Министерство науки и высшего образования Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

В. А. Онуфриев

ПРОЕКТИРОВАНИЕ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ УПРАВЛЕНИЯ

Рекомендовано федеральным учебно-методическим объединением в системе высшего образования по укрупненным группам специальностей и направлений подготовки 27.00.00 «Управление в технических системах» в качестве учебного пособия при реализации основных профессиональных образовательных программ подготовки бакалавров по направлению подготовки 27.03.04 «Управление в технических системах»

ПОЛИТЕХ-ПРЕСС
Санкт-Петербургский
политехнический университет
Петра Великого

Санкт-Петербург
2020

УДК 004.82(075.8)
ББК 32.813
О58

Р е ц е н з е н т ы:
Доктор технических наук, заместитель директора СПИИРАН
по научной работе *С. В. Кулешов*
Кандидат технических наук, старший научный сотрудник
Санкт-Петербургского филиала ПАО «НПО «Стрела» *А. В. Кваснов*

Онуфриев В. А. **Проектирование интеллектуальных систем управления** : учеб. пособие / В. А. Онуфриев. – СПб. : ПОЛИТЕХ-ПРЕСС, 2020. – 132 с.

Основное внимание уделено системам на основе знаний и их применению, а также инженерии знаний как средству работы со знаниями, которые должны лечь в основу интеллектуальных систем.

Пособие предназначено для студентов, изучающих курс «Проектирование интеллектуальных систем управления», включенный в учебные планы по направлениям подготовки 27.03.04 «Управление в технических системах», а также 02.03.02 «Фундаментальная информатика и информационные технологии».

Пособие может быть использовано не только в рамках изучаемых курсов, но и при выполнении научно-исследовательских работ.

Печатается по решению
Совета по издательской деятельности Ученого совета
Санкт-Петербургского политехнического университета Петра Великого.

ISBN 978-5-7422-6903-8

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В наше время пользуется популярностью тренд интеллектуализации: многие потребители и производители задумались над модернизацией своего быта и / или производства путем введения некоторых интеллектуальных систем.

Однако, термином “интеллектуальность” необходимо пользоваться с большой осторожностью и ответственностью, так как не все, называемое интеллектуальным, является таковым по факту, ведь, как и со всем остальным, происходит упрощение понятий и размывание их границ. Любую нейронную сеть стремятся назвать интеллектуальной, любой цифровой датчик – интеллектуальным, любое устройство в составе “Умного дома” – интеллектуальным (как и саму систему управления “Умным домом”).

Однако, при более близком рассмотрении можно легко убедиться, что в большинстве случаев употребление термина “интеллектуальный” является необдуманным и необоснованным, так как данное понятие неразрывно связано с работой в условиях неопределенности: когда нет ни готового (заранее определенного) алгоритма, ни всей полноты данных о ситуации, ни четкого понимания результата, к которому приведут те или иные из возможных действий. Если система в процессе своей работы не производит заранее непредзаданных экспериментов, генерируя новый опыт, если не анализирует полученный опыт с целью выявления новых закономерностей, если не осуществляет поиск недостающей информации, то такая система не имеет никакого отношения к интеллектуальным.

В связи со сказанным, правда, легко понять, каким трудным, наукоемким должен быть путь проектирования и реализации любой интеллектуальной системы. Это – многоэтапный процесс, в котором применяются различные математические, логические, программные средства.

В рамках данного учебного пособия мы рассмотрим некоторые из них. Большая его часть посвящена базам знаний и, соответственно, инженерии знаний как совокупности средств для выполнения различных операций с ними.

В первой части рассматривается само понятие знаний и наиболее важные формы их представления, а также некоторые вопросы их получения (извлечение, приобретение, формирование).

Во второй части освещаются вопросы структурирования знаний – это чаще всего необходимый для конвертирования знаний в код этап. Здесь же рассматривается такой вид структуризации, как представление знаний в виде правил, написанных на различных языках логики: высказываний, первого порядка и дескрипционной. Показана также реализация ручной и программной структуризации на примере таких проектов, как Wikipedia и DBpedia.

Далее, в третьей части, мы подходим к программной реализации баз знаний – через их формализацию, то есть, перевод на предназначенные для этого языки. В том числе приводятся в соответствие языки логик (первого порядка и дескрипционной) и средства языков представления знаний.

В завершающей, четвертой, части рассматривается концепция интеллектуальных систем на основе знаний – комплексный программный продукт, позволяющий разрешать неопределенности, включающий для этого в себя множество разнородных компонентов, которые, однако, должны быть синхронизированы в своей работе.

При чтении данного пособия рекомендуем рассматривать все материалы как средство для последующей программной реализации системы на основе знаний в виде программного продукта. В особенности это касается материалов, не связанных непосредственно с формализацией знаний.

ЧАСТЬ 1. ЗНАНИЯ И ПРЕДВАРИТЕЛЬНАЯ ИХ ПОДГОТОВКА

Интеллект и разумность

Термин “искусственный интеллект” давно стал настолько привычным не только в вопросах, связанных с вычислительной техникой и автоматизацией, но и в повседневной жизни, что о нем уже никто не задумывается. Между тем, он происходит от англ. термина “Artificial intelligence”, который переводится “искусственная разумность”, что есть совсем не то же самое, что интеллект.

Разумность определим как умение производить логические операции (вывод, обобщение и другие) в соответствии с основными законами формальной логики – то есть, как умение мыслить.

Интеллект определим через способность решать задачи в условиях недостаточности или неопределенности исходных данных, а также способность решать задачи заранее неизвестного типа.

Если проанализировать задачи, решаемые при помощи “искусственного интеллекта”, изучить саму их постановку и способы решения, то станет понятно, что имеется в виду именно разумность. Например, в задачах распознавания объектов или в задачах машинного зрения вообще, речь обычно идет о заранее известных объектах или об объектах с заранее определенным набором признаков.

Все задачи “интеллектуального” поиска также всегда имеют четкую, заранее определенную постановку задачи и фиксированные методы ее решения, сводящиеся пусть и к порой сложным, но предзаданным алгоритмам.

Значительно ближе к задачам с применением интеллекта находится прохождение теста Тьюринга, если делать упор не на стремление компьютерного агента быть неотличимым от человека (например, совершая свойственные последнему ошибки), а на возможность обсуждения любой заранее не заданной темы с демонстрацией ее понимания на уровне анализа и

синтеза в контексте последней, обработки данных заранее неизвестной структуры.

Достаточно немного поразмышлять на тему отличия этой задачи от предыдущих, чтобы понять, насколько последняя превосходит их по сложности. Она сложнее даже на уровне ее принципиального решения: нетрудно составить схему решения задачи распознавания объектов (<обнаружить неопознанный объект>–<запросить список необходимых и достаточных для распознания признаков>–<последовательно сравнить имеющиеся в базе признаки с признаками объекта>–<оценить уровень совпадения, определить наиболее выраженные>–<сделать вывод>), в то время как даже обладание полным списком возможных вопросов и возможных ответов по любой теме (что уже само по себе представляется сомнительным) все равно никак не будет способствовать “умению” машины анализировать и синтезировать произвольную информацию.

Другим примером задачи с применением интеллекта являются задания на определение коэффициента интеллекта – например, тест Айзенка (определение уровня IQ). Мы рекомендуем всем, кто никогда не проходил через такой тест, сделать это, держа в голове задачу создания такого программного агента, который бы смог, заранее не зная этих заданий, набрать максимальное количество баллов по результатам.

Более того, важно провести наблюдение за Вашим собственным процессом решения указанных задач. Как Вы это делаете и что Вы для этого используете? На *любой* из вопросов подобных тестов Вы сможете сформулировать именно логический ответ, который будет являться следствием проведенных логических операций – то есть, процесса мышления. В этой связи этот процесс следует рассмотреть подробнее.

В качестве примера используем распространенную в таких тестах задачу (Рис. 1).

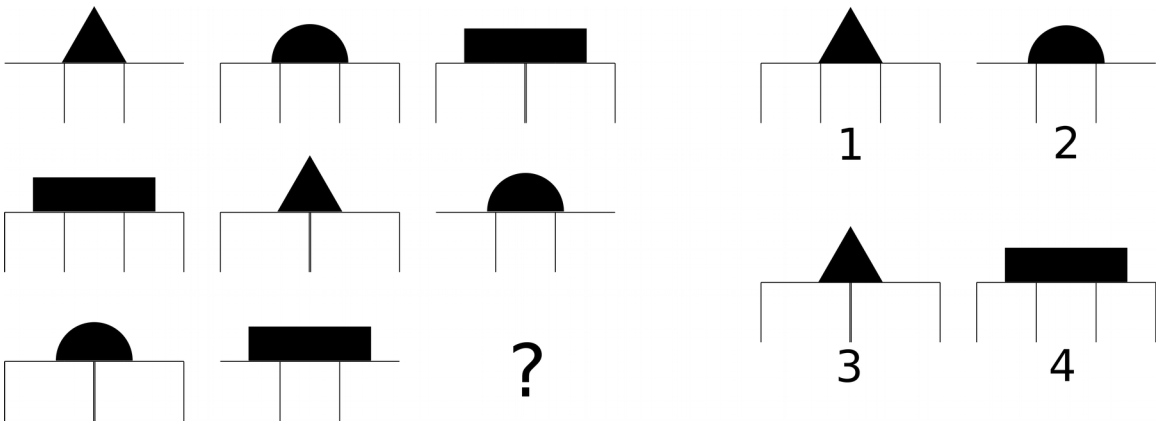

Рис. 1. Пример задачи

Самое первое, что можно подчеркнуть в ее решении – это тип задачи. Здесь мы имеет дело с задачей про отсутствие, про нехватку элемента. Что нам сказало об этом? Факт наличия вопросительного знака в самом теле задачи.

Так, наше мышление здесь начинается с суждения:

“?” указывает на Задачи на отсутствие

Суждение – законченная мысль, в которой один упоминаемый элемент раскрывается через другой. Иными словами, в нем обязательно содержатся подлежащее и сказуемое (в англоязычной литературе – субъект и предикат). В нашем примере, субъектом является “?”, предикатом – “указывает на”, а объектом (говоримым) – “Задачи на отсутствие”.

Обращаем внимание, что именно суждение, а не любая мысль, является элементарной единицей мышления – процесса логического вывода.

Что дальше?

Если у Вас уже есть опыт решения задач такого типа, то Вы понимаете, что они бывают графическими, математическими, лингвистическими – и прочими.

Перед нами – графическая задача, где решение зависит от расположения картинок (пропустим шаги в рассуждении о том, как мы классифицировали ее таким образом). Наше суждение:

“**Рассматриваемая задача является графической”**.

Кроме того, известно, что

Любая графическая задача имеет повторяющиеся элементы либо их повторяющиеся свойства, которые необходимо определить.

Последнее высказывание является не только суждением, но еще и **правилом** ввиду своей всеобщности, показанной квантором всеобщности “любая”.

Теперь, соединяя суждение “Рассматриваемая задача является графической” с правилом “Любая графическая задача имеет повторяющиеся элементы либо их повторяющиеся свойства”, делаем вывод, что

Рассматриваемая задача имеет повторяющиеся элементы либо их повторяющиеся свойства, которые необходимо определить.

Таким образом, из правила (о графических задачах) и из суждения (о конкретной задаче, относящейся как раз к классу графических) мы сделали логический вывод о том, что именно в ней нужно искать.

Через несколько шагов подобных рассуждений мы придем к ее решению. Что нам для этого понадобилось? Знание правил (суждений с элементом всеобщности) и умение применять эти правила к конкретной ситуации, для чего было бы необходимо, например, уметь распознавать факт повторяемости элемента на рисунке.

Могли ли бы мы решить данную задачу, не обладая обозначенным выше правилом? Вспомните, как Вы решали такую задачу впервые в жизни, еще не зная о повторяемости. Какое другое правило Вы при этом знали? Правило о том, что такие задачи определенно содержат в себе некоторую закономерность. Более того, у Вас уже был в голове в неявном виде список возможных закономерностей, так как увидев их, Вы бы их сразу распознали:

- повторяемость элементов;
- математические закономерности (постоянное возрастание, изменение чисел по закону…)
- лингвистические закономерности – повторение буквы, слов, изменение букв по закону, например: а, в, д, ё, з… и другие.

Как только бы Вы увидели любую из них, Вы бы попробовали каким-то образом ее применить к данной задаче и в конце концов бы обнаружили, что для повторения закономерности на место “?” нужно поставить строго определенный элемент, а для этого Вы должны бы были заранее предположить, что решение задачи состоит в сохранении закономерности.

Таким образом мы приходим к тому, что в основании решения человеком какой-либо задачи лежит либо знание определенных правил, либо умения эти правила предполагать и проверять (что уже говорит о его обладании правилами проверки и опровержения правил). **Ничто не говорит о принципиальной невозможности придания этих особенностей программным агентам**.

Подводя итоги, подчеркнем, что

- знание правил,
- умение выносить суждения,
- умение применять правила к суждениям

и лежат в основе принятия решений в задачах, требующих применения интеллекта, но они не требуются в не связанных с интеллектом задачах, например, при распознавании изображений (по крайней мере, в явном виде). Нужно четко разделять задачи, связанные с применением элементов искусственного интеллекта, и несвязанные с ним, так как для их решения требуются разные подходы и инструменты.

Таким образом, при проектировании интеллектуальных систем нашей задачей станет реализация трех указанных возможностей в программных агентах.

Однако, предварительно следует разобраться в еще одном важном терминологическом вопросе.

Информация, данные и знания

Для продолжения работы, чтобы точно понимать объект нашей работы, нам потребуется разобраться с такими понятиями как информация, данные и знания.

А перед этим – с самим процессом определения понятий. Вам часто требуется воспроизводить те или иные определения, поэтому нужно четко представлять, как это делать правильно.

Есть несколько способов определения понятий. Мы будем пользоваться, в основном, одним: через **родовое понятие и видовое отличие** (по-другому это называется "**интенсионал**"). При таком способе необходимо выделить ближайшее родовое понятие и минимальный набор признаков, отличающий данное понятие от других из этого же рода.

Родовое понятие – это термин, обозначающий более общее понятие, чем определяемое. Например, у понятия "стол" ближайшим родовым понятием можно назвать "мебель", у "кошки" – "млекопитающее" и т.д. Процесс перехода от понятия к его родовому понятию называется **обобщением** (генерализацией).

Видовое отличие можно выделять более, чем одним способом, в этой связи данное Вами определение может отличаться от словарного, но это само по себе еще не означает его ошибочности.

Теперь нужно определить понятия "информация", "данные" и "знания", так как мы далее будем концентрироваться на знаниях, а для этого нужно четко понимать, что это такое.

Какое из трех понятий является наиболее общим: включающим в себя все остальные? Таковым является **информация** – любые сведения и факты, которые могут быть восприняты и обработаны человеком либо компьютером. Очевидно, что и знания, и данные входят в это понятие. Обратите внимание на то, как трудно предложить родовое понятие для "информации". В этой связи этот термин остается только "схватывать".

Определение, данное выше, нельзя назвать определением через родовое понятие. Напротив, данное определение представлено в форме примеров (более узких, входящих в него понятий). Это равнозначно определению типа "птица – это воробей, гусь, страус". Определения такого типа называются

“**экстенсионал**”, и они имеют смысл только для таких объемных понятий как “информация”, когда нельзя дать интенсионал.

Обратите внимание, что в определении понятия “информация” ничего не сказано о форме представления этих сведений. Именно это позволит различить понятия информации и данных: **данные** – информация, представленная в структурированной форме. При этом форма их представления может быть графической, табличной, текстовой, звуковой и т.д., главное – наличие порядка, четко сформулированного и формализованного. Критерием того, что мы имеем дело с данными, не может служить факт хранения ее на компьютере.

Знания же выступают наименее широким понятием из перечисленных. Они обязательно являются структурированными, поэтому они – данные. Но не любые данные можно назвать знаниями. **Знания** – это представленные в понятном для человека либо программного агента данные о том, как пользоваться, применять и обрабатывать другие данные. Иначе говоря, это данные о способах работы с данными. Метаданные.

Данные и знания довольно легко различить: если мы видим таблицу с информацией об изменении курса валюты или же о показателях датчика температуры в реакции, то это данные. А вот информация о том, что (к примеру)

если эти первые данные, построенные в виде графика,
формируют собой определенную фигуру (“голову и плечи”),
то это говорит о скором будущем стремительном
падении курса валюты

– это уже знания.

И уже на этом этапе возникает вопрос: а как может применять эти знания машина? Ведь даже человек, имея четко сформулированное правило, вполне может испытывать трудности с его применением.

А для этого нужно понять, в какой форме могут существовать знания и по каким критериям их можно отличить от прочих данных. Не менее важно

определить, в какой форме они должны быть представлены для возможности их применения компьютерными системами.

Нейроинформатика и инженерия знаний

В середине XX века, когда возникла и стала реализовываться идея создания искусственного разума, ученые сформировали два пути развития этого направления. Эти две ветви отличались по своим базовым предпосылкам, хотя и имели общую цель.

В первом случае исследователи опирались на суждение о том, что в настоящее время существует лишь один известный носитель разумности – человеческий мозг. В этой связи, для того, чтобы воспроизвести его функциональность машинными средствами, нужно воспроизвести, прежде всего, его структуру. Таким образом, как легко понять, стала развиваться нейроинформатика, в рамках которой строятся искусственные нейронные сети.

Во втором случае исследователи обращались не к структуре человеческого мозга, а к содержанию самого мышления: к содержащимся в нем предпосылкам и правилам, к суждениям и фактам. Своей задачей они видели моделирование самих процессов, чтобы машина могла воспроизводить сам сознательный процесс мышления. При этом свой отход от биологических структур они аргументировали тем, что человеку уже удалось создать не существовавшие в природе механизмы (например, самолеты), способные в той или иной мере повторять или даже превосходить аналоги из природы.

Несколько десятков лет две данные линии разработки шли параллельно, однако каждая из них имела свои собственные недостатки, которые не позволяли в должной мере решать все более сложные задачи, которые перед ними ставили.

Коротко рассмотрим самые базовые принципы нейронных сетей, чтобы далее лучше понимать отличия между двумя указанными подходами к разработке интеллектуальных систем.

В основе нейронной сети, ее элементарным элементом, является искусственный нейрон – математическая модель биологического нейрона. Данная модель имеет один или несколько входных параметров (соответствующих коротким отросткам нейрона – дендритам) и один выходной параметр (соответствующий длинному отростку – аксону). Сигналы, дошедшие к нейрону через дендриты, суммируются, определенным образом обрабатываются нейроном и передаются на аксон.

В этой связи сам искусственный нейрон представляет собой ни что иное, как математическую функцию зависимости одного выходного параметра от суммы входных. Сама же такая функция называется функцией активации.

Соединяясь, нейроны формируют сеть. Искусственная нейронная сеть, таким образом, представляется в виде графа, состоящего из вершин (нейронов) и дуг (входных и выходных отростков).

Место соединения двух нейронов (“выхода” одного со “входом” другого) называется синапсом, и это сочленение на химическом уровне регулирует коэффициент усиления проводимого сигнала.

В математической интерпретации это означает наделение каждой из дуг графа собственным весом, что будет означать, что проходящий по ней сигнал будет умножаться на значение ее веса, тогда к следующему нейрону сигнал придет уже взвешенным.

Рассмотрим Рис. 2, показывающий часть искусственной нейронной сети.

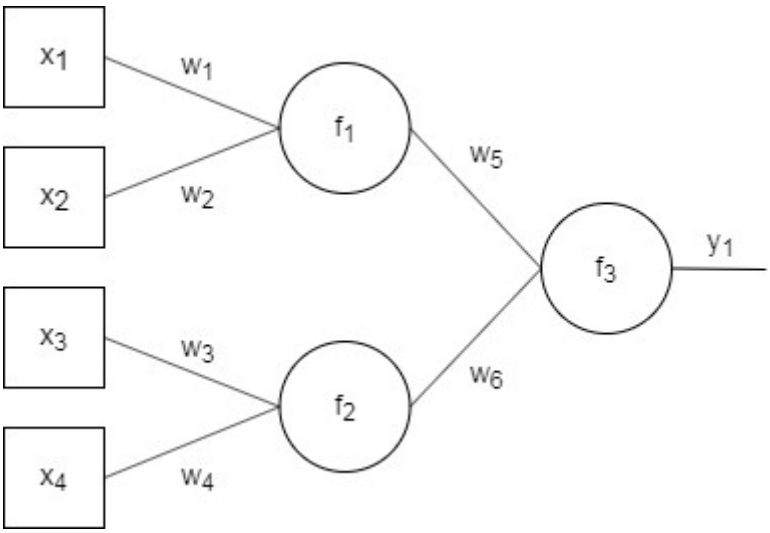

Рис. 2. Часть искусственной нейронной сети

Таким образом, выражение зависимости выхода данной выхода нейронной от ее входов имеет следующий вид:

(1)

Далее обычно говорят о том, что нейронную сеть нужно обучать, изменяя веса ее графа таким образом, чтобы любую комбинацию сигналов на входах она обрабатывала правильным образом.

Получается, что вся интеллектуальность нейронных сетей опирается на набор всех ее весов. А так как про нейронные сети можно сказать, что они “умеют” правильным образом обрабатывать данные и делать на их основе выводы, то можно говорить уже о **знаниях**, которыми нейронная сеть обладает.

Вместе с тем, эти знания представлены в нейронных сетях **неявно**. Это вызывает сложную и не всегда имеющую решение проблему перевода знаний из формы в виде набора матриц весов в интерпретируемую человеком форму. Например: какие знания можно извлечь из выражения (1), если известны численные значения всех представленных весов? А ведь это и есть знания, представленные в нейросетевой форме.

Более того, при изменении числа входов и/или выходов сеть будет нуждаться в переобучении, а представленные в ней знания станут недействительными.

Кроме того, для обучения действительно сложной нейронной сети (когда ее количество слоев измеряется десятками или даже сотнями) к вычислительному оборудованию предъявляются значительные требования, что в свое время заметно повлияло на энтузиазм и скорость исследований в данном направлении.

И это, несмотря на все неоспоримые положительные качества нейронных сетей, заставляет нас искать другие способы представления знаний и пойти по второму из представленных выше путей, когда нашей задачей является воспроизводство не формы мыслящего аппарата, а его содержания.

Тогда мы должны перейти от рассмотрения нейроинформатики более широкому разделу – к инженерии знаний.

Инженерия знаний – это раздел наук об искусственном интеллекте, исследующий вопросы, связанные со знаниями, а именно с их извлечением, структуризацией и формализацией. Это – три основные задачи данной дисциплины.

В результате выполнения этих трех задач знания представляются в явной форме, “понятной” для программных агентов, так как именно они являются конечным субъектом применения знаний. Необходимые для этого формы мы еще обсудим, а сейчас нужно сконцентрироваться на трудностях, которые стоят на пути работы с явно выраженными знаниями.

Основная форма хранения знаний – не численная, как в случае с нейронными сетями, а **символьная (текстовая)**. Это означает, что по умолчанию базы знаний – специальным образом организованные хранилища – не имеют встроенного математического аппарата. Из этого следует отсутствие по умолчанию способности к аппроксимации математических функций. В этой связи, процесс создания знаний в рамках таких хранилищ сильно отличается от создания знаний в рамках нейронной сети (то есть, ее обучения).

Базы знаний могут наполняться тремя способами: внесение уже имеющихся знаний, “вычисление” при помощи машины вывода и получение при помощи подключения различных анализаторов данных.

В первом случае уже известные знания (факты, правила) вносятся непосредственно в базу. Для этого должны быть последовательно пройдены уже упомянутые процедуры извлечения, структуризации и формализации знаний. Следует обратить внимание, что этот процесс поддается автоматизации, и этот вопрос мы рассмотрим отдельно.

Во втором случае речь идет об использовании логики (например, логики предикатов, дескрипционной логики). Тогда новые знания получаются на основе имеющихся путем применения взятых из базы правил к содержащимся в ней же фактам. То есть, в простых случаях мы имеем дело с **дедуктивным выводом** – вывод конкретных (частных) фактов на основании более общих правил. Сюда же можно отнести и поиск определенных закономерностей среди имеющихся знаний. Это есть движение от общего к частному, для чего используются различные **машины вывода** – программы, выполняющие дедуктивный вывод знаний (фактов и правил).

Третий случай является наиболее сложным и наукоемким и здесь есть две возможности: производить **индуктивный вывод** (от частного к общему) на основании имеющихся знаний в базе или на основании произвольного набора данных.

Для реализации первой возможности нужны специальные программы-анализаторы, исходящие из так называемого “**предположения об открытости мира**”, которое означает, что все, что в явном виде не прописано в базе знаний, тем не менее, может существовать (например, если база не содержит информации о существовании бессмертных существ, то их существование является возможным).

Для анализа же произвольного набора математических данных, потребуется использовать отдельные модули (в том числе, нейросетевые), так как сами по себе базы знаний “не умеют” самостоятельно извлекать математические закономерности, так как их собственное строение и логика работы не являются строго математическими.

Стоит подчеркнуть, что именно реализация возможностей дедуктивного и индуктивного выводов является ключом к созданию интеллектуальных систем на основе знаний.

Однако, для реализации этого сначала необходимо осуществить первичное наполнение базы знаниями, на основании которых затем уже можно будет реализовывать вывод. А для перехода к этой задаче необходимо предварительно рассмотреть собственно внутреннюю организацию базы знаний и базовые виды ее содержаний.

Классы, индивиды и свойства

База знаний, как и нейронная сеть, представляется в виде графа с вершинами и дугами. Однако, в отличие от нейронных сетей, база знаний не является математической моделью с входами и выходами. Содержание ее составляют некие элементы, представленные в текстовой форме, и связи (отношения) между ними – также текстовые.

Когда мы говорим о базах знаний, то мы говорим о представлении знаний в виде символов. И на этом пути существуют определенные трудности, которые придется как-то разрешать.

Изучением символов и их значений занимается наука **семиотика**. Нас же будут интересовать только некоторые аспекты, в число которых входит показанный ниже (Рис. 3) **треугольник семиотики (треугольник Фреге)**, представленный почти сто лет назад, но используемый и сейчас.

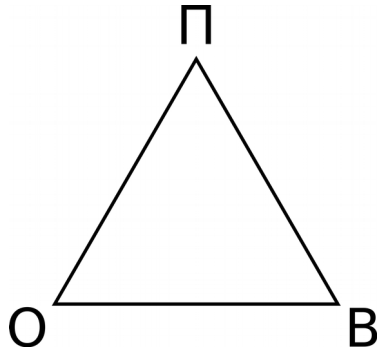

Рис. 3. Треугольник семиотики

Внимательно изучите изображение: «П» означает «понятие», «О» – «обозначение» (символ), а «В» – обозначаемая вещь. Сейчас необходимо понять связь этой конструкции с нашей дальнейшей работой.

Непосредственное содержание базы знаний представляется символами, однако, задачей баз знаний является описание **понятий** и **экземпляров (представителей) данных понятий**, стоящих за этими символами. В треугольнике семиотики им выделены соответственно верхняя (понятия) и правая (обозначаемые ими конкретные вещи) части.

Здесь для аналогии лучше всего упомянуть объектно-ориентированное программирование с его двумя весьма схожими терминами: **Класс** и **Экземпляр класса (индивид)**.

База знаний, помимо классов и их экземпляров (индивидов), обязательно содержит также и **свойства** – то есть, характеристики, которые могут относиться к классам и экземплярам (чаще – к последним). Свойства имеют имя (например, «масса» или «требует наличия») и значение (например, «500 кг» или «стальной трос»).

Класс мы будем рассматривать одновременно в двух аспектах: как понятие (Рис. 3), определенное описанным ранее способом через родовое понятие и видовое отличие, и как множество (совокупность) объектов, имеющих определенный общий собирающий признак.

В первом случае класс определяется через другой, более широкий (общий) класс. Так, класс “Нештатные ситуации” может быть определен через класс “События” с указанием признаков, отличающих наступление таких ситуаций от прочих событий, например, плохая предсказуемость, нежелательные последствия и т.д. Тогда речь идет об установлении отношения “**подкласс**” (**“subclass of”**). Класс определен как подмножество (подкласс) большего множества (класса).

Во втором случае, класс определяется через набор признаков, которому удовлетворяют его индивиды, например: “Краны с грузоподъемностью от 5

тонн”. Конечно, здесь тоже возможно определение через родовое понятие (“Кран”), но в момент описания класса отношение “подкласс” не было установлено явным образом, вместо этого было введено **ограничение (Restriction)** на свойство “грузоподъемность”. Тогда класс определен как множество индивидов, удовлетворяющих поставленному ограничению.

Сами же **экземпляры классов** (их представители, индивиды, instances, individuals) определяются исключительно своей принадлежностью к одному или нескольким классам одновременно. Для этого используется отношение **“тип” (“type”)** – одно из самых распространенных в графе базы знаний.

Теперь можно говорить, что любая вершина данного графа представляет собой либо класс, либо индивид. Вершина имеет идентификатор, символьное имя, которое, следовательно, соответствует либо классу, либо экземпляру – и идентификатор, который подбирается создателем базы знаний, должен однозначно это определять. Ребра же такого графа представляют собой отношения между «вершинами» (например, упомянутые type, subclass of и другие).

Форма представления базы знаний, описанной в форме упомянутого графа, называется **семантической сетью**. Она не является единственной (наряду с формами фреймовых систем и продукционных правил), но именно ее мы будем рассматривать далее.

Для этого следует рассмотреть отличие семантических сетей от остальных сетей и ввести определенную терминологию.

Синтаксис, семантика, контекст

Как уже можно было догадаться, создание базы знаний, если говорить про сам процесс ее наполнения, сводится к описанию ее элементов и описанию отношений между ними – то есть, к описанию ее графа.

И в этот момент, непосредственно создавая эти описания и эти связи, мы оказываемся в слое **синтаксиса**. С точки зрения грамматики, синтаксис (греч. σύνταξις) – это про принципы конструирования, связывания предложений в

конкретном языке. С точки зрения формальных языков (в том числе и языков программирования), синтаксис – это множество правил формирования более сложных выражений из взятых за основу символов. С точки зрения же баз знаний нас будет интересовать синтаксис как правила описания и связывания классов и экземпляров.

Для того, чтобы говорить о правилах, необходимо определить источник правил. В случае с базами знаний правила определяются так называемыми **языками представления знаний** (ЯПЗ), которые, как и в случае с языками программирования, могут вызывать синтаксические ошибки во время проверки кода.

Стоит отметить, что слой синтаксиса является наиболее легко проверяемым на наличие ошибок, так как анализ синтаксиса – это процесс сравнения проверяемого набора символов с заранее четко определенным шаблоном нормативной структуры данных.

Но что, если необходимо проверить не только саму форму представления знаний, но и их содержание? В этот момент слой синтаксиса оказывается недостаточным.

Представьте, что в Вашу задачу входит проверка правильности работы гиперссылок в системе связанных между собой электронных документов (или страниц). В этом случае работа в слое синтаксиса подразумевает две проверки:

- является ли гиперссылка таковой, перенаправляет ли она пользователя при нажатии на нее;
- не указывает ли она на несуществующую страницу, не приводит ли переход по ней к ошибке “404” (“страница не существует”).

Однако, очевидно, что для проверки правильности гиперссылок еще требуется убедиться, что ссылка с текстом “промышленное оборудование” действительно ведет на страницу с описанием именно промышленного оборудования, а не популярных медицинских препаратов. И такого рода проверки уже не имеют тривиального решения.

Здесь уже начинается слой **семантики** (греч. σεμαντικος) – раздела науки о языке, который изучает значение символов, их смысловую нагрузку. В этой связи, упомянутая проверка ссылки “промышленное оборудование” находится как раз в семантическом слое.

Так, для реализации упомянутой проверки ссылки программный агент должен перейти по ней, определить заголовок страницы, выяснить его семантическое значение и соотнести с семантическим значением гиперссылки.

Рассматриваемая форма баз знаний – это множество именно семантических (смысловых) связей класс-класс, класс-экземпляр, экземпляр-экземпляр. Построение базы – это не просто процесс указания ссылок (как в проекте Wikipedia), а именно смысловое описание, то есть четкое указание на характер связей между элементами, например, на принадлежность экземпляра к классу (“квадрокоптер является беспилотным летательным аппаратом”) или на причинно-следственную связь между явлениями (“нарушение процесса обессоливания нефти приводит к снижению качества фракции бензина”) и т.д. Такая форма представления базы знаний называется **семантическими сетями**.

К задачам семантики также относится изучение вопроса о том, как значение более сложного понятия может быть без участия человека выведено из значений более простых понятий с учетом синтаксиса их связей (например, в случае определения через родовое понятие и отличительные признаки).

Однако, здесь возникает следующая, более сложная проблема. Некоторые слова являются многозначными, а некоторые – омонимами. В обоих случаях одно и то же слово имеет различные значения. Это создает определенные трудности, ведь даже если заранее известен полный список возможных значений конкретного понятия, то как определить, которое из них использовано в данной ситуации?

И таким образом мы поднимаемся из слоя семантики в следующий слой – **контекста** (лат. contextus). Контекст – это более широкое смысловое поле, это

про отношение конкретного элемента с окружающими его элементами, которые как раз и определяют интерпретацию понятий.

Возьмем понятие “котел”. В контексте (в окружении) туристических принадлежностей, это понятие будет означает небольшую емкость для приготовления пищи в походных условиях. В контексте же водогрейного оборудования и котельных помещений понятие “котел” будет означать устройство, резервуар, снабженный нагревательными элементами, применяемый в промышленности.

Так, именно окружение, контекст, дает понимание о том, какое содержание понятия должно использоваться в данных условиях.

При этом, стоит помнить, что любая ситуация и любое понятие могут рассматриваться сразу в нескольких контекстах разной степени общности, например, процесс обмена данными между отправителем и получателем можно рассматривать как в социальном контексте (с его правилами и условиями), так и в более общем контексте места и времени контакта.

Однако, контекст сообщения или понятия вообще может быть не указан или неизвестен. В этом случае для выбора подходящего значения понятия требуется иметь представление о следующем за контекстом слое – о **прагматике** (греч. πραγμα), которая изучает цели и стремления взаимодействующих агентов, которые определяют необходимые контексты и, в последствии, значения понятий. Иными словами, прагматика изучает связь субъектов, применяющих языковые средства, с контекстами и семантикой данных средств.

Теперь, рассмотрев основные аспекты, связанные со знаниями, перейдем к постановке задач, которые могут быть решены системами на основе знаний.

Задачи систем на основе знаний

Самая первая задача, которая ассоциирована с применением систем на основе знаний – это **поиск информации** (как в сети Интернет, так и в любых других источниках).

И необходимость развития этого направления обнаруживается при работе с Интернет-поиском. Вначале, когда Интернет только зарождался, когда он предполагал не путешествия по страницам и глобального поиска, а лишь помогал реализовывать поиск документов в хранилищах и обеспечивать к ним доступ, такой проблемы, конечно, не было. Но когда возникла и реализовалась идея Web 1.0, а затем и Web 2.0, то стала явной проблема поиска на основе ключевых слов в поисковиках.

Дело в том, что по умолчанию поисковые программы не понимают значений слов и “воспринимают” любую информацию на странице не более, чем комбинации символов с неизвестным содержанием, неопределенным контекстом и применяемые с неизвестной целью. Какие проблемы это влечет за собой?

В первую очередь речь идет о выдаче результатов, многие из которых не релевантны запросу. Причина состоит в том, что не учитывается контекст запроса, вследствие чего возникают проблемы с многозначными словами, когда в результаты попадают страницы, связанные с совсем другими значениями (художественные кисти вместо кистей рук). Эта проблема может решаться, например, знаниями о статистике, о наиболее распространенных контекстах, которые и будут определять требуемые значения слов.

Кроме того, могут выводиться страницы, где искомое слово просто фигурирует, не являясь предметом текста либо страницы. Для решения этой проблемы поисковый агент должен уметь анализировать количество и качество связей (отношений) этого слова с другими на этой странице.

С другой стороны, обычный поиск по ключевым словам выводит не все результаты, потому что не учитывает, например, поиска по синонимам и по другим названиям (например, при поиске информации по слову “Ягуар” стандартными средствами вы бы не увидели страницы, содержащие вместо поискового слова его научное название – “Pantera onca”). Сюда же относится и

учет метафорических выражений, принятых “в народе” названий и прочих контекстуальных определений.

Для решения этой задачи система на основе знаний обращается к уже структурированной базе знаний, в которой понятия должны быть сохранены уже с учетом контекстов, синонимов и прочих особенностей (именно для этого в том числе и создаются семантические связи между элементами).

Стоит отдельно отметить, что база знаний может и, скорее, даже должна быть распределенной, что означает, что конкретный модуль системы может обращаться не только к локальной базе, но и к удаленной.

В отличие от задачи извлечения знаний из уже готовых баз, задача **извлечения знаний из неструктурированных источников** является одной из наиболее трудоемких ввиду необходимости сначала вычленить контекст самого исходного материала, затем произвести анализ связей и значений содержащихся в источнике элементов – при этом содержание и порядок их представления заранее не известны, – и лишь после этого решать задачу перевода извлеченных знания в ожидаемую форму, которая при этом должна быть определена заранее.

Задача требует применения синтаксического или иной анализатора, производящий для базы знаний материалы для сохранения. При этом система должна постоянно обучать саму себя с применением имеющегося и постоянно получаемого нового опыта, добавляя новые знания (закономерности, правила) в базу.

Также системы на основе знаний могут выполнять **обучающую функцию**. На основе баз знаний могут быть созданы тренирующие и тестирующие программы, может создаваться подробная аналитика по усвоению материалов по темам, выявление правил и зависимостей.

Далее можно рассматривать задачи **поддержки информационной целостности** систем (maintenance). Под целостностью будем понимать согласованность (непротиворечивость) информации. С одной стороны, речь идет о проверке материалов на предмет содержания противоречащих друг другу

фактов либо правил, а с другой – об упомянутой ранее ссылочной синтаксической и семантической целостности. С ростом увеличения объема материалов эта задача становится все более сложной для “ручного контроля”.

Сюда же можно отнести и задачу **хранения корпоративных знаний**, которая актуальна благодаря прямой зависимости эффективности работы компаний от знаний их сотрудников, что всегда сопряжено с “человеческим фактором” (возможность увольнения из компании до успешной передачи накопленных знаний конкретным сотрудником и т.д.).

Следующую задачу системы на основе знаний можно обобщенно назвать **задачей классификации**. Процесс ее решения отличается от аналогичного с применением нейронных сетей, где обычно на вход подаются сырые данные с классифицируемого объекта либо сам объект (например, рисунок), а затем он классифицируется на основании неочевидных нейросетевых расчетов. Здесь же на вход должны подаваться уже сами признаки в определенной, формализованной форме. А так как математические операции (например, обработка рисунка с целью извлечения признаков) в базах знаний не предусматриваются, то для решения задачи нейросети (и прочие математические модули) должны производить предварительную обработку входных данных (например, выделяя по рисунку факты наличия/отсутствия определенного рода изгибов и / или их количественные характеристики), которые затем и подаются на вход системы на основе знаний.

Более общим случаем для задачи классификации является задача **оценки**. При этом оцениваться может способ или средство достижения цели с точки зрения их эффективности; может рассчитываться степень уверенности в наступлении того или иного события или сценария; может оцениваться и сам результат на предмет соответствия изначальной постановке задачи.

Близкой к этому также является задача **прогнозирования**, заключающаяся в том, чтобы, например, выбрать наиболее вероятный сценарий дальнейшего

развития событий на основании текущих показателей процесса, правил и накопленного практического опыта.

Задача **управления в технических системах**, подразумевающая поддержание назначенных целевых функций управления в определенных интервалах, также требует взаимодействия с другими подсистемами – например, с системой сопряжения с основным управляющим контуром и его объектом, куда передаются управляющие воздействия, выбранные системой на основе поступающих с датчиков технологических параметров, истории и т.д. Здесь управляющие сигналы могут быть также предварительно обработаны при помощи математических средств. Системы на основе знаний в рамках управления в технических системах способны решать неопределенности (что рассмотрим в четвертом разделе данного пособия).

В случае, если система изначально ориентирована на работу с пользователем, на консультирование и помощь в принятии решений, то ее можно отнести к классу **экспертных систем**. Причем здесь нужно заметить, что не любая консультирующая система может быть названа экспертной, так как для этого последняя должна обладать тремя обязательными компонентами:

- собственно базой знаний;
- машиной вывода (*reasoner* – решатель);
- пользовательский интерфейс с подсистемой пояснений.

В этой связи, к любой экспертной системе можно задать вопросы:

- На каком из языков представления знаний реализована база?
- По какому принципу работает и как реализована машина вывода?
- Как преобразуется выводимая пользователю информация для того, чтобы приобрести читаемую форму?

Однако, сердцем любой системы на основе знаний является база знаний, поэтому в рамках данной книги мы последовательно рассмотрим все задачи и шаги, связанные с подготовкой знаний, с программной реализацией их хранилищ и с их применением.

Для дальнейшего движения в данном направлении изучим теперь виды знаний и формы их представлений.

Классификация знаний

Знания могут выступать в трех основных формах: факты, терминологические аксиомы и аксиомы роли.

Фактом называется суждение, описывающее либо отношение между двумя индивидами, либо фиксирующее принадлежность индивида определенному классу. Отношение *r* между индивидами *a* и *b* будем обозначать как *r* <*a*,*b*> или *arb*, а принадлежность индивида *a* классу *C* – *a:C*. Совокупность всех фактов называют **системой фактов** и в международных терминах обозначают **ABox** (assertional box).

Терминологической аксиомой называют суждение, фиксирующее либо эквивалентность двух классов, либо включение одного класса в другой. В первом случае эквивалентность классов *C* и *D* обозначается $C \equiv D$, а во втором включение класса *C* в класс *D* обозначается $C \subseteq D$. Совокупность терминологических аксиом называют **терминологией концептов,** обозначают через **TBox** (terminological box).

Аксиомой роли называют суждение, указывающее либо на транзитивность некоторой роли, либо на эквивалентность одной роли другой или что она является ее подролью. В первом случае транзитивность роли *isMadeOf* обозначается аксиомой Tr(*isMadeOf*) [Золин, 2018], эквивалентность ролей *canCause* и *canResult* обозначается $canCause \equiv canResult$, аналогично записывается $products \subseteq results$. Совокупность аксиом роли обозначают через **RBox** (relational box).

Подробнее аксиомы рассмотрим в другом разделе пособия.

Здесь следует отдельно подчеркнуть значение термина “роль”. В суждении

roboticArm hasPart effector

элемент “hasPart” можно назвать ролью (role), свойством (property), предикатом (predicate), отношением (relation). Все эти термины, будучи схожими, все же не

являются эквивалентными: например, предикат эквивалентен сказуемому в суждении и выражается свойством, но свойство может выступать не только в роли предиката (сказуемого), но и в форме субъекта (подлежащего) и объекта.

Однако, сказанное касается классификации знаний по форме, но не по содержанию. Содержательно знания можно классифицировать, например, по глубине и по источнику.

Первый признак классификации является особенно показательным, так как является важной характеристикой для оценки общего уровня качества базы знаний: содержит ли база знаний **глубокие** знания или же в ней представлены только **поверхностные**. Синонимом последних являются очевидные или понятные любому неспециалисту знания, к примеру,

кошка является животным

нейронная сеть имеет нейрон.

Такие знания могут создаваться в большом количестве и для их производства не требуется обладать опытом и навыками эксперта, в этой связи они чаще всего не представляют собой ценности.

Ясно, что основную часть любой базы знаний должны составлять глубокие знания. Примером таких знаний может выступать, например, суждение, что одновременная комбинация конкретных интервалов значений, полученных с конкретных датчиков, ведут с конкретной степенью уверенности к конкретной неполадке. Другим примером является знание о том, что для совершения конкретных маневров конкретному летательному аппарату требуется конкретное оборудование с конкретными (четко определенными) настройками.

С другой стороны знания можно классифицировать по источнику их получения как **первичные** и **вторичные**. Эта классификация важна с точки зрения хранения знаний: первичные знания – это полученные при непосредственном взаимодействии с источником, а вторичные – при анализе литературы, в которой описан изначальный источник. В этой связи знания,

полученные из энциклопедий (в том числе – Wikipedia) являются вторичными, а из научных статей и отчетов – первичными.

Кроме того, тематически знания можно разделить на следующие типы.

Что-знания описывают свойства сущности (экземпляра, класса либо отношения) и ее принадлежность к классу (или ее вхождения в более общий класс), например:

- бак 102 имеет критическую температуру 94 градуса,
- нейронная сеть это математическая модель.

Почему-знания описывают причинно-следственные отношения между индивидами. Особо стоит подчеркнуть, что знания этого типа описывают самостоятельно (часто – неконтролируемо) возникающие последствия тех или иных действий или событий:

- повышение уровня жидкости в резервуаре вызывает срабатывание датчика,
- неверное значение контрольной суммы вызывает ошибку передачи.

Зачем-знания, в отличие от почему-знаний, всегда подразумевают наличие намерения в совершении того или иного действия, и описывают его желаемый результат:

- для плавного снижения температуры нужно нажать кнопку 6,
- для активации переноса груза необходима функция Start().

Как-знания описывают непосредственно процессы: их последовательность, условия перехода к новым этапам, взаимодействие активных участников:

- для снижения затрат на производство следует 50% ресурсов вложить в увеличение производительности, 30% для снижения постоянных и 20% для снижения нерегулярных расходов,
- для отгрузки товара необходимо выполнение двух условий: получение оплаты и наличие товара на складе.

Когда-знания описывают последовательность и время наступления тех или иных событий или действий:

- сначала в течение 20 секунд происходит запуск приложения, затем каждый запрос отрабатывается не более, чем за 1.5 секунды,
- после запуска каждый робот мобильной группы ждет запроса от ведущего в течение двух секунд, иначе инициирует протокол определения ведущего.

Отдельно следует сказать о том, что помимо рассматриваемых в данном пособии семантических и логических моделей знаний, существуют еще фреймовые и продукционные. Скажем про последние: они строятся на создании множества правил вида “если – то”, однако для этих целей могут использоваться различные средства: от логических языков типа “Prolog” до нейронных сетей, которые в таком случае должны разрабатываться изначально как модель причинно-следственных связей между их входом и выходом.

Теперь, имея представление о классификации и видах знаний, можно переходить к первой большой задаче инженерии знаний – к задаче их получения.

Получение знаний

Первый вопрос, который нужно задать к Инженерии знаний: что является источником знаний?

Для ответа на него нужно вспомнить, что знания – это специальным образом структурированные данные, которые могут применять для своих задач машины, программные агенты. Примерный перечень задач представлен выше. Такие знания хранятся в ограниченном наборе форм – семантические базы знаний, нейронные сети, текстовый код на специальных языках и т.д. Именно они являются конечными хранилищами знаний, именно в них знания представлены в явном для машин – **формализованном** – виде.

Но так как в реальности чаще всего знания применяются людьми, а не машинами, то, очевидно, хранить знания в данной форме у людей не было необходимости до недавнего времени. В этой связи все знания, накопленные человеком, представлены в лучшем случае в **структурированной** форме, то есть, в форме схем, графов с именованными связями, диаграмм последовательностей и других формах.

Но для преобразования знаний в какую-либо форму, знания в любом неявном виде (хотя бы в виде текста, конспектов, пометок) должны быть получены из своего первоначального источника, в роли которого, к большому сожалению, чаще всего выступает человек.

Извлечение знаний

И в случае непосредственной работы с человеком мы сталкиваемся с процессом **извлечения знаний** (knowledge elicitation) [Гаврилова, Кудрявцев, Муромцев, 2016], который означает процесс взаимодействия инженера по знаниям (аналитика) с экспертом (человеком – источником знаний) с целью выявления и фиксации знаний эксперта (процесса его рассуждений при принятии решения, структуры его представления о предметной области).

Особенность данного процесса состоит в том, что самому человеку сложно получать и формулировать содержания собственной головы – и самостоятельно это сделать иногда бывает невозможно, в силу того, что, во-первых, этот процесс является непривычным для практико-ориентированных людей и поэтому требует развитых навыков рефлексии, а, во-вторых, самому эксперту почти невозможно проследить всю цепочку собственных рассуждений: для него некоторые умозаключения и выводы являются настолько очевидными, что он не замечает пропущенных промежуточных выводов и предпосылок в собственных рассуждениях. Например, рассуждение, которое для эксперта выглядит очевидным,

неустойчивость магнитометра к помехам -> требуется замена магнитометра

является ценным только при раскрытии промежуточных шагов:

неустойчивость магнитометра к помехам -> высокая дисперсия показаний -> невозможность определения направления робота -> требуется замена магнитометра.

Таким образом, целью инженера по знаниям является, с одной стороны, выявить и прояснить все неочевидные шаги в рассуждениях эксперта.

С другой стороны, эксперту может казаться, что он обладает меньшим объемом знаний, чем это есть на самом деле, поэтому требуется еще и “доставать” из его головы те содержания, о которых он сам может не знать.

В большинстве же случаев извлечение знаний подразумевает прямое взаимодействие аналитика (группы аналитиков) с экспертом (группой экспертов). В зависимости от характера поведения аналитика коммуникативные методы подразделяются на **пассивные** и **активные**.

Пассивные методы включают в себя наблюдение за работой эксперта, прослушивание его лекций и так далее. Особенностью этих методов является минимальные вмешательство в непосредственную работу эксперта и затраты его времени.

Активные методы также можно разделить на индивидуальные и групповые. К первым можно отнести интервьюирование и анкетирование, а ко вторым – организация для экспертов “мозговых штурмов”, круглых столов и различных форсайтных и ролевых игр. Подразумевается, что в обоих случаях аналитик не просто фиксирует материалы, а задает вопросы и направляет работу экспертной группы.

Формы работы могут различаться, а вот направленность всегда должна оставаться постоянной: особую значимость имеют именно глубокие знания, которые содержательно можно представить как знания, описывающие, например,

- характер связанности элементов, показателей. Сюда могут относиться формульные зависимости, принципиальные / функциональные схемы,

знания о сочетаемости / противоречивости некоторых свойств или их значений;

- причинно-следственные связи, к которым можно отнести последовательности / инструкции (какие действия являются правильными или неправильными для каких целей в каких условиях), правила постановки целей и правила планирования, индикаторы свойств (на наличие каких внутренних свойств указывают какие внешние признаки – и наоборот), индикаторы будущих событий: признаки для прогнозирования будущего;
- различные классификации: иерархии классов (как принадлежность к одному классу означает принадлежность и к другому), признаки классов (принадлежащие к определенному классу индивиды обладают рядом каких признаков), правила классификации: наличие определенных свойств означает принадлежность к классу с его признаками, степень надежности признаков классификации и их приоритетность.

Именно такие знания должны составлять основную содержательную часть результатов работы с экспертами.

Приобретение знаний

Однако, процесс получения знаний от эксперта можно несколько автоматизировать: часть процесса по получению знаний переложить на машинные агенты.

Тогда речь пойдет о другом явлении: о **приобретении знаний** (knowledge acquisition) – процессе наполнения базы знаний экспертом с использованием специализированных программных средств. Замысел состоит в том, что управляющая программа сама предлагает эксперту вопросы, сохраняет и специальным образом структурирует его ответы.

Для этих задач, конечно, должна использоваться уже не просто база знаний, а **система управления базами знаний** (СУБЗ) – специальный программный комплекс, обеспечивающий наполнение и изменение самих баз знаний (в том

числе и самостоятельное), а также реализующий интерфейс взаимодействия базы с пользователем.

При этом, система может либо принимать информацию от эксперта в виде заранее неизменяемых заполненных шаблонов, либо непосредственно задавать ему вопросы и вести с ним диалог, учитывая каждый раз его предыдущие ответы – то есть, реализуя некоторую стратегию.

В первом случае для эксперта подготавливаются, например, табличные формы, куда он самостоятельно вносит информацию. Так, одним из полей таблицы могут быть “Возможные виды нештатных ситуаций” при производстве азота, в другом поле напротив каждой ситуации может содержаться ее наиболее распространенная причина, в третьем – узел системы, который может быть поврежден, и так далее. Затем СУБЗ принимает данную таблицу на вход, преобразует всю информацию в код на одном из языков представления знаний. В этом случае аналитик, составляющий шаблоны, должен заранее четко знать, какие именно знания ему потребуются от данного конкретного эксперта.

Организация же диалоговой системы (второй случай) требуется в ситуациях слабой осведомленности аналитика о данной области знаний, когда заранее невозможно четко формулировать вопросы к эксперту или составить готовый шаблон. В этом случае можно опираться на то, что любая область описывается тремя уже упомянутыми формами знаний: фактами, терминологическими и ролевыми аксиомами. Содержательно же в результате должны получиться суждения, содержание которых будет отражать виды глубоких знаний, приведенных, например, в предыдущем пункте.

Так, примерная последовательность действий в этом случае может быть следующей:

1. Программа просит эксперта перечислить основные, наиболее значимые понятия, используемые в его области. В дальнейшем они станут основными классами (которые в данном контексте и имеют смысл понятий).

2. Программа просит определить введенные понятия, обозначив их как подклассы более общих (широких) понятий.
3. Программа просит указать признаки введенных классов.
4. Программа просит указать правила классификации (отнесения к указанным классам).

Далее производим переход к системе фактов.

5. Программа запрашивает важнейшие экземпляры предметной области – и просит их связать с классами, к которым они принадлежат. Например, указать основные рассматриваемые и описываемые аварии – экземпляры класса Авария.
6. Программа просит перечислить все важнейшие **литеральные** свойства (то есть, имеющие значения в виде чисел или строк, а не в виде других экземпляров) каждого экземпляра. Например, модуль Wi-Fi характеризуется максимальным уровнем потребляемого тока, максимальной и минимальной дальностью подключения между двумя точками и т.д. Обратите внимание, что тут речь идет о самих свойствах, а не их значениях.
7. Эксперт вводит значения указанных свойств – те, которые наиболее важно знать, а также ограничения на возможные значения свойств (их допустимые интервалы, количество возможных значений в рамках свойств одного индивида и др.).
8. Ввод важнейших видов отношений с другими экземплярами того же и другого классов – такие свойства назовем **объектными**. Они должны вводиться для каждого экземпляра.
9. Ввод ограничений и ролевых аксиом.
10. Связывание экземпляров через назначение отношений между ними. Они могут быть причинно-следственными (как факторы ошибки и неполадки, к которым они приводят), проясняющими характер связи (взаимоисключающие, сочетаемые) и др.

По мере ввода СУБЗ должна проверять вводимые данные на корректность и не переходить на следующий этап, пока не введен определенный объем информации на предыдущем.

После ввода экспертом всех сведений должна запускаться машина вывода, которая наполнит базу знаний дополнительными суждениями, логически следующими из введенной информации – именно для этого описываются ролевые и терминологические аксиомы.

Последним этапом будет проверка сформированной базы знаний на предмет целостности (непротиворечивости).

Машинное извлечение знаний

Отдельно можно рассматривать также процесс автоматического извлечения знаний из больших хранилищ информации – например, из сети Интернет, из книжных библиотек.

И здесь нужно разделить автоматическое извлечение знаний из неструктурированных источников и из структурированных источников.

В первом случае те же особенности и трудности, что возникают при работе с живым экспертом в процессе извлечения, дополнительно усугубляются тем, что при работе с неструктурированными источниками программа-“извлекатель” воспринимает все встречающиеся слова лишь как набор символов – и не имеет представления об их семантическом значении. В этой связи при простом поиске по ключевым словам релевантность найденных материалов не может быть гарантирована (искомое слово может быть не предметом текста, а лишь просто там встречаться), поэтому для более надежного поиска по неструктурированным текстам используют, например, программы для обработки естественного языка, которые способны не только производить синтаксический разбор предложений, но еще и могут на его основании строить семантические сети.

Другой задачей выступает поиск и извлечение материалов по предварительно структурированным данным. Здесь речь идет о существующих в сети открытых

базах знаний, по которым можно последовательно производить поиск. Примерами могут выступать порталы DBpedia.org, Wikidata.org. Сам процесс отправки запросов к таким хранилищам рассмотрим позже, а сейчас опишем логику работы.

Здесь работа начинается не с перечисления важнейших понятий, а с определения одного основного экземпляра.

Отправляется запрос к базе знаний на предмет нахождения в ней интересующего нас элемента, например, робототехнического манипулятора (robotic arm). Если не существует, то нужно изменить грамматическую форму искомого элемента, подобрать синоним либо произвести поиск по другой открытой базе знаний.

Когда элемент найден, извлечение знаний о нем будет означать поиск по всем отношениям, в которых он участвует:

- запрос всех классов, к которым он принадлежит;
- запрос всех литеральных свойств и их значений;
- запрос все объектных свойств и экземпляров, с которыми он связан через эти объектные свойства;
- запрос всех синонимов и других названий того же самого исследуемого элемента и повторение с ними всех указанных шагов.

В результате получим ряд суждений об исследуемом элементе в форме суждений.

Формирование знаний

Знания можно получать не только при разговоре с экспертом или анализе структурированных / неструктурированных текстовых источников, где они уже представлены в какой-либо (как минимум – неявной) форме. Знания могут быть сформированы непосредственно из данных, в которых они изначально не вшиты. Речь идет, например, о данных в виде таблицы / графика зависимости производительности от температуры и давления в контуре объекта.

В этом случае речь уже идет о формировании знаний – процессе анализа данных и выявления неочевидных закономерностей с использованием специального математического аппарата и программных средств.

Ярким примером данного процесса является обучение нейронной сети на некоторых табличных данных, в результате которого формируются модели связи различных параметров системы. Сюда же можно отнести различные методы аппроксимации зависимостей, кластерный анализ и другие.

В отдельную группу следует выделить методы автоматического порождения гипотез, например, ДСМ-метод. Он основывается не на чисто математических расчетах, как нейронные сети, а на так называемой формализации правдоподобных рассуждений, в котором центральное место занимают логические операции.

В рамках формирования знаний нас более всего интересуют две подзадачи, каждую из которых нам потребуется решать: задача вывода новых (правильных) знаний и задача удаления противоречивых (неправильных) знаний. В обоих случаях для этого будем использовать правила, на оснований которых производится формирование.

ЧАСТЬ 2. СТРУКТУРИЗАЦИЯ. ВИДЫ ЛОГИКИ. ТРИПЛЕТЫ

Понятие структуризации

После процедуры извлечения знания находятся в неявной (**неструктурированной**) форме, такой как конспекты, таблицы, протоколы собеседований, текстовые файлы и т.д. Этого может быть достаточно для реализации основанной на этих знаниях деятельности человека, но этого недостаточно для компьютерных систем. Почему?

Так как для практического применения знаний они должны быть упорядочены, то человек во время обучения и размышления над материалом вынужден заниматься **структуризацией** – то есть, выделением в потоке данных важных связей и формирование суждений, готовых к переводу на языки представления знаний.

Обычно результатом структуризации являются схемы и графы, отражающие характер и количество связей между элементами, а также это могут быть таблицы, формулы и т.д.

Однако, человек обычно производит эту операцию неявным для себя образом (если не занимается построением схем). Его не учили процессу упорядочивания содержания своего мышления, но каждый человек все равно делает это в своей голове – таким образом, ему достаточно для работы и неструктурированного материала, который будет им самостоятельно переработан.

Однако, программные агенты по умолчанию не обладают такой способностью, поэтому либо способность к структуризации и формализации нужно частично или полностью реализовывать программно, либо подготавливать для ввода в базу знаний уже формализованные материалы. Далее мы рассмотрим оба эти пути. Но в любом случае у инженера по знаниям возникает необходимость решения задачи перевода знаний из неструктурированной формы в структурированную.

Получающиеся в результате структуризации формы представления еще не могут быть использованы непосредственно программными агентами, так что программист, работающий с базами знаний, сможет уже без помощи эксперта перевести знания на ЯПЗ.

Рассмотрим процесс и способы структуризации по группам.

Структуризация в форме графических представлений

Первым и, возможно, наиболее известным средством структуризации знаний являются интеллект-карты, которые также называют MindMap. Они широко используются для отображения понятийных структур в преподавании, при проектировании программных систем, в бизнесе и в широком спектре других областей.

Такие карты (разумеется, при правильном составлении) очень наглядны. Их особенность состоит в том, что главный объект изучения располагается по центру карты в виде основного концепта, а от него ветвятся сначала основные направления рассмотрения, а затем и дальнейшая узловая структура. Примером может служить карта на Рис. 4.

Рис. 4. Интеллект-карта.

На первом уровне обобщения находятся элементы “Сотрудники”, “Клиенты” и др., а на втором “Количество”, “Стаж” и др. Всего показаны два уровня.

При внешней простоте разработка таких карт требует навыков системного аналитического мышления. Приведем основные правила построения таких карт.

1. Правило однородности (на одном уровне не должно быть понятий, которые бы выбивались из общего ряда, ни одно не должно быть “лишним”).
2. Правило постепенности обобщения: переход к следующему уровню обобщения должен быть понятен даже для читателя, не являющегося специалистом данной области.

Кроме того, существуют и рекомендации к построению:

- нужно использовать разную величину шрифта при отображении понятий разных уровней;
- полезно использовать цвет для выделения ветвей и уровней;
- визуальные образы и картинки также увеличивают выразительность и-карты.

Так, приведенная карта содержит несколько типичных упущений и ошибок, например:

- на первом уровне находится слишком много объектов, что затрудняет ее восприятие;
- эти объекты имеют разный уровень общности, то есть это слишком разнородные понятия;
- ветви имеют разную глубину детализации;
- не использованы образы и цвет, что важно в и-картах.

Как показывает практика, обучающиеся начинают понимать особенности радиантного (центрированного или иерархического) мышления после третьей либо четвертой построенной карты при строгом анализе ошибок.

Однако, недостатком и-карт с точки зрения структурирования знаний является нечеткое обозначение отношений между элементами, так как они

более приспособлены для демонстрации древовидной структуры произвольных фрагментов знаний.

В этой связи, служить большей однозначности и выразительности могут концептуальные карты (к-карты, concept maps, c-maps). Такие концептуальные карты состоят из узлов и направленных именованных отношений, соединяющих эти узлы. Связи могут быть различного типа, например, «является», «имеет свойство», «приводит к» и т. п. Любая разработка такой карты подразумевает анализ структурных взаимодействий между отдельными понятиями предметной области. Концептуальная карта представляется в виде графа, узлы которого отображают понятия (классы или их экземпляры), а направленные именованные дуги, соединяющие эти узлы, – отношения (связи).

В простейшем случае построение к-карты сводится к следующим шагам:

- определение контекста путем задания конкретного фокусирующего вопроса, определяющего главную тему и границы к-карты;
- выделение классов (концептов) – базовых понятий данной предметной области (обычно не более 15–20 понятий);
- построению связей между элементами – определению соотношений и взаимодействий;
- упорядочению графа – уточнению, удалению лишних связей, снятию противоречий.

Строя к-карты в процессе создания баз знаний или экспертных систем, специалисты получают наиболее полное представление о предметной области. Стоит еще раз подчеркнуть, что к-карты – не только цель, но и средство для более глубокого понимания специфики предметной области. В процессе построения к-карты, то есть при взаимодействии семантических связей нашей памяти с визуальной информацией, связи перестраиваются, порождая, в свою очередь, новые знания.

На Рис. 5 представлена концептуальная карта для системы образования города, разработанная в рамках Системного проекта г. Москвы [Гаврилова, Онуфриев, 2016].

Самой распространенной ошибкой при создании к-карт является неверное обозначение отношений между элементами: может быть неверно определено родовое понятие (“клен” -- является видом -- “лес”) или оно не несет полезной информации (“Электронная кожа” -- является видом -- “кожа”).

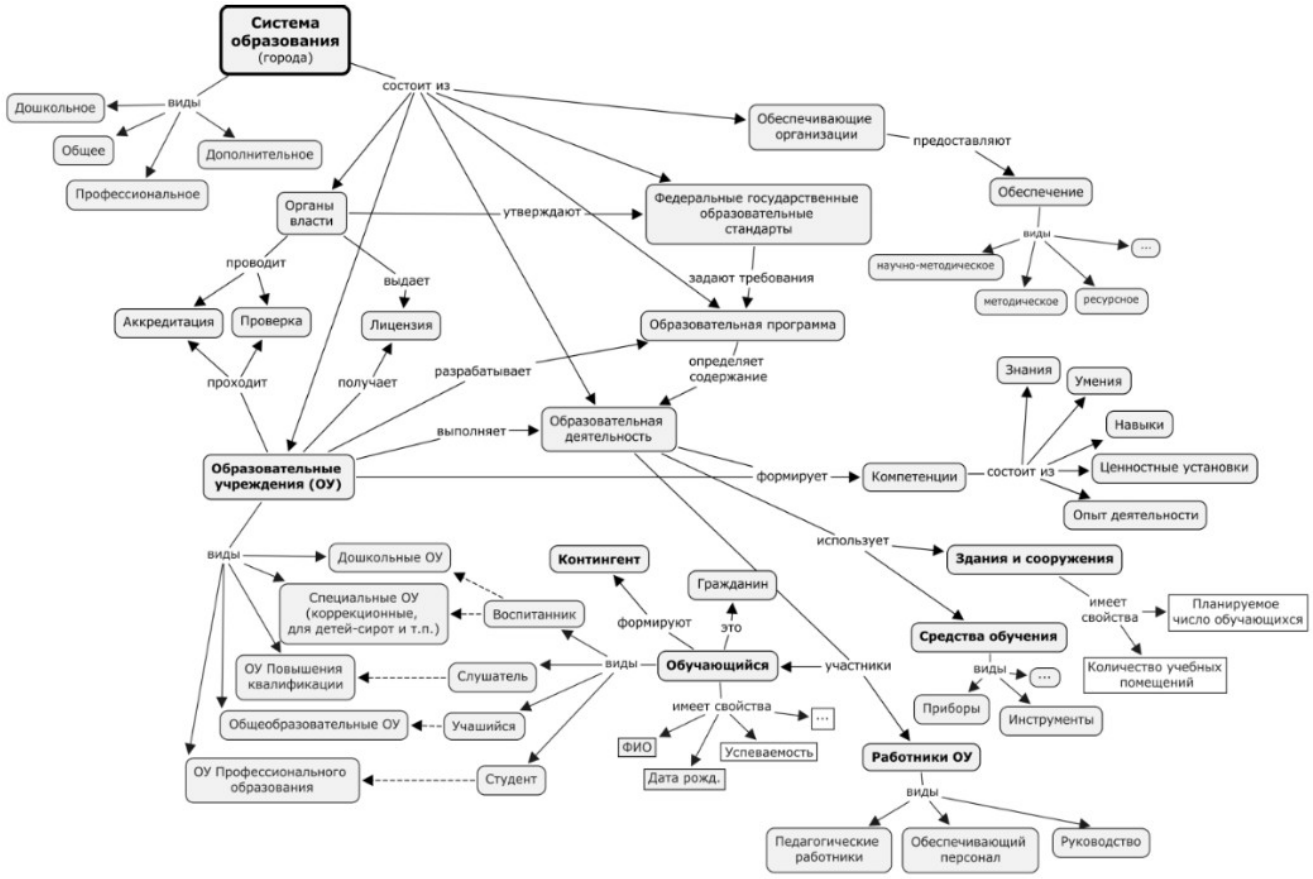

Рис. 5. Пример концептуальной карты

Для того, чтобы построить к-карту по произвольной области знаний, нужно сначала построить ее скелет. Он строится из основополагающих классов – важнейших понятий. После этого обозначаются наиболее часто упоминаемые индивиды, которые сразу же связываются с классами и между собой. Затем подходит этап обозначения правил: ролевых и терминологических аксиом.

Как можно заметить, данный процесс приобретения знаний в общем случае.

К графическим инструментам структуризации можно также отнести и диаграмму Entity-relationship (ER-model), которая более ориентирована на проектирование баз данных, однако позволяет получить наглядные изображения.

Для структуризации причинно-следственных связей может также использована так называемая карта аргументов, которую можно назвать частным случаем к-карты, имеющей в качестве отношений “потому что”, “однако”, “но”, а в качестве узлов – суждения.

В случае описания как- и когда-знаний и в любых ситуациях, когда требуется указывать последовательность действий и условия, могут использоваться такие инструменты, как диаграммы последовательности (sequence diagram), диаграммы swim lane (Рис. 6), диаграммы Ганта.

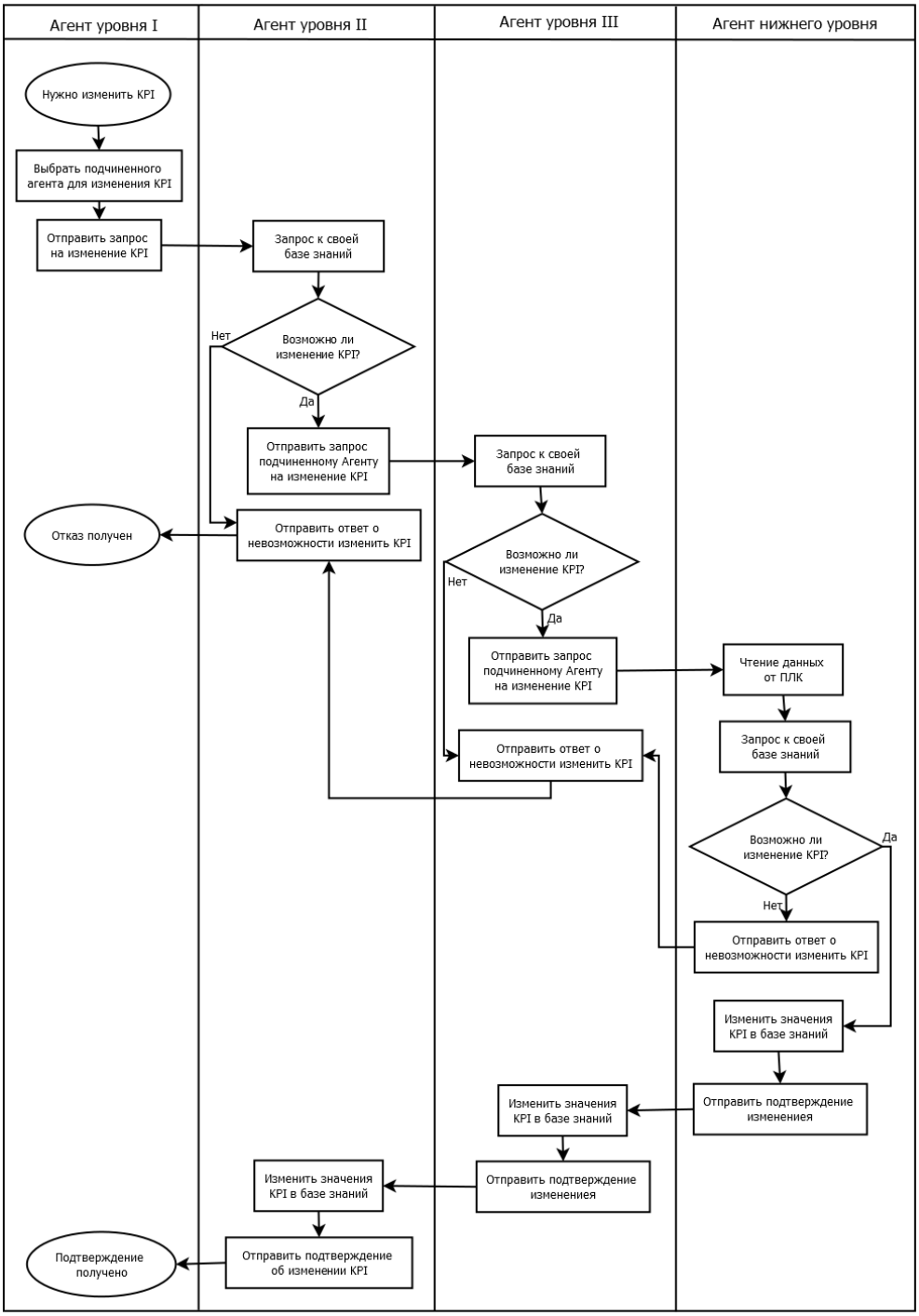

Рис 6. Диаграмма последовательности

Рис. 6 демонстрирует выразительную способность таких диаграмм в описании последовательностей и инструкций: на нем представлена разработанная нами алгоритмическая модель взаимодействия агентов в рамках мультиагентной системы управления предприятием [Ковалевский, Онуфриев, 2019].

И отдельно следует упомянуть также принятые в UML-схемах диаграммы классов и другие типы представлений. Их описание легко найти в соответствующих источниках, однако при использовании UML важно постоянно держать в голове семантическое значение всех используемых элементов. Например, отношение агрегации в диаграмме классов можно считать эквивалентным отношению “используется в” или “включает” в к-картах.

Текстовая структуризация

При этом, наиболее гибкой и в то же время привычной остается текстовая форма структуризации знаний. Здесь мы не ограничены в средствах, однако, если мы пользуемся этой формой, то логичнее для этого избрать такую текстовую форму, которая бы в наибольшей степени подходила для последующей формализации, то есть для преобразования в код на языках представления знаний.

Базовой формой, которую мы будем использовать – это форма триплетов. По факту, это **суждение**, четко разделенное на три (отсюда и название) части: субъект, предикат и объект.

Здесь субъект – это аналог подлежащего, известного из синтаксического анализа предложений. Важно: субъект – это не предложение, не суждение. Субъект выражается одним словом или же словосочетанием.

Предикат, который, как уже было сказано, в разных источниках может называться “роль”, “отношение”, соответствует либо глагольному сказуемому в суждении (“может вызвать”, “препятствует”, “имеет”), либо глагольной частью составного именного сказуемого (например, “является”, “был”, “считается”).

Объект, который иногда называют “значением свойства” (так как в роли предиката выступает свойство) соответствует либо дополнению (второстепенный член предложения) в случае простого глагольного сказуемого (например, “занос”, “книгу”, “установку №2”, “автомобиль”), либо именной частью сказуемого в случае составного именного (“операцией”).

Триплеты, состоящие из таких элементов, приведены ниже:

Тип сказуемого	Субъект	Предикат	Объект
Глагольное	торможение	может вызвать	занос
	повреждение мотора №3	приводит к	необходимости приземления
Именное	удаление	является	операцией

Так, любое суждение сколь угодно сложной формы может быть представлено через конечное число триплетов. Однако, в случае подготовки материалов к формализации нужно учитывать еще и следующие моменты.

1. Несмотря на правила склонения в русском языке, все три элемента в триплете **должны стоять в именительном падеже**, так как в противном случае возникнет необходимость вводить в СУБЗ модуль, который несмотря на различность окончаний будет распознавать в словах с одинаковой основой одну и ту же сущность. По умолчанию же они будут восприниматься как различные.
2. Любой триплет, как уже было сказано, содержит ровно три элемента. И средства обработки языков представления знаний распознают их по наличию пробела между ними. В этой связи, любой триплет должен содержать **ровно два знака “пробел”**.

С учетом этих замечаний, из приведенной выше таблицы получатся следующие триплеты:

торможение можетВызвать занос

повреждениеМотора3 приводитК необходимостьПриземления

удаление является операция

Таким образом, мы получаем список суждений в форме триплетов, которые затем могут быть формализованы.

Стоит заметить, что к-карты очень органично разбиваются на триплеты, кроме того, в редакторах к-карт может быть предусмотрен экспорт карты в форме триплетов (например “CMapTools” – открытое программное обеспечение).

При создании триплетов аналитик сам выбирает, какие свойства использовать и какие индивиды/классы ему для этого понадобятся. Формально при этом он ничем не ограничен. Вместе с тем, с точки зрения дальнейшего использования триплетов, рекомендуется использовать максимально унифицированные предикаты: используетсяДля (inOrderTo), вызывает/вызывается (causes / isCausedBy), состоитИз (consistsOf) и так далее. Для уточнения и внесения деталей (как именно используется, как именно вызывается) используйте дополнительные триплеты – сверх унифицированных, но для стандартных отношений лучше использовать приведенные выше, так как при работе с большой базой знаний, состоящей из тысяч триплетов, разнообразие отношений значительно затрудняет поиск по ней, составление запросов.

Рассмотрим также ситуацию, когда требуется представить в виде триплетов более сложные суждения, например: “Лекция по инженерии знаний состоится в 14:00 в понедельник в аудитории 310 и в 16:00 в четверг в аудитории 314”. При попытке создать такой ряд триплетов:

лекция -- время --> 14:00, лекция -- время --> 16:00

лекция -- место --> 310, лекция -- место --> 314,

мы попадаем в ситуацию, когда время “14:00” никак не связано с местом “310”, ввиду чего становится невозможным из этих триплетов понять, в которое время в котором месте состоится та или иная лекция, так как они никак не сгруппированы.

Решением данной проблемы является создание неких группирующих элементов, которые на Рис. 7 намеренно обозначены “????”, так как не имеет большого значения, что именно будет в них содержаться. Если такой узел не

имеет имени и используется исключительно для связки, то его называют **пустым узлом** (**blank node**).

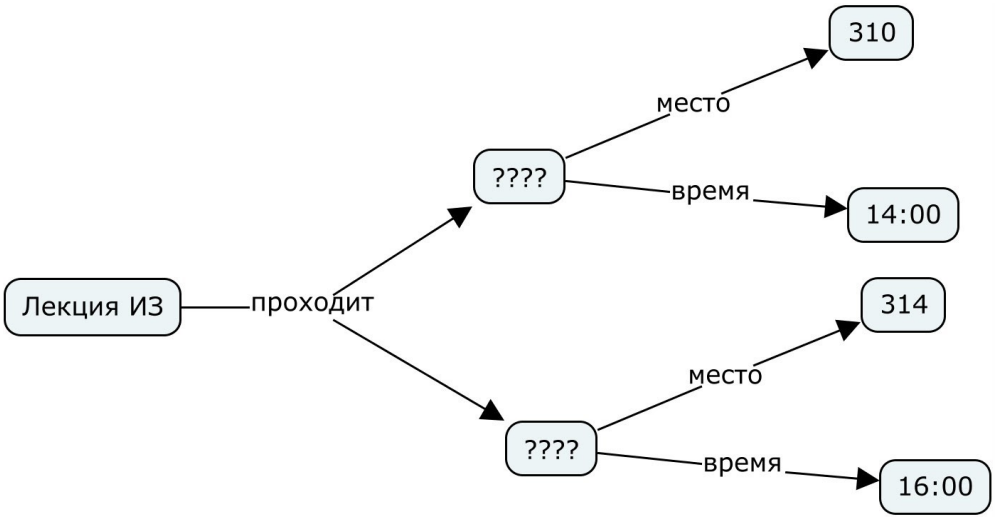

Рис. 7. Группирующие узлы

Следующий важный момент связан с тем, что любой класс можно рассматривать как совокупность его индивидов или как само понятие. Но в триплетах класс всегда используется именно как понятие. И так как субъектом либо объектом триплета могут выступать как индивиды (экземпляры классов), так и сами классы, то часто возникает соблазн, например, класс Студент соединить при помощи предиката “посещаетКурс” с объектом “Инженерия знаний”. Но это будет ошибкой, потому что, очевидно, этим мы хотели сказать не “класс Студент посещает курс ИЗ”, а “все экземпляры класса Студент (то есть, все студенты, – индивиды этого класса) посещают курс ИЗ”. Разница значительна. Если мы хотим сказать про “все индивиды класса”, то нужно использовать правила (о чем будет сказано ниже).

Сам же класс Студент в триплетах можно использовать довольно ограниченно и осторожно, так как использование класса Студент равнозначно использованию понятия “студент”. Таким образом, если в качестве субъекта триплета выступает класс Студент, то в роли его предиката могут быть либо связь подкласс-класс (subClassOf) или иные отношения с классами (например “эквивалентный класс”), либо определяющие отношения (будут рассмотрены в

разделе “Онтологии”), либо характеристики класса как множества (например, “мощность класса”).

Для создания более сложных логических выражений (в том числе и правил) нам далее потребуются языки логики.

Структуризация с использованием формальных языков

Логика высказываний

При структуризации знаний не следует пренебрегать использованием общеизвестных формул и математических нотаций, так как задача состоит в том, чтобы описать неструктурированные знания максимально однозначно и системно. Однако, алфавит простой математики недостаточно сложен, чтобы привычными формулами описать наиболее часто встречающиеся суждения. У него другие задачи.

Формулированием же суждений и описанием их занимается логика, точнее, ее определенные разделы, например, математическая логика. В ее рамках, в свою очередь, можно рассматривать и конкретные языки. Выделим три из них: язык логики высказываний, язык логики первого порядка и язык дескрипционной логики.

Самой простой из перечисленных является логика высказываний (Propositional Logic). Ее язык характеризуется тем, что его элементарной единицей является высказывание: целостное выражение, например “Москва является столицей России”. Логика высказываний не оперирует отдельными понятиями: она более ориентирована на анализ взаимосвязи высказываний.

Это накладывает свой отпечаток на ее язык, который содержит следующие элементы:

- отрицание (¬): $\neg a$ истинно, когда a ложно и наоборот. Обратите внимание, что любое высказывание обязательно либо истинно либо ложно;
- следование или импликация (→): например, $a \rightarrow b$ означает, что из высказывания a следует (логически выводимо) высказывание b;

– дизъюнкция $\vee$, логическое ИЛИ. Например, $a \vee b$ истинно, если истинно либо первое, либо второе высказывания;
– конъюнкция $\wedge$, логическое И. Например, $a \wedge b$ истинно, если истинно как первое, так и второе высказывание.

Можно рассмотреть в качестве конструкции для записи следующее сложное высказывание из “Маленький принц”, Сент-Экзюпери: “Если я прикажу генералу обратиться в чайку и он не сможет выполнить приказ, то виноват буду я, а не генерал” [Зюзьков, 2015].

Для его записи необходимо завести четыре высказывания: a – “я прикажу генералу превратиться в чайку”, b – “генерал сможет обратиться в чайку”, c – “я буду виноват”, d – “генерал будет виноват”. В результате получим:

$$a \wedge \neg b \rightarrow c \wedge \neg d.$$

Обратите внимание, что каждое высказывание, написанное на этом языке, представляет собой не класс, не экземпляр, а сразу целый триплет. По этой причине описывать связи экземпляров друг с другом и с классами на этом языке не представляется возможным.

Еще один недостаток языка логики высказываний, значимый с точки зрения структуризации знаний, – отсутствие инструментов для работы с всеобщностью: высказывание, касающееся некоторой совокупности элементов по форме не отличается от высказывания про один элемент.

Однако, данный язык вполне можно использоваться как промежуточный на пути к применению более сложных средств описания фактов и аксиом. Например, с его помощью можно сформулировать правило:

$$[\ (a \rightarrow b) \wedge b\ ] \rightarrow a,$$

которое затем можно будет проверить на предмет производства ложных суждений – и при помощи определенных инструментов убедиться в ошибочности этого правила.

Языки описания множеств и отношений

Следующим важным шагом к верному пониманию логических процессов в области знаний является знакомство с теорией множеств, особенно в части формальных описаний самих множеств и отношений между их отдельными элементами, так как в последующем мы все это будем использовать при разработке проектов.

Любой класс нужно понимать либо как само понятие, либо как множество индивидов, входящих в класс (при этом, помним, что при записи в форме триплетов всегда имеется в виду первое, а при записи с использованием множеств – второе). В этой связи, если некоторый класс A является подклассом класса B, то с точки зрения множеств мы можем сказать, что $A \subseteq B$ или множество A включается (является подмножеством) множества B, то есть: все индивиды класса A являются также индивидами класса B. Такое включение не является строгим, и любое множество является подмножеством самого себя.

Строгое включение обозначают символом $\subset$ и если $A \subset B$, то A является собственным подмножеством B, то есть, существуют элементы, принадлежащие последнему, но не первому. С точки зрения классов это означает существование индивидов второго класса, не принадлежащих первому.

Основными интересующими нас операциями здесь являются объединение $\cup$ и пересечение $\cap$, а также отрицание $\neg$. Например, $\neg A$ с точки зрения классов означает совокупность всех индивидов, не принадлежащих классу A, а если мы говорим, что $A \cup B \equiv C$, то это означает, что все индивиды, принадлежащие либо классу A, либо классу B, являются также и индивидами класса C.

И здесь нас особо интересуют законы де Моргана, например:

$$\neg(A \cup B) = \neg A \cap \neg B,$$

ведь тогда $\neg C = \neg A \cap \neg B$, то есть: все индивиды, не принадлежащие классу С, не могут принадлежать ни классу А, ни классу В. Казалось бы – речь идет об очевидных выводах, но они обладают математической точностью и легко

формализуются уже в программный код, который может проверять сотни классов базы знаний на наличие каких-либо противоречий.

Важно также рассмотреть отношения, фигурирующие в теории множеств.

Нас будут интересовать, в основном, бинарные связи между объектами множеств, так как впоследствии это будет применяться при формализации связей между индивидами в базах знаний.

Отношение между двумя индивидами описываются так: $\rho = <x, y>$. Затем нужно прояснить содержание отношения ρ, например $\rho = \{<x, y> \mid x$ является отцом $y\}$. Это имеет очевидную связь с языком триплетов.

В целях анализа и модернизации баз знаний нам потребуются **обратные** отношения. Так, можно определить ρ^{-1} $\{<x, y> \mid x$ является сыном $y\}$. То есть, если в базе знаний есть некоторые x и y, связанные одним из этих отношений, то можно сделать вывод, что y и x связаны обратным.

Интересны также и композиции отношений. Так, если введем отношение z $\{<x, y> \mid x$ является братом $y\}$, то композиция $\rho \circ z <x, y>$ будет означать "x является дядей y".

В то же время существует теорема об обратных композициях:

$$(\rho \circ z)^{-1} = z^{-1} \circ \rho^{-1}$$

Это хорошо иллюстрируется концептуальной картой на Рис. 8, где названия обратных свойств затенены красным цветом.

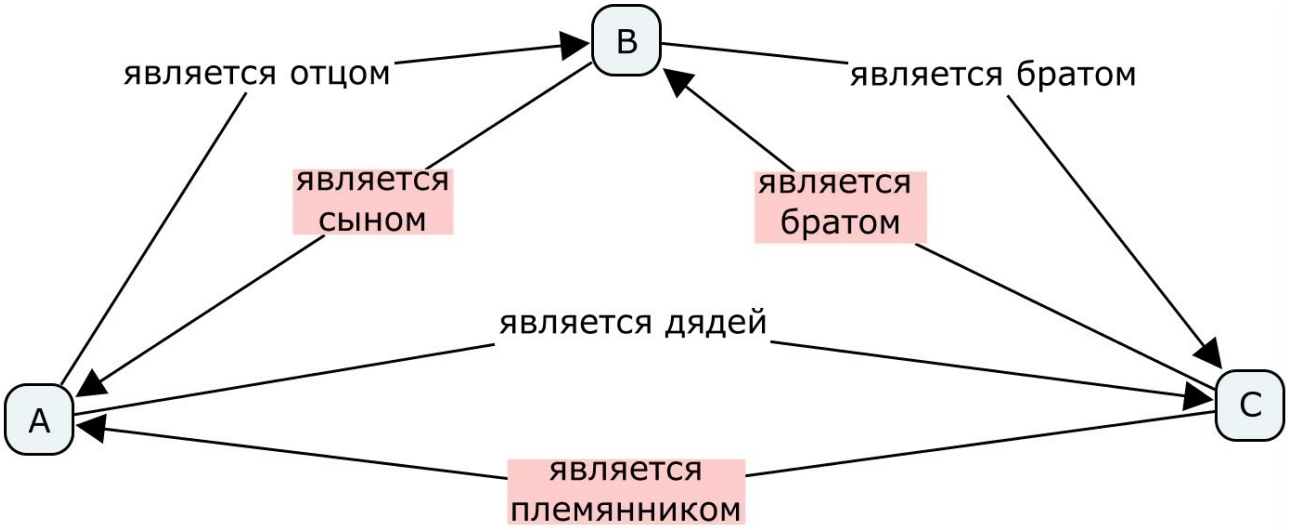

Рис. 8. Композиции отношений и обратные композиции

Помимо композиции свойств и обратных свойств, нужно обратить внимание и на **подсвойства**. Общая суть объясняется следующим образом:

$$\rho <a, b> \,\&\, \rho \subseteq z \;\rightarrow\; z <a, b>.$$

Для решения последующих задач по формированию знаний важно обращать внимание также и на существующие типы отношений, так как это дает дополнительный материал для логического вывода и проверки на наличие противоречий.

Первый тип – **транзитивные** отношения, примером которых является такое отношение $\rho <x, y>$, при котором из $\rho <a, b>$ и $\rho <b, c>$ можно сделать вывод, что $\rho <a, c>$. Иначе это можно записать как $\rho = \rho \circ \rho$. Классический пример транзитивного отношения – $\rho = \{<x, y> \mid x$ является подклассом $y\}$.

Также нас будут интересовать отношения, называемые **симметричными**. Это означает, что из связи $\rho <a, b>$ логически следует $\rho <b, a>$, то есть, отношение является обратным самому себе: $\rho = \rho^{-1}$. В пример такого отношения можно привести “является родственником”.

Здесь же можно сказать и про **антисимметричное** отношение, когда из связи $\rho <a, b>$ логически следует невозможность связи $\rho <b, a>$. Введение данного правила в базу знаний может быть использовано для поиска противоречий в ней.

Рефлексивное отношение подразумевает, что для любого элемента выполняется $\rho <a, a>$.

Функциональное отношение подразумевает единственность своего объекта у субъекта. Таким образом, если индивид связан функциональным отношением с двумя элементами, то из этого следует тождество последних:

$$\rho <a, b> \,\&\, \rho <a, c> \rightarrow b = c.$$

Логика предикатов

Язык логики предикатов, также известный как язык логики первого порядка (First order logic – FOL), лишен недостатков, характерных для языка логики высказываний.

Первой его особенностью, собственно, является применение предикатов. Термин “предикат” из логики первого порядка не нужно путать с термином “предикат” из ряда “субъект”, “предикат”, “объект” в триплетах.

Предикатом будем называть функцию с произвольным количеством входных параметров, имеющую значение либо “Истина”, либо “Ложь”. Так, если предикат $O(x)$ означает “x является оборудованием”, то значением предиката O(двигатель) является “Истина”.

Предикат в логике первого порядка всегда играет роль не части сказуемого, а сказуемого целиком. Например, если x – это произвольный индивид, а C(x) означает: “x является студентом”, то конструкция C(Николай) означает, что “Николай **является студентом**”. Если же мы рассмотрим триплет “Николай является студентом” как триплет, то предикатом (с точки зрения триплетов) в нем будет только одно слово “является”.

Обратите внимание, что смысловое содержание предиката $C(x)$ выбирается произвольно в соответствии с поставленной задачей и предметной областью.

В данном примере **одноместный** предикат C(Николай) соответствует целому триплету, соотносящему индивид “Николай” с классом (множеством) “Студент”. В целом, любой одноместный предикат может быть сведен именно к этому: к причислению индивида ко множеству (то есть, к классу). Это касается даже глагольных сказуемых: чтобы сказать, что экземпляр x бегает, нужно причислить его к предикату $Б(x)$ – существа, умеющие бегать (или бегающие в данный момент). Обратите внимание, что это все равно является приписыванием к классу.

Двуместные предикаты вида $P(x, y)$ могут использоваться уже с целью описания связей между двумя элементами, в нашем случае – индивидов.

Например, *З*(*x*, *y*) может означать “*x* запускает *y*” – то есть, в этом случае двуместный предикат означает целый триплет. Это также согласуется и с рассмотренными ранее отношениями в теории множеств.

Количество параметров и смысловое содержание предикатов определяет сам пользователь, поэтому любой предикат должен быть предварительно описан словесно.

Второй особенностью языка логики предикатов является наличие особых обозначений, называемых кванторами:

- ∀ – квантор всеобщности
- ∃ – квантор существования

Благодаря данным кванторам, мы можем говорить не только о конкретных индивидах, но и о множестве индивидов, отобранных по какому-либо критерию. Например,

∀*x*.*Б*(*x*) означает: “любой, кто умеет бегать”,

∀*x*.*З*(*x*, двигатель5) означает: “всё, что запускает двигатель 5”,

∀*x*.[*Б*(*x*) & *З*(*x*, двигатель5)] означает: “всё, что бегает и запускает двигатель 5”.

Далее уже можно строить утверждения, которые будут применимы ко всему указанному множеству.

Обратите внимание, что в указанных примерах каждый предикат описывает критерий, по которому выбирается множество удовлетворяющих ему элементов, а комбинировать несколько таких предикатов можно при помощью логических операций, например, логического И (символ “&”).

Кванторы всеобщности используются как для вывода новых знаний на основании старых, так и для проверки существующих знаний на противоречивость.

Кванторы существования используются в наших задачах не столь часто и могут применяться для второй задачи, связанной с проверкой на наличие несовместимых суждений.

Также отдельно нужно сказать и про правила, связанные с предикатами.

И первые описывают способ перехода от одного квантора к другому:

$$\forall x.A(x) \equiv \neg\exists x.\neg A(x);\ \forall x.\ \neg A(x) \equiv \neg\exists x.A(x),$$

$$\exists x.\ A(x) \equiv \neg\forall x.\ \neg A(x);\ \exists x.\ \neg A(x) \equiv \neg\forall x.A(x).$$

Второй набор правил объясняет раскрытие логических сложения и умножения при действиях с предикатами:

$$\forall x.[\ A(x) \vee \mathrm{B}(x)\ ] \equiv \exists x.A(x) \vee \exists x.B(x),$$

$$\forall x.[\ A(x)\ \&\ B(x)\ ] \equiv \forall x.A(x)\ \&\ \forall x.B(x).$$

Язык дескрипционной логики

Особую роль можно отвести следующему средству: множеству языков дескрипционной логики – довольно сложным, поэтому прекрасно адаптированным под задачи разработки и описания баз знаний.

Базовым языком дескрипционной логики является ALC (Attributive Language with Complement). Он уже ориентирован на описание соотношений между классами и на описание связи экземпляров и классов, поэтому в нем изначально существуют не только понятие класса (например, C), индивида (например, a) и отношения (например, R), но и конструкции вида: $a{:}C$ (принадлежность индивида к классу), aRb (связь двух индивидов отношением), описанные ранее как элементы системы фактов ABox, а также $C \equiv D$ (эквивалентность классов) и $C \subseteq D$ (включение одного класса в другой) – элементы терминологии TBox. Аналогично и для ролевых аксиом Rbox.

Введены также и классы $\perp$ и T, где первый означает пустой класс (пустое множество), а второй – класс-универсум, чьим подклассом будет любой из возможных классов.

Кроме того, в языке представлены так называемые **универсальные ограничения** вида $\forall R.C$ и **экзистенциальные ограничения** вида $\exists R.C$. В обоих случаях R – это свойство, а C – класс. Здесь под “ограничением” понимают как само условие, которое делит индивидов на удовлетворяющих ему

и не удовлетворяющих, так и само множество удовлетворяющих индивидов. В последнем случае понятие ограничения близко к понятию класса.

Для того, чтобы разобраться с универсальными и экзистенциальными ограничениями, необходимо ввести понятие “последователь по роли (по свойству)”. Если мы имеем триплет *aRb* (то есть, *R*<*a*,*b*>), то индивид *b* назовем **последователем индивида *a* по свойству (роли) *R*** (либо ***R*-последователем**).

Теперь ∀*R.C* можно определить как множество индивидов, у которых все *R*-последователи принадлежат только классу *C*. Иными словами: те, у кого нет ни одного *R*-последователя, не принадлежащего классу *C*. Примером может служить ограничение ∀имеетАвтора.Поэт – множество всех индивидов, чьими авторами являются только поэты (нет ни одного автора, не принадлежащего классу Поэт). На языке логики предикатов это можно было бы записать так: ∀*x,y* [имеетАвтора(*x, y*) & Поэт(*y*)].

∃*R.C* можно определить как множество индивидов, среди *R*-последователей которых существует хотя бы один индивид из класса *C*. Например, ∃написанныеРаботы.КандидатскаяДиссертация – множество всех индивидов, у которых среди написанных работ есть хотя бы одна, относящаяся к множеству (классу) кандидатских диссертаций. На языке логики предикатов это можно было бы записать так: ∀*x*∃*y* [P(*x, y*) & PhDThesis(*y*)].

Есть еще более простая конструкция: *R.ind*, где *ind* – индивид, а не класс, как в приведенных двух примерах. Таким ограничением описывается множество индивидов, чьим *R*-последователем является конкретный индивид. Например, имеетАвтора.ПушкинАС – это набор всех индивидов, чьим автором является А. С. Пушкин.

С добавлением к языку ACL нового функционала его название также изменяется, чтобы отразить его выразительную способность. Например, при добавлении к стандартным трем видам ограничения еще и численных, вида ≥*nR* и ≤*nR*, которые обозначают множество всех индивидов, имеющих не менее (не более) *n* штук *R*-последователей (это называется **ограничением**

кардинальности), к названию добавляется буква *N* (cardinality restrictions). При добавлении ограничений $\geq nR.C$ и $\leq nR.C$ (множество всех индивидов, имеющих не менее (не более) *n* штук *R*-последователей из класса *C*, добавляется буква *Q* (Qualified cardinality restrictions). В результате можно получить очень сложный и гибкий язык. Рекомендуем самостоятельно ознакомиться с расширениями логики ALC.

Важной особенностью таких описаний является то, что таким образом могут описываться в том числе и те знания, которые еще не существуют в данной базе, но которые будут выведены из существующих или добавлены аналитиком в дальнейшем. Такие описания работают на возможность реализации обучения баз знаний.

Описание правил

Итак, правилом является любое суждение (а, значит, и триплет), который касается не конкретных индивидов, а их множества. Последнее, в свою очередь, может быть описано принадлежностью индивидов к одному общему классу или тем, что все они удовлетворяют общему требованию, называемому ограничением.

В этой связи все правила можно разделить на два типа: применимые к классу и применимые к множеству индивидов, обозначенных через ограничение.

Простейший вид правил из первой группы – "**класс-класс**". Понимать данное название нужно так: "Если индивид относится к определенному классу, то этот же самый индивид одновременно принадлежит и другому классу".

Однако, это в точности соответствует вхождению одного класса в другой, поэтому правила данного типа наиболее просты, выражаются в форме триплета

Класс1 — являетсяПодклассомДля → Класс2

и могут быть интерпретированы как "любой индивид, относящийся к классу 1, относится также и к классу 2".

Применяться такие правила могут также и в случае с глагольной формулировкой, например, правило “любой робототехнический манипулятор требует точной настройки” можно сформулировать в форме триплета

Манипулятор — подклассДля → ОборудованиеТребующееНастройки.

При помощи средств логики предикатов это же суждение можно записать, например, так (обозначения для предикатов выбирают произвольно)

$$\forall x.[\ \text{М}(x) \rightarrow \text{Настр}(x)\ ].$$

Обратите внимание на интересный факт, что любой триплет, в качестве субъекта которого выступают классы, часто является правилом.

Обратим внимание также на правила о пересечении и объединении классов вида

$$\forall \text{x}.[\ \text{A(x)} \ \&\ \text{B(x)} \rightarrow \text{C(x)}\ ];$$

$$\forall \text{x}.[\ \text{A(x)} \vee \text{B(x)} \rightarrow \text{C(x)}\ ].$$

Они полезны тем, что создают предпосылку для применения упомянутых ранее законов де Моргана для вывода новых правил.

Правила с использованием квантора существования позволяют находить в базе знаний и исключать из нее противоречащие правилам триплеты. Например, если есть правило

$$\forall x\ \neg\exists y[\ \text{C}(x)\ \&\ \text{B}(x, y)\ ],$$

то нахождение в базе знаний связи B(x,y) является ошибкой и описывающий ее триплет должен быть удален.

Таким же образом работают и правила такого вида:

$$\neg\exists x[\text{C}(x)\ \&\ \text{B}(x, \text{датчикЗвука})].$$

Другой пример – правила вида **класс-свойство**. Они также относятся к множеству индивидов, ограниченных одним классом, – и предписывают им определенные свойства/отношения. Например, предположим, что любая матрица синаптического ядра (в рассматриваемой области) имеет размерность 3.

Пусть указанный класс имеет имя МатрицаЯдра. Обратите внимание, что **недопустимо** составлять триплет

МатрицаЯдра — имеетРазмерность → 3,

так как данное свойство относится ко всем индивидам класса, а не к самому классу как понятию, о чем уже было сказано ранее.

Для того же, чтобы выразить, что любой индивид класса МатрицаЯдра имеет размерность 3, используем язык логики первого порядка, для этого введем предикат МЯ(x), который означает, что индивид x принадлежит классу МатрицаЯдра. При таком обозначении конструкция $\forall x$.МЯ(x) описывает множество всех индивидов, принадлежащих МатрицеЯдра. Теперь можно заменить неверно составленный (показанный выше) триплет на верный:

$\forall x$.МЯ(x) – имеетРазмерность → 3.

Для представления целиком на языке логики первого порядка введем двуместный предикат Разм(x, y), означающий “индивид x имеет размерность y”. Тогда получим $\forall x$.[МЯ(x) → Разм(x, 3)].

На концептуальной карте правила отображаются следующим образом (Рис. 9).

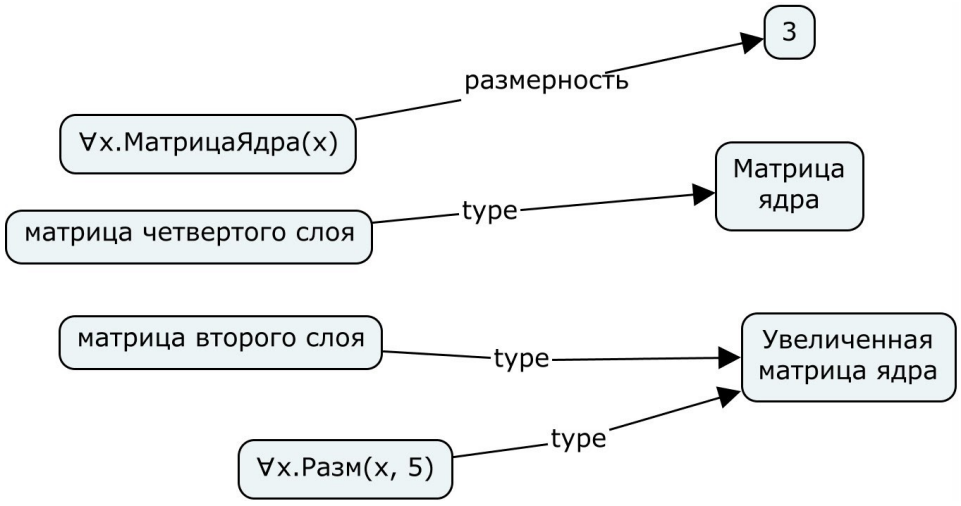

Рис. 9. Введение правил в концептуальную карту

Правила второго типа, в которых множественный субъект определяется не принадлежностью к общему классу, а произвольным ограничением, являются

более сложными, так как аналитик может свободно выбирать субъектное ограничение в формулировке правила.

Простейшие из них формулируются уже известным вам (из дескрипционной логики) образом: *R.ind*, что означает “множество всех индивидов, связанных по свойству *R* с индивидом *ind*”. Если перевести сказанное с языка дескрипционной логики на язык логики первого порядка, то получим

$$\forall x.R(x, ind).$$

После составления таким образом субъекта, он может быть либо отнесен к определенному классу, либо ему может быть присвоено какое-либо значение какого-либо свойства.

В первом случае получившиеся правила принадлежат к виду **свойство-класс** (в отличие от описанных выше правил класс-свойство). Примером такого правила может выступить “любой, в чьем списке изучаемых предметов присутствуют научно-исследовательская работа, является студентом четвертого года обучения”:

дисциплина.НИР — является → Студенты4года.

Примером же второго типа правил (**свойство-свойство**) может выступать следующее:

дисциплина.НИР — дисциплина → преддипломнаяПрактика,

что означает: у всех, у кого есть дисциплина «НИР», есть также и преддипломная практика.

Обратите внимание, что в качестве субъекта таких правил-суждений могут выступать не только простые, но и рассмотренные ранее универсальные ограничения и ограничения на существование, представимые на языке дескрипционной логики. Например, “Все, что может привести только к неполадкам, должно быть причислено к классу: “Негативный фактор”:

∀вызывает.Неполадки — является → НегативныйФактор.

Рассмотрим также и правила, касающиеся единственности (и их тоже приведем не конечный список):

– единственность индивида в классе:

$$\exists x A(x) \,\&\, \forall x, y (A(x) \,\&\, A(y) \rightarrow x = y);$$

– единственность последователя по свойству:

$$\forall c,x, y (A(c,x) \,\&\, A(c,y) \rightarrow x = y)$$

– единственность вида “Только индивиды класса A являются индивидами класса B” (например, только рыбы относятся к способным дышать под водой):

$$\neg\exists x. [\ \neg P(x) \,\&\, Д(x)\ ], \text{ из чего выводится } \forall x [\ Д(x) \rightarrow P(x)\ ].$$

Правила же, касаемые отношений, описываются ровно таким образом, как это было описано в разделе про описание отношений в теории множеств.

Вообще, используя языки логики предикатов и дескрипционной логики, благодаря гибкости последних можно описывать любые правила.

Онтологии

До этого были рассмотрены различные способы структурирования знаний, однако они сами по себе ничего не говорят о создании целостной системы, так как направлены на упорядочивание отдельных сегментов знаний. Теперь необходимо взглянуть на результат структуризации как на единое целое.

Структурировав знания, получим множество следующих элементов:

1. индивиды (экземпляры) – сущности, выбранные из множества других в данной предметной области, имеющие наибольшее значение для задач, ради которых и создается база знаний;
2. классы – они же понятия, они же множества, к которым относятся индивиды;
3. отношения (они же связи, роли, свойства) – они подразделяются на **объектные**, обозначающие отношения вида класс-класс, индивид-индивид, индивид-класс, и на **литеральные**, обозначающие отношения, в которых в роли объекта выступают значения числового или строкового типа, например максимальное давление (105), модель (“Grunfos 300”),

надпись (“Класс для описания органов управления”). Здесь же стоит выделить особенный тип отношений: это **определяющие** отношения или отношения определения, благодаря которым становится возможным не только обозначать для любого класса его родовое понятие (родительский класс), но и определять его через родовое понятие и отличительные свойства и их значения.

4. правила (они же аксиомы) – представляют собой суждения сразу о множестве элементов. Они используются для получения новых знаний или для проверки всей целостной онтологии на непротиворечивость. Сюда могут относиться ролевые аксиомы и терминологические аксиомы, а также другие правила, рассмотренные в соответствующем разделе.

Онтология – это адаптированная для компьютера форма описания некоторой предметной области, представляющая собой множество классов *C*, отношений между ними *R* (в том числе и определяющих *I*) и правил *A* (аксиом):

$$\{C, I, R, A\}.$$

Обратите внимание, что в состав онтологий не входят индивиды. Таким образом, онтология является костяком для будущего наполнения базы знаний индивидами.

Онтологии можно классифицировать по различным признакам. В первую очередь рассмотрим классификацию по составу указанных компонентов.

Самым простым из интересующих нас видов онтологий являются **словари**. Они содержат только два компонента: классы и их определения (определяющие свойства), которые здесь могут выступать как в форме литерального свойства, так и в форме семантической структуры с объектными свойствами и их значениями. В первом случае определение класса “Птица” может происходить через литеральное свойство “определение” и значение последнего “теплокровное хордовое, покрытое перьями”. Во втором случае класс птица определяется следующим образом:

птица – опред-е свойство → вид покрова – опред-е значение → перья.

Очевидно, что определяющих свойств и их значений может быть несколько.

Обратите внимание, что в онтологиях типа “словарь” отсутствуют любые отношения между классами, кроме определяющих, а также нет и аксиом.

Более сложной с точки зрения связей структурой является **таксономия**, представляющая множество классов, связанных либо только отношением “подкласс-класс”, либо также и отношениями определяющими. Это иерархическая структура, представляемая в виде раскрывающегося дерева. Ярким примером таксономии является представление товаров в Интернет-магазинах, где товары делятся на группы (множества, классы), затем на подгруппы и т.д.:

Ультразвуковые датчики — подкласс → Датчики — подкласс → Измерительное оборудование

Похожей структурой является **партономия** – набор экземпляров (конкретных представителей классов), связанных между собой только отношениями вида “часть-целое”. Здесь очень важно не путать отношение “часть-целое” с “экземпляр-класс”. Так, ножка стула является частью стула, но не является видом стула (в отличие от складного стула). Инженер-аналитик является представителем (видом) человека, не является частью человека (в отличие от руки). Для более целостного понимания рекомендую к изучению раздел 10.3 пособия Альберта Николаевича Книгина [Книгин, 2002].

Следующим видом онтологии является **тезаурус –** словарь, в котором помимо литеральных и объектных определений классов представлены также и рассмотренные ранее терминологические аксиомы, то есть отношения подкласс-класс и отношения эквивалентности классов. Здесь же могут фигурировать отношения синонимичности классов, могут присутствовать также указания на классы-антонимы, омонимы и др.

Далее, чем больше вводится новых правил и ограничений, тем более *тяжеловесной* называют онтологию. В самых тяжеловесных можно встретить и выражения на языке дескрипционной логики, и на языке логики предикатов, и

ограничения, свободно сконструированные аналитиками. Разумеется, что обработку всех аксиом базы знаний производит специальный программный модуль, **машина вывода (Reasoner)**, который имеет доступ к правилам и фактам.

Кроме того, онтологии можно классифицировать и по “замаху” их содержания.

Так, самым простым типом являются онтологии **задачи** (Task ontology), в которых описывают решение конкретной проблемы. В основном, там фигурируют классы, связанные с целями, средствами достижения (в том числе оборудованием), способами измерения результатов, ключевыми показателями качества – именно для одной выбранной задачи.

Чуть более содержательно широкими можно назвать онтологии **предметной области** (Domain ontology). В них описываются общие концепты (классы) конкретной области, но без привязки к задачам. Например, может быть описана металлургия в целом, происходящие процессы, способы управления ими, виды обслуживающего персонала и т.д.

На стыке онтологии задачи и онтологии предметной области лежит онтология **прикладная** (Application ontology). В ней рассматриваются несколько задач одной предметной области.

Более общими являются онтологии **верхнего уровня** (Top-Level ontology, Upper ontology, Foundation Ontology), которые могут описывать наиболее предельные концепты типа Событий, Процессов, Задач и даже Пространства / Времени, поэтому их можно использовать как некий общий шаблон при разработке более частных онтологий.

DBpedia как средство машинной структуризации знаний

Применение онтологий можно рассмотреть на примере DBpedia – большого международного проекта, развиваемого и поддерживаемого сообществом пользователей сети Интернет. Задача проекта – структурирование информации, содержащейся в другом, тоже открытом проекте, а именно: в проекте Wikipedia.

Последний содержит огромное количество полезной информации, однако ввиду ее неструктурированности (разумеется, с точки зрения инженерии знаний), ее применение программными агентами затруднено.

Идея состоит в том, чтобы структурировать часть представленной в Wikipedia информации в виде триплетов, а затем описать их на языках представления знаний. Однако, учитывая многообразие страниц, для этого требовалось создать некое универсальное для всех них средство, чтобы структуризация различных страниц происходила однотипно. Для этого средства структуризации требовался механизм группировки страниц Wikipedia – и он был найден: это так называемые шаблоны страниц (Templates).

На странице https://ru.wikipedia.org/wiki/Пушкин_Александр_Сергеевич, к примеру, если зайти в режим просмотра исходного кода, то можно увидеть конструкцию:

```
{{Писатель
| Имя = Александр Сергеевич Пушкин
| Изображение = Orest Kiprensky - Портрет поэта А.С.Пушкина - Google Art Project.jpg
| Ширина = 250px
| ... = …
| Дата рождения = 6.6.1799 (26.5)
| Дата смерти = 10.2.1837 (29.1)
| ... = ...
| ... = ...
}}
```

Это и есть так называемый шаблон страницы, имя которому, в данном случае – Писатель. Более подробная информация о нем приведена на странице https://ru.wikipedia.org/wiki/Шаблон:Писатель.

В Wikipedia реализован и механизм получения списка всех страниц, использующих любой из имеющихся шаблонов. Например, на ресурсе https://ru.wikipedia.org/w/index.php?title=Служебная:Ссылки_сюда/Template:Писатель можно увидеть страницы всех тех, кто был отнесен к писателям. Таким образом, можно легко заметить наличие механизма деления на классы и индивиды.

Более того, у каждого шаблона есть поля. Для “Писателя”, как показано выше, это “Имя”, “Изображение”, “Место рождения” и т.д. В правом верхнем углу любой опирающейся на шаблон страницы можно найти таблицу, в которой два столбца, так что она образует каждой своей строкой по одному триплету, где субъектом является сама страница (точнее, ресурс, представленный на ней), предикатом – содержимое ячейки первого столбца, а объектом -- содержимое ячейки второго столбца.

Следовательно, часть материалов Wikipedia уже структуризированы, но это все еще не результат формализации, поэтому далее требовалось так применить сведения о шаблонах Wikipedia и их полях, чтобы можно было разработать алгоритмы автоматической формализации знаний (перевода их в форму языков представления знаний).

Именно этот механизм (как и некоторые другие) реализован в проекте DBpedia, описан он на ресурсе http://mappings.dbpedia.org/. Идея состоит в соотнесении (англ. *mapping* – сопоставление, преобразование) онтологии Wikipedia, в которую входят в качестве классов сами шаблоны страниц и в качестве отношений между классами поля шаблонов страниц, с собственной онтологией DBpedia, в состав которой также входят классы и отношения (входящие в нее также типы данных “datatypes” здесь рассматривать не будем).

Процесс соотнесения шаблона Wikipedia с онтологией DBpedia означает указание,

1. которому классу онтологии DBpedia (Ontology Class) соответствует конкретный шаблон Wikipedia;
2. которому из отношений/свойств онтологии DBpedia (Ontology Property) соответствует каждое из полей этого шаблона.

Важным моментом здесь является тот факт, что Wikipedia мультиязычна, что создает потенциальную (и часто – реализующуюся) возможность ситуации, когда шаблоны дублируются на разных языках (как “Писатель” и “Infobox writer”). Так как они, по факту, означают то же самое по содержанию, то оба

должны быть соотнесены с одним и тем же классом на DBpedia. Аналогично можно сказать и про мультиязычное дублирование свойств (например, “имя” и “name”) – они должны быть также соотнесены с одним и тем же Ontology Property на DBpedia.

Для реализации сказанного применяется ресурс http://mappings.dbpedia.org/. Сами mappings (на примере Mapping ru:Писатель) выглядят следующим образом:

```
{{ TemplateMapping
| mapToClass = Writer
| mappings =
   {{ PropertyMapping | templateProperty = Имя | ontologyProperty = foaf:name }}
   {{ PropertyMapping | templateProperty = Оригинал имени | ontologyProperty = foaf:name }}
   {{ PropertyMapping | templateProperty = Псевдонимы | ontologyProperty = pseudonym }}
   {{ PropertyMapping | templateProperty = Имя при рождении | ontologyProperty = birthName }}
   {{ PropertyMapping | templateProperty = Дата рождения | ontologyProperty = birthDate }}
   {{ PropertyMapping | templateProperty = Место рождения | ontologyProperty = birthPlace }}
   {{ PropertyMapping | templateProperty = Премии | ontologyProperty = award }}
   {{ PropertyMapping | templateProperty = Сайт | ontologyProperty = foaf:homepage }}
}}
```

Пример взят из раздела Mapping ru:Писатель.

Обратите внимание, что в “templateProperty” указывается поле шаблона Wikipedia, в “ontologyProperty” – свойство из онтологии DBpedia, а в “mapToClass” – класс онтологии DBpedia.

После добавления mapping’а через некоторое время запускается скрипт, который находит все страницы в Wikipedia, сделанные по указанному шаблону, и формализует значения полей в код на языке представления знаний в соответствии с онтологией DBpedia.

В итоге работы системы получают страницы, которые для пользователя имеют следующее представление: http://dbpedia.org/page/Alexander_Pushkin. Вместе с тем, в исходном HTML-коде таких страниц обнаруживаются ссылки на код базы знаний, например: http://dbpedia.org/data/Alexander_Pushkin.n3.

Далее перейдем к рассмотрению собственно языков представления знаний, подходов и процессов перевода знаний на них.

ЧАСТЬ 3. ФОРМАЛИЗАЦИЯ

Способы формализации знаний

Знания в удобном для программных агентов формате (а только такие форматы нас интересуют!) могут быть представлены различным образом. И процесс перевода знаний в машиночитаемую (и -применяемую) форму из любых других форм – это и есть процесс формализации (representation).

Для выполнения этого действия нужно, во-первых, иметь еще не формализованные знания, а, во-вторых, знать, в какую именно форму эти знания планируется перевести.

И этих форм, конечно, существует больше одной.

В качестве первой из них выступают нейронные сети. Как уже было сказано ранее, в них знания хранятся в математической форме представления. Если говорить о нейронных сетях без обратных связей, то речь идет о матрицах весов, каждая из которых описывает топологию соединения нейронов в соответствующем слое, и о матрицах функций активации. Здесь могут храниться разные правила и разные суждения – но только в неявной форме, что, однако, не противоречит решению задачи формализации.

Нейронные сети применяются в задачах обработки математических (в том числе – преобразованных из графических) данных, когда поступающие данные нужно либо классифицировать, либо выполнить на их основании оценку, диагностику, прогнозирование, либо непосредственно их пересчитать и преобразовать в другие сигналы.

Нейронные сети являются конкретным видом математических моделей. В более общем случае знания могут храниться как раз **в форме математических моделей**, в которые входят все виды физических, химических, геометрических и других их видов. Они могут быть представлены в форме матриц и массивов, в форме функций на языках программирования, в форме файлов-проектов различных сред моделирования (Computer aided design – CAD) и т.д.

Следующий формат представления может быть представлен языками **описания продукционных правил**, например, Prolog. Последний, при этом, предлагает единую среду как для хранения знаний, так и для их внесения, извлечения, изменения и использования для выполнения логических операций. В нем можно задавать триплеты простым декларированием, например, триплет

Robot(arm)

означает, что манипулятор является роботом, относится к этому классу. Эта запись более всего соответствует логике предикатов.

Также можно задать правило

Equipment(x) :- Robot(x),

что означает известное нам правило типа “Класс-класс”.

Последующий запрос

?- Equipment(arm)

будет иметь значение “ИСТИНА”.

Другую форму нематематического представления знаний реализуют **технологии семантических сетей** (Semantic web technologies – далее SWT), которые обеспечивают максимально возможную гибкость содержаний баз знаний и могут также интегрироваться в проекты с использованием других форматов. При этом SWT – это вполне конкретный вид технологий, разрабатываемый Консорциумом всемирной паутины (WWW Consortium), так что наиболее полную информацию с обновлениями можно найти на официальном веб-портале.

Для разработки систем на основе знаний требуется, конечно, комбинировать различные формы представления и хранения знаний.

Далее же подробно рассмотрим технологии семантических сетей, которые вполне способны претендовать на роль «сердца» систем на основе знаний.

Ресурсы в Semantic web technologies

Технологии семантических сетей неразрывно связаны с концепцией Linked Data, которая выражается в стремлении упорядочить Интернет следующим

образом: сейчас он представляет собой множество документов (в том числе – динамически сгенерированных), связанных между собой в лучшем случае гиперссылками, но не связанных семантически (смыслово).

Семантическая связь подразумевает, что, например, со страницы с описанием какого-либо вида системы управления программный агент (а не только человек) может перейти на страницу с описанием составляющих этой системы, на страницу с ее применением, с описанием разработчика и т.д. Все это возможно при явном задавании смыслового значения самих связей, когда описаны и осмыслены сами связи (то есть, не просто сказано, что разработчик указан по ссылке, но и описано, что есть «разработчик», чем характеризуется это свойство и т.д.).

Концепция Linked Data означает именно такую связанность, что подразумевает, что разные пользователи создают веб-документы и приложения, при этом устанавливая семантические связи с другими уже существующими документами.

Именно из-за этого формализация при помощи Semantic Web Technologies начинается с понятия **ресурс**, которое означает атомарную единицу знаний (класс, индивид, предикат), имеющую свое описание, являющуюся субъектом одного или нескольких суждений. Иначе говоря, все, что имеет смысл рассматривать в базе знаний.

В этой связи стоит сразу же указать элементы, которые не являются ресурсом, так как ресурсы и не ресурсы формализуются различным образом. Ресурсом не являются числа и строки-константы. Если в базе знаний указывается, например, максимальная дальность того или иного измерителя, то она будет просто числом (15.65). Про само это значение не нужно ничего пояснять, оно не является отдельным субъектом рассмотрения. Равно как и текстовый комментарий для пользователей к тому же самому измерителю: “лазерный дальномер применяется для измерений”. Сама по себе эта строка, взятая в

кавычки, не участвует ни в каких других суждениях и является только объектом суждения (не субъектом).

Любые числа и текстовые строки в рамках SWT именуются **литералами** (literal). И последние, в отличие от любых ресурсов, не имеют никаких **идентификаторов**, которые служат для ресурсов уникальным именем.

Учитывая саму идею Linked Data, легко понять самую первую проблему, которая встает у пользователей: это миллионы ресурсов, которые являются общедоступными, поэтому должны чем-то отличаться друг от друга.

А чем сегодня идентифицируются ресурсы в сети Интернет? Уникальным идентификатором являются URL – Uniform Resource Locator, более известный как “адрес страницы”. Однако, это не единственный способ, позволяющий отделить друг от друга потенциально неограниченное количество элементов – и при этом не потерять всех возможностей URL. Более общим случаем последнего является URI – Uniform Resource Identifier, который не обязательно сводится к Интернет-адресу, так как может быть представлен не URL, а URN – Uniform Resource Name, который составляется по схеме

URN:<namespace>:<name> (конечно, без символа “пробел”).

Таким образом, если Вам не требуется включать реально существующий адрес элемента (или последний не существует), вместо

“http://my.fake.com/individuals/object1”

правильнее будет использовать

“URN:individuals:object1”.

В обоих случаях полный идентификатор элемента можно разделить на его **собственное имя** и на предшествующую ему часть, называемую **пространством имен** (namespace). Этот термин связан с группирующей функцией последнего, ведь и “http://my.fake.com/individuals/”, и “URN:individuals:” можно интерпретировать как некий каталог, в котором хранятся, в данном случае, индивиды. И чем больше становится база знаний,

тем все большую роль приобретает группировка ее элементов по пространствам имен.

Можно иметь одно пространство имен для любых элементов; можно иметь три (для классов, для предикатов, для индивидов); можно иметь больше, разделяя различные типы индивидов по разным пространствам. SWT не регламентируют общее количество и степень структурированности пространств, поэтому эту задачу решает для себя сам разработчик.

Ввиду того, что пространства имен содержат обычно более одного элемента (или одно пространство используется на всю базу знаний, что нежелательно, но не запрещено), то мы наблюдаем постоянное повторение пространства имен в триплете. А это значит, что при обозначении его кратким именем можно повысить читабельность кода базы знаний и уменьшить ее размер.

Если мы обозначим пространство имен “URN:individuals:” кратко “inds”, то при любом упоминании элементов вместо “URN:individuals:object1” можно будет использовать “inds:object1”. И такие обозначения как “inds” называются **префиксами**, каждый из которых обозначает свое пространство имен.

При этом сами префиксы и соответствующие им пространства имен вводит в код базы знаний сам разработчик, декларируя их наподобие представления переменных.

И равно как и в случае использования пространств имен в языках высокого уровня, пространства можно либо создавать, либо подключать уже существующие. А в этот момент мы еще ближе подходим к концепции Linked Data, так как и включение существующих пространств имен в свои базы данных, и распространение своих общедоступным пространств имен с содержащимися в них элементами служат созданию всеобщей сети связанных между собой элементов различных баз знаний, вместе составляющих единую базу.

С целью стандартизации хотя бы общих для всех хранилищ базовых элементов созданы несколько общедоступных пространств имен с наиболее

часто встречающимися и важными классами и предикатами. Мы рассмотрим их ниже.

Итак, на самом нижнем уровне SWT нужно ответить на вопрос о том, которые из предполагаемых элементов базы знаний будут являться ресурсами, а которые – литералами. Кстати, именно здесь проходит грань между семантическими базами знаний и нейронными сетями: первые подразумевают объектно-ориентированный подход, когда происходит, например, выбор между экземплярами классов или оперирование их свойствами, что проблематично реализовать при помощи нейронных сетей.

Нотации в Semantic web technologies

Когда определен способ описания каждого элемента в отдельности, то возникает вопрос о способах связывания ресурсов с ресурсами и ресурсов с литералами. Фактически, здесь мы переходим к вопросам написания кода самой базы знаний.

Вопросы нотации – это вопросы синтаксиса, так как в рамках SWT нотациями принято называть разработанные языки представления знаний, каждый из которых обладает собственным синтаксисом. И разработали их несколько:

- RDF/PHP,
- RDF/XML,
- N3/TURTLE,
- JSON-LD и некоторые другие.

Возьмем два триплета следующего вида

roboticArm isUsedFor detailsMoving И roboticArm hasPart endEffector

и рассмотрим, как они будут выглядеть выполненными в каждой из обозначенных нотаций.

В нотации **RDF/PHP**:

```
array ( 'urn:test:roboticArm' =>
array ( 'urn:test:isUsedFor' =>
```

```
array ( 0 =>
array ( 'type' => 'uri', 'value' => 'urn:test:detailsMoving', ), ), 'urn:test:hasPart'=>
array ( 0 => array ( 'type' => 'uri', 'value' => 'urn:test:endEffector',),),),)
```

В нотации **RDF/XML**:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:ns0="urn:test:">
    <rdf:Description rdf:about="urn:test:roboticArm">
        <ns0:isUsedFor rdf:resource="urn:test:detailsMoving"/>
        <ns0:hasPart rdf:resource="urn:test:endEffector"/>
    </rdf:Description>
</rdf:RDF>
```

В нотации **JSON-LD**:

```
[{"@id":"urn:test:detailsMoving"},
{"@id":"urn:test:endEffector"},
{"@id":"urn:test:roboticArm","urn:test:isUsedFor":
[{"@id":"urn:test:detailsMoving"}],
"urn:test:hasPart":[{"@id":"urn:test:endEffector"}]}]
```

И в нотации **N3/TURTLE**:

```
@prefix test:<URN:test:>.
test:roboticArm test:isUsedFor test:detailsMoving;
test:hasPart test:endEffector.
```

Ввиду наилучшей понятности и читаемости последней нотации мы будем далее пользоваться ею.

TURTLE (terse RDF triple language) предполагает представление триплетов в форме субъекта, предиката и объекта, просто разделенных пробелами, причем внутри каждого из этих элементов не должно быть пробелов. Исключение составляют только литералы-строки, которые берутся в кавычки и внутри которых можно пользоваться кириллицей и писать текст на естественном языке:

test:roboticArm rdfs:label “робототехнический манипулятор”.

Следует еще раз обратить внимание, что строка-литерал “робототехнический манипулятор” также не имеет префикса, что означает не включение ее ни в

какое пространство имен, потому что она не является ресурсом, а поэтому не обладает идентификатором (URI) и не может быть субъектом никакого триплета.

Все ресурсы же обязательно имеют идентификатор, в связи с чем для сокращения кода обычно пользуются префиксами, которые объявляются как показано выше:

@prefix test:<URN:test:>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.

При использовании префикса ресурсы описывают в форме

префикс:собственное_имя,

например, test:roboticArm.

Здесь важно учитывать, что при интерпретации кода базы знаний каждый префикс вместе со следующим за ним двоеточием (в примере выше – “test:”) будет заменен на содержание угловых скобок (в данном примере – на “URN:test:”). В связи с этим нужно следить, что содержание угловых скобок при представлении префикса должно заканчиваться на символ разделитель. Обычно роль последнего играют символы “/”, “:” или “#”. Поэтому если бы в примере выше префикс test был бы описан как “@prefix test:<URN:test>.”, то test:roboticArm был бы проинтерпретирован не как <URN:test:roboticArm>, а как <URN:testroboticArm>, то есть мы получили бы единое имя “testroboticArm”. Аналогично без символа “#” в угловых скобках при декларировании префикса rdfs элемент rdfs:label бы был воспринят как <http://www.w3.org/2000/01/rdf-schemalabel>.

Тут же нужно отметить, что использование префиксов не является требованием синтаксиса, поэтому триплет

test:roboticArm rdfs:label “робототехнический манипулятор”

может быть записан в виде

<URN:test:roboticArm> <http://www.w3.org/2000/01/rdf-schema#label>
“робототехнический манипулятор”.

В случае употребления идентификатора без префикса использование угловых скобок для каждого ресурса является строго обязательным.

Следующий важный момент связан уже с пунктуационной частью синтаксиса TURTLE.

Каждый триплет должен заканчиваться символом точки кроме случаев **сокращенной записи**. Последняя означает возможность употребления, во-первых, вместо конструкции из трех строк с повторением субъекта и предиката

```
test:roboticArm test:isUsedFor test:detailsTranport.
test:roboticArm test:isUsedFor test:heavyThingsTransport.
test:roboticArm test:isUsedFor test:researchWork.
```

выражения

```
test:roboticArm    test:isUsedFor    test:detailsTranport,
                                     test:heavyThingsTransport,
                                     test:researchWork.
```

Это те же три триплета, но записанные короче. Для этого объекты разделяются запятой, а точка употребляется единожды.

Во-вторых, сокращенную запись можно использовать также и при повторении одного и того же субъекта в нескольких триплетах, в связи с чем конструкцию из, опять же, трех триплетов

```
test:roboticArm    test:isUsedFor    test:detailsTranport.
test:roboticArm    rdfs:label        “робототехнический манипулятор”.
test:roboticArm    rdf:type          test:LabEquipment.
```

можно заменить более простой записью

```
test:roboticArm    test:isUsedFor    test:detailsTranport;
                   rdfs:label        “робототехнический манипулятор”;
                   rdf:type          test:LabEquipment.
```

Именно такая запись является наиболее предпочтительной: все триплеты должны группироваться (и, желательно, сортироваться) по субъекту с использованием символа “;”, как показано выше.

Если вспомнить про упомянутые ранее пустые узлы (blank nodes), то и для них выделены специальные синтаксические средства выражения: они не

являются ресурсом и поэтому не обладают идентификатором, однако все же могут играть роль субъекта в триплете.

Они обозначаются [ ], но в таком виде используются только если аналитик хочет сообщить, что ему нечего поставить на это место. В случае же применения для связки (как в примере с местом и временем лекции), предикаты и объекты, для которых пустой узел является субъектом, пишутся прямо внутри него. В этой связи упомянутый пример следует кодировать так:

```
test:lectureKE   test:takesPlace [   test:time    “14:00”;
                                     test:room    310 ];
                 test:takesPlace [   test:time    “16:00”;
                                     test:room    314 ].
```

То есть, субъект пустого узла в квадратных скобках просто пропущен, при этом в нашем примере в каждом из пустых узлов (в квадратных скобках) таким неявным образом указываются сразу два триплета, причем – без субъекта.

Однако, возможна ситуация, когда в базе знаний имеется несколько пустых узлов, которые есть необходимость различать между собой, чтобы иметь возможность прикреплять к ним предикаты и субъекты не одной строкой, как в примере выше, а в произвольных участках кода. Специально на этот случай существуют **идентификаторы пустых узлов** (blank node identifiers), которые вместо префикса всегда имеют символ “_”. В итоге описанный выше пример с введением идентификаторов пустых узлов можно представить следующим образом:

```
test:lectureKE   test:takesPlace    _:id01;
                 test:takesPlace    _:id02.
_:id01           test:time          “14:00”;
                 test:room          310.
_:id02           test:time          “16:00”;
                 test:room          314.
```

Далее, когда мы рассмотрели простейшие способы работы с ресурсами и литералами, нужно передвигаться к следующим уровням базовой модели SWT.

RDF и RDFS

Сама концепция Linked Data подразумевает не только возможность применения общедоступных пространств имен с содержащимися в них элементами, но и определенные правила стандартизации, направленные на возможность использования общих для всех конструкций.

W3C как разработчик SWT ввел несколько общедоступных пространств имен, призванных предоставить для всех пользователей единый набор элементов, – классов и свойств – которые пользователи будут применять для решения определенных задач без необходимости разработки своих собственных.

Например, рассмотрим задачу обозначения принадлежности конкретного индивида к классу или классам. Для этого можно было бы создать свое собственное свойство test:belongs или ему подобное. Но если каждый пользователь будет придумывать и вводить для этой цели свои собственные свойства, то применение чьей-либо базы знаний будет нуждаться в предварительном решении задачи поиска подходящих свойств для связывания индивида и класса. Но если это делает программный агент, то задача становится совсем не тривиальной.

Вместо этого всем пользователям при разработке своих баз предлагается использовать пространство имен RDF (Resource Description Framework) и содержащееся в нем свойство **rdf:type**, связывающее свой индивид-субъект с классом, являющимся объектом, например,

test:roboticArm rdf:type test:labEquipment.

При этом если вместо “rdf:type” написать просто “a”, то интерпретаторы, поддерживающие SWT, воспринимают “a” как сокращение от “rdf:type”, поэтому можно применять еще более короткую запись

test:roboticArm a test:labEquipment.

Конечно, подразумевается предварительная декларация префикса rdf:

@prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

При этом, данный URL является действующим, то есть, можно открыть файл описания всех элементов, содержащихся в данном пространстве имен, выделить среди них нужные для своего проекта.

Рассмотрим самые важные для нас элементы пространства имен RDF.

Класс **rdf:Statement** используется в случаях, если нужно высказать суждение о другом суждении. Например, об активации определенного триплета в базе знаний: если определенный датчик показывает температуру выше заданного порога, то нужно добавить в базу знаний информацию о том, что система переходит в нестабильное состояние. В этом примере триплет “система имеетСостояние нестабильное” должен до этого момента быть в статусе “неактивен”, а теперь – активироваться. Чтобы отделить такие суждения от прочих экземпляров класса rdf:Statement, можно обозначить их как экземпляры класса State (состояние), чем они семантически и являются.

Для разграничения таких статусов нужно создать экземпляр указанного класса:

test:nonStableState a rdf:Statement, test:State.

Затем привязать к нему субъект, предикат и объект триплета, который хотим иметь возможность активировать и дезактивировать, при помощи свойств **rdf:subject**, **rdf:predicate** и **rdf:object** соответственно следующим образом:

```
test:nonStableState   rdf:subject      test:system;
                      rdf:predicate    test:hasState;
                      rdf:object       test:nonStable;
                      test:status      test:Disabled.
```

Так, триплет о состоянии системы оказывается содержащимся в базе знаний, но не как триплет, а как три отдельных компонента, привязанных к экземпляру класса rdf:Statement. Тогда далее можно при определенных условиях передать в систему управления базой знаний список имен (идентификаторов) экземпляров rdf:Statement, которые нужно активировать / дезактивировать. Тогда, если предварительно был разработан или импортирован программный обработчик,

то он сможет найти по базе знаний все активированные экземпляры rdf:Statement и добавить в базу знаний их компоненты в явной форме, а также найти все неактивные и явным образом удалить их из базы знаний.

Следующим важным элементом являются списки, которые иногда называют коллекциями (RDF collection). Они будут иметь важное значение, когда мы будем рассматривать задачу формализации правил.

Рассмотрим пока без содержательных пояснений конструкцию:

test:disClasses1 owl:members (test:Book test:Car test:Person).

Элементы в круглых скобках, разделенные знаком пробела, и составляют компоненты списка **rdf:List**. Такая запись будет автоматически преобразована в довольно интересную связку:

```
test:disClasses1   owl:members    _:autos1.
_:autos1           rdf:first      test:Book;
                   rdf:rest       _:autos2.
_:autos2           rdf:first      test:Car;
                   rdf:rest       _:autos3.
_:autos3           rdf:first      test:Person;
                   rdf:rest       rdf:nil.
```

Каждый, знакомый со списками в языках программирования высокого уровня, легко узнает здесь указатели от каждого предыдущего элемента списка к каждому последующему, точнее: к каждому новому подсписку, обозначенному идентификатором пустого узла _:autos*N*. Применение элементов **rdf:first** и **rdf:rest** не требует пояснения, а индивид **rdf:nil** класса rdf:List обозначает пустой список, что логически схоже с NULL как завершающим элементом.

Основными причинами, по которым стоит использовать списки, являются простота заполнения списка в коде базы знаний (простое перечисление в скобках) и наличие специальных механизмов запроса элементов по их номерам, а также вывода всех элементов списка.

Ну и последнее, что нас может интересовать в данном пространстве имен, – это классы rdf:Bag, rdf:Seq и rdf:Alt, которые являются подклассами для rdfs:Container. Здесь не реализовано (в отличие от rdf:List) никаких специальных

механизмов получения элементов из контейнера, а сам код его создания пишется иначе:

```
test:bag1  a        rdf:Bag;
           rdf:_1   test:roboticArm;
           rdf:_2   test:quadCopter;
           rdf:_3   test:laserSensor.
```

При этом синтаксис для всех трех типов контейнеров одинаков. Различие между ними – смысловое: **rdf:Bag** включает в себя элементы, порядок перечисления которых не имеет значения (например, перечисление имеющегося оборудования); обозначенный в **rdf:Seq** порядок имеет важное значение (например, порядок действий при возникновении нештатной ситуации); контейнер же **rdf:Alt** включает в себя взаимоисключающие элементы (то есть, сам факт включения элементов в этот контейнер имеет семантическое значение).

И на этом весь интересующий нас объем пространства имен, обозначаемого префиксом rdf:, заканчивается.

Несмотря на наличие множества полезных элементов, их недостаточно для стандартизации некоторых важных описаний в базах знаний. Например, можно указать “test:roboticArm a test:labEquipment”, но при этом объект этого триплета, класс test:labEquipment, не приобретает для программных агентов никакого семантического значения, так как в rdf: нет средств установления связей между классами. Не говоря уже о том, что нет средств для явного определения test:labEquipment как класса.

По причине нехватки стандартных средств для решения этих (и иных) задач, было введено следующее пространство имен, обозначаемое префиксом **rdfs:**, который означает “RDF Schema”. То есть, речь идет уже о связывании в схему, о построении некоторой системы.

Так, в указанном пространстве имен, имеющем адрес http://www.w3.org/2000/01/rdf-schema# (который также является актуальным и

позволяет изучить содержимое), можно найти свойство **rdfs:subClassOf**. Предназначение последнего довольно прозрачно, стоит лишь уточнить, что его использование для связывания индивидов и классов неприемлемо, так как для этого применяют свойство rdf:type (или, сокращенно, “a”). Для обозначения же подсвойств применяют **rdfs:subPropertyOf**.

Здесь же, конечно, нельзя не упомянуть и класс **rdfs:Class**. Если нужно указать, что test:labEquipment является классом, то нужно написать

test:labEquipment a rdfs:Class.

То есть, любой класс – это индивид класса rdfs:Class. Это одна из редких ситуаций, когда сам класс рассматривается как индивид. Равно как любой предикат можно рассматривать как индивид общего класса **rdf:Property** из прошлого пространства имен.

Часто применяют также свойства **rdfs:label** и **rdfs:comment**:

test:nonStableState rdfs:label “Нестабильное состояние системы”;
rdfs:comment “Нестабильность, нужны действия”.

Они помогают в задаче вывода для пользователя материалов из базы знаний в понятной для него форме, так как конструкцию test:nonStableState он может понять неверно.

Последняя из интересующих нас здесь пар свойств – это **rdfs:domain** и **rdfs:range**. При этом, субъектом в триплете с любым из этих свойств должно выступать свойство, играющее роль предиката другого триплета. Иначе говоря, domain и range являются свойствами других свойств, характеризуют их.

Рассмотрим триплет

test:lowBatteryLevel test:causes test:WirelessAgentDisconnection.

Мы видим в данном примере, что у предиката, выраженного свойством test:causes, есть субъект test:lowBatteryLevel и объект test:WirelessAgentDisconnection. Если мы хотим указать, что *субъекты* предиката, выраженного свойством test:causes, должны принадлежать только классу, например, test:ProblemReason, то создаем триплет

test:causes rdfs:domain test:ProblemReason.

Соответственно, в этом случае индивид test:lowBatteryLevel (имеющий свойство test:causes) должен быть автоматически отнесен к классу test:ProblemReason.

Свойство rdfs:range работает так же, но регламентирует класс, которому должны принадлежать *объекты* предиката, выраженного описываемым свойством, например,

test:causes rdfs:range test:Problems.

Пространство имен, связанное с префиксом rdfs: также не обеспечивает достаточного для наших задач функционала, поэтому нам потребуется рассмотреть следующее. Но перед этим, чтобы можно было начинать работу с материалами базы знаний, мы рассмотрим другой инструмент – SPARQL.

SPARQL

Нужно вспомнить упомянутый ранее термин “системы управления базами знаний” (СУБЗ). СУБЗ – это программный модуль, выполняющий определенные действия с базой знаний, так как последняя является сама по себе лишь набором файлов с кодом на определенной нотации. То есть, сама база знаний не может обеспечить вообще никакого функционала по работе со знаниями, кроме непосредственно хранения фактов и правил.

СУБЗ же должна обеспечивать следующий функционал по работе со знаниями:

- запрос указанных пользователем знаний из базы;
- внесение указанных пользователем изменением в базу (добавление триплетов, их изменение и удаление);
- автоматический вывод новых знаний на основании старых (knowledge inference);
- автоматическая проверка базы знаний на наличие в ней противоречий (inconsistency checking);

– обеспечение интеграции с другими средствами хранения и обработки знаний (например, с хранилищем математических формул и модулем выполнения вычислений).

Большую часть этих функций объединяет необходимость запрашивать из базы знаний нужные для их выполнения факты и правила – с последующей обработкой результатов запроса. Это объясняет необходимость изучения и реализации механизма обработки запросов при работе с базами знаний.

Для этих целей в рамках SWT разработчики предложили инструмент, называемый SPARQL (рекурсивно обозначающий SPARQL Protocol and RDF Query Language). И, как следует из названия, он включает в себя сразу несколько компонентов:

– протокол (SPARQL Protocol);
– язык (SPARQL language);
– формат представления результата (SPARQL Query Results XML Format).

Необходимость введения протокола обоснована самой концепцией Linked Data, подразумевающей возможность для разных пользователей обращаться к любым открытым базам знаний. Но так как физически последние расположены на различных серверах (и в различных странах), то одним из самых простых способов было разработать надстройку над протоколом HTTP, чтобы обращаться к серверу, хранящему базу знаний, из которой нужно получить информацию.

Программный модуль, принимающий и обрабатывающий запросы на языке SPARQL (SPARQL-запросы), отправленные к нему через протокол SPARQL, называется **терминалом SPARQL** (SPARQL endpoint). И так как SPARQL протокол работает поверх HTTP, то для запроса из удаленного узла требуется отправить HTTP-запрос по IP-адресу или, в общем случае, по URL-адресу терминала.

Общая формула передачи запроса в виде HTTP-строки выглядит так:

{URL терминала}?query={текст запроса на языке SPARQL}

В случае с популярным терминалом http://dbpedia.org/sparql, соответственно, строка будет выглядеть, например, так:

http://dbpedia.org/sparql?query=select+distinct+%3FConcept+where+%7B%5B%5D+a+%3FConcept%7D+LIMIT+100

Обратите внимание, что так как, опять же, обозначенный протокол работает по HTTP, то сам переход по указанной ссылке уже означает факт отправки запроса, так что пользователь, перейдя по ней, сразу же увидит таблицу результатов.

И табличное представление результата – это тоже не случайность и не идея разработчиков данного терминала: это требование к формату представления результата SPARQL-запросов. Точнее сказать, таблица является наиболее удобной формой его выражения: каждая привязанная переменная – в новом столбце, а каждый результат – в отдельной строке. Представление результата для пользователя, конечно, является не основной целью терминала SPARQL, который нужен в большей степени для считывания этих данных программными агентами. И именно здесь принципиально важна унифицированность результатов. Благодаря ей агент может отправить запрос к любому действующему терминалу и результат любого из них он может обработать одним и тем же алгоритмом.

Осталось только разобраться с самим языком запросов, который неразрывно связан с термином **шаблон триплета** (triple pattern), означающим произвольный триплет, в котором один, два или все три элемента заменены **переменными**. Последние представляются в виде произвольного набора символов, начинающихся со знака ?, например, ?variable, ?newEqupment и т. д.

Первое, на что нужно обратить внимание: как и в привычных Вам языках программирования, выбор имени переменной никак не влияет на ее содержание, – только на удобочитаемость для разработчика. Ее же будущее содержание (значение) определяется контекстом ее применения в запросе.

Происходит это следующим образом: пользователь создает запрос, содержащий не менее одного шаблона триплета. При обработке запроса этот шаблон накладывается на все имеющиеся в базе триплеты, выбирая только соответствующие этому шаблону. Соответствующие шаблону – это триплеты, элементы которых совпадают с постоянными (не переменными) частями шаблонов. Например, шаблону “?equipmentInds rdf:type test:Equipment” соответствуют все имеющиеся в базе триплеты, у которых предикат и объект, соответственно, rdf:type и test:Equipment. Когда же сформирован список всех соответствующих шаблону триплетов, то результатом запроса (если не указано дополнительных условий) будет, в данном случае, список всех субъектов из соответствующих шаблону триплетов (то есть, список ресурсов, на чьем “месте” в их триплетах находится переменная ?equipmentInds).

Полный код такого запроса:

```
PREFIX test:<urn:test:>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?equipmentInds
where
{
        ?equipmentInds rdf:type test:Equipment.
}
```

Обратите внимание, что это запрос не к внешней базе знаний, а к локальной, с элементами из предыдущих рассмотренных примеров. В этой связи объявленные ранее пространства имен и префиксы должны быть также обозначены, хотя синтаксис немного отличается: отсутствует символ “@” и точка в конце строки объявления.

Ключевое слово *select* требует после себя перечисления всех переменных, значения которых запрашивающий желает получить, а сам запрос формируем после слова *where* в фигурных скобках. Показанный выше запрос приведет к списку возможных значений переменной ?equipmentInds – то есть, список индивидов класса test:Equipment, содержащихся в данной базе знаний.

Запрос же

```
PREFIX test:<urn:test:>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?equipmentInds ?classes
where
{
        ?equipmentInds rdf:type ?classes.
}
```

будет иметь ответ в виде таблицы уже из двух столбцов (так как после слова *select* указаны две переменные), в которой каждая строка – это пара индивид-класс.

Следующий шаг для создания более гибких запросов состоит в комбинировании шаблонов триплета. При этом, каждый из этих шаблонов следует рассматривать как множество соответствующих ему результатов (в виде значения переменных). Тогда комбинация триплетов соответствует комбинации множеств результатов (при их пересечении, объединении, вычитании и т.д.).

И простейшая здесь операция – это **пересечение** результатов. Синтаксически это описывается просто добавлением еще одного шаблона в запрос

```
...where
{
        ?equipmentInds      rdf:type      test:Equipment.
        ?equipmentInds      rdfs:label    ?indLabel.
}
```

Теперь в окончательное множество результатов попадут лишь те индивиды, которые одновременно описаны как индивиды класса test:Equipment и одновременно имеют произвольную надпись. Обратите внимание, что пересечение работает корректно только при наличии общих переменных в “пересекаемых” шаблонах. Шаблоны, перечисленные таким образом, образуют **группу шаблонов**.

Если же нужно получить в результате все индивиды класса test:Equipment, у которых, напротив, НЕТ никакой надписи, то вводится конструкция MINUS {}, реализующая вычитание второго множества из первого:

```
...where
```

```
{
        ?equipmentInds rdf:type test:Equipment.
        MINUS {?equipmentInds rdfs:label ?indLabel.}
}
```

Для объединения множеств используется аналогичным образом UNION {}.

Интерес представляет также и команда OPTIONAL {}, которая позволяет ставить необязательные условия, при невыполнении которых значения переменных не будут исключены, поэтому здесь возможно наличие пустых ячеек в таблице, то есть, в некоторых строках переменная оказывается не привязанной к значению. Такая ситуация невозможна без использования данной команды, так как в противном случае результат считается неудовлетворяющим критериям и такая строка исключается целиком.

Отдельно скажем о запросе всех элементов списка на примере

test:disClasses1 owl:members (test:Book test:Car test:Person).

Для этого отправляется запрос следующего вида

```
...where
{
        ?x          owl:members        ?list.
        ?list       rdf:rest*/rdf:first   ?item
}
```

Следующим важным вопросом является рассмотрение задачи фильтрации результатов. Это может быть выполнено при помощи ключевого слова *FILTER*:

```
...where
{
        ?equipmentInds rdf:type test:Equipment.
        FILTER (<условие>).
}
```

При этом, фильтр всегда относится сразу ко всей группе шаблонов, поэтому строка с этим ключевым словом может располагаться произвольно внутри группы.

Условие часто бывает связано со сравнением переменных между собой (?x = ?y), с проверкой на совпадение с ресурсом или литералом (?y = test:Class2 или ?z = 24 или ?y = “Problem”), с численным сравнением (?z < 44).

В условиях часто встречаются также и иные ключевые слова.

Для символьного сравнения внутри FILTER применяют *регулярные выражения*, например FILTER(*REGEX*(?x,<регулярное выражение>,<флаги>)).

Для фильтрации только тех значений переменной, которые представлены в списке (любых из них) применяют ключевое слово *IN*: FILTER(?x IN (test:Book, test:Car)). Отрицание: NOT IN.

В некоторых случаях может представлять интерес функция *bound*(?x), причем в отрицательной форме в комбинации с OPTIONAL {}. Данная функция приобретает значение “истина”, если переменной шаблона присвоено какое-либо значение. Так, конструкция

```
...where
{
        ?equipmentInds rdf:type test:Equipment.
        OPTIONAL {?equipmentInds rdfs:label ?label }
        FILTER (!bound(?label)).
}
```

позволяет вывести такие индивиды класса test:Equipment, у которых *нет* никакой надписи (то есть, переменная ?label оказывается пустой). Таким образом, мы получаем фильтр на отсутствие.

Условие может быть сложным и включать операции && (логическое “И”), || (логическое “ИЛИ”) и “!” (логическое “НЕ”).

Последнее, чего коснемся в этом разделе – это функции-агрегаторы, производящие расчеты по всей полученной по завершении обработки запроса выборке. Рассмотрим только одну из них: count(?x), считающую в общем случае количество шаблонов триплета, удовлетворяющих условиям поиска. Так, следующий запрос считает количество индивидов класса test:Equipment.

```
select count(?equipmentInds ) as ?count
where
```

```
{
        ?equipmentInds rdf:type test:Equipment
}
```

Обратите внимание, что в этом примере между словами *select* и *count* не указано никаких переменных. Включение переменных сразу изменяет результат вычислений: при наличии переменных между *select* и *count* обработчик вычисляет, сколько раз в результатах встречается каждое из разных значений указанной переменной. Так запрос в популярном терминале http://dbpedia.org/sparql следующего вида:

```
select ?author COUNT(?author) as ?count
where
{
        ?works dbo:author ?author.
}
```

подсчитывает, сколько работ написал каждый автор из встречающихся в результатах (технически: в скольких строках результата повторяется каждый автор в шаблоне ?works dbo:author ?author).

Далее уже можно отфильтровать тех людей, кто является автором, например, не менее, чем четырех работ.

```
select ?author COUNT(?author) as ?count
where
{
        ?works dbo:author ?author.
}
Group by ?author
having (count(?author) > 3)
```

Всю остальную информацию об использовании языка запросов SPARQL можно найти на сайте w3.org.

Применение библиотеки dotNetRDF

Сказанного выше уже достаточно для создания простейшей системы управления базами знаний. При этом, очевидно, сама база знаний не может

собою управлять – требуется приложение, которое будет к ней подключаться для получения знаний и для ее изменения. Соответственно, система управления должна уметь читать и записывать файлы базы знаний в одной из нотаций SWT, должна уметь отправлять SPARQL-запросы и обрабатывать ответы на них, должна уметь добавлять / изменять / удалять триплеты и их части в составе базы знаний.

Для того, чтобы не реализовывать весь обозначенный функционал вручную, нужно воспользоваться уже готовыми многочисленными библиотеками, которые существуют для популярных языков программирования:

- для языка разработки C# в составе dotNet – библиотека dotNetRDF;
- для языка Python – RDFLib;
- для языка Java – OWL API.

Мы будем далее рассматривать взаимодействие с базами знаний при помощи библиотеки dotNetRDF. Основные принципы можно перенести на другие библиотеки и средства разработки.

Главным собирающим термином здесь является **граф базы знаний**. Граф -- это множество вершин и направленных ребер. Причем здесь, в отличие от концептуальной карты **узлами** графа считаются и индивиды, и классы, и даже сами свойства.

Все триплеты базы знаний составляют граф, поэтому работу с ней производят через класс *Graph*. Действия с базой знаний – это действия с экземпляром указанного класса, который предварительно нужно инициализировать:

```
Graph kb = new Graph().
```

Заполнять граф базы знаний можно либо при помощи считывания из файлов, либо вручную/программно добавляя в граф отдельные триплеты.

В первом случае достаточно создать индивид класса *Notation3Parser* (в случае применения нотации TURTLE), а потом воспользоваться его методом *Load*, указав адрес считываемого файла, например:

```
Notation3Parser parser = new Notation3Parser();
parser.Load(kb, @"D:\Ontology\KB\kb.n3");
```

Во втором случае нужно формировать отдельные триплеты, а затем добавлять их в граф. В этом случае отдельно создаются (или извлекаются из базы знаний при помощи запроса) узел субъекта, предиката и объекта для каждого триплета. Нужно также учитывать, что бывают **узлы-ресурсы** (URI-node), а бывают и **литеральные узлы** для литералов (Literal node). Для работы с обоими типами используется интерфейс *Inode*:

```
INode subj = kb.CreateUriNode(“test:equipment”);
INode pred = kb.CreateUriNode(“test:hasMass”);
INode obj = kb.CreateLiteralNode(500);
```

Затем все три элемента одним триплетом добавляем в базу знаний:

```
kb.Assert(subj, pred, obj);
```

Можно предварительно сгруппировать их в триплет:

```
Triple trip1 = new Triple(subj, pred, obj);
```

который можно будет добавить тем же методом: kb.Assert(trip1).

Для удаления триплета из базы можно использовать аналогичным образом метод *Retract*.

В любой момент времени можно увидеть все триплеты базы знаний при помощи свойства *Triples*, а результат можно преобразовать в список при помощи метода ToList():

```
kb.Triples.ToList(),
```

тогда каждый элемент его будет экземпляром уже известного класса Triple, поэтому можно извлекать элемент из списка:

```
Triple trip2 =kb.Triples.ToList()[2].
```

Далее можно при помощи свойств Subject, Predicate и Object получить составные элементы триплета, каждый из которых совместим с *INode*, поэтому может быть применен для формирования и добавления новых триплетов. Обратите внимание, что есть два пути работы с узлами: создание их при

помощи CreateURINode/CreateLiteralNode или же использование готовых, например:

INode pred = trip1.Predicate;

Также можно пользоваться узлами, извлеченными из запроса, поэтому следующая важная задача – это обеспечение отправки запроса к базе знаний, что также подразумевает и обработку ответа на него. Для отправки запроса применяют метод *ExecuteQuery* класса Graph, на вход которого подается строка *string* самого запроса. Для обработки результата создают экземпляр *SparqlResultSet* и явно приводят к нему результат выполнения запроса:

SparqlResultSet rezSet = (SparqlResultSet) kb.ExecuteQuery(queryString)

В данном примере rezSet является списком отдельных результатов *SparqlResult*, поэтому можно использовать индекс для доступа к конкретному результату запроса, например,

SparqlResult rez = rezSet[5].

При этом нужно помнить, что rezSet – это та таблица результатов, которую мы видим при обработке запроса, например, на терминале dbpedia.org/sparql. Тогда rez – это одна строка таблицы, количество элементов в которой (как и число столбцов в таблице rezSet) зависит исключительно от количества переменных, стоящих в запросе после ключевого слова *select*. Таким образом, rez – это тоже список, но уже конкретных значений переменных.

В Таблице 1 показаны две строки из результатов следующего запроса

```
select distinct ?ind ?indLabel
where
{
        ?ind a ?Concept.
        ?ind rdfs:label ?indLabel.
} LIMIT 100
```

Для получения значения надписи (ind) из второй строки Таблицы 1 нужно (с учетом нумерации от нуля) обратиться при помощи rezSet[1][0] или же rezSet[1] [“ind”].

Таблица 1. Часть результатов запроса

ind	indLabel
http://dbpedia.org/resource/Innovation_economics	"Инновационная экономика"@ru
http://dbpedia.org/resource/International_Society_on_General_Relativity_and_Gravitation	"Международное общество общей теории относительности и гравитации"@ru

Очень важно при этом держать в голове и тот факт, что результаты запроса можно использовать для формирование триплетов, например,

INode pred = rezSet[1][“ind”],

а также их можно использовать для отправки новых запросов, например,

```
string dynamicQuery = @“
select ?pred ?obj
where
{
        <” + rezSet[1][“ind”] + @“> ?pred ?obj.
} LIMIT 100”;
```

Обратите внимание, что данная строка запроса динамически формируется с использованием результатов прошлого запроса. Кроме того, в строках запроса (как и в коде самих баз данных на TURTLE) при передаче полного идентификатора (то есть, без префиксов – сразу адрес), сформированного программно или взятого из, например, результатов запроса, нужно использовать треугольные скобки: <urn:test:equipment> или <{ rezSet[1][“ind”] }>, где { rezSet[1][“ind”] } – это значение переменной rezSet[1][“ind”] (имеющее тип Node).

Далее мы можем перейти к следующему уровню SWT, к которому в дальнейшем будем применять уже описанный программный инструментарий.

OWL

Предыдущий уровень базовой модели SWT обладает основным недостатком, связанным с решением задачи формирования знаний (проверки на целостность и вывод новых знаний). Точнее, именно для этих задач в пространстве имен RDFS инструментов не хватает. Единственные его элементы, которые могли бы применяться для создания правил, в будущем применимых для формирования знаний: rdfs:domain, rdfs:range, rdfs:subClassOf, rdfs:subPropertyOf. При этом рассмотренные выше правила типа “Класс-класс” вполне могут быть описаны конструкцией

test:A rdfs:subClassOf test:B.

Однако, остальные типы правил требуют другие выразительные инструменты для своего описания.

И для работы с ними нам потребуется более высокий уровень базовой модели SWT – уровень пространства имен Web Ontology Language (OWL), физический адрес которого можно также найти в любом из общедоступных списков. Если зайти внутрь него, то можно увидеть, что он содержит множество элементов.

Здесь сразу же нужно сделать важное пояснение: сам по себе язык OWL – это лишь общепринятый способ, стандартизированная форма представления правил. OWL не обеспечивает обработчиков этих правил, в нем самом по себе нет готовых программных средств для формирования знаний. Другое дело, что можно при желании найти в общем доступе пакеты, которые обеспечивают некоторую обработку – но это уже самостоятельные продукты.

Мы же будем для более глубокого понимания процесса работы машины вывода разрабатывать такие продукты самостоятельно, что также позволит и заниматься разработкой “под ключ”. Поэтому нужно изучить элементы пространства имен OWL и понять, как они могут применяться.

И первым среди них, с которого мы и начнем, является класс **owl:Class**, который является подклассом уже известного rdfs:Class, но подразумевает, что данные классы будут применяться для логических операций, которые должны

выполняться в рамках логических обработчиков (в машинах вывода) и должны служить целям формирования знаний: решению двух известных его подзадач.

В соответствии с этими подзадачами разделим элементы на категории, которые и рассмотрим по отдельности.

OWL-элементы для обнаружения противоречий

Противоречия (или нарушения целостности, **inconsistencies**) в базах знаний могут быть выражены в наличии одного или нескольких фактов, противоречащих одному из прописанных в базе правил (нарушающих их). Следовательно, для выявления противоречий нужно сначала создать и описать все те правила, которые регламентируют невозможные комбинации содержаний базы знаний.

Первым элементом в списке используемых для формулирования таких правил можно назвать свойство **owl:oneOf**, которое накладывает ограничение на возможные экземпляры конкретного класса, указывая их конечный список:

```
test:CurrentlyUsed   a           owl:Class;
                     owl:oneOf   ( test:Arm2  test:MCU6  ).
```

Соответственно, применяя правила с owl:oneOf для выявления противоречий в базе знаний, система должна сначала запросить все классы, описанные с использованием этого элемента (?x owl:oneOf ?y), а затем проверить индивиды, принадлежащие полученным классам, на наличие не входящих в ограниченный список, представленный в правиле.

Следующий элемент – **owl:disjointWith**, позволяющий прямо указать пару классов, которым никакой индивид не может принадлежать одновременно, иначе – противоречие:

```
test:Car owl:disjointWith test:Book.
```

Однако, таким образом неудобно указывать сразу целый список взаимно непересекающихся классов, поэтому во избежание прописывания множества пар, достаточно указать сразу список с использованием элемента **owl:AllDisjointClasses**.

```
[]      a                owl:AllDisjointClasses;
        owl:members      ( test:Book test:Car test:Person ).
```

Соответственно, если есть несколько таких списков, то вместо пустого узла нужно использовать либо индивид, либо именованный пустой узел.

Существует также понятие непересекающихся свойств, что описывается при помощи **owl:propertyDisjointWith**. Семантический смысл состоит в том, что два индивида не могут быть связаны между собой одновременно несколькими из списка непересекающихся свойств:

```
test:hasSon owl:propertyDisjointWith test:hasDaughter.
```

Здесь же нужно упомянуть указание типов свойств, которые {указания} сами по себе являются уже правилами. Например, если свойств является нерефлексивным (**owl:IrreflexiveProperty**), то никакой индивид не может быть в связан этим свойством с самим собой (это – уже правило). А если свойство является асимметричным (**owl:AsymmetricProperty**), то никакие два индивида не могут быть связаны друг с другом этим свойством “в обе стороны” (то есть, невозможна ситуация одновременного выполнения aRb и bRa).

Существует также и класс **owl:NegativePropertyAssertion** для гибкого обозначения недопустимости соединения конкретных элементов в триплет субъект-предикат-объект, например:

```
_:np1   a                       owl:NegativePropertyAssertion;
        owl:sourceIndividual    test:Arm1;
        owl:assertionProperty   test:inOrderTo;
        owl:targetIndividual    test:heavyMassTransport.
```

Для аналогичного использования литерального свойства (в отличие от объектного), применяют owl:targetValue (вместо owl:targetIndividual).

Как и в случае с rdf:Statement, такие триплеты могут быть представлены одним узлом (в данном примере _:np1) и могут быть активированы или деактивированы.

OWL-элементы для вывода новых знаний

Легко представить, как уже известные элементы rdfs:domain и rdfs:range могут быть использованы для вывода новых суждений: если в базе знаний найдено свойство, у которого указан domain или range, то находим все триплеты, где оно выступает предикатом и выводим новое суждение о том, что все его, соответственно, субъекты или объекты принадлежат указанным классам.

В этой же логике должна работать и машина вывода в рамках элементов пространства имен OWL: сначала ищем правило, потом суждение-предпосылку (одно или несколько), и на их основе создаем суждение-вывод.

Полезными для этой задачи являются типы свойств, такие как **owl:SymmetricProperty** и **owl:TransitiveProperty**. Если свойство R является симметричным, то из факта симметричности и связи *a*R*b* следует вывод-суждение *b*R*a*. Из транзитивности свойства R и наличия связей *a*R*b*, *b*R*c* следует вывод *a*R*c*.

По этой же логике работает и свойство **owl:inverseOf**, при помощи которого можно указать, что два свойства являются взаимно обратными. Из обратности свойств R и S и из суждения *a*R*b* следует вывод-суждение *b*S*a* – и наоборот.

Для работы с также рассмотренной композицией свойств применяется элемент owl:propertyChainAxiom, показывающий, композицией которых свойств является данное свойство:

```
test:grandParent    rdf:type                owl:ObjectProperty;
                    owl:propertyChainAxiom  ( test:parent test:parent ).
```

Для описания классов часто используют **owl:unionOf** / **owl:intersectionOf**, применяемые для того, чтобы выразить следующую аксиому: если индивид принадлежит, соответственно, любому одному / всем классам из списка, то он также принадлежит и указанному классу, например:

```
test:FragileEquipment owl:unionOf ( test:Sensor test:Microcontroller ).
```

Здесь критически важным является то, что список – это предпосылка, а указанный в качестве субъекта класс – вывод, но никак не наоборот.

Для указания пары дополняющих друг друга классов применяют свойство **owl:complementOf** в качестве предиката (в логике “если индивид не принадлежит первому классу, то принадлежит второму”).

Для указания на эквивалентность классов друг другу (или для связывания класса и *ограничения*) можно применить **owl:equivalentClass**. Это особенно важно в случае подключения баз знаний, в которых идентификаторы ресурсов представлены в виде кодов, как в примере

test:Robot owl:equivalentClass <https://www.wikidata.org/wiki/Q11012>.

Аналогичным образом, но уже по отношению к индивидам, используется свойство **owl:sameAs**.

Однако, все рассмотренные правила являются все же довольно простыми по форме и не позволяют формировать более сложных конструкций. В этой связи нужно далее рассмотреть возможности применения OWL для описания правил уже рассмотренной дескрипционной логики.

Описание дескрипционной логики

Модификация OWL 2 включает все необходимые для описания дескрипционной логики элементы.

Для обозначения пустого класса $\perp$ применяют owl:Nothing, а для обозначения класса-Универсума T – owl:Thing.

Остальные элементы связаны уже с характерным для дескрипционной логики понятием *ограничение*, которое можно определить как множество индивидов, удовлетворяющих некоторому условию. Последнее всегда связано со свойствами индивида, выступающими в роли его предикатов.

Вспомните, что такое терминологическая аксиома и каких двух видов они бывают. На этом этапе важно понять следующее: в выражениях $C \equiv D$ и $C \subseteq D$ в качестве C и D могут выступать как классы (то есть, их конкретные URI), так и ограничения, которые обычно записывают в виде пустых узлов, хотя они могут быть представлены и как именованные индивиды класса **owl:Restriction** с собственным идентификатором ресурса.

Ограничения всегда связаны со значениями свойств, так что ограничения классифицируются по характеру условий, накладываемых на эти значения.

Самые простые ограничения формулируются так: “множество всех индивидов, у которых значение свойства R равно …”. Значение свойства R (оно же – объект по предикату, выраженному свойством R) в рамках дескрипционной логики часто называют **R-последователем** (R-successor). Поэтому это же ограничение можно переформулировать как “множество всех индивидов, у которых R-последователем является …”.

Рассмотрим пример правила (терминологической аксиомы), содержащего такое ограничение:

```
test:RoboticArm a                    owl:Class;
    owl:equivalentClass     [        a                  owl:Restriction;
                                     owl:onProperty     test:hasPart;
                                     owl:hasValue       test:endEffector ].
```

Оно относится к типу $C \equiv D$, причем в качестве D выступает ограничение, описанное в данном случае как пустой узел. Ключевым здесь является использование **owl:hasValue**: это задает характер ограничения, в то время как свойство **owl:onProperty** используется во всех ограничениях.

Важно также учитывать, что в случае с эквивалентностью классов (owl:equivalentClass) правило должно работать в обе стороны, что означает, в данном случае, сразу два правила:

1. любой индивид, у которого последователем по свойству test:hasPart является test:endEffector, принадлежит классу test:RoboticArm;
2. если индивид принадлежит классу test:RoboticArm, то у него обязательно должен существовать последователь test:endEffector по свойству test:hasPart.

Оба из них могут выступать как правила для вывода новых знаний и как правила для проверки на наличие противоречий.

В случае же использования rdfs:subClassOf (что соответствует операции включения), такой двоякости не возникает – это приводит к одному правилу: на меньшее (включаемое) распространяется описание большего (включающего). Обратное неверно.

Для описания *универсальных ограничений* применяют элемент **owl:allValuesFrom**:

```
test:RoboticArm          a                    owl:Class;
        rdfs:subClassOf [ a                    owl:Restriction;
                          owl:onProperty       test:hasPart;
                          owl:allValuesFrom    test:RoboticComponent ].
```

Еще раз обратим внимание, что ограничение может выступать как в роли объекта, так и в роли субъекта терминологической аксиомы – или в обеих ролях одновременно в одном триплете.

Для описания *экзистенциальных ограничений* применяют элемент **owl:someValuesFrom**:

```
[    a                    owl:Restriction;
     owl:onProperty       test:madeByWorker;
     owl:someValuesFrom   test:BestWorkers]
                              rdfs:subClassOf   test:HighQualityProduction.
```

Здесь особые комментарии не требуются, кроме лишь того, что если при использовании rdfs:subClassOf экзистенциальное ограничение выступает в роли объекта терминологической аксиомы, то это правило может использоваться только для проверки на наличие противоречий.

OWL позволяет также описывать и рассмотренные ранее *ограничения кардинальности*. Разница лишь в употреблении одного из элементов: **owl:cardinality**, **owl:minCardinality** или **owl:maxCardinality** для указания на точное значение количества R-последователей любого класса:

```
[    a                    owl:Restriction;
     owl:onProperty       . . . ;
     owl:minCardinality   2                ]    rdfs:subClassOf   . . .
```

В случае же, если требуется описать ограничение вида “множество индивидов, у которых существует {n} R-последователей, принадлежащих конкретному классу C” (то есть, Qualified restriction), то нужно использовать **owl:onClass** для фиксации класса, а также **owl:qualifiedCardinality**, **owl:minQualifiedCardinality** или **owl:maxQualifiedCardinality** вместо рассмотренных в предыдущем абзаце соответственно:

```
[   a                          owl:Restriction;
    owl:onClass                . . . ;
    owl:onProperty             . . . ;
    owl: minQualifiedCardinality  5          ]   rdfs:subClassOf  . . .
```

На этом основные подходы к формализации знаний можно считать рассмотренными.

Динамические правила

Однако, подлинная интеллектуальность систем появляется не ранее, чем последние начинают не просто использовать правила для вывода новых фактов (или проверки существующих), а начинают применять правила для вывода или удаления других правил. Не просто правила для работы с фактами, а именно правила для работы с правилами для работы с фактами.

Основной подход, который мы будем использовать — это активация / деактивация правил. Идея состоит в том, чтобы последние хранились в базе знаний, но в такой форме, которая позволяет явным образом указать на статус всего правила (активно оно или нет). Следовательно, нам потребуется такая форма записи, в которой правило целиком “умещается” в один узел.

И с такой формой записи мы уже встречались, когда рассматривали пространство имен rdf и в нем класс Statement, в рамках экземпляра которого хранится сразу целый триплет. Он как бы и есть в базе знаний, но при этом по форме записи отличается от всех остальных триплетов.

Так, можно взять любое правило, рассмотренное в предыдущем разделе (а оно всегда представляется в форме триплета!), и преобразовать его в специальный вид, например:

```
test:rule5      a                   test:Rule, rdf:Statement;
                rdf:subject         test:Car;
                rdf:predicate       owl:disjointWith;
                rdf:object          test:Book;
                test:ruleStatus     “deactivated”.
```

На месте любого или двух компонентов правила может быть расположено ограничение.

Затем нужно проработать принципы активации правил. Последние могут быть активированы или деактивированы при наступлении события. В роли события могут выступать:

– получение сигнала от датчика / информационной системы либо получение команды от специального агента;
– вычисление определенных результатов сравнения некоторых значений между собой;
– активация одного или нескольких правил либо фактов (например, полученных в виде списка активированных экземпляров rdf:Statement).

В первом случае подразумевается, что информационная система выдает не само число (например, значение конкретного датчика или выхода нейронной сети), а уже полученную на основе аналитической обработки оценку в строковом виде, например: “Резкое повышение давления”.

Тогда само правило активации правила представим в виде

```
[   a                  test:RuleActivationReason;
    test:keyText       “Резкое повышение давления”;
    test:activatesRule test:rule5     ].
```

Теперь для активации всех правил по правилам такого шаблона нужно реализовать обработчик, который отправляет запрос вида “какие правила активирует ситуация резкого повышения давления?”, а все найденные правила переводит в состояние “Активирован”.

Если активатором правила служит некоторое единичное условие (например, активация определенного состояния (как в рассмотренном ранее примером с классом test:State) или активация конкретного правила (нахождение последнего в состоянии “Активирован”), то можно использовать простой триплет вида

. . . activatesRule . . . .

Его субъект – это предпосылка для активации, а объект – активируемый элемент.

В случае же комбинации условий вместо пустого узла нужно использовать, например, коллекцию, в которой каждый элемент представлен либо правилом, либо текущим состоянием, выраженным активированным экземпляром класса rdf:Statement (как в рассмотренном ранее примером с классом test:State):

(. . . . .) a test:RuleActivationReason.

Подразумевается, что все элементы коллекции-субъекта являются активированными – это и будет необходимой предпосылкой.

Для реализации других способов обучения потребуется дополнительный инструментарий, который рассмотрим позднее. Но прежде нужно обратить внимание, что обучение системы – это только одно из средств решения отдельного класса задач, а именно: задач с неопределенностью. Это понятие далее и рассмотрим.

Неопределенность

Особенность интеллектуальных систем состоит в том, что они способны решать задачи с существенной **неопределенностью**. Последнее, конечно, означает некоторую нехватку данных или знаний для дальнейших действий.

Однако, нужно четко понимать, что, например, если нам заранее неизвестно, какие именно значения температуры будет выдавать соответствующий датчик, то это само по себе еще не рассматривается как неопределенность. Если в систему поступают четкие показания датчика и есть знание о том, по которому из имеющихся в системе алгоритмов нужно действовать для любых возможных диапазонов температур, то ничто и никак не мешает системе принимать решения, поэтому к неопределенности эта ситуация отношения не имеет.

А вот если, в этом же примере, мы хотим, чтобы система продолжала верно работать даже в ситуации, когда датчики по неопределенной причине в любой момент времени перестали обеспечивать ее информацией, то речь уже идет о **неопределенности ситуации**, характеризующейся недостаточностью сведений о текущем состоянии дел (нет сведений от информационно-измерительной системы или же эти сведения нечитаемы); то есть даже если на борту системы имеется подходящий алгоритм, то для его запуска и начала действий все равно не хватает входных параметров (так как они неизвестны). Другим примером является ситуация с обнаружением неопознанного объекта, по которому, тем не менее, система обязана принять решение.

Но при всем том, интеллектуальная система не должна прекращать работу с ошибкой или уведомлением о недостатке данных (хотя уведомление, конечно, следует отправлять). Напротив, она должна в этом случае предпринять некоторые специальные действия: либо получить эти данные другим способом / из другого источника, либо получить другие (компенсирующие) данные, либо принять решение при отсутствии данных, но это решение не должно нарушить ход всего процесса и привести к нежелательным последствиям. Очевидно, что если во всех неопределенных ситуациях система принимает одно и то же решение (если она запрограммирована именно на это действие), то это говорит об отсутствии у нее какой-либо интеллектуальности. Интеллектуальная система же должна анализировать свой или чужой опыт, извлекать из него закономерности, совершать максимально безопасные тестовые действия, анализировать их результаты и ошибки и т. д.

Важно понять – интеллектуальная система должна создавать свой алгоритм (по крайней мере, его часть) в процессе своей работы.

Следующий тип – **неопределенность в алгоритмах действия**. Речь идет об отсутствии готового решения для реализации цели системы, даже если все необходимые входные данные приведены. Например, если известно, что от пользователя или вышестоящего агента (программы или устройства) в любой момент времени может поступить любая команда (в понятном для системы виде), при этом доступны любые требуемые данные о ситуации, однако, у системы может не быть заранее приготовленного способа исполнения полученной команды (с уровнем роста произвольности ее содержания риск отсутствия готового алгоритма растет). Другой пример: система управления автомобилем обнаружила на дороге некоторый объект. Все его характеристики известны, однако его параметры не соответствуют ожиданиям системы (например, агрессивное поведение). Сюда же относится ситуация с непредсказуемым изменением самой цели для системы.

При этом, опять же, у системы нет права ответить “ошибкой” и остановиться: она обязана найти решение, совершив для этого определенные действия, подходящие именно для этой конкретной ситуации.

Третий случай неопределенности – это **неопределенность в последствиях**. Это означает, что даже если известно, который алгоритм следует применить, то результаты его применения могут быть не такими, как того ожидает система. Речь может идти об изменении исполнительного или иного оборудования, о смене объекта управления.

При этом, как было сказано и ранее, интеллектуальная система должна быть готова обработать любое возникшее от ее действий последствие и в случае расхождения результата и ожидания – скорректировать свои модели и / или алгоритмы.

По факту, данное разделение не является строгим, так как указанные виды неопределенности взаимосвязаны, часто одна следует из другой.

Рассмотрим далее базовое устройство системы, потенциально способной к разрешению указанных неопределенностей.

Компоненты системы на основе знаний

Понятно, что для обладания интеллектуальностью программный комплекс не может состоять исключительно из семантической базы знаний и системы управления ею. Это связано с отсутствием математического аппарата в технологии SWT, что заметно усложняет хранение и обработку математических зависимостей и даже выполнение простых вычислительных операций с поступающими данными. База знаний, например, может содержать информацию о том, что нужно делать в случае резкого изменения контролируемого параметра, но SWT не помогут в отслеживании самого факта этого изменения. Кроме того, мы бы были лишены важнейшего источника новых правил и фактов – возможности анализировать имеющийся или новый опыт, архивные и оперативные данные о работе системы.

Таким образом, интеллектуальная система на основе знаний (СОЗ) необходимо должна включать в себя не только систему управления базами знаний, но еще и другие компоненты (Рис. 10).

Рис. 10. Компоненты системы на основе знаний

Рассмотрим компоненты по отдельности.

Информационно-измерительная система

В первую очередь следует упомянуть источник любых оперативных данных – информационно-измерительную систему. Под этим термином будем понимать совокупность всех датчиков, всех подключенных устройств и агентов, от которых можно принимать информацию. Иными словами, любые данные о внешней среде в систему на основе знаний поступают через информационно-измерительную систему.

В качестве выходной точки этой системы можно рассматривать совокупность файлов, в которые система записывает оперативные данные. В этом случае остальные компоненты могут обращаться к таким файлам с доступом на чтение

для получения из них данных с последующей отправкой их, например, в базу данных или в систему мониторинга.

Компонент управления формулами

Его назначение состоит в хранении математических (в том числе логических) зависимостей, а также в выполнении их вызовов для расчетов.

Главным отличием данного компонента является, во-первых, такое хранение произвольных аналитических зависимостей, при котором последние могут быть модифицированы прямо во время исполнения основного процесса системы, а, во-вторых, возможность вызова требующейся функции по ее уникальному идентификатору (хранимому также в базе знаний).

Первая особенность объясняется тем, что интеллектуальность системы предполагает возможность обучаться – в том числе речь идет и о возможности уточнять математические модели во время работы. Вторая же связана с возможностью применения различных формул для вычисления одних и тех же параметров в зависимости от ситуации -- при этом, раз речь идет об обучении, то заранее неизвестно, в которой из ситуаций будет применяться та или иная формула.

Для практической реализации указанных возможностей можно применять, например, динамические языки программирования (типа Python, в котором интерпретация файлов с кодом происходит “на лету”), на которых и должны быть описаны математические функции для вычислений. В случае использования для разработки этого модуля среды Visual Studio и языка C#, мы использовали средство IronPython, позволяющее вызывать из основного кода на C# функции на Python, даже если последние были модифицированы уже после запуска основной программы.

Это же средство помогло реализовать и вторую особенность, так как позволяет вызывать функцию пусть и с заранее фиксированным именем, но из произвольного файла, имя которого указывается при помощи переменной.

В базе знаний же, соответственно, описывается для каждой формулы идентификатор функции, в которой она реализована, и условия ее применимости (например, для расчета которой величины требуется формула и какие входные параметры для этого требуются), для чего у индивидов класса “Формула” можно использовать свойства “идентификатор”, “выходная величина”, “входной параметр”.

Для непосредственного применения произвольной формулы в *компоненте управления формулами* должна быть предусмотрена функция, входными параметрами которой будут все указанные свойства, которая будет использовать, например, опять же, IronPython для выполнения расчета и которая будет возвращать полученное число либо его качественную оценку, например, “Превышение допустимого значения”.

Так как данный компонент должен обеспечивать возможность оперативного изменения формул, то необходимо решить вопрос с его технической реализацией.

И важный момент здесь следующий: в отличие от компонентов анализатора опыта и управления нейронными сетями, данный компонент не рассчитан на проведение процедур идентификации, аппроксимации, интерполяции и подобных. Вместо этого, данный компонент запрашивает уже созданные формулы различным образом:

- запрос у эксперта – через пользовательский интерфейс;
- импорт из файлов проектов;
- импорт из сторонних CAD-программ с применением стандарта Component Object Model (COM) как показано в [Onufriev et al, 2019].

В случае импорта потребуется разрабатывать специальные модули сопряжения, задача которых – сохранить формулы в подходящей для использования данным компонентом форме.

Компонент управления нейросетями

Человек как интеллектуальный агент при решении задач пользуется способностью распознавать объекты вокруг себя и на изображениях, формы линий на графиках, паттерны расположений элементов и т.д. И так как зачастую интеллектуальная система тоже должна решать эти задачи, а чаще всего для этого применяются нейронные сети, то в состав интеллектуальных систем также должен входить **компонент управления нейросетями**.

Помня, что нейронные сети способны решать множество задач, в том числе принимать решения о поведении сложных систем, рекомендуется все же в рамках систем на основе знаний делегировать нейронным сетям, напротив, самые простые задачи, такие как:

- обнаружение наличия (или отсутствия) определенного объекта в видеопотоке;
- фиксация факта быстрого (или медленного) возрастания (или убывания) определенного сигнала;
- обнаружение попадания значения сигнала (или группы сигналов) в определенный(-е) диапазон(-ы) значений;
- фиксация сигнала (или комбинации сигналов) определенной формы: острый пик, парабола, экстремум и др.

Во всех указанных случаях речь идет только об обнаружении, без принятия каких-либо решений, так как чем больше функций требуют от конкретной нейронной сети, чем больше она задач решает одновременно, тем сложнее ее структура и обучение, тем больше возможность ошибки в ее работе и настройке.

В этой связи *компонент управления нейросетями* должен содержать множество максимально простых нейронных сетей, которые лишь являются источником предобработанных данных, для последующего принятия решений другими компонентами.

Стоит учесть, что некоторые сети обучаются во время работы (и в этом случае данный компонент отвечает за непосредственное исполнение алгоритмов обучения), а другие формируются по типу нейронных сетей на основе знаний (KBNN, KBANN).

Система мониторинга

Система мониторинга применяется для отслеживания разного рода изменений, происходящих с потоком оперативных данных. Эта система может использовать одновременно нейронные сети, формулы и другие средства вычислений. Результат должен быть отправлен в систему управления базами знаний для последующей обработки.

Однако, нейронные сети, как и формулы, по умолчанию на выходе возвращают численные значения. Если все эти данные сразу неконтролируемо отправлять в СУБЗ, то мы получим множество запросов, которые, по факту, могут быть повторяющимися и/или не требующими никаких действий. В этой связи система мониторинга должна отправлять в СУБЗ запросы только в ситуациях обнаружения определенных признаков, а также определять частоту этих запросов.

С другой стороны, SPARQL-запрос к БЗ, включающий в себя сравнение с числовым значением, которое в базе знаний привязано к определенному показателю, имеет бóльшую длину и будет дольше исполняться, чем запрос к состоянию, обозначенному строкой. При этом, структура этого запроса относительно проста и описана выше.

Система мониторинга должна отвечать не только за обработку результатов математического анализа данных, но и за формирование самих средств обработки. При этом в качестве последних могут выступать не только нейронные сети, но и любые алгоритмы обработки информации, причем эти алгоритмы должны быть программным образом выводимы из хранимой в базе знаний информации. Это возможно сделать, например, путем подстановки

запрошенных из базы знаний критических значений в шаблоны Python-функций либо в файлы, в которых хранятся коэффициенты нейросетей.

Система мониторинга использует для своих задач нейросети и/или формулы, из чего следует необходимость реализации взаимодействия соответствующих компонентов.

Кроме того, система мониторинга поставляет другому компоненту – анализатору опыта – информацию о факте отклонения некоторых показателей от нормы, что, фактически, предписывает ей функционал по отслеживанию необходимости запуска процесса коррекции существующих моделей.

Системы управления базами данных и базами знаний

Следующим компонентом является **система управления базами данных**. Последние применяются для хранения текущих показателей управляемого процесса, которые затем могут быть использованы для анализа. Речь при этом идет не только о технических показателях нижнего уровня типа температуры, давления и концентрации газов, но и о показателях эффективности более высокого уровня, от эффективности работы условного “цеха” до стратегических показателей целого предприятия или отрасли. СУБД должна обеспечивать возможность оперативной записи новых данных в условиях, когда заранее не понятно, сколько полей данных (столбцов) потребуется хранить – и это число может быть переменным.

Базы знаний же не применяются для архивации информации, вместо этого они содержат:

- цели управления и их индикаторы;
- информацию о последовательности необходимых действий в ответ на поступающие из нейронных сетей и формул качественные оценки ситуаций (вида “Резкое повышение давления”);
- информацию о том, для расчетов каких параметров требуется которая из формул / нейронных сетей (идентификатор функции-обработчика) и какие входные параметры для этого требуются;

- извлеченную из опыта информацию, например, о том, какие действия при каких условиях чаще всего приводили к правильному результату;
- информацию об управляемых процессах, выделяемых нормах времени, подпроцессах и об их последовательности, об их показателях эффективности и о возможных исполнителях каждого из них;
- информацию об агентах (и работники, и оборудование), которые могут быть задействованы в процессах, об их состоянии, об их взаимозаменяемости, о критериях выбора исполнителя;
- информацию о максимально возможной производительности агентов (для распределения нагрузки);
- последовательность действий при взаимодействии агентов в рамках решения общих или связанных задач;
- правила для логического вывода и проверки на непротиворечивость.

Таким образом, **система управления базами знаний** хранит и генерирует информацию о том, что и в какой последовательности нужно предпринимать в соответствии с целями.

Далее представлен неполный список получаемого материала на выходе СУБЗ при взаимодействии с ней:

- список управляющих воздействий, которые нужно совершить для улучшения показателей эффективности;
- список оценок состояния системы в целом;
- список предполагаемых причин / последствий данной ситуации;
- идентификатор формулы / нейронной сети для расчета конкретной величины;
- список исполнителей, которые могут быть направлены на решение задачи, а также соответствие их ресурсов выдвигаемым требованиям и т. д.

Здесь важно учитывать взаимодействие с формулами и нейронными сетями: система управления базами знаний должна принимать от них сгенерированные сообщения об оценке ситуации, что означает хранение в ней списка таких сообщений, чтобы можно было отправлять SPARQL-запрос, содержащий в себе соответствующую информацию.

Более того, в случае задачи слежения за отклонением показателей эффективности процессов от установленных норм, математический аппарат должен формироваться автоматически на основании уставок, хранящихся в БЗ. Если применяются программные методы для сравнения значений, то они должны производить сравнения с полученными из СУБД уставками. Если применяются нейронные сети, то они не требуют обучения – это вопрос вычисления “весов” по легко выводимым на основании требуемых уставок формулам.

Компонент анализатора опыта

Если система собирает достаточно данных о работе, то можно говорить уже о работе другого модуля: **компонента анализатора опыта**. В первую очередь речь идет об обнаружении каких-либо закономерностей в накопленных данных, об определении связей (математических моделей) между различными показателями. Для решения таких задач данный компонент должен привлекать статистический аппарат, а также обучать собственные нейронные сети и применять другие технологии Data Mining’a.

При реализации данного компонента требуется ответить на следующие вопросы:

- какое событие может стать достаточным основанием для запуска анализа опыта?
- которые именно данные из всех имеющихся нужно начинать анализировать?
- на предмет чего их нужно начинать анализировать?

Для одновременного решения указанных вопросов предлагается применять уже упомянутую систему мониторинга, которая предназначается как раз для слежения за состоянием сигналов и в этом случае может взаимодействовать с анализатором опыта и возбуждать процесс поиска новых знаний на основе полученных данных.

Первым индикатором, на который должна реагировать данная система, является факт отклонения одного или нескольких показателей эффективности от заданного заранее (или ранее вычисленного) значения. Это в общем случае *может* означать (и поэтому это требует проверки), что общая модель, по которой мы рассчитали требуемое управляющее воздействие для обеспечения желаемого показателя, работает неверно. Обратите внимание, что речь необязательно идет о неверной *математической* зависимости, ведь можно говорить также и об установлении неверной причинно-следственной связи между совершаемыми действиями и ожидаемым от них результатом. Более общий случай запуска анализатора – это получение неожиданного результата на выходе (даже в случае его положительности).

Анализироваться при рассмотренном основании для запуска процесса вычисления моделей должны полученные на опыте связи между желаемыми результатами (показателями) и входными воздействиями.

Второе общее основание для запуска вычислений – факт обнаружения определенной повторяемости. Это во многом соответствует логике обучения с подкреплением (которое по своей сути является дрессировкой), когда после нескольких повторений одни и те же действия приводят к одному и тому же результату, что позволяет создать правило “если сделать А, то результатом может стать В”. Для реализации такого анализа в базе данных должны содержаться, помимо прочих, и сведения о действии и об их последствиях.

Таким образом, система мониторинга должна регулярно обращаться к базе данных и, обрабатывая постоянно записываемый в нее поток оперативных

данных, следить за состоянием ключевых показателей и периодически отслеживать повторяемость причинно-следственных связей.

При обнаружении отклонений или повторяемости должен быть активирован компонент анализатора опыта, уже непосредственно производящий вычисления. В случае отклонения показателей он должен реализовывать алгоритмы идентификации (в том числе – нейросетевой), а в случае упомянутой повторяемости – алгоритмы статистического анализа.

Следует обратить внимание, что нейронные сети применяются в рамках систем на основе знаний в двух режимах:

1) нейронная сеть постоянно проверяет входящий поток оперативных или архивных данных;
2) нейронной сети при происшествии одного из обозначенных ранее событий выполняет аппроксимацию данных для вычисления новой модели.

В первом случае нейронная сеть должна до своего использования быть подготовленной для обнаружения требуемых / нежелательных элементов или закономерностей. Также возможно и ее переобучение уже во время работы. Этот режим может применяться для мониторинга ситуации на управляемом объекте для контроля текущего состояния.

Во втором случае получение моделей достигается за счет запуска процесса обучения сети на имеющихся данных. Важно то, что и запуск обучения, и достижение его результатов должно контролироваться самой системой при минимальном участии человека. Таким образом, здесь встает проблема оперативного неконтролируемого обучения.

Оговоримся отдельно о том, что основанием для запуска процесса анализа опыта может служить также и сигнал компонента экспериментов (рассмотрен ниже), выполнившего эксперимент и сгенерировавшего новый опыт для анализа.

Интерфейс пользователя

Данный компонент нужен для взаимодействия с пользователем системы. При этом, следует оговориться, что пользователя у системы на основе знаний может вообще не быть, потому как она может изначально проектироваться как полностью автономная.

Интерфейс пользователя служит для получения данных от пользователя или для визуализации текущей ситуации для оператора / ответственного лица.

На вход от пользователя данный компонент может принимать:

- команды для исполнения;
- знания;
- суждения на естественном языке (например, в диалоговых системах);
- файлы (в том числе и графические) для обработки.

На выход для пользователя интерфейс может выдавать:

- предупреждения, уведомления;
- рекомендации;
- численную статистику;
- графики и диаграммы, визуализирующие процесс;
- знания в структурированной форме.

В случае работы с технологическими процессами данный компонент может выступать как так называемый человеко-машинный интерфейс (HMI).

Система автоматического управления и диспетчеризации

Если СОЗ выполняет не только роль системы поддержки принятия решений, но и участвует в управлении теми или иными процессами, то для этого в ней должен быть представлен компонент, непосредственно взаимодействующий с управляемыми объектами. Фактически, система автоматического управления и диспетчеризации (САУиД) требуется для конвертации математических и алгоритмических моделей в физические управляющие сигналы или команды.

Управляющие воздействия могут отправляться

- на программируемые логические контроллеры для корректировки процессов, в которых они участвуют;
- агентам-исполнителям, в роли которых могут выступать как люди (рабочие, пользователи и т.д.), так программы (симуляторы, игровые агенты и т.д.) и устройства (двигатели, нагреватели и т.д.);
- другим системам управления (например, более низкого уровня) или другим системам на основе знаний.

Система автоматического управления и диспетчеризации в общем случае включает в себя управляемый объект (или объекты), набор устройств для отправки управляющих воздействий (устройства сопряжения с объектом), вычислители (реализованные на базе программируемых логических контроллеров, микроконтроллерных / микропроцессорных систем и иных устройств), набор средств для считывания сигналов с объекта.

Компонент экспериментов

Если в ситуации неопределенности системе неоткуда взять данные / знания для дальнейших действий, то остается только совершать некоторые шаги экспериментального характера. Речь идет именно о ситуации создания нового опыта (в отличие от действий компонента анализатора опыта, который работает с уже существующими к этому моменту данными), то есть, производство пробного действия с анализом полученных результатов.

Основная сложность состоит, в основном, в вероятной опасности экспериментального действия, связанной с опасностью неопознанного объекта, с риском повреждения оборудования при неправильном действии, с возможным отсутствием доступа к объекту (закрытостью объекта).

Кроме того, эксперименты должны производиться в соответствии с определенной системностью, с некоторыми правилами планирования и проведения эксперимента, так как задачей является получение систематических, а не случайных результатов.

В связи с этим, задача данного компонента заключается в определении целей и условий эксперимента, в его непосредственном планировании, в получении экспериментальных данных (с их предварительной проверкой) для последующей передачи на обработку. Правила проведения экспериментов, по которым работает данный компонент, могут храниться в базах знаний.

Стоит отдельно обратить внимание на то, что задачи планирования и проведения эксперимента непосредственно связаны с задачами **порождения гипотез** и **проверки гипотез**. Каждая из этих задач являются довольно сложными и требуют отдельного рассмотрения. Но в рамках данного пособия в качестве реализации одного из подходов для решения первой из них мы лишь укажем ДСМ-метод [Гаврилова, Кудрявцев, Муромцев, 2016] и методы обучения ассоциативным правилам (Association rule learning). В качестве же подхода к решению второй можно рассмотреть табло-алгоритм (Tableau algorithm) [Золин, 2018].

Табло-алгоритм применяется для доказательства либо опровержения высказываний, представленных на одном из языков логик. Его модификации существуют для рассмотренных здесь языков: логики высказываний, логики предикатов и дескрипционной логики.

Подход прост: чтобы доказать утверждение нужно показать противоречивость опровергающего его утверждения. Иначе говоря, если нужно при помощи данного подхода доказать $[\ (a \rightarrow b) \wedge b\ ] \rightarrow a$, то требуется показать, что $\neg\ \{\ [\ (a \rightarrow b) \wedge b\ ] \rightarrow a\ \}$ является противоречием, несостоятельным утверждением. А для доказательства (уже из дескрипционной логики), что $\exists R.A \sqcup \neg \forall R.(\neg B \sqcup A) \sqsubseteq \exists R.B$, нужно доказать противоречивость утверждения $\neg\ \{\ \exists R.A \sqcup \neg \forall R.(\neg B \sqcup A) \sqsubseteq \exists R.B\ \}$.

Дальнейшие шаги являются принципиально похожими во всех трех языках логик: логические утверждения преобразуются в ветви (которые и называются “табло”), что дают расширяющуюся ветвистую структуру. Далее к каждой из ветвей применяют правила (связанные с раскрытием конъюнкции, дизъюнкции,

раскрытием ограничений и т.д.), по которым эти ветви либо продлеваются, либо закрываются, либо остаются без изменений. Если к моменту, когда дальнейшие шаги алгоритма, преобразующие ветви, невозможны, то либо все ветви оказываются закрыты (“заблокированы”), либо некоторые остаются незакрытыми. В первом случае выдвинутое утверждение, отрицающее первоначальное, считается противоречивым, а исходное – доказанным. Во втором же случае исходное утверждение считается не обладающим всеобщностью, поэтому опровергается.

Конечно, решение задач порождения гипотез и проверки гипотез не исчерпываются ДСМ-методом и табло-алгоритмом, однако знакомство с ними поможет хотя бы схематично представить подходы к решению указанных задач.

Компонент поиска

Интеллектуальная система должна действовать в ситуации неопределенности, для чего она может либо анализировать предыдущий опыт с целью вывода новых знаний, либо создать новый опыт (путем проведения экспериментов), либо выполнить поиск недостающих знаний во внешних по отношению к ней источниках.

Для решения последней из указанных задач предназначается компонент поиска, в функционал которого входят:

- получение информации о том, какие именно (о чем?) требуются данные / знания;
- формулирование запроса и отправка его либо в структурированные источники знаний (DbPedia, Wikidata и др.), либо в неструктурированные (поиск в литературе, в Глобальной Сети и т.д.);
- анализ полученных при поиске результатов, структуризация и формализация в форму знаний.

Ядро

Представляет собой центральный, связующий элемент системы.

Главной его задачей является организация информационной связи между ядром и всеми остальными компонентами: фактически, доставка сообщений между ними. Таким образом, ядро – это набор средств для считывания данных / команд из их источников и отправка их адресатам.

Здесь важно понимать, что само ядро не принимает решений о способах и средствах обработки поступающих данных – за каждое из решений отвечает соответствующий задаче компонент, что нужно учитывать при разработке каждого из них.

Более того, понимание самих процессов информационного обмена между компонентами должно предшествовать программной реализации “внутренностей” последних, так как только оно дает понимание о работе системы как единого целого. А для этого, в свою очередь, необходимо выбрать форму реализации ядра.

В первом случае ядро реализуется как целостная, самостоятельная единица, набор функций, независимый от компонентов. Тогда сами компоненты только готовят данные для отправки, а ядро в определенной последовательности циклично опрашивает все компоненты, на каждой итерации выполняя одну или несколько транзакций (примерно как это выполняется в цикле работы программируемого логического контроллера). Конечно, здесь можно реализовывать и параллельные вычисления, если аппаратное обеспечение дает такие возможности.

Во втором случае ядро является рассредоточенным – то есть, его функции распределены по компонентам. Тогда каждый из них должен помимо собственных функций реализовывать информационный обмен с другими компонентами. В такой форме реализации ядра как отдельного компонента не существует.

Мультиагентная организация системы на основе знаний

Сказанное ранее про СОЗ не означает, что она представляет собой комплекс, целиком расположенный на одном вычислительном устройстве. Более того,

если представить систему управления целым предприятием, когда речь идет о большом количестве сигналов, устройств, показателей, неопределенностей, влияющих факторов, взаимосвязей и т.д., то при такой централизации производительность всей системы будет в большой степени зависеть от характеристик центрального вычислительного устройства и качества каналов соединения с ним – так как все решения принимаются централизованно, что нежелательно.

Это одно из оснований перехода в рамках тренда интеллектуализации к распределенным системам, когда вместо единственного устройства-агента принятия решений их существует несколько. **Мультиагентность** здесь мы будем рассматривать как строение системы на основе знаний, представленное множеством программных агентов именно на нескольких физических устройствах.

Примером такой системы может служить описанная в работе [Ковалевский, Онуфриев, 2019] система управления заводом, когда общий функционал предприятия обеспечивает множество агентов (реализованных на платформах Raspberry Pi, каждый из которых контролирует свой участок предприятия), объединенных в одну сеть. При этом, каждый из них представляет собой систему на основе знаний, хотя любая из них будет значительно более простой по функционалу и устройству, чем результат их объединения, – который как раз и представляет собой мультиагентную систему на основе знаний.

Однако, потенциально снимая указанное ограничение, такой переход также ставит вопросы о том, в какой форме будет реализован каждый из компонентов СОЗ, если самих интеллектуальных устройств теперь имеется более одного и во всех реализованы схожие по функционалу компоненты.

Возможные формы: **сетевой компонент, локальный компонент** и **распределенный компонент**.

Сетевой компонент – это первая форма реализации компонента. Такой компонент не хранится на большинстве агентов, доступ к нему осуществляется

по сети. При этом, содержательно обмен с ним не отличается от ситуации, если бы он был “на борту”. Примером такого компонента может служить база знаний, которая физически расположена на единственном устройстве и к которой остальные агенты по необходимости обращаются.

Второй формой является локальный компонент, суть которого можно прояснить фразой “у каждого свой”. Это означает, что такой компонент может присутствовать на каждом устройстве, но направлен на решение, в основном, задач только этого устройства – и, соответственно, имеет содержание, требуемое для работы именно этого устройства. Например, база данных, которая может присутствовать у каждого агента, но хранить нужную только ему информацию. Так, содержательно такие локальные базы данных могут не пересекаться у разных агентов. Однако, локальные компоненты могут быть доступны для остальных агентов.

Третья и самая сложная форма – это форма распределенных компонентов, подразумевающая, что части последнего, даже будучи физически рассредоточены на разных устройствах, все равно составляют единую, целостную систему. Это означает, что между частями единой системы необходимо организовать взаимодействие. Так, если это распределенная база данных, то ее части (которые могут пересекаться между собой), должны синхронизироваться.

Необходимость решения вопроса о формах компонентов связана, с одной стороны, с тем, что ни один агент не должен контролировать всю систему целиком (последняя в этом случае была бы централизованной, ввиду чего мультиагентность бы была излишней), а, следовательно, его собственных сведений может быть недостаточно для выполнения некоторых задач – значит, некоторые компоненты должны быть, как минимум, доступны для других агентов. С другой стороны, если множество агентов будут одновременно обращаться за информацией к одному и тому же источнику, то это может привести к высокой зависимости их работы от качества связи и пропускной

способности каналов. Однако, если вместо этого компоненты будут локальными или распределенными (особенно если при этом информация в них дублируется), то это порождает множество задач, связанных с диспетчеризацией и синхронизацией данных.

Рассмотрим компоненты по отдельности.

База данных не должна функционировать как сетевая, так как в этом случае в ее задачи будет входить сбор всех оперативных данных от каждого устройства. При этом, данные о работе одного из агентов могут быть необходимы для работы другого, поэтому допустимой формой будет локальная БД с возможностью доступа к ней других агентов по необходимости.

Система мониторинга должна быть реализована в локальной форме и работать с локальной базой данных.

Аналогичное можно сказать и про информационно измерительную систему.

А база знаний, в зависимости от потенциальной частоты ее востребованности вполне может быть представлена и в сетевой форме. Для этого компонента возможна любая из этих форм, однако, чем более “локальны” базы знаний, тем сложнее агентам будет “понимать”, в которой из доступных баз знаний нужно искать требуемые для каждого конкретного случая знания.

Компоненты управления нейронными сетями и формулами должны функционировать, скорее, локально, решая задачи устройства, на котором они расположены.

Анализатор опыта может не являться постоянно задействованным компонентом, поэтому для него, помимо локальной формы, вполне допустима и сетевая. Аналогично можно сказать и про компоненты поиска и экспериментов.

Интерфейс пользователя также может быть как сетевой (например, на пользовательском терминале или в форме веб-приложения), так и локальный – если в последнем есть потребность.

Если применяется система автоматического управления, то вряд ли все устройства должны иметь к ней доступ.

В любом случае, вопрос о форме реализации того или иного компонента должен решаться применительно к каждой конкретной проектируемой системе на основе знаний.

Важно иметь в виду, что не существует готовых “рецептов изготовления” систем на основе баз знаний. Это все еще остается во многом научной задачей, требующей индивидуального и творческого подхода.

Однако, именно системы, решающие существенные неопределенности (и только они) могут претендовать на то, чтобы называться интеллектуальными.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Гаврилова Т.А., Кудрявцев Д.В., Муромцев Д.И. Инженерия знаний. Модели и методы: Учебник. — СПб.:Издательство «Лань», 2016. — 324 с.: ил.
2. Гаврилова Т.А., Онуфриев В.А. Анализ ошибок студентов при визуальном структурировании знаний // Компьютерные инструменты в образовании, 2016. № 6. С. 42–54.
3. Золин Е.Е. Дескрипционная логика (спецкурс), кафедра математической логики и теории алгоритмов, Механико-математический факультет, Московский государственный университет им. М.В.Ломоносова, http://lpcs.math.msu.su/~zolin/dl/ (29.02.2020).
4. Зюзьков В.М. Математическая логика и теория алгоритмов : учебное пособие / В. М. Зюзьков. — Томск : Эль Контент, 2015. — 236 с.
5. Книгин А.Н. Учение о категориях: учебное пособие для студентов философских факультетов. — Томск, 2002. — 193 с.
6. Ковалевский В.Э., Онуфриев В.А. Мультиагентные алгоритмы согласования ключевых показателей эффективности предприятия // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. 2019. Т. 12, № 3. С. 67–80. DOI: 10.18721/JCSTCS.12306.
7. Khokhlovskiy, V.; Oleynikov, V.; Kostenko, D.; Onufriev, V. & Potekhin, V. (2019). Modernisation of a Production Process Using Multicriteria Optimisation Logic and Augmented Reality, Proceedings of the 30th DAAAM International Symposium, pp.0500-0507, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-22-8, ISSN 1726-9679, Vienna, Austria DOI: 10.2507/30th.daaam.proceedings.067.

Онуфриев Вадим Александрович

ПРОЕКТИРОВАНИЕ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ УПРАВЛЕНИЯ

Учебное пособие

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 95 3005 – учебная литература

Подписано в печать 05.03.2020. Формат 60×84/16. Печать цифровая.
Усл. печ. л. 8,25. Тираж 28. Заказ 0547.

Отпечатано с готового оригинал-макета, предоставленного автором,
в Издательско-полиграфическом центре Политехнического университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 552-77-17; 550-40-14.