Интеграция методов инженерии знаний и инженерии программ: система управления знаниями Knowledge.NET

Сафонов Владимир Олегович
(v_o_safonov@mail.ru)
Новиков Антон Владимирович
Сигалин Максим Владимирович
Смоляков Алексей Леонидович
Черепанов Дмитрий Геннадьевич

Введение

Инженерия знаний (knowledge engineering) – одно из важных направлений современного программирования, занимающееся развитием языков, методов и систем представления и обработки знаний на компьютерах. Оно возникло в 1950-х гг. как один из разделов *искусственного интеллекта*, в связи с тем, что при решении задач творческого характера с помощью компьютеров оказалось необходимым не только разрабатывать алгоритмы и программы, реализующие те или иные методы (например, стратегию поиска решения сначала в глубину), но и явным образом (отдельно от программ) хранить, изменять и пополнять *базы знаний*, содержащие в обобщенном виде необходимую для решения задач информацию о предметной области.
Как выяснилось, знания, необходимые для решения многих нетривиальных практических задач с использованием компьютераов, носят *гибридный* характер, то есть, требуются не только *процедурные знания* (алгоритмы, их программные реализации и типовые процедуры решения задач – т.е. те формы знаний, которые фактически всегда неявно используются для компьютерного решения любой задачи, но в виде, жестко "зашитом" в исходный текст программы), но также *концептуальные знания* (определения концепций проблемной области и отношений между ними), *фактуальные знания* (конкретные факты и их связи между собой) и *эвристические знания* (неформальные правила рассуждений, отражающие практический опыт решения задач в проблемной области).
Соответственно, основными методами представления знаний на компьютерах, сложившимися в течение нескольких десятилетий, являются:
- *продукции,* или *правила (rules)* – конструкции вида *ЕСЛИ условие ТО действие*, используемые для *продукционного вывода*, при котором на каждом шаге, при истинности условия некоторого выбранного правила, активизируется его действие; оно может заключаться в добавлении в *рабочее множество* гипотез некоторого утверждения. Правила удобны для представления эвристических знаний и структурируются в виде *наборов правил (rule sets)*, каждый из которых применяется для вывода некоторого *целевого утверждения (goal)*;
- *фреймы (frames)* – иерархические структуры, удобные для представления знаний о концепциях и их взаимосвязи; фрейм состоит из *имени* и *слотов* (slots), каждый из которых имеет свое *имя* и *значение* (как правило, ссылку на другой фрейм); в соответствии с оригинальным подходом автора данной концепции M.Minsky (США), фреймы могут использоваться не только для представления структур знаний, но и для *сопоставления* с объектами исследуемой проблемной области, с целью их анализа и классификации;
- *семантические сети (semantic nets)* – нагруженные ориентированные мультиграфы, которые, как и фреймы, удобны для представления концепций и их взаимоотношений и являются, по-видимому, наиболее общей формой представления знаний.
Методы представления знаний получили особенно широкое распространение, начиная с 1970-х гг., когда весьма популярным направлением искусственного интеллекта стали *экспертные системы (expert systems)* [1] - интеллектуальные программы, основанные на использовании отдельно хранимой, пополняемой *базы знаний* о предметной области, и играющие в данной области роль *эксперта*, умеющего решать в ней, как правило, достаточно узкий, ограниченный применимостью конкретных,

представленными в базе знаний, круг задач диагностики, планирования, прогнозирования и т.д.
В настоящее время развитие методов и инструментов представления знаний приобрело особую актуальность, в связи с необходимостью улучшения качества информационного поиска в Интернете.

1. Проблема интеграции методов инженерии знаний и программной инженерии

По мере развития методов инженерии знаний, их теоретические, методологические и технологические основы все более отдалялись от основ *инженерии программ (software engineering)* – методов спецификации, проектирования, реализации, тестирования и сопровождения программ и программных продуктов.
Исторически сложилось так, что инженерия знаний, начиная с 1960-х гг. (создание языка ЛИСП для символьной обработки) и с 1970-х гг. (создание языка Пролог, основанного на идее использования метода резолюций для эффективного логического вывода в исчислении предикатов первого порядка, в качестве основы для систем искусственного интеллекта), развивалась на основе совершенно иных вычислительных моделей и парадигм, нежели традиционная программная инженерия (ФОРТРАН, АЛГОЛ, Паскаль, структурное и модульное программирование, абстрактные типы данных, объектно-ориентированное программирование и т.д.).
Для инженерии знаний использовались методы математической логики, теории продукций, системы преобразований термов (term-rewriting systems), а опыт и результаты в данной области накапливались, главным образом, усилиями специалистов в указанных теоретических областях, программирующих на языках ЛИСП, Пролог и их диалектах, применение которых требует иных методов мышления, принципиально отличных от тех, которые используются при программировании на процедурных или объектно-ориентированных языках.
Напротив, в инженерии программ использовались и развивались языки, методы и инструменты, основанные на более традиционных вычислительных моделях, с точки зрения которых, использование языков ЛИСП или Пролог выглядело “экзотикой”.
Таким образом, за десятилетия развития обеих дисциплин сложился серьезный разрыв между ними.
Специалисты по инженерии знаний мыслят в терминах, подобных следующим: *“концепт”, “факт”, “гипотеза”, “атом”, “список”, “условие”, “заключение”, “целевое утверждение”, “граф”, “логическая связка”, “фрейм”, “демон”, “противоречие”, “возврат (back-tracking)”, “нечеткость”, “коэффициент уверенности”, “степень истинности”*..
Характерный перечень терминов, используемый специалистом по инженерии программ, следующий: *“программа”, “описание”, “оператор”, “переменная”, “ячейка памяти”, “значение”, “присваивание”, “массив”, “структура”, “процедура”, “параметр”, “программный модуль”, “класс”, “объект”, “метод”, “исключительная ситуация”, “стек”, “куча”*.
Налицо, к сожалению, практически полное взаимное непонимание между специалистами обеих категорий; пересечение множеств самих таких специалистов, увы, практически пусто.
Этот феномен нашел отражение и в языках и инструментах программирования, используемых в обеих областях. Не встретишь, например, специалиста по объектно-ориентированному программированию, использующего современные платформы Java и .NET, который в то же время был бы фанатиком логического программирования и разрабатывал бы многие свои программы в системах Turbo Prolog или Visual Prolog.
Однако, на практике, для решения очень многих реальных задач должно использоваться именно *сочетание методов программной инженерии и методов инженерии знаний*. Например, для решения задачи планирования развития сложных технических систем, которая была поставлена перед нами одним из заказчиков уже почти 20 лет назад, характерно сочетание *алгоритмических методов* дискретного программирования (для решения крупных частей задачи) с *методами искуственного интеллекта* (например, основанными на наборах эвристических правил), которые необходимо использовать для оценки результатов применения этих алгоритмических методов решения.
Увы, ни одна из известных, широко используемых платформ “традиционного” программирования (Java, .NET, Eiffel, Active Oberon и др.) не содержит адекватных средств представления знаний, которые были бы включены в эту систему в качестве основных понятий и конструкций базовых языков.

И наоборот, ни одна из известных систем инженерии знаний (Common LISP, Turbo Prolog,Visual Prolog, KEE, Ontolingua, Protégé и др.) не содержит адекватных, современных, удобных и эффективно реализованных средств “традиционного” программирования. Семантика конструкций подобных систем, как правило, формулируется в терминах их реализации на одном из диалектов Лиспа. Ни один специалист не станет, например, на ЛИСПе или Прологе реализовывать алгоритмы численного решения задач. Возможности же использования в системах инженерии знаний программных модулей на традиционных языках либо отсутствуют, либо ограничены, например, возможностью запустить из набора правил целое приложение для Windows (.exe) и использовать его результат, записанный в некоторый файл.
Описанный семантический разрыв между методами инженерии программ и инженерии знаний все более препятствует успешному решению многих реальных практических задач с помощью компьтеров и, по нашему мнению, должен быть преодолен в будущих системах.

В связи с этим, в Санкт-Петербургском университете, в лаборатории проф. В.О. Сафонова (до 2001 г. – лаборатории технологии программирования и экспертных систем, с 2001 г. – лаборатории Java-технологии НИИ математики и механики), начиная с 1980-х – 1990-х гг., разрабатываются методы *интеграции инженерии знаний и инженерии программ*. Они уже реализованы в языке представления знаний Турбо-Эксперт (1991 г.) [3] – расширении Турбо-Паскаля средствами представления гибридных знаний, и в экспериментальных версиях языков и систем Java Expert [4] (расширения Java средствами представления знаний) и C# Expert [5] – расширения языка C# средствами представления фреймовых и продукционных знаний).
В основе всех этих языков и систем, как и в основе описанных в данной работе языка и системы Knowledge.NET, лежит принцип *расширения современных языков и систем программирования средствами (концепциями, языковыми конструкциями, библиотеками и др.) инженерии знаний*.
На наш взгляд, только такое сочетание методов и конструкций “традиционного” программирования (т.е. инженерии программ) с методами и конструкциями инженерии знаний в одной системе программирования (платформе для разработки программ) позволит, наконец, создавать с их помощью гибкие, удобные, эффективные, понятные пользователю и адекватные для сопровождения и развития *интеллектуальные решения (intelligent solutions*) для реальных практических задач.
Как это выглядит на практике с точки зрения пользователя такой системы программирования? Инженер знаний и программ, проектировщик и разработчик такого интеллектуального решения, использует весь арсенал методов представления гибридных знаний (онтологии, фреймы, наборы правил и др.) для формализованного описания предметной области. Для реализации же более традиционных “алгоритмических” компонент он использует конструкции современного, эффективно реализуемого, базового языка программирования (Java, C# и др.), расширением которого является инструментальный язык данной гибридной системы.
Для реализации расширений, предназначенных для представления знаний, используется *конвертирование* – перевод этих расширений в программу на *базовом языке* (Java, C# и др.), содержащую вызовы методов специализированной библиотеки (API), также являющейся частью данной системы и обеспечивающей поддержку реализации наиболее сложных по семантике конструкций для представления и обработки знаний.
Для компиляции в исполняемый объектный код полученных в результате конвертирования исходных текстов на базовом языке используется его “штатный” компилятор; например, для Java – компилятор *javac*, входящий в состав Java Developer’s Kit (JDK) фирмы Sun Microsystems; для C# - компилятор фирмы Microsoft, входящий в состав Microsoft.NET Framework.
Отметим, что идеи и принципы интеграции методов инженерии знаний и инженерии программ получили в последние годы распространение и среди западных специалистов – например, в работе [2] и в докладах на крупных международных конференциях их развивает профессор Jeff Zhuk (University of Phoenix, Arizona, USA), с которым наш коллектив поддерживает творческое содружество.

2. Архитектура системы Knowledge.NET

Как уже отмечалось в одной из наших предыдущих работ, опубликованных в журнале [6], одной из

наиболее перспективных платформ для разработки программного обеспечения в настоящее время является платформа Microsoft.NET. Естественно, в нашем коллективе возникла идея разработки современной системы представления знаний на базе данной платформы – *системы Knowledge.NET*. При ее разработке использован опыт другого нашего проекта для Microsoft.NET – *системы аспектно-ориентированного программирования Aspect.NET* [7, 8].

Система Knowledge.NET [9] базируется на одноименном языке, который является расширением языка C# - наиболее современного и развитого языка программирования в настоящее время - концепциями и конструкциями для инженерии знаний.
Система предназначена для разработки и использования библиотек (баз) знаний из разнообразных предметных областей, которые могут проектироваться и реализовываться непосредственно на языке Knowledge.NET (средствами специализированного редактора и визуализатора знаний), а также могут автоматически извлекаться из WWW. В системе предусмотрено конвертирование полученных, сформулированных на языке Knowledge.NET знаний в общеупотребительный формат представления знаний – *KIF (Knowledge Interchange Format)* [10].
В соответствии с современными идеями и тенденциями, на основе успешного опыта разработки системы Aspect.NET, система Knowledge.NET реализована как *расширение (add-in)* одной из наиболее современных интегрированных сред современного программирования – *Microsoft Visual Studio.NET 2005 (Whidbey)*. Этот важнейший принцип архитектуры Knowledge.NET обеспечивает возможность применения при разработке баз знаний в системе Knowledge.NET всего широкого спектра возможностей проектирования, разработки и отладки программ, предоставляемых Microsoft Visual Studio.NET.

Система Knowledge.NET состоит из следующих основных программных компонент:
- *конвертора с языка Knowledge.NET в базовый язык C#* платформы Microsoft.NET (последующая компиляция программ на C#, полученных в результате конвертирования, обеспечивается штатными средствами интегрированной среды Visual Studio.NET);
- *редактора и визуализатора знаний Knowledge Editor*, обеспечивающего визуализацию, проектирование, реализацию и модификацию в интерактивном режиме исходных текстов баз знаний на языке Knowledge.NET;
- *системы Knowledge Prospector извлечения знаний в формате Knowledge.NET из текстов на естественном языке* (ориентированную на извлечение знаний из Интернета);
- *конвертора KIF Converter знаний из формата Knowledge.NET в общеупотребительный формат KIF* [10].

3. Язык представление знаний Knowledge.NET

Язык Knowledge.NET является расширением языка C# средствами представления гибридных знаний, основанными на концепции *онтологии*, предложенной в начале 1990-х гг. в работах Стэнфордского университета (США). Онтология – это разновидность спецификации предметной области, выраженной в терминах ее *концептов* (понятий) и их взаимосвязей В качестве основы языка представления знаний Knowledge .NET используется фреймовые языки C# Expert **Ошибка! Источник ссылки не найден.** и Turbo Expert **Ошибка! Источник ссылки не найден.**. Однако, в отличие от обычных фреймовых языков, Knowledge.NET расширен языковыми конструкциями для представления онтологических знаний. Семантика онтологии Knowledge .NET схожа с концепцией языка представления знаний OWL, разработанного W3C консорциумом **Ошибка! Источник ссылки не найден.**. Также следует отметить, что в Knowledge .NET используется терминология, принятая в онтологических системах (Concept-Property-Individual), вместо терминологии фреймовых систем (Frame-Slot-Instance).

3.1. Концепты

Для представления множества экземпляров одного типа в онтологической модели используется понятие концепта (Concept). Аналогично, в объектно-ориентированной модели объекты описываются с помощью классов.

В качестве примера рассмотрим онтологию *Транспортные Средства (Vehicles)*. В рамках данной онтологии можно предположить, что все автомобили являются экземплярами некоторого концепта *Автомобиль (Automobile)*. При описании сложных онтологий концепты организованы в виде иерархической структуры (концепт-подконцепт), или *таксономии.* В вершине дерева концептов находится концепт *Thing*, от которого унаследованы все остальные концепты. Также введен концепт *Nothing.* Последний является подконцептом любого концепта (т.е. унаследован от любого концепта)

3.2. Подконцепты

Концепт может быть конкретизацией (унаследован от) одного или более других концептов. Если концепт **A** унаследован от концептов **B** и **C**, в программе на Knowledge .NET это записывается с помощью одной из следующих конструкций:

1. **A** is_subconcept_of **B,C**;

   Данная, наиболее короткая, форма записи может быть использована, если определяемая сущность (в данном случае концепт) задается с помощью только одного параметра (is_subconcept_of)

2. **A**
   {
   is_subconcept_of **B**,**C;**
   }

   Данная форма записи используется, если сущность определяется несколькими параметрами.
3. **A**
   {
   is_subconcept_of **B**;
   is_subconcept_of **C**;
   }

   Если параметр (в данном примере is_subconcept_of) в качестве значения получает несколько сущностей (в данном примере: **B** и **C**), то данный параметр можно продублировать для каждой полученной сущности. Дублирование параметра особенно удобно, если одна из сущностей является анонимным концептом.

3.3. Разъединенные концепты

Иногда онтология подразумевает, что некоторый экземпляр не может быть одновременно реализацией некоторого множества концептов. Такие концепты называются *разъединенными*, или *дизъюнктными*(*disjoint*). Например, оба концепта *Самолет (Plane)* и *Подводная Лодка (Submarine)* являются подконцептами концепта *Транспортное Средство.* Тем не менее, конкретный экземпляр *Транспортного средства* (индивид) не может одновременно являться *Самолетом* и *Подводной Лодкой*. Разъединенные концепты определяются с помощью слова *disjoint*.

Vehicle is_subconcept_of **Thing;**
Plane is_subconcept_of **Vehicle**;
Submarine is_subconcept_of **Vehicle;**
disjoint **Plane, Submarine;**

Разъединенные концепты можно также определять с помощью слова *disjoint_with,* используемого внутри определения концепта. Семантика *disjoint_with* слегка отличается от семантики *disjoint* и может быть удобна в некоторых ситуациях. Например, предположим, что онтология *Транспортные Средства* содержит также концепт *Корабль (Ship)*. Концепты *Корабль* и *Самолет* являются разъединенными концептами, а концепты *Корабль* и *Подводная Лодка* будем считать не разъединенными концептами. В таком случае разъединенные концепты удобно определить следующим образом:

Vehicle is_subconcept_of **Thing;**
Submarine is_subconcept_of **Vehicle;**
Ship is_subconcept_of **Vehicle;**

Plane
{
is_subconcept_of **Vehicle**;
disjoint_with **Ship, Submarine;**
}

3.4. Набор экземпляров

Концепт может быть определен с помощью перечисления всех его экземпляров. Такие концепты называются *Наборами*. Если перечисляемый в наборе экземпляр не определен в системе, Knowledge.NET создает экземпляр с данным именем на основе концепта *Thing*.

enumerated concept **CONCEPT_IDEN** is_one _of **INDIVIDUAL1, INDIVIDUAL2, …**;

где:
CONCEPT_IDEN – идентификатор концепта;
INDIVIDUAL1, INDIVIDUAL2 – идентификаторы экземпляров.

3.5. Объединение, пересечение и дополнение концептов

Концепт может быть определен как объединение (union), пересечение (intersection) или дополнение (complement) множеств экземпляров некоторого числа других концептов:

concept **A** is_intersection_of **B, C, D…**
concept **A** is_union_of **B, C, D…**
concept **A** is_complement_of **B, C, D…**

где **A, B, C, D –** концепты.

Концепт **A –** дополнение некоторого числа других концептов - является множеством всех экземпляров, которые не принадлежат перечисленным концептам (**B, C, D…**).
Рассмотрим следующий пример. Пусть определена некоторая онтология, состоящая из следующих концептов и экземпляров:

концепт **A** и его экземпляры **a1, a2**
концепт **B** и его экземпляры **b1, b2, b3**
концепт **C** и его экземпляр **c1**
концепт **D** и его экземплры **d1, d2, d3, d4, d5**

Тогда выражение «concept **E** is complement of **B, C**» определяет концепт, состоящий из экземпляров концептов **A, D (a1, a2, d1-d5).**
Особенно интересно использовать дополнение концептов для *анонимных (inline)* концептов. Например:

concept **E**
{
is_complement_of
{

IsLinivgAt has_value **Russia**

}

}

Здесь концепт E описывает множество всех людей, живущих не в России.

Конструкции *is_complement_of*, *is_union_of*, *is_intersection_of* можно вкладывать друг в друга.

3.6. Свойства

Свойство представляет связь между двумя сущностями (Во фреймовых системах свойства называются слотами). Поддерживаются следующие основные типы свойств:

- *Простые свойства*
  Простое свойство связывает экземпляр со значением некоторого простого типа данных, например, со строкой (string) или целым числом (int).
- *Объектные свойства*
  Объектное свойство связывает между собой два экземпляра.
- *Аннотации*
  Аннотации также относятся к свойствам. С помощью свойства *Аннотация* пользователь может добавить описание (метаданные) к концептам, экземплярам, объектным и простым свойствам.

3.7. Домен и область значения

Для каждого свойства должен быть определен *домен (domain)* и *область значений (range)*. Например, в онтологии *Транспортные Средства* может быть определено свойство *ИмеетМаксимальнуюСкорость (HasMaxSpeed),* у которого *домен* содержит концепт *Транспортное Средство,* а *область значений* есть простой тип *uint*. На языке Knowledge .NET свойство *ИмеетМаксимальнуюСкорость* определяется следующим образом:

```
datatype property HasMaxSpeed
{
        domain Vehicle;
        range uint;
}
```

Отметим, что *домен* и *область значений* может содержать более одного идентификатора. В этом случае идентификаторы перечисляются через запятую.

3.8. Простые свойства

Простое свойство связывает экземпляр с не более чем одним значением некоторого простого типа данных. Например, свойство *ИмеетМаксимальнуюСкорость (HasMaxSpeed)* связывает экземпляры концепта *Транспортное Средство* со значением простого типа данных *uint.* Синтаксис определения простого свойства:

```
datatype property PROP_NAME
{
        [domain A, B, C, …;]
        [range X, Y, Z;]
}
```

где:

- PROP_NAME – название свойства. Хотя в Knowledge .NET не предусмотрено ограничений на наименования свойств, рекомендуется, чтобы имя начиналось со слова Has (имеет...) или Is (является...). Например: HasParent, IsParentOf.

- A, B, C – идентификаторы концептов. Если *домен* не определен, то в качестве домена по умолчанию используется концепт *Thing*.

- X, Y, Z – идентификаторы простых типов. Если *область значений* не определена, то значение может быть любым простым типом.

3.9. Объектные свойства

3.9.1. Общее описание

Объектное свойство связывает между собой два экземпляра. Например, в рассматриваемой нами онтологии *Транспортные Средства* концепт *Транспортное Средство* мы можем связать с концептом *Цвет (Color)* с помощью свойства *ИмеетЦвет (HasColor)*:

#ontology "Vehicles**"**

#concepts
Color is_subconcept_of **Thing;**

Vehicle is_subconcept_of **Thing;**
Plane is_subconcept_of **Vehicle**;
Submarine is_subconcept_of **Vehicle;**
disjoint **Plane, Submarine;**

disjoint **Color, Vehicle;**

#properties

object property **HasColor**
{
domain **Vehicle;**
range **Color;**
}

#end_of_ontology "Vehicles"

3.9.2. Подсвойства

В Knowledge .NET объектные свойства, так же, как и концепты, образуют иерархическую структуру. Другими словами, допускается наследование свойств. Подсвойство имеет в качестве *домена* и *области значений* подмножества *домена* и *области значений* родителя. Определяется подсвойство с помощью ключевого слова *is_subproperty_of*. Например, если свойство **X** является подсвойством свойства **Y,** это должно быть записано следующим образом:

object property **X** is_subproperty_of **Y**
{
domain **A, B, C, …;**
range **K, L, M, …;**
}

Множественное наследование для свойств не допускается.

3.9.3. Обратные свойства

Объектное свойство может иметь соответствующее обратное (инвертированное) свойство. Если некоторое свойство соединяет экземпляр **A** с экземпляром **B,** то обратное свойство соединяет экземпляр **B** с экземпляром **A**. Например, если машина Иванова А.К. *ИмеетЦвет (HasColor)* Красный, то Красный *ЯвляетсяЦветом (isColorOf)* машины Иванова А.К. Таким образом, в данном

примере свойство *ЯвляетсяЦветом* – это обратное свойство по отношению к свойству *ИмеетЦвет*. Обратные свойства определяются с помощью ключевого слова *inverse*.

```
object property HasColor
{
domain Vehicle;
range    Color;
inverse IsColorOf;
}
```

3.9.4. Функциональные свойства

Функциональные свойства связывают экземпляр не более чем с одним экземпляром из области значений. Например, свойство *ИмеетЦвет* из примера, описанного в разделе 0, является функциональным, потому что экземпляр концепта *Транспортное Средство* не может быть покрашен сразу в два цвета. В то же время, обратное свойство *ЯвляетсяЦветом* - не функциональное, так как несколько экземпляров могут иметь один и тот же цвет. Функциональные свойства определяются с помощью модификатора *functional*.

```
functional object property HasColor
{
domain Vehicle;
range    Color;
inverse  IsColorOf;
}
```

3.9.5. Обратные функциональные свойства

Так же, как и прямые свойства, обратные свойства могут быть функциональными. Для определения обратного функционального свойства используется модификатор *functional* перед ключевым словом *inverse*.

```
object property IsColorOf
{
domain Color;
range Vehicle;
functional inverse HasColor;
}
```

3.9.6. Транзитивные свойства

Свойства, удовлетворяющие следующему правилу, называются транзитивными: если экземпляр **A** связан с экземпляром **B** через свойство **P** и экземпляр **B** связан с экземпляром **C** через свойство **P**, то экземпляр **A** связан с экземпляром **C** через свойство **P**. Следует отметить две особенности транзитивных свойств:

1. Будем полагать, что свойство не может быть транзитивным и функциональным одновременно;
2. Если свойство транзитивно, то и обратное свойство транзитивно;

С теоретической точки зрения, свойство может быть транзитивным и функциональным одновременно только в случае, когда свойство соединяет экземпляр концепта сам с собой, что нетрудно доказать. Однако подобная ситуация представляется нам вырожденной и не поддерживается на уровне конструкций языка.

В качестве примера рассмотрим онтологию *Генеалогического Древа.* В данной онтологии определен концепт *Человек (Human)* и свойство *ЯвляетсяПредком (IsAncestorOf),* у которого *домен* совпадает с *областью значений* и состоит из единственного концепта *Человек.* Очевидно, что свойство *ЯвляетсяПредком* транзитивно. Действительно, предположим что Алексей *ЯвляетсяПредком* Сергея, а Сергей *ЯвляетсяПредком* Ирины, тогда Алексей *ЯвляетсяПредком* Ирины. Такую онтологию

можно записать с использованием модификатора *transitive*.

#ontology “family-tree”

#**concepts**
Human is_subconcept_of **Thing**;

#properties
transitive object property **IsAncestorOf**
{
domain **Human;**
range **Human;**
inverse **HasAncestor;**
}
#end_of_ontology “family-tree”

3.9.7. Симметричные свойства

Свойство **P** называется симметричным, если из условия, что свойство **P** связывает концепт **A** с концептом **B,** следует связь концепта **B** с концептом **A** также через свойство **P.** Для симметричных свойств *область значений* совпадает с *доменом*, поэтому при записи область значений для симметричных свойств будем опускать. Также отметим, что, по определению симметричного свойства, оно обратно самому себе. В качестве примера рассмотрим свойство *ЯвляетсяБратомИлиСестрой (HasSibling)*. Симметричность данного свойства очевидна из его семантики. Действительно, если Алексей *ЯвляетсяБратомИлиСестрой* Оксаны, тогда Оксана *ЯвляетсяБратомИлиСестрой* Алексея. В Knowledge .NET симметричные свойства определяются с помощью модификатора *symmetric*.
Свойство *ЯвляетсяБратомИлиСестрой* в онтологии *Генеалогическое Древо* может быть определено следующим образом

symmetric object property **HasSibling**
{
domain **Human;**
}

3.10. Аннотации

Важным типом свойств являются Аннотации. С помощью свойства *Аннотация* пользователь может добавить описание (метаданные) к концептам, экземплярам, объектным и простым свойствам, правилам. Аннотация может содержать следующие поля:

1. Информация о Версии (version_info)
   Описание версии сущности (концепта, экземпляра, свойства);
2. Название (label)
   Развернутое название сущности на естественном языке.
3. Комментарий (comment)
   Автор (created_by)
4. Дата Создания (creation_date)

Все поля являются строками в формате UNICODE. Ниже приведен синтаксис определения аннотации.

annotation
{
[version_info “**SOME_TEXT**”;**]**
[label “**SOME_TEXT**”;**]**
[comment “**SOME_TEXT**”;**]**
[created_by “**SOME_TEXT**”;**]**

[creation_date "**SOME_TEXT**";**]**

}

где **SOME_TEXT –** любой текст в формате UNICODE.

3.11. Ограничения

В Knowledge .NET свойства могут быть использованы для задания *ограничений*. В онтологии каждому ограничению удовлетворяет ноль или более экземпляров. Таким образом, можно рассматривать ограничения как концепты. Поддерживаются три основных типа ограничений:

1. Ограничения по квантору;
2. Ограничения по мощности;
3. Ограничения по значению;

3.11.1. Ограничения по квантору

Ограничения по квантору состоят из трех компонентов:

1. Свойство
   Любое простое или объектное свойство, определенное в онтологии.
2. Квантор
   Допускается квантор существования (∃, some_values_from) и квантор всеобщности (∀, all_values_from).
3. Наполнитель
   Концепт или простой тип, который может быть *анонимным* концептом, т.е. концептом, не имеющим имени и заданным явно в рассматриваемом фрагменте исходного кода.

В качестве примера рассмотрим онтологию *Генеалогического Древа,* введенную выше. Так как у каждого человека есть предки, мы можем ввести следующее ограничение в данной онтологии:

HasAncestor some_values_from **Human**

Данное ограничение определяет некоторое множество экземпляров, которые обладают следующими свойствами:

1. Унаследованы от концепта *Человек*.
2. Есть хотя бы один предок.

Другими словами, введенное нами отношение само по себе определяет некоторый новый концепт (множество экземпляров).

Ранее мы рассмотрели свойство *ЯвляетсяБратомИлиСестрой (HasSibling)*. Теперь с помощью ограничений определим концепт *БратИлиСестра (Sibling)*.

```
concept Sibling
{
        is_subconcept_of
        {
                Human;
                HasSibling some_values_from Sibling;
        }
}
```

3.11.2. Ограничения по мощности

В Knowledge .NET можно определить для свойства ограничение на количество связей с другими экземплярами или простыми типами.

Поддерживаются следующие ограничения по мощности:

1. Не более чем (max_cardinality)
2. Не менее чем (min_cardinality)
3. Точное количество (cardinality)

В качестве примера введем отношение *ЯвляетсяРебенком (IsChildOf)* в онтологии *Генеалогическое Древо*. Поскольку *Человек* всегда имеет ровно два родителя, определение концепта *Человек* имеет следующий вид:

```
#concepts
concept Human
{
        is_subconcept_of
        {
                Thing;
                HasSibling some_values_from Sibling;
                IsChildOf cardinality 2;
        }
}

#properties
object property IsChildOf
{
        domain Human;
        range Human;
        inverse HasChild;
}
```

3.11.3. Ограничения по значению

Ограничение по значению (has_value) позволяет определять концепты, основанные на существовании определенных экземпляров. Таким образом, экземпляр будет принадлежать данному концепту, только если одно из значений его свойств совпадает со значением, заданным в концепте.

Введем в онтологию из раздела 0, свойство *ЖивetВ* (*IsLivingAt*) и концепт (набор экземпляров) *Country*, тогда концепт *ЖительРоссии (ResidentOfRussia)* будет определяться следующим образом:

```
#concepts

enumerated concept Country is_one_of Russia, USA, UK, Germany, France, Italy, China, Egypt;

concept ResidentOfRussia
{
        is_subconcept_of
        {
                Human;
                IsLivingAt cardinality 1;
                IsLivingAt has_value Russia;
        }
}


#properties

object property IsLivingAt
{
        domain Human;
        range Country;
}
```

3.12. Дополнительные имена

Для концептов и экземпляров можно определить несколько имен (*aliases*). Например, чтобы иметь возможность использовать идентификатор *Машина (Car)* для концепта *Автомобиль,* необходимо ввести следующее определение:

```
concept Automobile
{
        alias "Car";
        is_subconcept_of Vehicle;
}
```

3.13. Эквивалентные концепты

Концепт **A** и концепт **B** называются эквивалентными, если это явно указано с помощью ключевого слова *equivalent_concepts* в конструкции следующего вида.

```
equivalent_concepts A, B, C, …;
```

A, B, C – концепты

3.14. Экземпляры

Экземпляры (или Индивиды) представляют объекты в определяемой онтологии. Важное отличие экземпляров от объектов в ООП в том, что один и тот же экземпляр может иметь несколько разных имен. Например, *ВАЗ 21010 номерной знак Y123AK* и *машина Иванова И.Л.* могут являться идентификаторами одного и того же экземпляра.

В программе экземпляр задается с помощью ключевого слова *individual*. Концепты, реализуемые данным экземпляром, определяются после ключевого слова *is_a*. Значения свойств задаются с помощью одной из ниже перечисленных конструкций:

1. Если присваивается только одно значение:
   NAME_OF_PROPERTY = VALUE_OF_PROPERTY;
2. Если присваивается два и более значений:
   NAME_OF_PROPERTY = {VALUE_OF_PROPERTY1, VALUE_OF_PROPERTY2, …}

Пример создания экземпляра концепта *Человек*:

```
individual Dmitry
{
        alias "Dima";
        is_a Human;
        HasSibling = Nina;
        HasAncestor = {Victor, Ludmila, Grigory, Alena};
        HasChild = {Egor, Maria};

        annotation
        {
                version_info "1.0";
                created_by "Alexander Fedorov";
                creation_date "01 July 2005";
        }
}
```

3.15. Одинаковые экземпляры

Если экземпляр **A** совпадает с экземпляром **B,** то данный факт может быть определен в онтологии с помощью ключевого слова *same:*

same **A, B, C,…;**
где **A, B, C –** экземпляры.

Необходимо отметить, что ключевые слова *alias* и *same* имеют различную семантику.
Слово *same* используется для объединения различных экземпляров в одну сущность. Например, инженер знаний создает новую онтологию, объединяя существующие онтологии. В исходных онтологиях, одни и те же экземпляры могут быть описаны по-разному, но в новой онтологии данные экземпляры должны трактоваться как одна сущность (экземпляр). Чтобы объединить эти экземпляры, инженер знаний использует слово *same*.
Alias, в свою очередь, - это способ ассоциировать один и тот же физический объект с разными именами.
Таким образом, можно сказать, что с помощью оператора *same* объединяются разные физические представления, а с помощью оператора *alias* одному и тому же физическому представлению присваиваются разные логические имена.

3.16. Различные экземпляры

Ключевые слова *different_from*, *all_different* позволяют явно определить в онтологии, что данные экземпляры концептов различны.

Например, если необходимо явно задать экземпляры *Билл Смит, Джефф Крамак* и *Стивен Фриман* концепта *Человек* как отличные друг от друга, это можно сделать с помощью следующей конструкции:

all_different **Bill_Smith**, **Jeff_Kramak**, **Stephen_Freeman**;

Если необходимо определить, что Джефф_Крамак отличен от экземпляров Билл_Смит и Вильям_Смит, а о том, различны ли экземпляры Билл_Смит и Вильям_Смит, нам ничего не известно, тогда удобнее воспользоваться конструкцией:

individual **Jeff_Kramak**
{
is_a **Human;**
different_from **Bill_Smith, William_Smith;**
}

3.16. Язык запросов

Язык запросов позволяет выбирать из онтологии экземпляры по некоторому заданному критерию (запросу).
Следует отметить, что сами по себе запросы не являются онтологическими знаниями. Язык запросов является инструментом, который позволяет осуществлять экспертную деятельность на заданной онтологии.
В качестве примера рассмотрим следующий запрос в онтологии *Транспортные Средства*: “Все транспортные средства, максимальная скорость которых больше чем 100 км/ч”. Результатом выполнения данного запроса будет набор экземпляров концепта *Транспортное Средство*, у которых значение свойства *ИмеетМаксимальнуюСкорость* превышает 100 км/ч.
Язык запросов поддерживается с помощью класса *QueryEngine,* определенного в библиотеке классов Knowledge.NET.

3.17. Синтаксис запроса

Запрос позволяет инженеру знаний получить коллекцию экземпляров онтологии, которые удовлетворяют определенному критерию. Все запросы можно разделить на три основных типа:

1. Запросы, критерий которых основан на некоторой коллекции концептов и значениях их свойств;
2. Запросы, критерий которых основан только на значениях свойств, без указания концептов;

3. Запросы, критерий которых содержит только перечисление концептов, без указания свойств и их значений.

3.17.1. Общее описание

С точки зрения синтаксиса, запрос представляет собой текстовую строку (*string*) в следующем формате:

individuals of concepts **CONCEPT1, CONCEPT2, CONCEPT3…**
where **EXP1**

Фрагмент "of concepts **CONCEPT1, CONCEPT2, CONCEPT3…"** задает множество концептов, из которых будут выбираться экземпляры. Данная часть запроса не является обязательной (например, она не определяется в запросах второго типа).

Некоторые из концептов могут быть *Набором Экземпляров,* который определен с помощью подзапроса. Например:

individuals of concepts **CONCEPT1, CONCEPT2,**
individuals of **CONCEPT3, CONCEPT4…**
where **EXP2**
where **EXP1**

Фрагмент "where **EXP1"** задает ограничения на свойства выбираемых концептов (критерий). Описание синтаксиса выражения **EXP1** приведено ниже. Данная часть запроса не является обязательной (например, она не определяется в запросах третьего типа). Тем не менее, одна из частей запроса "*of concepts*" или "*where*" должна быть обязательно задана в запросе.

Пример запроса без критерия:

individuals of **Vehicle**

Данный запрос в онтологии *Транспортные Средства* возвращает все экземпляры концепта *Транспортное Средство* и его подконцептов (т.е. экземпляры концептов *Автомобиль, Корабль, Подводная лодка* и *Самолет*). Данный запрос относится к третьему типу.

3.17.2. Критерий запроса

Критерий запроса задает ограничения на свойства выбираемых концептов. Например, запрос в онтологии *Транспортные Средства*: "Все транспортные средства, максимальная скорость которых больше чем 100км/ч" на языке Knowledge.NET записывается следующем образом:

individuals of concepts **Vehicle** where **HasMaxSpeed** > 100

Формальное описание грамматики критерия запроса в форме Бэкуса-Наура представлено ниже:

EXP ::= OBJ_PROP OBJ_OP ID
OBJ_OP ::= contains | does_not_contain
EXP ::= DATA_PROP DATA_OP VALUE
EXP ::= EXP OP EXP
EXP ::= (EXP)
OP ::= or | and

Пояснения:

EXP — критерий запроса. Критерий запроса задается после ключевого слова “where”. Выражение может состоять из нескольких подвыражений, соединенных логическими связками «И» и «ИЛИ». Логическая связка «И» задается с помощью ключевого слова *and*. Логическая связка «ИЛИ» определяется с помощью ключевого слова *or*. Для определения приоритета логических операций используются круглые скобки. Например:

(**EXP1** or **EXP2**) and **EXP3**

OBJ_PROP — идентификатор объектного свойства. Если идентификатор совпадает с каким-либо ключевым словом запроса, идентификатор пишется в квадратных скобках. Например:
[individuals]

ID — идентификатор некоторого экземпляра. Если идентификатор совпадает с каким-либо ключевым словом запроса, идентификатор пишется в квадратных скобках.

OBJ_OP — двуместная связка между экземпляром и объектным свойством. Поддерживаются следующие типы связок: *contains* и *does_not_contain.*

Рассмотрим пример запроса с использованием объектного свойства. Запрос составлен на онтологии *Генеалогическое Дерево*, определенной выше:

individuals of **Human** where **HasChild** contains **Nina**

Поскольку домен свойства **HasChild** содержит в себе единственный концепт **Human,** на данной онтологии вышеприведенный запрос аналогичен следующему:

individuals where **HasChild** contains **Nina**

DATA_PROP — идентификатор простого свойства. Если идентификатор совпадает с каким-либо ключевым словом запроса, идентификатор пишется в квадратных скобках.

VALUE — значение свойства. Числовые значения указываются без кавычек. В качестве разделителя между целой и дробной частью используется точка (например: 3.1415926). Числа в шестнадцатиричной системе исчисления записываются в формате: 0x1234. Текстовые строки берутся в одиночные кавычки (например: ‘Hello World!’).

DATA_OP — двуместная связка между простым свойством и некоторым значением. Поддерживаются следующие типы связок:

- Логические свойства (bool):
  ’=’ (равно)
  ’!=’ (отлично от)

- Числовые свойства (int, float, long и другие):
  ‘=’ (равно)
  ’!=’ (отлично от)
  ‘<’ (больше)
  ‘>’ (меньше)
  ‘<=’ (больше или равно)

‘>=’ (меньше или равно)

- Строковые свойства (string):
  ’=’ (равно)
  ’!=’(отлично от)
  ’<’(больше в лексикографическом порядке)
  ‘>’ (меньше в лексикографическом порядке)
  ‘<=’ (больше или равно в лексикографическом порядке)
  ‘>=’ (меньше или равно в лексикографическом порядке)
  ’contains’ (содержит хотя бы одно вхождение текста)
  ’does_not_contain’ (не содержит ни одного вхождения текста)
  ’begins_with’ (начинается с текста)
  ’ends_with’ (заканчивается текстом)

Рассмотрим пример запроса, содержащего простые и объектные свойства, а также логические связки между ними. Данный запрос составлен для онтологии *Транспортные Средства*:

```
individuals of Automobile
  where (Color contains Red) or (HasMaxSpeed > 100 and HasMaxSpeed <= 250)
```

Данный запрос возвращает все экземпляры концепта *Автомобиль,* которые либо красного цвета, либо у которых максимальная скорость больше 100 км/ч и не превышает 250 км/ч.

Отметим, что в вышеприведенном примере конструкция «individuals of **Automobile**» имеет существенное значение. Результат запроса, приведенного ниже, может быть отличен от результата предыдущего запроса, поскольку выборка будет проводиться по всем экземплярам концепта **Vehicle,** а не только по экземплярам концепта **Automobile**:

```
individuals
  where (Color contains Red) or (HasMaxSpeed > 100 and HasMaxSpeed <= 250)
```

3.18. Выполнение запроса

Выполнение запросов происходит с помощью класса *QueryEngine,* определенного в библиотеке классов Knowledge.NET. Для выполнения запроса, используется статический метод:

```
ICollection<Individual> QueryEngine.execute(string query)
```

Данный метод в качестве параметра получает запрос и возвращает коллекцию экземпляров, полученную в результате выполнения запроса. В случае, если запрос содержит ошибку, метод execute() генерирует исключение *QueryValidationException*.

3.19. Наборы правил

Набор правил задает совокупность продукций, используемых для вывода. Основные компоненты набора правил:

1. Локальные переменные, объявленные в наборе правил в C# нотации;
2. Целевая переменная;
3. Правила, составляющие набор.

Форма представления набора правил близка к принятым в языках KEE и Турбо-Эксперт. Контекстом набора правил является онтология, в которой объявлен данный набор.

3.20. Описание структуры набора правил

Набор правил имеет следующую структуру:

```
ruleset IDEN
{
        C#_CODE
        goal V;

        [rule RULE_IDEN
        {
                if (B1)
                then S2;
                [else S3;]
                [priority INT_VALUE;]
        }]+
}
```

Пояснения:

IDEN — идентификатор набора правил

C#_CODE — некоторый код на C#, содержащий определения локальных переменных и констант, используемых в правилах вывода. Одна из этих переменных может быть помечена как целевая (goal) переменная. Целевая переменная может быть переопределена при вызове метода consult().

RULE_IDEN — имя правила;

B1 — условие правила; представляется в виде выражения, принимающего значения истина (true) или ложь (false);

S2 — заключение (код C#, исполняемый в случае истинности условия);

S3 — код C#, исполняемый в случае, если выражение **B1** ложно;

priority **INT_VALUE** — приоритет правила; данный параметр может использоваться машиной вывода для разрешения конфликтов (в случае, если на каком-либо шаге вывода несколько правил имеют истинное значение условия). Отметим, что машина вывода *ProductionSystem*, поставляемая вместе с библиотекой Knowledge.NET, не использует приоритеты.

3.21. Машина вывода

Библиотека классов Knowledge. NET предоставляет доступ к наборам правил через специальные интерфейсы, что позволяет разработчику реализовать свою собственную машину вывода.
Также библиотека содержит простую машину вывода, реализованную в виде класса *ProductionSystem*. Данный класс предоставляет следующие статические свойства и методы:

- *public static bool showAnnotations* – данное свойство задает, будут ли показываться аннотации, определенные в правилах в ходе консультации, или нет;

- *public static void consult (string ruleset);*
  public static void consult (string ruleset, string goal);
  public static void consult (string ruleset, string goal, ChainingMethod method);

Данные методы начинают консультацию, используя заданный набор правил.

Параметры:

ruleset – имя набора правил, с использованием которого будет осуществляться консультация;

goal – некоторая локальная переменная из контекста набора правил. Данный параметр не обязателен: если цель не определена, выводится цель, определенная в наборе правил;

method – стратегия логического вывода, может быть прямой или обратной (по умолчанию используется прямой вывод).

Особенности текущей реализации машины вывода:

- В настоящее время машина вывода поддерживает только прямую стратегию вывода;
- Метод разрешения конфликтов, используемый машиной, заключается в выборе правила, имя которого будет первым, исходя из лексикографического порядка;

3.22. Общая структура программы на Knowledge .NET

Программа на Knowledge .NET состоит из двух частей:

1. Исходный код C#;
2. Исходный код, специфичный для Knowledge .NET (Концепты, Свойства, Экземпляры)

Исходный код Knowledge .NET отделяется от (обычного) исходного кода C# с помощью ключевого слова ***"#ontology"***.

Из исходного кода на C# имеется возможность использовать концепты и их свойства, используя стандартный способ адресации: *имя_концепта.свойство*. Также вместе с Knowledge.NET поставляется библиотека, которая позволяет делать удобные запросы к онтологическим знаниям.

Пример программы на языке Knowledge.NET:

```
using System;

// C# native code
namespace HelloWorld
{
   class Hello
   {
      [STAThread]
      static void Main(string[] args)
      {
          Console.out.WriteLine ("Vehicle: " + Lada.HasName);
      }
   }
}

// Knowledge .NET specific code

#ontology "Vehicles"

#concepts
```

```
Color is_subconcept_of Thing;

Vehicle
{
  is_subconcept_of Thing;
  some_values_from HasName string;
  cardinality HasName 1;
}

Plane is_subconcept_of Vehicle;
Submarine is_subconcept_of Vehicle;
disjoint Plane, Submarine;

disjoint Color, Vehicle;

#properties

object property HasColor
{
        domain Vehicle;
        range Color;
}

functional datatype property HasName
{
        domain Vehicle;
        range string;
}
#individuals

individual Lada
{
        is_a Vehicle;
        HasName = "Lada";
}

#end_of_ontology "Vehicles"
```

4. Редактор знаний Knowledge Editor

Редактор знаний предназначен для визуализации, ввода и модификации знаний на языке Knowledge.NET. Редактор реализован как *расширение (add-in)* для Visual Studio.NET 2005, то есть он запускается автоматически вместе с Visual Studio, а его графический пользовательский интерфейс (GUI) фактически становится частью Visual Studio GUI. Такой подход

4.1. Создание проекта

Add-in позволяет пользователям просматривать и редактировать знания, представленные как в текстовом виде, так и в графическом.
Окно add-in появляется при запуске Visual Studio (Рис. 1).

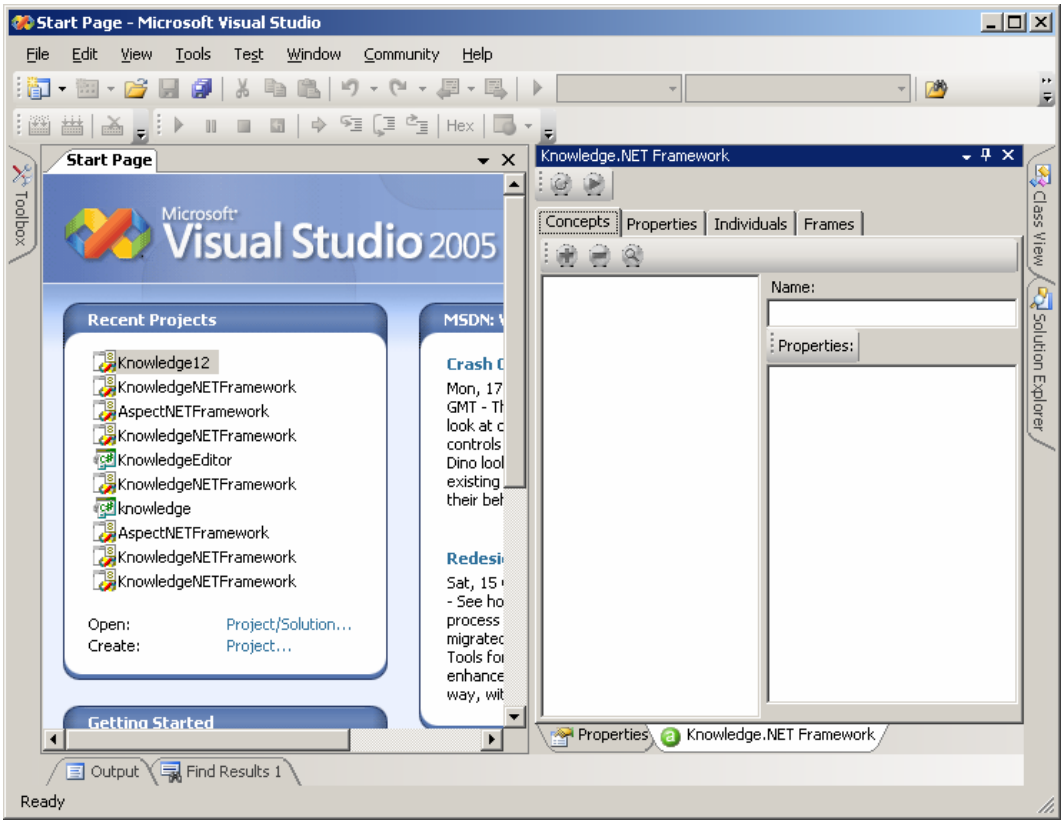


Рис. 1. Окно add-in при запуске Visual Studio.

Для удобства пользователей системы Knowledge.NET стандартный для Visual Studio.NET набор видов проекта расширен новым – *Knowledge*. Для того, чтобы создать новый проект в Knowledge.NET, пользователь может воспользоваться шаблоном “Knowledge”, который является Visual C# шаблоном (Рис. 2).

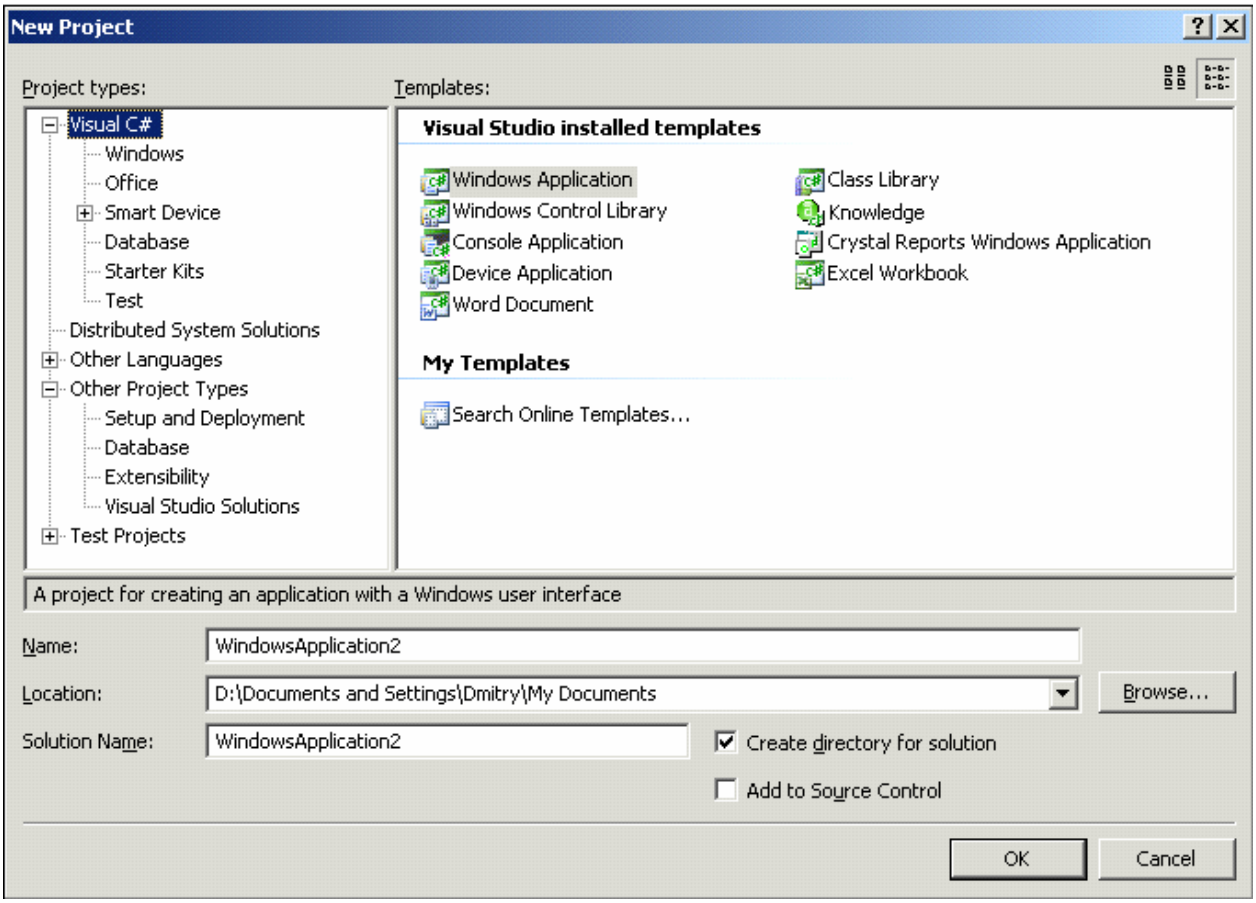


Рис. 2. Окно выбора шаблона.

Для поддержки этапов проектирования и реализации, мастер (wizard) генерирует заглушку (шаблон) в файле *template.expert* для будущего описания знаний пользователем на языке Knowledge.NET (Рис. 3).

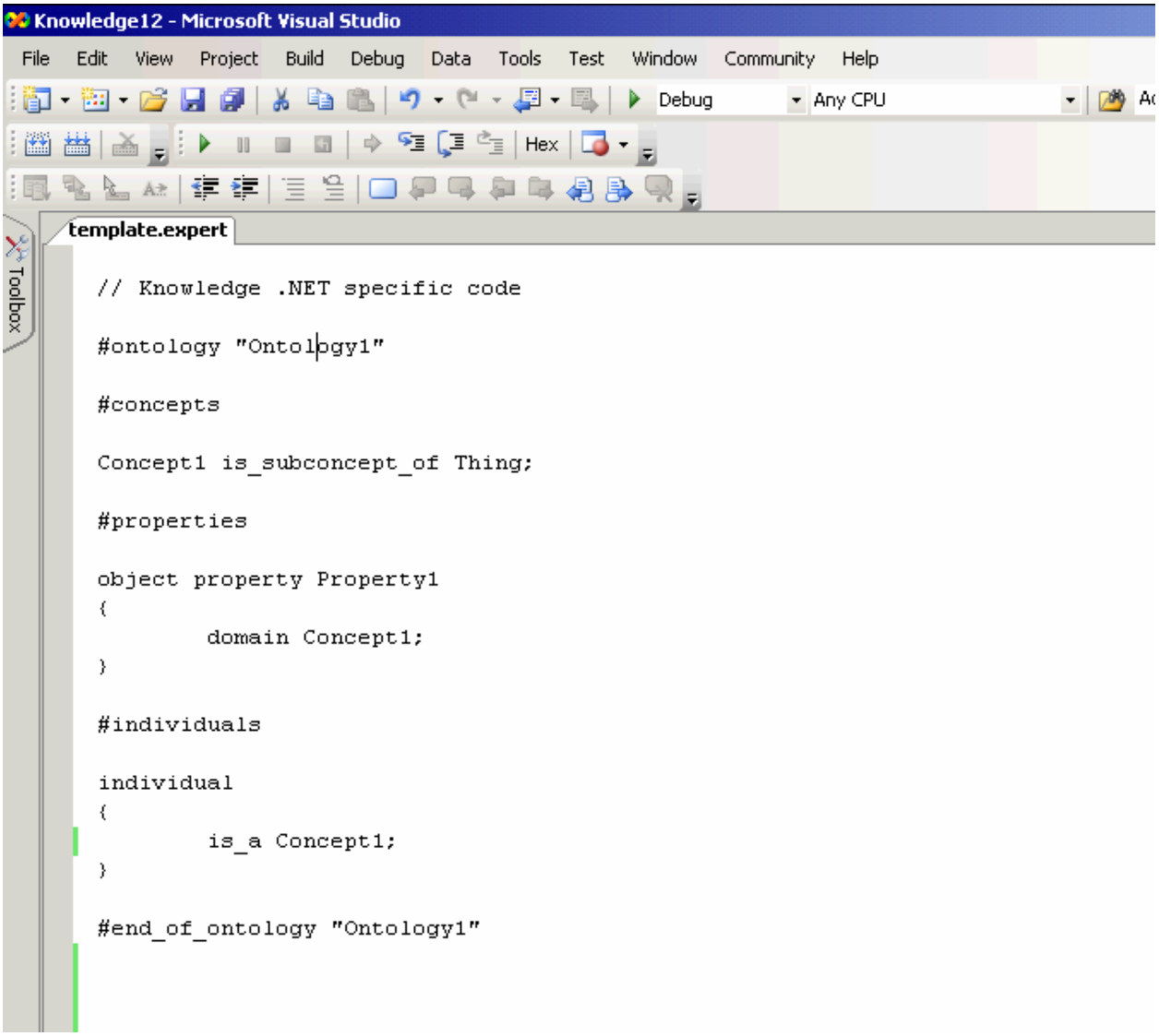


Рис. 3. Пример исходного кода на языке Knowledge.NET

4.2. Запуск конвертора

Интерфейс Knowledge.NET включает следующие кнопки на панели инструментов:  (“Refresh”) и  (“Convert”).

Функция “Refresh” (“Обновить”) позволяет выполнить синхронизацию текстового и графического представлений. Более подробно представления рассматриваются ниже.

Необходимой стадией разработки знаний является конвертирование знаний, описанных на языке Knowledge.NET, в исходный код C#. Запуск конвертора осуществляется нажатием на кнопку “Convert”. Результат конвертирования пользователь может увидеть в стандартном окне Output (Рис. 4).

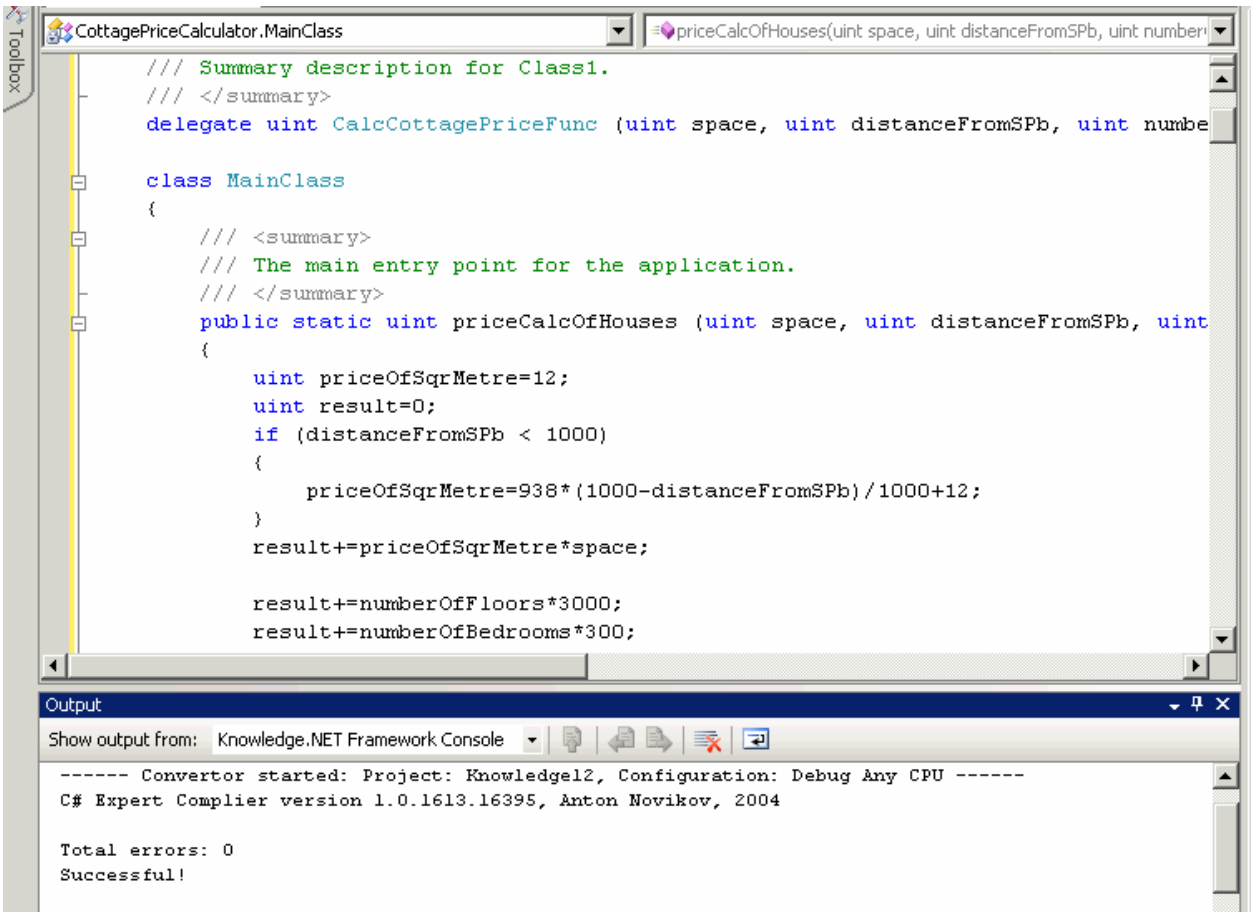


Рис 4. Сохранения результата конвертирования в стандартном окне Output

Запуск компилятора C# осуществляется при помощи стандартного интерфейса Visual Studio.NET 2005.

4.3. Текстовое представление знаний

Система Knowledge.NET позволяет пользователю редактировать текстовое представление знаний на языке Knowledge.NET. Создание и редактирование осуществляется с использованием стандартного интерфейса среды Visual Studio 2005 (Рис. 5).

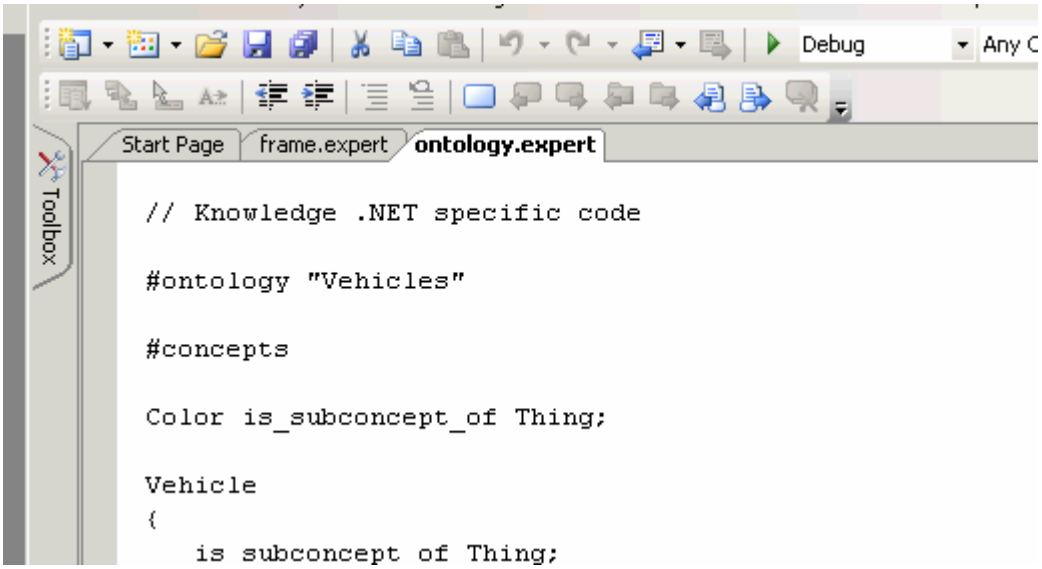


Рис. 5. Фрагмент текстового представления знаний на языке Knowledge.NET

В системе реализована функциональность, позволяющая автоматически синхронизировать текстовое и графическое представления. При редактировании текста происходит обновление графического представления “на лету”. В свою очередь, при модификации графического представления, пользователь может проследить соответствующие изменения в текстовом представлении.

4.4. Графическое представление знаний

Интерфейс Knowledge.NET включает следующие вкладки:

Каждая вкладка содержит графические компоненты, позволяющие пользователю просматривать и редактировать соответствующие виды знаний. Первые три вкладки предоставляют интерфейс с онтологическими знаниями, четвертая вкладка – с фреймовыми знаниями.

4.4.1. Вкладка Concepts

В данной вкладке (Рис. 6) пользователь может просматривать иерархическую структуру концептов, а также информацию о свойствах, которые принадлежат данному концепту.

Любой концепт в системе Knowledge.NET должен быть подконцептом концепта “Thing”.
Новый концепт может быть добавлен пользователем нажатием на кнопку  (“Add”), расположенной на панели инструментов вкладки “Concepts”. Удалить концепт можно нажатием на кнопку  (“Delete”).

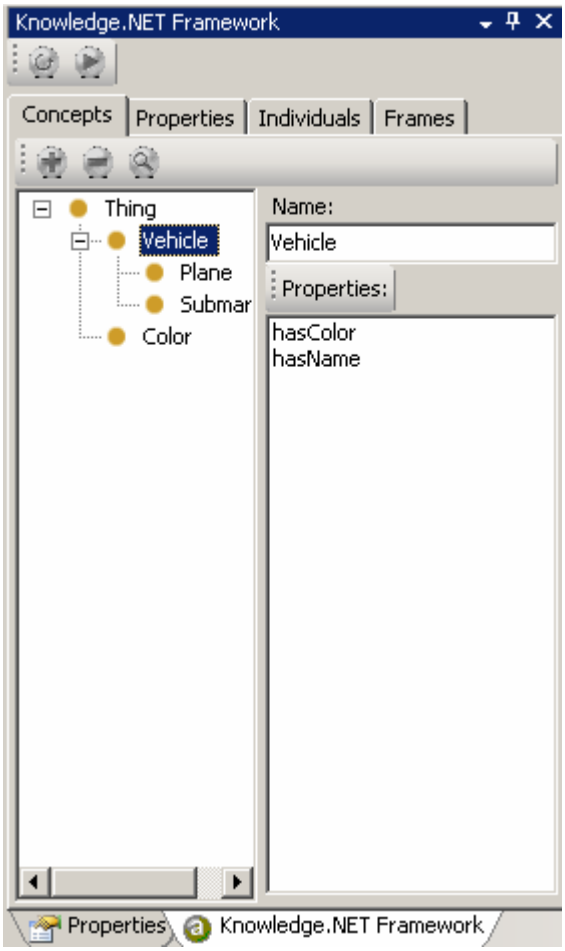


Рис. 6. Вкладка Concepts

Создание концептов

При создании концепта необходимо:
- специфицировать имя нового концепта
- выбрать концепты, которые будут родителями нового концепта

При создании концепта появляется модальный диалог (Рис. 7), в котором пользователю необходимо указать эту информацию.

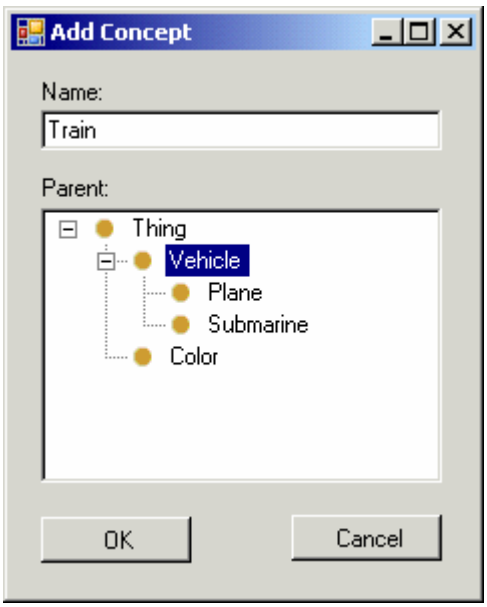
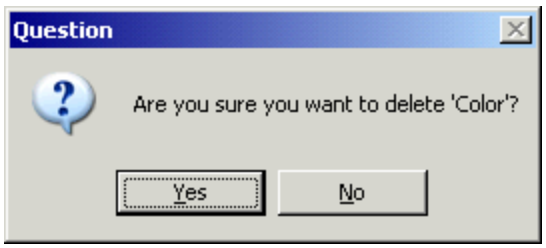


Рис. 7. Модальное окно при добавлении концепта

После добавления концепта обновляется как графическое, так и текстовое представление.

Удаление концептов

При удалении концепта пользователю необходимо подтвердить выполнение операции.



После удаления концепта обновляется как графическое, так и текстовое представление.

4.4.2. Вкладка Properties

В данной вкладке (Рис. 8) пользователь может просматривать иерархическую структуру свойств, а также информацию о свойствах - имя, тип значений, домен, область значений.

Новое свойство может быть добавлено пользователем нажатием на кнопку  (“Add”), расположенную на панели инструментов вкладки “Properties”. Удалить свойство можно нажатием на кнопку  (“Delete”).

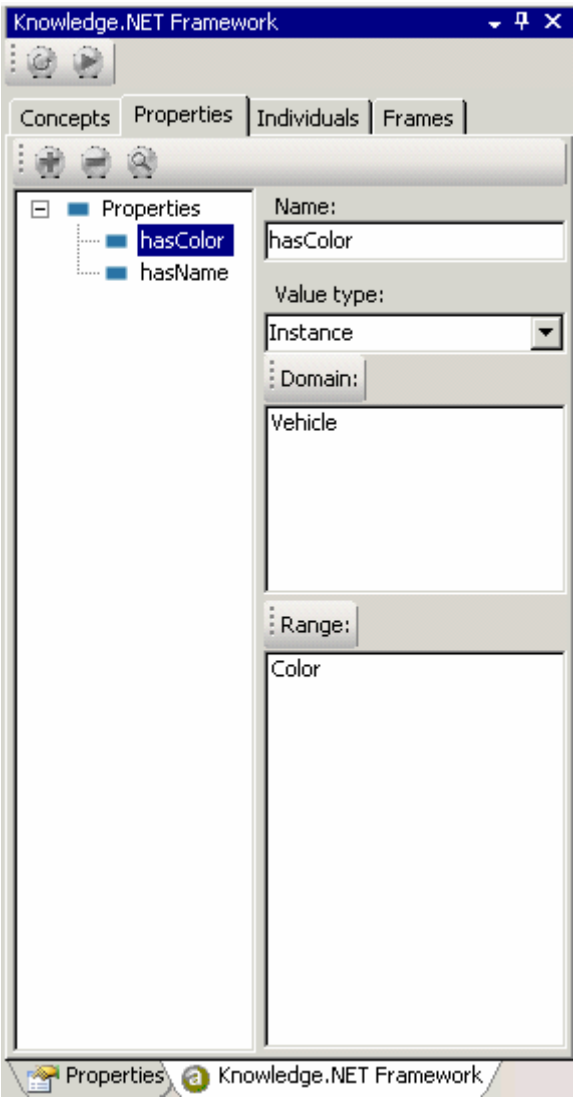


Рис. 8. Вкладка Properties

Создание свойств

При создании свойства необходимо:
- специфицировать имя нового свойства
- специфицировать домен, тип и область значений

При создании свойства появляется модальный диалог (Рис. 9), в котором пользователю необходимо указать эту информацию.

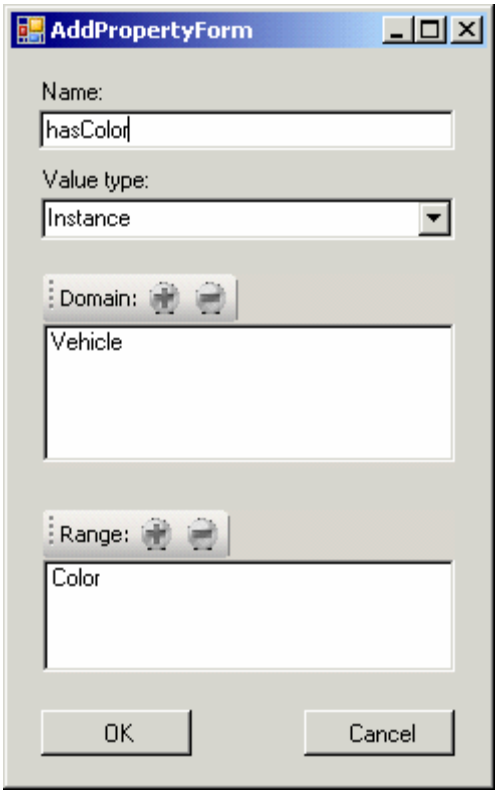
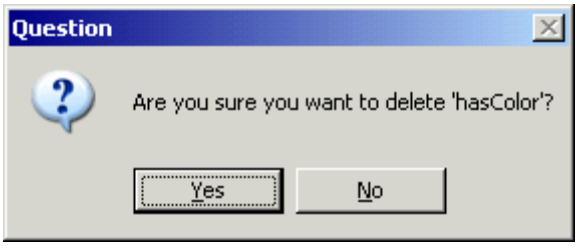


Рис. 9. Модальное окно при добавлении свойства

После добавления свойства обновляется как графическое, так и текстовое представление.

Удаление свойств

При удалении свойства пользователю необходимо подтвердить выполнение операции.



После удаления свойства обновляется как графическое, так и текстовое представление.

4.4.3. Вкладка Individuals

В данной вкладке (Рис. 10) пользователь может просматривать иерархическую структуру концептов, а также экземпляры, реализующие концепты.

Новый экземпляр может быть добавлен пользователем нажатием на кнопку  (“Add”), расположенную на панели инструментов вкладки “Individuals”. Удалить экземпляр можно нажатием на кнопку  (“Delete”).

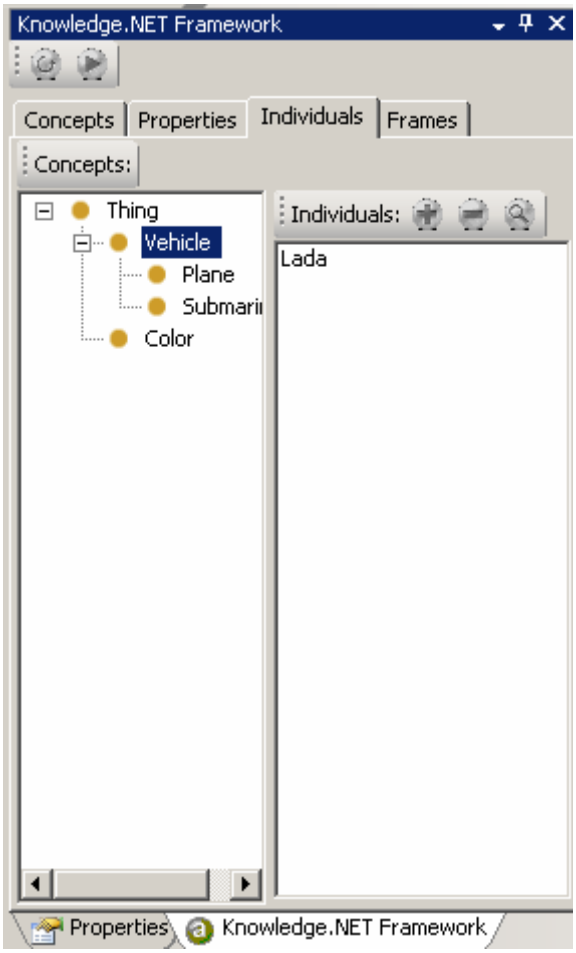


Рис. 10. Вкладка Individuals

Создание экземпляров

При создании экземпляра необходимо:
- специфицировать имя нового экземпляра
- выбрать концепт, который реализуется экземпляром
- специфицировать значения свойств

При создании экземпляра появляется модальный диалог (Рис. 11), в котором пользователю необходимо указать информацию.

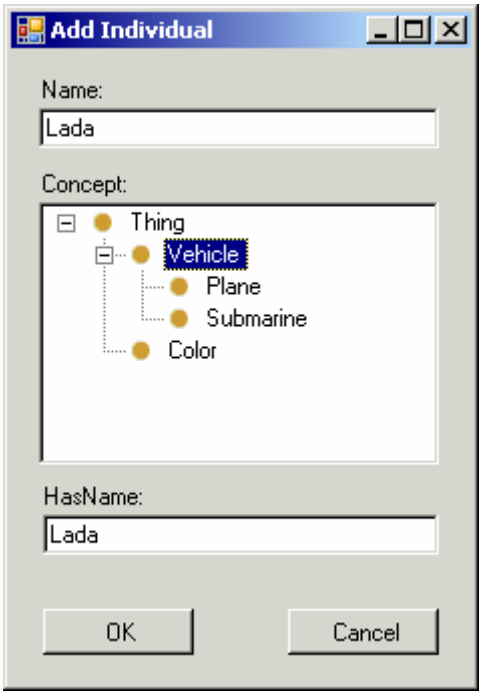
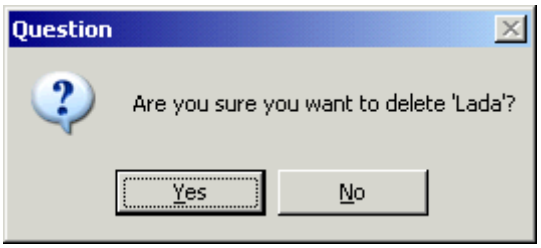


Рис. 11. Модальное окно при добавлении экземпляра

После добавления экземпляра обновляется как графическое, так и текстовое представление.

Удаление экземпляров

При удалении экземпляра пользователю необходимо подтвердить выполнение операции.



После удаления экземпляра обновляется как графическое, так и текстовое представление.

4.4.4. Вкладка Frames

В данной вкладке (Рис. 12) пользователь может просмотреть общую структуру фреймовых знаний.

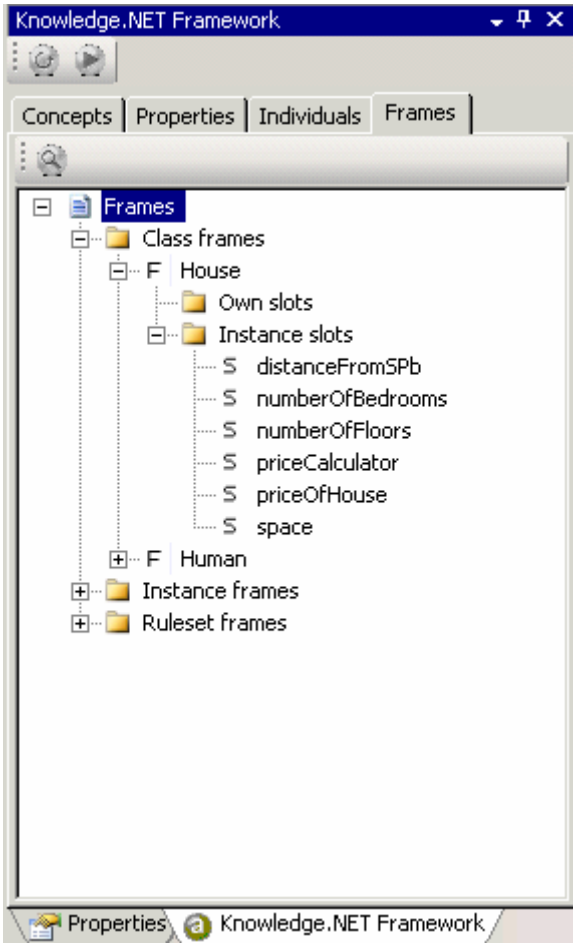


Рис. 12. Вкладка Frames

4.5. Связь представлений

Как было сказано ранее, обновление представлений происходит синхронно. Таким образом, пользователь может выбрать и использовать наиболее удобную для себя форму представления.

4.5.1. Навигация в браузере знаний

Выбрав существующий объект из графического представления (концепт, слот, экземпляр, фрейм) пользователь может по нажатию на кнопку  (“Browse”) мышкой перейти к определению этого объекта в текстовом представлении (Рис. 13).

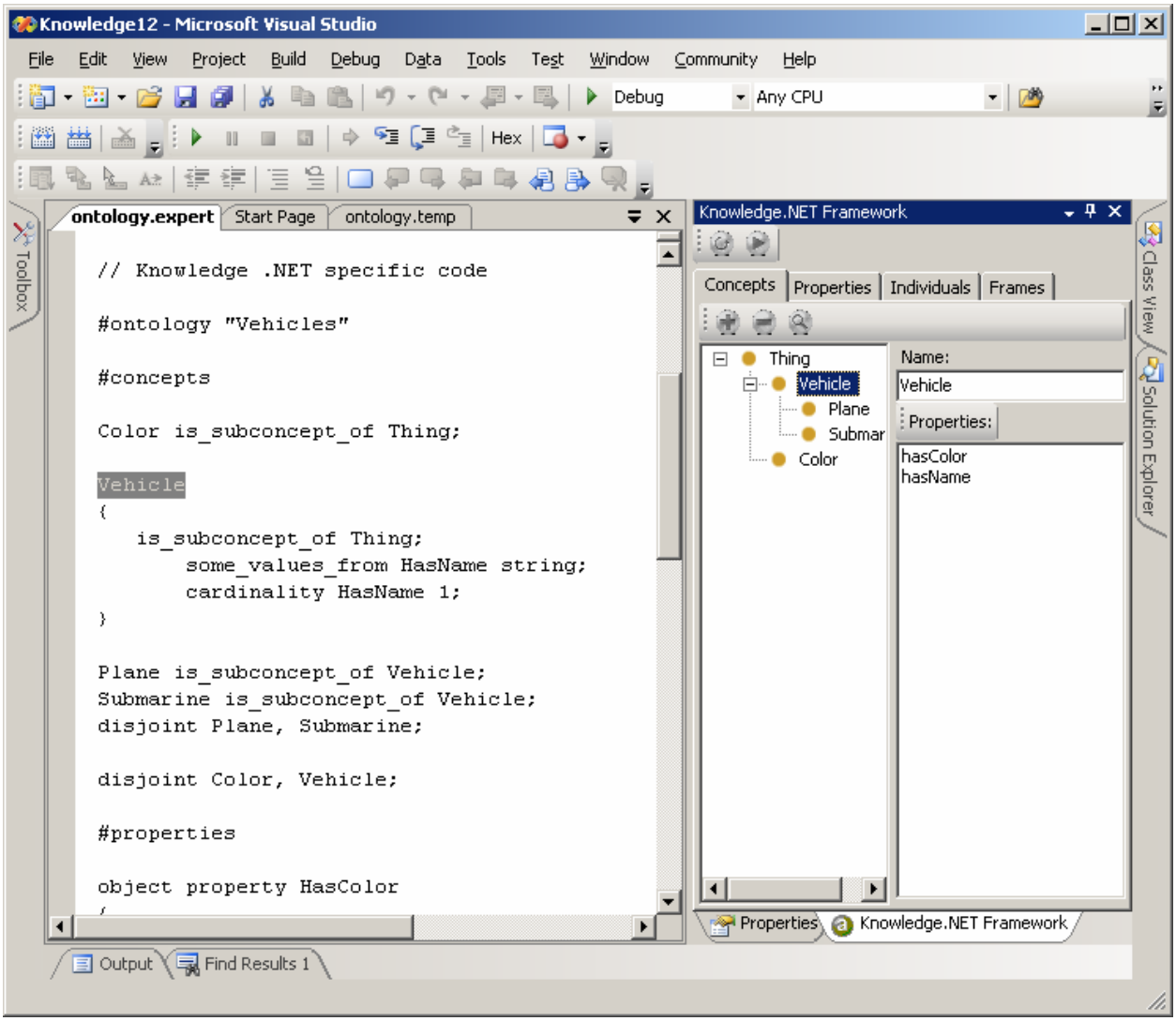


Рис. 13. Пример навигации от графического представления к текстовому представлению

5. Система извлечения знаний Knowledge Prospector

Система извлечения знаний KnowledgeProspector, входящая в состав проекта Knowledge.NET, предназначена для извлечения знаний из текстовых (в частности, HTML) документов, и их представления в формате Knowledge.NET. Основное назначение системы – извлечение знаний из Интернета.

Система основана на алгоритме извлечения знаний, состоящего из трех основных частей:

- *Морфологический анализ* текста и построение по его результатам *набора сущностей*.
- *Семантический анализ* наборов сущностей и построение по его результатам *графа знаний*.
- *Анализ графа знаний.*

5.1. Описание работы алгоритма извлечения знаний

Первый этап работы алгоритма - это перевод текста с естественного языка в набор *сущностей*.

В текущей версии системы реализован анализ текста на русском языке.

Данный этап осуществляется при помощи разного вида словарей:

- *MRD-словаря*. Данный словарь содержит около 170 тысяч лексем русского языка. Словарь позволяет производить морфологический анализ слов и определять грамматические характеристики слова.

- *XML-словаря*. Это словарь, который содержит дополнительные конструкции для перевода слов языка в сущности. Например, для слова "подкласс" можно задать свойство, которое позволит при последующем анализе сущностей рассуждать о связях между словами, связанными с этим словом.
- Словарей, создаваемых пользователем. Для создания словаря необходимо реализовать интерфейс *IDictionary* и подключить словарь к экземпляру класса *DictionaryManager*.

Второй этап работы заключается в анализе набора сущностей. Для анализа наборов сущностей используются *правила построения графа знаний*. Одним из самых эффективных правил, с точки зрения получения количества связей и удобства настройки, является правило применения шаблонов обработки сущностей. Результат работы второго этапа – построение графа знаний. Вершины графа представляют сущности, а дуги – связи между ними.

Последующий этап заключается в анализе графа знаний. В задачи данного этапа входит:
- объединение различных свойств в единые классы. Например, свойства "большой", "огромный", "маленький" имеет смысл объединить в один класс.
- Удаление из графа избыточных связей. Например, если у родительского класса имеется некоторое свойство, то у его наследников его можно не указывать.

5.2. Первичный анализ входного текста

5.2.1. Словари

Русский морфологический словарь Диалинг

Словарь позволяет проводить морфологический анализ слов русского языка и получать грамматические характеристики содержащихся в нём слов. Позволяет определить нормальную форму слова, все словоформы слова и их морфологические характеристики.

XML- словарь

Словарь позволяет описывать разные типы сущностей, но основным его применением является расширенное описание сущностей типа "связь". Для них можно указать тип *связи*, который соответствует типу *отношения*, и свойство *связи* (*отношения*).

5.2.2. Сущности

Выделение сущностей является результатом первичного анализа входного текста. Существует два вида сущностей:
- *Обычные* сущности ("неизвестная", "разделитель", "связь"). Эти сущности не могут быть добавлены в граф знаний.
- *Настоящие* сущности ("класс", "свойство", "тип данных", "индивидуал"). Сущности могут быть вершинами графа знаний.

В простейшем случае сущность содержит только слово естественного языка, из которого она получена ("неизвестная" сущность). Если слово было найдено в морфологическом словаре, то ему приписываются морфологические и грамматические свойства, которые впоследствии можно использовать при анализе наборов сущностей.

Сущность "класс" (class)

То же самое, что и понятие "класс" в системе OWL [11]. Если в словаре указано, что слово является существительным, то оно автоматически будет преобразовано в сущность типа "класс".

Сущность "свойство" (property)

То же самое, что и понятие "свойство" в OWL [11]. Если в словаре указано, что слово является прилагательным, то оно автоматически будет преобразовано в сущность типа "свойство".

Сущность "разделитель" (separator)

Представляет различные символы пунктуации. Разделителями считаются следующие знаки:

- "." – точка.
- " ", "\t", "\n" – пробел, символ табуляции, символ новой строки.
- "," – запятая.
- ":" – двоеточие.
- ";" – точка с запятой.
- "(", ")" – открывающая и закрывающая круглые скобки.
- "[", "]" – открывающая и закрывающая квадратные скобки.
- "{", "}" – открывающая и закрывающая фигурные скобки.
- "<", ">" – знак "меньше" и "больше".

Сущность "время" (datetime)

Представляет дату и время. Содержит следующие поля:

- Год
- Месяц
- День
- Час
- Минута
- Секунда

Сущность *время,* в отличие от типа данных C# *DateTime*, позволяет указывать значения только для некоторых полей, например, *месяц*; все остальные поля могут быть пустыми.

Сущность "Целое число" (integer)

Представляет целое число. Если в тексте встретится число, то оно будет преобразовано в сущность "целое число" на этапе первичного анализа.

Сущность "индивидуал" (individual)

То же самое, что и понятие "индивидуал" в OWL [11].

Сущность "связь" (relationship)

Слово, которое было описано в XML-словаре как связь между другими словами. Содержит такие параметры, как свойства и тип связи. Свойства могут быть:

- Транзитивными
- Симметричными
- Функциональными.

Тип связи – это то "отношение", которое будет построено при семантическом анализе сущностей.

Сущность "неизвестная" (unknown)

Если для слова не удалось определить сущность одного из предыдущих типов, то ему ставится в соответствие "неизвестная" сущность.

5.3. Семантический анализ наборов сущностей

5.3.1. Отношения

Отношения используются для связей между сущностями в графе знаний. Отношение может быть построено только между *настоящими сущностями*.

Отношение между "свойством" и "классом"

Связывает сущность типа "свойство" с сущностью типа "класс". Фактически при этом *классу* приписывается данное *свойство*.

Отношение "подкласс"

Обозначает, что одна сущность является подклассом другой сущности. Применяется для *классов* и *свойств*.

Отношение "эквивалентность"

Обозначает эквивалентность между двумя сущностями. Применяется к сущностям типа "класс".

5.3.2. Правила построения графа знаний

Построение графа знаний основано на наборе правил, которые последовательно выполняются на всём наборе сущностей. В текущей версии системы реализованы и применяются следующие правила:

- Правило применения шаблонов обработки наборов сущностей.
- Правило добавления всех *настоящих сущностей* в граф знаний.

5.3.3. Шаблоны обработки набора сущностей

Шаблоны являются эффективным и удобным средством построения графа знаний. С помощью шаблонов может быть реализовано, например, распознавание дат, чисел, фамилий, имен и отчеств людей. Также шаблоны позволяют создавать новые сущности, которые могут участвовать в дальнейшем анализе текста. С помощью шаблонов можно организовать особую обработку частицы "не" и вопросительных предложений. Многие из шаблонов не зависят от языка, на котором написан исходный текст.

Шаблон состоит из трёх частей, отделенных друг от друга с помощью двух последовательных символов: "->". Первая часть шаблона – целое число, которое представляет приоритет шаблона. Шаблон с более высоким приоритетом будет выполнен раньше, чем шаблон с более низким приоритетом. Если приоритеты совпадают, то шаблоны выполняются параллельно. Не рекомендуется, чтобы два шаблона, которые выполняют схожие действия, имели одинаковый приоритет, так как это нарушит их работу. Пример - шаблон, который может удалить некоторую сущность из набора, и шаблон, который будет использовать эту сущность при построении связи. Во второй части шаблонаописываются ограничения на набор сущностей (будем называть эту часть *регулярным выражением над сущностями*), для которых будут применены действия из третьей части шаблона (*обработчик набора сущностей*).

5.3.3.1. Язык описания регулярных выражений над сущностями

5.3.3.1.1. Описание

Язык предназначен для описания требований к структуре набора сущностей. Если набор удовлетворяет этим требованиям, к нему применяются правила обработки, описанные далее. Регулярное выражение состоит из последовательности элементов. Элементы отделяются друг от друга пробелами. Каждый элемент состоит из двух частей - ограничений на сущность и ограничений на встречаемость этого элемента.

Ограничения на сущность.

Ограничения на сущность заключаются в квадратные скобки. В случае, если ограничение состоит из одного элемента, скобки можно опустить.

Ограничения на сущность описываются с помощью бинарных логических операций:

- "&" – логическое "И".
- "|" – логическое "ИЛИ".

Если бинарная операция не указана, по умолчанию подразумевается операция ИЛИ.
Порядок выполнения операндов указывается с помощью круглых скобок. Если скобки не указаны, то порядок определяется традиционным соотношением приоритетов операндов (самый высокий приоритет - у операции "НЕ", затем - "И", затем "ИЛИ"). Порядок выполнения в случае одинаковых приоритетов операций - слева направо.

Каждый элемент ограничения может быть двух видов:

- *Мета-элемент*, проверка которого будет осуществляться с помощью одного из ниже указанных правил. Указывается с помощью символа "#" и последующего описания правила.
- *Слово на естественном языке* (любая последовательность символов, начинающаяся не с символа "#"). В этом случае проверка элемента будет считаться успешной, если сущность построена из того же слова, что и указанное в элементе.

Существуют следующие виды правил:

- "#E" – проверки принадлежности сущности к некоторому классу. Класс указывается через точку после "#E". Доступны следующие сокращения для классов:
  - "P" – сущность является *свойством* (сокращение от property).
  - "C" – сущность является *классом* (сокращение от class).
  - "I" – сущность является *индивидуалом* (сокращение от individual).
  - "S"– сущность является *разделителем* (сокращение от separator).
  - "U" – неизвестная сущность (сокращение от unknown).
  - "Int" – сущность является *целым числом* (сокращение от Integer).
  - "DT" – сущность является *временем* (сокращение от DateTime).
- "#M" – проверка части речи, к которой относится слово, из которого была построена сущность. Часть речи указывается через точку после "#M". Используются англоязычные обозначения частей речи:
  - Numeral – числительное
  - Noun – существительное
  - Adjective – прилагательное
  - Verb – глагол.
- "#S" – проверка принадлежности к одному из специальных классов слов. Класс указывается через точку после "#S". Доступны следующие классы:
  - Month – все слова, обозначающие месяц. Все слова указаны в специальном файле.

Ограничение на встречаемость.
Ограничение на встречаемость записывается сразу же после ограничений на сущность.

- Пустой символ – обозначает, что данному элементу должна удовлетворять ровно одна сущность.
- Символ "+" – что элементу должна удовлетворять как минимум одна сущность.
- Символ "*" – обозначает, что элементу может удовлетворять любое число сущностей (в том числе - 0).
- Символ "?" – элементу может соответствовать не более чем одна сущность.

5.3.3.1.2. Примеры

- #P+ #C – означает все наборы сущностей, которые можно разбить на два непустых подмножества - в первом из них все сущности будут *свойствами*, а во втором будет только одна сущность типа *класс*.
- [#P #M.Adjective]+ #C – означает все наборы сущностей, которые можно разбить на два непустых подмножества, в первом из которых все сущности являются *свойствами* или

должны быть представлены *прилагательными*, а во втором содержится только одна сущность типа *класс*.

- #С *является подмножество* #С – все наборы из четырех сущностей, первая и четвертая из которых являются классами, а значения второй и третьей равны соответственно "является" и "подмножество". Несоответствие русской орфографии в примере объясняется тем, что для модуля морфологической обработки слов все слова в подобном шаблоне указываются в *нормальной форме* (единственном числе, именительном падеже и т.д.). При морфологическом анализе будут правильно распознаны другие вормы данного слова.

5.3.3.2. Язык описания обработчиков набора сущностей

5.3.3.2.1. Описание

Язык предназначен для описания действий, которые требуется выполнить над набором сущностей, удовлетворяющих регулярному выражению шаблона. Обработчик сущностей состоит из набора *действий*, отделяемых друг от друга пробелами. Действия делятся на два вида:

- Действия, создающие новые сущности
- Действия, модифицирующие набор сущностей или строящие отношения между ними.

Ниже приведены список всех возможных действий и их описание.

Действие Add

Назначение: Используется для добавления новых сущностей в граф знаний. Эти сущности будут добавлены в набор и не будут анализироваться шаблонами или другими правилами второго этапа.
Синтаксис: *Add (действие-создающее-сущность)*
Описание: Выполняет действие, создающее сущность, и добавляет полученную сущность в граф знаний.

Действие Replace

Назначение: Используется для замены нескольких существующих сущностей новой сущностью.
Синтаксис: *Replace (index, count, действие-создающее-сущность).*
Описание: Заменяет *count* сущностей, начиная с *index*, на новую сущность, созданную указанным действием.

Действие NewDateTime

Назначение: Используется для создания новой сущности типа *время*.
Синтаксис: *NewDateTime (параметр= индекс, ...)*
Пример: *NewDateTime (d=1, m=2, y=3)*
Описание: Создает сущность *время* и инициализирует его параметры значениями сущностей, указанными в качестве индексов. Допустимы следующие параметры:

- "Y" – год.
- "M" – месяц.
- "D" – день.
- "H" – час.
- "Min" – минута.
- "S" – секунда.

Действие PR (Property Relationship)

Назначение: Используется для связывания *свойства* с *классом*.
Синтаксис: *PR (index1, index2)*
Описание: В качестве аргументов указываются индексы элемента регулярного выражения. Строится декартово произведение множеств сущностей, удовлетворяющих первому элементу (номер которого указан в первом аргументе) и второму элементу (его номер

– второй аргумент). Все пары из построенного произведения связываются между собой с помощью объекта класса *PropertyRelationship*.

Действие SC (Subclass Relationship)

Назначение: Используется для обозначения того факта, что один *класс является* подклассом другого.

Синтаксис: *SC (index1, index2)*

Описание: Действие налогично действию PR, за исключением того, что сущности в данном случае связываются при помощи экземпляра класса *SubclassRelationship*.

5.3.3.3. Примеры работы шаблонов

Примеры шаблонов без использования слов на естественном языке:

1) Набор сущностей:	P1 P2 P3 C1
Пример на естественном языке:	большой красивый светлый дом
Шаблон:	#E.P+ # E.C -> PR(1, 2)
Построенные связи:	PR(P1, C1) PR(P2, C1) PR(P3, C1)
2) Набор сущностей:	Int1(15) C1("январь") Int2(1994) C2("год")
Пример на естественном языке:	15 января 1994 года.
Шаблон:	#E.INT #S.Month #E.INT -> Replace(0, 3, NewDateTime(d=2, m=1, y=0))
Выполненные действия:	Сущности Int1, C1, Int2 будут заменены на новую сущность DateTime, у которой будут установлены следующие значения параметров: день = 15, месяц = 1, год = 1994.

Примеры шаблонов со словами на естественном языке:

1) Набор сущностей:	C1 E("бывают") P1
Пример на естественном языке:	дома бывают кирпичными
Шаблон:	#E.C бывают #E.P -> PR(3, 1)
Построенные связи:	PR(P1, C1)
2) Набор сущностей:	C1 E("(") P1 E("и") P2 E(")")
Пример на естественном языке:	дома (кирпичные и панельные)
Шаблон:	#E.C (#E.P и #E.P) -> PR(3,1) PR(5,1)
Построенные связи:	PR(P1, C1) PR(P2, C2)

6. Конвертор знаний в формат KIF

Конвертор знаний системы Knowledge.NET выполняет конвертирование из языка Knowledge.NET в формат языка KIF (Knowledge Interchange Format) [10], с целью их последующего использования экспертами, работающими в других системах, поддерживающих формат KIF. Подобная компонента в системе Knowledge.NET необходима, так как формат KIF уже в течение ряда лет считается de facto стандартом в области представления знаний.

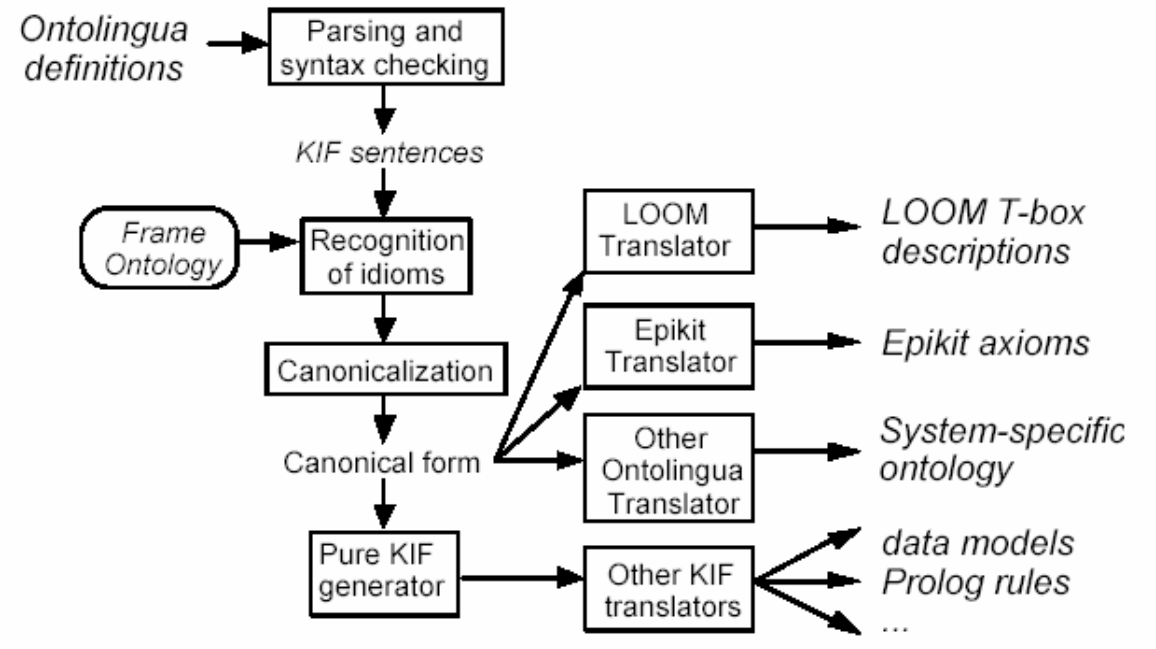
Процесс конвертирования разбит на два этапа. На первом этапе происходит разбор входного документа в формате Knowledge.NET и создается его внутреннее представление, соответствующее

конструкциям языка Ontolingua [12].. Данный подход удобен тем, что пользователь может конвертировать полученный документ в другие языки представления знаний, используя набор трансляторов системы Ontolingua (в частности, транслятор в KIF).

На втором этапе происходит преобразование из внутреннего представления в формат языка KIF.

6.1. Обзор системы Ontolingua

Система Ontolingua [12] представляет собой окружение разработки, предоставляющее набор функций для работы с онтологиями (просмотр, создание, редактирование и использование). Язык Ontolingua является надстройкой над KIF и имеет конструкции для представления фреймов/онтологий. Также система имеет ряд трансляторов в другие языки представления знаний – Loop, Epikit, Express, Generic-Frame, Algernon, IDL и в базовый язык KIF. На рисунке ниже показан процесс трансформации:



6.2. Общее описание конструкций Ontolingua, получаемых после конвертирования

Фреймы и концепты описываются следующей конструкцией (будем называть ее структурой, когда речь идет не о конкретном виде знаний):

(DEFINE-CLASS <*structure-name*> (<*argument-list*>)
<docstring>
{:def | :iff-def} <*sent-with-arg-vars*>
[:constraints <*sent-with-arg-vars*>]
[:equivalent <*sent-with-arg-vars*>]
[:sufficient <*sent-with-arg-vars*>]
[:default-constraints <*sent-with-arg-vars*>]
[:axiom-def <*sent-without-arg-vars*>]
[:axiom-constraints <*sent-without-arg-vars*>]
[:axiom-defaults (<*sent-without-arg-vars*>*)]
[:class-slots (<*slot-spec*>*)]
[:instance-slots (<*extended-slot-spec*>*)]
[:default-slot-values (<*slot-spec*>*)]
[:issues <*issue-tree*>])

<*docstring*> - Комментарий, в котором описан тип исходной структуры на языке Knowledge.NET (класс, фрейм, онтология). При дальнейшем конвертировании в KIF игнорируется.

<*structure-name*> - Идентификатор структуры (например, Human). Результат конвертирования концепта представляет собой <ontology-name> **.** <structure-name>, где ontology-name –имя онтологии, в которой задается концепт с именем structure-name.

<*argument-list*> - Список переменных KIF, который используется в остальных конструкциях, имеющих атрибут <*sent-with-arg-vars*> (будет описано ниже). Каждая переменная имеет вид ?IDENTIFICATOR.

Следующие ключевые слова могут располагаться в произвольном порядке:

:DEF - Выражение KIF, описывающее необходимые условия, которым должна удовлетворять структура, и использующее символьный аргумент как свободную (не связанную) переменную. Например, выражение

```
(define-class human (?human)
  "The human animal."
  :def (subclass-of animal ?human))
```

задает структуру Человек, являющуюся подклассом структуры Животное.

:IFF-DEF - Выражение, подобное :DEF, но, в отличие от нее, задает еще и необходимые условия:

```
(define-class female-person (?person)
  "female humans"
  :iff-def (and (subclass-of human ?person)
                (= (gender ?person) female)))
```

Смысл данной записи: женщина – это человек, имеющий женский пол.

Выражения, относящиеся к этим ключевым словам, задают только общее *определение* структуры и не затрагивают знания о том, чему должны удовлетворять ее экземпляры.
Нельзя использовать ключевые слова :DEF и :IFF-DEF одновременно.

:CONSTRAINTS – Значение этого ключевого слова является выражением KIF, которое определяет необходимые ограничения на части структуры и не является частью определения. Это выражение определяет условия, которые должны быть истинными для всех *экземпляров* структуры.

:SUFFICIENT – Выражение KIF, определяющее достаточные условия, которым должны удовлетворять части структуры.

:EQUIVALENT - Значение этого ключевого слова является выражением KIF, которое определяет необходимые и достаточные ограничения на части структуры и не является частью определения. По смыслу эквивалентно разделу :IFF-DEF.

:DEFAULT-CONSTRAINTS - Значением этого ключевого слова является список выражений KIF, которые определяют то, что «является по умолчанию» для членов структуры.

:AXIOM-DEF, :AXIOM-CONSTRAINTS и :AXIOM-DEFAULTS являются аналогом :DEF-IFF-DEF, :CONSTRAINTS и :DEFAULT-CONSTRAINTS, соответственно, но не используют ARGUMENT-LIST.

В разделе :AXIOM-DEF, например, описываются отношения наследования у концептов в форме двуместных отношений Subclass-of <потомок> <предок>.

Например:

```
(define-class human (?human)
  "The human animal."
  :axiom-def (subclass-of human animal))
```

Смысл: human является подконцептом animal.

Множественное наследование записывается следующим образом:

(and (subclass-of A B) (subclass-of A C) …)

:ISSUES – Значение этого аргумента является Lisp-список строк, описывающих вспомогательную информацию о структуре (время создания, автор, пояснения и т.д.). Например, следующая аннотация концепта в Knowledge.NET

```
annotation
{
        version_info "1.0.0";
        label "Some text";
        comment "This is comment";
        created_by "Ivanov I.I.";
        creation_date "10.05.2005";
}
```

после конвертирования примет вид:

```
:issues
        ((:version-info "1.0.0")
         (:label "Some text")
         (:comment "This is comment")
         (:created-by "Ivanov I.I.")
         (:creation-date "10.05.2005"))
```

При дальнейшем конвертировании в KIF игнорируется.

Ontolingua поддерживает дополнительный синтаксис для определения выражений в стиле фрейм/слот. Для этого используются разделы :CLASS-SLOTS (собственные слоты фрейма как класса), :INSTANCE-SLOTS (слоты, принадлежащие экземплярам, унаследованных от данного класса) и :DEFAULT-SLOT-VALUES (задает значения по умолчанию).
Эти конструкции эквивалентны :DEF-IFF-DEF, :CONSTRAINTS и :DEFAULT-CONSTRAINTS, соответственно, и преобразуются к ним на втором этапе конвертирования. Поэтому рассмотрим только особые случаи использования.
Каждое выражение SLOT-SPEC для :CLASS-SLOTS и :DEFAULT-SLOT-VALUES имеет следующий формат:

```
(slot-name slot-value1 slot-value2 ...)
```

где `SLOT-NAME` – имя бинарного отношения. Первый аргумент этого отношения неявно выражает структуру (т.е. фрейм), с которой этот слот связан. Каждый SLOT-VALUE – второй аргумент.
При дальнейшей трансформации примут вид следующих выражений KIF:

```
(and (slot-name frame slot-value-1)
(slot-name frame slot-value-2)... )
```

Например, отношение наследование для фреймов задается в этом разделе и имеет вид:

(define-class A ...
:class-slots (subclass-of B C D ...)
...
), что эквивалентно

(define-class A ...

:DEF (and (subclass-of A B)
(subclass-of A C)
(subclass-of A D)...)
...
)

:INSTANCE-SLOTS – аналогично предыдущей конструкции, но для задания фасетов слота. Имеет вид:

(facet facet-value1 facet-value2...).

что эквивалентно конструкции KIF:

(and (facet class slot facet-value1)
(facet class slot facet-value2)

Для определения экземпляров структуры используется следующая конструкция

(DEFINE-INSTANCE <*instance-name*> (<*class-name*>+)
[<*docstring*>]
[:= <*term-expression-without-arg-vars*>]
[:axiom-def <*sent-without-arg-vars*>]
[:slots (<*slot-spec*>*)]
[:issues <*issue-tree*>])

где (<docstring>, :AXIOM-DEF и :ISSUES – описаны выше для конструкции define-class, :SLOTS – эквивалентно разделу :CLASS-SLOTS конструкции define-class)

<*class-name*>+ - список всех структур, реализуемых данным экземпляром.

TERM-EXPRESSION-WITHOUT-ARGS-VARS – выражение KIF, определяющее значение экземпляра, например:

(define-instance PI (real-number)
"PI is the ratio of the perimeter of a circle to
its diameter."
:= 3.14159)

Для определения свойств используется конструкция:

(DEFINE-RELATION <*relation-name*> (<*var*>+)
[<*docstring*>]
{:def | :iff-def} <*sent-with-arg-vars*>
[:constraints <*sent-with-arg-vars*>]
[:axiom-def <*sent-without-arg-vars*>]
разделы которой аналогичны разделам в конструкции define-class.

6.3. Концепты

Для определения концептов используется ключевое слово define-class. В вершине дерева концептов, как и в Knowledge.NET, находится концепт *Thing*.

6.3.1. Подконцепты

Для определения подконцептов используется отношение SUBCLASS-OF, находящееся в разделе :AXIOM-DEF.
Пример:

Knowledge.NET:

***A** is_subconcept_of **B,C**;*

Вид после конвертирования:

```
(define-class A (?a)
  ...
  :AXIOM-DEF (and (subclass-of A B) (subclass-of A C))
  ...
)
```

6.3.2. Разъединенные концепты

В Ontolingua не существует специального отношения, определяющего, являются ли концепты разъединенными. Поэтому используется отношения INSTANCE-OF в разделе :CONSTRAINTS. INSTANCE-OF – бинарное отношение, указывающее, является ли первый аргумент выражения экземпляром второго аргумента.

Примеры:

Knowledge.NET:

```
Plane
{
        is_subconcept_of Vehicle;
disjoint_with Ship, Submarine;
}
```

Вид после конвертирования:

```
(define-class Plane (?Plane)
        "Generated by KIFConvertor, Knowledge.NET; concept Plane"
        ...
  :constraints (AND (instance-of ?Plane Plane)
                            (not (instance-of ?Plane Ship))
                            (not (instance-of ?Plane Submarine))...)
        ...)
```

Knowledge.NET:

```
disjoint Plane, Submarine, Ship;
```

Вид после конвертирования:

```
(define-class Plane (?Plane)
        "Generated by KIFConvertor, Knowledge.NET; concept Plane"
        ...
  :constraints (AND (instance-of ?Plane Plane)
                            (not (instance-of ?Plane Submarine))
                            (not (instance-of ?Plane Ship))...)
        ...)
```

(define-class **Submarine** *(?Submarine)*
"Generated by KIFConvertor, Knowledge.NET; concept ***Submarine"***
...
:constraints (AND (instance-of ?Submarine ***Submarine****)*
*(not (instance-of ?****Submarine Ship****))...)*
...)

6.3.3. Набор экземпляров

Набор экземпляров задается посредством функции ONE-OF, находящегося в части :AXIOM-DEF.
Например:

Knowledge.NET:
enumerated concept **CONCEPT_IDEN** is_one _of **INDIVIDUAL1, INDIVIDUAL2, …**;

Вид после конвертирования:

```
(define-class CONCEPT_IDEN (?v1 ...)
        "Generated by KIFConvertor, Knowledge.NET; enumerated concept  CONCEPT_IDEN"
        :axiom-def (one-of INDIVIDUAL1 INDIVIDUAL2 ...)))
```

6.3.4. Пересечение концептов

Задается посредством отношения INSTANCE-OF и логической связки AND, находящихся в разделе :CONSTRAINTS
Например:

Knowledge.NET:
concept **A** is_intersection_of **B, C, D…**

Вид после конвертирования:

```
(define-class A (?a)
        "Generated by KIFConvertor, Knowledge.NET; concept A"
    :constraints (and (instance-of ?a B) (instance-of ?a C)
                                (instance-of ?a D)...)))
```

6.3.5. Объединение концептов

Задается посредством отношения INSTANCE-OF и логической связки OR, находящимися в разделе :CONSTRAINTS
Например:

Knowledge.NET:
concept **A** is_union_of **B, C, D…**

Вид после конвертирования:

```
(define-class A (?a)
        "Generated by KIFConvertor, Knowledge.NET; concept A"
    :constraints (or (instance-of ?a B) (instance-of ?a C)
                                (instance-of ?a D)...)))
```

6.3.6. Свойства

Свойства описываются конструкцией define-relation.

6.3.7. Домен и область значения

Для задания доменов используется бинарное отношение domain; последнему в качестве первого аргумента передается имя свойства, в котором домен используется, а в качестве второго – имя структуры (онтологии), которой принадлежит свойство.
Область значения описывается аналогично бинарным отношением range, только в качестве второго аргумента используется имя типа.
Оба отношения находятся в разделе :axiom-def

Например:

Knowledge.NET:

```
datatype property HasMaxSpeed
{
        domain Vehicle, Car;
        range int;
}
```

Вид после конвертирования:

```
(define-relation HasMaxSpeed (?Owner ?MaxSpeed)
        "Generated by KIFConvertor, Knowledge.NET; property HasMaxSpeed"
  :axiom-def ((domain HasMaxSpeed Vehicle)
                (domain HasMaxSpeed Car)
                (range HasMaxSpeed int))
```

6.3.8. Подсвойства

Для представления подсвойств используется отношение subrelation-of, аналогичное отношению subclass-of в классах.
Пример:
Knowledge.NET:

```
object property X is_subproperty_of Y
{
        domain A, B, …;
        range K, …;
}
```

Вид после конвертирования:

```
(define-relation X (?X ?Z)
        "Generated by KIFConvertor, Knowledge.NET; property X"
  :axiom-def ((subrelation-of Y)
                (domain X A)
                (domain X B) ...
                (range X K) ...)
```

6.3.9. Обратные свойства

Задаются бинарным отношением в разделе :AXIOM-DEF, которое имеет синтаксис:

Inverse Relation-Name Inverse-Relation-Name

где Relation-Name – имя свойства, к которому определяется обратное (т.е. свойство, в котором задается это отношение), Inverse-Relation-Name – имя обратного отношения.

Пример:

Knowledge.NET:

```
object property HasColor
{
domain Vehicle;
range    Color;
inverse IsColorOf;
}
```

Вид после конвертирования:

```
(define-relation HasColor (?Owner ?Color)
        "Generated by KIFConvertor, Knowledge.NET; property HasColor "
  :axiom-def ((domain HasColor Vehicle)
                        (range HasColor Color)
                        (inverse HasColor IsColorOf))
```

6.3.10. Функциональные свойства

Задаются бинарным отношением SINGLE-VALUED-SLOT в разделе :AXIOM-DEF. В качестве первого аргумента выступает имя класса, в качестве второго – имя отношения.
Например:
Knowledge.NET:

```
functional object property HasColor
{
domain Vehicle;
range    Color;
}
```

Вид после конвертирования:

```
(define-relation HasColor (?Owner ?Color)
        "Generated by KIFConvertor, Knowledge.NET; property HasColor"
  :axiom-def ((domain HasColor Vehicle)
                        (range HasColor Color)
                        (single-valued-slot Vehicle HasColor))
    )
```

6.3.11. Транзитивные свойства

Задаются унарным отношением transitive в разделе :CONSTRAINTS, которое имеет синтаксис:

transitive-relation Relation-Name

где Relation-Name – имя свойства, в котором задается это отношение.
Пример:
Knowledge.NET:

```
transitive object property IsAncestorOf
{
        domain Human;
        range Human;
        inverse HasAncestor;
}
```

Вид после конвертирования:

```
(define-relation IsAncestorOf (?Ancestor ?Child)
        "Generated by KIFConvertor, Knowledge.NET; property IsAncestorOf"
```

:axioms ((domain **IsAncestorOf Human**)
(range **IsAncestorOf Human**))
:constraints (transitive-relation **IsAncestorOf**))

6.3.12. Симметричные свойства

Задаются унарным отношением symmentric-relation в разделе :CONSTRAINTS, которое имеет синтаксис:

symmentric-relation Relation-Name

где Relation-Name – имя свойства, в котором задается это отношение.
Пример:
Knowledge.NET:

```
symmentric object property HasSibling
{
    domain Human;
    range Human;
}
```

Вид после конвертирования:

```
(define-relation HasSibling (?Human)
    "Generated by KIFConvertor, Knowledge.NET; property HasSibling"
  :axiom-def ((domain HasSibling Human)
              (range HasSibling Human))
   :constraints (symmentric-relation HasSibling))
```

6.3.13. Ограничения

Типы ограничений задаются в конструкции define-class в разделе :CONSTRAINTS

Ограничения по квантору

Для описания существования используется следующее выражение KIF:

(exists ?v1 ?v2 (and (instance-of ?v1 C1) (instance-of ?v2 C2) (Property ?v1 ?v2)))

где C1 – имя концепта, для которого устанавливается ограничение, C2 – наполнитель, Property – имя отношения.
Например:
Knowledge.NET:

```
concept Sibling
{
    is_subconcept_of
    {
        Human;
        HasSibling some_values_from Sibling;
    }
}
```

Вид после конвертирования:

```
(define-class Sibling (?Sibling)
    "Generated by KIFConvertor, Knowledge.NET; concept Sibling"
  ...
   :axiom-def (subclass-of Sibling Human)
```

```
:constraints (exists ?human ?other-human
                    (and (instance-of ?human Sibling)
                         (instance-of ?other-human Sibling)
                         (HasSibling ?human ?other-human))))
```

Для описания всеобщности используется следующее выражение KIF:

(forall (?v1 ?v2) (=> (and (instance-of ?v1 C1) (Property ?v1 ?v2)) (instance-of ?v2 C2)))

где C1, C2 и Property – то же самое, что и для квантора существования.
Например:
Knowledge.NET:

```
concept Father
{
    is_subconcept_of
    {
        Human;
        HasDaughter all_values_from Woman;
    }
}
```

Вид после конвертирования:

```
(define-class Father (?Human)
        "Generated by KIFConvertor, Knowledge.NET; concept Sibling"
  ...
   :axiom-def (subclass-of Father Human)
        :constraints (forall (?v1 ?v2)
             (=> (and (instance-of ?v1 Father)
                      (HasDaughter ?v1 ?v2))
                 (instance-of ?v2 Woman)))
```

Ограничения по мощности

Ограничение «Не более чем» (max_cardinality) преобразуется в вид:
MAXIMUM-VALUE-CARDINALITY <domain-class> <relation> n
Ограничение «Не менее чем» (min_cardinality) преобразуется в вид:
MAXIMUM- VALUE -CARDINALITY <domain-class> <relation> n
Ограничение «Точное количество» (cardinality) преобразуется в вид:
VALUE -CARDINALITY <domain-class> <relation> n,
где domain-class – имя концепта, к которому применяется отношение, relation – имя отношение, n – неотрицательное целое число. Описываются в разделе :CONSTRAINTS.
Например:

Knowledge.NET:

```
concept Human
{
    is_subconcept_of
    {
        Thing;
        IsChildOf cardinality 2;
    }
```

}

Вид после конвертирования:

(define-class **Human** *(?***Human***)*
"Generated by KIFConvertor, Knowledge.NET; concept **Human***"*
:axiom-def (subclass-of **Human Thing**)
:constraints (value-cardinality **IsChildOf** 2*)*

Ограничения по значению

Задаются тернарным KIF-отношением

HAS-VALUE <domain-class> <relation> n

Описываются в разделе :CONSTRAINTS.

Например:

Knowledge.NET:

```
concept ResidentOfRussia
{
    is_subconcept_of
    {
        Human;
        IsLivingAt cardinality 1;
        IsLivingAt has_value Russia;
    }
}
```

Вид после конвертирования:

(define-class **ResidentOfRussia** *(?***ResidentOfRussia***)*
"Generated by KIFConvertor, Knowledge.NET; concept ***Automobile****"*
:axiom-def (AND (subclass-of **ResidentOfRussia Human***))*
:*constraints (and (value-cardinality* **IsLivingAt** 1*)*
(has-value **ResidentOfRussia IsLivingAt Russia***))*

6.3.14. Дополнительные имена

Задаются посредством отношения *ALIAS*, находящегося в части :AXIOM-DEF. Например:

Knowledge.NET:

```
concept Automobile
{
        alias "Car";
        is_subconcept_of Vehicle;
}
```

Вид после конвертирования:

```
(define-class Automobile (?Automobile)
        "Generated by KIFConvertor, Knowledge.NET; concept Automobile"
  :axiom-def (AND (subclass-of Automobile Vehicle)
                  (alias Automobile Car)))
```

6.3.15. Эквивалентные концепты

В Ontolingua не существует специального отношения, определяющего эквивалентность концептов. Поэтому эквивалентность задается с помощью отношения (по аналогии с разъединенными концептами) INSTANCE-OF и логической связки «эквивалентность» в разделе :CONSTRAINTS. Например:

Knowledge.NET:

```
equivalent_concepts A, B, C, …;
```

Вид после конвертирования:

```
(define-class A (?A)
        "Generated by KIFConvertor, Knowledge.NET; concept A"
        ...
  :constraints (forall ?x (<=> (instance-of ?x A)
                               (instance-of ?x B)
                               (instance-of ?x C) ...))
        ...)
```

6.3.16. Одинаковые экземпляры

Задаются через отношение SAME-VALUES.
Пример:
Knowledge.NET:

```
same A, B, C,…;
```

Вид после конвертирования:

```
(and (same-values A B) (same-values A C) (same-values B C) ...)
```

6.3.17. Различные экземпляры

Также задаются через отношение SAME-VALUES и логической операцией NOT.
Пример:
Knowledge.NET:

```
all_different Bill_Smith, Jeff_Kramak, Stephen_Freeman;
```

Вид после конвертирования:

(and (not (same-values **Bill_Smith Jeff_Kramak**))
(not (same-values **Bill_Smith Stephen_Freeman**))
(not (same-values **Jeff_Kramak Stephen_Freeman**)) ...)

6.4. Фреймы

В данном разделе представлены отношения над слотами и примеры.
Атрибуты, демоны и ограничения на слоты при конвертировании игнорируются.

6.4.1. Фрейм-класс

Запись в Knowledge.NET:

frame class **FR_IDEN** {
[own_slots
[is_a **FR_IDEN1**; | **FR_IDEN2**, …]
[**SLOTS**]
]
[instance_slots
[**SLOTS**]
]

[**SLOTS**]:= [**SLOTS-WITHOUT-VALUES**] | [**SLOTS-WITH-DEF-VALUES**]
[**SLOTS-WITHOUT-VALUES**]:= **SlotType1 slot1**
[**SLOTS-WITH-DEF-VALUES**]:=

SlotType1 slot1 = **defaultValue |**

facets
{
type **SlotType1**;
default_value **defaultValue**;
}**slot1**;

Вид после конвертирования:

(define-class **FR_IDEN** (?**FR_IDEN**)
[:CLASS-SLOTS (and (subclass-of **FR_IDEN FR_IDEN1**)
(subclass-of **FR_IDEN FR_IDEN2**) ...)
[**SLOTS-WITHOUT-VALUES**]]
[:INSTANCE-SLOTS ([**SLOTS**])]
[:DEFAULT-SLOTS-VAULES ([**SLOTS-WITH-DEF-VALUES**])]
)

[**SLOTS-WITHOUT-VALUES**]:= (HAS-VALUE-OF-TYPE **slot1 SlotType1**)
(HAS-SLOT-VALUE **slot1 defaultValue**)

[**SLOTS-WITH-DEF-VALUES**]:= (HAS-SLOT-VALUE **slot1 defaultValue**)

Конструкция

facets
{
type **SlotType1**;

default_value **defaultValue**;
}**slot1**;

приводится к виду **SlotType1 slot1** = **defaultValue**

6.4.2. Фрейм-экземпляр

Используется конструкция define-instance. Для задания собственных слотов используется раздел :SLOTS, выражения которого аналогичны выражениям :CLASS-SLOTS и :DEFAULT-SLOTS-VALUES.

6.5. Конвертирование в KIF

При дальнейшем конвертировании в KIF происходит следующее:
1) Определяются все используемые отношения. Их представление в выражениях языка KIF записывается в выходной файл.
2) Конструкции define-relation и define-class трансформируются в конструкцию DEFRELATION языка KIF следующим образом:
:IFF-DEF заменяется на ":="
:DEF заменяется на ":=>"
:CONSTRAINTS заменяется на "=>"
:EQUIVALENT заменяется на "<=>"
:AXIOM-DEF заменяется на :AXIOM
3) Для всех примитивных чисел, используемых в рассматриваемом (предназначенном для конвертирования) представлении на Knowledge. NET, задаются соответствующие отношения (посредством операции defrelation), с использованием двух встроенных отношений KIF над числами:
Integer: (integer **t**) означает, что объект, обозначенный t, – целое число.
Real: (real **t**) означает, что объект, обозначенный t, – вещественное число.
Например, тип byte будет выражен следующим образом:

```
(defrelation byte (?x) :=
  (and (natural ?x) (<= ?x 255)))
```

где natural – отношение вида:

```
(defrelation natural (?x) :=
  (and (integer ?x) (>= ?x 0)))
```

Заключение

На данном этапе работа над системой Knowledge.NET и ее применением находится лишь на начальном этапе. Созданный первый прототип системы будет развиваться и использоваться для экспериментов по решению разнообразных практических задач, требующих сочетания методов традиционного программирования и инженерии знаний.
В дальнейшем планируется интеграция системы Knowledge.NET с системой Aspect.NET, с целью использования возможностей Knowledge.NET для инженерии знаний об аспектах (aspect-oriented knowledge engineering).
Мы надеемся, что наши исследования внесут свой вклад в решение важной задачи преодолени\ разрыва между программной инженерией и инженерией знаний.
Система Knowledge.NET будет также использована для обучения студентов математико-механического факультета СПбГУ.
Коллектив разработчиков будет благодарен за интерес к нашей системе, за любые предложения,

замечания и рекомендации.

Литература

1. Сафонов В.О. Экспертные системы – интеллектуальные помощники специалистов. – СПб.: “Знание”, 1992.
2. J. Zhuk. Integration-ready architecture and design. – Cambridge University Press, 2004.
3. Сафонов В.О. и др. Язык представления знаний Турбо-Эксперт. – Кибернетика, 1991, N 5.
4. Safonov V.O., Cherepanov D.V. Java extension by production knowledge representation constructs and its implementation. – In: Proceedings of International Conference “110th Anniversary of Radio Invention” and Regional IEEE Conference, St. Petersburg, 2005.
5. Новиков А.В. C# Expert – расширение языка C# средствами представления знаний. – Дипломная работа, Санкт-Петербургский государственный университет. Математико-механический факультет. Кафедра информатики. Научный руководитель – проф. В.О. Сафонов, 2004.
6. Сафонов В.О. Платформа Microsoft.NET: принципы, возможности, перспективы. – “Компьютерные инструменты в образовании”, 2004, N 5.
7. Safonov V.O., Grigoryev D.A. Aspect.NET: aspect-oriented programming for Microsoft.NET in practice. – “.NET Developer’s Journal”, 2005, N 7.
8. Web-страницы проекта Aspect.NET: *http://www.msdnaa.net/curriculum/?id=6219.*
9. Web-страницы проекта Knowledge.NET: *http://www.knowledge-net.ru*
10. Спецификация языка KIF: http://logic.stanford.edu/kif/dpans.html
11. OWL Web Ontology Overview. *http://www.w3.org/TR/owl-features/*
12. Ontolingua Reference Manual: http://www-ksl.stanford.edu/htw/dme/thermal-kb-tour/ONTOLINGUA.html