф разворачиваются в пространстве с именем «trinidata», которое необходимо создать до развертывания.

1 Указать base64encode(имя:пароль)

Загрузите файлы развертывания из репозитория и внесите исправления в файл mdm/perm.json, укажите в нем группы пользователей из Keycloak, соответствующие им originator'ы из MDM и их токены для авторизации, например:

```
[
    {
        "group": "/OntoAdminGroup",
        "originator": "admin",
        "token": "Ad19miN7"
    },
    {
        "group": "/OntoModelerGroup",
        "originator": "editor",
        "token": "e4D7iTor8"
    }
]
```

Внесите исправления в файл mdm/env-agmir-init.env, укажите в нем реквизиты для подключения к БД:

```
DATABASE_URL=postgres://agmir:agmir@10.0.0.1:5432/db_
agmir
```

Также отредактируйте файл mdm/env-agmir.env, смотрите комментарии в файле:

```
NODE_ENV=development
# Количество обработчиков
NUM_WORKERS=2
# Включить режим отладки
DEBUG=true
# Временная зона по IANA, должна соответствовать зоне MDM
TZ=Europe/Moscow
# Адрес по которому будет выполняться обращение к API
(backend)
# менять путь /agmir/api можно только при построении
образа
BASEURL=http://host_address/agmir/api
# Метод авторизации, credentials или sso (Keycloak)
AUTH_METHOD=sso
# Адрес для редиректа после успешной аутентификации
# менять путь /agmir/ можно только при построении
образа
AUTH_SSO_SUCCESS_REDIRECT=http://host_address/agmir/
# Если SSL-сертификат Keycloak не проходит проверку,
установить в 0
NODE_TLS_REJECT_UNAUTHORIZED=0
# Адрес сервера Keycloak
KEYCLOAK_URL=http://keycloak_host
# Область Keycloak
KEYCLOAK_REALM=master
# Идентификатор клиента в Keycloak
KEYCLOAK_CLIENT_ID=onto
# Токен клиента в Keycloak
KEYCLOAK_CLIENT_SECRET=49dbd53d-4aa1-483f-a95a-
813e6959a013
# Контекст в Keycloak
KEYCLOAK_CONTEXT=auth
# Порт, на котором будет запущен backend редактора
PORT=3000
# Адрес на котором доступен REST-интерфейс АрхиГраф.MDM
MDM_URL=http://mdm-dev.trinidata.svc.cluster.local/
rest/index.php
# Пользователь, от имени которого выполняются запросы
```

```
по умолчанию
MDM_DEFAULT_USER=test
# Тип брокера, для получения информации об измененных
объектах модели
# может принимать значение ApacheKafka или RabbitMQ
SUBSCRIPTIONS_BROKER=ApacheKafka
# Адрес и порт брокера Kafka
KAFKA_URL=kafka://kafka1.kafka.svc.cluster.local:9092
# Группа брокера Kafka
KAFKA_GROUP=default
# Если в качестве брокера выбран RabbitMQ, то реквизиты
указать так:
# RABBITMQ_URL=rabbitmq://{USER}:{PASSWORD}@
{HOST}:{PORT}
# Реквизиты для подключения к БД
DATABASE_URL=postgres://agmir:agmir@10.0.0.1:5432/db_
agmir
```

Внесите исправления в файл base2/deployment.yml, указав корректные ссылки на образы (предполагается, что они уже загружены в локальные репозитории):

```
…
  - name: migrations
    image: gitlab.trinidata.ru:5005/nodejs/agmir_new/
migrations:master
…
  - name: front
    image: gitlab.trinidata.ru:5005/nodejs/agmir_new/
front:master
…
  - name: back
    image: gitlab.trinidata.ru:5005/nodejs/agmir_new/
back:master
…
```

Внесите исправления в файл mdm/docker-config.json, указав корректную ссылку на репозиторий Docker и реквизиты для авторизации в формате user:password, закодированные при помощи функции base64:

```
{
  "auths": {
    "gitlab.trinidata.ru:5005": {
      "auth": "base64encoded"[1]
    }
  },
  "HttpHeaders": {
    "User-Agent": "Docker-Client/19.03.3 (linux)"
  }
}
```

Сгенерируйте финальный файла развертывания, выполнив команду из каталога, в который были загружены манифесты:

```
kubectl kustomize mdm > ready.yml
```

Разверните АрхиГраф.Мир, выполнив команду из каталога с полученным файлом:

```
kubectl apply -f ready.yml
```

1 Указать base64encode(имя:пароль)

Дождитесь загрузки образов и развертывания контейнеров, это может занять некоторое время. Контролируйте процесс, отслеживая запуск сущностей при помощи команды:

```
kubectl -n trinidata get pod
```

При успешном развертывании должны быть запущены два контейнера, входящие в состав одного пода:

```
NAME                      READY   STATUS    RESTARTS   AGE
agmir-74f7989c6d-lhxs2    2/2     Running   0          1m
```

Если мы развернули Мир по приведенной выше схеме с использованием пространства имен «trinidata», то его целевым адресом станет:

```
agmir.trinidata.svc.cluster.local:80
```

Обратите внимание, что образы редактора АрхиГраф.Мир собраны так, чтобы он открывался по адресу /agmir/. В рамках описанного развертывания для корректной работы требуется передавать путь /agmir/ в контейнер редактора при машрутизации трафика.

Подключение кэширования в Redis для АрхиГраф.MDM

Кэширование объектов модели используется для ускорения работы и сокращения запросов к серверу Fuseki. Для включения кэширования необходимо изменить настроечные константы в АрхиГраф.MDM, обратившись к утилите mdmctl. Ее вызов необходимо выполнять из консоли сервера узла-мастера Kubenetes, используя команду для управления кластером, например:

```
kubectl -n trinidata exec -it mdm-dev-7bf577cc7c-46hbn
-c php -- /
mdmctl <...>
```

Символ прямого слеша «/» в данном случае означает перенос части команды на следующую строку. Далее мы будем опускать начало команды, включая символ слеша, предполагая, что вы уже знаете, как выполнить команду внутри контейнера с MDM.

Установите значение константы, определяющей параметры кэша; укажите имя сервера, порт, номер пространства ключей (selector), пароль (если есть) и режим кластера, если используется кластер Redis:

```
mdmctl set constant use_cache_params /
{"server":"redis","port":"6379","selector":1,"password"
:"","cluster":0}
```

Обратите внимание, что в режиме кластера номер пространства ключей не имеет значения.

Установите значение константы, определяющей тип кеша в «redis»:

```
mdmctl set constant use_cache_mode redis
```

Установите значение константы, отвечающей за включение кэши-

рования в хранилище с моделью в «1»:

```
mdmctl set constant use_cache_model_storage 1
```

Для проверки выполните в панели АрхиГраф.MDM запрос на получение модели:

```
<?xml version="1.0" encoding="UTF-8"?>
<GetDataSchema
  Endpoint="Demo"
  Originator="test"
  WithoutInherited="0"
  WithoutRangeInherited="1"
  WithoutAttributes="0"
/>
```

При корректной работе механизма кэширования в возвращаемом ответе должен присутствовать флаг CacheUsed="1", указывающий на использование кэша.

После развертывания всех компонентов необходимо настроить сбор логов в Kibana.

Работа с REST API АрхиГраф.MDM

В нашем примере нужно решить как минимум две задачи обработки поступающих данных: создать и актуализировать в MDM список датчиков, которые могут генерировать тревоги, и организовать обработку в реальном времени поступающих от них тревог. Предположим, что те и другие данные мы можем получить от автоматизированных систем пожарной и охранной сигнализации: список датчиков – в виде текстового файла, а тревоги – в виде JSON-объектов, передаваемых на REST-сервис, предоставляемый со стороны наших компонентов. Нам нужно реализовать два компонента: один будет вызываться по расписанию и обрабатывать список датчиков, другой – вызываться HTTP POST-запросом со стороны автоматизированных систем сигнализации.

Скрипт, импортирующий список датчиков, должен реализовывать следующий алгоритм для каждой строки обрабатываемого файла:

- Определить идентификатор датчика;
- Выполнить запрос к MDM, чтобы определить, зарегистрирован ли уже в ней датчик с таким идентификатором;
- Если датчик не зарегистрирован – сформировать запрос на его создание, преобразовав значения всех свойств датчика из исходного файла в значения соответствующих свойств объекта онтологии;
- Если датчик уже существует в онтологии – сформировать запрос на изменение значений его свойств.

Если какой-либо датчик из числа присутствующих в онтологии отсутствует в обрабатываемом файле – установить ему пометку на удаление, присвоив значение true специальному атрибуту `archigraph:archive`.

Для реализации такого алгоритма адаптеру потребуется выполнить следующие запросы[1].

1) Запрос на получение датчика с определенным значением идентификатора во внешней системе. В синтаксисе XML этот запрос имеет такой вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<GetObjectsGroup Endpoint="Demo" Originator="test">
    <ObjectType Code="http://www.w3.org/ns/ssn/
Sensor"/>
    <FilterGroup Operation="And">
          <Filter Attribute="http://trinidata.
ru/archigraph-mdm/LocalCode" Value="123"
Comparison="Equal">
          <Attribute AttributeId="http://trinidata.ru/
archigraph-mdm/archive" Type="Literal" Value="false" />
    </FilterGroup>
</GetObjectsGroup>
```

1 Полное описание синтаксиса запросов АрхиГраф.MDM приведено в документе https://trinidata.ru/files/ArchiGraphAPI.pdf

В синтаксисе JSON тот же запрос имеет вид:

```
{"GetObjectsGroup": {
    "ObjectType":[ {"Code":"http://www.w3.org/ns/ssn/
Sensor"} ],
    "FilterGroup":[ { "Operation":"And",
          "Filter":[ {
                "Attribute":"http://trinidata.ru/
archigraph-mdm/LocalCode",
                "Value":"123",
                "Comparison":"Equal"
                }, {
                "Attribute":"http://trinidata.ru/
archigraph-mdm/archive",
                "Value":"false",
                "Comparison":"Equal"
                } ]
          } ]
    }
}
```

В условии фильтра в этом запросе используется служебный атрибут АрхиГраф.MDM `http://trinidata.ru/archigraph-mdm/LocalCode`, который предназначен для хранения идентификаторов объектов во внешних автоматизированных системах.

В ответ на приведенный запрос будет получен либо пустой пакет Items, что говорит об отсутствии датчика с указанным кодом в данных MDM, либо пакет Items со вложенным элементом Item, в свойстве Code которого будет содержаться код MDM (идентификатор, URI) объекта, представляющего датчик в MDM.

2) Если датчик отсутствует в системе, необходимо сформировать запрос на его создание. Запрос имеет следующий вид в синтаксисе XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<UpdateObject Endpoint="Demo" Originator="test"
OperationId="000004">
    <Item LocalCode="123" OperationId="0000041">
          <Type TypeId="http://www.w3.org/ns/ssn/
Sensor" />
          <Attribute Type="Literal"
AttributeId="http://www.w3.org/2000/01/rdf-
schema#label" Value="Датчик 123" />
    </Item>
</UpdateObject>
```

В параметре LocalCode тега Item передается код датчика в системе, откуда получена информация о нем. Это значение будет записано в служебный атрибут `http://trinidata.ru/archigraph-mdm/LocalCode`, по которому в первом запросе мы извлекали информацию об объекте. В ответ на этот запрос MDM вернет сообщение о создании объекта (пакет OperationResults), в котором будет содержаться присвоенный ему MDM глобальный код (URI).

Разумеется, в реальном сценарии у датчика будет куда больше значений свойств, в том числе информация о координатах его места расположения – принципы работы с геометрией и географией в АрхиГраф.MDM мы рассмотрим далее.

3) Если датчик уже существует в MDM, необходимо сравнить значения его свойств (которые будут получены в ответ на запрос из п. 1) с теми значениями, которые получены из внешней системы. Если значения каких-либо свойств отличаются, нужно сформировать запрос на изменение свойств датчика, который будет полностью идентичен запросу из п. 2) – единственное отличие состоит в том, что вместо атрибута LocalCode нужно передать в теге Item атрибут Code, содержащий уникальный идентификатор уже существующего датчика в MDM. В теле запроса можно перечислить только те атрибуты, значения которых изменились – значения остальных атрибутов, не указанных в запросе, останутся в MDM неизменными.

4) По окончании обработки нужно сравнить список датчиков в MDM со списком, полученным от системы-источника. Для этого нужно получить полный список датчиков из MDM запросом:

```
<?xml version="1.0" encoding="UTF-8"?>
<GetObjectsGroup Endpoint="Demo" Originator="test">
    <ObjectType Code="http://www.w3.org/ns/ssn/
Sensor"/>
</GetObjectsGroup>
```

Если какой-либо из существующих в MDM датчиков не указан в данных внешней системы, т.е. был демонтирован, то нужно или физически удалить его из MDM запросом DeleteObject, или пометить как неактуальный, присвоив его атрибуту `http://trinidata.ru/archigraph-mdm/archive` значение false.

В результате выполнения приведенного алгоритма список датчиков в MDM будет актуализирован. Далее мы можем перейти к конструированию алгоритма записи в MDM временных рядов – показаний датчиков, тревог и созданных на их основании событий.

Обработка данных с помощью правил логического вывода

Простейший вариант скрипта, записывающего в MDM результаты измерений и тревоги, поступающие от датчиков, можно реализовать с помощью уже рассмотренных нами видов запросов. Однако при этом часто требуется организовать какую-либо логическую обработку поступающих данных. Одно из преимуществ использования онтологий в корпоративных автоматизированных системах состоит в том, что такую обработку можно перенести с уровня программного кода на уровень правил логического вывода. Такие правила можно изменять по ходу работы программы – а значит, менять логику ее функционирования без вмешательства в программный код или перезапуска.

Наиболее современной спецификацией, определяющей способы выражения в онтологиях правил логического вывода, является SHACL[1]. Первоначальным назначением этой спецификации было представление правил контроля логической целостности данных – эта ее часть называется SHACL Constraints. Затем спецификация была расширена дополнительными возможностями (SHACL Advanced Features), среди которых – создание правил логического вывода для дополнения фактов, внесенных в онтологию в качестве аксиом. Подчеркнем, что АрхиГраф.MDM обеспечивает выполнение правил логического вывода SHACL не только для данных, находящихся в графовой СУБД (RDF triple store), но и для данных из любых хранилищ под управлением MDM.

Принцип работы ограничений SHACL Constraints состоит в следующем. Для одного или нескольких классов модели создается правило, которое проверяет какое-либо условие, заданное логическим выражением, в котором элементарные условия объединены скобками и операторами И, ИЛИ. Правило само по себе является индивидуальным объектом онтологии, записанным в соответствии с метамоделью, определенной спецификацией SHACL. В момент создания или изменения индивидуальных объектов классов, к которым привязано правило, происходит проверка логического условия, и если оно не выполняется – в онтологии создается индивидуальный объект, представляющий результат проверки. Этот объект ссылается на нарушенное правило и на объект, для которого оно не выполнено. Программные компоненты, работающие с онтологией, могут обрабатывать появление таких объектов – например, выводить их список в интерфейсе администратора системы или отправлять уведомления лицам, ответственным за качество данных.

В платформе АрхиГраф правила создаются при помощи конструктора, реализованного в составе продукта АрхиГраф.СУЗ. Рассмо-

1 https://www.w3.org/TR/shacl/, https://www.w3.org/TR/shacl-af/

трим в качестве примера создание правила, которое контролирует, что тревога обязательно порождена каким-либо датчиком, не помеченным как неисправный.

Форма свойств правила в интерфейсе АрхиГраф.СУЗ выглядит так:

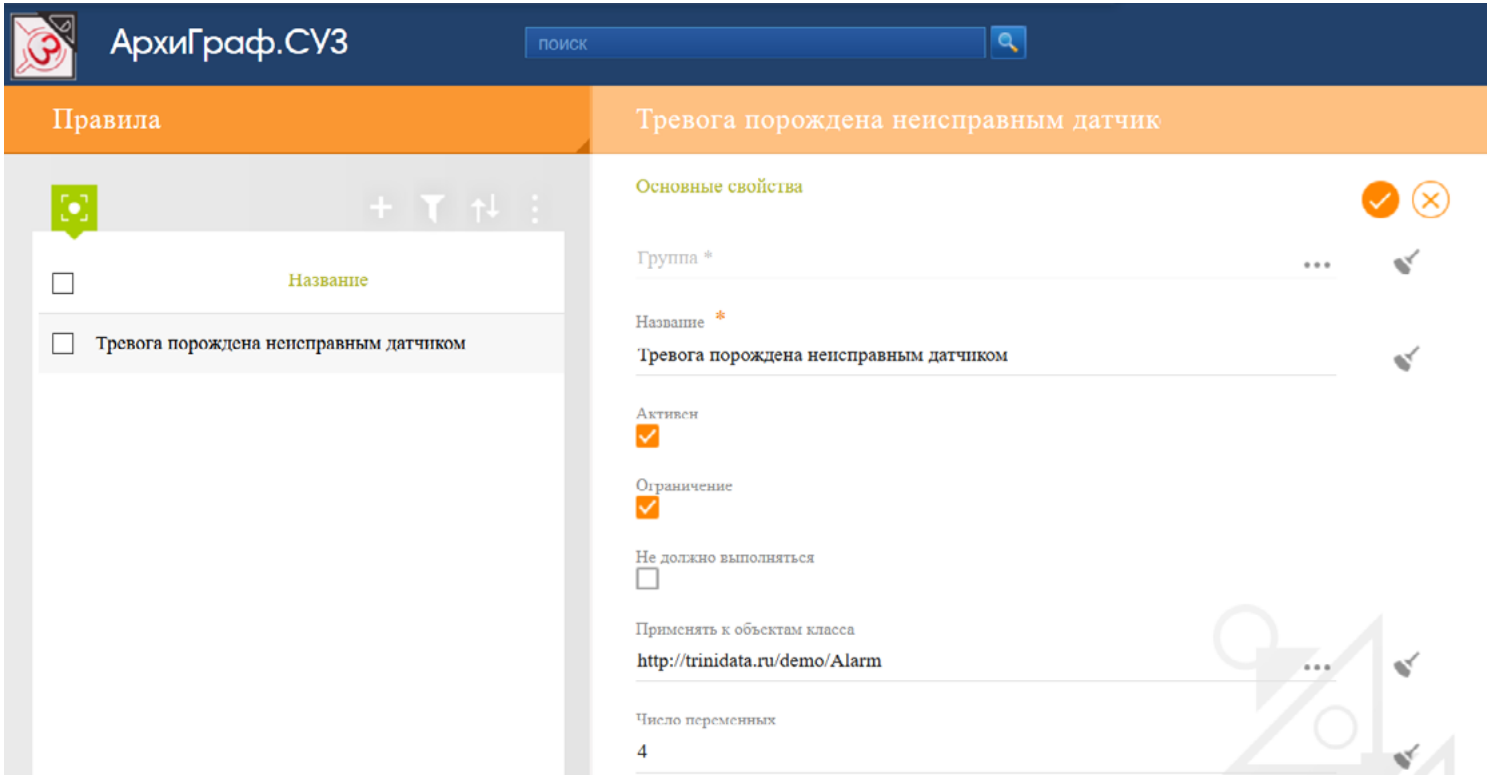


Рис. 12. Форма свойств правила в интерфейсе АрхиГраф.СУЗ

Обратим внимание на установленный переключатель «Ограничение», который определяет, что мы создали именно SHACL Constraint, и на поле «Применять к объектам класса», в котором выбран класс `Alarm`.

Далее в той же форме нужно создать условия правила:

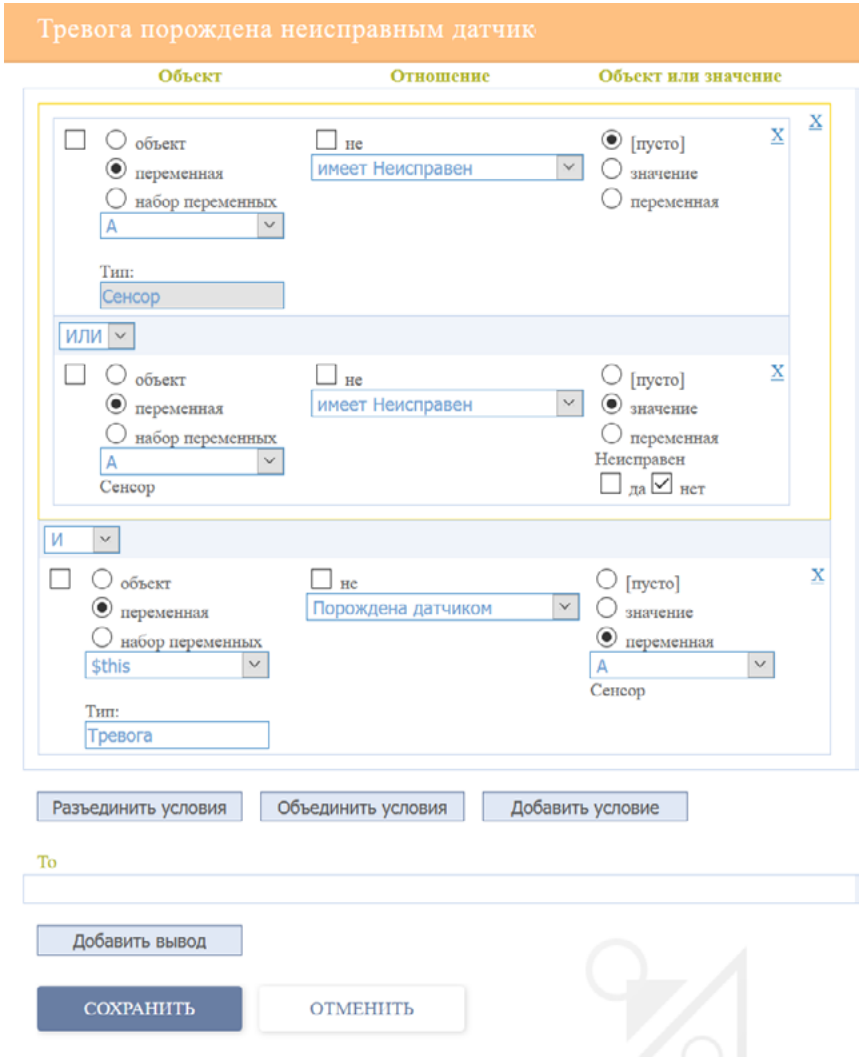


Рис. 13. Условия правила проверки логической целостности

Правило состоит из трех элементарных логических условий. Два первых проверяют, что некий датчик, обозначенный переменной A, не имеет значения булевского свойства «Неисправен», либо имеет его значение, равное false. Эти условия объединены между собой логическим ИЛИ.

С помощью логического И к ним присоединено третье условие, которое проверяет, что объект класса «Тревога», обозначенный переменной $this (эта переменная означает тот самый созданный или измененный объект, который проверяется правилом), порожден датчиком A. Вывод правилу проверки логической целостности не нужен: его результат состоит в соответствии либо в несоответствии объекта $this указанным условиям.

Таким образом, правилу будут удовлетворять только такие тревоги, которые порождены исправными датчиками. Тревоги, не связанные ни с каким датчиком, либо связанные с неисправным датчиком, не пройдут проверку. Условие правила в виде SPARQL-запроса можно увидеть в АрхиГраф.Мир на странице свойств класса, к которому оно привязано:

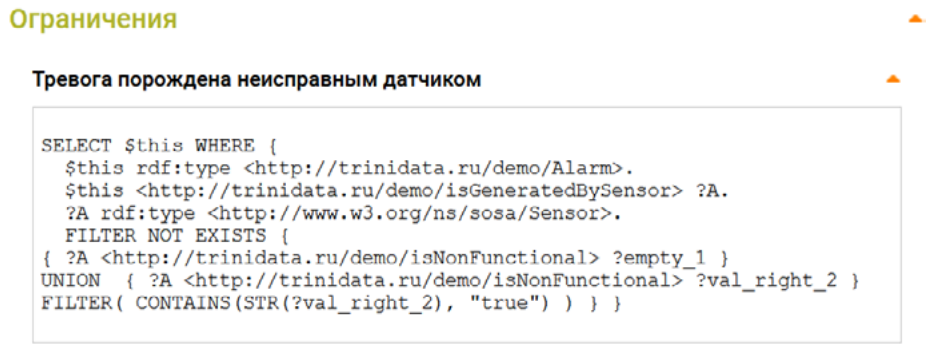


Рис. 14. SPARQL-запрос для проверки правила логической целостности[1]

Отметим, что платформа АрхиГраф содержит возможности, расширяющие спецификацию SHACL – например, условие правила может быть сформулировано как позитивное или негативное утверждение, т.е. правильным может быть или выполнение, или не выполнение условия (переключатель «Не должно выполняться» в свойствах правила).

Для того, чтобы проверка логических правил выполнялась, в запросе на изменение UpdateObject необходимо установить параметр Check="1". Если условия проверки не будут выполнены, то изменения в свойства объекта все равно будут внесены, но результат операции будет возвращен с сообщением о нарушении:

```
<OperationResults Endpoint="Demo" Destination="test"
OperationId="000004" >
  <OperationResult Result="success"
Code="Alarm_89609384-9524-40f6-bb5b-532d08bb750c"
Message="Тревога порождена неисправным датчиком"/>
</OperationResults>
```

Или в синтаксисе JSON:

```
{"OperationResults": {"Endpoint": "Demo",
"Destination": "test", "OperationId": "000004",
"OperationResult": [ { "Result": "success", "Code":
"Alarm_89609384-9524-40f6-bb5b-532d08bb750c",
"Message": "Тревога порождена неисправным датчиком" } ]
} }
```

Редактор АрхиГраф.Мир устанавливает этот флаг при редактировании объектов, поэтому если создать несоответствие правилу через редактор – на странице свойств отредактированного объекта информация о нарушении будет отображена в таком виде:

1 Строго говоря, здесь должен быть запрос ASK, а не SELECT, но ввиду ограниченного пока функционала внутренней точки доступа SPARQL в нашей платформе мы временно используем запрос SELECT и проверяем, вернет ли он какой-либо результат (что эквивалентно ответу true на запрос ASK) или не вернет (эквивалентно ответу false на запрос ASK)

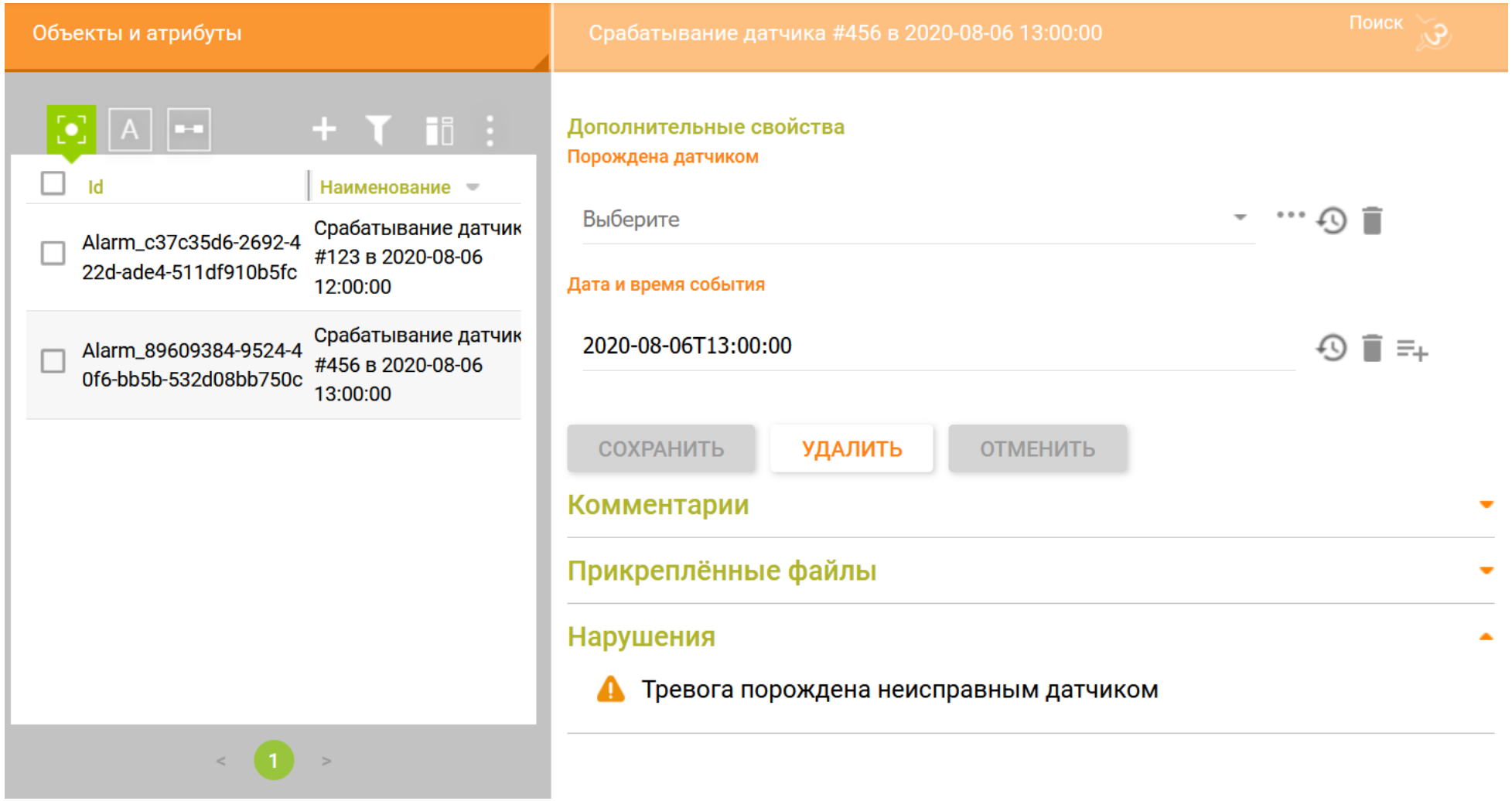


Рис. 15. Информация о нарушении контроля логической целостности объекта в АрхиГраф.Мир

В данных АрхиГраф.MDM будет сформирован следующий объект:

```
<Items Endpoint="autotest" Destination="test"
CacheUsed="1" >
    <Item Code="http://www.w3.org/ns/shacl#ValidationR
esult_5ec8dd6c694f4c3ee7777187a59be7a0" Name="http://
trinidata.ru/demo/Alarm_89609384-9524-40f6-bb5b-
532d08bb750c violation of 84b42aa3ed6f3232fce7d04980
0c6531">
        <Type TypeId="http://www.w3.org/ns/
shacl#ValidationResult" Name="ValidationResult"/>
        <Attribute AttributeId="http://www.
w3.org/2000/01/rdf-schema#label" Type="Literal"
Value="http://trinidata.ru/demo/Alarm_89609384-9524-
40f6-bb5b-532d08bb750c violation of 84b42aa3ed6f3232fce
7d049800c6531"/>
        <Attribute AttributeId="http://www.w3.org/ns/
shacl#sourceConstraintComponent" Type="Reference" Value
="84b42aa3ed6f3232fce7d049800c6531"/>
        <Attribute AttributeId="http://www.
w3.org/ns/shacl#focusNode" Type="Reference"
Value="Alarm_89609384-9524-40f6-bb5b-532d08bb750c"
Name="Срабатывание датчика #456 в 2020-08-06 13:00:00
"/>
    </Item>
</Items>
```

Этот объект можно получить, запросив объекты класса http://www.w3.org/ns/shacl#ValidationResult, у которых свойство http://www.w3.org/ns/shacl#focusNode указывает на объект, результаты проверки которого логическими правилами нужно отобразить. Свойство http://www.w3.org/ns/shacl#sourceConstraintComponent указывает на правило, которое было нарушено. Таким способом можно построить инструменты управления качеством данных, предоставляющие дата-стюарду сведения об элементах данных, не прошедших контроль логической целостности, или просто выводить предупреждения для пользователей, занимающихся вводом данных.

Правила логического вывода SHACL Rules работают схожим образом: они применяются при создании или изменении свойств индивидуальных объектов тех классов, к которым привязано правило. В случае, если логическое условие выполнено, правило создает в онтологии один или несколько новых фактов – то есть присваивает значения свойствам существующих объектов или оздает новые.

В качестве примера рассмотрим следующее правило: если сработал датчик задымления и одновременно датчик температуры показывает высокую температуру, то необходимо сформировать событие «Пожар». Правило будет иметь следующую структуру:

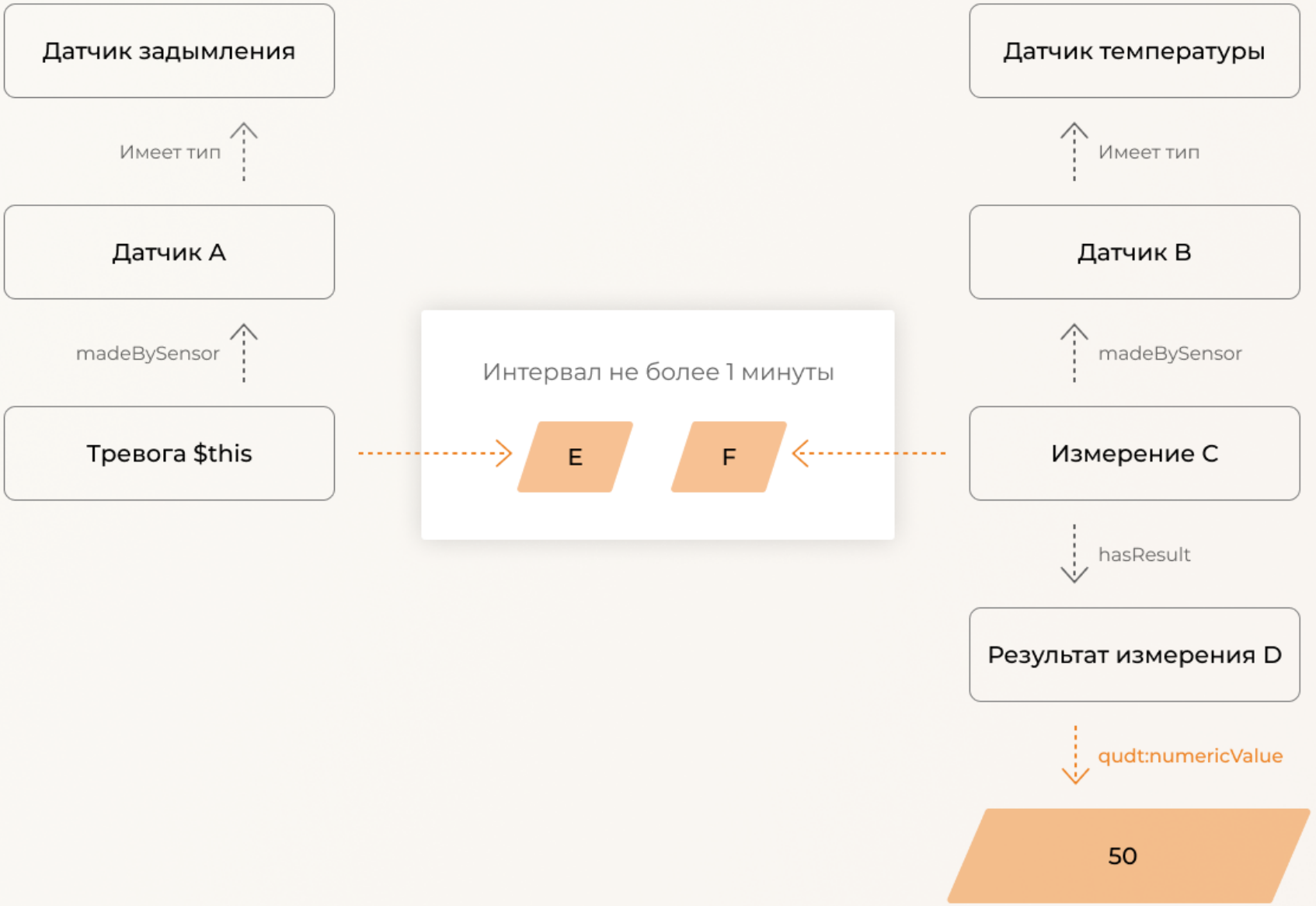
```
Если
    Датчик A является датчиком задымления
    Датчик B является датчиком температуры
    Тревога $this порождена датчиком A
    Измерение C выполнено датчиком B
    Измерение C имеет результат D
    Результат D имеет числовое значение > 50 (темпера-
тура больше 50 градусов)
    Тревога $this произошла в момент E
    Измерение C произошло в момент F
    Интервал между моментами E и F составляет не более
1 минуты
```

```
Тогда
    Создать событие "Новое"
    Событие "Новое" произошло в момент E
    Событие "Новое" относится к типу "Пожар"
```

Правило немного упрощено, т.к. в реальном сценарии должно еще проверяться совпадение места установки обоих датчиков.

В графическом виде условие приведенного правила можно представить так:

Рис. 16. Структура правила логического вывода



Как видно из этого примера, переменные в логических правилах могут означать как объекты, с которыми проверяемый объект связан значениями своих свойств-связей, так и значения его свойств-литералов. На значения свойств-литералов можно налагать ограничения, зависящие от типа литерала: для чисел и дат – «больше», «меньше», «равно» и др., для строк – «содержит» и «не содержит». Сравнивать литеральные значения можно как с константами (фиксированными значениями), так и с другими переменными того же типа.

Особое внимание обратим на проверку интервала между тревогой и измерением. Для этого в редакторе правил предусмотрен следующий инструмент:

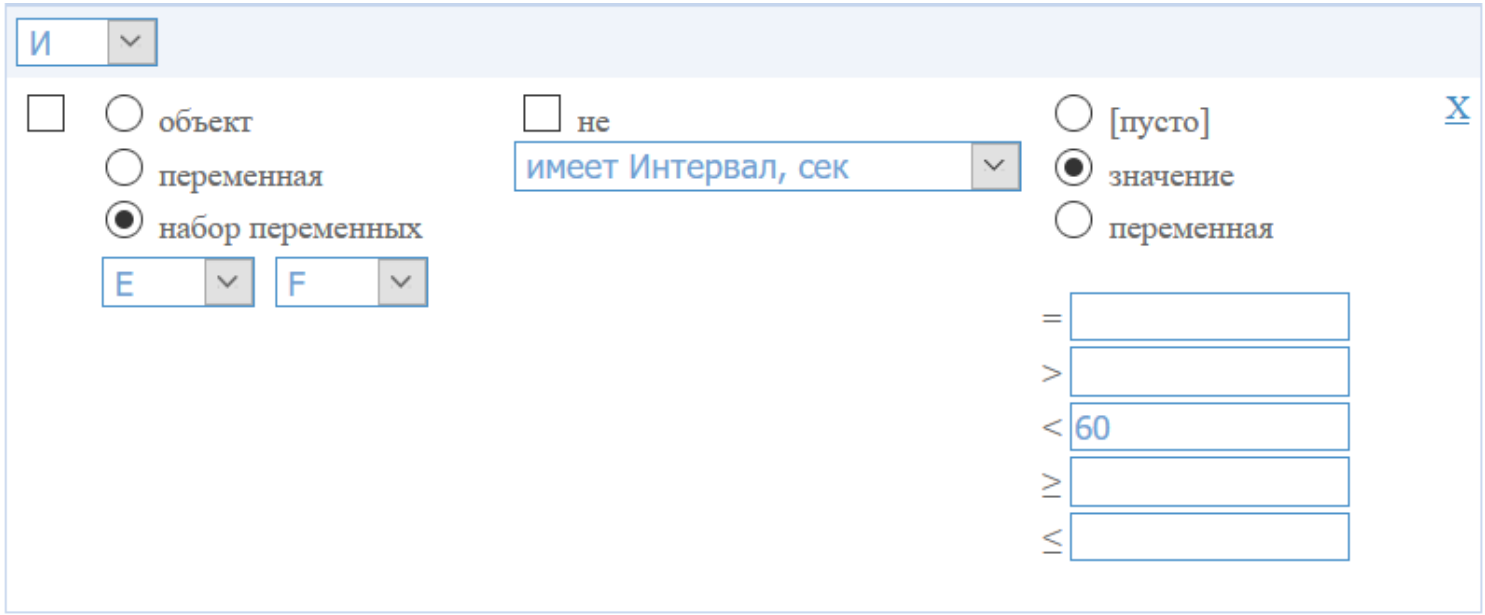


Рис. 17. Инструмент проверки интервала между значениями двух переменных

После того, как при конструировании правила определены переменные E и F, хранящие значения литерального свойства «Дата и время события» соответственно тревоги и измерения, в очередном условии правила можно установить переключатель «набор переменных», выбрать в выпадающих меню переменные E и F, в поле «Отношение» выбрать «Интервал, сек», в правой части задать условие на значение интервала: он должен быть меньше 60.

Для того, чтобы правило логического вывода выполнилось при создании или изменении тревоги, нужно передать в запросе UpdateObject параметр Logic=»1». MDM применит к изменяемому объекту все правила, привязанные к тем классам, к которым принадлежит объект. В случае, если условие правила выполнено, будет применен его вывод, который в конструкторе АрхиГраф.СУЗ имеет следующий вид:

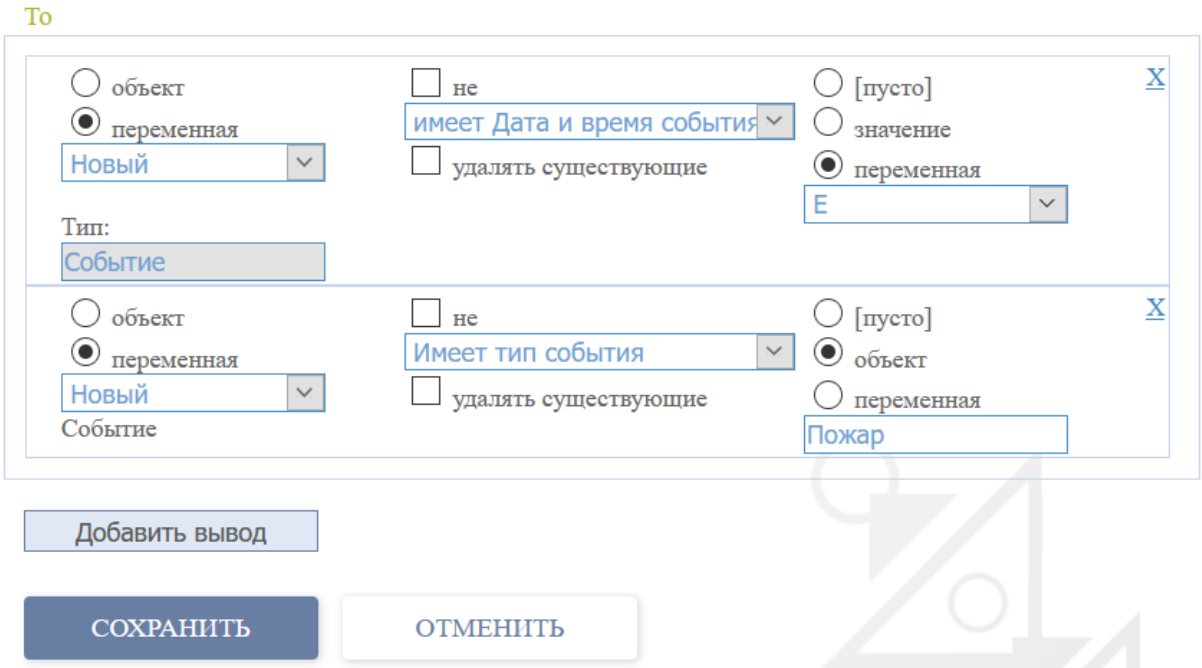


Рис. 18. Вывод правила в интерфейсе АрхиГраф.СУЗ

Таким образом, в результате срабатывания правила будет создан объект класса «Событие», имеющий тип «Пожар» (выбирается из классификатора типов событий), происшедшее в тот же момент, когда был получен сигнал тревоги. Можно дополнить набор свойств события – связать его с тревогой, местом установки датчика и др. На появление в модели объектов класса «Событие» должны быть подписаны программные компоненты пользовательского интерфейса, отвечающие за отображение ленты событий в приложении ситуационного центра, а также за рассылку оповещений ответственным лицам.

Подобные правила логического вывода удобно использовать для организации конвейеров логической обработки поступающей в систему информации, не зависящих от программного кода, полностью реализованных на уровне онтологической модели и программных средств работы с ней.

Получение данных от MDM по подписке

В нашем сценарии необходимо построить визуальный интерфейс ПО ситуационного центра, который будет отображать список тревог, поступающих от оборудования, и созданных на их основе событий. Было бы крайне нерационально постоянно опрашивать MDM для получения информации о новых событиях – гораздо правильнее, наоборот, получать от нее уведомления о них. Для решения этой задачи используется механизм подписок.

Любой программный компонент может подписаться на получение уведомлений о создании, изменении, удалении объектов определенных классов через API MDM. В нашем примере компоненту, обеспечивающему работу визуального интерфейса ситуационного центра, нужно подписаться на получение информации о тревогах и результатах измерений (объекты классов `Alarm` и `http://www.w3.org/ns/sosa/Result`). Для этого нужно выполнить запрос, который в синтаксисе XML имеет такой вид:

```
<UpdateSubscription Endpoint="demo" Originator="test">
    <Subscribe Format="json" Delayed="0" Objects="1"
Broker="Kafka" Host="127.0.0.1">
          <ObjectType Code="Alarm" />
          <ObjectType Code="http://www.w3.org/ns/sosa/
Result" />
    </Subscribe>
</UpdateSubscription>
```

В параметре Endpoint нужно передать идентификатор точки доступа, в параметре Originator – идентификатор отправителя запроса. В параметрах тега Subscribe передается тип брокера очереди и реквизиты подключения к нему. Параметр Model определяет, нужно ли получать уведомления об изменениях в TBox (структуре данных), а параметр Objects – об изменениях в ABox (конкретных объектах). Во вложенных тегах ObjectType передаются идентификаторы классов, на изменения сведений об объектах которых подписывается компонент. Как и в других запросах, идентификаторы классов со стандартным префиксом могут передаваться без префикса.

Тот же запрос в синтаксисе JSON:

```
{"UpdateSubscription":
    {"Endpoint": "demo", "Originator": "test",
"Subscribe":
          [ { "Format": "json", "Delayed": "0",
"Objects": "1", "Broker": "Kafka", "Host": "127.0.0.1",
"ObjectType":
                [ { "Code": "Alarm" }, { "Code":
"http://www.w3.org/ns/sosa/Result" } ]
          } ]
    }
}
```

В запросе указываются классы, интересующие подписчика, а также реквизиты очереди, в которую он ожидает получать пакеты с информацией об объектах этих классов. Очередь может быть создана в брокере Kafka или RabbitMQ на любом узле, с которым сможет установить соединение MDM.

Формат сообщений, поступающих от MDM по подписке, идентичен формату сообщений, которыми MDM отвечает на запросы GetObjectsGroup на получение информации о наборах объектов – только корневым тегом в них вместо Items является SubscriptionItems.

Существует возможность не только получать от MDM уведомления об изменениях, происходящих с моделью и объектами, но и отправлять в MDM любые запросы на изменение или чтение данных. Такой способ предпочтителен при организации обработки больших массивов данных, так как он гарантирует, что MDM не будет перегружена запросами. Для реализации такого способа нужно создать пару очередей в брокере RabbitMQ или Kafka: в одну из них приложение будет помещать запросы к MDM, через другую – получать ответы на них. Запросы имеют точно такой же формат, как при обращении через HTTP-интерфейс, и могут быть записаны в синтаксисе XML или JSON.

Для того, чтобы MDM начала обрабатывать очередь входящих сообщений и помещать ответы на них в исходящую очередь, нужно зарегистрировать в ней эту пару очередей. Для этого нужно выполнить следующую команду:

```
mdmctl add queue --in SYNCH_QUEUE_IN --out SYNCH_QUEUE_
OUT --input_host 127.0.0.1 --input_port 5672 --broker
RabbitMQ --input_login admin --input_password mypass
--output_host 127.0.0.1 --output_port 5672 --output_
login admin --output_password mypass
```

В синтаксисе этого вызова после команды add queue следуют параметры:

- --in SYNCH_QUEUE_IN – имя очереди, в которую будут помещаться запросы к MDM
- --out SYNCH_QUEUE_OUT – имя очереди, в которую MDM будет помещать ответы
- --input_host 127.0.0.1 – адрес или имя хоста, где размещен брокер входящей очереди
- --input_port 5672 – порт для подключения к брокеру входящей очереди
- --broker RabbitMQ – тип брокера, RabbitMQ или Kafka
- --input_login admin – логин к брокеру входящей очереди
- --input_password mypass – пароль к брокеру входящей очереди
- --output_host 127.0.0.1 – адрес или имя хоста, где размещен брокер исходящей очереди
- --output_port 5672 – порт для подключения к брокеру исходящей очереди

- --output_login admin – логин к брокеру исходящей очереди
- --output_password mypass – пароль к брокеру исходящей очереди

Для просмотра списка очередей, на обработку которых настроена MDM, нужно выполнить команду:

```
mdmctl show queue
```

В нашем примере бэк-энд пользовательского интерфейса ситуационного центра должен подписаться на сообщения о тревогах и новых результатах измерений, выполненных датчиками. Затем информация об этих объектах должна передаваться во фронт-энд по протоколу websocket. Фронт-энд, построенный по принципам реактивной верстки, должен обновляться для отображения актуальной информации для пользователей.

Важно отметить, что при помощи рассмотренного механизма можно подписаться на изменения не только в данных, но и в модели под управлением MDM. Это позволяет создавать приложения, управляемые со стороны онтологической модели: такие приложения должны быть готовы на изменения в модели и реагировать на них, изменяя алгоритмы своей работы. Как минимум приложение должно быть готово к изменению состава свойств информационных объектов, и при появлении новых свойств добавлять соответствующие им поля в экранные формы (можно построить метамодель для описания экранных форм и задавать в ней правила отображения свойств для того, чтобы указывать порядок их следования, временно скрывать или показывать поля для отображения тех или иных свойств). Более развитые приложения могут быть готовы и к появлению в модели новых типов информационных объектов и обрабатывать объекты этих типов, следуя логике, применимой к их надклассам (например, отображать объекты новых типов на плане здания, используя правила, определенные для надкласса объектов) или описанной в специальной части онтологической модели (отображать объекты новых типов, если для них создано правило отображения как отдельный объект в онтологии).

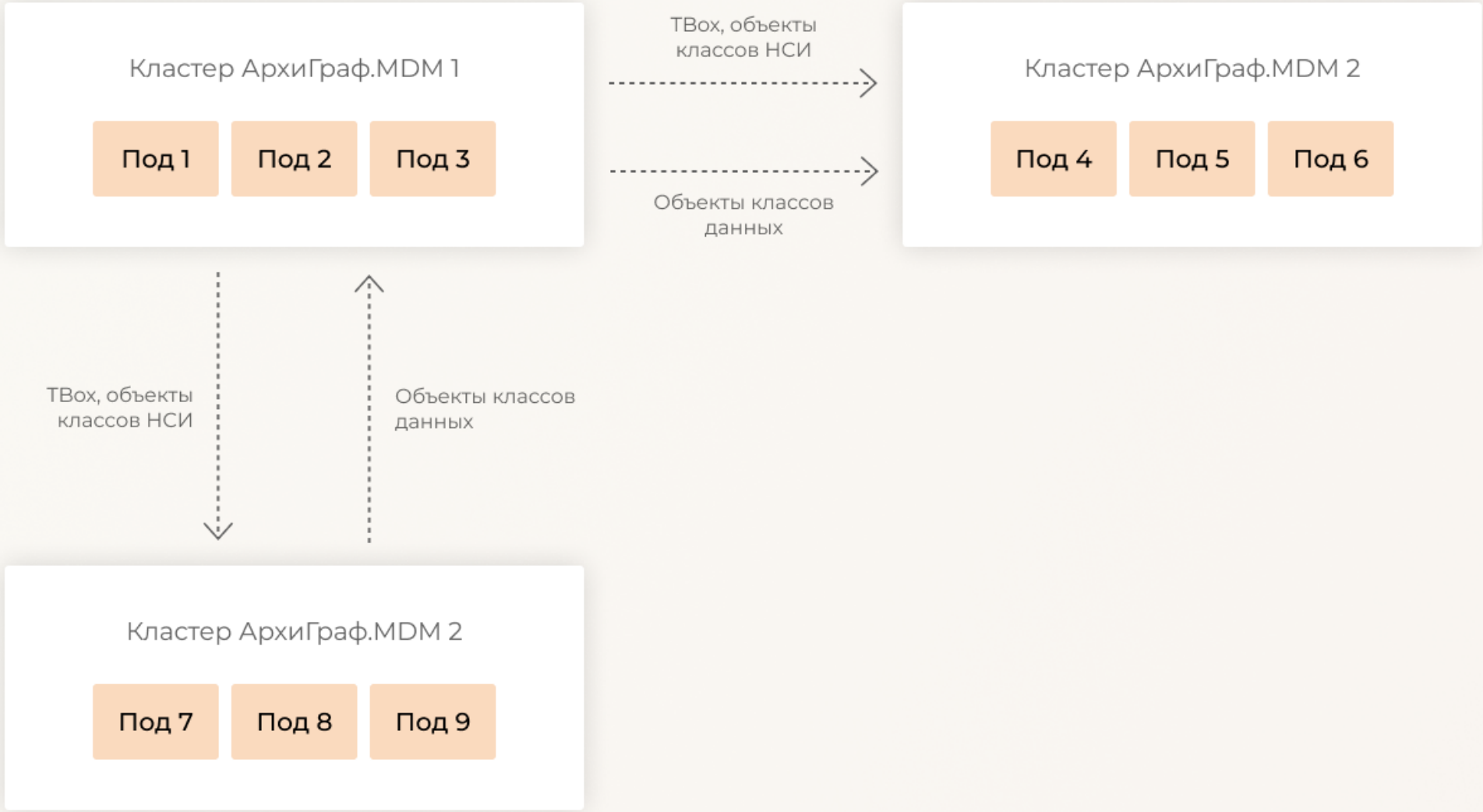
Синхронизация нескольких экземпляров MDM

Важно отметить, что разные экземпляры MDM могут подписываться на уведомления об изменениях модели и данных друг друга. Это позволяет организовать полную или частичную синхронизацию данных между несколькими экземплярами MDM.

Если быть точнее, то синхронизация настраивается не между отдельными экземплярами, а между кластерами под управлением Kubernetes или Docker Swarm, каждый из которых содержит N подов MDM (см. главу «Развертывание»). Как правило, один из таких кластеров выбирается в качестве основного, остальные обслуживают какие-либо подразделения или площадки, на которых ведет деятельность организация. Изменения в модели и НСИ выполняются только на основном кластере и по подписке распространяются на дополнительные кластеры. Фактические данные, наоборот, чаще вносятся на дополнительных кластерах и передаются оттуда по подписке на основной кластер. Графически схема взаимодействия показана на рис. 19.

В зависимости от функциональных задач подписка может осуществляться не на все классы (хотя MDM предоставляет возможность подписаться на все, передав в качестве идентификатора класса в запросе на подписку значение __root__). Представим, что организация в нашем примере обслуживает не одно, а несколько зданий, расположенных в разных городах. В каждом здании есть собственная служба ситуационного центра. Использовать единый централизованный кластер АрхиГраф.MDM всем ситуационным центрам было бы рискованно, так как отключение интернета приведет к неработоспособности всей системы в целом. Необходимо обеспечить автономный режим работы, при котором кластер АрхиГраф.MDM расположен непосредственно в здании, зарезервирован, снабжен надежным резервным источником питания и сохраняет работоспособность в случае отключения интернета. В этом случае основной кластер MDM может находиться на серверах управляющей компании, а дополнительные кластеры – в каждом из обслуживаемых зданий. Редактирование модели осуществляется только со стороны основного кластера, внесенные там изменения автоматически немедленно распространяются на все площадки. Внесение фактических данных – например, по составу датчиков – наоборот, происходит на вспомогательных площадках и реплицируется на основную. Часть событийной информации может мигрировать по тому же пути: например, имеет смысл передавать важные события и тревоги на центральную площадку, но вряд ли стоит транслировать туда временные ряды показаний всех датчиков температуры и др. величин.

Рис. 19. Схема синхронизации между несколькими кластерами АрхиГраф.MDM



Похожая схема может быть использована для создания резервного кластера MDM. В этом случае настраиваются два идентичных кластера, между ними организуется взаимная подписка на всю модель и данные. Пользовательские компоненты должны обладать реквизитами подключения к обоим кластерам, и при недоступности основного автоматически переключаться на резервный. Таким образом, в случае аварии основного кластера пользователи незаметным для себя образом перейдут на работу с резервным кластером – до тех пор, пока не будет восстановлена работа основного. Маршрутизацию запросов на кластер можно организовать и через единую точку входа на прокси-сервере, но это создает потенциальную уязвимость в виде самого прокси-сервера. Те изменения, которые будут внесены в модель и данные в период работы с резервным кластером, будут автоматически переданы на основной благодаря механизму подписки. Поскольку подписка функционирует через очереди Kafka или RabbitMQ, передаваемые изменения будут накапливаться в очереди до тех пор, пока вышедший из строя кластер не восстановит работу и не начнет обрабатывать эти сообщения.

Точка доступа SPARQL

Мы рассмотрели основные способы использования платформы АрхиГраф. В принципе, этих способов достаточно для построения автоматизированных систем, использующих онтологические модели и соответствующие способы обработки данных, такие как правила логического вывода. Далее мы рассмотрим некоторые специальные возможности платформы АрхиГраф.

В качестве основного варианта взаимодействия программных компонентов с платформой мы рекомендуем использовать ее REST-интерфейс или обмен через очереди, но «родным» способом доступа к онтологическим данным является протокол SPARQL. Платформа АрхиГраф предоставляет точку доступа SPARQL, которая обеспечивает возможность работы с любыми данными, размещенными в хранилищах платформы. Подчеркнем, что возможно извлечение с помощью SPARQL и данных, физически расположенных в реляционных или документ-ориентированных СУБД, подключенных к АрхиГраф.MDM.

На момент написания этого текста точка доступа SPARQL предоставляет только возможность извлечения данных, и не обеспечивает их записи или удаления. Точка доступа SPARQL находится в процессе активного развития и еще не поддерживает всех возможностей протокола и имеет экспериментальный статус. Тем не менее, ее можно использовать в сценариях извлечения связанных объектов – несколько запросов к REST-интерфейсу можно заменить на один запрос к SPARQL.

Чтобы наглядно продемонстрировать работу с точкой доступа SPARQL, воспользуемся панелью выполнения запросов Apache Fuseki. В ней можно указать произвольный URL точки доступа SPARQL, и таким образом выполнить запрос не к Fuseki, а к любой точке доступа. Для примера мы взяли точку доступа, предоставляемую демо-версией АрхиГраф.MDM, расположенной по адресу http://demo.trinidata.ru/mdm/. Добавив к этому адресу код точки доступа autotest и адрес интерфейса чтения из точки доступа query, получим полный адрес http://demo.trinidata.ru/mdm/autotest/query. Этот адрес необходимо ввести в поле SPARQL Endpoint панели Fuseki, как показано на рис. 20.

Построим запрос, который извлечет все тревоги за определенный интервал времени, датчики, породившие эти тревоги, и названия датчиков. На рис. 20 показан запрос и результат его выполнения.

Обратим внимание, что в нашем примере объекты классов Alarm и Sensor находятся в базе данных Postgres. Это не мешает АрхиГраф.MDM ответить на запрос так, как будто объекты расположены в графовой базе данных.

Приведем другой пример запроса. Пусть мы хотим одним запросом получить измерения, выполненные определенным датчиком, и их результаты. Запрос и его результат выглядят так, как показано на рис. 21:

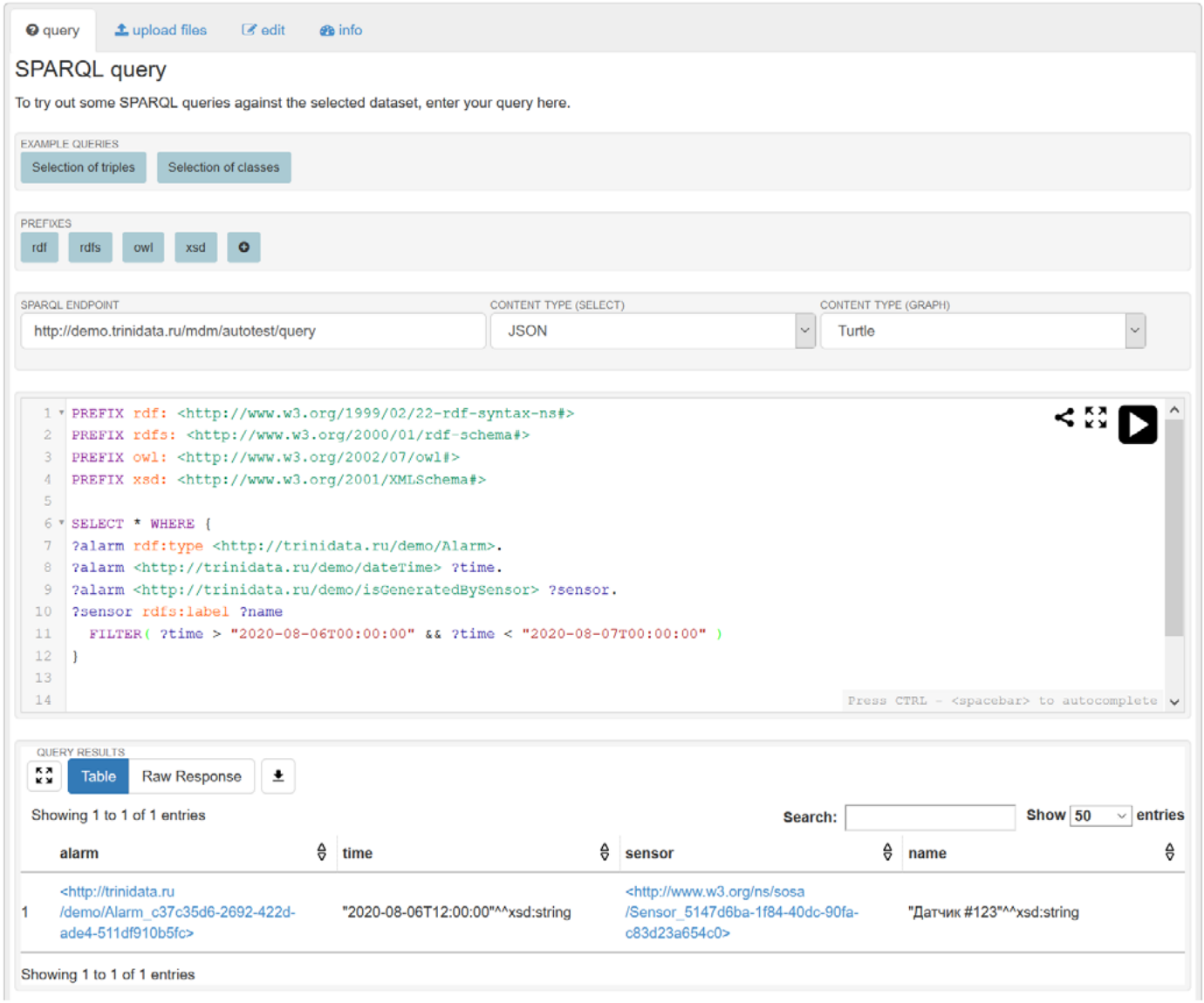


Рис. 20. Запрос к SPARQL точке доступа АрхиГраф.MDM, выполненный через панель Apache Fuseki

```
SELECT * WHERE {
?observation rdf:type <http://www.w3.org/ns/sosa/Observation>.
?observation <http://trinidata.ru/demo/dateTime> ?time.
?observation <http://www.w3.org/ns/sosa/madeBySensor> <http://www.w3.org/ns/sosa/Sensor_bea3e5e0-df58-498b-ac2b-5dbf702131c2>.
?observation <http://www.w3.org/ns/sosa/hasResult> ?result.
?result <http://trinidata.ru/demo/hasUnit> ?unit.
?result <http://qudt.org/vocab/unit/numericValue> ?value
}
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries

Search: Show 50 entries

	observation	time	result	value	unit
1	<http://www.w3.org/ns/sosa/Observation_5a6aebce-bdb4-49ba-bf69-3a481527c0db>	"2020-08-08T13:21:22"	<http://www.w3.org/ns/sosa/Result_a9a62080-c682-47ba-b713-40e819e21360>	"22"	<http://qudt.org/vocab/unit/DEG_C>

Showing 1 to 1 of 1 entries

Рис. 21. Запрос на получение измерения и его результатов

Работа с географией и геометрией

В бизнес-приложениях, в том числе в ПО ситуационных центров, часто необходимо работать с данными, имеющими пространственную привязку. Платформа АрхиГраф предоставляет специальные встроенные типы данных, основанные на GeoJSON: PointType, LineType, PolygonType и MultipolygonType для описания геометрических форм и работы с ними.

В рассматриваемом нами примере ситуационного центра необходимо хранить контуры зданий и элементов прилегающей территории, а также координаты мест установки датчиков и другого оборудования. Одним из типичных запросов с использованием геометрии может быть получение всех датчиков, находящихся на территории определенного здания.

Для отражения координат и контуров в модели данных необходимо создать свойство-литерал `hasPlace` типа PointType, присущее объектам класса Sensor, и свойство `hasContour` типа PolygonType, присущее объектам класса `Building` (Здание). Значения для этих свойств будем задавать в виде географических координат WGS 84 в формате GeoJSON, где значения широты и долготы задаются в виде действительных чисел. Например, координаты датчика (значение свойства типа PointType) могут быть заданы так: [56.1488, 63.5655]. Контур здания (значение свойства типа PolygonType) в простейшем случае четырехугольного здания может быть задано так: [[[56.1489, 63.5653], [56.1489, 63.5658], [56.1481, 63.5658], [56.1481, 63.5653]]].

Возможность поиска объектов по координатам зависит от того, предоставляет ли такую функциональность хранилище, в котором находятся объекты соответствующих классов – в других хранилищах значения геометрических свойств будут сохраняться, но не будут использоваться в операциях поиска. Например, возможность работы с геометрическими типами данных предоставляет база данных Postgres с расширением PostGIS. Необходимо создать отдельные поля в таблице Postgres с типами Point или Polygon, и задать их мэппинг на свойства онтологии.

Нужно иметь в виду, что MDM сохраняет значения геометрических свойств дважды: в исходном текстовом варианте и в виде GeoJSON для обработки алгоритмами геопоиска. Если используется хранилище postgresql_jsonb, то исходное текстовое значение будет храниться в поле data вместе с другими атрибутами, но дополнительно нужно для каждого геометрического атрибута создать отдельный столбец типа geography (если хранилище типа postgres, то надо создавать и столбец для хранения текстового значения). При задании мэппинга для столбца с геометрией нужно использовать суффикс |geo в поле с идентификатором свойства в модели, например: hasPlace|geo. Полностью команда задания мэппинга для столбца с геоданными может выглядеть так:

```
mdmctl bind property "hasPlace|geo" "Хранилище данных
Demo Postgres" test place
```

После настройки мэппинга и сохранения в MDM координат конкретных объектов можно будет выполнить поиск датчиков, расположенных в пределах здания, следующим образом. Нужно извлечь координаты интересующего здания, разделить полученный массив GeoJSON на отдельные точки, и сформировать запрос к MDM такого вида:

```
<?xml version="1.0" encoding="UTF-8"?>
<GetObjectsGroup Endpoint="autotest" Originator="test">
    <ObjectType Code="http://www.w3.org/ns/sosa/
Sensor"/>
    <Geometry Type="Polygon" Attribute="hasPlace">
          <Coordinates Longitude="56.1489" Latitude =
"63.5653" />
          <Coordinates Longitude="56.1489" Latitude =
"63.5658" />
          <Coordinates Longitude="56.1481" Latitude =
"63.5658" />
          <Coordinates Longitude="56.1481" Latitude =
"63.5653" />
    </Geometry>
</GetObjectsGroup>
```

Каждая точка многоугольника должна превратиться в отдельный элемент Coordinates в запросе. Можно выполнить такой запрос и в формате JSON, структура элементов JSON будет аналогична приведенному примеру.

Поддержка многоязычности

Платформа АрхиГраф.MDM предоставляет возможность работы со значениями строковых атрибутов-литералов на разных языках. Это означает, что для каждого атрибута каждого объекта могут быть заданы значения, аннотированные языковыми метками – например, название объекта (`rdfs:label`) на русском и английском языках. В системе можно зарегистрировать любое количество языков.

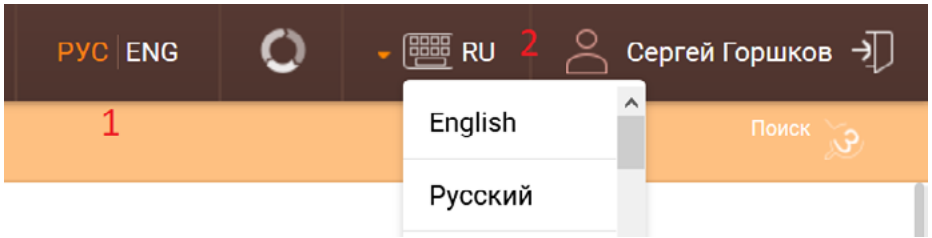


Рис. 22. Переключатели языка в интерфейсе АрхиГраф.Мир

Для создания языка в АрхиГраф.MDM нужно выполнить такую команду:

```
mdmctl add language it "Italiano"
```

Здесь после команды add language указывается двухбуквенная аббревиатура (ISO-код) языка, а затем в кавычках – его читаемое название.

Для просмотра списка языков, доступных в системе, нужно выполнить команду:

```
mdmctl show language
```

В интерфейсе редактора АрхиГраф.Мир в правой верхней части экрана присутствуют два переключателя языка:

Переключатель, обозначенный цифрой 1, позволяет выбрать язык интерфейса – на нем отображаются подписи полей и кнопок. Переключатель, обозначенный цифрой 2, позволяет выбрать язык контента из списка языков, зарегистрированных в MDM. После того, как выбран язык контента, отличающийся от основного языка, во всех текстовых полях ввода значений свойств объектов (таких как название объекта) начинают отображаться значения на соответствующем языке. При сохранении объекта все сохраненные строковые значения также помечаются выбранным языком. Эти языковые метки в дальнейшем могут использовать любые приложения, работающие с данными MDM через ее программный интерфейс.

Полнотекстовый поиск в онтологических данных

На данный момент встроенные инструменты базовой версии платформы АрхиГраф не поддерживают полнотекстовый поиск, однако существует решение, позволяющее реализовать его с помощью дополнительных программных компонентов.

Для реализации полнотекстового поиска необходимо развернуть кластер Solr, который обеспечивает построение полнотекстовых индексов. Как правило, в онтологии присутствует ограниченное число классов, объекты которых имеют текстовые свойства, по которым нужно построить индекс – например, это может быть класс «Документы», каждый объект которого представляет собой какой-либо документ из корпоративного хранилища. В Solr создается схема, позволяющая хранить как структурированные свойства объектов тех классов, по свойствам которых строится полнотекстовый индекс, так и значения индексируемых текстовых полей.

Создание индекса выполняется специальным скриптом-краулером, который обрабатывает исходные документы, создает соответствующий каждому из них объект в хранилище АрхиГраф.MDM и зеркальный объект в Solr. Другой программный компонент (адаптер хранилища MDM) обеспечивает поиск с учетом морфологии и релевантности запросу по таким объектам, получение сниппетов документов от Solr и их передачу в клиентское приложение. Детали настройки этих компонентов здесь не приводятся, так как они не входят в базовый комплект поставки продуктов платформы АрхиГраф.

Глоссарии, лексические модели и преобразование в факты текста на естественном языке

Онтологии предназначены для обработки структурированных фактов, каждый из которых представляет собой элемент знания. Но человеческое мышление неразрывно связано с языком, поэтому трудно говорить о построении моделей знания, не включающих лексический компонент. Часто возникающей задачей является также преобразование текста на естественном языке в набор фактов, правило логического вывода или в запрос к структурированным знаниям. В этом разделе мы рассмотрим инструменты, которые предоставляют средства онтологического моделирования и платформа АрхиГраф для решения подобных задач.

Среди онтологий, применяемых для создания моделей лексики и текста, необходимо выделить SKOS (Simple Knowledge Organization System)[1] – популярную онтологию для создания глоссариев, описывающих термины и связи между ними. В нашей практике мы широко используем SKOS для построения модели лексики, связанной с онтологией какой-либо предметной области. Добавление связи между термином (`skos:Concept`) и классами или свойствами модели предметной области позволяет связать между собой эти две модели. На основе таких моделей удобно строить приложения-глоссарии, позволяющие пользователю просматривать «словарь» лексики определенной предметной области, осуществлять навигацию по терминам (переходить к более широким или более узким терминам, синонимам и др.), переходить к элементам модели предметной области, связанным с термином. С помощью SKOS можно выражать наличие нескольких определений у каждого термина, разные значения термина в разных контекстах.

Онтология NIF (NLP Interchange Format)[2] предназначена для онтологического представления результатов разбора текста, выполненного инструментами NLP (Natural Language Processing, обработка текстов на естественном языке). NIF позволяет смоделировать в онтологии элементы текста: абзацы, фразы, отдельные слова, их положение друг относительно друга. Существуют также онтологии для представления NER (Named Entity Recognition, обнаружение именованных сущностей) – шаблонов обнаружения в тексте именованных сущностей или значений литеральных

1 https://www.w3.org/TR/skos-reference/

2 https://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core/nif-core.html

свойств (дат, интервалов и др.), выражаемых в тексте эмоциональных оценок и др.

С использованием таких онтологий нами построен дополнительный компонент нашей платформы – АрхиГраф.Логос, представляющий собой сервис преобразования текста на естественном языке в набор фактов или запрос. С помощью этого сервиса, например, можно реализовать сценарии поиска структурированной информации в ответ на запросы, сформулированные на естественном языке. Пусть пользователь вводит запрос в строку поиска в автоматизированной системе ситуационного центра: «Датчики температуры в корпусе А, показывавшие температуру больше 20 градусов Цельсия сегодня после 12:00». Если модель предметной области сопровождена моделью лексики, в которой каждому из использованных в этой фразе терминов («датчик температуры», «корпус», «показывавший» и т.д.) сопоставлен какой-либо класс или свойство модели предметной области, запрос будет распознан и преобразован в SPARQL-запрос. Результаты SPARQL-запроса, выполненного MDM, будут представлять собой набор структурированных данных, являющийся ответом на запрос пользователя. Сервис распознавания текста толерантен к отсутствию определений некоторых терминов в онтологии, и пытается «достроить» смысл фразы, выстраивая предполагаемые связи между теми терминами, смысл которых удалось распознать – это значительно повышает вероятность успешного ответа на запрос пользователя.

Возможен и обратный сценарий использования – преобразование текста в факты. Представим, что одной из задач ситуационного центра является регистрация и классификация сообщений, поступающих от персонала и посетителей здания. Сообщения принимаются в виде звонков на телефон дежурного, который записывает сообщенный заявителем текст в систему. Пусть заявитель сообщил дежурному, что «на втором этаже корпуса А в западном крыле наблюдается задымление». Система должна автоматически классифицировать это сообщение и связать его с другой имеющейся информацией. По приведенному тексту не составляет труда определить тип события («Задымление») и его локализацию (корпус А, второй этаж, западное крыло). Система может автоматически обработать полученный текст и дополнить объект, представляющий сообщение в онтологии, структурированными свойствами, ссылающимися на тип и место события.

Наконец, третий сценарий работы с текстом на естественном языке состоит в автоматической или полуавтоматической генерации правил логического вывода. Работа с конструктором правил требует некоторой квалификации от аналитика, и при большом числе необходимых правил может потребовать значительного времени. Сервис распознавания естественного языка позволяет превращать выражения типа «Если в здании возникло задымление, необходимо проинформировать службу 112» в правила SHACL Rules. Например, в данном случае правило может создать объект класса «Задача» для дежурного, описывающий его обязанность проинформировать службу 112, в случае появления в данных объекта класса «Событие», имеющего тип «Задымление» и локализованного в здании.

В заключение приведем пример преобразования текста запроса на естественном языке в SPARQL-запрос при помощи демонстрационного интерфейса сервиса АрхиГраф.Логос (рис. 23).

```
@prefix fire: <http://trinidata.ru/fire/>
@prefix mdm: <http://trinidata.ru/archigraph-mdm/>
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT *
WHERE {
  ?vp_mflhdexj rdf:type fire:ВозгораниеПожар ;
     fire:объектГорения ?z_vq0if7i7 , ?pp_s0kfu8fb .

  ?pp_s0kfu8fb rdf:type fire:ПодъездПарадная ;
     fire:относитсяККомпонентуПространственнойЗастройки ?z_vq0if7i7 ;
     fire:имеетНомер 1 .

  ?z_vq0if7i7 rdf:type fire:Здание ;
     fire:ИмеетМатериалСтен fire:МатериалСтенКирпич ;
     fire:ИмеетФункциюЗданияПомещения fire:ФункцияАдминистративное .
}
```

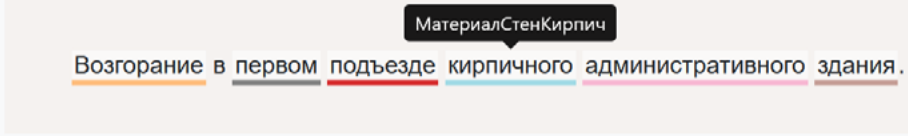


Рис. 23. Результат преобразования запроса на естественном языке в SPARQL-запрос

Как видно из примера, система определила объекты, о которых говорится в запросе -возгорание, подъезд, здание. Для каждого объекта определен тип. Установлены связи объектов между собой, следующие из грамматической структуры предложения. Выявлены дополнительные свойства, присущие каждому из объектов. В результате построен SPARQL-запрос, выполнение которого вернет объекты, отвечающие условиям сформулированного пользователем запроса на естественном языке. Аналогичным образом сервис может выполнять преобразование выражений на естественном языке в набор фактов, которыми нужно дополнить имеющийся в системе набор данных.

Получение данных из внешних источников через MDM

К АрхиГраф.MDM можно подключить реляционные и документ-ориентированные хранилища данных и работать с ними через ее программный интерфейс. А можно ли подключить к ней внешние хранилища, например, базы данных каких-либо автоматизированных систем или даже предоставляемые ими веб-сервисы? Да, можно. Для этого наша платформа предлагает два способа: создание «прозрачных» адаптеров хранилищ MDM или использование расширения для MDM – «логической витрины данных».

«Прозрачные» адаптеры названы так потому, что незаметным для вызывающего компонента образом обращаются к внешним хранилищам для получения данных. Создание «прозрачного» адаптера требует программирования и подключения к MDM программного модуля, способного работать с каким-либо внешним источником данных. Этот модуль должен быть реализовывать определенный программный интерфейс, рассмотрение которого выходит за рамки данного документа. Такой программный модуль становится одним типов хранилищ MDM, после чего конкретный источник данных можно добавить в ее конфигурацию командой mdmctl add storage и сопоставить один или несколько классов этому источнику.

«Логическая витрина данных» – более универсальный вариант, который позволяет настраивать правила извлечения данных из хранилищ внешних систем в самой онтологии. Логическая витрина данных представляет собой адаптер MDM, который можно добавить в ее конфигурацию командой mdmctl add storage ldm (внимание: данный адаптер не поставляется в базовой версии продукта АрхиГраф.MDM и должен быть приобретен отдельно). После этого нужно импортировать в онтологию метамодель для описания правил мэппинга – файл с метамоделью поставляется вместе с адаптером. После этого нужно настроить правила соответствия элементов модели предметной области элементам структуры данных систем-источников.

Структурированные данные систем-источников могут быть представлены в виде таблиц реляционных баз данных или элементов структуры документов XML, JSON. Далее в этом разделе мы ограничимся рассмотрением правил мэппинга элементов онтологической модели на структуру базы данных. Подобные принципы применяются и для составления правил мэппинга онтологической модели с тегами XML, элементами коллекций JSON.

Задача установления соответствия между структурами данных в системах-источниках и онтологией предметной области состоит в описании правил преобразования данных из информационного пространства систем-источников в пространство онтологии. Для этого нужно установить следующие соответствия:

- между таблицами СУБД источников и классами онтологической модели;
- между столбцами таблиц СУБД источников и свойствами-литералами и свойствами-связями онтологической модели;
- между записями таблиц СУБД и индивидуальными объектами онтологической модели, представляющими информацию об элементах нормативно-справочной информации: например, элементами справочника единиц измерения и т.д.;
- кроме того, необходимо задать правила присвоения идентификаторов индивидуальным объектам, извлекаемым из систем-источников.

Для представления правил всех указанных видов разработана метамодель, которая включает следующие классы верхнего уровня:

- Автоматизированная система – идентифицирует систему-источник данных;
- Правило соответствия класса – устанавливает соответствие между классом модели и таблицей СУБД системы-источника;
- Правило соответствия атрибута – устанавливает соответствие между свойством модели и столбцом СУБД системы-источника;
- Правило соответствия таблицы-связи атрибуту – используется в случаях, когда таблица-связь СУБД проецируется на атрибут-связь онтологии;
- Правило идентификации объекта – определяет способ присвоения идентификаторов объектам, извлекаемым из систем-источников;
- Соответствие идентификаторов – определяет таблицу однозначного соответствия идентификаторов элементов НСИ системы-источника и онтологии.

Каждый класс может содержать ряд подклассов, выделяемых как на основании разновидности правила (например, способа присвоения идентификаторов), так и на основании применимости правил к определенной системе-источнику и/или типу сущности. В двух последних случаях подклассы выделяются только для удобства навигации по модели, специфичные атрибуты к этим классам не привязываются.

Далее подробно рассматриваются все виды правил и требования к их интерпретации в программном обеспечении логической витрины данных.

Соответствие классов таблицам базы данных. Каждый объект этого класса представляет правило установления соответствия между классом онтологии и таблицей системы-источника. Объекты класса могут обладать значениями следующих атрибутов:

Таблица 3. Структура описания правила мэппинга класса

Идентификатор	Название	Описание	Пример
СистемаИсточник	Система-источник	Система-источник	Система 1
ТаблицаВСистеме Источнике	Таблица или сущность в системе-источнике	Таблица в СУБД системы-источника	Devices
КлассМодели	Класс модели	Класс модели онтологии, которому соответствуют записи таблицы	Sensor
УсловиеНаВыборку ИзТаблицы	Условие на выборку из таблицы	SQL-условие на выборку записей из таблицы (для таблиц, содержащих объекты разных классов	Del=0

В случае, если система-источник является сервисом, в качестве значения атрибута `ТаблицаВСистемеИсточнике` задается идентификатор тега XML или другого элемента структуры данных.

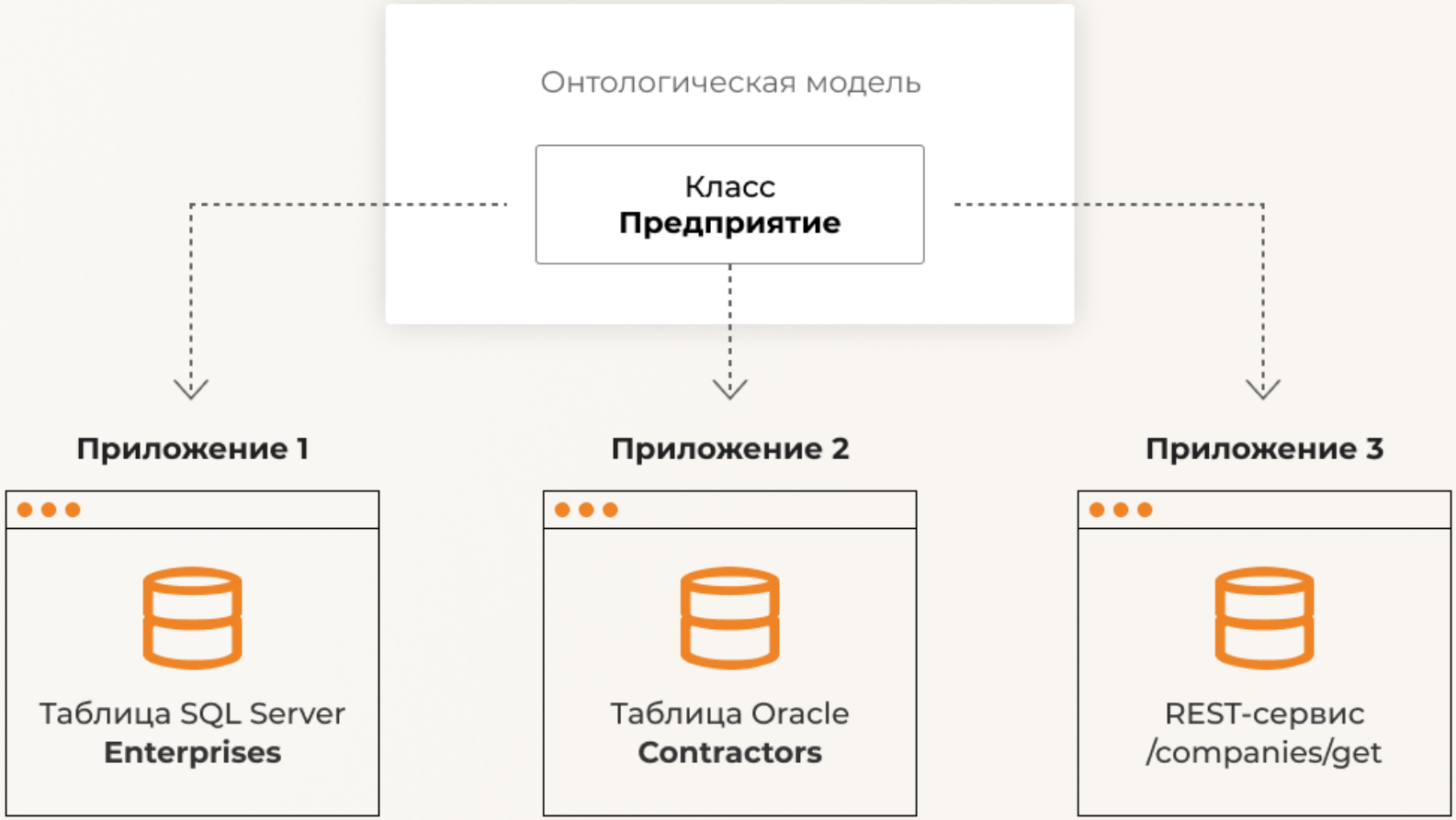
Реквизиты подключения к СУБД системы-источника и имя базы данных зафиксированы в объекте класса `СистемаИсточник`.

Возможны ситуации, когда одной таблице в системе-источнике соответствует несколько классов модели. В таких ситуациях одна и та же таблица будет соответствовать двум классам модели, для чего будет создано два правила. Далее одни столбцы таблицы могут быть поставлены в соответствие атрибутам объектов одного класса, другие – другого.

Возможна и ситуация, когда одному классу модели соответствует несколько таблиц в СУБД системы-источника. Для мэппинга объектов на таблицы этой системы должны быть созданы два правила.

Пример мэппинга классов и таблиц показан на рис. 24.

Рис. 24. Пример мэппинга классов на таблицы системы-источника данных



Соответствие свойств онтологии столбцам таблиц базы данных. Каждый объект этого класса представляет правило установления соответствия между свойством онтологии и столбцом таблицы системы-источника. Объекты класса могут обладать значениями следующих атрибутов:

Таблица 4. Структура описания правила мэппинга свойств

Идентификатор	Название	Описание	Пример
СистемаИсточник	Система-источник	Система-источник	Система 1
АтрибутВСистеме Источнике	Атрибут в системе-источнике	Имя столбца в СУБД системы-источника	Vat_number
АтрибутМодели	Атрибут модели	Атрибут онтологии, которому соответствуют значения из столбца	ИНН
Регулярное Выражение	Регулярное выражение	Регулярное выражение для получения значения из данных, извлеченных из системы-источника	(\d+)

Регулярное выражение используется в случае, когда необходимо провести дополнительную обработку значения после извлечения из системы-источника.

Тип извлекаемого значения определяется в соответствии с диапазоном значений свойства, указанного как `АтрибутВСистемеИсточнике`. Диапазоном значений может являться как XSD-тип для свойств-литералов, так и объект другого класса для свойств-указателей.

Пример мэппинга свойств показан на рис. 25.

В случае, когда одна таблица системы-источника соответствует разным сущностям онтологической модели, для каждого столбца могут быть созданы правила мэппинга на разные атрибуты в зависимости от рассматриваемого класса.

Рис. 25. Пример мэппинга свойств на столбцы таблиц системы-источника данных

Соответствие свойств-указателей таблицам связей. При проектировании реляционных СУБД для выражения связей между объектами часто создаются промежуточные таблицы, имеющие два внешних ключа на две связываемые таблицы. Если в онтологической модели соответствующая связь представлена свойством-указателем (это возможно благодаря тому, что в онтологической модели каждое свойство каждого объекта может иметь любое число значений, а не одно, как в реляционной СУБД), применяется описываемый специальный вид правила сопоставления. Объекты данного класса могут обладать значениями следующих атрибутов:

Таблица 5. Структура описания правила мэппинга таблицы-связи на свойство-указатель

Идентификатор	Название	Описание	Пример
СистемаИсточник	Система-источник	Система-источник	Система 1
ТаблицаВСистемеИсточнике	Таблица или сущность в системе-источнике	Таблица в СУБД системы-источника	Eqip_installation
КлассМодели	Класс модели	Класс онтологии, которому соответствуют записи таблицы	УстановкаОборудованияНаТехническомМесте
АтрибутМодели	Атрибут модели	Атрибут онтологии, которому соответствуют значения из столбца	УстановленоНаТМ
ССылкаНаОбъектИсточникСвязи	Ссылка на объект-источник связи	Имя столбца в таблице, содержащего идентификатор объекта-источника связи	InstallationID
ССылкаНаОбъектПриемникСвязи	Ссылка на объект-приемник связи	Имя столбца в таблице, содержащего идентификатор объекта-приемника связи	TechPlace

В приведенном здесь примере таблица-связка Equip_installation, используемая в Системе 1 для связи между единицами оборудования и техническими местами, на которых они установлены, проецируется на атрибут `УстановленоНаТМ` объектов класса `УстановкаОборудованияНаТехническомМесте`.

Важно отметить, что связи в онтологической модели являются направленными, а в СУБД направление связи при использовании связующей таблицы явно не задается. Поэтому значения атрибутов «Ссылка на объект-источник связи» и «Ссылка на объект-приемник связи» должны задаваться в соответствии с тем, какую область применения и диапазон значений имеет атрибут модели, которому соответствует связующая таблица.

Мэппинг элементов справочников и перечислений в составе НСИ. Для элементов перечислений и объектов из каталогов, входящих в состав НСИ онтологии, мэппинг может устанавливаться на уровне соответствия идентификаторов конкретных объектов. Примером может служить мэппинг кодов регионов РФ.

Предположим, что в онтологию загружен перечень регионов РФ с численными кодами. Далее для каждого региона должен быть задан его идентификатор в каждой из систем-источников (при этом должны быть задана также правила соответствия таблиц и атрибутов для класса `Регион`).

Объекты класса, описывающего соответствие идентификаторов, могут обладать следующими атрибутами.

Важно отметить, что одному объекту в НСИ консолидированной модели может соответствовать несколько объектов в системе-источнике.

Таблица 6. Структура таблиц мэппинга идентификаторов конкретных объектов в системах-источниках и онтологии

Идентификатор	Название	Описание	Пример
КлассМодели	Класс модели	Класс модели, к которому принадлежат сопоставляемые объекты	Регион
КодВСистемеИсточнике	Код в системе-источнике	Код объекта в системе-источнике	66
ОбъектНси	Объект НСИ	Идентификатор объекта в консолидированной системе	СвердловскаяОбл
СистемаИсточник	Система-источник		Система 1

Этот же механизм соответствия используется для встроенных перечислений, когда какие-то справочники запрограммированы в самой системе, и никакой таблицы для их хранения нет вообще.

Правила идентификации объектов по ключевым атрибутам. Для индивидуальных объектов, которые не имеют соответствующих элементов в классах НСИ онтологии и формируются только при извлечении данных из систем-источников по запросу, необходимо определить правила присвоения им глобальных идентификаторов. Такие правила, в том числе, позволяют объединять информацию об одних и тех же объектах, извлеченную из разных систем-источников.

Для отражения таких правил в метамодели создан класс `ПравилоИдентификацииОбъектаПоКлючевымАтрибутам`, объекты которого имеют следующие атрибуты:

Таблица 7. Структура правил присвоения идентификаторов объектам, извлекаемым из систем-источников

Идентификатор	Название	Описание	Пример
КлассМодели	Класс модели	Класс модели, к которому принадлежат идентифицируемые объекты	Персона
КлючевойАтрибут	Ключевой атрибут	Атрибут, значение которого используется для формирования «естественного ключа» объекта	rdf:type ИНН

Обратим внимание на то, что эти правила не имеют ссылки на систему-источник, т.к. правила присвоения глобальных идентификаторов должны быть одинаковыми независимо от системы источника.

В таблице 7 показан пример задания правила генерации идентификатора для объектов класса `Персона`. Идентификатор создается из идентификатора класса (значение предиката `rdf:type`) и значения свойства `ИНН`. В результате может быть получен, например, такой идентификатор: `Персона_667100000000`.