GDAŃSK UNIVERSITY OF TECHNOLOGY
Faculty of Electronics,
Telecommunications and Informatics

Piotr Piotrowski

Knowledge Views and their application in systems engineering

PhD Dissertation

Supervisor:

prof. Krzysztof Goczyła
Faculty of Electronics,
Telecommunications and Informatics
Gdansk University of Technology

Gdańsk, 2010

Simplicity does not precede complexity, but follows it.

"Epigrams in Programming" by Alan J. Perlis

Contents

Chapter 1

Introduction

1.1 The rationale

There is a growing interest in Semantic Web technologies [100], however their adoption in the IT industry is still poor. Semantic Web technologies are still attributed more to the academia rather than to the industry. Nevertheless some big companies like Hewlett-Packard (involved in developing Jena [52]) or Oracle (support for storage and query of semantic content is included in Oracle 11g Database [71]) have already invested in the semantic technologies.

There is a number of reasons why semantic technologies are still more of a novelty. For one thing, introduction of new technologies is always risky—companies usually act according to the old adage:

> If it isn't broken, don't fix it.

This is the reason why old technologies are so prevailing, which leads to the need for mixing new and legacy technologies.

The high risk is also the result of lack of standards. There are some existing Semantic Web standards proposed by W3C, for example RDF [56, 62, 47], SPARQL [91] or OWL [83, 66] and OWL 2 [101], however that is only the beginning. These standards are being developed slowly, for example SPARQL, the query language for RDF, was standardised 4 years after RDF and some submissions are still pending, for example SWRL [48], whose status did not change since year 2004.

Another reason for slow adoption of semantic technologies is the scepticism and lack of knowledge about those technologies on the part of software developers. Experience gained during the author's work in the PIPS (Personalised Information Platform for Life and Heath Services [89]) project funded by the European Commission under the Framework 6 call showed that software developers are not eager to learn semantic technologies, which are not yet mature and therefore prone to changes. Moreover, talking with software engineers on various occasions, including scientific conferences, confirmed the lack of knowledge and scepticism already mentioned. This creates a deadlock: semantic technologies are not used, because they are not mature and they are not mature, because they are not used (there is little feedback).

In both cases the introduction of a layer between semantic tools and traditional components could be a remedy. This layer would hide semantic technologies providing standard interfaces commonly used in systems engineering. First of all that would increase compatibility between different technologies, in some cases those technologies could even become interchangeable. This would allow gradual introduction of new technologies and decrease the risk involved in adopting them. Secondly developers would deal with interfaces they are already familiar with and know how to use. The layer on top of semantic technologies would allow developers to test them without investing much time into learning new APIs. This approach is evolutionary rather than revolutionary, which can positively influence adoption of semantic technologies in the industry.

Apart from the mere mapping of the models behind the technologies there is also a need for model transformation. The experience from the PIPS project showed that the data schema defined by ontologies created by knowledge engineers were often not appropriate for direct use in the business logic of the applications. Some transformations of the data schema were required to cope with differences between the ontology model and the required object or relational model.

There are different opinions about Semantic Web ideals [3]. Some of sceptical opinions state that the vision is too far fetched [63], however some of the technologies developed within the Semantic Web initiative will in time become more common. This is the reason why this work focuses on the introduction of knowledge bases into contemporary systems, which is in the author's opinion achievable in short term. Moreover, the adoption of Semantic Web technologies in information systems may influence their further advancement.

1.2 Goals and the thesis proposition

The main goals of this work are:

1. To define mappings of the ontology model to models commonly used in contemporary information systems, like object model, relational model and RDF;

2. To define a set of transformations for the ontology model which allows adjusting a given model to the needs of the business logic of an application using it.

Achieving these goals contribute to proving the following thesis proposition:

> **The Knowledge Views, which hide the ontology model and provide models commonly used in systems engineering, allow easy usage of knowledge bases as information sources in contemporary information systems.**

1.3 Thesis structure

This thesis is structured into the following chapters:

Chapter 2 contains an overview of the state of the art. It focuses on areas related to component integration, in particular on interoperability and interchangeability as well as the role of standards in component integration. Another area covered in this chapter concerns ontology description languages commonly used in software engineering. Moreover this chapter mentions the role of views in information integration. Finally, a selection of tools that influenced the proposed solution is described.

Chapter 3 constitutes the main contribution of this thesis. In particular it introduces the notion of the Knowledge Views. It defines mappings of ontological model to relational and object models. Moreover it introduces transformation language that allows users to adjust the models to their needs. This chapter discuses compatibility of Knowledge Views with legacy information sources. Lastly it presents the general architecture of the proposed solution.

Chapter 4 describes case studies that present how the Knowledge Views work. Additionally, this chapter presents the results of the experiments performed to evaluate the proposed solution. This chapter focuses on proving the thesis proposition.

Chapter 5 describes details of the implementation of the Knowledge Views that was done to test feasibility of the proposed solution. The implementation was used in the case studies and experiments described in Chapter 4.

Chapter 6 summarises the work and outlines possibilities of future work that would enhance the Knowledge Views idea.

Appendices A and B contain materials used during experiments described in Chapter 4.

Appendix C contains the specification of the xNeeK language that was created as part of this work. This language is used to define Knowledge Views in a declarative manner. xNeeK includes the transformation language described in Chapters 3 and 5.

Appendix D contains Javadoc documentation of Knowledge Views library.

Appendix E contains Javadoc documentation of Object Views library.

Appendix F contains Javadoc documentation of RDF Views library.

1.4 Assumptions, notations and abbreviations

1.4.1 Assumptions

This thesis refers to description logic (DL) and its notation as defined in [4]. If not stated otherwise, the terms ontology and ontology model refer to description logic with addition of Horn rules as in SWRL [48].

1.4.2 Notations and abbreviations

$\top$ universal concept (Top)

$C \sqcap D$ intersection of concepts

$C \sqcup D$ union of concepts

$\exists r.C$ existential quantification

$\forall r.C$ value restriction

$\leq nr.C$ qualified at-most n restriction

$r \circ s$ role chain

$C \sqsubseteq D$ concept subsumption

$C(i)$ concept assertion

$r(s, o)$ role or attribute assertion

DL Description Logic

ERD	Entity-Relationship Diagram
ERM	Entity-Relationship Model
JDBC	Java Database Connectivity
J2EE	Java 2 Platform, Enterprise Edition (before version 5)
JEE	Java Platform, Enterprise Edition (starting from version 5)
JPA	Java Persistence API
MOF	Meta Object Facility
OQL	Object Query Language
OWL	Web Ontology Language
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SWRL	Semantic Web Rule Language
UML	Unified Modelling Language

Chapter 2

State of the art

Many domains have influenced the work on the Knowledge Views and the choice of approaches. In this chapter some of the ideas or technologies that have had the most significant impact are shortly described. This includes introduction to interoperability, ontology description languages, data integration and a few of the existing technologies.

2.1 Interoperability, standards and interchangeability

When integrating different components into systems there are some issues to be considered. Among other they include interoperability of the components, existing standards, but also whether several components that are candidates for performing some function are interchangeable.

2.1.1 Interoperability

Interoperability is one of the most important characteristics that make software components easy to integrate with each other. It means there is no need to adjust the components to enable them to work with each other. IEEE defined interoperability as [49]:

> The ability of two or more systems or components to exchange information and to use the information that has been exchanged.

This definition states that the components need to be able to use the exchanged information. This does not mean that the components need to "understand" the information the same way. For example, let us assume a waiter in a restaurant is asked to describe a dish and he says it is spicy. One person at the table might interpret this information as “this dish may be tasty”, because that person likes spicy food. Another person at the table might interpret this as “this dish is not tasty”, because that person does not like spicy food. These interpretations are opposite, but nevertheless both customers successfully used the information. This observation is important since it is assumed in this work that components exchange information only, the interpretation of the information depends on the receiver.

Creating interoperable components

Interoperability requires taking care of many details on each abstraction level. A low level example is the endianness problem that still haunts programmers when exchanging binary information, for example IPv4 uses big endian convention, while little endian is used in x86 processor architecture. Another example is when writing an application in an assembly language, to call a library function one has to know its application binary interface (ABI). This includes knowing how to pass arguments. There is a number of ways a function can receive arguments. This depends on the processor architecture and the programming language. Happily this has been standardised and automated. In higher level programming languages the details mentioned are taken care of by the compiler. On the level of the high level programming languages what is important is to know the application programming interface (API), that is what functions or classes are available, what arguments and in which order should be passed to functions or methods, etc. In concurrent computing inter-process communication (IPC) is what matters. Once again there are many ways two processes can communicate with each other, for example through a file, shared memory, a pipeline, etc. In distributed computing message passing is used. On top of message passing remote procedure call (RPC) is implemented. Again there are many implementations of RPC depending on the technology and some that are technology independent, like CORBA [74, 75, 76] or SOAP based RPC [69]. As can be seen interoperability cannot be taken for granted.

When creating a new component programmers find themselves on one of the mentioned abstraction levels. Any given level usually hides problems and choices of the levels below, nevertheless any given level also provides its own choices. For example, when writing a component that provides some methods for remote clients, one can choose CORBA to implement RPC, newer SOAP or some other technology. The choice is not that obvious since every technology has its strengths and weaknesses. Developers have to consider what clients will use the component, what technology is used by those clients, whether the component will be used in a real-time system, etc. Sometimes

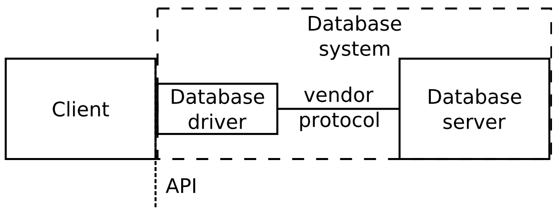


Figure 2.1: Database boundaries

a technically superior form of communication can be replaced by a more popular one, because more popular is almost by definition more interoperable. There are no universal guidelines on choosing a communication method, however as a rule of thumb using standard solutions instead of in-house ones increases interoperability.

Interoperability and database systems

Together with the SPARQL language [91] the SPARQL Protocol [18] has been defined. WSDL [10] is used to describe the protocol, which means the SPARQL compatible RDF stores are Web Services. This definition exactly specifies how a SPARQL query is encoded and sent through the Internet—the specification contains some examples of messages that are sent.

In database systems the situation is different: communication is not standardised, only programming interfaces are. SQL is just a language, therefore it covers only one part of the cited interoperability definition—it is still unknown how to pass the SQL statement. As a remedy, the Open Database Connectivity (ODBC) standard has been devised and in case of Java the Java Database Connectivity (JDBC). These are just APIs, that is a programmer knows what functions or methods to call and knows nothing of what is done behind the scenes. Behind the scenes is a driver that implements the API and translates the calls of the individual methods to messages sent to the database server. The messages are sent using a vendor specific protocol. They can be sent over the Internet or passed to a server on the same machine via some inter-process communication form. There might not even be a separate server process, as is in case of embedded databases. Notice that the driver is actually an integral part of the database system rather than the client (see Fig. 2.1). This is true even though in most cases the client and the database server reside on separate machines and it is the client that embeds the driver. The standardised API is the border between the client and the database. Communication between the client and the server is defined in terms of method calls, while communication between the driver and

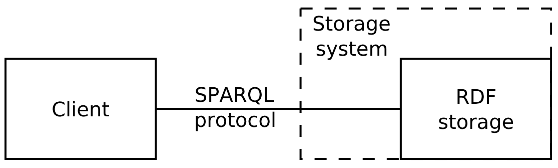


Figure 2.2: RDF storage boundaries

the server is an internal matter. In SPARQL, on the other hand, the border between systems is the network connection (see Fig. 2.2)—SPARQL protocol is optimised for remote calls and is suboptimal in embedded uses.

Making components interoperable

In the nowadays very popular Service Oriented Architecture (SOA) [53] interoperability is a must. It is often the case that a new application is created with SOA in mind from the very beginning. However, there is much legacy software in use that cannot be easily abandoned and that was built before the age of Web Services, SOAP, REST [26], etc. What should be done with such software? How to integrate it well with technologically newer components, that is how to make those components interoperable? Even in the Web Services world there are incompatibilities, for example there are the mentioned SOAP-based as well as RESTful Web Services. This hinders interoperability. In all of these cases into play come such ideas as: wrappers, adaptors, converters, drivers, etc. They are not limited to software engineering, but are also used in other fields, for instance electrical engineering: transformers, voltage converters, rectifiers, socket/plug adaptors. The devices mentioned cope with incompatibilities of electric current like voltage, type (AC/DC) or differences in socket/plug shapes.

To make legacy components interoperable three design patterns are commonly used. These are: adaptors, converters and drivers.

Adaptors Adaptor design pattern [19] is a well known design pattern in software engineering. It is simple, but simplicity is a virtue, because it needs to be implemented quite often. This pattern produces a layer on top of some component or in other words wraps the component, hence its alternate name: wrapper. Thanks to this pattern the wrapped component seems like a different one with different API, but similar function.

Converters Converters work on the data level, that is they change the message passed between components. This complements the adaptor function which focuses mainly on how to pass the message. Often adaptors make no sense without converters, therefore adaptors might embed converters. Converters can make such small changes like byte order, that is endianness, or can translate one sophisticated format to another.

Drivers Drivers provide an abstraction layer on top of some device or database. This layer hides some proprietary interface and provides a standardised one, therefore could be considered a special case of an adaptor. The difference is that the driver can be considered a part of the underlying component (see Fig. 2.1) and is usually created by the component vendor.

2.1.2 Standards and interchangeability

Apart from interoperability there is another feature that is very important in component-based development. It is interchangeability, that is the ability for two components to be exchanged without additional work. The ability to substitute one implementation with another is addressed by some design patterns, for example the bridge pattern [19]. Interchangeability becomes more and more important in modern software engineering, since developers want to have the freedom of choosing and changing the component's vendor with little or no cost.

There is a lot of software with the same or similar functionality. When writing software that uses some third party library one has to choose the vendor. Unfortunately it is often the case that different vendors provide incompatible interfaces. What if we want to support several vendors and to be able to switch between them? This can be useful when for example one implementation is characterised by predictable processing time, while another is faster on average, which may mean the first one is suitable for real-time systems the other is better in other uses. To support two or more implementations of some function with incompatible interfaces our component would need specialised communication code for every supported implementation (see Fig. 2.3). This may be inconvenient and costly to maintain. This is why standards are so important—by unifying interfaces they make components interchangeable (see Fig. 2.4).

If supporting multiple vendors is impossible or impractical, standards can reduce the risk of choosing a particular vendor, because, if the quality of the chosen implementation does not fulfil user's requirements anymore, the vendor can be easily changed. Unfortunately, the process of creating a good standard is lengthy. Usually many different vendors propose their solutions. With the usage of those proprietary solutions their strengths and weaknesses become apparent and in time the best features of different solutions are combined in a single one that eventually becomes the standard. In the meantime

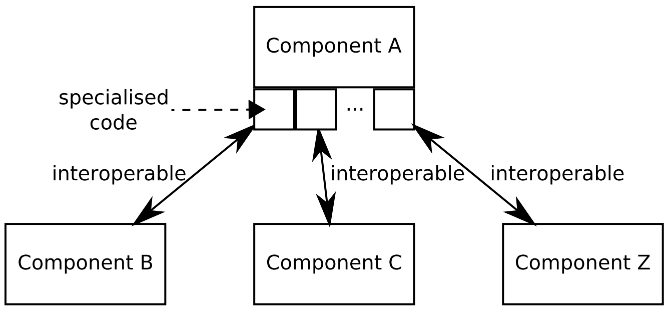


Figure 2.3: Interoperability in the absence of standardised interfaces

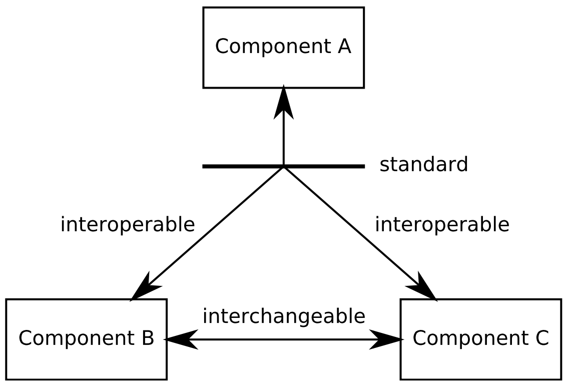


Figure 2.4: Standards and interchangeability

users can choose one of those solutions with the risk that it will cease to be supported. It is helpful if the new solutions are backward compatible with some already existing standard, which ensures a backup solution. An example of this approach is the AMD64 technology [1] which became widely adopted mainly because it was backward compatible rather than superior to its rivals. This backward compatibility allowed the usage of old tools and knowledge, and allowed gradual movement towards exploiting new capabilities.

2.2 Common ontology description languages

There are many definitions of an ontology in computer science. One of the often cited is [12]:

> An ontology is a formal specification of a shared conceptualization.

However for this thesis a more appropriate is an earlier and more general one [45]:

> An ontology is an explicit specification of a conceptualization.

The reason for choosing this definition is the lack of the keyword “shared”. Much emphasis is put on consensus and therefore reuse of ontologies. But is it really critical? Certainly it is good if some work can be reused, which software engineers are well aware of. Nevertheless they also know that sometimes, and not so rarely, there is a need for a very specialised piece of software that is unsuitable for reusing. Not mentioning that creating a good reusable component requires much more effort than creating a specialised one, therefore it is not always economically justified to create components that can be reused. The same goes for hardware, for example central processing units (CPUs) versus graphics processing units (GPUs)—CPUs are more general, whereas GPUs are more specialised, but for the time being they outperform CPUs in certain tasks. Therefore, why not employ the same principal in ontology engineering: reusable ontologies are the ultimate goal, but let us acknowledge the existence and value of the specialised ones. There are top (foundational) ontologies [65] and domain ontologies (for example medical ontologies [72]), but there are also application ontologies that may [50] or may not be based on either one.

In this section the focus is placed on languages that are already commonly used in software engineering for modelling. Those languages can be called ontology description languages. However they are used for creating rather specialised models dedicated to solving some particular problems and also may contain certain details that are technology dependent.

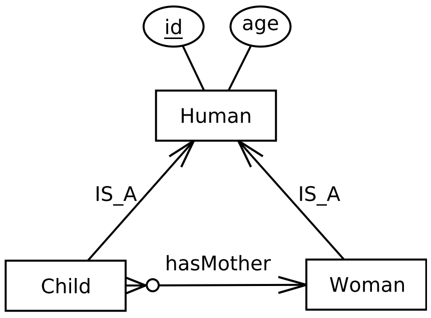


Figure 2.5: A sample entity-relationship diagram

2.2.1 Entity-relationship model

The entity-relationship model (ERM) [17] is one of the most commonly used world description methods in software engineering. It is hard to imagine a software engineer that does not know it. The reason for this is that nowadays relational databases are present in the majority of applications either as stand alone servers or as embedded libraries. Over the years ERM, together with transition rules of ERM to relational model, proved to be very convenient for designing database schemata. One of the ERM's advantages is its simplicity and ability to visualise a model via an entity-relationship diagram (ERD).

Let us look at an example. Fig. 2.5 shows a simple entity-relationship diagram. There are three entity sets: Human, Child and Woman. Human has two attributes: id and age. Both Child and Woman inherit from Human. Moreover there is a relationship between those two entity sets called hasMother. This relationship has some constraints. For one thing every Child has a single mother from the Woman entity set, while a Woman can have any number of children, including none.

After creating an entity-relationship diagram, the next step, while designing a relational database schema, would be to translate the ERM to the relational model. One possible outcome is as follows:

$$\begin{aligned} &Human(\underline{id}, age) \\ &Woman(\underline{id}\ \textbf{REF}\ \ Human) \\ &Child(\underline{id}\ \textbf{REF}\ \ Human, hasMother\ \textbf{REF}\ \ Woman) \\ \\ &id \in integer \\ &age \in integer \end{aligned} \tag{2.1}$$

The translation can be done by strictly obeying some simple rules, however there is some information added that was not present on the ERD. The added information are the domains of the attributes.

Listing 2.1: A sample database schema

```
CREATE TABLE Human
  (id INTEGER PRIMARY KEY, age INTEGER);
CREATE TABLE Woman
  (id INTEGER PRIMARY KEY REFERENCES Human);
CREATE TABLE Child
  (id INTEGER PRIMARY KEY REFERENCES Human,
    hasMother INTEGER NOT NULL REFERENCES Woman);
```

The last step is to produce the source code in SQL that creates a database whose schema conforms to the original ERD. The code is presented in Listing 2.1. Each of the three notations (Fig. 2.5, (2.1) and Listing 2.1) represent the same model or in other words describe the same part of reality, but their uses are different, for example ERD is human readable, while SQL is machine readable.

2.2.2 Unified Modelling Language

Unified Modelling Language (UML) [77, 78] is widely used in object-oriented software design. There is some criticism of UML misuse [7, 8], as well as UML itself (even in the form of a satire [67]). Nevertheless, nowadays virtually any software engineer knows at least the basics of UML. Even though UML covers both static as well as dynamic aspects of the object model, this section focuses on the static aspect, more precisely on the class diagram. The purpose of such narrowing is to cover only the part corresponding to the ERM or description logic, which lacks behaviour.

The model from Fig. 2.5 can also be presented as an UML class diagram (see Fig. 2.6) and source code (see Listing 2.2). Notice the difference between the ERD and UML diagram: there is no id attribute in the UML version. The attribute being the primary key in relational model is no longer needed in the object model, because every object has identity—even two objects of the same class with the same content can be distinguished. However, the id attribute can be useful to distinguish objects in case of distributed systems, when the object is passed through value between various systems. This is a technical issue rather than the shortcoming of the ideal object model. Even though there are some differences between relational and object models the two can be mapped to each other. The mapping is not always one-to-one, but can be useful in a wide range of use cases.

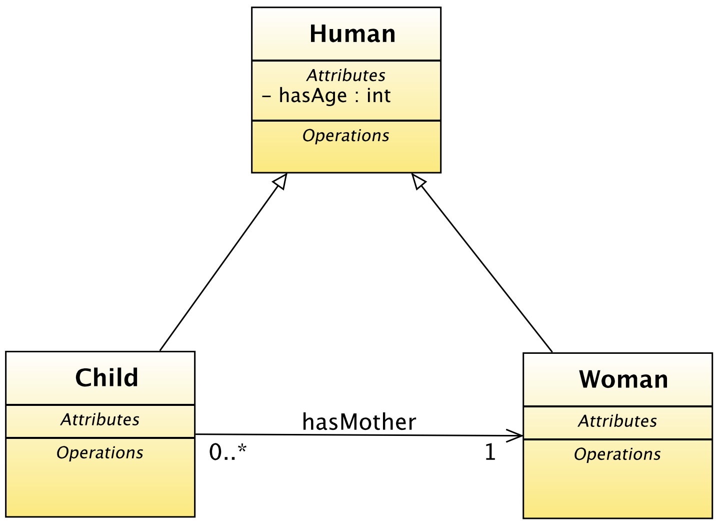


Figure 2.6: A sample class diagram

Listing 2.2: A sample Java implementation

```
public class Human {
    private int hasAge;
}

public class Child extends Human {
    private Woman hasMother;
}

public class Woman extends Human {
}
```

Listing 2.3: Sample OWL 2 ontology

```
Prefix(:=<http://a.b.c/sampleOntology#>)
Ontology(<http://a.b.c/sampleOntology>
  Annotation(rdfs:label "Sample OWL 2 ontology")

  SubClassOf(:Human owl:Thing)
  FunctionalDataProperty(:hasAge)
  DataPropertyDomain(:hasAge a:Human)
  DataPropertyRange(:hasAge xsd:integer)

  SubClassOf(:Woman :Human)

  SubClassOf(:Child :Human)
  FunctionalObjectProperty(:hasMother)
  ObjectPropertyDomain(:hasMother :Child)
  ObjectPropertyRange(:hasMother :Woman)
  SubClassOf(:Child ObjectMinCardinality(1 :hasMother))
)
```

2.2.3 Web Ontology Language

Web Ontology Language (OWL) [66, 101] is not as widely known by software engineers, but it has been adopted by the Semantic Web community and standardised by W3C. As opposed to the ERM and UML this language is said to be “semantic”.

The same model as in the previous two cases can also be expressed in OWL. Listing 2.3 shows the model in OWL 2 functional style syntax [70]. There is a variety of syntax for OWL, therefore this example can be rewritten in several ways. The syntax chosen for this example is among the more human readable ones. Tools usually use the RDF/XML based syntax, which is practically unreadable by a human even for small ontologies. OWL contains a lot of syntactic sugar constructs. The same ontology as a description logic

(DL) axiom set looks as follows:

$$\begin{aligned}
\exists hasAge.\top &\sqsubseteq Human\\
\top &\sqsubseteq \forall hasAge.integer\\
\top &\sqsubseteq \leq 1 hasAge.\top\\
Woman &\sqsubseteq Human\\
Child &\sqsubseteq Human\\
\exists hasMother.\top &\sqsubseteq Child\\
\top &\sqsubseteq \forall hasMother.Woman\\
\top &\sqsubseteq \leq 1 hasMother.\top\\
Child &\sqsubseteq \exists hasMother.\top
\end{aligned} \tag{2.2}$$

Notice that in the OWL model the id attribute has been omitted as well. OWL adopted the convention of using IRIs [23] as identifiers. However two different IRIs can still denote the same individual.

2.3 Views and information integration

The Knowledge Views are strongly influenced by the concept of views in general. Because of some properties of views, the Knowledge Views gained the ability to integrate information, therefore a few words on the information integration issue are also in place.

2.3.1 Views

The concept of views is well known to any software engineer familiar with contemporary databases. The main idea is to define a virtual object (a table in case of relational databases) in terms of other existing or virtual objects. In relational databases the definition of the new virtual table is encoded as a query. In object-oriented programming such patterns as adaptor or facade [19] can also be considered views, because those patterns define new interfaces (APIs) in terms of existing ones. New virtual objects are created because existing ones are for some reasons inappropriate. However, views can also be used for hiding existing objects. For example in databases hiding some information can be required for security reasons. In programming hiding can increase loose coupling and thus increase interchangeability of components. This is a very desirable feature, because one can "tinker" with what is behind a view without touching the rest of the system.

There are two kinds of views: materialised and non-materialised. The difference is that in materialised views mappings between new and old objects (a query in case of databases) are evaluated once or from time to time, but not during every request, while in non-materialised views the evaluation is performed on every access. These two approaches to views have different properties. In general materialised views tend to become stale quickly, that is they do not contain up-to-date data, while non-materialised views tend to be less efficient. This trade-off has to be taken into account when choosing between those two approaches. However, from the user's point of view those views should not be distinguishable—if they are they no longer are interchangeable.

2.3.2 Information integration

The concept of views has also its application in information integration where one schema is defined in terms of another. Two common approaches are: Local-as-View (LaV) [60, 99] and Global-as-View (GaV) [16, 99].

In LaV approach the information sources to be integrated are treated as if they were views on the global information schema. This requires a complex algorithm [59] that rewrites the queries to the global schema in terms of the local schemata/views. The advantage of this approach is the ease of adding new sources, because the only thing to do is to define the mapping for the new source.

The GaV approach differs from LaV in that the global schema is treated as the view, which is defined in terms of local schemata. This seems to be more intuitive and has the advantage of simple query rewriting, since it only requires execution of mappings. A disadvantage attributed to GaV is that adding new information sources may cause the need of changing mediators that are responsible for communicating with the sources and combining information from them.

Let us consider the following example. There is a global (target) schema t containing one predicate: $\mathit{t{:}uncle}(x, y)$. There are also two information sources *s1* and *s2*. The first one contains only one predicate: $\mathit{s1{:}uncle}(x, y)$. The other contains two predicates: $\mathit{s2{:}uncle}(x, y)$ and $\mathit{s2{:}parent}(x, y)$. In LaV approach the mappings would look as follows:

$$\begin{aligned} \mathit{s1{:}uncle}(x, y) &\leftarrow \mathit{t{:}uncle}(x, y) \\ \mathit{s2{:}uncle}(x, y) &\leftarrow \mathit{t{:}uncle}(x, y) \end{aligned} \tag{2.3}$$

In GaV the mappings are:

$$\begin{aligned} \mathit{t{:}uncle}(x, y) &\leftarrow \mathit{s1{:}uncle}(x, y) \\ \mathit{t{:}uncle}(x, y) &\leftarrow \mathit{s2{:}uncle}(x, y) \end{aligned} \tag{2.4}$$

So far both approaches look very similar. Now let us add a new source *s3* that contains a single predicate: $\mathit{s3{:}brother}(x, y)$. In GaV only one additional rule is needed to take advantage of the new information:

$$t{:}uncle(x,z) \leftarrow s2{:}parent(y,z) \wedge s3{:}brother(x,y) \qquad (2.5)$$

This rule states that x is an *t:uncle* of z if y is a *s2:parent* of z and x is a *s3:brother* of y. How about LaV? We know that every *t:uncle* is someone's *s3:brother*, but it is not enough, because there is no way of knowing whose *s3:brother* is the *t:uncle*. It turns out that to take advantage of the third source in LaV the target schema needs to be changed and changing the target schema can have further consequences.

The above example proves that the amount of work needed to add a new source does not necessarily depend on the direction of the mappings alone, but other factors have to be considered as well. For example the design of mediators can be a factor here.

Apart from the two approaches presented, there are also other solutions, for example GLaV [28] or BGLaV [102]. Moreover, there is some effort to employ ontologies in the integration process, for example [64].

2.4 Existing technologies

There is a number of technologies that employ ideas similar to those proposed in this thesis. Only the more prominent of the technologies are shortly described below to outline their advantages that influenced the solution proposed in this thesis, as well as disadvantages which should be overcome.

2.4.1 Java Persistence API

Java Persistence API (JPA) [20] is a standard that provides a way of querying a relational database as if it were an object database. The main advantage of such a solution is that querying from an object-oriented programming language is easier than by using JDBC [2] directly, which is a low level access method. There are several factors that cause this. For one thing a wrapper like JPA hides the impedance mismatch between relational databases and object-oriented programming languages. However, a more important cause for this solution's success is that using JPA requires significantly less code than using JDBC. This means that there is less code to write, less code to maintain and less place for bugs. This even outweighs the penalty caused by the performance hit that an additional layer produces. JPA is dedicated for relational databases, but the idea can be easily expanded to other data sources.

2.4.2 LINQ

LINQ [68] is the Microsoft's response to JPA. As it is a newer technology, it provides some new features compared to JPA. The most significant is the ability to query various data sources. They include relational databases, XML files, object collections, etc. The uniform way of accessing various types of storage causes those types to be interchangeable and therefore allows more flexibility in software development.

Another feature that LINQ introduces is the embedding of the query language into the programming language. This is supposed to make querying even easier, but it requires the solution's author to have full control over the programming language and can cause the language to become overcomplicated. If too much syntactic sugar is introduced, the programmer has to invest more effort into learning fully the language used.

2.4.3 JXPath

JXPath [54] is a library that allows querying object trees complying to Java Beans guidelines using XPath [9] query language. It is an example of using a query language that was devised for querying one model, here XML, to query another model, here the object model. This solution is based on the fact that object trees can be easily mapped to XML. It is not a complete solution for querying object storages, but can be useful when using a fully-fledged database is an excess, while handwritten code traversing the object trees would overcomplicate things.

2.4.4 ActiveRDF

ActiveRDF [79, 80] is a library that provides object-oriented API to RDF storages by mapping RDFS [13] to the object model. It is written in Ruby programming language. The language choice is not incidental, because this library uses some features, for example dynamic typing, that are not common to all object-oriented programming languages. This makes the solution not universal, for example it cannot be applied in commonly used statically typed languages like Java or C++. In C#, another popular programming language, the dynamic typing has only recently started to be introduced. ActiveRDF also uses some other language features that are not common. The author of ActiveRDF discards a group of commonly used languages as not suitable for dealing with RDF. ActiveRDF addresses the problem of convenient access to RDF storages from object-oriented programming languages, however the solution is applicable only to a small group of languages omitting statically typed languages that are still more widely used.

2.4.5 Jastor

Jastor [51] is a library that generates Java classes that implement mapping of OWL to Java based on the ideas from [55]. The mapping proposed in [55] is to some extent similar to the ontology-object mapping in this thesis (see Section 3.2.2). For example both mapping propositions map OWL classes to interfaces with accessors corresponding to OWL properties (roles and attributes in DL nomenclature). There are, however, some crucial differences between the two mapping propositions. The most notable difference is that in [55] some OWL property restrictions are encoded as Java code that validates values of properties whenever they change. There are several reasons why no such thing is done in the solution proposed in this thesis. First of all the validating code duplicates logic present in the knowledge base underneath. Moreover, in some complex cases validation cannot be performed on every property change, since consistency can depend on combined values of several properties—several property changes might require treating them as an atomic operation. These are just some of the reasons why proposition from [55] was not adopted in this thesis. The Jastor library itself has yet another significant shortcoming—the classes generated from an OWL ontology are tightly coupled with the Jena library [52] and cannot be used with any other knowledge base.

2.4.6 Summary

From the technologies presented above both ActiveRDF and Jastor have similar purpose to the Knowledge Views, that is to provide a convenient way to access knowledge bases. However both have significant shortcomings that limit their application: ActiveRDF is tied to Ruby programming language and cannot be applied in Java or C++, Jastor is tied to Jena as the underlying knowledge base. Moreover, they focus only on providing an object-oriented interface to knowledge bases. The Knowledge Views on the other hand strive to be more general and more powerful. The Knowledge Views do not rely on specific language features and they provide more than just an object-oriented interface. Moreover, the Knowledge Views offer a systems engineer powerful and flexible transformation and modularisation capabilities. Some of ideas gained from JPA, LINQ and JXPath helped the author in achieving these goals.

Chapter 3

Knowledge Views

The Knowledge Views role is to help in the integration of Semantic Web technologies with traditional technologies used in contemporary information systems. The two most important aspects of the Knowledge Views are model mapping and model transformation. However, what is also important is the architecture that defines how individual elements should work together to achieve the goals of this thesis.

3.1 The Knowledge View concept

A Knowledge View is an abstraction layer that hides a knowledge source and provides the information from the source in a different model. The knowledge source provides knowledge as a description logic ontology. The user can choose which information is relevant and should be provided by the view. Moreover, the user may choose to change the schema so that the result better suits his needs. This is similar to database views where the view's definition is a query that may select only some information as well as transform it. The Knowledge View may provide as a result:

- DL View—a description logic ontology,
- RDF View—an RDF model with RDF triples,
- Object View—an object model with objects,
- Data View—a relational database schema with data.

The concept of views is flexible enough to allow more kinds of models.

A Knowledge View can play a variety of roles. First of all it can be used to simplify access to a knowledge base. Knowledge bases are not widely used and software engineers are seldom familiar with them. Furthermore, there is still no single standardised interface provided by various knowledge bases. In this situation an abstraction layer hiding all these issues and providing a well known model like an object model can greatly ease the usage of knowledge bases.

The Knowledge Views can also ease the process of integrating data-oriented and knowledge-oriented components. For example, if some application uses a database as a data source, changing the data source to a knowledge base would only require defining a Data View and no modification of the application.

The Knowledge Views provide more functionality like data integration, however, this is rather a side effect of the chosen approach and will be discussed later.

While developing the Knowledge Views, some assumptions were made. The first assumption is that each model is created to solve some particular problem. Since a view is a model, each view is created for a specific purpose. If, for example, an application uses a single knowledge source, but each component of that application uses it differently, there is a good reason to create several views, even if information provided by those views overlaps. This approach promotes modularity.

By providing different models, Knowledge Views define mappings between the ontological model and the resulting models. In general the mappings are not one-to-one, i.e. some information can be lost. This is not considered to be a disadvantage. Each model: relational, object or ontological has been devised with some particular purpose in mind. As a result the models have different expressivity. These three models have some common features like the concept of classes of items, relationships between them and attributes. However, there are also some important differences. For example, the object model, apart from the structure, also has behaviour. The relational and ontological models are both static. In the proposed solution the assumption is that models are not interchangeable, i.e. each model has its purpose and information is passed between different models only to allow them to fulfil their purposes. For instance, in an application each of the models could be used in a different layer (see Fig. 3.1). The relational model is used in databases, therefore, it is present in the database layer that stores information and ensures their efficient retrieval. The knowledge layer provides reasoning capabilities, i.e. can produce some facts that are not explicitly stored in the database. The business logic layer written in an object-oriented programming language can perform some actions based on the facts provided by the knowledge layer. The presentation layer visualises the results and might not use any of the previous models. Each layer receives some information from its neighbours,

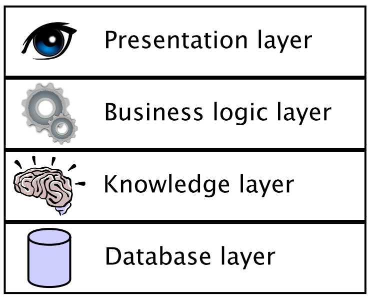


Figure 3.1: Sample layers in an application

but does not have to have full knowledge about the models in the other layers, which is consistent with the "information hiding" modularisation technique postulated in [82]. In the proposed solution it is assumed that in run time only assertions are passed between models, while axioms are only used internally by the knowledge layer for the sole purpose of reasoning. Since axioms define the data schema, they are used indirectly during mapping. As a result of this assumption, the expressivity of the knowledge source, on top of which a Knowledge View is created, does not influence the view.

This approach can be compared to how people communicate: they pass information, but how the same information is interpreted and what action it induces depends on the person receiving the information. For example, the information that there is fire causes most people to run away from the fire, but there is a small group of people that head towards the fire and they are called firemen. In software engineering, especially in multi-tier systems, it is common that communication between tiers or components is performed by passing structures or objects with no behaviour. Each tier keeps functions working on the information passed in such an object separate from it. Putting all the behaviour from all the tiers using the same data object into that object would only cause problems with code maintenance, providing no or little benefit. This may seem contradictory to the object-oriented programing, but it can also be seen as taking advantage of the best of both worlds.

In the Knowledge Views one-to-one mapping is not considered to be crucial. In model mapping in general, it is possible not to loose any information while mapping one model to another one. However, this requires the target model to have expressivity at least equal to the source model's expressivity. Creating a super-model that is able to express all information from every other model would be impractical, because such a model would be too complex to use or even to comprehend. Modelling means neglecting information that is considered insignificant. If some important information is hard to represent in a model, this means the model has been chosen inappropriately.

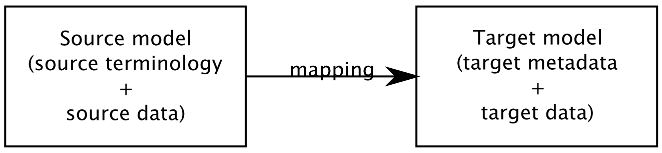


Figure 3.2: Mapping process in general

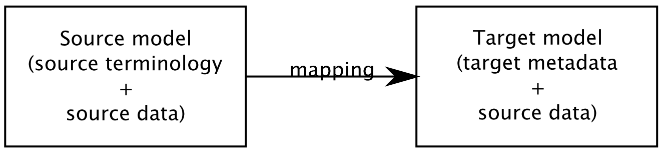


Figure 3.3: Narrowed mapping process

This approach conforms to the following statement [27]:

> There is no right or wrong model, merely one that is more useful for the job at hand.

3.2 Model mapping

Before discussing about model mapping a short clarification of what is meant under this term is in place. Mapping is a function, therefore model mapping would be some kind of function that changes one model into another. However, what exactly is changed? To answer that, different abstraction levels need to be considered: information level, model level and metamodel level. In Meta Object Facility (MOF) [73] these levels are called: M0 layer, M1 layer and M2 layer respectively. If we want to change an ontology model to an object model, changes on the metamodel (M2) level are necessary, since in an ontology model there are concepts, roles and attributes, while in an object model there are classes and fields. For example a *Cat* concept would change into a *Cat* class. The lower level (model or M1 level) can also bear other names: terminology (in ontology model), schema or metadata. On this level the *Cat* concept or class could be changed into the *Felis catus* concept or class. If one model expresses temperature in °C and another in °F a change on the information level (M0) is also necessary.

According to what was said about the three abstraction levels, a general notion of model mapping would contain changes on all three levels (see Fig. 3.2). However, in the proposed solution, the term model mapping has been narrowed to encompass only changes on the metamodel level (M2) (see Fig. 3.3).

Definition 1 (Model mapping) *A model mapping is a function that associates entities of the source metamodel with entities of the target metamodel. Not every entity of the source model needs to have its image in the target model.*

Mappings do not have to preserve all the semantics of the source model. The target model only takes such information from the source model which is considered significant. This means that semantics can be lost in the process. However, some semantics can be added as well, since, depending on the exact model, some arbitrary assumptions can be made.

The reason for adopting such a solution is simplicity and user friendliness. Allowing the mappings to lose some semantic information can make the mappings simpler, therefore easier to understand and more intuitive. Having simple mappings the user can more easily determine the resulting model knowing the source model. Some additional arbitrary assumptions can be made in the mappings, to make the mappings more intuitive.

This solution does not seem so entirely unreasonable for anyone who knows at least two human languages and has seen a movie with subtitles where the actors spoke one language and subtitles where in another. The translated dialogues can differ from the original ones and therefore convey a different message. However, translating everything literally would produce awkward sentences that sometimes would make no sense.

3.2.1 Ontology-relational mapping

The ontology-relational mapping is a set of rules describing how to represent a given ontology model as an entity-relationship model:

1. Each concept is represented by a single entity set.

2. Each entity set representing a concept has an attribute being the primary key that contains the individual's identity.

3. Each functional role is represented by a many-to-one relationship.

4. Each non-functional role is represented by a many-to-many relationship.

5. Each functional attribute is represented by an attribute in any entity set representing a concept which is not disjoint with the attribute's domain.

6. Each non-functional attribute is represented by a single entity set with two attributes: the identity of the corresponding entity and the value of the attribute. It is connected to the appropriate entity set with a many-to-many relationship.

7. The subsumes relationship is represented by a IS_A relationship.

The entity-relationship model can be mapped to a relational model in accordance with the rules commonly used for designing relational databases [29]. The rules allow some freedom of choice, therefore clarification of the chosen approach is necessary:

1. Many-to-one relationships are represented by a reference on the many side of the relationship.

2. IS_A relationships are represented by a pair of relations: one representing the more general entity set and the other, having a reference to the first one, containing only those attributes of the more specific entity set that are not present in the first relation.

It is important to note that this mapping is not one-to-one. It cannot be reversed, because some information is lost, for example information on the disjointness of the concepts from the ontology.

Let us try to create a relational database schema from the ontology (3.1).

$$\begin{aligned} Woman &\sqsubseteq Human \\ Child &\sqsubseteq Human \\ \exists hasMother.\top &\sqsubseteq Child \\ \top &\sqsubseteq \forall hasMother.Woman \\ \exists hasAge.\top &\sqsubseteq Human \\ \top &\sqsubseteq \forall hasAge.integer \end{aligned} \tag{3.1}$$

Fig. 3.4 shows the entity-relationship diagram obtained from the source ontology according to the specified mapping rules. Listing 3.1 shows the SQL statements creating the resulting database.

When looking at the resulting model (see Fig. 3.4) one could not help but notice that a child can have many mothers. The same holds for hasAge. This is due to the lack of a proper statement in the source ontology. This does not necessarily mean the source ontology is erroneous, it may as well mean that such a statement was not necessary in the reasoning process and therefore was omitted deliberately. It is not always possible or even reasonable to modify the source model, so there is a need for some means to transform the model so the results would seem more natural. In this case the resulting

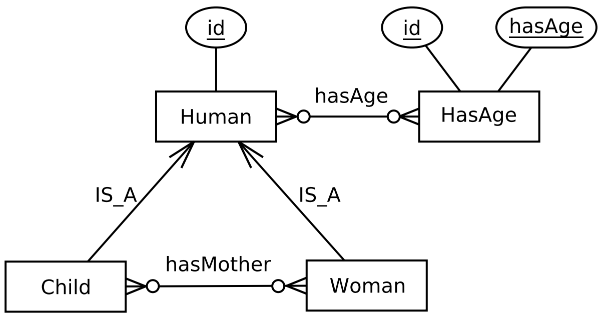


Figure 3.4: Ontology (3.1) mapped to ERD

Listing 3.1: Ontology (3.1) mapped to SQL

```
CREATE TABLE Human
   (id VARCHAR(255) PRIMARY KEY);
CREATE TABLE Woman
   (id VARCHAR(255) PRIMARY KEY REFERENCES Human);
CREATE TABLE Child
   (id VARCHAR(255) PRIMARY KEY REFERENCES Human);
CREATE TABLE HasAge
   (id VARCHAR(255) REFERENCES Human, hasAge INTEGER,
     PRIMARY KEY (id, hasAge));
CREATE TABLE HasMother
   (subject VARCHAR(255) REFERENCES Child,
     object VARCHAR(255) REFERENCES Woman,
     PRIMARY KEY (subject, object));
```

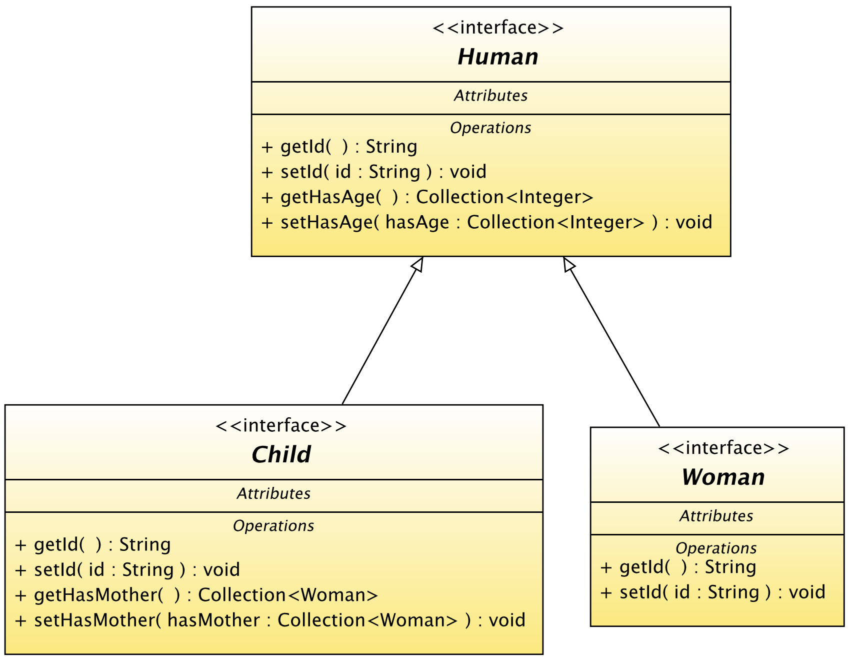
schema could be simplified by removing HasAge and HasMother tables and introducing appropriate columns in Human and Child tables. This issue is addressed in Section 3.3.

3.2.2 Ontology-object mapping

Having the similarities between models as a base, ontology-object mapping rules can be defined:

1. Each concept is represented by an interface.
2. Each interface representing a concept has a pair of accessors to a value that plays the role of the object identity (the identity handling issue is described more thoroughly below).
3. Each functional role is represented by a pair of accessors. The accessors are present in all the interfaces that belong to the role's domain. The type of the accessors is an interface representing the range of the role.
4. Each non-functional role is represented just like a functional one, except that the type of the accessors is a collection.
5. Each functional attribute is represented like a functional role, except that the accessors' type corresponds to the attribute's type.
6. Each non-functional attribute is represented like a functional one except that the type of the accessors is a collection.
7. The subsumes relationship is represented by inheritance.

In the object model objects have identities. Similarly in the ontology model individuals have identities. Why not take advantage of this similarity and map each individual to a single object? There are several problems with such an approach. First of all, in the lack of unique name assumption (UNA) [4], two different identifiers in an ontology can refer to the same individual. At a certain point in time there might not be enough information to decide whether two identifiers point to the same individual or not. In the object model, on the other hand, we can always decide whether two references point to the same object or not. Moreover in RDF and OWL identifiers are global, while in most object-oriented programming languages object identities are local. Existing object-oriented programming languages are only an approximation of the ideal object model and restrictions of programming languages need to be considered. The problem of object identities being local



<<interface>>
Human
Attributes
Operations
+ getId( ) : String
+ setId(id : String) : void
+ getHasAge( ) : Collection<Integer>
+ setHasAge(hasAge : Collection<Integer>) : void
<<interface>>
Child
Attributes
Operations
+ getId( ) : String
+ setId(id : String) : void
+ getHasMother( ) : Collection<Woman>
+ setHasMother(hasMother : Collection<Woman>) : void
<<interface>>
Woman
Attributes
Operations
+ getId( ) : String
+ setId(id : String) : void

Figure 3.5: Ontology (3.1) mapped to UML

is especially visible in distributed environments. The usual practice to cope with this problem is just to add an additional field to the object and compare objects by value and not by reference. This is why the ontology-object mapping requires a pair of accessors for object identity.

Even though the object model supports multiple inheritance, it has been chosen to use interfaces rather than classes, since some languages, like Java, support multiple inheritance only through interfaces. Choosing interfaces leads to another restriction: interfaces cannot have attributes, therefore accessors are used in their place. These assumptions, however, are not important for the mapping idea and were made only to handle some technical peculiarities in some object-oriented programming languages.

Let us try to create a class diagram from the ontology (3.1) using the defined mapping rules (see Fig. 3.5). Once it is done, source code of the interfaces can be generated (see Listing 3.2).

Similarly as it was with ontology-relational mapping, the problem with the source ontology not containing some information relevant to the target model manifested itself in the Child having a collection of Mothers.

Listing 3.2: Ontology (3.1) mapped to Java

```
public interface Human {
    String getId();
    void setId(String id);
    java.util.Collection<Integer> getHasAge();
    void setHasAge(
            java.util.Collection<Integer> hasAge);
}

public interface Woman extends Human {
    String getId();
    void setId(String id);
}

public interface Child extends Human {
    String getId();
    void setId(String id);
    java.util.Collection<Woman> getHasMother();
    void setHasMother(
            java.util.Collection<Woman> hasMother);
}
```

3.2.3 Query language mapping

In addition to mapping of the models themselves, there is a need to map the query languages of individual models. For the relational model there is SQL, for the object model OQL and for RDF SPARQL. A common approach to query language mapping is to translate query in one language to a corresponding query in another language. Examples of work using this approach are [103] and [24]. This method cannot be applied in all circumstances. For example, when query languages have different expressiveness, a single query in one language might correspond to a series of queries in another language or exact mapping is impossible altogether.

In this work queries in one language are not translated directly to queries in another language. Instead, the queries are interpreted. During the interpretation atomic operations are performed on the underlying source. Those atomic operations evaluate unary and binary predicates, as well as some built-in predicates for operations like arithmetic comparison. For convenience the predicates can be grouped into Horn rules. Often a single query cannot be expresses as a single rule, therefore the results of individual rules are combined using set-theoretic union and intersection. The Knowledge Views can be placed on top of various knowledge bases, therefore the atomic operations are translated to the query language of a particular knowledge base. This is done by an adaptor for the particular knowledge base. It is much easier to implement a few atomic operations in every new adaptor than to implement a complete translation library in each adaptor. Thus, the chosen solution facilitates the addition of new adaptors for new knowledge bases or even other information sources. Another advantage of this solution is that it does not require the knowledge base or other information source to support any particular query language and does not place any requirements on the query language supported by the source. As a consequence this solution does not take advantage of advanced query optimisers present in sources like relational databases. In the chosen approach generality is placed before performance.

3.3 Model transformation

As already pointed out, when mapping one model to another, it might be the case that source model does not contain some information that is significant for the target model. Moreover there might be some more differences between the source model and the one needed by the user. Let us consider the following example. A car dealer has information about the cars he sells that conform to the following terminology snippet:

$$\begin{aligned} Car &\sqsubseteq \exists hasEngine.Engine \\ Engine &\sqsubseteq \exists hasPower.int \end{aligned} \tag{3.2}$$

He wants to have a web page where the information about the cars is shortened to the following:

$$Car \sqsubseteq \exists hasHorsePower.int \tag{3.3}$$

Assuming the web page is created in some object-oriented technology like Java, ontology-object mappings might be useful. However, mappings as defined by the Definition 1 do not suffice, since they would only map a *Car* to a *Car* and an *Engine* to an *Engine* in another model. The mapping definition does not allow changing the schema. In this example, on the other hand, the user wishes to change the schema by removing the *Engine* concept and introducing *hasHorsePower* attribute that replaces $hasEngine \circ hasPower$ chain. Moreover, the units of engine power might change from kW to hp and the value might need recalculation. This is why there is a need for model transformation in addition to model mapping.

Definition 2 (Model transformation) *A model transformation defines entities of the target model (metadata) in terms of entities of the source model (metadata).*

What is important is that model transformation operates on a lower level than model mapping. In Meta Object Facility [73] terminology model transformation operates on the M1 level, whereas model mapping on the M2 level. Changes on the model level may require changing data, for instance if source model expresses power in kW, and target model requires hp.

An example of model transformation can be an SQL query, because it defines a new relation in terms of existing relations. Similarly OQL, apart from returning existing objects, can also define new objects in terms of existing ones. This is why these query languages can be treated as transformation languages. SPARQL, on the other hand, has several types of queries [91]. A SELECT SPARQL query produces a relation from an RDF graph, which does not fit the model transformation definition. However, a CONSTRUCT SPARQL query produces an RDF graph from an RDF graph, therefore it can be used as a transformation language.

The model mapping and the model transformation operate on different levels and only by combining both of them the desired effect can be achieved. However, there are two ways to combine these two processes:

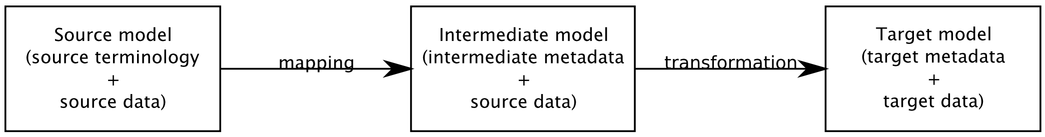


Figure 3.6: “Mapping first” combination of mapping and transformation

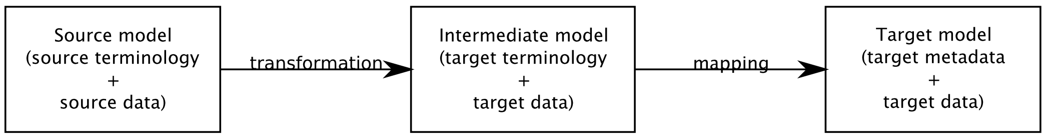


Figure 3.7: “Transformation first” combination of mapping and transformation

1. Mapping first and then transformation (see Fig. 3.6).
2. Transformation first then mapping (see Fig. 3.7).

For illustration let us assume that an ontology with terminology (3.2) is our source model, however, developers require the following relational model:

$$\begin{aligned} &Car(\underline{id}, hasHorsePower) \\ \\ id &\in integer \\ hasHorsePower &\in integer \end{aligned} \tag{3.4}$$

This is the target model. In the “mapping first” approach (see Fig. 3.6) the first step is mapping, therefore the intermediate model is the relational model that conforms to the source terminology (3.2):

$$\begin{aligned} &Car(\underline{id}, hasEngine\ \mathbf{REF}\ \ Engine) \\ &Engine(\underline{id}, hasPower) \\ \\ id &\in integer \\ hasPower &\in integer \end{aligned} \tag{3.5}$$

The second step is transforming (3.5) into (3.4). This can be done with a single SQL query (see Listing 3.3). As seen in the above example the “mapping first” approach has the advantage of reusing existing query languages as transformation languages. Nevertheless, this approach also has some disadvantages.

The most important disadvantage is that often transformation using existing transformation languages is unidirectional. It means that in general it is not possible to insert data through the transformation back to the source

Listing 3.3: SQL query transforming (3.5) into (3.4)

```
SELECT Car.id, CAST(haspower/0.73549875 AS INTEGER)
  FROM Car, Engine WHERE hasEngine = Engine.id
```

model. For example, database views might in some cases allow the insert statement, however in general database views are read-only. SPARQL on the other hand does not support inserting information into the repository at all.

The second disadvantage is that not every model has its transformation language. For instance, even though OQL [15] exists, object-oriented programming languages usually do not support querying collections of objects, with the exception of Microsoft's .NET languages implementing LINQ [68].

The second, "transformation first", approach (see Fig. 3.7) first performs transformation and the resulting model is mapped to the target model. Using the above example, the source model (3.2) would be first transformed into the intermediate model (3.3) and then mapped to the target model (3.4). The transformation language used in this case will be discussed in Section 3.3.1. The first advantage of this approach is that only one transformation language is needed, since the transformation is performed only on the source model, which in Knowledge Views is always the ontological model.

This solution is not without flaws—given a source and a target model, it is not always possible to create a transformation that, combined with mappings, would produce the desired target model from the source one. However, this solution seems powerful enough to produce usable models, even if they are not identical to the desired ones. In this solution it is paramount for the mappings to be as simple as possible, since the creator of transformations has to be able to predict how the model after transformation would be further changed during the mapping process.

The second approach, with the transformation before the mapping, has been chosen to be used in Knowledge Views. Nevertheless, both solutions had been tried prior to making the final decision. Incorporating the transformation into the mapping process had also been considered. It had been rejected, though, because it made the mappings more complex and that would influence the ease of use.

3.3.1 Transformation language

The transformation process needs an adequate language. Since the transformation is performed on DL models, the language has to operate on DL entities: concepts, roles, attributes, individuals and values. In database views a new table, which is a realisation of an entity set, is defined with a query. In Knowledge Views concepts, roles and attributes, which are interpreted as sets of entities, are also defined with queries. The query language, which is a part of the transformation language, is rule-based and it resembles rules in SWRL [48].

A rule consists of a head and a body. The head is a concept, role or attribute atom. The body is a conjunction of atoms. Apart from the concept, role and attribute atoms, the body can also contain built-in atoms. Built-ins greatly enhance expressivity. Built-ins provide means to perform arithmetic operations, operations on strings and so on. The set of built-ins can be easily expanded.

A sample rule:

$$\mathit{Adult}(i) \leftarrow (\mathit{Man} \sqcup \mathit{Woman})(i) \land \mathit{hasAge}(i, a) \land \mathit{built\text{-}in\text{:}ge}(a, 18) \tag{3.6}$$

states that a man or a woman with age above or equal to 18 is an adult. This example shows the usage of built-ins as well as the possibility of using complex expressions as predicates. Note the difference between the above rule and the following pair of rules:

$$\begin{array}{lcl} \mathit{Adult}(i) & \leftarrow & \mathit{Man}(i) \land \mathit{hasAge}(i, a) \land \mathit{built\text{-}in\text{:}ge}(a, 18) \\ \mathit{Adult}(i) & \leftarrow & \mathit{Woman}(i) \land \mathit{hasAge}(i, a) \land \mathit{built\text{-}in\text{:}ge}(a, 18) \end{array} \tag{3.7}$$

All adults derived from (3.7) will be derived from (3.6), but not the other way round. The reason is that there might be some individuals that are either a woman or man, but we do not know which one. This shows that the possibility of using DL expressions as predicates adds some expressivity.

Rules are unidirectional, i.e. they are a description of how to create assertions in the target ontology based on the assertions from the source ontology. Those assertions can be read, but what if one would like to insert some assertions through the view? When considering rules like:

$$\mathit{Person}(i) \leftarrow (\mathit{Man} \sqcup \mathit{Woman})(i) \tag{3.8}$$

one might think nothing simpler than reversing the rule. However, another rule:

$$\mathit{hasUncle}(x, z) \leftarrow \mathit{hasFather}(x, y) \land \mathit{hasBrother}(y, z) \tag{3.9}$$

might be more problematic. When inserting an assertion

$$\mathit{hasUncle}(\mathrm{Mary}, \mathrm{Ben}) \tag{3.10}$$

variables x and z from the rule are bound, but y is not. A solution would be to assign a unique random individual id to y. This could be compared to blank nodes in RDF. Let us take into consideration another rule:

$$\begin{aligned} \textit{hasTotalLength}(x, v) \leftarrow\ & \textit{hasLength1}(x, v1) \wedge \textit{hasLength2}(x, v2) \\ & \wedge \textit{built-in:add}(v, v1, v2) \end{aligned} \tag{3.11}$$

In this example there are infinitely many pairs of $v1$ and $v2$ that satisfy the rule's body for a given v. However, not every pair is legal since the relationship between those values is defined by the *add* built-in.

This is a typical problem when trying to inverse a function that is not one-to-one. The adopted solution to this problem is to allow the user to define reverse rules that would create source ontology assertions from the target ontology assertions. The forward rules work like queries on the source model, while the reverse rules work like triggers. Nevertheless both sets of rules are written using the same syntax. The proposed solution's simplicity makes it easy to predict the results of inserting an assertion through the view. However, it does not guarantee that after inserting an assertion, it can be retrieved afterwards.

The transformation language presented is a part of the Knowledge View definition language called xNeeK. xNeeK was developed as a part of this thesis and is further described in Section 5.1.4. A specification of xNeeK is in Appendix C.

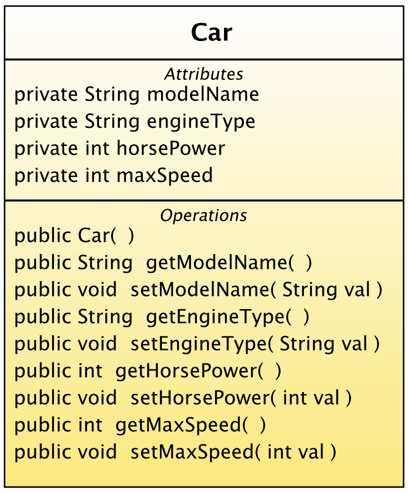
3.3.2 Transformation example

Let us go back to the car dealer example. The dealer has a knowledge source with information about cars conforming to the following terminology:

$$\begin{aligned} Car &\sqsubseteq \exists hasEngine.Engine \\ Engine &\sqsubseteq \exists hasPower.int \\ Engine &\sqsubseteq \exists hasCompressionRatio.double \\ DieselEngine &\sqsubseteq Engine \\ PetrolEngine &\sqsubseteq Engine \\ Car &\sqsubseteq \exists hasMaxSpeed.int \\ Car &\sqsubseteq \exists hasModelName.string \end{aligned} \tag{3.12}$$

This is the source model (see Fig. 3.7). This terminology defines four concepts. The first concept is *Car* with *hasEngine* role, *hasMaxSpeed* attribute and *hasModelName* attribute. The second concept is *Engine* with *hasPower* attribute and *hasCompressionRatio* attribute. The last two concepts are *DieselEngine* and *PetrolEngine*. They are both types of *Engine*.

The dealer wants to create an information web page using object-oriented technology like Java. Information to be presented on the web page does not



```
private String modelName
private String engineType
private int horsePower
private int maxSpeed
```

```
public Car( )
public String  getModelName( )
public void  setModelName( String val )
public String  getEngineType( )
public void  setEngineType( String val )
public int  getHorsePower( )
public void  setHorsePower( int val )
public int  getMaxSpeed( )
public void  setMaxSpeed( int val )
```

Figure 3.8: Sample Car class

contain everything present in the source ontology and is stored in a single class (see Fig. 3.8) that constitutes the target model (see Fig. 3.7).

Having the given terminology and the class diagram of the required classes, the next step is to determine the terminology that corresponds to the required object model—the intermediate model (see Fig. 3.7):

$$\begin{aligned}
\exists modelName.\top &\sqsubseteq Car\\
\top &\sqsubseteq \forall modelName.string\\
\top &\sqsubseteq\ \leq 1 modelName.\top\\
\exists engineType.\top &\sqsubseteq Car\\
\top &\sqsubseteq \forall engineType.string\\
\top &\sqsubseteq\ \leq 1 engineType.\top\\
\exists horsePower.\top &\sqsubseteq Car\\
\top &\sqsubseteq \forall horsePower.int\\
\top &\sqsubseteq\ \leq 1 horsePower.\top\\
\exists maxSpeed.\top &\sqsubseteq Car\\
\top &\sqsubseteq \forall maxSpeed.int\\
\top &\sqsubseteq\ \leq 1 maxSpeed.\top
\end{aligned} \tag{3.13}$$

In this terminology there are three statements for every attribute in the *Car* class. The first statement defines the *modelName*'s domain, the second defines the range and the third specifies that the attribute is functional. Similarly domains and ranges for *engineType*, *horsePower* and *maxSpeed* are defined. All the attributes are functional.

Finally, once the source model and the intermediate model are known, the transformation between them can be defined:

$$\begin{aligned}
\mathit{Car}(i) &\leftarrow \mathit{Car}(i)\\
\mathit{modelName}(s,o) &\leftarrow \mathit{Car}(s) \wedge \mathit{hasModelName}(s,o)\\
\mathit{engineType}(s,\text{"diesel"}) &\leftarrow \mathit{Car}(s) \wedge \mathit{hasEngine}(s,o)\\
&\quad\ \wedge \mathit{DieselEngine}(o)\\
\mathit{engineType}(s,\text{"petrol"}) &\leftarrow \mathit{Car}(s) \wedge \mathit{hasEngine}(s,o)\\
&\quad\ \wedge \mathit{PetrolEngine}(o)\\
\mathit{horsePower}(s,o) &\leftarrow \mathit{Car}(s) \wedge \mathit{hasEngine}(s,e)\\
&\quad\ \wedge \mathit{hasPower}(e,p)\\
&\quad\ \wedge \mathit{built\text{-}in{:}mul}(o,p,1.36)\\
\mathit{maxSpeed}(s,o) &\leftarrow \mathit{Car}(s) \wedge \mathit{hasMaxSpeed}(s,o)
\end{aligned} \tag{3.14}$$

The first rule just rewrites all cars from one ontology to the other. The second rule rewrites the model name. The third and fourth rules assign engine type label. The fifth rule calculates horse power from power in kW, assigning it to the car instead of the engine. The last rule rewrites max speed attribute.

3.4 Knowledge sources

Knowledge Views are created on top of knowledge sources. But what exactly is a knowledge source?

Definition 3 (Knowledge source) *A knowledge source is a provider of information together with metadata (terminology). A knowledge source performs reasoning over the existing information.*

The terminology from the knowledge source is used during the mapping process to create the target model. The information is passed through to the user. However, if the optional transformation step is used, the information is changed and then passed to the user, but the original terminology is hidden and only the target terminology of the transformation process is visible. This makes the original terminology unimportant from the user point of view—it can always be changed.

Moreover, terminology is only used once during the view creation. It is assumed that terminology changes infrequently. This assumption is still valid for most information systems, since neither database schemata nor classes definitions change very often in a working environment.

This gives some additional possibilities. Instead of a knowledge source let us consider an information source. An information source differs from a knowledge source in that even though it conforms to some metadata, it does not necessarily expose it. Furthermore, information source does not perform reasoning. Transformation process can be used to augment an information source with an arbitrary terminology. After the transformation, a reasoning process that uses the added terminology and the transformed information can be performed. This means that by adding an additional layer on top of information sources, those information sources can be changed into knowledge sources. As a consequence Knowledge Views can be created on top of an information source and not only on top of a knowledge source. The Knowledge Views have been equipped with capabilities to reason over external information sources similar to the capabilities of KL system [42], SDL library [5] or as described in [25]. These reasoning capabilities of Knowledge Views are more thoroughly described in Sections 4.1.2 and 5.1.2.

Considering information encoded in the predicate form:

$$\begin{gathered} \mathit{Car}(\text{Clio}) \\ \mathit{hasEngine}(\text{Clio}, 1.5 \text{ dCi } 85) \\ \mathit{hasPower}(1.5 \text{ dCi } 85, 63) \end{gathered} \tag{3.15}$$

which takes part in the transformation process, information integration is relatively easy. Information from various sources can be transformed to conform to a common terminology using the already defined transformation process and then combined with the union operator. However, the information integrity is not automatically checked. The integrity can be checked using the reasoning capabilities and tested by issuing appropriate queries. A complete integrity check may require reading all information from all the sources and therefore might not always be feasible.

The capabilities of reasoning over external information sources as well as information integration are a side effect of the chosen approach rather than a goal by itself. However, these capabilities allow Knowledge Views to utilise information scattered across the Internet and stored using various technologies like knowledge bases, databases or plain HTML pages. This is very valuable, especially in the era of distributed information systems.

3.5 General architecture

All the described elements of Knowledge Views are combined in the architecture depicted in Fig. 3.9. It is a layered architecture, where each layer is responsible for one of the described functions like mapping or transformation.

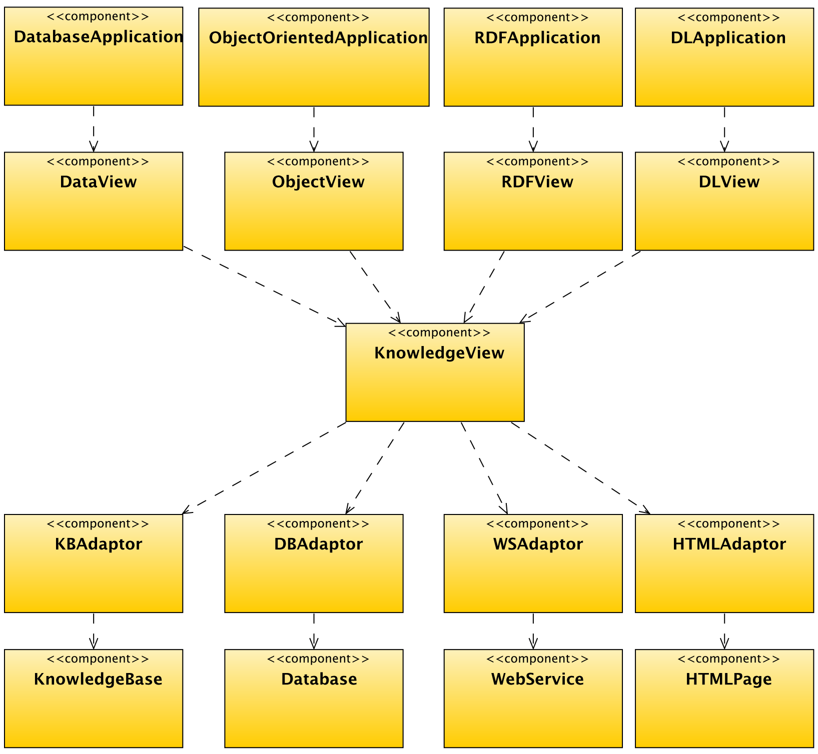


Figure 3.9: Knowledge Views' architecture

3.5.1 The application layer

The upper layer in this architecture are applications. These can be different applications, i.e. using different storage types: databases, object storage, RDF storage, etc. One way for those applications to benefit from reasoning is to change the storage to a knowledge base. However, one must take into account the costs and risk of such a change. Changing a storage type in an application requires modifications in the source code of the application. This produces cost. Moreover changing anything in a working application comes with the risk of introducing new errors into the source code. Additional costs can be caused by the need of developers to familiarise with a new storage type. One way to mitigate these costs and risk is to use a wrapper on the knowledge base that provides access to the knowledge base via the interface already used by the application. This way the storage can be substituted without any modifications in the application. However, such a change might not be seamless if the application utilises some vendor specific extensions of the storage in question. In such a situation even changing the storage to a storage of the same type, but different vendor, is problematic. These kinds of problems are not considered an issue, since it is the developer's conscious choice to bind the system to vendor specific solutions. The Knowledge Views can be considered as a wrapper providing the interface already used by the application, therefore providing means to gradually migrate existing applications to Semantic Web technologies.

3.5.2 The views layer

The second layer consists of views components. These components are responsible for providing the appropriate interfaces:

- DataView—SQL,
- ObjectView—OQL,
- RDFView—SPARQL,
- DLView—a DL query language.

This layer is also responsible for the mappings:

- DataView—ontology-relational mapping,
- ObjectView—ontology-object mapping,
- RDFView—encoding DL ontology in RDF,
- DLView—no mapping needed.

3.5.3 The Knowledge View layer

The Knowledge View layer is the core of the architecture. It provides the transformation capabilities used by the upper layer. Moreover, it provides some reasoning capabilities that allow using external sources which lack them. This layer can also combine information from various and possibly heterogeneous external sources like: knowledge bases, databases, web services, web pages, etc.

3.5.4 The adaptor layer

The adaptor layer wraps various information sources to provide a common interface that can be directly used by the Knowledge View layer. This includes encoding information in a predicate form. An adaptor can do as little as forwarding a call to the underlying source like a knowledge base adaptor does or can contain some caching and indexing code that might be required by an HTML adaptor, since parsing a set of HTML pages during every query is not reasonable. Moreover, there might be some general configurable adaptors like a database adaptor, thanks to the standardisation of database systems, or there might be a need for a separate adaptor for every source, which is the case with web services, since they only standardise the information exchange protocol, not the information semantics.

This layer in general is a realisation of the adaptor design pattern [19], but also uses other related design patterns. The exact selection of techniques is dependent on the specifics of the source wrapped, but also on some non-functional requirements like performance requirements.

3.5.5 The information sources

There is a wide variety of information sources throughout the Internet: knowledge bases, databases, HTML pages, sources accessed via web services, etc. They all have their advantages and disadvantages and it is not possible to just take all the sources and rewrite all the information to some common format, for example RDF. Moreover, obsolete technologies have the tendency to remain in use, even if newer, superior technologies could easily replace them. Fortran is an example of an over a half century old technology that is still in use in spite of all the people who wish to forget it. In the seventies Edsger W. Dijkstra stated [21]:

> The sooner we can forget that FORTRAN ever existed, the better, for as a vehicle of thought it is no longer adequate: it wastes our brainpower, and it is too risky and therefore too expensive to

> use. FORTRAN'S tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes. I pray daily that more of my fellow-programmers may find the means of freeing themselves from the curse of compatibility.

Yet, over thirty more years have passed and Fortran is still widely used, for example in numerical weather prediction. Therefore, to be successful the Knowledge Views need to be able to utilise any kind of information source.

The various information sources can be treated as potentially distributed services providing required information, hence the proposed architecture corresponds in a way with the Service Oriented Architecture (SOA) [53]. Moreover, one instance of the Knowledge Views can become a source for another instance of the Knowledge Views creating a hierarchy, where each layer adds some value to the overall result. In this way this is also similar to SOA, where services can use one another.

Chapter 4

Case studies and experiments

To prove the proposition stated in Chapter 1 case studies were conducted and some experiments were performed. They focus on showing compatibility of Knowledge Views with contemporary information systems as well as on showing the ease of use of the Knowledge Views.

4.1 Simulation with the Knowledge Views

This case study has been inspired by [58]—it shows an example configuration of Knowledge Views components that can be used to simulate malware spreading throughout a local network. This example focuses on the usage of the individual components of the Knowledge Views leaving the process of malware spreading simplified. Nevertheless, this example shows that some analysis of network vulnerability can be performed at a very low cost using logic-based tools.

This example assumes there is an information source describing malware. The malware description contains its requirements, reproduction capabilities and the ability to install other malware. The information is similar to that contained on web pages maintained by antivirus software producers. This information source is considered to be external, i.e. it could be maintained by some antimalware organisation and is available only as a read-only source for the Knowledge Views user.

Another information source is the network layout. It contains the information about the nodes, i.e. what software is installed on each node and how the nodes are connected. The connections are defined in terms of the ability to download or upload files, however this is just a special case of malware spreading through services running on a node, for example HTTP server or FTP server. The multiuser nature of contemporary operating systems is not

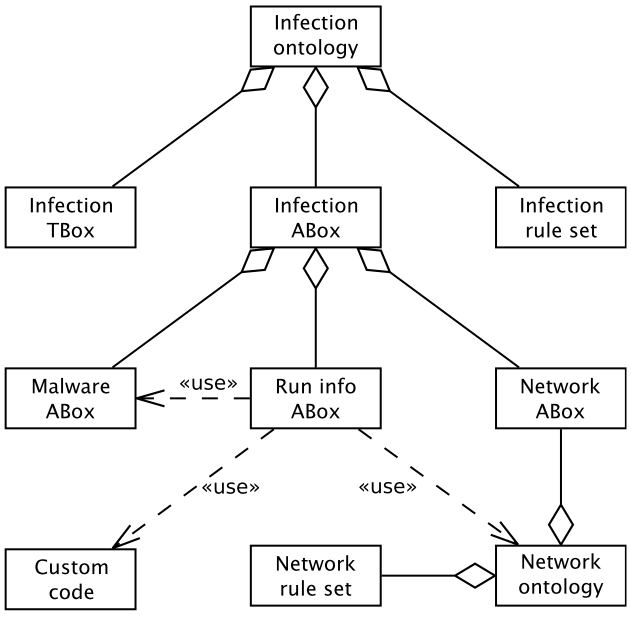


Figure 4.1: The infection ontology

modelled so as not to obscure the example, even though it is crucial for the computer security. This omission, however, can reflect the common usage of computers—non-power users often work using the computer's administrative account, because they are not aware of the risks or they consider it inconvenient to use a limited account for non-administrative tasks. The network layout information source is considered to be internal, i.e. the user has full control over it.

In this example the role of the Knowledge Views library is to combine the information sources, reason over them and provide the user application with a simple interface to access the information.

4.1.1 Ontology

The ontology used in this example is not a simple file loaded to a knowledge base, but rather a set of modules (see Fig. 4.1) that interact with each other to produce the illusion of a knowledge base with a single ontology loaded.

This example shows some of the possible applications of the view concept in ontology creation. There are two information sources given. They are seen as ABoxes (the malware ABox and the network ABox). Additionally some custom code that queries other ABoxes is also seen as an ABox (run info ABox). The network ABox combined with the network rule set is seen as the network ontology. The malware, network and run info ABoxes together are seen as a single ABox (the infection ABox). Finally the infection ABox combined with the infection TBox and the infection rule set is seen as a single ontology (the whole infection ontology) that is used by an application. One of the main advantages of configuring an ontology using some components rather than creating one monolithic entity is the ability to change individual components or even the whole configuration without changing the resulting ontology or rather the resulting view of the ontology. It is also worth noting that the information sources are used directly without the need of rewriting their content—rewriting a whole information source is not always feasible. Moreover, this approach supports semantic modularity, which can help to comprehend the created ontology if used properly.

Terminology

The terminology (infection TBox) is provided to the system via an OWL file which corresponds to the (4.1) DL axiom set. This terminology defines three concepts: *Computer*, *Malware* and *Software*. A *Computer canBrowse* another *Computer* or *canDownloadFrom* another *Computer*. A *Malware canRunOn* a *Computer*. A *Computer canUploadTo* a *Computer*. A *Malware* has at most one *description*. A *Computer* can be *endangeredBy* a *Malware*. A *Computer* can have *Software* installed on it, which is denoted by *hasInstalled* role. A *Malware* can install other *Malware*, which is denoted by *hasPayload* role. A *Computer* can have at most one *ip* address. A *Malware* can have at most one *name*. A *Malware requires* particular *Software* to be able to run on a *Computer*. A *Malware* can represent a particular *riskLevel*. Once infected a *Computer* is a *sourceOf Malware*.

In this example, the terminology is used to define the data schema visible by the user rather than to perform reasoning. However, even such a simple terminology can be used to infer some new facts not explicitly given. For example, there is no need to state explicitly which individuals belong to the *Software* concept—this can be inferred based on the range of *hasInstalled* and *requires* roles.

Rules

In this example the infection rule set defines how malware propagates throughout network—which computers are endangered by malware and which computers can be used to distribute malware. The rules drive the reasoning pro-

$$
\begin{aligned}
Computer &\sqsubseteq \top \\
Malware &\sqsubseteq \top \\
Software &\sqsubseteq \top \\
\exists canBrowse.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall canBrowse.Computer \\
\exists canDownloadFrom.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall canDownloadFrom.Computer \\
\exists canRunOn.\top &\sqsubseteq Malware \\
\top &\sqsubseteq \forall canRunOn.Computer \\
\exists canUploadTo.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall canUploadTo.Computer \\
\exists description.\top &\sqsubseteq Malware \\
\top &\sqsubseteq \forall description.string \\
\top &\sqsubseteq \leq 1 description.\top \\
\exists endangeredBy.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall endangeredBy.Malware \\
\exists hasInstalled.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall hasInstalled.Software \\
\exists hasPayload.\top &\sqsubseteq Malware \\
\top &\sqsubseteq \forall hasPayload.Malware \\
\exists ip.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall ip.string \\
\top &\sqsubseteq \leq 1 ip.\top \\
\exists name.\top &\sqsubseteq Malware \\
\top &\sqsubseteq \forall name.string \\
\top &\sqsubseteq \leq 1 name.\top \\
\exists requires.\top &\sqsubseteq Malware \\
\top &\sqsubseteq \forall requires.Software \\
\exists riskLevel.\top &\sqsubseteq Malware \\
\top &\sqsubseteq \forall riskLevel.int \\
\top &\sqsubseteq \leq 1 riskLevel.\top \\
\exists sourceOf.\top &\sqsubseteq Computer \\
\top &\sqsubseteq \forall sourceOf.Malware
\end{aligned}
\tag{4.1}
$$

cess. The infection rule set consists of the following rules:

$$\begin{aligned}
\mathit{sourceOf}(\mathit{internet}, m) &\leftarrow \mathit{Malware}(m) \\
\mathit{endangeredBy}(c, m) &\leftarrow \mathit{canRunOn}(m, c) \wedge \mathit{sourceOf}(c, m) \\
\mathit{endangeredBy}(c, m) &\leftarrow \mathit{canRunOn}(m, c) \\
&\quad \wedge \mathit{canDownloadFrom}(c, s) \wedge \mathit{sourceOf}(s, m) \\
\mathit{endangeredBy}(c, m) &\leftarrow \mathit{canRunOn}(m, c) \wedge \mathit{canBrowse}(c, s) \\
&\quad \wedge \mathit{sourceOf}(s, m) \\
\mathit{sourceOf}(s, m) &\leftarrow \mathit{hasPayload}(n, m) \wedge \mathit{endangeredBy}(s, n) \\
\mathit{sourceOf}(s, m) &\leftarrow \mathit{canUploadTo}(c, s) \wedge \mathit{sourceOf}(c, m)
\end{aligned} \tag{4.2}$$

The first rule states that *internet* is the *sourceOf* all *Malware*. The second rule states that if a computer c is a *sourceOf* a malware m that *canRunOn* on the computer then the computer is *endangeredBy* the malware. For example if a computer runs an FTP server and the malware executable is on the FTP server then this computer is a *sourceOf* the malware. If, moreover, the malware is able to run on this computer then this computer is *endangeredBy* it. The third rule states that if a malware m *canRunOn* a computer c and the computer *canDownloadFrom* another computer s which is a *sourceOf* the malware m then the computer c is *endangeredBy* the malware m. The fourth rule is similar to the third rule. The only difference is that in the fourth rule a computer c *canBrowse* computer s, while in the third rule a computer c *canDownloadFrom* computer s. The fifth rule states that if a computer s is *endangeredBy* malware n and this malware *hasPayload* in the form of malware m then the computer s is a *sourceOf* malware m. The last rule states that if a computer c is a *sourceOf* malware m and this computer *canUploadTo* a computer s then the computer s also becomes a *sourceOf* malware m. These rules can be easily extended to support other means of malware propagation, for example transmission via USB flash drives.

ABox

The third part of the infection ontology is the infection ABox. This ABox by itself only merges the three underlying ABoxes: the malware ABox, the network ABox and the run info ABox. It is introduced solely to combine what is below it into a single ABox, so the upper layer does not have to explicitly handle multiple sources, which would unnecessarily complicate the matters.

The malware ABox provides information about malware. In particular it provides assertions for:

- *Malware*,
- *description*,

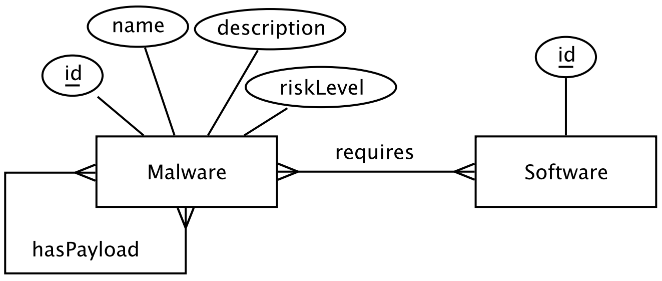


Figure 4.2: Malware database ERD

- *hasPayload*,
- *name*
- *requires* and
- *riskLevel*

from the (4.1) terminology. This ABox is backed by a relational database with a schema conforming to the ERD in Fig. 4.2.

The network ABox provides information about computers and connections between them. This ABox provides assertions for:

- *Computer*,
- *canBrowse*,
- *canDownloadFrom*,
- *canUploadTo*,
- *hasInstalled* and
- *ip*.

This ABox is backed by a relational database with a schema conforming to the ERD in Fig. 4.3.

The third ABox, the run info ABox, unlike the previous two, is not materialised in a database. Instead, it is a piece of software that can handle queries for instances of the *canRunOn* role. A *Malware canRunOn* a *Computer* if all the requirements of the *Malware* are met by the software installed on the *Computer*. The requirements are retrieved from the malware ABox, while the software installed on a particular *Computer* are retrieved from the network ontology. The Knowledge View's reasoning capabilities are not used

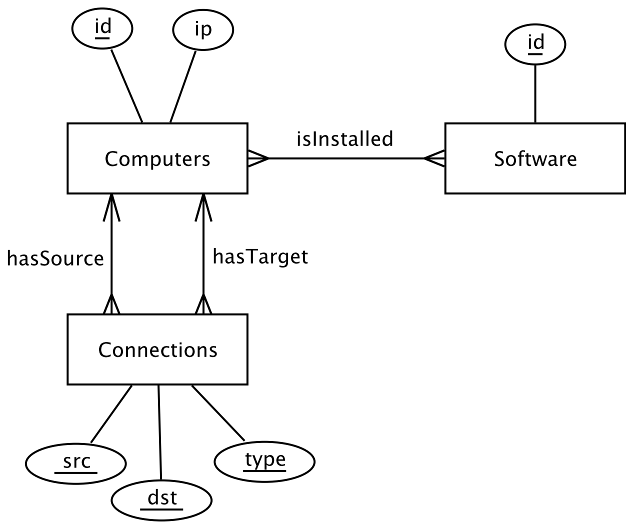


Figure 4.3: Network database ERD

to confront the two retrieved sets, because the reasoning is performed under open world assumption (OWA) [4]. As a consequence of this assumption, during the reasoning process, the requirements of a given *Malware* are interpreted as known requirements with the possibility that there are also some unknown requirements. This makes it impossible to determine whether all requirements are met. The custom code, on the other hand, can query the malware ABox for requirements and assume that the results contain all requirements. Then they can be confronted with the software installed on a given *Computer*.

The requirements might not always match exactly the installed software: for example some malware might require Windows operating system, while some computer might have Windows Vista installed. Windows Vista is still a Windows operating system, therefore the requirement is met. However, the information that Windows Vista is Windows has to be given explicitly. Moreover Windows Vista Service Pack 2 is both Windows Vista and Windows operating system and so on. To handle this case the network ontology is created (see Fig. 4.1). This ontology combines the network ABox with the network rule set that consists of the following rule:

$$\mathit{hasInstalled}(c, g) \leftarrow \mathit{hasInstalled}(c, s) \wedge \mathit{is}(s, g) \tag{4.3}$$

The instances of the *is* role act as a bridge between the network ABox and the malware ABox. These instances are stored in the network ABox to avoid creating yet another ABox for a single role, although such a configuration is also possible.

4.1.2 Reasoning

In the ontology configuration presented reasoning is performed by several components (see Fig. 4.1). The network ontology produces new facts based on the assertions from network ABox and network rule set. The network ontology user does not know which of the provided facts were stated explicitly and which were inferred. Since network ontology is used only as an ABox, it is seen as a set of assertions. From the user point of view it is not important where those assertions come from and whether they are materialised in some underlying source or calculated on demand.

The run info ABox, which is a custom code, also performs reasoning, i.e. it produces new facts based on some already known. However, this reasoning does not conform to description logic or Horn logic. Moreover, this reasoning is performed with closed world assumption (CWA) [4]. This component is seen as a repository of *canRunOn* assertions and the fact that those assertions are inferred on demand, and not stored, is unimportant.

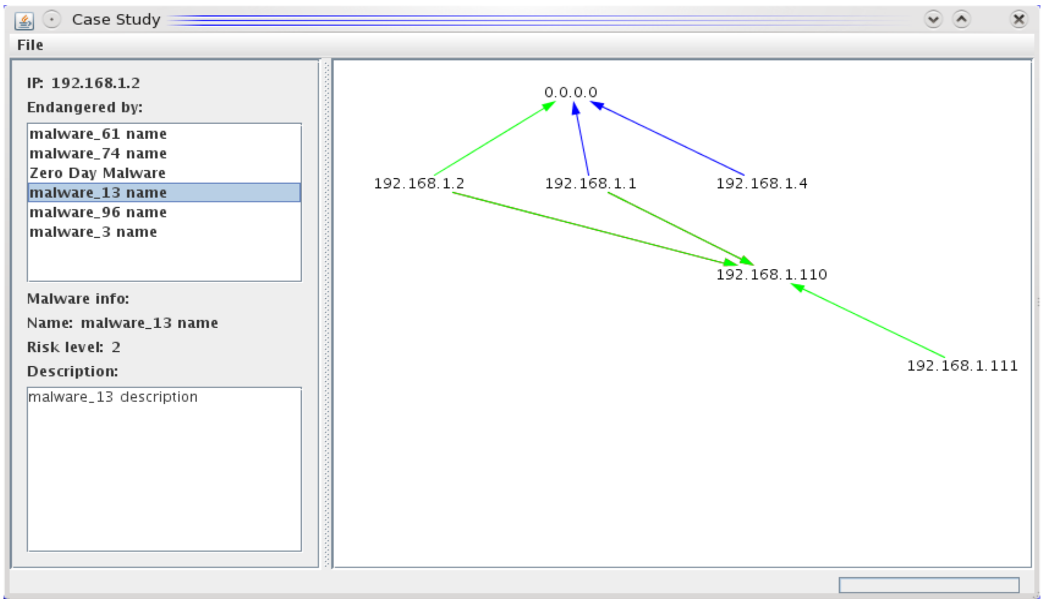
The next component which performs reasoning is the infection TBox. It performs terminological reasoning only. Again, the implementation is not important—this can be any existing DL reasoner wrapped to satisfy a certain interface that can respond to a basic set of terminological queries.

Finally, the infection ontology reasons over the infection ABox using the infection rule set and infection TBox. The results of this last reasoning are visible to the user. The user sees this ontology as a single monolithic entity, whereas in fact it consists of several components that reason over information from several sources.

The general idea is to provide the user only with the final results presented as a single view. All details are hidden under the hood, for example different kinds of reasoning used in various components. All details concerning the ontology implementation are known only to the knowledge engineer that created the ontology. All the components can be replaced at any time as long as they provide the same response to the same queries. This modular nature of the ontology implementation can help the knowledge engineer in the ontology creation process.

4.1.3 Application

On top of the ontology a sample application has been built. It draws a graph representing the computer network (see Fig. 4.4). When a computer is selected a list of malware that endanger that computer is displayed. Selecting some malware from the list causes the application to display information about that malware. The application also provides capabilities to edit the



IP: 192.168.1.2
Endangered by:
malware_61 name
malware_74 name
Zero Day Malware
malware_13 name
malware_96 name
malware_3 name

Malware info:
Name: malware_13 name
Risk level: 2
Description:
malware_13 description

Figure 4.4: Sample Object View application

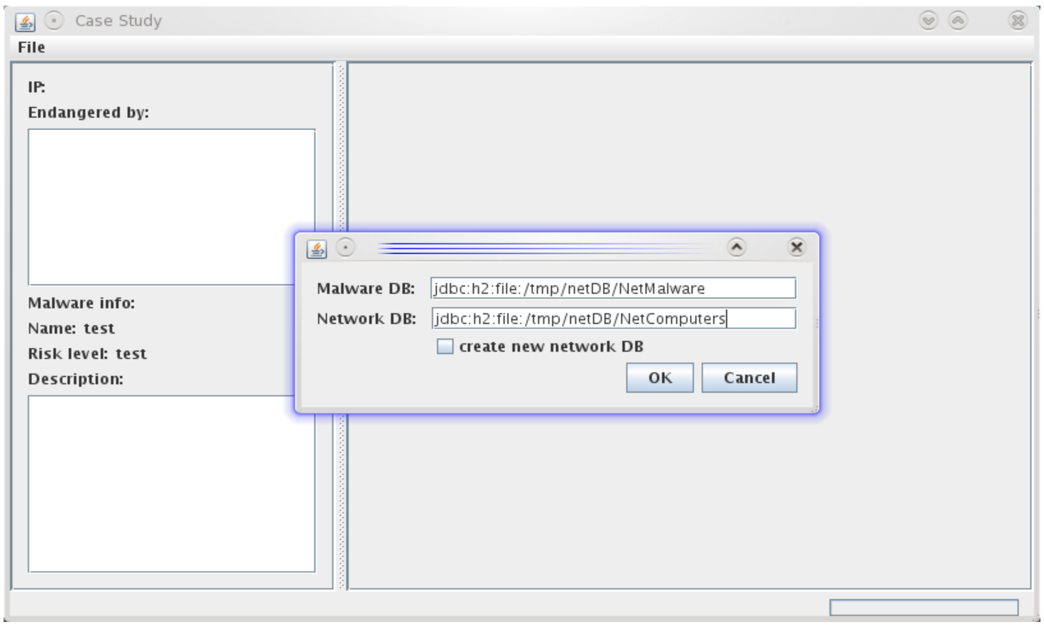
network, however the malware information is read-only.

Usage of the application

First, the application user inserts the URLs of the sources to be used. There is a possibility of connecting to an existing network layout database or a new one can be created (see Fig. 4.5). Once the application has connected to the sources the user can:

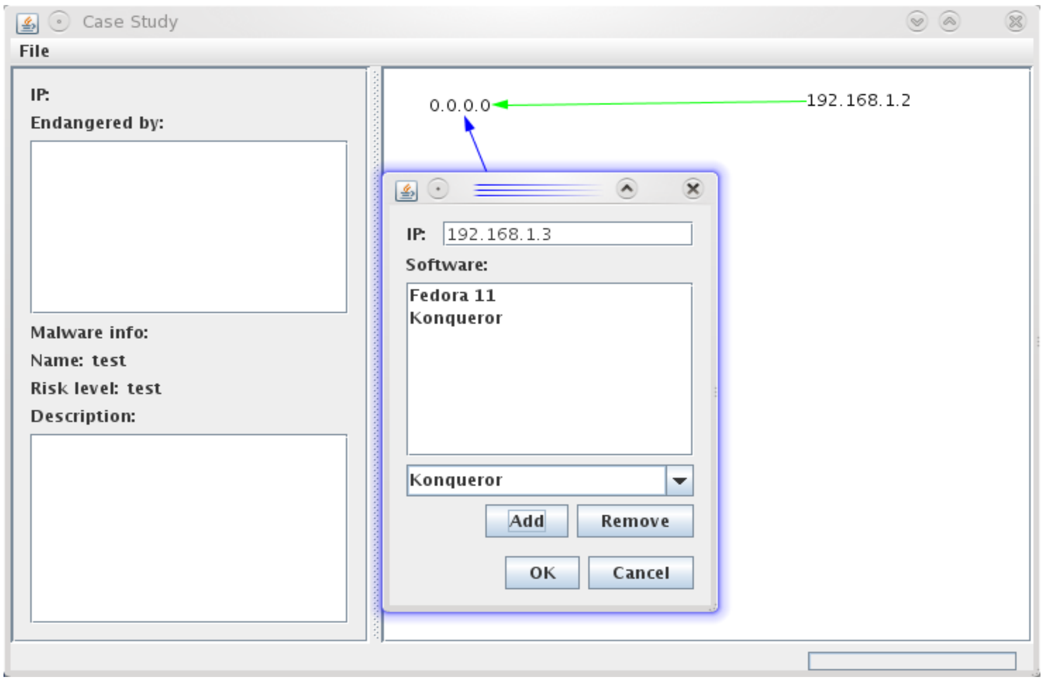
- add new computers to the network (see Fig. 4.6),
- edit which software is installed on a computer,
- remove existing computers,
- define new connections between computers,
- remove existing connections,
- see what malware endangers the selected computer,
- see information about malware endangering the selected computer.

The application displays only some basic information concerning the malware spread throughout the network. However, the Knowledge View, on top of



Malware DB: jdbc:h2:file:/tmp/netDB/NetMalware
Network DB: jdbc:h2:file:/tmp/netDB/NetComputers
☐ create new network DB

Figure 4.5: Connection dialog



IP: 192.168.1.3
Software:
Fedora 11
Konqueror

Konqueror

Figure 4.6: New computer dialog

which the application is built, can provide more information. Therefore, the application can be easily extended to present more information, for example it could:

- list all endangered computers,
- list all computers endangered by malware having risk level above a given threshold, etc.

However, for large networks and large malware databases “list all” queries can consume considerable amount of time, since they could require reading whole databases and reasoning over all the retrieved information.

The View concept in the application

The application uses a single ontology, possibly composed of several components, in two different ways. On one hand the application presents the reasoning results, i.e. which malware endangers which computers. This is accompanied by the information about the malware. On the other hand the application queries the ontology to retrieve the network layout to draw the graph of the network. In those two usages different information is required, therefore two distinct views are created.

Computer view The information about a computer that is presented to the user contains:

- IP,
- a collection of malware that endangers the computer.

However, when creating a new computer there is one more piece of information about the computer that is necessary for the reasoning process: the list of installed software. Information about malware that is presented to the user contains:

- name,
- risk level,
- description.

Therefore, the application needs classes corresponding to the class diagram depicted in Fig. 4.7. The instances of these classes are provided by an Object View which is built on top of a Knowledge View that selects only the

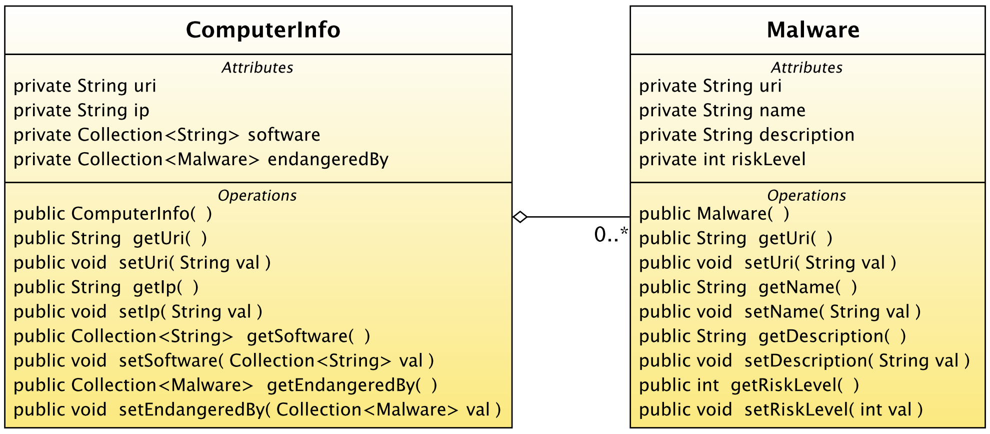

ComputerInfo
Attributes
private String uri
private String ip
private Collection<String> software
private Collection<Malware> endangeredBy
Operations
public ComputerInfo( )
public String getUri( )
public void setUri(String val)
public String getIp( )
public void setIp(String val)
public Collection<String> getSoftware( )
public void setSoftware(Collection<String> val)
public Collection<Malware> getEndangeredBy( )
public void setEndangeredBy(Collection<Malware> val)
0..*
Malware
Attributes
private String uri
private String name
private String description
private int riskLevel
Operations
public Malware( )
public String getUri( )
public void setUri(String val)
public String getName( )
public void setName(String val)
public String getDescription( )
public void setDescription(String val)
public int getRiskLevel( )
public void setRiskLevel(int val)

Figure 4.7: Computer view class diagram

necessary information from the source ontology and provides rules for writing facts back to the ontology. The Knowledge View is defined by two sets of transformation rules: forward rule set and reverse rule set. In the first rule set (forward rule set) the consequent of each rule defines information provided to the Object View, while the antecedent of each rule matches information in the source ontology:

$$\begin{aligned}
\mathit{ComputerInfo}(i) &\leftarrow \mathit{Computer}(i) \\
\mathit{ip}(s,o) &\leftarrow \mathit{ip}(s,o) \\
\mathit{software}(s,o) &\leftarrow \mathit{hasInstalled}(s,i) \wedge \mathit{built\text{-}in{:}uriToString}(i,o) \\
\mathit{endangeredBy}(s,o) &\leftarrow \mathit{endangeredBy}(s,o) \\
\mathit{Malware}(i) &\leftarrow \mathit{Malware}(i) \\
\mathit{name}(s,o) &\leftarrow \mathit{name}(s,o) \\
\mathit{riskLevel}(s,o) &\leftarrow \mathit{riskLevel}(s,o) \\
\mathit{description}(s,o) &\leftarrow \mathit{description}(s,o)
\end{aligned} \tag{4.4}$$

The first of the above rules rewrites instances of *Computer* concept as instances of *ComputerInfo* concept—only the name of the concept changes. The third rule for all instances of *hasInstalled* role transforms the URI denoting role filler i identity into a string value o. The results are written as instances of *software* attribute. The rest of the rules simply rewrite instances of *Malware* concept, *endangeredBy* role as well as *ip*, *name*, *riskLevel* and *description* attributes. The names of variables i, s and o are insignificant i.e. they are randomly chosen letters.

The above rule set allows the application to read information from the source ontology. To be able to write new information into the source ontology through the Knowledge View a second rule set (reverse rule set) is needed. The second rule set works in the opposite direction than the first rule set. The antecedent matches information passed from the Object View and the consequent is stored in the source ontology:

$$\begin{aligned} \mathit{Computer}(i) &\leftarrow \mathit{ComputerInfo}(i) \\ \mathit{ip}(s,o) &\leftarrow \mathit{ip}(s,o) \\ \mathit{hasInstalled}(s,i) &\leftarrow \mathit{software}(s,o) \wedge \mathit{built\text{-}in{:}uriToString}(i,o) \end{aligned} \tag{4.5}$$

These rules change string values back to *Software* individuals, but what is more important is that there are fewer rules in this set that there were in the first rule set—not everything that can be read through this view can be inserted back into the source ontology. On a write request the Object View acts in a very simple way: it encodes as assertions all the information stored in the object graph that is passed to become persistent and inserts the assertions back to the underlying ontology. However, in this case some of the information should not be written to the ontology, in particular the $\mathit{endangeredBy}(s,o)$ assertions. If $\mathit{endangeredBy}(s,o)$ assertions were written to the ontology and the information from which those assertions were inferred changed, the ontology would no longer contain valid information. Moreover, no information about malware is written back since the source containing the information is read-only. Thanks to the Knowledge View consisting of these rules the Object View does not need to know which of the assertions should be written back and which should not. It simply stores all the assertions and the underlying view selects what will actually be written back and what will be omitted.

Network view The library used for visualising the network layout requires the network to be modelled with both nodes and connections represented as classes. This is different from the source ontology (4.1) where connections are modelled as three roles: *canBrowse*, *canDownloadFrom* and *canUploadTo*. Therefore, a view is created that changes the data schema in a way that allows the retrieved objects to be directly used by the visualisation library (see Fig. 4.8).

The rules that retrieve *Computer* and its *ip* from the source ontology are straightforward:

$$\begin{aligned} \mathit{Computer}(i) &\leftarrow \mathit{Computer}(i) \\ \mathit{ip}(s,o) &\leftarrow \mathit{ip}(s,o) \end{aligned} \tag{4.6}$$

They just rewrite the information to the view. However rules that retrieve information about *Connection*s (4.7) are more complex. These rules transform

$$
\begin{aligned}
\mathit{Connection}(i) \leftarrow{} & \mathit{canBrowse}(s,o) \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+browse+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{source}(i,s) \leftarrow{} & \mathit{canBrowse}(s,o) \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+browse+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{target}(i,o) \leftarrow{} & \mathit{canBrowse}(s,o) \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+browse+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{type}(i,1) \leftarrow{} & \mathit{canBrowse}(s,o) \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+browse+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{Connection}(i) \leftarrow{} & \mathit{canDownloadFrom}(s,o)\\
& \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+download+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{source}(i,s) \leftarrow{} & \mathit{canDownloadFrom}(s,o)\\
& \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+download+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{target}(i,o) \leftarrow{} & \mathit{canDownloadFrom}(s,o)\\
& \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+download+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
\mathit{type}(i,2) \leftarrow{} & \mathit{canDownloadFrom}(s,o)\\
& \wedge \textit{built-in:uriToString}(s,ss)\\
& \wedge \textit{built-in:uriToString}(o,os)\\
& \wedge \textit{built-in:stringConcat}(is,ss,\text{"+download+"},os)\\
& \wedge \textit{built-in:uriToString}(i,is)\\
& \vdots
\end{aligned}
\tag{4.7}
$$

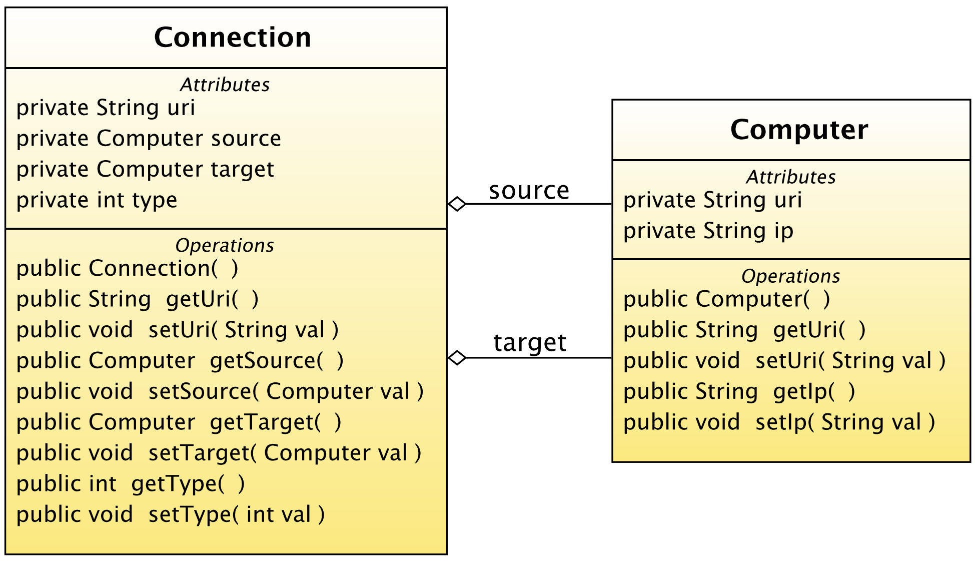


Figure 4.8: Network view class diagram

roles *canBrowse*, *canDownloadFrom* and *canUploadTo* to concept *Connection*, which was not present in the original ontology. To do this for every role instance a new individual has to be created. That individual is the subject of *source* and *target* roles. Additionally the *type* attribute is introduced to distinguish between the original three roles. These rules could be written in an equivalent shorter version using conjunction in the head of the rules.

This example shows that a single ontology modelling a domain in a certain way can be easily used by users (human or software) that require different models. All the different users can see the source ontology through a different view. This way all the users get the models they require, while still using a single source that is shared.

Apart from the presented rules this view has also some reverse rules that enable us to store new connections in the underlying source:

$$\begin{aligned} canBrowse(s,o) &\leftarrow type(i,1) \wedge source(i,s) \wedge target(i,o) \\ canDownloadFrom(s,o) &\leftarrow type(i,2) \wedge source(i,s) \wedge target(i,o) \\ canUploadTo(s,o) &\leftarrow type(i,3) \wedge source(i,s) \wedge target(i,o) \end{aligned} \tag{4.8}$$

These rules change the instances of *Connection* back to instances of the appropriate roles. The reverse rules in this view ignore *Computer* and *ip*. This is because this view contains only partial information about *Computer*s. Let us consider what would happen if writing *Computer* had been allowed and someone decided to remove an object of *Connection* class. The object view created on top of this knowledge view would translate the *Connection* object to a set of assertions. However, since *Connection* has references to *Computer* objects (see Fig. 4.8), those objects would also be included in the deletion process. Assertions stating that an individual is a *Computer* and which *ip* it has would be deleted, leaving information about installed software intact in the source ontology. That could lead to inconsistencies.

When working with Object Views one has to be careful, because objects are mapped to sets of assertions and if an object has references, the store and delete operations work on the whole object graph recursively. On one hand working with whole objects can help maintain consistency, but on the other hand the recursive nature of the load, store and delete operations can lead to unintentional loading or deleting large parts or even the whole underlying knowledge base with just one procedure call. The presence of connections between the individuals in the knowledge base is important for reasoning, therefore it is not always possible to handle this problem by changing the source knowledge base. However, the described situation can be avoided without changes to the source by creating a carefully designed Knowledge or Object View. As a result the classes used by the application can represent chunks of information of reasonable sizes.

Using the view concept for read requests is fairly easy and straightforward, while one has to be careful when designing the write access. This, however, is true not only for the Object Views or the Knowledge Views, it is true for any view that presents only partial information. This is why views in database systems often do not provide the write access—partial information can break consistency restrictions.

4.1.4 Knowledge engineer vs software engineer

Since the Knowledge Views are on the border between the "classic" computer systems and knowledge enabled ones, two roles can be distinguished, when using the Knowledge Views: a knowledge engineer and a software engineer. Knowledge engineers and software engineers are concerned with different aspects of the Knowledge Views, therefore they need to be considered separately.

Knowledge engineer

The knowledge engineer is responsible for creating the ontology to be used in an application, therefore in this case study this engineer is responsible for the work described in Section 4.1.1.

Simplicity One of the things that the knowledge engineer encounters most frequently while working with the Knowledge Views are the ABox and TBox interfaces (see Chapter 5), since most components implement one of those interfaces or both. These interfaces can be compared to the Model interface in Jena [52] and RDF2Go [92] or OWLOntology in OWL API [81]. The interfaces in Jena, RDF2Go and OWL API are designed to reflect RDF, RDFS or OWL and have plenty of methods (Model in Jena has over 100 methods). The interfaces proposed in the Knowledge Views (see Fig. 5.2), on the other hand, reflect description logic. Moreover they are minimalistic—there are as few methods as possible (ABox has 14 methods of which only 6 are mandatory and TBox has 13 methods), while trying not to fall into the Turing tar-pit [84]. This design follows the humorous sentence stated by Alan J. Perlis in [84]:

> Syntactic sugar causes cancer of the semi-colons.

There are two main goals behind choosing such a design. The first is the ease of learning—the fewer methods, the easier to learn. The second is expandability. To write new components, for example new adaptors for external data sources, only few methods need to be implemented. The run info ABox from the case study example (see Fig. 4.1) implements 6 mandatory methods of the ABox interface and only one of these methods is longer than a single line of code.

Modularity Another aspect of Knowledge Views' usage is modularity of the created ontology. Modularity can help when working on complex ontologies. In a monolithic ontology a single axiom can break the integrity of the whole ontology, however introducing modularity can restrict the influence of axioms, assertions and rules. When working with small parts instead of the ontology as a whole, it is also easier to find errors, moreover the idea of unit testing can be introduced to ontology creation process.

Ontology modularisation is still an open problem and there is much work concerning it. An approach to modularisation presented in [97] can be considered interesting, because it uses Distributed Description Logics (DDL) [11] to implement modularisation solution. DDL, on the other hand, was devised with information integration in mind rather than modularisation. This shows that information integration and modularisation are in fact related. The main difference between modularisation and integration is that in integration we have small independent entities that we want to combine, while in modularisation the process is reversed—we have some big and complex entity that we want to divide. Similarly in Knowledge Views modularisation is achieved by using tools devised for integration purposes and therefore modularisation is rather a side effect of the chosen approach rather than one of the goals.

Some other modularisation approaches include $\mathcal{E}$-Connections [43], SIM method [39, 38] or S-modules [40, 33, 35]. This last modularisation approach depends on an algebra that operates on fragments of knowledge called conglomerates. Most of the operators of this algebra can be relatively easily implemented using Knowledge Views. Unlike the mentioned modularisation solutions the Knowledge Views treat ontologies more as software modules than as entities described in terms of formal logic.

The example presented in this chapter demonstrates yet another advantage of modularity in Knowledge Views—the ability to mix different reasoning methods in a single ontology. In the example, apart from description logic and rule-based reasoning, there is some custom reasoning in the run info ABox. This ABox performs reasoning that is hard with the open world assumption. Expressing that small part of the ontology in a programming language can save time and effort.

Reuse of ontology components is also supported by modularity. In the example (see Fig. 4.1) the network ABox is a part of both the network ontology and the infection ABox. This can lead to time savings, because one well tested ontology component can be utilised many times.

Freedom of choice Freedom of choice is another advantage of the Knowledge Views. The freedom manifests in the ability to use different already existing tools in combination with the Knowledge Views. For example the knowledge engineer can use tools already implemented in the Knowledge Views library (see Section 5.1.1) to create an ontology in a similar way to the one described in this chapter, yet he can choose to use some third party knowledge base and just implement an adaptor, so it can work with the upper layers of the Knowledge Views (see Fig. 3.9). As a part of this project an adaptor for Jena has been created, but new adaptors can be created as well, for example for Sesame [95] or some other RDF storage system or knowledge base. Developers like to choose the tools by themselves and giving them such freedom can lead to products with higher quality and faster development process.

Software engineer

The software engineer receives a knowledge base prepared by the knowledge engineer. However, the software engineer does not use the description logic interface directly. Instead a view is created: an Object View for ease of use in

Listing 4.1: Sample OQL queries

```
SELECT c FROM Connection c WHERE c.source.ip = ?;
SELECT c FROM Connection c WHERE c.target.ip = ?;
SELECT c FROM Computer c;
```

an object-oriented programming language, Data View for compatibility with relational databases or some other views if needed.

In the case study application two Object Views were created. The creation of the views is the hardest part, since it is on the border of knowledge and software engineer's competencies. The knowledge engineer has to provide a Knowledge View that corresponds to the data schema required by the software engineer. However, once it is done the mappings between the models are straightforward. In the Object Views the mappings are defined as annotations (see Section 5.3 and Appendix E.2), just like in JPA [20]. Since annotating classes in order to map them to a data source is known in software engineering, this process should be intuitive and easy to learn by the software engineer. In the sample application there are 5 annotated classes. The process of creating annotated classes can also be automated by generating classes from the TBox provided by the knowledge engineer. The generated classes can be afterwards easily adjusted to better suit their purpose in the application.

Once the Object View is ready the knowledge-based layer is hidden from the software engineer. What the software engineer sees is an object store, which can be queried with OQL [15]. Sample queries from the application are in Listing 4.1. These queries are quite simple, however more complex queries are also possible, provided that the Object Views implement complete OQL. Apart from querying, the application persists new objects, merges changes into the object repository and removes objects. All these operations are done in a way similar to JPA, which should make it easy to learn.

4.2 Compatibility

The Knowledge Views are a layer between an information source and an application. Therefore when speaking about compatibility two kinds of compatibility can be considered: compatibility with various information sources and compatibility with existing applications. The Knowledge Views can play the role of a bridge between different models and different paradigms, for example they can combine a relational information source, ontological reasoning and object-oriented interface. This supports the idea of many models in a single application distributed among several components. Having a single model throughout the entire application seems attractive, however it is less flexible and more importantly it requires a model that is expressive enough to be able to handle different uses in different components. Moreover having a single model is more and more difficult in the era of heterogeneous systems.

4.2.1 Compatibility with information sources

Nowadays there are plenty of kinds of information sources: relational databases, object databases, XML databases, RDF stores, HTML files, various legacy formats, for example MARC bibliographic data [61], which is widely used and therefore hard to replace, and many more. It is often not possible to just rewrite all legacy data to some new format even if the format is superior in every respect. Therefore the ability to understand many formats can be useful.

In the Knowledge Views the adaptors play the role of translators of different formats to the form of unary and binary predicates used in the description logic. Many data formats can be translated to the predicate form. Relational databases encode data as n-ary predicates, which can be easily transformed to a set of unary and binary predicates. Therefore anything that can be stored in a relational database can also be used by the Knowledge Views. If a database table has a composite primary key, it is possible to create a surrogate key or generate some unique id based on the attributes belonging to the primary key.

The case study application is an example of usage of databases as the information source. Moreover there is an information source that is a piece of custom code, which proves how flexible this architecture is. In some experiments also an RDF file was used as the source. These three types of information sources have been chosen as examples, because databases are widely used in software engineering, RDF is used in knowledge engineering and custom code shows the power of the solution. As far as the Knowledge Views are concerned these sources are interchangeable, because they are wrapped by appropriate adaptors that implement the same interface.

4.2.2 Compatibility with applications

Compatibility with applications means providing various interfaces that are used by contemporary applications or conform to standards. The Knowledge Views provide several interfaces widely used in software engineering and are

Listing 4.2: OQL query returning people with a given surname

```
SELECT p FROM Person p WHERE p.surname = "SomeSurname"
```

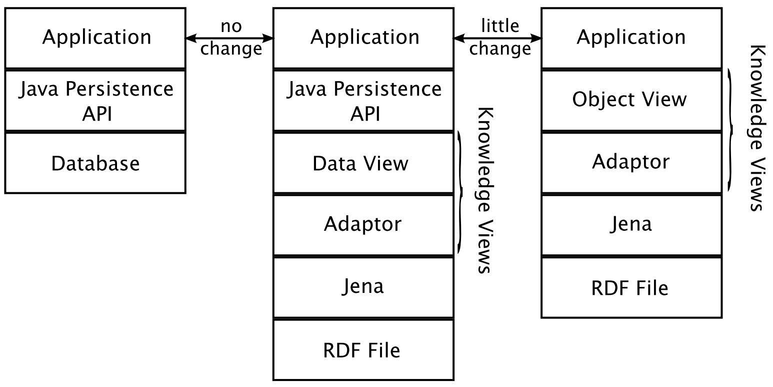


Figure 4.9: Three applications with different information sources

designed to allow the addition of new interfaces. This compatibility makes it easier to integrate already existing components with new semantic ones.

The case study application issues OQL queries that are compatible with Java Persistence Query Language, however the Object Views API (see Appendix E) differs from the JPA. This shows that the compatibility can be achieved on different levels. This can be compared to the application programming interface (API) and application binary interface (ABI) compatibility: an application can be compatible on the API level and still not compatible on the ABI level.

To better show the different levels of compatibility with existing components that can be achieved with help of the Knowledge Views three versions of the same application has been written. The application accesses Friend of a Friend (FOAF) [14] data and issues a query returning all people with some surname (see Listing 4.2).

The first application uses a relational database with the FOAF data rewritten from the source RDF file. The application does not use the JDBC API directly but uses the Java Persistence API (see Fig. 4.9 left).

The second application (see Fig. 4.9 middle) has exactly the same source code, however there is one difference. In the JPA configuration file the JDBC driver and URI had to be changed to point to another source. The URI points to a manifest that describes how to construct the view. The manifest is interpreted by the Data Views JDBC driver. In this example the Data View emulates the relational database used in the first application. This time, however, the original RDF file is used as the source with the Jena library as an RDF parser.

The third application (see Fig. 4.9 right) no longer uses the JPA, which is replaced with an Object View. The Object Views API (see Appendix E) is similar, but not identical to the JPA, therefore some changes to the application code were necessary. Nevertheless the query was kept intact. Moreover the Java bean class used as an entity bean by the JPA in the previous applications did not change on the API level but was only annotated differently. This means that if the application itself was further divided into a data access layer and a business layer, where data is passed between these layers through the entity bean objects, then the business layer would not change, only the data access layer would change. The benefit of loosing full compatibility in this example is slight performance gain, because there are fewer layers.

This example shows that it is in fact possible to use the Knowledge Views in an existing application with little or no change in the existing code. Thanks to injecting semantic components to an existing system, the system can gain some additional capabilities, like reasoning.

4.3 Gradual introduction of knowledge bases

Let us consider a situation, where some company gathers information about something as a side effect of the company's true business activity. The information can be incomplete therefore it is hard to use. Utilising the information is not mission critical, nevertheless taking advantage of the information could be beneficial. Examples of companies that conform to this description are providers of social network related services. People store lots of information in those services. Since very little information about a user is mandatory, the information is usually incomplete. Sometimes the information is used for marketing purposes, for example displaying one's first name on an advertisement banner. However, even more information that is useful can be retrieved from the information explicitly given by the users due to reasoning. Some simple examples will be presented to prove the point. No legal issues concerning privacy will be discussed here.

To sum up let us assume there is a social network service that wishes to enhance or provide new services based on the optional information users provide. The first step is to enhance the module that presents advertisements. At present the advertisements are shown more or less at random, which may annoy users who are shown advertisements not targeted at them, while users potentially interested in the advertised product might never see the advertisement. This way the advertisement might be ineffective.

4.3.1 The classic approach

First let us try to solve the problem the "classic" way, that is without use of knowledge bases. The first advertisement is for women cosmetics that we wish to show only to women. Nothing simpler than to query our relational database for the gender column in the users profile table before displaying the advertisement. However as mentioned before the column might be empty, since the information is not mandatory. Therefore if the column is empty, we issue another query to retrieve the first name, which might indicate the user's gender. If that is inconclusive, the next step might be checking surname, since in some languages, Polish included, it may be possible to differentiate gender by the surname suffix. As we see even in this very simple example we get several special cases that need to be considered and there might be even more cues in the profile pointing to the user's gender. Hard coding every query with some additional code for processing first name and surname is not a good idea since it would be hard to change or add new cases for handling more cues.

The next advertisement is for a casting that calls for women in their twenties. The code and queries for checking user's gender is ready and can be reused, but there is the need for some code checking user's age. Again the column containing the year of birth might be empty and some reasoning can be performed to fill the information. In this case there is no need for exact birth year—it is sufficient to assign the user to some age group category. Where can pointers to the user's age be found? Let us assume the social network service can store information about user's education. By checking in what years the user attended what kind of school the age can be inferred. For example if the user attended high school about ten years ago, we can assume that she is in her twenties. Another example might be that the user is currently a student and so on. With the addition of a new advertisement new reasoning code has to be added—this is unacceptable especially if there are many advertisement providers.

Another advertisement is for a newly opened restaurant in Gdańsk. The target group of this advertisement are people living in or often visiting the city. The users can declare the city in their profile, but again it is optional and can be inferred from other information. The information about education can be useful—if the user currently studies at some school in Gdańsk, then we can assume he or she often visits the city.

There is much useful information that can be inferred from information given by users, even if the information is partial. The examples given above are quite simple, but more complex examples are also easy to imagine. Using knowledge bases to handle reasoning allows clean design of the system. There

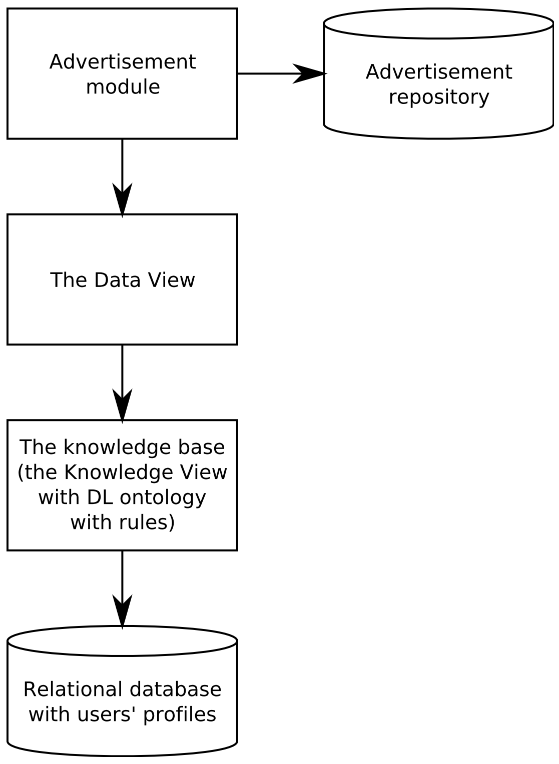

Advertisement module
Advertisement repository
The Data View
The knowledge base (the Knowledge View with DL ontology with rules)
Relational database with users' profiles

Figure 4.10: Architecture of the Knowledge Views approach

is no need for a programmer to handle numerous special cases—they are all taken care of by the reasoning engine. This improves maintainability. Moreover, terminology and rules can be checked by a reasoner for consistency—source code written in a Turing complete programming language cannot be checked in general.

4.3.2 The Knowledge Views approach

In the Knowledge Views approach the advertisement module would use two repositories (see Fig. 4.10): the advertisement repository and the knowledge base. The advertisement repository would contain pairs: an advertisement and an SQL query template returning one row if the advertisement is to be shown to the given user or an empty set if the advertisement is inappropriate. The advertisement queries would be sent to the knowledge base through the Data View. The Data View is there so that queries could be stated in SQL—a language widely known. Another advantage of having the Data View there is that if for some reason there is trouble with this solution, the knowledge base can be scraped and substituted with a regular database view on top of

Listing 4.3: SQL query for user's gender

```
SELECT * FROM User WHERE id = ? AND gender='W'
```

Listing 4.4: SQL query for young women

```
SELECT * FROM User WHERE id = ? AND ageGroup='20s'
  AND gender='W'
```

the profiles database. In such a case the system would still work, however the results would be poorer.

Let us consider the first advertisement that is targeted at women. The advertisement repository would contain a query checking the gender column (see Listing 4.3). The *gender* attribute in the knowledge base would be defined as an SQL query on the profiles database as well as rules that infer it from other information:

$$\begin{aligned} \mathit{gender}(x,z) &\leftarrow \mathit{isGenderProvided}(x,\text{false}) \wedge \mathit{firstName}(x,y) \\ &\quad \wedge \mathit{firstNameGender}(y,z) \\ \mathit{gender}(x,z) &\leftarrow \mathit{isGenderProvided}(x,\text{false}) \wedge \mathit{surname}(x,y) \\ &\quad \wedge \mathit{surnameGender}(y,z) \end{aligned} \tag{4.9}$$

In those rules the *isGenderProvided* predicate is mapped to an SQL query on the original database and plays the role of a guardian so that the information explicitly given have priority. The *firstName* and *surname* are mapped to queries, while *firstNameGender* and *surnameGender* are custom code predicates.

The second advertisement is targeted at women in their twenties (see Listing 4.4). In the knowledge base the *ageGroup* attribute would be defined as a query to the profile database and as the following rules:

$$\begin{aligned} \mathit{ageGroup}(u,\text{"20s"}) &\leftarrow \mathit{belongsTo}(u,c) \wedge \mathit{hasSchool}(c,s) \\ &\quad \wedge \mathit{HighSchool}(s) \wedge \mathit{hasBegining}(c,b) \\ &\quad \wedge \mathit{hasYear}(\text{today},y) \\ &\quad \wedge \mathit{built\text{-}in\text{:}subtract}(d,y,b) \\ &\quad \wedge \mathit{built\text{-}in\text{:}le}(5,d) \wedge \mathit{built\text{-}in\text{:}le}(d,15) \\ \mathit{ageGroup}(u,\text{"20s"}) &\leftarrow \mathit{belongsTo}(u,c) \wedge \mathit{hasSchool}(c,s) \\ &\quad \wedge \mathit{University}(s) \wedge \mathit{hasBegining}(c,b) \\ &\quad \wedge \mathit{hasYear}(\text{today},y) \\ &\quad \wedge \mathit{built\text{-}in\text{:}subtract}(d,y,b) \\ &\quad \wedge \mathit{built\text{-}in\text{:}le}(1,d) \wedge \mathit{built\text{-}in\text{:}le}(d,11) \\ &\vdots \end{aligned} \tag{4.10}$$

Listing 4.5: SQL query for people visiting Gdańsk

```
SELECT * FROM User WHERE id = ? AND visits='Gdansk'
```

The first rule means that if the user u belonged to a class c in school s which is a high school and that class started 5 to 15 years ago then the user is in her twenties. The second rule is for universities, therefore the year range is different.

The third advertisement is targeted at people connected with Gdańsk (see Listing 4.5). The *visits* attribute is defined as a query to the profile database and as a rule:

$$\begin{aligned} \mathit{visits}(u,p) \leftarrow\ & \mathit{belongsTo}(u,c) \wedge \mathit{hasSchool}(c,s) \wedge \mathit{hasLocation}(s,p) \\ & \wedge \mathit{hasBegining}(c,b) \wedge \mathit{hasEnd}(c,e) \\ & \wedge \mathit{hasYear}(\text{today},y) \wedge \mathit{built\text{-}in\text{:}le}(b,y) \\ & \wedge \mathit{built\text{-}in\text{:}le}(y,e) \end{aligned} \tag{4.11}$$

The rule states that the user u visits p if the user belongs to a class c in a school s located in p. The end of the rule ensures that the user currently belongs to the class and it is not historical information.

In the presented examples the addition of new advertisements requires the addition of new mappings and rules to the knowledge base. What is important only the knowledge base is affected, while the advertisement module is kept simple, since the reasoning is moved to the knowledge base. When there is a greater number of rules in the knowledge base there will be good chances that adding a new advertisement would require no change at all, because the knowledge base will already be able to answer the advertisement query.

4.3.3 Summary

Anything that the knowledge base can infer can also be inferred by writing custom code. So what are the advantages of using a knowledge base? The most important is maintainability achieved due to loose coupling. Changes can be made more easily without fear of breaking the existing code. Moreover there is a clear assignment of responsibilities to the modules. Another advantage is that queries to the knowledge base can be simpler than if they were sent directly to the database underneath.

The penalty of using the knowledge base is that it should be maintained by a knowledge engineer instead of a software engineer. However, thanks to the Knowledge Views the knowledge base can be easily queried by a regular software engineer without special knowledge of knowledge engineering.

This example shows that it is possible to gradually introduce a knowledge base to an existing system, while retaining a backup solution, which reduces the risk. Of course at first the knowledge base should be introduced in modules that are not mission critical, but when the technology turns out to perform well it can be included in other more important modules.

4.4 Ease of use experiments

Experiments have been conducted to test the ease of use of the proposed solution from the programmers point of view. The first experiment was conducted with seasoned software engineers—their remarks were taken into account in the subsequent version of the Knowledge Views library (see Chapter 5). The next experiment was conducted with computer science students—if inexperienced students are able to easily learn the proposed solution then experienced software developers should have no problems in learning it either.

4.4.1 Seasoned software engineers

An experiment was conducted on two software engineers to check how fast they can learn the Object View API (see Appendix E). Another important purpose of this experiment was to gather some insight from the engineers concerning the proposed API. The engineers were asked to do the same task using Object Views and Jena with SPARQL, so they could compare the proposed solution with an alternative.

First they were asked several questions to evaluate their knowledge. They had working knowledge of Java Persistence API. They said they did not know what OQL is, however they new Java Persistence Query Language (JPQL) or its superset Hibernate Query Language (HQL), which are to a high degree compatible with OQL. The engineers had heard of RDF, however never used it and did not know SPARQL. The engineers had no prior knowledge of any of the APIs used in the experiment.

Next the engineers were given instructions (see Appendix A) to write a program that reads and queries an RDF file containing real life FOAF [14] data. The file was downloaded from the Digital Enterprise Institute web site[1]. The instructions contained some pointers telling where to start. Moreover the participants received an archive with all necessary files.

Two versions of the application were to be written: first one using the Object Views and the second using Jena. This order of the tasks had been

[1]`http://www.deri.ie/`

Table 4.1: Ease of use experiment time results

Participant	Object Views	Jena
First	18 [min]	16 [min]
Second	26 [min]	21 [min]

Listing 4.6: OQL query for the ease of use experiment

```
SELECT p FROM Person p WHERE p.surname = "SomeSurname"
```

chosen deliberately to favour Jena—while doing the first task the participants had to get to know what are they to do, while during the second task it was just rewriting the application using another library. Table 4.1 contains times spent by each participant on each task. These times seem to favour Jena, however the differences are insignificant. Moreover there were several circumstances that caused the results to favour Jena. One of them is the already mentioned order of tasks, which could be eliminated only by gathering a large enough group of seasoned software engineers to be divided into two groups with different orders of tasks. Moreover there was a significant difference between these two tasks. Since the participants already knew OQL syntax (through the knowledge of JPQL and HQL), they were asked to write the query returning all people with a given surname by themselves (see Listing 4.6). In the second task, on the other hand, the SPARQL query was given (see Listing 4.7), because the participants did not know SPARQL and learning a new query language would require too much time for this experiment. As a result the participants had to write the Object View version of the application from scratch, while in case of Jena version they only had to look appropriate methods up in the library's Javadoc—the requirement of writing the query was deliberately omitted in the Jena version. This shows that relying on technologies and standards already known to the developers exempts them from learning new languages that consumes time.

An interesting thing to note is that the SPARQL query is much longer than the equivalent OQL query (compare Listings 4.6 and 4.7). The main reason for it is that some of the information that needs to be explicitly given in the SPARQL query is taken from the class definition in OQL. This information is reused between different OQL queries that reference the same class. In SPARQL this is repeated in every query. This is the consequence of RDF being a low level format in comparison with objects.

After performing the tasks the participants were asked which API they preferred and why. They unanimously pointed out the Object Views. The

Listing 4.7: Ease of use experiment SPARQL query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?uri ?title ?firstName ?surname ?phone
WHERE {
  ?uri foaf:title ?title ;
       foaf:firstName ?firstName ;
       foaf:surname ?surname ;
       foaf:phone ?phone
  FILTER (?surname = "SomeSurname")
}
```

Listing 4.8: Ease of use experiment Object View code

```
JenaKB jkb = new JenaKB(inputStream, false);
ObjectViewManager ovm = new ObjectViewManager(
    jkb, Person.class);
Collection<Person> results = ovm.executeQuery(
    "SELECT p FROM Person p" +
    " WHERE p.surname = \"SomeSurname\"",
    Person.class);
```

main reason was that the code using Object Views was shorter (compare Listings 4.8 and 4.9) and that there was less mechanical work. However it was pointed out by the participants that having a generator for the annotated classes needed by the Object Views would make the task considerably easier. The generator was actually ready by the time of the experiment, however it was deliberately made unavailable for the task to force the participants to annotate the Person class by themselves—this also extended the time required for the Object Views task. The participants also stated that having code samples would make the task of using Object Views even easier. This was also not provided on purpose (only Javadoc documentation for Object Views was available), because the task would be finished in just a few minutes by copying and pasting and this would not guarantee that the participants actually understand what they are doing.

Moreover the participants pointed out some details in Object Views that break the conventions they were used to. First of all Object Views required setters to be annotated not getters like in JPA (this has been corrected after this experiment). Another remark was that they were used to `'` as the character string delimiter in query languages instead of `"`, because it is eas-

Listing 4.9: Jena code for the ease of use experiment

```
Model model = ModelFactory.createOntologyModel();
model.read(inputStream, "");
QueryExecution queryExecution = QueryExecutionFactory
    .create("PREFIX␣foaf:␣<http://xmlns.com/foaf/0.1/>"
    + "␣SELECT␣?uri␣?title␣?firstName␣?surname␣?phone"
    + "␣WHERE␣{"
    + "␣?uri␣foaf:title␣?title␣;"
    + "␣foaf:firstName␣?firstName␣;"
    + "␣foaf:surname␣?surname␣;"
    + "␣foaf:phone␣?phone"
    + "␣FILTER␣(?surname␣=␣\"Kruk\")"
    + "␣}", model);
ResultSet result = queryExecution.execSelect();
Collection<Person> results = new ArrayList<Person>();
while (result.hasNext()) {
  QuerySolution solution = result.nextSolution();
  Person p = new Person(
      solution.get("?uri").toString(),
      solution.get("?surname").toString(),
      solution.get("?firstName").toString(),
      solution.get("?title").toString(),
      solution.get("?phone").toString());
  results.add(p);
}
```

ier to embed it in a programming language where " is the delimiter (both delimiters are permitted in the recent Object Views version). These remarks clearly show that software engineers are accustomed to some conventions and it is important for them that these conventions are followed to the smallest detail.

This experiment shows that the Object Views library is easy enough to be learnt by a seasoned software engineer in less than half an hour, even without additional tools and sample code. This is partly due to conforming to known conventions and reusing a known query language. Moreover the code using the Object Views is more compact than the code using RDF handling library Jena, which operates on a lower level and which was designed with different goals in mind.

4.4.2 Students

Another experiment was conducted on computer science students to check whether inexperienced students were able to quickly start using the Knowledge Views—if students can quickly learn the proposed solution, then seasoned programmers should have no problems either. Furthermore the students had been selected based on their academic curricula to check the significance of basing the proposed solution on known technologies, that is JDBC and JPA.

The experiment was conducted with five groups of students. Three of the groups had had classes on JEE including JDBC and JPA. The other two groups had no or very little prior experience with JEE. Most of the students had no experience with semantic technologies and knowledge bases. The students had 1.5 hour to get to know the Data Views and Object Views libraries and query a knowledge base using them (see Appendix B). Before starting the tasks students were asked to fill a questionnaire, in which they declared their knowledge of related technologies. Those declarations often diverged from the real skills of the students as observed during the experiment, therefore the skills where judged based on the specialisations of the groups and their curricula.

This experiment showed (see Table 4.2) that all of the students after JEE course managed to finish the first task (Data Views task) and most of them finished the second task (Object Views task). The students without JEE experience had more problems with both tasks—only about a half of them finished the first task and one fourth finished the second task. These statistics clearly show that if new technologies are based on standards already known to the target group, those technologies are easier to learn and use.

Another conclusion from this experiment is that if the proposed solution

Table 4.2: Ease of use experiment with students

Student group	**Student count**	**Data View task**	**Data View task %**	**Object View task**	**Object View task %**
after JEE course	24	24	100.0	21	87.5
without JEE experience	24	13	54.2	6	25.0

is easy enough for students to learn within 1.5 hour, the solution should also be easy enough to be learnt by seasoned programmers that have much more experience with the technologies Data Views and Object Views are based on. Moreover the lack of knowledge base related skills on the part of the students did not influence finishing the tasks. This means that the Knowledge Views successfully hid the semantic technologies and provided that the knowledge base is prepared by a knowledge engineer, it can be used by software engineers without additional training.

Chapter 5

The Knowledge Views implementation

The Knowledge Views as a whole are designed to be lightweight and extensible. Being lightweight allows the implementation to be embedded in various system types including stand alone applications as well as client-server applications. Extensibility helps in adding new functions, but also helps substituting the existing implementations of particular components with better ones, for example to improve performance. The implementation of Knowledge Views is divided into several main modules: the core of the Knowledge Views and the individual views providing model mapping capabilities. The implementation is a proof of concept rather than production quality software, however it provides firm foundations for future work.

5.1 Knowledge Views core

The core of the Knowledge Views is designed according to the Unix philosophy [94]:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

The core of Knowledge Views is a set of tools, often consisting of a single class, each performing only one function like: ABox merging or reasoning. The tools can work together, for example two ABoxes can be merged and then reasoned over as a single ABox or two ABoxes could be used in reasoning separately and then joined together yielding a different result. However, instead of handling text, most of the tools handle ABoxes and provide the ABox interface so a pipeline similar to Unix pipeline can be created.

5.1.1 The Knowledge Views tools

The tools created as a part of the thesis include:

- ABoxView—defines a new ABox based on a source ABox and transformation rules (no reasoning, implements the ABox interface);
- InferencingKnowledgeView—performs reasoning on a source ABox based on a given TBox and rules creating a knowledge base (implements the ABox and TBox interfaces);
- KnowledgeView—combines an ABox with a TBox creating a knowledge base (no reasoning, implements the ABox and TBox interfaces);
- MemoryABox—an ABox that stores assertions in memory (implements the ABox interface);
- MergedABox—combines several ABoxes into one (implements the ABox interface);
- TBoxView—creates a TBox that defines new concepts, roles and attributes based on a given TBox (implements the TBox interface).

Figure. 5.1 presents the UML diagram of these tools. The diagram shows the dependencies of the classes: which interfaces each class implements and which interfaces each class uses. A more detailed view of the ABox, TBox and KnowledgeBase interfaces is shown in Fig. 5.2. This is a minimal set of tools that was required to built a functioning prototype. However more tools can be added as needed and used in conjunction with the existing ones.

Additionally two adaptors were created:

- DatabaseKB—maps SQL queries to assertion sets (implements the ABox interface),
- JenaKB—translates the Jena API to the one used in the Knowledge Views (implements the ABox and TBox interfaces).

Some of the above tools implement the TBox interface. This interface plays two roles: it is used in reasoning by InferencingKnowledgeView and it defines the data schema. The second role is important for the Object Views and Data Views, where classes and the database schema can be automatically generated from a TBox. In particular information about inheritance, domains and ranges of roles and attributes as well as whether roles and attributes are functional is used.

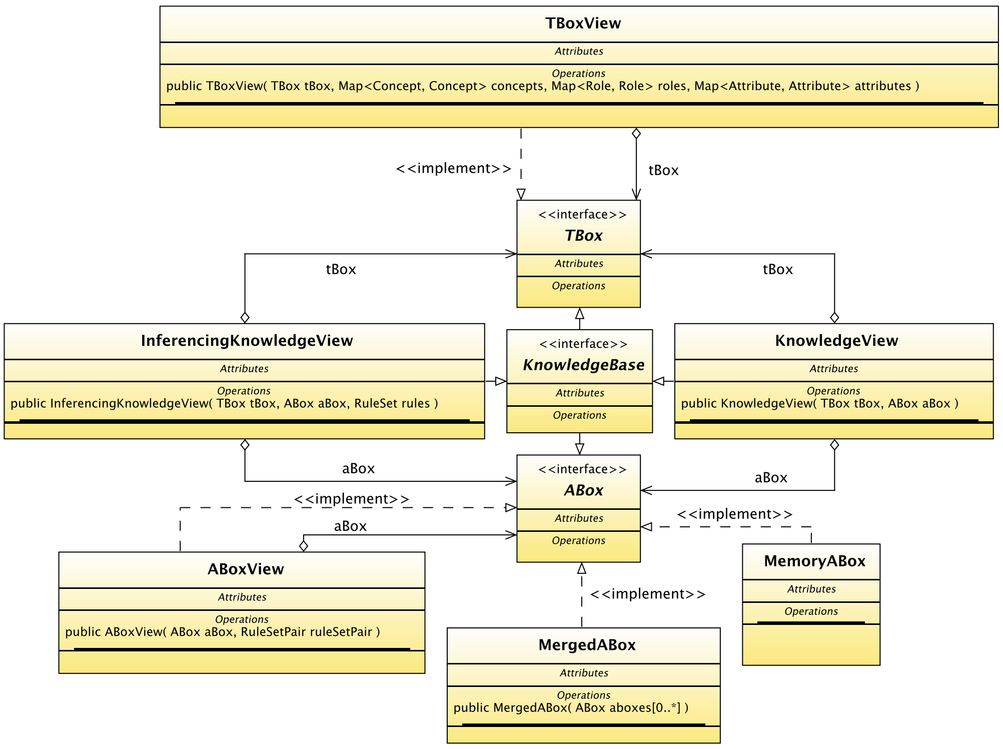


Figure 5.1: The UML diagram of the Knowledge Views tools

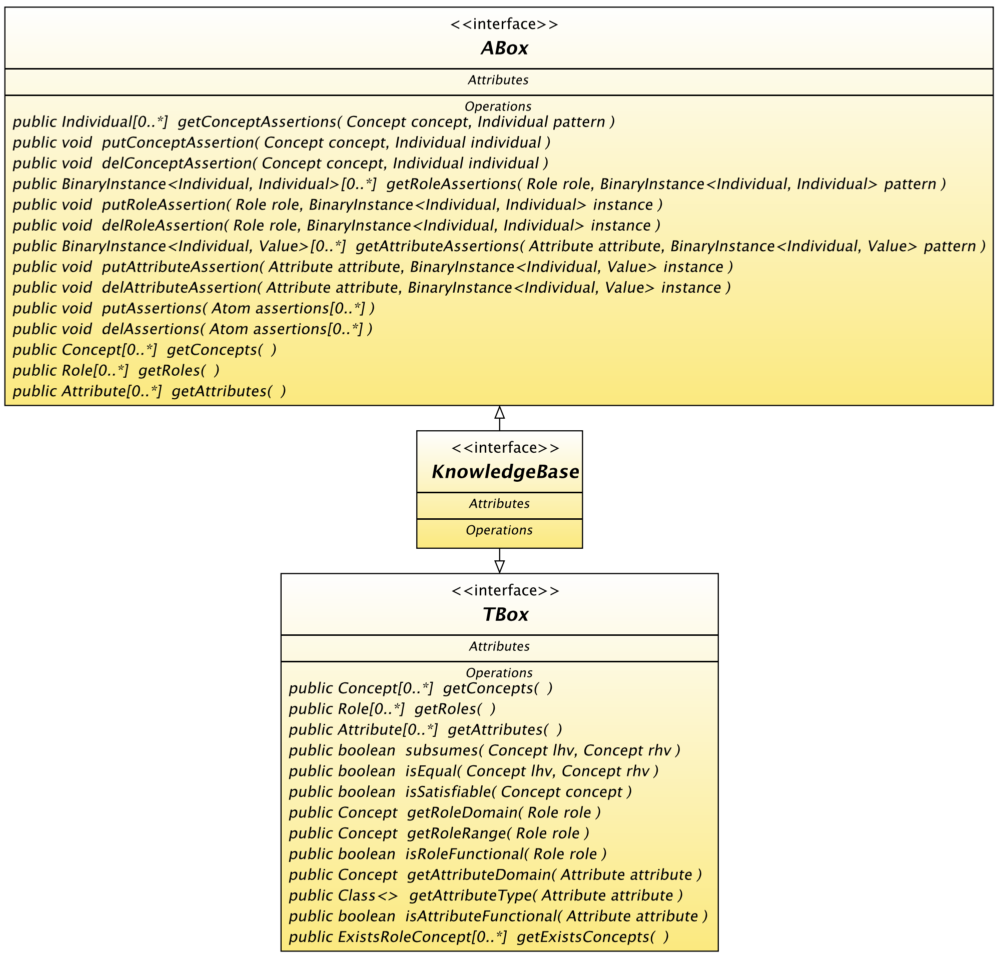

<<interface>>
ABox
Attributes
Operations
public Individual[0..*] getConceptAssertions(Concept concept, Individual pattern)
public void putConceptAssertion(Concept concept, Individual individual)
public void delConceptAssertion(Concept concept, Individual individual)
public BinaryInstance<Individual, Individual>[0..*] getRoleAssertions(Role role, BinaryInstance<Individual, Individual> pattern)
public void putRoleAssertion(Role role, BinaryInstance<Individual, Individual> instance)
public void delRoleAssertion(Role role, BinaryInstance<Individual, Individual> instance)
public BinaryInstance<Individual, Value>[0..*] getAttributeAssertions(Attribute attribute, BinaryInstance<Individual, Value> pattern)
public void putAttributeAssertion(Attribute attribute, BinaryInstance<Individual, Value> instance)
public void delAttributeAssertion(Attribute attribute, BinaryInstance<Individual, Value> instance)
public void putAssertions(Atom assertions[0..*])
public void delAssertions(Atom assertions[0..*])
public Concept[0..*] getConcepts( )
public Role[0..*] getRoles( )
public Attribute[0..*] getAttributes( )
<<interface>>
KnowledgeBase
Attributes
Operations
<<interface>>
TBox
Attributes
Operations
public Concept[0..*] getConcepts( )
public Role[0..*] getRoles( )
public Attribute[0..*] getAttributes( )
public boolean subsumes(Concept lhv, Concept rhv)
public boolean isEqual(Concept lhv, Concept rhv)
public boolean isSatisfiable(Concept concept)
public Concept getRoleDomain(Role role)
public Concept getRoleRange(Role role)
public boolean isRoleFunctional(Role role)
public Concept getAttributeDomain(Attribute attribute)
public Class<> getAttributeType(Attribute attribute)
public boolean isAttributeFunctional(Attribute attribute)
public ExistsRoleConcept[0..*] getExistsConcepts( )

Figure 5.2: The UML diagram for the Knowledge Views interfaces

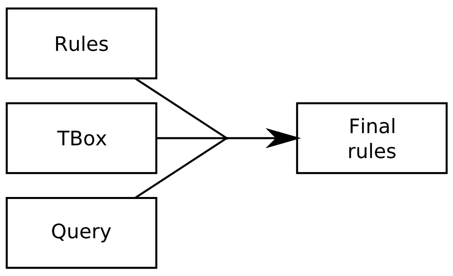


Figure 5.3: Query rewriting

5.1.2 Reasoning

The InferencingKnowledgeView performs reasoning, which is important when creating a view that changes terminology. Without reasoning an assertion set provided by the view may be incomplete. However, in some situations reasoning in a view may be undesirable. For example if a knowledge engineer knows that reasoning will not add any new assertions, reasoning will only introduce an unnecessary performance penalty. For such cases the KnowledgeView is more suitable.

The current implementation of the InferencingKnowledgeView is not complete, but provides some basic reasoning capabilities that were sufficient to present the Knowledge Views prototype potential. The reasoning is based on rules with some DL additions. As a future work the implementation will be extended towards the SWRL expressivity.

The reasoning implementation in the InferencingKnowledgeView is similar to the query rewriting approach employed in QuOnto [90] and KL [103]. A user provided rule set, a TBox and a query are used to create a new rule set (see Fig. 5.3). The resulting rule set is then used to answer the original query. In the Knowledge Views the query rewriting is done in two steps (see Fig. 5.4): first an intermediate rule set is derived from the initial rule set and the TBox, then the final rule set is created that takes into account the query. The first step is independent from the query therefore can be done once during initialisation of the InferencingKnowledgeView. For simple queries the second step might add no additional rules to the rule set.

The Knowledge Views differ from the QuOnto and KL in that the InferencingKnowledgeView does not require the information sources to provide the SQL interface. The InferencingKnowledgeView uses a datalog-based query language and an external TBox reasoner. It is sufficient for the information sources to provide the information as sets of assertions—no high level query language like SQL is required. This approach can give poorer performance results, but is more general and supports heterogeneous distributed

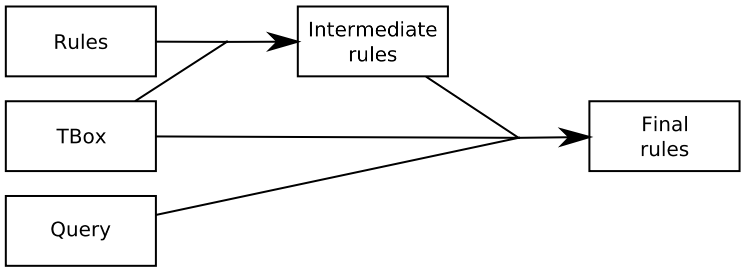


Figure 5.4: Query rewriting in Knowledge Views

information sources. Moreover rules support is already included, since the implementation is based on rules.

Rules used in the reasoning process can contain all the constructs that are available for the model transformation rules (see Section 3.3.1), in particular the built-ins (see Appendix C for more details on available built-ins). This causes the reasoning to be undecidable. This implementation of reasoning does not protect from creating undecidable ontologies, which can lead to infinite loops. On the contrary it adds additional capabilities by allowing mixing the DL and rule-based reasoning with custom code (see Section 4.1.1). This approach allows creation of ontologies that are even more expressive than SWRL ontologies, but at the cost of loosing decidability. It is up to the knowledge engineer to make sure that the knowledge base will not enter an infinite loop. The knowledge engineer can choose not to use the additional potentially dangerous capabilities and have guarantee of decidability, but the capabilities are there and may be exploited if needed. If the knowledge engineer chooses to use the extended expressivity, then the modularisation capabilities of the Knowledge Views can help isolate risk posing parts to help testing and maintaining the whole ontology. This approach is similar to programming where Turing complete languages are commonly used, but programmers must face the halting problem [96]. This can be compared to a hammer with which one can hurt oneself, but having a safe hammer that can cause no harm is useless, because it would be impossible to hammer a nail with it. An opposite approach is proposed in [44], where the Description Logic Programs narrow expressivity to obtain interoperability of logic programs.

5.1.3 Ontology merging

Another important function implemented by the Knowledge Views tool set is ontology merging. Here, ontology merging is understood as combining information from various ontologies to create a new ontology. In this implementation each source ontology is treated as an ABox. The source ABoxes can be transformed using the ABoxView to comply to a common terminology. Next the resulting ABoxes are merged using a set-theory union operation implemented by the MergedABox. As the last step the merged ABox is used by the InferencingKnowledgeView together with a TBox representing the terminology of the new ontology.

Such and example shows how the tools can cooperate to produce complex results. Even with just a few tools there is a number of possibilities due to ability to use them in different configurations.

5.1.4 eXtended NeeK language (xNeeK)

xNeeK is a language that gives possibility to define a Knowledge View in a declarative way. This has several advantages. First of all it is a knowledge engineer that defines a Knowledge View and not every knowledge engineer has to be fluent in programming languages, therefore allowing a knowledge engineer to define a view with an XML configuration file should make the job easier—knowledge engineers know XML, since they use XML-based languages like RDF, OWL, SWRL, etc. The configuration file, called a **manifest**, is passed to a software engineer. The manifest defines the knowledge source, which is wrapped by a Data View or an Object View—depending on the software engineer's decision. Such a view is then used directly by the business logic in an application. In a way the xNeeK language draws a line between responsibilities of the two engineers.

The second advantage of xNeeK is that the manifest can be easily changed without need of recompilation. Unless the data schema exposed by the view defined in the manifest changes, there is no need for a software engineer intervention if a knowledge engineer changes something in the knowledge source.

Another advantage is the ease of storing an XML manifest. It can be embedded within an application or it can reside as a configuration file in an applications directory. Multiple manifests with different views used by an application can be easily managed.

Listing 5.1 shows a sample xNeeK manifest that defines an Inferencing-KnowledgeView with three arguments: a TBox (sampleTBox.owl), an ABox (sampleABox.owl) and some rule set. This example shows that views can be nested in other views—this gives possibility of creating complex views. Comparing the Knowledge Views API (see Appendix D) with this example can reveal that in general xNeeK provides a way to define a sequence of class constructor calls outside the programming language, which, as stated before, can be beneficial to the knowledge engineer.

Listing 5.2 shows the rule set that is missing in Listing 5.1. The rule set

Listing 5.1: Sample xNeeK manifest

```
<?xml version="1.0" encoding="UTF-8"?>

<neek:view xmlns:neek='http://kio.pg.gda.pl/neek'
    type="pl.gda.pg.km.kv.InferencingKnowledgeView">

  <neek:view type="pl.gda.pg.km.kv.adapters.JenaKB">
    <neek:str>http://a.b.c/sampleTBox.owl</neek:str>
    <neek:bool>false</neek:bool>
  </neek:view>
  <neek:view type="pl.gda.pg.km.kv.adapters.JenaKB">
    <neek:str>http://a.b.c/sampleABox.owl</neek:str>
    <neek:bool>false</neek:bool>
  </neek:view>
  <neek:ruleSet>
    ...
  </neek:ruleSet>
</neek:view>
```

consists of a single rule:

$$\begin{aligned} \mathit{Student}(u) \leftarrow\ & \mathit{belongsTo}(u, c) \wedge \mathit{hasSchool}(c, s) \wedge \mathit{University}(s) \\ & \wedge \mathit{hasBegining}(c, b) \wedge \mathit{hasEnd}(c, e) \\ & \wedge \mathit{hasYear}(\text{today}, y) \wedge \mathit{built\text{-}in{:}le}(b, y) \\ & \wedge \mathit{built\text{-}in{:}le}(y, e) \end{aligned} \tag{5.1}$$

The rule states that if someone belongs to a class that has a school that is a university and today is between the starting and ending year of the class, then that someone is a student. The embedded rule language is similar to RuleML [93], which is embedded in SWRL. Using RuleML had been considered, but a decision to create a new simple rule language has been made. In the future, however, it is possible that RuleML will be used in xNeeK. Similarly to SWRL xNeeK allows built-ins which can perform arithmetic operations, string operations, etc. In this example a number comparison built-in *built-in:le* is used—the predicate is true if the first argument is less than or equal to the second argument.

The xNeeK language is named after its predecessor: the NeeK language [30]. After long evolution xNeeK has very little in common with its ancestor. The original NeeK language was designed for Data Views and gave little possibility of customisation—a view was directly derived from a terminology and

Listing 5.2: Sample xNeeK rules

```
<neek:ruleSet>
  <neek:concept>
    <neek:head>
      <neek:atomicConcept name="&p;Student"/>
      <neek:individual isVariable="true" name="u"/>
    </neek:head>
    <neek:roleAtom>
      <neek:atomicRole name="&p;belongsTo"/>
      <neek:subject isVariable="true" name="u"/>
      <neek:object isVariable="true" name="c"/>
    </neek:roleAtom>
    <neek:roleAtom>
      <neek:atomicRole name="&p;hasSchool"/>
      <neek:subject isVariable="true" name="c"/>
      <neek:object isVariable="true" name="s"/>
    </neek:roleAtom>
    <neek:conceptAtom>
      <neek:atomicConcept name="&p;University"/>
      <neek:individual isVariable="true" name="s"/>
    </neek:conceptAtom>
    <neek:attributeAtom>
      <neek:atomicAttribute name="&p;hasBegining"/>
      <neek:subject isVariable="true" name="c"/>
      <neek:object isVariable="true" value="b"/>
    </neek:attributeAtom>
    <neek:attributeAtom>
      <neek:atomicAttribute name="&p;hasEnd"/>
      <neek:subject isVariable="true" name="c"/>
      <neek:object isVariable="true" value="e"/>
    </neek:attributeAtom>
    <neek:attributeAtom>
      <neek:atomicAttribute name="&p;hasYear"/>
      <neek:subject isVariable="false"
          name="&p;today"/>
      <neek:object isVariable="true" value="y"/>
    </neek:attributeAtom>
     ...
  </neek:concept>
</neek:ruleSet>
```

the user could only select which entities are to be included in the view and which excluded from the view. xNeeK gives much more possibilities to the user. It not only allows selecting what is of interest, but also provides means to transform knowledge sources, which was very early found to be necessary.

The full xNeeK specification can be found in Appendix C. The specification is developed in sync with the Knowledge Views implementation—additional DL constructs will be added together with their support in the Knowledge Views. This way the user is not confused by some constructs present in the documentation that at present cannot be used.

5.2 Data Views

The Data Views provide a way to access a knowledge base via the JDBC API. There are two ways to create a Data View. Moreover there are two types of views: materialised and non-materialised, which are described in Section 5.2.3.

5.2.1 DataView class

The main class of the Data Views is the DataView class. This is the class that changes the Knowledge Views API into the JDBC API. To start using this class one has to provide:

- a knowledge base or
- an ABox together with mappings.

In the first case the Data Views create a database schema based on the terminology stored in the knowledge base according to the rules enumerated in Section 3.2.1. In the second case an ABox is given with mappings instead of terminology. Here the database schema is given by the user as opposed to being generated automatically. The user enumerates the tables and columns with matching concepts, roles and attributes. This gives possibility of selecting only a part of knowledge from the underlying knowledge base. Moreover it allows the user to name the tables and columns. If more adjustments to the terminology are needed, transformation capabilities of the Knowledge Views in the lower layer can be used.

Once a DataView object is created, a JDBC connection can be retrieved from it (see Listing 5.3). Since the connection is standard compliant, it can be used in any way a connection retrieved from the java.sql.DriverManager can be used. With this connection the underlying knowledge base is seen as a standard relational database.

Listing 5.3: Data Views usage with DataView class

```
KnowledgeBase knowledgeBase = new JenaKB(
    new FileInputStream("/dev/shm/test.owl"), false);
DataView dataView = new DataView(knowledgeBase);
Connection connection = dataView.getDVConnection();
Statement statement = connection.createStatement();
ResultSet resultSet = statement
    .executeQuery("SELECT␣*␣FROM␣OperatingSystem");

while (resultSet.next()) {
  System.out.println(resultSet.getString(1));
}

resultSet.close();
statement.close();
connection.close();
```

5.2.2 DriverManager

The other way to retrieve a JDBC connection for a knowledge base is by using the java.sql.DriverManager—that is in the usual way a connection for a database is retrieved in Java. This way Data Views can seamlessly cooperate with other standard technologies like for example JPA. To retrieve a JDBC connection to a knowledge base in this manner the user has to provide the java.sql.DriverManager with a proper URL (see Listing 5.4). In the example shown in Listing 5.4, the Data View is created on top of an OWL file. The database schema is created based on the terminology found in the file. Virtual database tables contain the ABox information from the file. To access a Data View on top of a complex Knowledge View an URL to an xNeeK manifest has to be provided instead of an OWL file. The JDBC URL pointing to an xNeeK manifest has the following form:

```
jdbc:dv:neek:http://server.domain/xNeeKManifest.xml
```

or

```
jdbc:dv:neek:file:///someDirectory/xNeeKManifest.xml
```

5.2.3 Materialised vs. non-materialised views

Similarly as it is in database systems, in Data Views there are two kinds of views: materialised and non-materialised. Materialised views were imple-

Listing 5.4: Data Views usage with DriverManager

```
Connection connection = DriverManager
     .getConnection("jdbc:dv:file:/dev/shm/test.owl");
Statement statement = connection.createStatement();
ResultSet resultSet = statement
     .executeQuery("SELECT␣*␣FROM␣OperatingSystem");

while (resultSet.next()) {
  System.out.println(resultSet.getString(1));
}

resultSet.close();
statement.close();
connection.close();
```

mented first. They are implemented as a function that copies the required information from the knowledge base to a database using a provided connection. The materialised views have some advantages over the non-materialised ones:

- indexes can be build on any column in the view,
- all the reasoning needed to derive facts in the view is performed only once,
- some advanced features of the target database can be used, for example building database views on top of the materialised Data View.

The materialised views can provide much shorter response times for queries than non-materialised views, because all the required information is already cached in the database and indexes can be built. However the materialised views are read-only and need to be refreshed periodically, possibly containing stale data between two refreshes. Moreover the refresh operation can take considerable amount of time, because potentially large amount of data must be copied between the knowledge base and the database.

Non-materialised Data Views can respond to queries more slowly, but the data is never stale, which is very important for certain applications. Moreover these views can support the write operation. Currently the implementation of non-materialised Data Views uses H2 RDBMS [46], in particular its linked table functionality and ability to create an alias for a function that acts like a database table. This implementation is rather a proof of concept. A full implementation of the JDBC API for the non-materialised Data Views is planned as a future work.

5.3 Object Views

The Object Views provide object-oriented access to a knowledge base. The access point to the Object Views functionality is the ObjectViewManager class (see Appendix E). To instantiate this class one has to provide an ABox and a set of class objects. These class objects constitute the terminology. They can be generated from an existing terminology by the ObjectViewGenerator tool included in the Object Views or they can be hand written. The classes need to adhere to the rules enumerated in Section 3.2.2.

The mapping rules in Section 3.2.2 state that concepts are mapped to interfaces to handle multiple inheritance. Interfaces cannot be instantiated, but classes can, therefore the ObjectViewManager works with classes only, not interfaces. This is not contradictory to the mapping rules, because an instance of a class that implements an interface is also an instance of that interface. In general every concept of interest is mapped to an interface, but for each interface there is a class that implements it (see Fig. 5.5). Since the ObjectViewManager does not use the interfaces they can be abandoned if the user does not need them. The ObjectViewGenerator generates both interfaces and classes.

The classes passed to the ObjectViewManager's constructor are Plain Old Java Object (POJO) classes. They are loosely coupled with the Object Views library through annotations, therefore they can be freely passed and reused in different parts of the client application—even serialised and sent over the network. The annotations in these classes define the mapping of concepts, roles and attributes to classes and fields. There are four types of annotations:

- Concept—annotates a class indicating which concept is mapped to it,
- Uri—annotates the getter of a field in which the URI of an individual is stored,
- Role—annotates the getter of a field indicating which role is mapped to it,
- Attribute—annotates the getter of a field indicating which attribute is mapped to it.

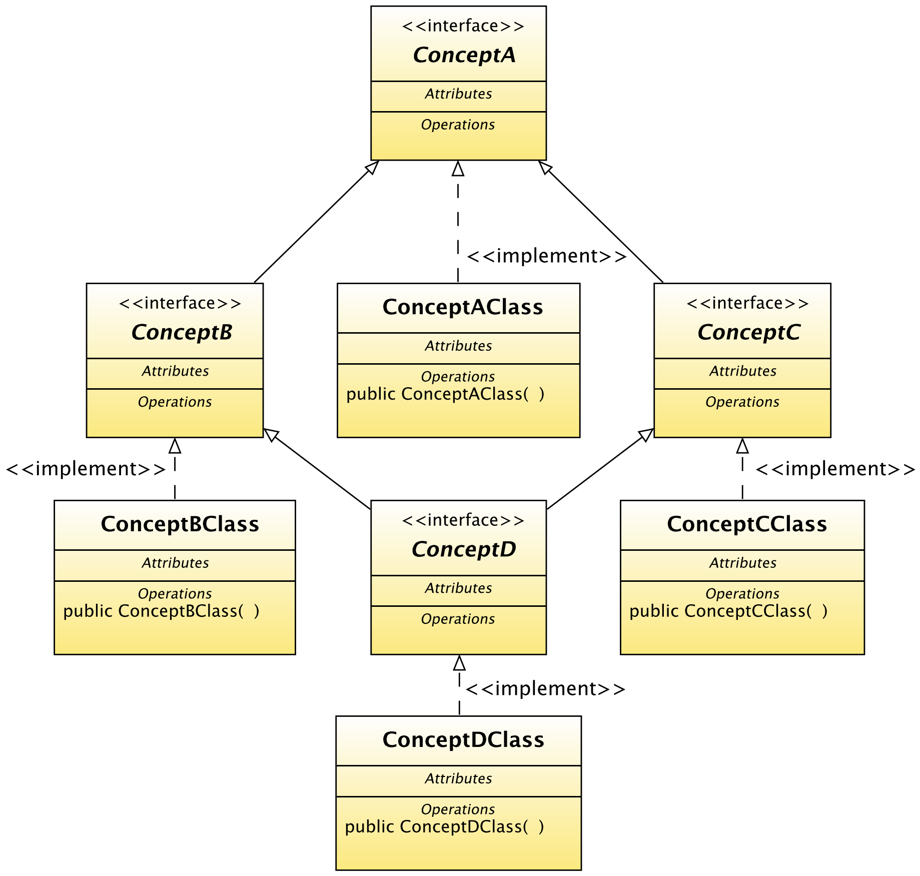


Figure 5.5: Diamond inheritance in Object Views

See Appendix E.2 for more details

Listings 5.5 and 5.6 show two sample annotated classes with their interfaces. The WineClass and WineryClass are marked with the Concept annotation with the names of the corresponding concepts. Another mandatory annotation is the Uri annotation in both classes—this denotes the field which stores the identity of the individual. Each field denoting a role or an attribute is marked with an appropriate Role or Attribute annotation—maker and year in the WineClass class. The WineryClass class is an example of a minimal class with no Role or Attribute annotation.

To instantiate an ObjectViewManager object one needs to pass a set of annotated classes to its constructor. The constructor gathers information about the ontology-object mapping from the annotations using the Java reflection API. If a class has a role field then the class being the range of the role is processed recursively. The query answering is performed by the ObjectViewManager in two stages. First the query is processed to give a set of individuals fulfilling the conditions from the query. In the second stage an object is created for every individual. For every annotated field in the object a query is issued to the underlying knowledge base and the result is assigned to the field. If the field represents a role the process is repeated recursively.

This implementation uses two features of the Java language: annotations and reflection. However, unlike ActiveRDF (see Section 2.4.4), the concept of Knowledge Views does not rely on language specific features. Annotations provide a convenient way to add metadata to Java classes, however before introduction of annotations to Java language the same goal had been achieved by storing metadata in a separate file. For example in J2EE metadata had been stored in separate XML files, called descriptors. The second feature of Java language used in Knowledge Views implementation, reflection, is supported by many modern languages, therefore a similar implementation can be created for other programming languages as well. It is even possible to create a similar object-oriented interface to a knowledge base in some languages that do not support reflection. For example in C++ the ontology-object mapping could be encoded using the C++ language feature called pointers to members.

5.4 RDF Views

The RDF Views library provides possibility to export assertions from an ABox to an RDF file and issue SPARQL queries. Encoding an ABox in the RDF format is straightforward, since RDF was designed for storing simple assertions. For SPARQL query processing ARQ library has been used. ARQ

Listing 5.5: Annotated Wine class

```
public interface Wine {
    public String getUri();
    public void setUri(String uri);
    public Collection<Winery> getMaker();
    public void setMaker(Collection<Winery> maker);
    public int getYear();
    public void setYear(int year);
}
@Concept("Wine")
public class WineClass implements Wine {
    private String uri;
    @Uri public String getUri() {
        return uri;
    }
    public void setUri(String uri) {
        this.uri = uri;
    }
    private Collection<Winery> maker;
    @Role(uri="maker", range=WineryClass.class)
    public Collection<Winery> getMaker() {
        return maker;
    }
    public void setMaker(Collection<Winery> maker) {
        this.maker = maker;
    }
    private int year;
    @Attribute("year") public int getYear() {
        return year;
    }
    public void setYear(int year) {
        this.year = year;
    }
}
```

Listing 5.6: Annotated Winery class

```
public interface Winery {
    public String getUri();
    public void setUri(String uri);
}
@Concept("Winery")
public class WineryClass implements Winery {
    private String uri;
    @Uri public String getUri() {
        return uri;
    }
    public void setUri(String uri) {
        this.uri = uri;
    }
}
```

is a part of the Jena library [52].

The RDF Views ignore TBox completely, even though OWL specification [83] defines how to encode OWL terminology as RDF. This is because a TBox in the Knowledge Views is not an OWL terminology. It is a component that can respond to subsumption queries—for Knowledge Views it is irrelevant what knowledge representation is used internally, for example it can be some executable computer code. Such encapsulation of TBox has several benefits which are well known from object-oriented programming. The drawback is that it is not easy to reverse engineer the complete terminology using only the provided TBox interface. However, it is possible to create a view that would masquerade the TBox as an ABox providing at least some information about the terminology, for example subsumption of atomic concepts. Such ABox can be wrapped by the RDF Views.

The RDF Views have been implemented, because RDF is the basic format used in Semantic Web and supporting RDF and SPARQL makes the Knowledge Views interoperable with many existing Semantic Web tools. From perspective of application business logic the RDF Views are the least convenient, because RDF is very low level. It focuses on single assertions, while business logic in an application usually operates on higher level entities like objects in object-oriented programming.

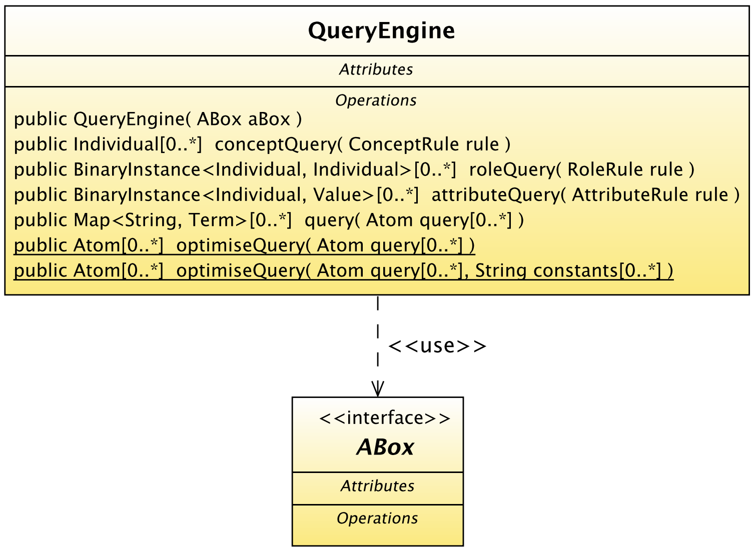

QueryEngine
Attributes
Operations
public QueryEngine(ABox aBox)
public Individual[0..*] conceptQuery(ConceptRule rule)
public BinaryInstance<Individual, Individual>[0..*] roleQuery(RoleRule rule)
public BinaryInstance<Individual, Value>[0..*] attributeQuery(AttributeRule rule)
public Map<String, Term>[0..*] query(Atom query[0..*])
public Atom[0..*] optimiseQuery(Atom query[0..*])
public Atom[0..*] optimiseQuery(Atom query[0..*], String constants[0..*])
<<use>>
<<interface>>
ABox
Attributes
Operations

Figure 5.6: QueryEngine class diagram

5.5 DL Views

Following the naming convention of the previous views, the DL Views provide the description logic ontology model. This is already done by the core Knowledge Views with the ABox and the TBox interfaces. Therefore the core Knowledge Views can be considered an implementation of the DL Views. The core Knowledge Views also provide a rule-based query language that is used in the model transformation. This query language is a part of xNeeK language (see Section 5.1.4 and Appendix C). It is an ABox query language only—to query the terminology the TBox interface has to be used directly. The QueryEngine class (see Fig. 5.6) exports the API that allows the users to use the query language.

For the DL Views to be truly on a par with the rest of the views a standard or at least a widely accepted DL query language has to be implemented. DIG [22, 6] is a DL query language that could become the query language of the DL Views. Implementing a standard compliant query language processor for the DL Views that would replace the current rule-based ABox query language is among the tasks for future work.

Chapter 6

Summary

The main contribution of this thesis is the concept of the Knowledge Views defined and elaborated upon in Chapter 3. Since Knowledge Views depend on two processes: model mapping and model transformation, those processes were also defined in Chapter 3. The general idea of model mapping was presented. Moreover rules that realise ontology-relational mapping and ontology-object mapping were defined. Tightly coupled with the model mapping problem query language mapping was discussed and an adopted solution presented. The general idea of the second process the Knowledge Views depend on, model transformation, was presented. Moreover a rule-based ontology transformation language that is used in Knowledge Views was defined. Chapter 3 ends with a presentation of the general architecture of Knowledge Views. The two most important layers of the proposed architecture correspond to the ideas of model mapping and model transformation defined earlier.

In Chapter 4 case studies were presented that show the ease of interfacing Knowledge Views with existing technology, in particular with relational databases and programs written in object-oriented language—Java in this case. Additionally experiments with both seasoned and beginner software engineers were conducted. Results of the experiments are presented in Chapter 4. Experiments with seasoned software engineers proved that they are accustomed to certain conventions known from existing technologies—they noticed even small differences between the initial version of Knowledge Views library and what they are used to. Those differences were removed in the subsequent versions of Knowledge Views library to further enhance compatibility with existing technologies. Experiments with beginner software engineers proved the ease of use of Knowledge Views—software engineering students where able to start using the proposed library within a couple of hours.

The Knowledge Views concept was realised as the Knowledge Views Java library. The implementation is described in Chapter 5 and Appendices D through F. The implementation is divided in several parts: Knowledge Views core, Data Views, Object Views, RDF Views and DL Views. Knowledge Views core is mainly responsible for model transformation. Transformations can be expressed using xNeeK defined in Chapter 5 and Appendix C. Data Views realise ontology-relational mapping and Object Views realise ontology-object mapping. RDF Views provide compatibility with RDF model and DL Views with description logic.

In that way, the goals stated in Chapter 1 have been achieved, that is:

1. Mappings of the ontology model to object model, relational model and RDF have been defined and implemented as Object Views, Data Views and RDF Views.

2. Two sets of transformations for ontology model have been defined. Transformations of a single model are encoded as a rule-based language, while the abilities of combining several models are realised as a set of tools in the Knowledge Views library. Both sets of transformations can be expressed in the xNeeK language.

The idea of the Knowledge Views has been realised, implemented and validated. Case studies have been conducted and experiments carried out to prove the thesis proposition that the Knowledge Views allow for easy usage of knowledge bases in contemporary information systems, by providing compatibility layer with existing systems and an easy to use API for developers.

Initial implementations of Data Views and Object Views and therefore partial results of this thesis were utilised in the PIPS (Personalised Information Platform for Life and Heath Services [89]) project funded by the European Commission under the Framework 6 call. The Data Views were used to augment query answering capabilities of the knowledge base used as the core of knowledge management system (KMS) [41] in the project. In particular some queries that required closed world assumption were processed through the Data Views. The Object Views constituted a layer between the knowledge management system and a web portal. Its role was to increase the ease of use of the knowledge base.

To recapitulate, the main achievements of this thesis are:

- Definition of the Knowledge Views concept and design of extensible Knowledge Views architecture,

- Definition of ontology-relational and ontology-object mappings,

- Definition of a rule-based transformation language and its implementation as xNeeK language,

- Implementation of the Knowledge Views concept as Knowledge Views, Data Views, Object Views and RDF Views libraries,
- Validation of the Knowledge Views concept by experiments performed with seasoned and beginner software developers.

6.1 Pros and cons of the proposed solution

Apart from such features of the proposed solution as the ease of use that where among the priorities, there are also some other notable advantages. Generality is one of them. Generality of this solution can be understood in several ways. First of all the methods described here can be applied when implementing similar libraries for object-oriented languages other than Java. Generality is also applied to information sources, that is the Knowledge Views can take advantage of various information sources, not only those providing SQL or SPARQL interface. Generality is closely related to extensibility. The Knowledge Views can be easily extended to provide other views if needed, but also the set of available transformations can be further extended in the future.

The mentioned advantages of generality and extensibility are coupled with performance penalties. Usually there is a trade-off between those features and performance. Optimisation focuses on particular cases improving their performance, but maintaining too many special cases can lead to coding by exception anti-pattern. As Alan J. Perlis stated in his epigrams [84]:

> Optimization hinders evolution.

The sole idea of adding an additional layer on top of a knowledge base is bound to have negative effect on performance of querying the knowledge base. However, if the performance is not satisfactory, a number of techniques can be applied that are already in use in various implementations of Java Persistence API or similar libraries.

6.2 Future work

There is much work that can still be done around the Knowledge Views concept. One of tasks is to change the existing prototype into a production quality product. To be suitable for uses in multi-user concurrent environments the Knowledge Views require addition of transaction handling, which is still a new concept in knowledge bases. The expansion of the supported expressivity of terminologies, which requires improvements in reasoning capabilities of the Knowledge Views, is also an important task. The set of operations on ABoxes and TBoxes can be extended to support handling of multiple sources containing inconsistent information. Such capabilities seem a must in the world of the Internet, where there are many sources providing contradictory information. Another task for future consideration is the inclusion of support for n-ary predicates in the Knowledge Views. This last task could lead to some improvements in performance. As it can be seen the work already done is a core that gives firm grounding for future expansion. The Knowledge Views were designed from the beginning with expandability in mind.

List of Figures

List of Tables

List of Listings

Bibliography

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, September 2007. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf` (accessed 6.04.2010).

[2] Andersen, L. *JDBC 4.0 Specification.* Sun Microsystems, November 2006.

[3] Anderson, J. Q., and Rainie, L. The Fate of the Semantic Web. Tech. rep., Pew Research Center's Internet & American Life Project, May 2010.

[4] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., Eds. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, September 2007.

[5] Bąk, J., and Jędrzejek, C. Wnioskowanie hybrydowe w relacyjnej bazie danych wykorzystujące podejście semantyczne. In *Bazy danych. Rozwój metod i technologii. Architektura, metody formalne i zaawansowana analiza danych*, S. Kozielski, B. Małysiak, P. Kasprowski, and D. Mrozek, Eds. Wydawnictwo Komunikacji i Łączności, 2008, pp. 333–348.

[6] Bechhofer, S. *DIG 2.0: The DIG Description Logic Interface.* DIG Working Group, September 2006. `http://dig.cs.manchester.ac.uk/index.html` (accessed 15.10.2009).

[7] Bell, A. E. Death by UML Fever. *Queue 2*, 1 (2004), 72–80.

[8] Bell, A. E. UML Fever: Diagnosis and Recovery. *Queue 3*, 2 (2005), 48–56.

[9] Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J. *XML Path Language (XPath) 2.0.* W3C, January 2007. `http://www.w3.org/TR/xpath20/` (accessed 2.04.2010).

[10] Booth, D., and Liu, C. K. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer.* W3C, June 2007. `http://www.w3.org/TR/wsdl20-primer/` (accessed 2.04.2010).

[11] Borgida, A., and Serafini, L. Distributed Description Logics: Assimilating Information from Peer Sources. In *Journal on Data Semantics*, vol. 2800 of *Lecture Notes in Computer Science.* Springer Berlin/Heidelberg, 2003, pp. 153–184.

[12] Borst, W. N. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse.* PhD thesis, Universiteit Twente, September 1997.

[13] Brickley, D., and Guha, R. *RDF Vocabulary Description Language 1.0: RDF Schema.* W3C, February 2004. `http://www.w3.org/TR/rdf-schema/` (accessed 6.04.2010).

[14] Brickley, D., and Miller, L. *FOAF Vocabulary Specification 0.91*, November 2007. `http://xmlns.com/foaf/spec/` (accessed 27.07.2009).

[15] Cattell, R. G. G., and Barry, D. K., Eds. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[16] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J. D., and Widom, J. The TSIMMIS Project: Integration of heterogeneous information sources. In *Proceedings of the 16th Meeting of the Information Processing Society of Japan* (1994).

[17] Chen, P. P.-S. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems 1*, 1 (1976), 9–36.

[18] Clark, K. G., Feigenbaum, L., and Torres, E. *SPARQL Protocol for RDF.* W3C, January 2008. `http://www.w3.org/TR/rdf-sparql-protocol/` (accessed 6.03.2010).

[19] Cooper, J. W. *Java Design Patterns: A Tutorial.* Addison-Wesley Professional, February 2000.

[20] DeMichiel, L., and Keith, M. Java Persistence API. In *JSR 220: Enterprise JavaBeans, Version 3.0.* May 2006.

[21] Dijkstra, E. W. The humble programmer. *Communications of the ACM 15*, 10 (October 1972), 859–866.

[22] DL Implementation Group (DIG). `http://dl.kr.org/dig/index.html` (accessed 23.06.2008).

[23] Duerst, M., and Suignard, M. *Internationalized Resource Identifiers (IRIs).* The Internet Society, 2005. `http://tools.ietf.org/html/rfc3987` (accessed 27.03.2010).

[24] Elliott, B., Cheng, E., Thomas-Ogbuji, C., and Ozsoyoglu, Z. M. A complete translation from SPARQL into efficient SQL. In *The 2009 International Database Engineering and Applications Symposium* (2009), ACM, pp. 31–42.

[25] Falkowski, M., and Jędrzejek, C. An efficient SQL-based querying method to RDF schemata. In *TPD 2007: II Krajowa Konferencja Naukowa Technologie Przetwarzania Danych*, T. Morzy, M. Gorawski, and R. Wrembel, Eds. Wydawnictwo Politechniki Poznańskiej, 2007, pp. 162–173.

[26] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, 2000.

[27] Fowler, M. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[28] Friedman, M., Levy, A., and Millstein, T. Navigational plans for data integration. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence* (Menlo Park, CA, USA, 1999), American Association for Artificial Intelligence, pp. 67–73.

[29] Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database Systems: The Complete Book.* Prentice Hall, June 2008.

[30] Goczyła, K., Grabowska, T., Waloszek, W., and Zawadzki, M. Designing World Closures for Knowledge-based System Engineering. In *Software engineering: evolution and emerging technologies*, K. Zieliński and T. Szuc, Eds. IOS Press, 2005, pp. 271–282.

[31] Goczyła, K., and Piotrowski, P. Application of Knowledge Views. *Studia Informatica 31*, 2A (89) (2010), 77–88.

[32] Goczyła, K., and Piotrowski, P. Introduction of knowledge bases to existing systems using the Knowledge Views. *Zeszyty naukowe Wydziału Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej 19* (2010), 61–66.

[33] Goczyła, K., Piotrowski, P., Waloszek, A., Waloszek, W., and Zawadzka, T. Terminological and assertional queries in KQL knowledge access language. Accepted for publication on ICCCI 2010.

[34] Goczyła, K., Piotrowski, P., Waloszek, A., Waloszek, W., and Zawadzka, T. Język KQL jako realizacja idei języka SQL dla bazy wiedzy. *Studia Informatica 31*, 2A (89) (2010), 47–61.

[35] Goczyła, K., Piotrowski, P., Waloszek, A., Waloszek, W., and Zawadzka, T. KQL – język dostępu do konglomeratowych baz wiedzy. In *TPD 2010: III Krajowa Konferencja Naukowa Technologie Przetwarzania Danych* (2010), pp. 126–137.

[36] Goczyła, K., Piotrowski, P., Waloszek, A., Waloszek, W., and Zawadzka, T. KQL as application of SQL rationale for knowledge bases. *Zeszyty naukowe Wydziału Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej 18* (2010), 69–74.

[37] Goczyła, K., Piotrowski, P., Waloszek, A., Waloszek, W., Zawadzka, T., and Zawadzki, M. Zarządzanie wiedzą ontologiczną w środowisku semantycznego internetu. In *Inżynieria oprogramowania – od teorii do praktyki.* Wydawnictwa Komunikacji i Łączności, 2008.

[38] Goczyła, K., Waloszek, A., and Waloszek, W. Hierarchiczny podział przestrzeni ontologii na konteksty. In *Bazy danych. Nowe technologie – Architektura, metody formalne i zaawansowany analiza danych*, S. Kozielski, B. Małysiak, P. Kasprowski, and D. Mrozek, Eds. Wydawnictwo Komunikacji i Łączności, 2007, pp. 247–260.

[39] Goczyła, K., Waloszek, W., and Waloszek, A. Contextualization of a DL Knowledge Base. In *Proceedings of the 20th International Workshop on Description Logics* (2007). `http://ceur-ws.org/Vol-250/paper_55.pdf` (accessed 13.08.2010).

[40] Goczyła, K., Waloszek, W., and Waloszek, A. A Semantic Algebra for Modularized Description Logics Knowledge Bases. In *Proceedings of the 22nd International Workshop on Description Logics* (2009). `http://ceur-ws.org/Vol-477/paper_67.pdf` (accessed 13.08.2010).

[41] Goczyła, K., Waloszek, W., Zawadzka, T., and Zawadzki, M. Inference mechanisms for knowledge management system in e-health environment. In *Software engineering: evolution and emerging technologies* (2005), IOS Press, pp. 418–423.

[42] Goczyła, K., Zawadzka, T., and Zawadzki, M. Wnioskowanie z danych zapisanych w zewnętrznych źródłach w systemie zarządzania wiedzą. In *Bazy danych. Nowe technologie – Architektura, metody formalne i zaawansowana analiza danych*, S. Kozielski, B. Małysiak, P. Kasprowski, and D. Mrozek, Eds. Wydawnictwo Komunikacji i Łączności, 2007, pp. 283–293.

[43] Grau, B. C., Parsia, B., and Sirin, E. Combining OWL ontologies using E-Connections. *Web Semantics: Science, Services and Agents on the World Wide Web 4*, 1 (2006), 40–59.

[44] Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. Description Logic Programs: Combining Logic Programs with Description Logic. In *In Proc. of the Twelfth International World Wide Web Conference* (2003).

[45] Gruber, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition 5*, 2 (1993), 199–220.

[46] H2 Database Engine. `http://www.h2database.com` (accessed 15.09.2009).

[47] Hayes, P. *RDF Semantics.* W3C, February 2004. `http://www.w3.org/TR/rdf-mt/` (accessed 23.06.2008).

[48] Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* W3C, May 2004. `http://www.w3.org/Submission/SWRL/` (accessed 19.06.2008).

[49] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*, 1990. IEEE Std 610.12-1990.

[50] Jarrar, M., and Meersman, R. Ontology Engineering – The DOGMA Approach. In *Advances in Web Semantics I*. Springer, 2009, pp. 7–34.

[51] Jastor – Typesafe, Ontology Driven RDF Access from Java. `http://jastor.sourceforge.net/` (accessed 19.06.2008).

[52] Jena – A Semantic Web Framework for Java. `http://jena.sourceforge.net/` (accessed 19.06.2008).

[53] Josuttis, N. M. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, 2007.

[54] The JXPath Component. `http://commons.apache.org/jxpath/` (accessed 19.03.2010).

[55] Kalyanpur, A., Pastor, D. J., Battle, S., and Padget, J. Automatic Mapping of OWL Ontologies into Java. In *Proceedings of Software Engineering and Knowledge Engineering* (Banff, Canada, 6 2004).

[56] Klyne, G., and Carroll, J. J. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C, February 2004. `http://www.w3.org/TR/rdf-concepts/` (accessed 23.06.2008).

[57] Kruk, S. R., Cygan, M., Piotrowski, P., Samp, K., and Westerski, A. Building a heterogeneous network of digital libraries on the Semantic Web. In *Semantics Systems From Visions to Applications: Proceedings of the Semantics 2006* (2006), pp. 31–36.

[58] Leszczyna, R., Fovino, I. N., and Masera, M. Simulating malware with MAlSim. *Journal in Computer Virology* (July 2008).

[59] Levy, A. Y. Answering queries using views: A survey. *The VLDB Journal 10*, 4 (December 2001), 270–294.

[60] Levy, A. Y., Rajaraman, A., and Ordille, J. J. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Data Bases* (1996), T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds., Morgan Kaufmann, pp. 251–262.

[61] MARC Standards. `http://www.loc.gov/marc/` (accessed 24.10.2008).

[62] Manola, F., and Miller, E. *RDF Primer.* W3C, February 2004. `http://www.w3.org/TR/rdf-primer/` (accessed 23.06.2008).

[63] Marshall, C. C., and Shipman, F. M. Which semantic web? In *HYPERTEXT '03: Proceedings of the fourteenth ACM conference on Hypertext and hypermedia* (New York, NY, USA, 2003), ACM, pp. 57–66.

[64] Martín, L., Anguita, A., Maojo, V., Bonsma, E., Bucur, A. I. D., Vrijnsen, J., Brochhausen, M., Cocos, C., Stenzhorn, H., Tsiknakis, M., Doerr, M., and Kondylakis, H. Ontology Based Integration of Distributed and Heterogeneous Data Sources in ACGT. In *HEALTHINF (1)* (2008), L. Azevedo and A. R. Londral, Eds., INSTICC - Institute for Systems and Technologies of Information, Control and Communication, pp. 301–306.

[65] Masolo, C., Borgo, S., Gangemi, A., Guarino, N., and Oltramari, A. WonderWeb Deliverable D18: Ontology Library. Tech. rep., Laboratory For Applied Ontology, 2003.

[66] McGuinness, D. L., and van Harmelen, F. *OWL Web Ontology Language Overview.* W3C, February 2004. `http://www.w3.org/TR/owl-features/` (accessed 27.03.2010).

[67] Meyer, B. UML: The Positive Spin. *Cutter IT Journal (formerly American Programmer) X*, 3 (1997). `http://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html` (accessed 25.03.2010).

[68] Microsoft. The LINQ Project. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx` (accessed 25.03.2009).

[69] Mitra, N., and Lafon, Y. *SOAP Version 1.2 Part 0: Primer (Second Edition).* W3C, April 2007. `http://www.w3.org/TR/soap12-part0/` (accessed 2.04.2010).

[70] Motik, B., Patel-Schneider, P. F., and Parsia, B. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax.* W3C, October 2009. `http://www.w3.org/TR/owl-syntax/` (accessed 27.03.2010).

[71] Murray, C. *Semantic Technologies Developer's Guide, 11g Release 2 (11.2).* Oracle Corporation, March 2010. `http://download.oracle.`

com/docs/cd/E11882_01/appdev.112/e11828/title.htm (accessed 23.04.2010).

[72] National Center for Biomedical Ontology. NCBO Bioportal. http://bioportal.bioontology.org/ (accessed 6.04.2010).

[73] Object Management Group. *Meta Object Facility (MOF) Specification Version 1.4.1*, July 2005. http://www.omg.org/spec/MOF/ISO/19502/PDF/ (accessed 22.03.2009).

[74] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1: CORBA Interfaces*, 2008.

[75] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 2: CORBA Interoperability*, 2008.

[76] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 3: CORBA Component Model*, 2008.

[77] Object Management Group. *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.2*, 2009.

[78] Object Management Group. *OMG Unified Modeling LanguageTM (OMG UML), Superstructure Version 2.2*, 2009.

[79] Oren, E. *Algorithms and Components for Application Development on the Semantic Web.* PhD thesis, National University of Ireland, January 2008.

[80] Oren, E., Delbru, R., Gerke, S., Haller, A., and Decker, S. ActiveRDF: Object-oriented semantic web programming. In *Proceedings of the International World-Wide Web Conference* (Banff, Canada, 5 2007).

[81] The OWL API. http://owlapi.sourceforge.net/ (accessed 14.07.2009).

[82] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (1972), 1053–1058.

[83] Patel-Schneider, P. F., Hayes, P., and Horrocks, I. *OWL Web Ontology Language Semantics and Abstract Syntax.* W3C, February 2004. `http://www.w3.org/TR/owl-semantics/` (accessed 15.10.2009).

[84] Perlis, A. J. Special Feature: Epigrams on programming. *SIGPLAN Not. 17*, 9 (1982), 7–13.

[85] Piotrowski, P. Implementacja widoków danych na bazę wiedzy. In *TPD 2007: II Krajowa Konferencja Naukowa Technologie przetwarzania danych* (2007), T. Morzy, M. Gorawski, and R. Wrembel, Eds., Wydawnictwo Politechniki Poznańskiej, pp. 174–185.

[86] Piotrowski, P. Knowledge base views. In *Proceeedings of the 1st International Conference on Information Technology* (2008), pp. 35–38.

[87] Piotrowski, P. Object Views – metoda mapowania obiektowo-ontologicznego. In *Bazy Danych. Rozwój metod i technologii – Bezpieczeństwo, wybrane technologie i zastosowania.* Wydawnictwa Komunikacji i Łączności, 2008, pp. 207–216.

[88] Piotrowski, P. The Internet as a knowledge source in Knowledge Views. *Studia Informatica 30*, 2A (83) (2009), 201–211.

[89] Personalised Information Platform for Life and Heath Services. `http://www.ist-world.org/ProjectDetails.aspx?ProjectId=f07c448095e6427d86c87f3e05e8075e` (accessed 23.04.2010).

[90] Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Linking Data to Ontologies. *J. on Data Semantics X* (2008), 133–173.

[91] Prud'hommeaux, E., and Seaborne, A. *SPARQL Query Language for RDF.* W3C, January 2008. `http://www.w3.org/TR/rdf-sparql-query/` (accessed 19.06.2008).

[92] RDF2Go. `http://rdf2go.semweb4j.org/` (accessed 14.07.2009).

[93] The Rule Markup Initiative. `http://ruleml.org/` (accessed 12.10.2009).

[94] Salus, P. H. *A Quarter Century of UNIX.* Addison-Wesley Professional, June 1994.

[95] Sesame. `http://www.openrdf.org/` (accessed 19.06.2008).

[96] Sipser, M. *Wprowadzenie do teorii obliczeń*. Wydawnictwa Naukowo-Techniczne, 2009.

[97] Stuckenschmidt, H. Implementing Modular Ontologies with Distributed Description Logics. In *Proceedings of the first international workshop on modular ontologies at the International Semantic Web Conference ISWC'06* (2006). `http://ki.informatik.uni-mannheim.de/fileadmin/publication/stuckenschmidt07implementing.pdf` (accessed 13.08.2010).

[98] Sun Microsystems. *Java SE 6 Documentation*. `http://java.sun.com/javase/6/docs`.

[99] Ullman, J. D. Information Integration Using Logical Views. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory* (London, UK, 1997), Springer-Verlag, pp. 19–40.

[100] W3C Semantic Web Activity. `http://www.w3.org/2001/sw/` (accessed 20.06.2008).

[101] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview*. W3C, October 2009. `http://www.w3.org/TR/owl-overview/` (accessed 27.03.2010).

[102] Xu, L., and Embley, D. W. Combining the Best of Global-as-View and Local-as-View for Data Integration. In *Information Systems Technologies and Its Applications* (2004), A. E. Doroshenko, T. A. Halpin, S. W. Liddle, and H. C. Mayr, Eds., vol. 48 of *LNI*, GI, pp. 123–136.

[103] Zawadzka, T. *Integracja heterogenicznych źródeł wiedzy z wykorzystaniem logiki opisowej*. PhD thesis, Gdańsk University of Technology, 2008.

Appendix A

Ease of use experiment 1

A.1 Description

In this exercise the goal is to create an application that reads an RDF file containing FOAF information, searches the information for a particular person or a group of people and prints the information found. The application is to be created by extending a given template. The template includes a method that prints the information found and a class Person in which information about a single person is to be stored. The Person class corresponds to the `http://xmlns.com/foaf/0.1/Person` concept (OWL class). The fields of the class correspond to the following FOAF entities:

- uri – URI of the particular `Person` instance,
- surname – `http://xmlns.com/foaf/0.1/surname` attribute (OWL datatype property),
- firstName – `http://xmlns.com/foaf/0.1/firstName` attribute (OWL datatype property),
- title – `http://xmlns.com/foaf/0.1/title` attribute (OWL datatype property),
- phone – `http://xmlns.com/foaf/0.1/phone` attribute (OWL datatype property)[1].

The participant's task is to write Java code that reads the RDF file, queries it and passes the results to the method that prints them. The task is to be done twice: first using the Object View API, then using the Jena API.

[1]According to the FOAF specification this should be a role (OWL object property), however the test file containing real life data uses it as an attribute (OWL datatype property)

A.2 Object View API

1. Read the provided RDF file using pl.gda.pg.km.kv.adapters.JenaKB from the Knowledge View API. Do not use the Jena API (com.hp.hpl.jena package).
2. Use pl.gda.pg.km.ov.ObjectViewManager from the Object View API for querying.
3. Annotate the Person class.
4. Issue OQL query that finds people by surname.

A.3 Jena API

1. Read the provided RDF file using com.hp.hpl.jena.rdf.model.ModelFactory and Model from the Jena API[2].
2. Use com.hp.hpl.jena.query.QueryExecutionFactory and QueryExecution from the ARQ API[3] for querying.
3. Issue SPARQL query that finds people by surname.

[2]`http://jena.sourceforge.net/javadoc/index.html`
[3]`http://jena.sourceforge.net/ARQ/javadoc/index.html`

Appendix B

Ease of use experiment 2

Questionnaire

1. Do you know[1] RDF, RDFS, OWL, DIG or other ontology definition language and which?...

2. Do you know SPARQL, DIG or other ontology query language and which?

   ....

3. Do you know SQL?...

4. Do you know JDBC?...

5. Do you know Java Persistence API (JPA)?...........................

6. Do you know Object Query Language (OQL), Java Persistence Query Language (JPQL) or HQL (Hibernate Query Language) and which?

   ....

7. What time is it?...

   Now start doing the tasks—instructions are on the next page.

8. What time is it (after doing the tasks)?.............................

9. Have you successfully finished the Data Views task? ....

10. Have you successfully finished the Object Views task?................

[1]"Know" refers to working knowledge, that is can you use the technologies

11. Any comments (for example what was the hardest part)? . . . .
. . . .
. . . .
. . . .

B.1 Description

The task is to write two Java applications. The first one queries a knowledge base using the JDBC interface (the Data Views). The second application queries the same knowledge base using an object interface (the Object Views). The goal of both applications is to list all people with surname "Piotrowski" and print the following information: surname, first name, title, phone. The manifest describing the knowledge base to be queried can be found at:

- `http://knot221.eti.pg.gda.pl/piotr/kv/e4/manifest.xml`.

Required libraries can be downloaded from

- `http://knot221.eti.pg.gda.pl/piotr/kv/e4/lib.zip`.

Documentation needed for the task can be found at:

- `http://knot221.eti.pg.gda.pl/piotr/kv/e4/KnowledgeView/javadoc/` – the Knowledge Views,
- `http://knot221.eti.pg.gda.pl/piotr/kv/e4/DataView/javadoc/` – the Data Views,
- `http://knot221.eti.pg.gda.pl/piotr/kv/e4/ObjectView/javadoc/` – the Object Views,
- `http://knot221.eti.pg.gda.pl/piotr/kv/e4/HOWTO.xhtml` – *how to* with some tips.

The tasks can be done in any order!

B.2 Data Views

1. Connect to the knowledge base using the DriverManager and a properly constructed JDBC URL pointing to the knowledge base manifest.
2. Issue a query listing all people with surname "Piotrowski" using SQL.
3. Print the required information about the returned people.

B.3 Object Views

1. Generate classes for the knowledge base from the manifest.
2. Create a knowledge base object from the manifest.
3. Create an Object View for this knowledge base.
4. Issue a query listing all people with surname "Piotrowski" using OQL (Java Persistence Query Language has the same syntax for simple queries as OQL).
5. Print the required information about the returned people.

Appendix C

xNeeK specification

C.1 Notation convention

- <tag/> – an XML tag,
- <tag1/> <tag2/> – a sequence of tags,
- (<tag1/> <tag2/>) – grouping of tags,
- <tag/>* – a sequence of any number of <tag/> (including 0),
- <tag/>{x, y} – a sequence of x to y tags,
- <tag1/> | <tag2/> – alternative.

C.2 Root elements

- <view/> – type: View (see Section C.3.1),
- <ruleSetPair/> – type: RuleSetPair (see Section C.3.2),
- <ruleSet/> – type: RuleSet (see Section C.3.3).

C.3 Types

C.3.1 View

Definition of a view.

Attributes

- type – required, type: string, qualified class name

Body

(<view/> | <ruleSetPair/> | <ruleSet/> | <str/> | <int/> | <bool/>)*

- <view/> – type: View (see Section C.3.1), embedded view passed as an argument;
- <ruleSetPair/> – type: RuleSetPair (see Section C.3.2), embedded pair of rule sets passed as an argument;
- <ruleSet/> – type: RuleSet (see Section C.3.3), embedded rule set passed as an argument;
- <str/> – type: string, string passed as an argument;
- <int/> – type: int, integer passed as an argument;
- <bool/> – type: boolean, boolean value passed as an argument.

Example

```
<view type="pl.gda.pg.km.kv.adapters.JenaKB">
  <str>http://a.b.c/sampleTBox.owl</str>
  <bool>true</bool>
</view>
```

C.3.2 RuleSetPair

Pair of rule sets. The second set reverses the transformation done by the first set of rules.

Attributes

None

Body

<rules/> <reverseRules/>

- <rules/> – type: RuleSet (see Section C.3.3), forward rules;
- <reverseRules/> – type: RuleSet (see Section C.3.3), reverse rules.

Example

```
<ruleSetPair>
  <rules>
    <concept>
      <head>
        <atomicConcept name="C1"/>
        <individual name="i" isVariable="true"/>
      </head>
      <conceptAtom>
        <atomicConcept name="C2"/>
        <individual name="i" isVariable="true"/>
      </conceptAtom>
    </concept>
  </rules>
  <reverseRules>
    <concept>
      <head>
        <atomicConcept name="C2"/>
        <individual name="i" isVariable="true"/>
      </head>
      <conceptAtom>
        <atomicConcept name="C1"/>
        <individual name="i" isVariable="true"/>
      </conceptAtom>
    </concept>
  </reverseRules>
</ruleSetPair>
```

C.3.3 RuleSet

Set of rules.

Attributes

None

Body

<concept/>* <role/>* <attribute/>*

- <concept/> – type: ConceptRule (see Section C.3.4), rule defining a concept;
- <role/> – type: RoleRule (see Section C.3.5), rule defining a role;
- <attribute/> – type: AttributeRule (see Section C.3.6), rule defining an attribute.

Example

```
<rules>
  <concept>
    <head>
      <atomicConcept name="C1"/>
      <individual name="i" isVariable="true"/>
    </head>
    <conceptAtom>
      <atomicConcept name="C2"/>
      <individual name="i" isVariable="true"/>
    </conceptAtom>
  </concept>
</rules>
```

C.3.4 ConceptRule

Rule that defines a concept.

Attributes

None

Body

<head/> (<conceptAtom/> | <roleAtom/> | <attributeAtom/> | <builtinAtom/>)*

- <head/> – type: ConceptAtom (see Section C.3.7), head of the rule;
- <conceptAtom/> – type: ConceptAtom (see Section C.3.7), concept atom belonging to the body of the rule;
- <roleAtom/> – type: RoleAtom (see Section C.3.8), role atom belonging to the body of the rule;

- <attributeAtom/> – type: AttributeAtom (see Section C.3.9), attribute atom belonging to the body of the rule;
- <builtinAtom/> – type: BuiltinAtom (see Section C.3.10), built-in atom belonging to the body of the rule.

Example

```
<concept>
  <head>
    <atomicConcept name="C1"/>
    <individual name="i" isVariable="true"/>
  </head>
  <conceptAtom>
    <atomicConcept name="C2"/>
    <individual name="i" isVariable="true"/>
  </conceptAtom>
</concept>
```

C.3.5 RoleRule

Rule that defines a role.

Attributes

None

Body

<head/> (<conceptAtom/> | <roleAtom/> | <attributeAtom/> | <builtinAtom/>)*

- <head/> – type: RoleAtom (see Section C.3.8), head of the rule;
- <conceptAtom/> – type: ConceptAtom (see Section C.3.7), concept atom belonging to the body of the rule;
- <roleAtom/> – type: RoleAtom (see Section C.3.8), role atom belonging to the body of the rule;
- <attributeAtom/> – type: AttributeAtom (see Section C.3.9), attribute atom belonging to the body of the rule;

- <builtinAtom/> – type: BuiltinAtom (see Section C.3.10), built-in atom belonging to the body of the rule.

Example

```
<role>
  <head>
    <atomicRole name="R1"/>
    <subject name="s" isVariable="true"/>
    <object name="o" isVariable="true"/>
  </head>
  <roleAtom>
    <atomicRole name="R2"/>
    <subject name="s" isVariable="true"/>
    <object name="o" isVariable="true"/>
  </roleAtom>
</role>
```

C.3.6 AttributeRule

Rule that defines an attribute.

Attributes

None

Body

<head/> (<conceptAtom/> | <roleAtom/> | <attributeAtom/> | <builtinAtom/>)*

- <head/> – type: AttributeAtom (see Section C.3.9), head of the rule;
- <conceptAtom/> – type: ConceptAtom (see Section C.3.7), concept atom belonging to the body of the rule;
- <roleAtom/> – type: RoleAtom (see Section C.3.8), role atom belonging to the body of the rule;
- <attributeAtom/> – type: AttributeAtom (see Section C.3.9), attribute atom belonging to the body of the rule;

- <builtinAtom/> – type: BuiltinAtom (see Section C.3.10), built-in atom belonging to the body of the rule.

Example

```
<attribute>
  <head>
    <atomicAttribute name="A1"/>
    <subject name="s" isVariable="true"/>
    <object value="o" isVariable="true"/>
  </head>
  <attributeAtom>
    <atomicAttribute name="A2"/>
    <subject name="s" isVariable="true"/>
    <object value="o" isVariable="true"/>
  </attributeAtom>
</attribute>
```

C.3.7 ConceptAtom

Concept atom.

Attributes

None

Body

(<atomicConcept/> | <notConcept/> | <andConcept/> | <orConcept/> | <existsRoleConcept/>) <individual/>

- <atomicConcept/> – type: AtomicConcept (see Section C.3.11), atomic concept;
- <notConcept/> – type: NotConcept (see Section C.3.12), concept complement;
- <andConcept/> – type: AndConcept (see Section C.3.13), intersection of concepts;
- <orConcept/> – type: OrConcept (see Section C.3.14), union of concepts;

- <existsRoleConcept/> – type: ExistsRoleConcept (see Section C.3.15), existentially qualified concept;
- <individual/> – type: Individual (see Section C.3.18), individual.

Example

```
<conceptAtom>
  <atomicConcept name="C"/>
  <individual name="i" isVariable="true"/>
</conceptAtom>
```

C.3.8 RoleAtom

Role atom.

Attributes

None

Body

<atomicRole/> <subject/> <object/>

- <atomicRole/> – type: AtomicRole (see Section C.3.16), atomic role;
- <subject/> – type: Individual (see Section C.3.18), subject of the role;
- <object/> – type: Individual (see Section C.3.18), object of the role.

Example

```
<roleAtom>
  <atomicRole name="R"/>
  <subject name="s" isVariable="true"/>
  <object name="o" isVariable="true"/>
</roleAtom>
```

C.3.9 AttributeAtom

Attribute atom.

Attributes

None

Body

<atomicAttribute/> <subject/> <object/>

- <atomicAttribute/> – type: AtomicAttribute (see Section C.3.17), atomic attribute;
- <subject/> – type: Individual (see Section C.3.18), subject of the attribute;
- <object/> – type: Value (see Section C.3.19), object of the attribute.

Example

```
<attributeAtom>
  <atomicAttribute name="A"/>
  <subject name="s" isVariable="true"/>
  <object value="o" isVariable="true"/>
</attributeAtom>
```

C.3.10 BuiltinAtom

Built-in atom.

Attributes

None

Body

<builtin/> (<individual/> | <value/>)*

- <builtin/> – type: BuiltinType (see Section C.3.20), built-in predicate name;
- <individual/> – type: Individual (see Section C.3.18), individual being an argument of the built-in predicate;
- <value/> – type: Value (see Section C.3.19), value being an argument of the built-in predicate.

Example

```
<builtinAtom>
  <builtin>stringConcat</builtin>
  <value value="result" isVariable="true"/>
  <value value="s1" isVariable="true"/>
  <value value="s2" isVariable="true"/>
</builtinAtom>
```

C.3.11 AtomicConcept

Atomic concept.

Attributes

- name – required, type: string, name of the concept.

Body

None

Example

```
<atomicConcept name="C"/>
```

C.3.12 NotConcept

Complement of a concept.

Attributes

None

Body

<atomicConcept/> | <notConcept/> | <andConcept/> | <orConcept/> | <existsRoleConcept/>

- <atomicConcept/> – type: AtomicConcept (see Section C.3.11), atomic concept;

- <notConcept/> – type: NotConcept (see Section C.3.12), concept complement;
- <andConcept/> – type: AndConcept (see Section C.3.13), intersection of concepts;
- <orConcept/> – type: OrConcept (see Section C.3.14), union of concepts;
- <existsRoleConcept/> – type: ExistsRoleConcept (see Section C.3.15), existentially qualified concept;

Example

```
<notConcept>
  <atomicConcept name="C"/>
</notConcept>
```

C.3.13 AndConcept

Intersection of concepts.

Attributes

None

Body

(<atomicConcept/> | <notConcept/> | <andConcept/> | <orConcept/> | <existsRoleConcept/>){2, 2}

- <atomicConcept/> – type: AtomicConcept (see Section C.3.11), atomic concept;
- <notConcept/> – type: NotConcept (see Section C.3.12), concept complement;
- <andConcept/> – type: AndConcept (see Section C.3.13), intersection of concepts;
- <orConcept/> – type: OrConcept (see Section C.3.14), union of concepts;
- <existsRoleConcept/> – type: ExistsRoleConcept (see Section C.3.15), existentially qualified concept;

Example

```
<andConcept>
  <atomicConcept name="C1"/>
  <atomicConcept name="C2"/>
</andConcept>
```

C.3.14 OrConcept

Union of concepts.

Attributes

None

Body

(<atomicConcept/> | <notConcept/> | <andConcept/> | <orConcept/> | <existsRoleConcept/>){2, 2}

- <atomicConcept/> – type: AtomicConcept (see Section C.3.11), atomic concept;
- <notConcept/> – type: NotConcept (see Section C.3.12), concept complement;
- <andConcept/> – type: AndConcept (see Section C.3.13), intersection of concepts;
- <orConcept/> – type: OrConcept (see Section C.3.14), union of concepts;
- <existsRoleConcept/> – type: ExistsRoleConcept (see Section C.3.15), existentially qualified concept;

Example

```
<orConcept>
  <atomicConcept name="C1"/>
  <atomicConcept name="C2"/>
</orConcept>
```

C.3.15 ExistsRoleConcept

Existentially qualified concept.

Attributes

None

Body

<atomicRole/> (<atomicConcept/> | <notConcept/> | <andConcept/> | <orConcept/> | <existsRoleConcept/>)

- <atomicRole/> – type: AtomicRole (see Section C.3.16), atomic role;
- <atomicConcept/> – type: AtomicConcept (see Section C.3.11), atomic concept;
- <notConcept/> – type: NotConcept (see Section C.3.12), concept complement;
- <andConcept/> – type: AndConcept (see Section C.3.13), intersection of concepts;
- <orConcept/> – type: OrConcept (see Section C.3.14), union of concepts;
- <existsRoleConcept/> – type: ExistsRoleConcept (see Section C.3.15), existentially qualified concept;

Example

```
<existsRoleConcept>
  <atomicRole name="R"/>
  <atomicConcept name="C"/>
</existsRoleConcept>
```

C.3.16 AtomicRole

Atomic role.

Attributes

- name – required, type: string, name of the role.

Body

None

Example

```
<atomicRole name="R"/>
```

C.3.17 AtomicAttribute

Atomic attribute.

Attributes

- name – required, type: string, name of the attribute.

Body

None

Example

```
<atomicAttribute name="A"/>
```

C.3.18 Individual

Individual.

Attributes

- name – required, type: string, name of the individual or variable;
- isVariable – required, type: boolean, is it a variable to be bound.

Body

None

Example

```
<individual name="i" isVariable="false"/>
```

C.3.19 Value

Some value.

Attributes

- value – required, type: string, value or variable name;
- isVariable – required, type: boolean, is it a variable to be bound.
- type – optional, type: string, type of the value.

Body

None

Example

```
<value value="123" type="int" isVariable="false"/>
```

C.3.20 BuiltinType

Name of a built-in predicate, one of (type: string):

- uriToString – converts individual (first argument) to string representation of its URI (second argument), can also be used to create new individuals from strings,
- stringConcat – concatenates two strings (second and third argument) into third string (first argument),
- equal – checks for value equality,
- notEqual – checks for value inequality,
- lessThan – less than predicate,
- lessThanOrEqual – less than or equal predicate,
- greaterThan – greater than predicate,
- greaterThanOrEqual – greater than or equal predicate.

Appendix D

Knowledge Views Javadocs

D.1 Package pl.gda.pg.km.kv

D.1.1 Interface ABox

An interface representing an information source.

```
public interface ABox
```

Method getConceptAssertions

Returns a collection of Individuals fulfiling the given criteria.

```
public java.util.Collection<pl.gda.pg.km.kv.rules.Individual>
getConceptAssertions(
   pl.gda.pg.km.kv.ontology.Concept concept,
   pl.gda.pg.km.kv.rules.Individual pattern)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- concept – Concept to which the Individuals belong
- pattern – pattern of individuals to be matched, null for a wildcard

Returns: a collection of Individuals fulfiling the given criteria

Method putConceptAssertion

Inserts the given concept assertion into the information source.

```
public void putConceptAssertion(
  pl.gda.pg.km.kv.ontology.Concept concept,
  pl.gda.pg.km.kv.rules.Individual individual)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
  java.lang.UnsupportedOperationException
```

Parameters:

- concept – the concept whose assertion to insert into the information source.
- individual – the individual to insert into the information source.

Method delConceptAssertion

Deletes the given concept assertion from the information source.

```
public void delConceptAssertion(
  pl.gda.pg.km.kv.ontology.Concept concept,
  pl.gda.pg.km.kv.rules.Individual individual)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
  java.lang.UnsupportedOperationException
```

Parameters:

- concept – the concept whose assertion to delete.
- individual – the individual to delete.

Method getRoleAssertions

Returns a collection of role instances fulfiling the given criteria.

```
public java.util.Collection<
    pl.gda.pg.km.kv.ontology.BinaryInstance<
      pl.gda.pg.km.kv.rules.Individual,
      pl.gda.pg.km.kv.rules.Individual>> getRoleAssertions(
  pl.gda.pg.km.kv.ontology.Role role,
  pl.gda.pg.km.kv.ontology.BinaryInstance<
    pl.gda.pg.km.kv.rules.Individual,
    pl.gda.pg.km.kv.rules.Individual> pattern)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- role – Role to which the Individuals belong
- pattern – Pattern of role instances to return

Returns: a collection of role instances fulfiling the given criteria

Method putRoleAssertion

Inserts the given role assertion into the information source.

```
public void putRoleAssertion(
  pl.gda.pg.km.kv.ontology.Role role,
  pl.gda.pg.km.kv.ontology.BinaryInstance<
    pl.gda.pg.km.kv.rules.Individual,
    pl.gda.pg.km.kv.rules.Individual> instance)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
  java.lang.UnsupportedOperationException
```

Parameters:

- role – the role whose assertion to insert into the information source
- instance – the instance to insert into the information source

Method delRoleAssertion

Deletes the given role assertion from the information source.

```
public void delRoleAssertion(
  pl.gda.pg.km.kv.ontology.Role role,
  pl.gda.pg.km.kv.ontology.BinaryInstance<
    pl.gda.pg.km.kv.rules.Individual,
    pl.gda.pg.km.kv.rules.Individual> instance)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
  java.lang.UnsupportedOperationException
```

Parameters:

- role – the role whose assertion to delete.
- instance – the instance to delete.

Method getAttributeAssertions

Returns a collection of attribute instances fulfiling the given criteria.

```
public java.util.Collection<
    pl.gda.pg.km.kv.ontology.BinaryInstance<
      pl.gda.pg.km.kv.rules.Individual,
      pl.gda.pg.km.kv.rules.Value>> getAttributeAssertions(
  pl.gda.pg.km.kv.ontology.Attribute attribute,
  pl.gda.pg.km.kv.ontology.BinaryInstance<
    pl.gda.pg.km.kv.rules.Individual,
    pl.gda.pg.km.kv.rules.Value> pattern)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- attribute – Attribute to which the instance belong
- pattern – pattern of attribute instances to return

Returns: a collection of attribute instances fulfiling the given criteria

Method putAttributeAssertion

Inserts the given attribute assertion into the information source.

```
public void putAttributeAssertion(
  pl.gda.pg.km.kv.ontology.Attribute attribute,
  pl.gda.pg.km.kv.ontology.BinaryInstance<
    pl.gda.pg.km.kv.rules.Individual,
    pl.gda.pg.km.kv.rules.Value> instance)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
  java.lang.UnsupportedOperationException
```

Parameters:

- attribute – the attribute whose assertion to insert into the information source
- instance – instance to insert into the information source

Method delAttributeAssertion

Deletes the given attribute assertion from the information source.

```
public void delAttributeAssertion(
  pl.gda.pg.km.kv.ontology.Attribute attribute,
  pl.gda.pg.km.kv.ontology.BinaryInstance<
    pl.gda.pg.km.kv.rules.Individual,
    pl.gda.pg.km.kv.rules.Value> instance)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
  java.lang.UnsupportedOperationException
```

Parameters:

- attribute – the attribute whose assertion to delete
- instance – the instance to delete

Method putAssertions

Inserts a collection of assertion into the information source.

```
public void putAssertions(
   java.util.Collection<pl.gda.pg.km.kv.rules.Atom> assertions)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
   java.lang.UnsupportedOperationException
```

Parameters:

- assertions – assertions to insert into the information source

Method delAssertions

Deletes a collection of assertions from the information source

```
public void delAssertions(
   java.util.Collection<pl.gda.pg.km.kv.rules.Atom> assertions)
throws pl.gda.pg.km.kv.KnowledgeBaseException,
   java.lang.UnsupportedOperationException
```

Parameters:

- assertions – assertions to delete

Method getConcepts

Returns a collection of concepts this information source supports.

```
public java.util.Collection<pl.gda.pg.km.kv.ontology.Concept>
getConcepts()
```

Returns: a collection of concepts this information source supports

Method getRoles

Returns a collection of roles this information source supports.

```
public java.util.Collection<pl.gda.pg.km.kv.ontology.Role>
getRoles()
```

Returns: a collection of roles this information source supports

Method getAttributes

Returns a collection of attributes this information source supports.

```
public java.util.Collection<pl.gda.pg.km.kv.ontology.Attribute>
getAttributes()
```

Returns: a collection of attributes this information source supports

D.1.2 Interface TBox

An interface representing a terminology

```
public interface TBox
```

Method getConcepts

Returns a collection of supported concepts.

```
public java.util.Collection<pl.gda.pg.km.kv.ontology.Concept>
getConcepts()
```

Returns: a collection of supported concepts

Method getRoles

Returns a collection of supported roles.

```
public java.util.Collection<pl.gda.pg.km.kv.ontology.Role>
getRoles()
```

Returns: a collection of supported roles

Method getAttributes

Returns a collection of supported attributes.

```
public java.util.Collection<pl.gda.pg.km.kv.ontology.Attribute>
getAttributes()
```

Returns: a collection of supported attributes

Method subsumes

Returns true if lhv subsumes rhv.

```
public boolean subsumes(
  pl.gda.pg.km.kv.ontology.Concept lhv,
  pl.gda.pg.km.kv.ontology.Concept rhv)
```

Parameters:

- lhv – subsumer
- rhv – subsumee

Returns: true if lhv subsumes rhv

Method isEqual

A convenience method that checks if lhv subsumes rhv and rhv subsumes lhv.

```
public boolean isEqual(
  pl.gda.pg.km.kv.ontology.Concept lhv,
  pl.gda.pg.km.kv.ontology.Concept rhv)
```

Parameters:

- lhv – first concept to check for equivalence
- rhv – second concept to check for equivalence

Returns:

Method isSatisfiable

Checks if the given concept is satisfiable.

```
public boolean isSatisfiable(
  pl.gda.pg.km.kv.ontology.Concept concept)
```

Parameters:

- concept – a concept to check fo satisfiability

Returns: true if concept is satisfiable

Method getRoleDomain

Returns the domain of the given role.

```
public pl.gda.pg.km.kv.ontology.Concept getRoleDomain(
  pl.gda.pg.km.kv.ontology.Role role)
```

Parameters:

- role – role whose domain to return

Returns: domain of the given role

Method getRoleRange

Returns the range of the given role

```
public pl.gda.pg.km.kv.ontology.Concept getRoleRange(
   pl.gda.pg.km.kv.ontology.Role role)
```

Parameters:

- role – role whose range to return

Returns: range of the given role

Method isRoleFunctional

Checks if the given role is functional

```
public boolean isRoleFunctional(
   pl.gda.pg.km.kv.ontology.Role role)
```

Parameters:

- role – role to check whether it is functional

Returns: true if the role is functional

Method getAttributeDomain

Returns the domain of the given attribute.

```
public pl.gda.pg.km.kv.ontology.Concept getAttributeDomain(
   pl.gda.pg.km.kv.ontology.Attribute attribute)
```

Parameters:

- attribute – attribute whose domain to return

Returns: domain of the given attribute

Method getAttributeType

Returns the range of the given attribute.

```
public java.lang.Class<?> getAttributeType(
   pl.gda.pg.km.kv.ontology.Attribute attribute)
```

Parameters:

- attribute – attribute whose range to return

Returns: range of the given attribute

Method isAttributeFunctional

Checks if the given attribute is functional.

```
public boolean isAttributeFunctional(
  pl.gda.pg.km.kv.ontology.Attribute attribute)
```

Parameters:

- attribute – attribute to check whether it is function

Returns: true if the attribute is functional

D.1.3 Interface KnowledgeBase

A knowledge base - implements both the ABox and TBox interfaces.

```
public interface KnowledgeBase
extends pl.gda.pg.km.kv.ABox, pl.gda.pg.km.kv.TBox
```

D.1.4 Class ABoxView

Performes ABox transformation according to a given RuleSetPair.

```
public ABoxView
extends java.lang.Object
implements pl.gda.pg.km.kv.ABox
```

Constructor ABoxView

Creates a new instance of ABoxView.

```
public ABoxView(
  pl.gda.pg.km.kv.ABox aBox,
  pl.gda.pg.km.kv.neek.RuleSetPair ruleSetPair)
```

Parameters:

- aBox – an ABox that is the information source to transform
- ruleSetPair – a RuleSetPair used to transform the given ABox

Methods from interface ABox

This class implements methods defined in interface ABox.

D.1.5 Class InferencingKnowledgeView

A Knowledge View that performs reasoning overt the given ABox according to the given TBox and RuleSet.

```
public InferencingKnowledgeView
extends java.lang.Object
implements pl.gda.pg.km.kv.KnowledgeBase
```

Constructor InferencingKnowledgeView

Creates a new instance of InferencingKnowledgeView.

```
public InferencingKnowledgeView(
  pl.gda.pg.km.kv.TBox tBox,
  pl.gda.pg.km.kv.ABox aBox,
  pl.gda.pg.km.kv.neek.RuleSet  rules)
```

Parameters:

- tBox – TBox used in reasoning
- aBox – ABox to reason over
- rules – Horn rules being part of the knowledge base (content of rules can be changed by this object)

Methods from interface ABox

This class implements methods defined in interface ABox.

Methods from interface TBox

This class implements methods defined in interface TBox.

D.1.6 Class KnowledgeView

A Knowledge View that does not perform reasoning. Is only used to combine an ABox with a TBox.

```
public KnowledgeView
extends java.lang.Object
implements pl.gda.pg.km.kv.KnowledgeBase
```

Constructor KnowledgeView

Creates a new instance of KnowledgeView.

```
public KnowledgeView(
  pl.gda.pg.km.kv.TBox tBox,
  pl.gda.pg.km.kv.ABox aBox)
```

Parameters:

- tBox – TBox providing terminology
- aBox – ABox providing assertions

Methods from interface ABox

This class implements methods defined in interface ABox.

Methods from interface TBox

This class implements methods defined in interface TBox.

D.1.7 Class MemoryABox

An ABox that stores assertions in memory.

```
public MemoryABox
extends java.lang.Object
implements pl.gda.pg.km.kv.ABox
```

Constructor MemoryABox

Create a new empty instance of MemoryABox.

```
public MemoryABox()
```

Method clear

Removes all assertions from this ABox

```
public void clear()
```

Method clearAll

Removes all assertions from this ABox as well as removes the information about supported concepts, roles and attributes.

```
public void clearAll()
```

Methods from interface ABox

This class implements methods defined in interface ABox.

D.1.8 Class MergedABox

An ABox that combines several other ABoxes.

```
public MergedABox
extends java.lang.Object
implements pl.gda.pg.km.kv.ABox
```

Constructor MergedABox

Creates a new instance of MergedABox that combines assertions from the given ABoxes.

```
public MergedABox(
   java.util.Collection<pl.gda.pg.km.kv.ABox> aboxes)
```

Parameters:

- aboxes – ABoxes to combine

Methods from interface ABox

This class implements methods defined in interface ABox.

D.1.9 Class QueryEngine

Class that provides querying capabilities for a given ABox.

```
public QueryEngine
extends java.lang.Object
```

Constructor QueryEngine

Creates a new QueryEngine for the given ABox.

```
public QueryEngine(
  pl.gda.pg.km.kv.ABox aBox)
```

Parameters:

- aBox – ABox to be queried

Method conceptQuery

Returns a collection of individuals that are a result of evaluating the given ConceptRule.

```
public java.util.Collection<pl.gda.pg.km.kv.rules.Individual>
conceptQuery(
  pl.gda.pg.km.kv.rules.ConceptRule rule)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- rule – a rule to evaluate

Returns: a collection of individuals that are a result of evaluating the given ConceptRule

Method roleQuery

Returns a collection of role instances that are a result of evaluating the given RoleRule.

```
public java.util.Collection<
    pl.gda.pg.km.kv.ontology.BinaryInstance<
      pl.gda.pg.km.kv.rules.Individual,
      pl.gda.pg.km.kv.rules.Individual>> roleQuery(
  pl.gda.pg.km.kv.rules.RoleRule rule)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- rule – a rule to evaluate

Returns: a collection of role instances that are a result of evaluating the given RoleRule

Method attributeQuery

Returns a collection of attribute instances that are a result of evaluating the given AttributeRule.

```
public java.util.Collection<
    pl.gda.pg.km.kv.ontology.BinaryInstance<
      pl.gda.pg.km.kv.rules.Individual,
      pl.gda.pg.km.kv.rules.Value>> attributeQuery(
  pl.gda.pg.km.kv.rules.AttributeRule rule)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- rule – a rule to evaluate

Returns: a collection of attribute instances that are a result of evaluating the given AttributeRule

Method query

Returns a list of bindings of loose variables in the given atom list.

```
public java.util.List<java.util.Map<java.lang.String,
    pl.gda.pg.km.kv.rules.Term>> query(
  java.util.List<pl.gda.pg.km.kv.rules.Atom> query)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- query – atom list to evaluate

Returns: a list of bindings of loose variables in the given atom list

Method optimiseQuery

Optimises the query by changing atom order

```
public static java.util.List<pl.gda.pg.km.kv.rules.Atom>
optimiseQuery(
  java.util.List<pl.gda.pg.km.kv.rules.Atom> query)
```

Parameters:

- query – query to optimise

Returns: optimised query

Method optimiseQuery

Optimises the query by changing atom order. Constants are a set of variable names that should be considered already bound.

```
public static java.util.List<pl.gda.pg.km.kv.rules.Atom>
optimiseQuery(
  java.util.List<pl.gda.pg.km.kv.rules.Atom> query,
  java.util.Set<java.lang.String> constants)
```

Parameters:

- query – query to optimise
- constants – a set of variable names that should be considered already bound

Returns: optimised query

D.1.10 Class ViewFactory

A factory used to create KnowledgeBases, ABoxes and TBoxes from an xNeeK manifest object (View).

```
public ViewFactory
extends java.lang.Object
```

Constructor ViewFactory

Creates a new instance of ViewFactory.

```
public ViewFactory()
```

Method createABox

Creates an ABox from a manifest object.

```
public static pl.gda.pg.km.kv.ABox createABox(
  pl.gda.pg.km.kv.neek.View view)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- view – the manifest object

Returns: the ABox

Method createTBox

Creates a TBox from a manifest object

```
public static pl.gda.pg.km.kv.TBox createTBox(
  pl.gda.pg.km.kv.neek.View view)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- view – the manifest object

Returns: the TBox

Method createKnowledgeBase

Creates a KnowledgeBase from a manifest object

```
public static pl.gda.pg.km.kv.KnowledgeBase createKnowledgeBase(
  pl.gda.pg.km.kv.neek.View view)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- view – the manifest object

Returns: the KnowledgeBase

D.1.11 Class TBoxView

Create aliases for concept, role and attribute expressions from another TBox

```
public TBoxView
extends java.lang.Object
implements pl.gda.pg.km.kv.TBox
```

Constructor TBoxView

Creates a new instance of TBoxView.

```
public TBoxView(
  pl.gda.pg.km.kv.TBox tBox,
  java.util.Map<pl.gda.pg.km.kv.ontology.Concept,
    pl.gda.pg.km.kv.ontology.Concept> concepts,
  java.util.Map<pl.gda.pg.km.kv.ontology.Role,
    pl.gda.pg.km.kv.ontology.Role> roles,
  java.util.Map<pl.gda.pg.km.kv.ontology.Attribute,
    pl.gda.pg.km.kv.ontology.Attribute> attributes)
```

Parameters:

- tBox – TBox being the source terminology
- concepts – pairs of concepts, first concept is from this TBox and second is from source TBox
- roles – pairs of roles, first role is from this TBox and second is from source TBox
- attributes – pairs of attributes, first attribute is from this TBox and second is from source TBox

Methods from interface TBox

This class implements methods defined in interface TBox.

Appendix E

Object Views Javadocs

E.1 Package pl.gda.pg.km.ov

E.1.1 Class ObjectViewGenerator

Class used for generating classes and interfaces from terminology.

```
public ObjectViewGenerator
extends java.lang.Object
```

Constructor ObjectViewGenerator

Creates a new generator of the given TBox.

```
public ObjectViewGenerator(
   pl.gda.pg.km.kv.TBox knowledgeBase)
```

Parameters:

- knowledgeBase – TBox to generate classes and interfaces from

Method main

Method implementing command line program for generating classes and interfaces.

```
public static void main(
   java.lang.String[] args)
throws java.io.IOException
```

Parameters:

- args – the command line arguments

Method generateCode

Generates code for the TBox that will be placed in the given basePackage

```
public java.util.Map<java.lang.String, java.lang.String>
generateCode(
  java.lang.String basePackage)
```

Parameters:

- basePackage – the package to place the classes and interfaces in

Returns: pairs of class/interface names and their source code.

E.1.2 Class ObjectViewManager

Tha class used to create an Object View.

```
public ObjectViewManager
extends java.lang.Object
```

Constructor ObjectViewManager

Creates a new instance of ObjectViewManager for the given knowledge base and ontology representing classes. All the classes given as types need to be annotated with Concept and have to have exactly one Uri annotated getter with a matching setter. The classes should use Role and Attribute annotations to map getters to roles and attributes from the knowledge base.

```
public ObjectViewManager(
  pl.gda.pg.km.kv.ABox knowledgeBase,
  java.lang.Class[] types)
```

Parameters:

- knowledgeBase – knowledge base to use
- types – annotated classes to be used to represent the query answeres

Method query

Methods that queries for instance with given id that needs to be of the specified class.

```
public T query(
  java.lang.String id,
  java.lang.Class<T> type)
throws pl.gda.pg.km.ov.ObjectViewException
```

Parameters:

- id – the URI of the instance.
- type – the annotated class to be returned.

Returns: object representing the instance or null if the instance is not of the specified class.

Method query

Method that queries for instances with given ids that need to be of the specified annotated class.

```
public java.util.Collection<T> query(
  java.util.Collection<java.lang.String> ids,
  java.lang.Class<T> type)
throws pl.gda.pg.km.ov.ObjectViewException
```

Parameters:

- ids – the URIs of the instances.
- type – the annotated class to be returned.

Returns: objects representing the instances. If some instance is not of the specified class null is included.

Method query

Method returning all instance of the given Concept that need to be of the specified annotated class.

```
public java.util.Collection<T> query(
  pl.gda.pg.km.kv.ontology.Concept concept,
  java.lang.Class<T> type)
throws pl.gda.pg.km.ov.ObjectViewException
```

Parameters:

- concept – concept to which the returned instances belong
- type – the annotated class to be returned

Returns: instance of the given Concept

Method persist

Persists the given Object. The object has to be an instance of an annotated class.

```
public void persist(
  java.lang.Object obj)
throws pl.gda.pg.km.ov.ObjectViewException
```

Parameters:

- obj – object to be persisted

Method remove

Removes the given Object from the repository. The object has to be an instance of an annotated class.

```
public void remove(
  java.lang.Object obj)
throws pl.gda.pg.km.ov.ObjectViewException
```

Parameters:

- obj – object to be removed from the repository.

Method merge

Merges changes done to the Object with data in the repository. The object has to be an instance of an annotated class.

```
public void merge(
  java.lang.Object obj)
throws pl.gda.pg.km.ov.ObjectViewException
```

Parameters:

- obj – object to be merged with its image in the repository

Method executeQuery

Performs an OQL query and returns results as a collection of instances of the given annotated class.

```
public java.util.Collection<T> executeQuery(
  java.lang.String query,
  java.lang.Class<T> type)
throws pl.gda.pg.km.ov.ObjectViewException,
  pl.gda.pg.km.ov.query.QueryException
```

Parameters:

- query – an OQL query to be executed
- type – type of the objects returned

Returns: collection of instances being the result of the query

E.2 Package pl.gda.pg.km.ov.annotation

E.2.1 Annotation Attribute

This annotation marks which getter is to be used to get the value of the attribute specified by this annotation argument.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Attribute
```

Element value

The URI of the attribute.

```
public java.lang.String value
```

E.2.2 Annotation Concept

This annotation specifies that the annotated class represents the concept specified by the URI being this annotation attribute.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Concept
```

Element value

The URI of the concept the annotated class represents.

```
public java.lang.String value
```

E.2.3 Annotation Role

This annotation marks which getter is to be used to get the role fillers of the role specified by this annotation argument.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Role
```

Element uri

The URI of the role.

```
public java.lang.String uri
```

Element range

The annotated class the role fillers should be.

```
public java.lang.Class<?> range
```

E.2.4 Annotation Uri

This annotation marks which getter is to be used to get the URI of the instance. The method has to return a String.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Uri
```

Appendix F

RDF Views Javadocs

F.1 Package pl.gda.pg.km.rv

F.1.1 Class RDFView

An RDFView that allows querying ABoxes using SPARQL.

```
public RDFView
extends java.lang.Object
```

Constructor RDFView

Creates a new RDFView for the given ABox

```
public RDFView(
   pl.gda.pg.km.kv.ABox aBox)
```

Parameters:

- aBox – ABox that will be used for querying.

Method getABox

Returns the ABox that is being queried.

```
public pl.gda.pg.km.kv.ABox getABox()
```

Returns: the ABox that is being queried

Method selectQuery

Performs a SELECT query.

```
public pl.gda.pg.km.rv.result.Sparql selectQuery(
  java.lang.String queryString)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- queryString – SPARQL SELECT query to issue.

Returns: result of the SPARQL query.

Method askQuery

Performs an ASK query.

```
public pl.gda.pg.km.rv.result.Sparql askQuery(
  java.lang.String queryString)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- queryString – SPARQL ASK query to issue.

Returns: result of the SPARQL query

Method constructQuery

Performs a CONSTRUCT query.

```
public pl.gda.pg.km.rv.RDFView constructQuery(
  java.lang.String queryString)
throws pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- queryString – SPARQL CONSTRUCT query to issue.

Returns: an RDFView that represents the resulting RDF graph

Method dump

Dumps the content of the undelying ABox as an RDF file.

```
public void dump(
  java.io.OutputStream out)
throws java.io.IOException,
  pl.gda.pg.km.kv.KnowledgeBaseException
```

Parameters:

- out – stream to dump content to

Method knowledgeBaseToABox

Creates an ABox the contains an RDF graph for both assertions and terminology

```
public static pl.gda.pg.km.kv.ABox knowledgeBaseToABox(
   pl.gda.pg.km.kv.KnowledgeBase knowledgeBase)
```

Parameters:

- knowledgeBase – knowledge base that contains the assertions and terminology

Returns: an ABox the contains an RDF graph for both assertions and terminology.

Method tBoxToABox

Converts a TBox to an RDF ABox.

```
public static pl.gda.pg.km.kv.ABox tBoxToABox(
   pl.gda.pg.km.kv.TBox tBox)
```

Parameters:

- tBox – TBox to convert

Returns: return an RDF ABox for the given TBox

Appendix G

Data Views Javadocs

Data Views library implements the standard JDBC API [2], therefore it conforms to the standard Java SE 6 Javadocs [98].

Widoki na bazę wiedzy i ich zastosowanie w inżynierii systemów

Streszczenie

1 Wstęp

1.1 Problematyka

Można zaobserwować rosnące zainteresowanie technologiami Semantic Web [100]. Jednakże wykorzystanie tych technologii w przemyśle informatycznym jest niewielkie. Większe zainteresowanie technologiami Semantic Web można zaobserwować w środowisku akademickim niż w przemyśle. Mimo to niektóre duże firmy, takie jak Hewlett-Packard, Oracle, czy Google, zainwestowały pewne środki w technologie semantyczne.

Jest wiele powodów, dla których technologie semantyczne są ciągle uważane za nowość. Po pierwsze wdrażanie nowych technologii jest zawsze ryzykowne. Powoduje to, że trudno jest wyprzeć stare, sprawdzone technologie. Powoduje to również częste mieszanie nowych technologii ze starymi.

Brak standardów jest kolejnym czynnikiem zwiększające ryzyko wdrażania technologii semantycznych. Istnieje kilka standardów promowanych przez W3C, np. RDF [56, 62, 47], SPARQL [91], czy też OWL [83, 66] i OWL 2 [101]. Jest to jednak ciągle niewystarczające. Proces powstawania standardów jest zazwyczaj długotrwały, a pierwsze wersje standardów są często niedopracowane i wymagają sprzężenia zwrotnego od użytkowników.

Kolejną przyczyną powolnego wdrażania technologii semantycznych jest sceptycyzm i brak wiedzy na temat tych technologii wśród inżynierów oprogramowania. Doświadczenie zdobyte przez autora w trakcie projektu PIPS (Personalised Information Platform for Life and Heath Services [89]) potwierdza stwierdzenie, że inżynierowie oprogramowania niechętnie poznają technologie semantyczne, które są jeszcze niedojrzałe, a przez to bardziej podatne na zmiany. Również dyskusje z inżynierami oprogramowania wykazywały ich brak wiedzy i sceptycyzm w stosunku do technologii semantycznych. Powoduje to swoiste zakleszczenie: technologie semantyczne nie są używane, ponieważ nie są dojrzałe, a nie są dojrzałe, ponieważ nie są używane, więc nie ma sprzężenia zwrotnego od użytkowników.

Aby zmniejszyć ryzyko związane z wdrażaniem nowych technologii oraz zminimalizować nakład pracy inżynierów na zapoznanie się z nimi, można przesłonić technologie semantyczne warstwą, która będzie dostarczała interfejsy powszechnie używane w inżynierii systemów. Taka warstwa pośredniczyłaby pomiędzy technologiami, a zarazem zwiększałaby kompatybilność tych technologii. Ponadto inżynier miałby do czynienia ze znanymi sobie interfejsami. Dzięki temu mógłby skorzystać z technologii semantycznych bez konieczności poznawania wszystkich związanych z nimi niuansów – byłyby one przesłonięte. Podejście to jest ewolucyjne, a nie rewolucyjne – pozwala mieszać technologie, a przez to stopniowo wprowadzać technologie semantyczne do praktycznego stosowania w klasycznych systemach informatycznych.

1.2 Cele i teza pracy

Głównymi celami pracy są:

1. Zdefiniowanie odwzorowań modelu ontologicznego na modele powszechnie używane we współczesnych systemach informatycznych, takie jak model obiektowy, relacyjny i RDF.
2. Zdefiniowanie zestawu przekształceń dla modelu ontologicznego, który pozwoli dostosować dany model do potrzeb wynikających z logiki aplikacji z niego korzystającej.

Tezą pracy jest:

> **Widoki na bazę wiedzy, które ukrywają model ontologiczny i udostępniają jeden z modeli powszechnie znanych w inżynierii systemów, pozwalają na łatwe wykorzystanie baz wiedzy jako źródeł danych we współczesnych systemach informatycznych.**

2 Przegląd bieżącego stanu wiedzy

Kilka różnych dziedzin miało wpływ na rozwój widoków na bazę wiedzy. Najważniejszymi z nich są: interoperacyjność i wymienność komponentów, co jest ściśle powiązane ze standardami, języki opisu ontologii oraz integracja danych. Ponadto istotny wpływ na rozwój widoków na bazę wiedzy miało kilka już istniejących technologii.

2.1 Interoperacyjność, standardy i wymienność komponentów

Interoperacyjność to zdolność systemów do wymiany informacji oraz wykorzystania wymienianych informacji [49]. Dla przykładu, programując w języku Java aplikację bazodanową, korzystamy z interfejsu programistycznego

(API) JDBC [2], natomiast zapytania zapisujemy w języku SQL. JDBC określa sposób wymiany informacji poprzez wywoływanie odpowiednich metod, natomiast SQL określa właściwą wiadomość, która jest przesyłana.

Gdy tworzy się komponent, który ma mieć możliwość współpracy z innymi komponentami, trzeba mieć na uwadze interoperacyjność na różnych poziomach abstrakcji. Dla przykładu, aby przesłać czterobajtową liczbę całkowitą, trzeba wiedzieć, w jakiej kolejności przesłać pojedyncze bajty. Na wyższym poziomie, gdy wywołuje się funkcje programistyczne, trzeba przestrzegać interfejsu ABI, który definiuje między innymi, w jaki sposób przekazywać parametry do funkcji, np. czy przekazać je za pośrednictwem rejestrów, czy też poprzez stos. Na jeszcze wyższym poziomie jest interfejs API, itd.

Standardy są istotnym elementem wpływającym na interoperacyjność. Definiują na przykład, w jaki sposób komponenty mają wymieniać ze sobą informacje. W przypadku relacyjnych baz danych ustandaryzowano dla języka Java interfejs programistyczny JDBC, który stanowi medium wymiany informacji pomiędzy aplikacją a bazą danych. Dzięki temu standardowi oraz standardowi SQL możliwa jest w Javie wymiana bazy danych na bazę danych innego producenta. Taką wymianę bazy danych utrudnia korzystanie z funkcji specyficznej dla konkretnego producenta, czyli wykraczanie poza standardy.

Inaczej niż w przypadku relacyjnych baz danych ma się sprawa z repozytoriami RDF. Ustandaryzowany został język zapytań SPARQL [91], który jest odpowiednikiem SQL-a, oraz protokół komunikacyjny SPARQL [18]. Protokół SPARQL jest zdefiniowany jako usługa sieciowa, czyli jest na wyższym poziomie abstrakcji niż interfejs programistyczny JDBC.

Nie zawsze komponenty tworzone są z myślą o interoperacyjności. Czasami jest potrzeba dostosowania istniejących komponentów do współpracy ze sobą. Typowymi wzorcami projektowymi wykorzystywanymi w tym celu są adaptory, konwertery oraz sterowniki.

2.2 Języki definiowania ontologii

W środowiskach związanych z inicjatywą Semantic Web podstawowym językiem stosowanym do definiowania ontologii jest OWL [66, 101]. Jednakże w inżynierii oprogramowania były już stosowane języki definiowania ontologii na długo przed powstaniem inicjatywy Semantic Web. Dwoma najważniejszymi są ERM [17] oraz UML [77, 78].

ERM, UML oraz OWL mają inne przeznaczenie, lecz w pewnych aspektach są do siebie podobne. Każdy z wymienionych języków pozwala na definiowanie zbiorów bytów, które posiadają cechy wspólne. W ERM zbiory te nazywają się zbiorami encji, w języku UML klasami, w języku OWL również są to klasy, które w terminologii logiki opisowej są konceptami. W każdym z tych języków wspomniane byty posiadają atrybuty. Ponadto możliwe jest definiowanie związków pomiędzy bytami. Istotna jest też tożsamość bytów, lecz tutaj zaczynają się różnice. W ERM tożsamość jest definiowana przez wartości atrybutów. W języku UML obiekt ma tożsamość niezależną od jego zawartości. W języku OWL tożsamość wyznacza etykieta, dokładniej IRI. Dwa różne IRI mogą wskazywać na to samo indywiduum, więc samo porównywanie IRI nie jest wystarczające, aby ustalić tożsamość indywiduum.

Inne przeznaczenie każdego z wymienionych języków powoduje, że różnic między nimi jest więcej, np. UML, jako jedyny w tej grupie ma możliwość opisania czynności. W rozprawie tej uwaga jest jednak skupiana na podobieństwach, a nie na różnicach.

2.3 Widoki a integracja informacji

Idea widoków jest szeroko rozpowszechniona w informatyce i ma różne zastosowania, np. ukrywanie poufnych danych w bazach danych. Jednymi z ważniejszych cech widoków są: zdolność do ukrywania informacji oraz zdolność do przekształcania informacji. Interesującym zastosowaniem widoków, w którym cechy te są wykorzystywane, jest integracja informacji.

Dwoma podstawowymi podejściami do integracji informacji z wykorzystaniem widoków są: Local-as-View (LaV) [60, 99] oraz Global-as-View (GaV) [16, 99]. W podejściu LaV lokalne źródła informacji są traktowane jako widoki na schemat globalny, zatem odwzorowania pomiędzy schematami lokalnymi a schematem globalnym polegają na definiowaniu bytów ze schematów lokalnych za pomocą bytów ze schematu globalnego. W podejściu GaV jest odwrotnie. Schemat globalny jest traktowany jako widok na schematy lokalne. W tym przypadku odwzorowania definiują byty ze schematu globalnego za pomocą bytów ze schematów lokalnych. Oba te podejścia mają swoje zalety i wady. Warto jednak zauważyć, że GaV wydaje się być bardziej intuicyjny, czego konsekwencją są prostsze algorytmy obsługujące GaV. Oba te podejścia pokazują, że samo wykorzystanie koncepcji widoków daje, w pewnym sensie „za darmo”, możliwości w zakresie integracji danych.

2.4 Istniejące technologie

Wpływ na niniejszą pracę miały istniejące aktualnie technologie. Najważniejsze z nich to Java Persistence API (JPA) [20], LINQ [68], JXPath [54], ActiveRDF [79, 80] oraz Jastor [51].

JPA jest nakładką na relacyjne bazy danych upodobniającą je do obiektowych baz danych. Dostęp do bazy danych jest realizowany za pomocą obiektowego języka zapytań. Podobnym do JPA rozwiązaniem jest LINQ. Poza dostępem do relacyjnych baz danych LINQ umożliwia również dostęp do innych źródeł informacji, np. plików XML oraz kolekcji obiektów. LINQ jest ponadto zintegrowany z językiem programowania – wymaga to pełnej kontroli nad językiem programowania, co zazwyczaj jest niemożliwe.

JXPath jest biblioteką programistyczną, która umożliwia odpytywanie drzewa obiektów z wykorzystaniem języka zapytań XPath [9], który jest przeznaczony dla języka XML. Jest to ciekawy przykład wykorzystania podobieństw pomiędzy językiem XML a modelem obiektowym w celu umożliwienia wykorzystania języka zapytań XPath w programach pisanych w obiektowym języku programowania.

ActiveRDF jest biblioteką programistyczną, która umożliwia obiektowy dostęp do repozytoriów RDF. Podstawową wadą tego rozwiązania jest brak ogólności. ActiveRDF wykorzystuje specyficzne cechy obiektowych języków programowania z dynamiczną typizacją. Autor biblioteki ActiveRDF odrzuca języki o statycznej typizacji, takie jak Java, C++, czy do niedawna C#, jako języki, które nie nadają się do obsługi RDF-a. Biorąc pod uwagę, że wymienione języki o statycznej typizacji są powszechnie używane, rozwiązanie to ma bardzo ograniczoną stosowalność.

Jastor jest biblioteką realizującą odwzorowanie ontologiczno-obiektowe zaproponowane w [55]. Odwzorowanie zastosowane w bibliotece Jastor różni się w pewnych istotnych elementach od odwzorowania zaproponowanego w niniejszej pracy. Istotną wadą biblioteki Jastor jako implementacji odwzorowania ontologiczno-obiektowego jest ścisłe związanie jej z biblioteką Jena [52], która stanowi jedyną bazę wiedzy, z którą Jastor może współpracować.

3 Widoki na bazę wiedzy

3.1 Koncepcja widoków na bazę wiedzy

Widoki na bazę wiedzy stanowią nakładkę na bazę wiedzy, która umożliwia dostęp do bazy wiedzy jak do relacyjnej bazy danych, obiektowej bazy danych, czy też repozytorium RDF. Ponadto użytkownik widoku może wybrać, które informacje mają być dostępne poprzez widok oraz przekształcić schemat udostępnianych informacji – jest to funkcjonalność znana dobrze z widoków w relacyjnych bazach danych. W ramach niniejszej pracy zostały opracowane następujące rodzaje widoków:

- DL View – ontologia sformułowana w logice opisowej,
- RDF View – model RDF wraz z trójkami RDF,

- Object View – model obiektowy wraz z obiektami,
- Data View – schemat relacyjnej bazy danych wraz z danymi.

Koncepcja widoków jest na tyle elastyczna, że umożliwia wprowadzenie nowych rodzajów widoków.

Podstawowymi zastosowaniami widoków na bazę wiedzy są uproszczenie dostępu do bazy wiedzy i zwiększenie kompatybilności baz wiedzy ze współczesnymi systemami informatycznymi. Ponadto przyjęte rozwiązania umożliwiły zastosowanie widoków na bazę wiedzy w integracji informacji oraz modularyzacji ontologii – są to jednak swoiste „skutki uboczne" i nie stanowią głównego przedmiotu rozprawy.

Widoki na bazę wiedzy starają się patrzeć na bazę wiedzy z punktu widzenia jej użytkownika, a nie twórcy. Konsekwencją tego jest umożliwienie przekształcania schematu, a nie tylko odwzorowanie modelu. Gdy pierwsze wersje widoków na bazę wiedzy były wykorzystane w projekcie PIPS [89], transformacja schematu danych okazała się nieodzowna – bazy wiedzy są często tworzone z myślą o wnioskowaniu i do tego celu dostosowana jest ontologia, natomiast wykorzystanie wywnioskowanych informacji w aplikacji często wymaga innego schematu.

Różne moduły w systemie informatycznym mogą wykorzystywać różne modele (ontologiczny, relacyjny, obiektowy) w zależności od swoich potrzeb. Ponieważ moduły są hermetyczne, modele w nich zawarte są prywatne i nie muszą, a często wręcz nie powinny, być udostępniane na zewnątrz. Moduły wymieniają się informacjami „publicznymi", co stanowi ich interfejs. Konsekwencją tej obserwacji jest założenie, że widoki na bazę wiedzy nie muszą odwzorowywać modeli wzajemnie jednoznacznie. Ponadto takie odwzorowanie jest często niemożliwe ze względu na różną ekspresywność poszczególnych modeli. Dla przykładu, podczas odwzorowania modelu ontologicznego na model relacyjny przenoszony jest schemat danych, natomiast pewne relacje między konceptami, które są niezbędne do wnioskowania, są pomijane.

3.2 Odwzorowanie modeli

Odwzorowanie modeli polega na przyporządkowaniu bytów źródłowego metamodelu do bytów docelowego metamodelu. W nomenklaturze Meta Object Facility (MOF) [73] jest to poziom M2. Dla przykładu, podczas odwzorowania modelu ERM na model obiektowy zbiór encji staje się klasą, związek referencją, a atrybut atrybutem. Odwzorowania mogą być stratne, tzn. nie wszystkie elementy jednego metamodelu muszą przekładać się na elementy drugiego metamodelu. Nie zawsze możliwe jest bezstratne odwzorowanie, a nawet jeśli jest możliwe, nie zawsze jest to potrzebne.

Opracowane w ramach niniejszej pracy odwzorowanie ontologiczno-relacyjne definiowane jest przez następujące reguły:

1. Każdy koncept jest reprezentowany przez jeden zbiór encji.
2. Każdy zbiór encji posiada atrybut będący kluczem głównym, który reprezentuje tożsamość indywiduum.
3. Każda rola funkcyjna jest reprezentowana przez związek typu „wiele do jednego”.
4. Każda rola niefunkcyjna jest reprezentowana przez związek typu „wiele do wielu”.
5. Każdy atrybut funkcyjny jest reprezentowany przez atrybut w każdym zbiorze encji, który nie jest rozłączny z dziedziną atrybutu.
6. Każdy atrybut niefunkcyjny jest reprezentowany przez zbiór encji z dwoma atrybutami: tożsamością encji posiadającej dany atrybut i wartością danego atrybutu. Ten zbiór encji jest połączony z odpowiednim zbiorem encji związkiem typu „wiele do wielu”.
7. Zawieranie konceptów jest reprezentowane przez związek IS_A.

Powstały w wyniku zastosowania powyższych reguł model związków encji jest przekształcany do modelu relacyjnego zgodnie z powszechnie stosowanymi zasadami.

Opracowane w ramach niniejszej pracy odwzorowanie ontologiczno-obiektowe definiowane jest przez następujące reguły:

1. Każdy koncept jest reprezentowany przez interfejs.
2. Każdy interfejs posiada parę akcesorów do wartości stanowiącej tożsamość obiektu.
3. Każda rola funkcyjna jest reprezentowana przez parę akcesorów w każdym interfejsie, który należy do dziedziny roli. Typ akcesorów jest interfejsem reprezentującym zakres roli.
4. Każda rola niefunkcyjna jest reprezentowana analogicznie do roli funkcyjnej, przy czym typem akcesorów jest kolekcja referencji typu zgodnego z zakresem roli.
5. Każdy atrybut funkcyjny jest reprezentowany analogicznie do roli funkcyjnej, przy czym typ akcesorów odpowiada typowi atrybutu.

6. Każdy atrybut niefunkcyjny jest reprezentowany analogicznie do funkcyjnego, przy czym typem akcesorów jest kolekcja wartości zgodnych z typem atrybutu.

7. Zawieranie konceptów jest reprezentowane przez dziedziczenie.

Odwzorowanie korzysta z interfejsów, a nie z klas, ponieważ wielokrotne dziedziczenie w niektórych językach programowania obiektowego jest możliwe tylko w przypadku interfejsów.

Odwzorowaniu modelu musi również towarzyszyć odwzorowanie języków zapytań. Często spotykanym rozwiązaniem jest przepisywanie zapytania w języku źródłowym na zapytanie w języku docelowym. Nie zawsze jest to możliwe ze względu na potencjalne różnice w ekspresywności języków zapytań. W niniejszej pracy zapytanie źródłowe jest interpretowane poprzez wykonywanie sekwencji elementarnych zapytań w języku docelowym.

3.3 Transformacja modeli

Doświadczenie z pierwszymi wersjami widoków na bazę wiedzy pokazało, że samo odwzorowanie modeli to często za mało. Schemat danych wynikający z ontologii, która była projektowana przede wszystkim z myślą o wnioskowaniu, nie zawsze odpowiadał potrzebom użytkowników bazy wiedzy. Widoki na bazę wiedzy zostały zaopatrzone w funkcję transformacji modeli, aby udostępniany model lepiej odpowiadał potrzebom logiki biznesowej aplikacji korzystającej z bazy wiedzy.

Podczas gdy odwzorowanie modeli operuje na bytach metamodelu (poziom MOF M2), to transformacja operuje na poziomie modelu (poziom MOF M1). Transformacja modelu definiuje byty modelu docelowego w terminach bytów modelu źródłowego. Dla przykładu koncept *Człowiek* w modelu docelowym może być zdefiniowany jako *Kobieta* lub *Mężczyzna*, gdzie koncepty *Kobieta* i *Mężczyzna* pochodzą z modelu źródłowego. Podobnie język SQL może być uznany za język transformacji, ponieważ można za jego pomocą zdefiniować nową wirtualną tabelę (widok bazodanowy), korzystając z tabel już istniejących.

W niniejszej pracy przyjęto, że najpierw dokonywana jest transformacja modelu ontologicznego, który następnie jest odwzorowywany na model docelowy. Dzięki takiemu założeniu wystarczający jest jeden język transformacji. Gdyby transformacja następowała po odwzorowaniu, potrzebne byłyby języki transformacji dla każdego modelu docelowego: relacyjnego, obiektowego itp. Niektóre modele posiadają swoje języki transformacji, lecz nie wszystkie. Model obiektowy posiada język transformacji OQL [15], lecz nie jest on wbudowany w języki programowania obiektowego.

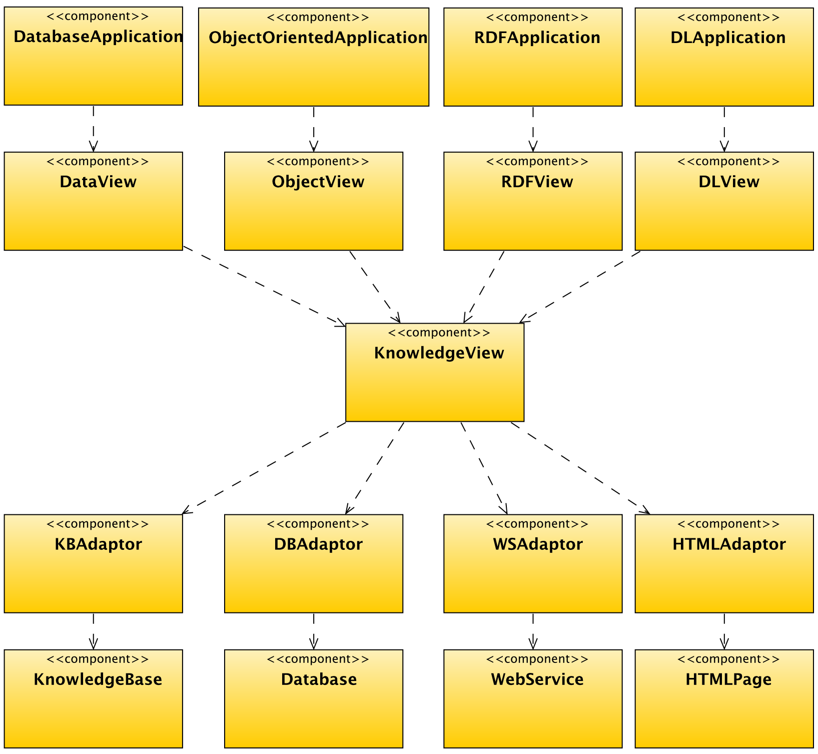
W ramach niniejszej pracy zaproponowano język transformacji dla ontologii zgodnych z logiką opisową, który jest oparty na regułach Horna. Język ten przypomina reguły z języka SWRL [48]. W ciele reguł możliwe jest stosowanie konceptów złożonych jako predykatów. Ponadto możliwe jest stosowanie predykatów wbudowanych, które umożliwiają między innymi wykonywanie operacji arytmetycznych.

3.4 Źródła wiedzy

Początkowo widoki na bazę wiedzy były tworzone z myślą o bazach wiedzy jako źródłach wiedzy. Źródło wiedzy było rozumiane jako źródło informacji z wbudowanym wnioskowaniem. Przyjęte rozwiązania pozwoliły jednak na uogólnienie, dzięki któremu zamiast źródeł wiedzy można było stosować źródła informacji, jak np. relacyjna baza danych, natomiast proste wnioskowanie zostało wbudowane w widoki na bazę wiedzy.

3.5 Ogólna architektura

Architektura widoków na bazę wiedzy jest warstwowa (patrz rys. 1). Na najniższym poziomie znajdują się źródła informacji, np. bazy wiedzy, bazy danych, usługi sieciowe itp. To, jakie źródła mogą zostać użyte, zależy od warstwy wyższej, która zawiera adaptory. Jeśli istnieje adaptor dla danego źródła informacji, to może ono zostać wykorzystane w połączeniu z widokami na bazę wiedzy. Architektura została pomyślana tak, aby łatwo można było napisać nowe adaptory dla nowych typów źródeł. Powyżej adaptorów umiejscowiona jest biblioteka Knowledge Views, która stanowi rdzeń całego rozwiązania. To ona udostępnia funkcje transformacji modeli, integracji źródeł informacji, prostego wnioskowania oraz modularyzacji ontologii. Powyżej biblioteki Knowledge Views umiejscowione są biblioteki Data Views, Object Views oraz RDF Views. Te biblioteki odwzorowują model ontologiczny na model relacyjny (Data Views), obiektowy (Object Views) oraz RDF (RDF Views). Architektura została tak pomyślana, aby możliwe było dodanie nowych bibliotek odwzorowujących na kolejne modele. Z bibliotek odwzorowujących korzystają bezpośrednio aplikacje. To, z której biblioteki odwzorowującej korzysta konkretna aplikacja, zależy od jej twórcy – wybierany jest ten model, który najlepiej pasuje do danej aplikacji.



Rysunek 1: Architektura widoków na bazę wiedzy

4 Studia przypadku i eksperymenty

Aby dowieść tezę pracy, zostały przeprowadzone studia przypadku oraz eksperymenty.

4.1 Symulacja z użyciem widoków na bazę wiedzy

Pierwsze studium przypadku pokazuje przykładowe wykorzystanie widoków na bazę wiedzy do symulacji rozpowszechniania się szkodliwego oprogramowania w sieci komputerowej. Studium to prezentuje poszczególne możliwości widoków na bazę wiedzy.

Jedną z możliwości widoków na bazę wiedzy, które zostały zaprezentowane, jest integracja kilku, w tym przypadku dwóch, źródeł informacji. Oba źródła to relacyjne bazy danych. Do jednego ze źródeł aplikacja ma uprawnienia do odczytu i zapisu, do drugiego ma tylko uprawnienia do odczytu.

Kolejną możliwością zaprezentowaną w tym studium jest zapisywanie informacji przez aplikację poprzez widok do źródła informacji. Ponieważ część informacji jest tylko do odczytu, twórca widoku decyduje, co powinno być zapisane, a co nie.

Zaprezentowana jest również możliwość modularyzacji wykorzystywanej ontologii. Modularyzacja ta umożliwia mieszanie różnych rodzajów wnioskowania – każdy moduł może wnioskować w inny sposób. W tym studium wykorzystywane jest oparte na regułach wnioskowanie w świecie otwartym (OWA) [4] oraz wnioskowanie zaimplementowane w języku programowania, które działa w świecie zamkniętym (CWA) [4].

Aplikacja korzysta z dwóch różnych widoków na tę samą ontologię. Każdy z widoków skupia się na innych szczegółach, chociaż pewne informacje są dostępne w obu widokach. Aplikacja jest napisana w obiektowym języku programowania i wykorzystuje bibliotekę Object Views do odwzorowania ontologii na model obiektowy.

4.2 Kompatybilność

W koncepcji widoków na bazę wiedzy bardzo ważną rolę odgrywa kompatybilność. Z jednej strony jest to kompatybilność z różnymi źródłami informacji, z drugiej strony kompatybilność ze współczesnymi aplikacjami. Aby to zobrazować, została napisana aplikacja korzystająca z relacyjnej bazy danych za pośrednictwem standardowej biblioteki Java Persistence API (JPA), implementującej odwzorowanie relacyjno-obiektowe. Następnie baza danych została zastąpiona przez widok relacyjny (Data View), pod którym ukryty był plik RDF zawierający te same informacje, co baza danych. Aby wykonać ten zabieg, niepotrzebne były żadne zmiany w aplikacji, jedynie zmodyfikowany został adres bazy danych w konfiguracji JPA. Dokładnie taka sama zmiana byłaby potrzebna przy zmianie producenta bazy danych. Dowodzi to, że widoki na bazę wiedzy są kompatybilne z aplikacjami bazodanowymi, ponieważ pozwalają zastąpić bazę danych bazą wiedzy bez modyfikacji aplikacji.

Została wykonana jeszcze jedna próba, w której we wspomnianej aplikacji bazodanowej zamieniono standardową bibliotekę JPA na bibliotekę Object Views, pod którą ukryty był plik RDF. W tym przypadku aplikacja wymagała drobnych zmian w miejscach, w których odwoływała się do biblioteki JPA. Odwołania te musiały być zastąpione odpowiadającymi im odwołaniami do biblioteki Object Views. Zachowana była jednak kompatybilność na poziomie języka zapytań – zapytania przekazywane do JPA i do Object Views były identyczne.

4.3 Stopniowe wdrażanie baz wiedzy

Ponieważ widoki na bazę wiedzy umożliwiają zastąpienie bazy danych bazą wiedzy, możliwe jest stopniowe wprowadzanie baz wiedzy do współczesnych systemów informatycznych w miejscach, gdzie do tej pory były wykorzystywane bazy danych. Nie chodzi tu o kompletne zastąpienie baz danych, ponieważ ciągle będą one pełniły rolę wydajnych źródeł informacji. Ponadto nie zawsze jest sens zastępowania bazy danych bazą wiedzy. Celem wprowadzenia baz wiedzy jest przeniesienie części wnioskowania wykonywanego w logice biznesowej aplikacji do bazy wiedzy, co znacznie upraszcza aplikację.

Aby zaprezentować, jak można wdrożyć bazę wiedzy w aplikacji, zostało przeprowadzone studium przypadku. W zaprezentowanym przykładzie została wprowadzona baza wiedzy, która wykonywała część wnioskowania, które w klasycznym podejściu wykonywane byłoby przez logikę biznesową. W tym przykładzie wykorzystano bibliotekę Data Views, dzięki czemu programista logiki biznesowej korzysta ze znanego mu interfejsu i języka zapytań SQL. Z punktu widzenia programisty baza wiedzy jest bazą danych, z której można pobrać informacje, które nigdy tam nie były umieszczane, lecz które można wywnioskować z innych informacji jawnie do niej wstawionych.

4.4 Łatwość użycia

W koncepcji widoków na bazę wiedzy ważną rolę odgrywa również łatwość użycia. Środkiem użytym, aby osiągnąć ten cel, jest implementacja przez widoki na bazę wiedzy standardowych interfejsów programistycznych (tak jest w przypadku biblioteki Data Views) oraz upodobnienie stworzonych bibliotek do już istniejących i powszechnie stosowanych rozwiązań (tak jest w przypadku biblioteki Object Views). Aby zbadać, czy faktycznie stworzone rozwiązania są łatwe w użyciu przez programistów, zostało przeprowadzonych kilka eksperymentów.

Pierwszy eksperyment został przeprowadzony z doświadczonymi programistami. Jego celem było sprawdzenie ile nakładu pracy jest wymagane, by zacząć używać biblioteki Object Views. Rozwiązania przyjęte w bibliotece Object Views były wzorowane na standardowej bibliotece Java Persistence API, którą testowani programiści znali. Zadanie polegało na napisaniu prostej aplikacji odpytującej bazę wiedzy poprzez bibliotekę Object Views. Jednym z elementów zadania było „ręczne" dostosowanie klas reprezentujących ontologię do wymogów biblioteki Object Views, mimo iż w skład Object Views wchodziło narzędzie, które pozwalało automatycznie wygenerować te klasy z ontologii. Ponadto programiści nie mieli dostępu do przykładowych kodów źródłowych, co sprawiło, że zadanie nie było zbyt proste. Dla porównania taką samą aplikację należało wykonać z wykorzystaniem biblioteki Jena [52], która służy do manipulowania plikami OWL. Po wykonaniu zadania programiści zostali poproszeni o uwagi dotyczące biblioteki, której mieli użyć. Programiści jednogłośnie stwierdzili, że wygodniejsza w użyciu jest biblioteka Object Views, ponieważ powstały kod był krótszy i napisanie go wymagało mniej mechanicznej pracy. Uczestnicy stwierdzili również, że możliwość użycia wspomnianego generatora klas uprościłoby zadanie. Ponadto stwierdzili, że posiadanie przykładowych kodów źródłowych również uprościłoby zadanie. Wśród uwag dotyczących testowanej biblioteki uczestnicy wymienili kilka drobnych różnic pomiędzy testowaną biblioteką a biblioteką Java Persistence API – jest to o tyle ciekawe, że pokazuje, że programiści mocno przyzwyczajają się do znanych im rozwiązań, więc warto nowe rozwiązania upodabniać do znanych, aby w ten sposób ułatwić proces przejścia na nowe rozwiązania. Różnice wymienione przez uczestników zostały usunięte w kolejnych wersjach biblioteki.

Kolejny eksperyment został przeprowadzony ze studentami informatyki. Jego celem było zbadanie, jak szybko niedoświadczeni programiści są w stanie zacząć używać bibliotek stworzonych w ramach niniejszej pracy. Testowana była biblioteka Data Views, która implementuje standardowy interfejs JDBC, oraz biblioteka Object Views, która jest wzorowana na standardowej bibliotece Java Persistence API (JPA). Studenci zostali dobrani w dwie grupy. Studenci z pierwszej grupy uczyli się już w ramach studiów o JDBC oraz JPA. Druga grupa nie miała takiego doświadczenia. Taki dobór grup miał na celu zbadanie wpływu doświadczenia z korzystania z wymienionych technologii na zdolność wykonania zadań. Studenci mieli 1,5 godziny na napisanie prostej aplikacji odpytującej bazę wiedzy poprzez Data Views oraz Object Views. Zgodnie z oczekiwaniami grupa studentów, która miała uprzednie doświadczenie z technologiami JDBC i JPA, lepiej poradziła sobie z zadaniem. Istotnym wnioskiem z przeprowadzonego eksperymentu jest to, że nawet niedoświadczeni programiści, jakimi byli studenci, są w stanie nauczyć się proponowanych rozwiązań w ciągu zaledwie kilku godzin, co oznacza, że proponowane rozwiązania są łatwe w użyciu.

5 Implementacja widoków na bazę wiedzy

Widoki na bazę wiedzy zostały zaimplementowane jako zestaw bibliotek: Knowledge Views, Data Views, Object Views oraz RDF Views.

5.1 Knowledge Views

Biblioteka Knowledge Views stanowi rdzeń całego rozwiązania. Biblioteka ta udostępnia klasy implementujące funkcje transformacji modeli, integracji źródeł informacji oraz prostego wnioskowania. W bibliotece tej zostały też zaimplementowane dwa przykładowe adaptory dla źródeł informacji. Jeden z adaptorów obsługuje relacyjne bazy danych, drugi pliki OWL i RDF poprzez bibliotekę Jena [52], która może działać jako baza wiedzy. Biblioteka Knowledge Views została zaprojektowana w sposób umożliwiający rozszerzenie zbioru funkcji oraz dostępnych adaptorów.

Możliwości wnioskowania zawarte w bibliotece Knowledge Views można przyrównać do rozwiązań takich jak QuOnto [90] i KL [103], ponieważ umożliwia wnioskowanie z informacji zawartych w zewnętrznych źródłach. Biblioteka Knowledge Views różni się od QuOnto i KL przede wszystkim tym, że nie wymaga, aby źródło informacji było relacyjną bazą danych. Kosztem ogólności jest jednak wydajność.

Istotnym elementem wchodzącym w skład biblioteki Knowledge Views jest język xNeeK. Jest to język, który pozwala definiować widoki na bazę wiedzy w sposób deklaratywny. Z jego poziomu można wykorzystać klasy zawarte w bibliotece Knowledge Views, które służą do integracji źródeł, transformacji itp. Aby obsłużyć transformacje modeli, xNeeK zawiera możliwość zapisania reguł transformacji, które zostały opisane w rozdziale 3.

5.2 Data Views

Biblioteka Data Views realizuje odwzorowanie ontologiczno-relacyjne. Implementuje standardowy interfejs dostępu do relacyjnych baz danych JDBC [2]. Aby połączyć się z bazą wiedzy za pośrednictwem tej biblioteki, wystarczy użyć standardowej klasy java.sql.DriverManager i podać URL w postaci:

```
jdbc:dv:neek:http://server.domain/xNeeKManifest.xml
```

lub

```
jdbc:dv:neek:file:///someDirectory/xNeeKManifest.xml
```

który wskazuje manifest xNeeK. Można też podać URL wskazujący bezpośrednio plik OWL:

```
jdbc:dv:file:/dev/shm/test.owl
```

Zaprezentowany powyżej sposób dostępu do bazy wiedzy tworzy widok niezmaterializowany. Biblioteka Data Views pozwala również tworzyć widoki zmaterializowane. Polega to na tym, że użytkownik podaje połączenie do bazy danych, natomiast biblioteka tworzy schemat odpowiadający ontologii i wypełnia go danymi. Ten rodzaj widoku pozwala na korzystanie z zaawansowanych możliwości relacyjnych baz danych, np. indeksów. W trakcie tworzenia widoku zmaterializowanego wnioski pochodzące z bazy wiedzy są również zapisywane w bazie danych, zatem proces wnioskowania jest jednorazowy. Podstawową wadą widoków zmaterializowanych jest to, że dane w nich zawarte starzeją się, co w niektórych zastosowaniach jest niedopuszczalne.

5.3 Object Views

Biblioteka Object Views realizuje odwzorowanie ontologiczno-obiektowe. Jest wzorowana przede wszystkim na bibliotece Java Persistence API. Klasy, które reprezentują ontologię, są zwykłymi klasami języka Java (ang. Plain Old Java Object, POJO). Nie muszą one implementować żadnego specjalnego interfejsu. Odwzorowanie w nich zawarte jest zapisywane za pomocą adnotacji. Klasy te można automatycznie wygenerować z ontologii. W trakcie działania programu odwzorowanie jest odczytywane z klas za pomocą mechanizmu refleksji.

5.4 RDF Views

Biblioteka RDF Views umożliwia wyeksportowanie informacji zawartych w ontologii do pliku RDF. Ponadto umożliwia odpytywanie bazy wiedzy za pomocą języka SPARQL. Biblioteka RDF Views skupia się na asercjach, a ignoruje terminologię. Mimo iż ontologie w języku OWL można zapisać w całości, łącznie z terminologią, jako pliki RDF, to bazy wiedzy, które są zakrywane przez RDF Views, nie muszą być zgodne z językiem OWL. Przez to wydobycie kompletnej terminologii z bazy wiedzy nie zawsze jest możliwe. Główną motywacją implementacji tej biblioteki jest fakt, iż RDF jest silnie wspierany przez środowiska związane z inicjatywą Semantic Web, więc biblioteka ta zwiększa kompatybilność tego rozwiązania z innymi rozwiązaniami Semantic Web.

5.5 DL Views

DL Views nie zostało zrealizowane jako osobna biblioteka, ponieważ jego rolę spełnia biblioteka Knowledge Views, która operuje na modelu ontologicznym zgodnym z logiką opisową.

6 Podsumowanie

Głównym efektem pracy jest zdefiniowanie koncepcji widoków na bazę wiedzy przedstawionej w rozdziale 3. Widoki na bazę wiedzy zależą od procesu odwzorowania modeli oraz transformacji modeli, które również są przedstawione w rozdziale 3. Poza przedstawieniem ogólnej idei odwzorowania modeli zostały zdefiniowane reguły odwzorowania ontologiczno-relacyjnego oraz ontologiczno-obiektowego. Zostało również przedstawione przyjęte rozwiązanie realizacji odwzorowania języków zapytań, które jest silnie związane z odwzorowaniem modeli. Poza przedstawieniem idei transformacji modeli zaprezentowany został opracowany regułowy język transformacji. Rozdział 3 kończy się prezentacją ogólnej architektury widoków na bazę wiedzy. Dwie najważniejsze warstwy w zaproponowanej architekturze odpowiadają odwzorowaniu modeli oraz transformacji modeli.

W rozdziale 4 zostały zaprezentowane studia przypadku, które ilustrują łatwość integracji widoków na bazę wiedzy z istniejącymi technologiami, w szczególności z relacyjnymi bazami danych oraz programami napisanymi w obiektowym języku programowania, jakim jest Java. Ponadto zostały przeprowadzone eksperymenty z doświadczonymi oraz początkującymi programistami. Wnioski z przeprowadzonych eksperymentów zaprezentowane są w rozdziale 4. Eksperymenty z doświadczonymi programistami pokazały, że programiści przyzwyczajają się do konwencji wykorzystywanych w znanych im rozwiązaniach – uczestnicy zauważyli nawet drobne różnice pomiędzy pierwszą implementacją widoków na bazę wiedzy a rozwiązaniami, które znali. W kolejnych wersjach widoków na bazę wiedzy różnice te zostały usunięte, aby zwiększyć poziom kompatybilności z istniejącymi rozwiązaniami. Eksperymenty z początkującymi programistami pokazały, że widoki na bazę wiedzy są na tyle proste w użyciu, że studenci informatyki są w stanie je poznać w zaledwie kilka godzin.

Koncepcja widoków na bazę wiedzy została zrealizowana jako zestaw bibliotek programistycznych. Implementacja widoków na bazę wiedzy została opisana w rozdziale 5 oraz w dodatkach od D do F. W skład implementacji wchodzą biblioteki: Knowledge Views, Data Views, Object Views oraz RDF Views. Biblioteka Knowledge Views jest odpowiedzialna za transformacje modeli, które mogą być zapisane za pomocą języka xNeeK, który został opisany w rozdziale 5 oraz w dodatku C. Biblioteka Data Views realizuje odwzorowanie ontologiczno-relacyjne, a biblioteka Object Views odwzorowanie ontologiczno-obiektowe. Biblioteka RDF Views zapewnia kompatybilność rozwiązania z modelem RDF. Biblioteka Knowledge Views umożliwia ponadto dostęp do bazy wiedzy zgodny z logiką opisową.

W ten sposób cele określone w rozdziale 1 zostały osiągnięte, tzn.:

1. Odwzorowania modelu ontologicznego na model obiektowy, relacyjny oraz RDF zostały zdefiniowane i zaimplementowane jako biblioteki Object Views, Data Views oraz RDF Views.

2. Dwa zestawy transformacji modelu ontologicznego zostały zdefiniowane. Transformacje pojedynczego modelu zostały zapisane jako regułowy język transformacji, natomiast możliwości łączenia kilku modeli zostały zrealizowane jako zestaw klas w bibliotece Knowledge Views. Oba zestawy transformacji mogą być wyrażone w języku xNeeK.

Idea widoków na bazę wiedzy została zrealizowana, zaimplementowana i poddana walidacji. Przeprowadzone studia przypadku oraz eksperymenty dowodzą postawioną tezę, że widoki na bazę wiedzy pozwalają na łatwe wykorzystanie baz wiedzy we współczesnych systemach informatycznych poprzez udostępnienie warstwy zapewniającej kompatybilność z istniejącymi systemami oraz poprzez udostępnienie prostego w użyciu interfejsu programistycznego.

Pierwsze implementacje bibliotek Data Views oraz Object Views, a zatem częściowe efekty niniejszej pracy, zostały wykorzystane w projekcie PIPS (Personalised Information Platform for Life and Heath Services [89]) finansowanym ze środków Komisji Europejskiej w ramach 6. Programu Ramowego. Biblioteka Data Views została wykorzystana, aby zwiększyć możliwości odpowiadania na zapytania kierowane do bazy wiedzy, która była rdzeniem systemu zarządzania wiedzą (KMS) [41]. W szczególności zapytania wymagające założenia o świece zamkniętym były obsługiwane przez bibliotekę Data Views. Biblioteka Object Views stanowiła warstwę pomiędzy systemem zarządzania wiedzą a portalem internetowym. Jej rolą było ułatwienie korzystania z bazy wiedzy.

Reasumując, do głównych osiągnięć rozprawy należą:

- zdefiniowanie koncepcji widoków na bazę wiedzy i zaproponowanie rozszerzalnej architektury widoków na bazę wiedzy,
- zdefiniowanie odwzorowań ontologiczno-relacyjnych i ontologiczno-obiektowych,
- zdefiniowanie regułowego języka transformacji i zaimplementowanie go w postaci języka xNeeK,
- zaimplementowanie koncepcji widoków na bazę wiedzy w postaci bibliotek: Knowledge Views, Data Views, Object Views i RDF Views,
- poddanie koncepcji widoków na bazę wiedzy walidacji poprzez przeprowadzenie eksperymentów z doświadczonymi i początkującymi programistami.

6.1 Wady i zalety rozwiązania

Poza kluczowymi cechami zaproponowanego rozwiązania, jakimi są łatwość użycia oraz kompatybilność, widoki na bazę wiedzy mają kilka innych istotnych zalet. Jedną z nich jest ogólność. Ogólność może być tu rozumiana na kilka sposobów. Po pierwsze metody opisane w niniejszej pracy mogą być zastosowane do stworzenia podobnych bibliotek programistycznych w językach programowania obiektowego innych niż Java. Ogólność ma zastosowanie również w stosunku do źródeł informacji, tzn. widoki na bazę wiedzy mogą współpracować z różnymi źródłami informacji, nie tylko z takimi, które implementują interfejsy SQL lub SPARQL. Ogólność jest także powiązana z rozszerzalnością. Widoki na bazę wiedzy mogą zostać rozszerzone o kolejne typy widoków dla innych modeli. Ponadto zbiór dostępnych transformacji również może zostać rozszerzony w miarę potrzeby.

Wspomniana ogólność i rozszerzalność pociąga za sobą koszty wydajnościowe. Coś za coś – zazwyczaj możliwe jest uzyskanie albo ogólności i rozszerzalności, albo wydajności. Alan J. Perlis w swoich epigramach [84] napisał:

> „Optymalizacja utrudnia ewolucję.”

Sama koncepcja dodania warstwy nad bazą wiedzy musi pociągać za pogorszenie wydajności. Jednakże, jeśli wydajność w praktycznych zastosowaniach okaże się niewystarczająca, możliwe jest zastosowanie technik optymalizacyjnych znanych z różnych implementacji biblioteki Java Persistence API lub bibliotek podobnych.

6.2 Możliwości rozwoju

Widoki na bazę wiedzy można dalej rozwijać, np. można dodać obsługę transakcji, aby widoki na bazę wiedzy nadawały się do wykorzystania w środowisku wieloużytkownikowym. Można również polepszyć możliwości wnioskowania zawarte w bibliotece Knowledge Views. Możliwy jest również rozwój dostępnego zestawu transformacji, aby móc dodać obsługę niespójnych źródeł informacji – jest to dość istotna funkcja w świecie internetu, w którym występuje duża liczba źródeł podających sprzeczne informacje. Warte rozważenia jest rozszerzenie możliwości widoków o obsługę predykatów n-arnych – w pewnych przypadkach mogłoby to pozytywnie wpłynąć na wydajność. Widoki na bazę wiedzy były projektowane od początku z myślą o rozszerzalności i stanowią dobry fundament do dalszego rozwoju metod integracji świata baz wiedzy ze światem klasycznej inżynierii systemów.