STI · INNSBRUCK

Semantic Technology Institute (STI) Innsbruck

WSML Logical Expression Templates

by
Werner Bliem

Supervised by Dr. Michal Zaremba

Bachelor Thesis

Submitted to the

Department of Computer Science
University of Innsbruck
Austria

June 2008

Abstract

The goal of this thesis is to create a mechanism and tools to make the querying of WSMO ontologies easier for non-experts. The elaborated solution should enable users to focus more on their domain-specific problems and to care just a little or not at all about the underlying concepts and the formalism of the WSML query language.

To achieve this aim, we took a template-based approach. At the heart of this approach lies the template-language WSLET (WSML Logical Expression Templates) that will be defined, and used together with a natural language description, to create a WSML query template. This query template can then be interpreted by a graphical user interface and presented to a user in an intuitive, form-like manner. The interface should also assist in filling in the right values, so one can finally execute a pragmatic query.

Contents

Chapter 1

Introduction

Sophisticated information systems often face the problem of making their functionalities accessible to non-experts. This becomes even more obvious when the usage of the system depends on the understanding of information-theoretical concepts. For some applications this aspect may not be of great importance, especially if the audience of non-IT-experts is very small. Modern knowledge systems, however, are one type of application that has to take this aspect very serious. Their functionality is almost always based on logics, while their audience is expected to be mainly non-IT-experts – people working in different areas of research and our everyday life, who want to benefit by using them.

The usage process of knowledge systems consists, in coarse, of its *design*, the *creation of data* and the *querying of data* (or other kinds of data processing like planning tasks). Especially the latter two activities have to be supported by intuitive tools so people of every background can use the functionalities of these systems at their full potential.

1.1 The need for an easier query mechanism

When it comes to the querying of data, the problem of the underlying logics becomes very obvious. The native language for querying is hardly ever based on a simple formalism, whose semantics are very intuitive, but mostly on complex syntactical constructs whose understanding depends on the knowledge of concepts, which can not be easily grasped without the study of logics and formal languages. However, the users of a knowledge system want to think about their domain-specific problems, and not about learning the details of its implementa-

tion, in order to be able to use it. So, the question is, what can be done to make the querying task easier and, in particular, what do people want to use for it?

People like to use what they are used to. When it comes to searching information in IT-systems they are used to search-forms. For the ordinary user, querying data-sets is hidden behind a mask where he enters keywords and/or values. The special quality of ontology-based information is its association with a *context*, that allows for e.g. its classification or deduction of 'hidden' data. This context is almost always created by using some sort of logics. But the concepts of logics, that associate semantics with information, are only intuitive on an informal level. Their exact and sound use, which includes learning the formalism to represent them, is everything else than trivial. This thesis presents a solution, that tries to overcome these difficulties when querying data, by taking a template-based approach.

1.2 A simple usage scenario

To see the advantages of this solution and to think about possible alternatives, let's consider the following scenario. Information experts and non-IT-staff work together in a company that is big enough to have it's own IT-department. The company is very progressive and uses a knowledge system for managing its data. They have a couple of ontolgies that are tailored to their professional domain(s), and some employees create data according to this model. However, the vast majority of the staff, as well as customers and business partners, want to use the system to obtain information. Unfortunately, only some of them have the knowledge to create queries using the native formalism. The others cannot use the system directly, but have to ask others for help. Especially our IT-staff get's bothered day in and day out with questions. For them it's always the same tiring and time consuming process. To illustrate it, let's consider the following example.

We assume the company has an ontology that covers different aspcets of human resource management (HRM). This ontology is also thightly connected with an accounting ontology. For statistical reasons an employee of the HRM-department wants to know the percentage of business travels that went overseas last year. To do so the knowledge system has to be queried. However, since the employee is not trained in posing queries to the knowledge system, he turns to the IT-departement, having the confidence that they'll get an answer to the question.

Our system experts receive the employee's human language request and have to translate it into the syntactical formalism the system understands. (Every now and then they also receive a request that is not formulated precisely enough, so they have to turn back to the requester to get additional information.) After the system returns the results of the query, one of the computer scientists delivers them back to the person that asked for it. In our example the results will then be used to create statistics for an annual report.

However, both, users and in particular experts, think that the additional communication effort is tedious. So, what could be done to help them out?

One simple approach would be to predict the most common queries and to provide some interface to the user, where he can choose one from a list. This list would most probably be very large and would have to be updated frequently. These updates can result from either new user request or, what's even worse, when the conceptual model changes. This concretely means, that many predefined queries cannot be used for different ontologies, and also that we provide the user only a very restricted choice. So, this predefined queries approach has some enormous downsides. At the heart of the problem lies the non-existing flexibility of this solution. It's very obvious that this approach is neither helpful for the IT-staff nor for the rest of the crew. So, what are the alternatives?

Well, we could use what we're already used to – that is the specification of values to find the information we need, just like in traditional (relational) database queries.

But in contrast to traditional database systems, that are used for a highly performant way of managing data, the interrelations of our data is more complex in logic based knowledge systems. We have data-objects (instances), that are associated with constraints and rules that allow us to deduce new knowledge. Special constructs (called axioms), created by using logical operators, define these constraints and rules. These operators along with others also allow us to search for objects. So, native queries in these systems consist of formal expressions that allow us to consider objects and values and some special relationships they are involved in.

This insight leads us to another approach that could make the life of our information experts easier – provide some kind of coarse structure that makes up the query, and let the user fill in the values and objects to make the search concrete.

At this point one may wonder about some things. First, what is meant by

structure? We will discuss this shortly when talking about templates. Another question might be, why not making a system that makes it easy for the user himself to create a valid structure including the values, i.e. a valid native query. Though this consideration sounds interesting, one has to keep in mind that the possible derivations of a complex formal language can be almost – for human scales – infinite. Additionally, it would be a really hard job to predict what the user has in mind to offer him the most appropriate alternative. Thus, the pragmatic use could be a really difficult thing to achieve for the user.

Last but not least, our IT-staff does not become redundant thanks to the fact that they'll have to create a new structure from time to time. Now, the structure combined with the placeholders for the values and objects, is what we call a template. And templates will be the focus of interest in the next section.

1.3 Templates – A proven mechanism

Templates are nothing new. Templates have a long history of usage, ranging from the beginning of living nature to the almost omni-presence in human culture. Especially in engineering and manufacturing they became an essential aid for the creation of things and execution of tasks.

Human beings are very much used to templates. We encounter them daily – whenever we fill out a form (for applications, contracts etc.), whenever we follow a routine in our working instruction, e.g. when we bake cookies. The list of situations in which we fall back on some sort of templates is almost infinite. However, templates are not an invention of mankind. Templates are actually as old as life itself. Our cells are full of templates, enabling them to produce every complex molecule we need to live.

Templates are a way to *carry information* and they are also a way to *abstract information* from the user of the template. They enable us to create instances that have a specific form and qualities. By the help of templates we can create them very quickly and without the need to care about every detail. At the same time there are types of templates flexible enough to adapt the needed instance to a specific situation.

These are exactly the properties our user-friendly query system should have. The user wants to quickly create a query that gives him useful information. Essentially he simply wants to fill in a form. The only thing he wants to know, in

order to use a template, is what it's good for. Thus we have to take care that the semantics of the template is correctly propagated to the user. Indeed, templates almost always come with instructions in the world of humans. In our case, we will add a *natural language* description of the inherent properties of our templates, so the user will know which one to choose, and will be able to fill in the values to adapt it to his specific situation.

Since we want to create templates for WSML Logical Expressions we'll have to analyze this subset of the WSML language in order to find a reasonable approach for how our templates should look like. And for sure this approach will then have to be formalized. It suggests itself to call the resulting template-language *WSML Logical Expression Templates*, or short *WSLET*.

Chapter 2

Background

In this chapter we will briefly discuss the Web Service Ontology (WSMO) and the Web Service Modeling Language (WSML), since they're the fundamental building blocks of the template language WSLET that will be defined in chapter 3. We will further talk about template processing, which is the underlying mechanism of this template approach for WSML queries. Last but not least we will also mention the concrete use case that was the initial motivation for this work.

2.1 WSMO and WSML

The *Web Service Modeling Ontology* [5] (WSMO) is a conceptual model that is intended to be used for the semantical annotation of Web Services, thereby creating *Semantic Web Services* (SWS). The idea behind SWS is to create an automatic integration technology that is based on semantic information. This concretely means that tasks like the discovery and composition of web services is done automatically in order to achieve a user-specified goal. WSMO tries to cover all the relevant aspects that are considered to constitute the core-elements of Semantic Web Services. It further comprises the specification of ontologies as a data-model, that is fundamental regarding the other elements reliance upon them.

For this thesis, WSMO ontologies are of particular interest, because its aim is to make the querying of these ontologies easier for non-IT-experts. The querying of WSMO ontologies is normally done by using WSML Logical Expressions [6, Sec. 2.8] (WSML LE). WSML LE are part of the *Web Service Modeling Language* [4] (WSML) which is the formal language of WSMO. WSML not only provides a syntax for the description of web services, it also defines the formal semantics of the

language elements based on logical formalisms, i.e. *Description Logics*, *First-Order Logic* and *Logic Programming*. Therefore, WSML comes in 5 different language variants that allow for a trade off between the expressiveness of the description and the complexity and decidability of inferences that might result for an implementing application. Further, there are different specification forms and syntaxes supported by WSML, the most noticeable of which might be the *human readable syntax*. Nevertheless, there exists the possibility of representing the annotation in XML, WSML/RDF, as well as a mapping between WSML-Core and OWL-DL descriptions.

2.2 Template processing

The underlying mechanism of the template approach for WSML queries is based on template processing – a widely used technique in software engineering. We will therefore shortly illustrate the basics of this technique before talking about the details of WSML Logical Expression Templates. After all, this is what brings us the benefit of easily creating WSML queries.

Even though template processing is used in a wide range of contexts, the most notably of which may be the dynamic generation of web content, there are underlying concept and processing steps that are almost always the same. Figure 2.1 tries to depict the major entities and their connections, that are involved in template processing.

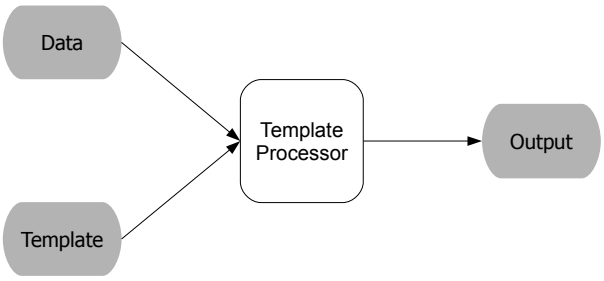

Figure 2.1 Abstract view of a template system

Its not very hard to see what a template processor, often referred to as template engine, does. In one sentence – it simply combines a template and data, from an associated data-model, to generate some sort of meaningful output. That

is the easy and abstract view of a template system. Depending on its concrete context of application there may be of course great differences in where the data comes from, what information the template carries and what actions are performed within the template engine.

In the following we will briefly discuss how things are working for WSML query templates and depict it in a big picture.

2.2.1 Processing query templates

Figure 2.2 shows the big picture of the template-based approach for WSML queries.

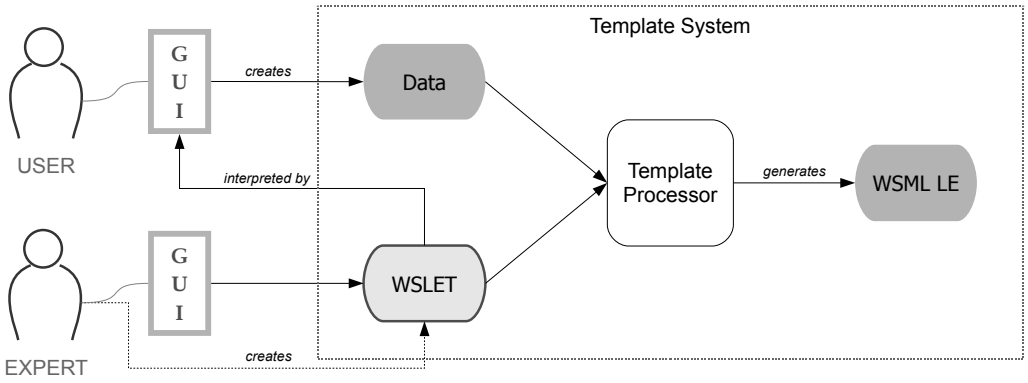

Figure 2.2 The big picture of query templates

At the heart of this approach lies WSLET. As we will see in chapter 3, WSLET is a template language that combines WSML Logical Expressions with typed placeholders, called *template-variables*. These typed placeholders are like the fields of a form that are intended to be filled in by a user. Thus the user provides the data for the template processor. However, to abstract the details of the data-model that is required by the template engine, the user interacts with a GUI, which brings the data in the right form. So, more precisely, in our system a GUI is intended to be the data-provider. The template engine will then take the WSLET and the data as an input and perform a kind of macro-substitution, in a way similar to a pre-processor. The result of this substitution will be a WSML Logical Expression (WSML LE) – our query – that can then be sent to the reasoner to obtain some useful information.

Template processing is actually the last step in the life of a query template. Fortunately, templates can be reused, so they only pass away when they're deleted. However, templates only exist after they're created.

2.2.2 Creating query templates

The being of a template demands the being of its creator. The creator of query templates is supposed to be an information expert, who is familiar with WSLET. In addition to creating a WSLET he should also provide a natural language description, that is intended to mediate the inherent semantics of a query template to a user. To make his life easier and the creation process of templates less error prone, he most probably wants to be supported by tools for these tasks, though they're not necessary.

2.2.3 Using query templates

Query templates are designed from the point of view of a user. As already mentioned they are intended to make the querying of a WSMO ontology, specified in WSML, easier for a non-IT-expert. Since humans tend to be attracted by nice graphical representations, one could say that query templates are by nature supposed to be used in connection with a graphical user interface. Therefore, a suitable graphical user interface should fulfill two purposes:

- Presenting the query template to the user in an intuitive manner –
  The GUI has to interpret the WSLET and combine it with the natural language description in order to achieve an intuitive representation for an end-user.

- Creating the input data for the template processor –
  For this requirement the GUI has to provide means to make it easy for a user to specify values, and further take care that this input is translated to the data-model the template processor accepts.

2.2.4 The benefit of query templates

To better understand the benefit query templates are supposed to bring, let's consider the following example. Assume we have an ontology that has a concept

called *person*. Such an ontology may be used e.g. in a company with a certain amount of employees. Among others, the concept person has an attribute that takes a person's first name as an argument – and may therefore be called *hasFirstName*. Knowing the first name of a person is very important and there is a good reason why. People who have name day get chocolate from the human resource department. So, if the staff of the HR-department knows who has name day today, they also know who they'll treat with chocolate. Under the assumption that today is "John", the following query may be of great value:

?x[hasFirstName **hasValue** "John"] **memberOf** person

There may be e.g. two results: 'John Q.' and 'John Doe'. So, these guys are the lucky ones today.

Leaving cacao beans aside, we also just saw an example of a WSML Logical Expression a user currently has to write, to query a WSML-specified ontology. For a non-IT expert this query says something like: "Give me all persons with the first name John". An expert on the other hand might also read it like this: "Give me all instances of the concept person that have an attribute hasFirstName with the value John". He may read it like this because he is aware of the underlying concepts. This awareness also allows him to create query templates. He could for example create a template that allows searching persons by their first name. To achieve this he would create a WSLET, along with a natural language description, that looks like this:

?x[hasFirstName **hasValue** **$**{*attributevalue*}] **memberOf** person

From this WSLET one can see two things. First of all, the name "John" was substituted by a template-variable (**$**{*attributevalue*}). As already mentioned, a template-variable serves as a placeholder for value(s) that may be filled in by a user that wants to execute the query template. The second thing, we can see, is that the template uses the local name of a concept (person) and an attribute (hasFirstName). Therefore, this template is not ontology-independent, and may only be usable for particular ontologies. We won't elaborate this point but leave it mentioned here.

Getting back to our example, let's shortly describe what could happen now with a query template. Since it contains a WSLET and a natural language description a GUI can now generate a nice and intuitive representation of it. It may look like the sentence below.

Give me all persons that listen to this name.

As we can see, the template-variable has become a link that may lead a user to an interface, where he can fill in or choose values. Whenever a user changes that value of a template-variable, it may trigger the template processor to generate a query that can finally be sent to the parser. This is just an example of how the representation of a query template might look like. Important is the fact that it's intended to be created from the WSLET and the natural language description. How WSLET looks like will be discussed in chapter 3.

2.3 SUPER – A real world context

The motivation of this work rose from a concrete context of application.

"*Business Process Management focuses on managing the execution of IT-supported business operations from a business expert's process view rather than from a technical perspective. The underlying motivation for BPM is that organizations need to continuously align their running business processes, as executed within multiple heterogeneous systems, with the required processes as derived from business needs. BPM has gained significant attention in both research and industry, and a range of BPM tools are available. However, the degree of mechanization in BPM is currently very limited. The major obstacle preventing a coherent view on business processes is that the business processes are not accessible to machine reasoning. Additionally, businesses cannot query their process space by logical expressions, e.g. in order to identify activities relevant to comply with regulations.*"[1]

"*The major objective of SUPER is to raise Business Process Management (BPM) to the business level, where it belongs, from the IT level where it mostly resides now. This objective requires that BPM is accessible at the level of semantics of business experts ... Therefore this project aims at providing a semantic-based and context-aware frame- work, based on Semantic Web Services technology that acquires, organises, shares and uses the knowledge embedded in business processes within existing IT systems and software, and within employees' heads, in order to make companies more adaptive.*"[3]

The WSMO approach[5] is the used Semantic Web Service technology of this project. Among many others, one of the scientific objectives of SUPER is the "*development of process query tools*"[2].

For sure there already exist tools for querying WSMO ontologies, however, their usage is based on writing WSML Logical Expressions. As motivated in section 1.1 and 1.2 this way of querying may not be appropriate for end-users, who want to focus on their domain-specific problems. While the context of SUPER is business process management, the querying approach presented in this document is flexible enough to be used in many other scenarios.

Chapter 3

WSML Logical Expression Templates

WSML Logical Expressions are a way to define axioms in WSML [4]. These axioms are written in a logical formalism and are used to define the rules and constraints of properties and objects. The rules and constraints in turn make it possible to deduce new, called hidden, knowledge. So, logical expressions may be used to define an ontology, but they are as well used to query an ontology.

As mentioned earlier, such a logical expression has the form of objects and values that are connected by logical operators. Actually, an object itself is just a collection of values, that are interrelated with each other in a more or less complex way. Thus it's an abstraction referred to by a name, or more precisely spoken, by an identifier.

So, from a syntactical point of view, we can try to see a logical expression in a way like an arithmetic expression – values (or operands) connected by operators. And just like in arithmetical expressions these values may not be fixed. This means there may be several values for a part of an expression, to make that expression valid. Quiet obvious, we're talking about *variables* – and variables are, in a majority of the cases, the reason we pose a query. Therefore, they give us a first hint where to place our 'fields in the form'.

In this section the reader will encounter several tables that are used to display production rules of either the WSML or WSLET grammar. The used symbol names for non-terminals an tokens are the ones of the respective SableCC grammar specification. To make the reading of the rules easier, tokens are written **bold face** while non-terminals are written *slanted*.

3.1 Analyzing WSML Logical Expressions

When we take a look at the grammar specification of WSML [6, App. A.1], more precisely, on the production rules that define the logical expression subset of WSML, we first notice rules that represent a precedence hierarchy for various logical connectors.

Table 3.1 Production Rules Excerpt of the WSML Grammar (1)

Nonterminal		***Rule(s)-Components***
log_expr	=	[head]: *expr* **t_implied_by_lp** [body]: *expr* **endpoint** | **t_constraint** *expr* **endpoint** | *expr* **endpoint**
expr	=	*expr imply_op disjunction* | *disjunction*
disjunction	=	*conjunction* | *disjunction* **t_or** *conjunction*
conjunction	=	*subexpr* | *conjunction* **t_and** *subexpr*
subexpr	=	**t_not** *subexpr* | *simple* | **lpar** *expr* **rpar** | *quantified*
quantified	=	*quantifier_key variablelist* **lpar** *expr* **rpar**

With minor modifications taken from: Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/#grammar:log_expr>.

We can find out for example that the head of a logic programming rule is implied by its body. The head and the body in turn consist of an expression. We see that within such an expression we have the following derivation hierarchy (which is the reverse of the operators precedence): implication, disjunction and conjunction. We can also see that we may have negated or quantified expressions as well as even more complex groupings between parentheses. This is just a very cursorily description of what can be seen in Table 3.1, which not only gives an example on how to read such a grammar specification, but also shows the grammatical entry point of WSML LE. However, these rules are not the focus of interest in this chapter.

3.1.1 Terms and values

After we saw how an expression can be build or connected by some well known logical operators, we notice a more interesting rule. It's called simple and can be seen in table 3.2.

Table 3.2 Production Rules Excerpt of the WSML Grammar (2)

Nonterminal		***Rule(s)-Components***
simple	=	*molecule* | *comparison* | *term*

With minor modifications taken from: Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/#grammar:simple>.

What makes this rule interesting is that we encounter the notion of a *term*. If we would look a bit further in the grammar, we would see that a term can be a value, a variable or something called a nb_anonymous (numbered anonymous identifier). A value in turn can be an id (IRI or anonymous identifier) or a one of the build-in values, which is a number or a string (see Table 3.3).

Table 3.3 Production Rules Excerpt of the WSML Grammar (3)

Nonterminal		***Rule(s)-Components***
term	=	*value* | **variable** | **nb_anonymous**
value	=	*functionsymbol* | *id* | *number* | **string**

With minor modifications taken from: Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/#grammar:term>.

It must be remembered that *values* are what we want our user to specify. So, we reached a point in the WSML grammar, where we have something we can work with. We could say now – okay, let the expert build the structure of the query by defining how terms should be connected with all kinds of operators and

let the user decide what a term should look like. This is actually the base idea we will follow, however there are further details we have to take into account.

So, a term can be many things, but as we will see, the logical operators may only allow certain things to stand between them. What we need is a mechanism that allows us to choose what the term should be (a variable – any value – or a specific value), and that may even check if it is appropriate at this place in the query. This sounds pretty much like *language semantics* and we'll keep it in mind for some later discussion.

3.1.2 Molecules

At this point we have an idea what we would like our placeholders for the 'form fields' to be. They should be an alternative to terms that let the user decide what 'value' the term will take. To get a better understanding of how terms may be used let's further investigate the WSML grammar.

The next production rule tells us which token-sequence builds a thing called *molecule* (see Table 3.4). The rule contains some interesting nonterminals that stand for a special kind of operators. One is called *cpt_op* and it actually stands for the tokens **t_memberof** (*memberOf*) and **t_subconcept** (*subConceptOf*). We find the other operators by tracing the nonterminal *attr_specification* that eventually brings us to the tokens **t_hasvalue** (*hasValue*), **t_oftype** (*ofType*) and **t_impliestype** (*impliesType*). In a query these operators can be used to give us a set of objects that fulfill special properties.

Table 3.4 Production Rules Excerpt of the WSML Grammar (4)

Nonterminal		***Rule(s)-Components***
molecule	=	*term attr_specification? cpt_op termlist* | *term cpt_op termlist attr_specification* | *term attr_specification*

With minor modifications taken from: Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/#grammar:molecule>.

It is very important to emphasize the existence of these operators since the definition of all the abstract and concrete objects (concepts and instances) combined with the relationships between these objects (attributes and relations) are at the heart of modeling ontologies. This is achieved by the use of these oper-

ators and that's why they are so important and interesting. Another thing that makes them interesting is the fact, that, syntactically, they can be used between any term, but semantically there are of course restrictions what can stand before and after such an operator. An understanding of these operators is essential for the creation and querying of ontologies. That's why we'll take a closer look at what they stand for and how they're used.

3.1.3 Some special operators

By the modeling of concepts we aim at creating an abstract description of the objects of our world and thoughts. Such a description usually includes a classification of the concept itself, along with the properties that are in the focus of interest of our objects. We can then relate specific instances with the concepts of our model, and assign concrete values.

The keyword *subConceptOf* gives us a way to classify concepts, i.e. to bring them into some hierarchical relation, also called subsumption relation. The operator is really self explaining, so *A subConceptOf B* means that "concept A is a sub-concept of the concept B".

When we bring our ontology to 'life', we create instances according to the conceptual model that we call our ontology. It's a bit like inserting a data-set into the tables of a relational database. However, when we create an instance we can also state to which concept in our ontology it belongs by making use of the operator *memberOf*. *I memberOf C* means that "instance I is a member of class C".

Till now, the relations we made in our ontology, were only of classifying nature. There may be, however, other properties of an object that are at least as interesting. When we want to define these properties on the conceptual level we talk about *attributes* and *relations*. For both we need to define the range of their values, which can be a datatype or another concept. For this range definition there are two operators called *ofType* and *impliesType*. The first one is restrictive in the way that if you specify a value for e.g. an attribute it must be of the defined type. It is not allowed to specify a value of a different type for this attribute. The latter infers that if you have an attribute value it must be of the defined type.

Just like for the membership relation, when being on the level of instances, we want to assign values to our attributes/relations. We do this by making use of the keyword *hasValue*. The meaning of this operator is very straightforward,

e.g. *A hasValue X* means that "attribute A has the value X". And that's already it!

So far we only talked about the definition of ontologies, but fortunately querying makes use of exactly the same operators. And it's easy to imagine how it works – just take a look at the following example:

?x **memberOf** C and ?x[A **hasValue** Y]

This WSML logical expression represents a query that says: "Give me all instances that are a member of concept C and whose attribute A has the value Y".

Getting a good feeling for these operators is definitely one of the most important things to successfully use WSML Logical Expressions. However, there are many other operators and formalism that are equally important that have been left out at this place. Without a certain knowledge it might be time-consuming for the normal user to learn such a formalism. For this reason it makes sense to hide this level of detail from the user and to give him the means to fill in the interesting gaps an expert can define by creating a query template.

3.2 Thinking about placeholders

We already saw that a logical expression consists of an hierarchical construct that is created by combining operators with 'values'. In this section we want to get a bit more precisely of what these 'values' can be.

3.2.1 What do terms abstract?

On the syntactical level we talk about terms at this level of abstraction. Terms are what glue together the operators and vice versa. So terms actually stand for operands. A term again is itself an abstract concept. When we take a look at the grammer (see Table 3.5), we see that a term can be on of *value*, **variable** or **nb_anonymous** (numbered anonymous identifier).

From the syntactical point of view it's very easy what a **variable** and **nb_anonymous** is. To keep it brief, the first is a question mark followed by some alphanumerical characters (e.g. ?x2) while the latter is is something that starts with _# and is followed by some digits. A value is just a bit more complex – it can be a *functionsymbol*, an *id* a *number* or a **string**. The latter two are very straightforward to understand (they actually represent the basic build in types in

Table 3.5 Production Rules Excerpt of the WSML Grammar (5)

Nonterminal		***Rule(s)-Components***
term	=	*value* | **variable** | **nb_anonymous**

With minor modifications taken from: Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/#grammar:term>.

WSML - strings, integers and decimals). For how ids should look like, one does best by consolidating the WSML Language Reference [6]. For now lets consider them as some kind of hopefully unique names. What's left are functionsymbols that are syntactically an identifier followed by parameters.

So, when we try to summarize all those complicated levels of abstraction, a term will be either an identifier, an identifier that has some parameters, a variable, a string or an arithmetical expression.

Knowing how things should look, to be usable for filling gaps, is one part of the deal. What's still unanswered is what those things are and where they can be used in a meaningful way. Hence, let's talk about the semantics of our placeholders.

3.2.2 The need for typed placeholders

We start with contemplating the meaning of identifiers. Usually things get identifiers or names so we are able to talk about them, or technically speaking, to reference them, at some later point. This concept is familiar to every programmer. In the world of ontologies we use identifiers to define concepts, attributes and relations as well as instances and relationinstances. Query languages also use identifiers to be able to reference particular entities. But further we may be interested in a set of identifiable objects, that fulfill our requested properties. We denote our interest to find such things by writing a *variable* at the place where such an object would appear. Therefore, in WSML, variables occur in place of *concepts*, *attributes*, *instances*, *relation arguments* or *attribute values*.

As we can see there is a very strong connection between identifiers and variables. It seems like almost everywhere an identifier can be used, also a variable may be adequate. We also see, that variables implicitly carry a certain, of the

above mentioned, types. So we would have to think about *typing* our placeholders.

The only thing left are strings and numerical values. Since these build-in simple types are used to build more complex types, it's quite obvious that they are used as attribute-types and attribute-values respectively.

We get our final hint for the specification of placeholders by the defined semantics of the operators. We know everything a term can be, and we know that a term cannot be another logical expression. Hence, the final step, that will lead us to a correct usage of different placeholders, is to figure out the semantics of our operators. As we will see, different operators will go with different types of ontological entities.

How they go together is written in the WSML Language Reference [6, Sec. 2.8] and reproduced (with minor modifications) in the following list for our convenience:

- α subConceptOf γ is an atomic formula, where α and γ both identify concepts.
- α memberOf γ is an atomic formula, where α identifies an instance and γ identifies (a) concept(s).
- $\alpha[\beta$ ofType $\gamma]$ is an atomic formula, where, α identifies a concept, β identifies an attribute and γ identifies (a) concept(s).
- $\alpha[\beta$ impliesType $\gamma]$ is an atomic formula, where α identifies a concept, β identifies an attribute and γ identifies (a) concept(s).
- $\alpha[\beta$ hasValue $\gamma]$ is an atomic formula in where α identifies an instance, β identifies an attribute and γ identifies (an) instance(s).

This finally leads us to the specification of placeholders that will be called *template-variables*.

3.3 WSLET Specification

The basic idea behind WSLET-based query templates is to not only store WSML Logical Expressions, but to provide a mechanism, that allows for a more flexible reuse of predefined queries. This is achieved by inserting typed placeholders, called template-variables, into WSML Logical Expressions. The placeholders stand for values that should be provided by an end-user in order to obtain some pragmatical information.

To ensure the meaningful use of these placeholders, there necessarily have to be some syntactical and semantical requirements for WSLET that will be discussed in the following.

3.3.1 Syntax

WSLET is based on the *WSML Logical Expression* (WSML LE) subset of the WSML grammar specification. The WSLET grammar (see Appendix A) is actually an augmentation of this grammar-subset, in the way that every valid WSML LE is also a valid WSLET. At the heart of WSLET lies the syntactical construct that makes up a template-variable. As already mentioned template-variables serve as placeholders for values or variables, which are supposed to be specified by a user. We also know that values and variables are abstracted by a syntactical element called term. Therefore, the appropriate place of a template-variable must be in the production rule of a term. Table 3.6 shows you the WSLET production rules of a term.

Table 3.6 Place of the template-variables (tvar) in the WSLET Grammar

Nonterminal		***Rule(s)-Components***
term	=	*value* | **variable** | **nb_anonymous** | *tvar*

As you can see a template-variable (*tvar*) is a possible alternative when deriving a term. This means whenever a term is used in the WSLET grammar, it could stand for the token sequence that makes up a template-variable. But how does this token sequence look like? The answer to this question can be found in table 3.7.

Table 3.7 Syntactical definition of a template-variable

Nonterminal		***Rule(s)-Components***
tvar	=	**dollar lbrace** *tvar_type tvar_name_decl*? **rbrace**
tvar_name_decl	=	**lbracket name rbracket**
tvar_type	=	**t_concept** | **t_conceptlist** | **t_attribute** | **t_instance** | **t_instancelist** | **t_attributevalue** | **t_attributevaluelist**

You obtain the template-variable-type keywords (*tvar_type*), by omitting the prefix 't_' of the tokennames (e.g. *concept*, *instance*, *attribute*...). The **name**-token stands, in coarse, for a alpha-numerical sequence that starts with a letter or an underscore. As we see, a template-variable has a type and an optional name. The meaning of these properties and how they are intended to be interpreted by an application is discussed in section 3.3.3.

So, syntactically, a template-variable can be used everywhere a term is used. However an arbitrary use, may result in a meaningless expression. To make the creation of query templates less error prone, some semantical requirements are suggested, that a WSLET should fulfill.

3.3.2 Semantic requirements

Since terms occur in many places of the WSLET grammar, the definition of extensive (or even complete) semantic requirements would go beyond the scope of this thesis. It may also not be neccessary to discuss every aspect of semantic analysis, because many might be similiar to the ones we will elaborate. Further, the current version of the grammar may not be final, so it might be too early to define language semantics. Therefore the requirements we discuss here, only comprise these situations where a template-variable is used in combination with the following operators: *memberOf*, *subConceptOf*, *hasValue*, *ofType*, *impliesType* (see Section 3.1.3 for a discussion of the operators). Most of the queries will fall back to one of these operators, and that's why it's of great value to discuss their sound use with template-variables.

As we saw in section 3.1.2 these operators occur in syntactical constructs called *molecule*(s). Molecules can have three different forms, that are shown in the corresponding production rule (see table 3.8). The semantical check we intend requires an interpretation of each production rule, to figure out all sound combinations of operators and operands.

Table 3.8 Production rules of a molecule

Nonterminal		***Rule(s)-Components***
molecule	=	*term attr_specification? cpt_op termlist* | *term cpt_op termlist attr_specification* | *term attr_specification*

With minor modifications taken from: Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/#grammar:molecule>.

In which manner this can be achieved is open to the implementing parser/semantical analyzer. We renounce here to do a comprehensive case analysis, that would span a couple of pages. However, it suggests itself to check the semantical correctness of such an expression in a hierarchical fashion starting with the conceptual operator(cpt_op) if existing. Depending on its type there are requirements, given in the WSML Specification, of what can stand on its right and its left side (see Section 3.2.2). Note, that these operands (that are called *term* and *termlist* in the production rules) can also be variables. What's left is the validation of the attribute specification, if existing. This specification is strongly connected with the type of term it refers to (standing to its left). Again, the reader is refered to the WSML Language Reference [6] for further details.

3.3.3 How to interpret Template Variables

Calling this section "WSLET Excution Semantics" would be a bit exaggerating, since the treatment, of what to do with WSLET, that is given here, is just not formal enough. In section 3.3.1 we saw that a template-variable has some properties – a *type* and an *optional name*. We will briefly explain here what they're good for.

The type of a template-variable tells us what kinds of values are valid for this placeholder and has to be respected by an application that interprets a WSLET. Most of the types are self-explaining. Table 3.9 summarizes the variables types and the allowed values they may take.

Table 3.9 Template-variable-types and their allowed values

Template-variable-type	***Allowed Values***
concept	Identifier of a concept Identifier of a build-in type Variable
conceptlist	Identifier(s) of (a) concept(s) Identifier(s) of build-in type(s) Variable(s)
attribute	Identifier of an attribute Variable
instance	Identifier of an instance Variable
instancelist	Identifier(s) of (an) instance(s) Variable(s)
attributevalue	Identifier of an instance data value (e.g. integer, string) Variable
attributevaluelist	Identifier(s) of (an) instance(s) data value(s) Variable(s)

Only attributevalue(list) abstracts two kinds of values at once, namely instance(s) and data value(s) (build-in types like integer or string). Note, that a list is a syntactical construct, that is able to contain multiple comma separated values, e.g. { identifier1, “Hallo”, id2 }. Further, every template-variable can also be replaced by a WSML variable. In case a user does not specify a value, a variable, with a name not shared with any of the other variables in the WSLET, must by default take the place of the template-variable.

The meaning of the optional name of template-variable is quickly explained. If we want two or more template-variables of the same type to have the same value, we got to give them the same name. This requires the interpreter of the template to synchronize the values, in case one of them, associated with a template-variable, changes.

Chapter 4

Implementation and tool support

Query templates are designed to support the end-user in creating meaningful WSML queries. Therefore it suggests itself to provide some user-interface that interprets a query template and shows the user an intuitive representation. The interface should also assist him to fill in the right values, so he can finally execute a pragmatic query.

Not only using query templates should be supported by a graphical user interface, but also their creation. Creating templates involves a few steps for which a developer can definitely need support. After all, tools are there to save a lot of time, prevent a lot of errors, and save a lot of nerves.

This section will present you a prototypical implementation of tools that help their users with the management, creation and execution of WSLET-based query templates. The discussion will not get into technical detail, but focus on provided functionalities, associated mechanisms and chosen ways to solve problems.

4.1 WSLET core functionalities

The major sense of graphical user interfaces is to make application functionalities accessible in a convenient and intuitive manner. This also means they are more or less useless for an end-user as long as the underlying functionalities are not available.

So before we can talk about the query tools, we have to talk about the underlying core functionalities. Since at the heart of a query template lies WSLET, the core functionalities are all connected to this template language.

4.1.1 Syntactical checking

WSLET is a formal language and is specified in a context free grammar. Therefore a sentence in this language can be easily checked for its syntactical correctness by the help of a parser. Checking if a sequence of symbols is a valid WSLET is an important functionality, that is needed at many places. Beside from algorithms that have to process a WSLET, there are also persons that care about the correctness of a WSLET – most of all it's creator. After all he wants his template to be something beneficial – providing the user an erroneous query template can not be in his interest.

A syntactical invalid WSLET can lead to two kinds of problems. The first one is that an error may be propagated to the generated WSML LE, which can eventually cause a reasoner to not process the query. The other problem occurs when a template-variable within a WSLET is erroneous – this will lead to a failure of the GUI that has to interpret the query template. These two problems are reason enough to have a WSLET parser. Fortunately, creating one comes very cheap and fast thanks to the existence of so called parser generators. These handy tools take a grammar specification, that is usually written in a BNF-like syntax to create at least a language recognizer out of it. 'At least' already indicates that they differ in the functionalities they provide, and depending on what one needs there may be many or few reasons to choose one tool over the other.

In general such a tool creates the source code of a parser for a specific, or multiple, output-language(s). It may also create the lexer, which is a program or routine that performs the lexical analysis of a character sequence, and that is used by the parser. Apart from this, some parser generators may also allow to execute actions when they encounter a specific syntactical construct. Others create parsers that do error recovery, which means they continue parsing when encountering an invalid token, and possibly even collect all errors on their way. Others again generate complete parse trees that can later be used to do e.g. some semantical analysis, or to translate the input text into some other formal language (as done by compilers). From this short and incomplete enumeration of different features, one can see that it may not be easy to choose an appropriate parser generator for one's specific needs.

However, choosing one for WSLET was not that hard. Since one of the main targets was to fully support the Logical Expression subset of WSML, it suggested itself to use the same tool that was used to create the WSML parser. The grammar

specification in the correct form already existed for this parser generator, and the only thing that needed to be done was to augment the grammar.

The used parser generator is called SableCC [1]. Since the decision to use this tool was actually not based on provided functionalities, we can save some paper and skip a long discussion of pros and cons. However, being a helpful tool, SableCC has some nice features that deserve to be mentioned here. SableCC generates an LALR(1) based parser. The grammar for the parser can be specified using an EBNF-like syntax supporting the *, + and ? operators (like used in regular expressions). Further SableCC automatically generates strictly-typed abstract syntax trees (AST) and tree walkers.

Especially, this last feature turned out to be particular useful when implementing the semantical analysis of WSLET.

4.1.2 Semantical analysis

After a WSLET turns out be syntactically correct the semantical analyzer takes over to determine how meaningful a template is. This functionality is achieved by walking the abstract syntax tree, generated by the parser, in a depth-first traversal. The semantical analysis is needed, because the WSLET grammer allows constructs that may not result in a meaningful query. This gets more clear when taking a look at an easy example.

Let us consider a query template that allows us to search for concepts that are a member of a given instance (?x **memberOf $**{*instance*}). If you frowned while reading this sentence you already have an idea why semantical analysis can be very useful – it's just not possible to search for a concept that is a member of a particular instance. The hierarchy is defined just the other way round: an instance belongs to some concept(s) (so the correct version of the WSLET would be ?x **memberOf $**{*concept*}). Allowing the wrong case to be used in a template would just annoy the user, because he would be confronted with useless result obtained from the reasoner. The implemented semantical analyzer recognizes such an error, and further generates a meaningful message that proposes some valid alternatives to be used instead of the erroneous token.

As already mentioned in the proposal for semantic requirements in section 3.3.2, it would be a lot of work to check for all cases that may end in a meaningless

[1] SableCC. Accessed on 11 April 2008 <http://sablecc.org/>.

template. Therefore, the implemented analyzer only focuses on the cases where a template-variable is used in combination with the following operators: *memberOf*, *subConceptOf*, *hasValue*, *ofType*, *impliesType* (see Section 3.1.3 for a discussion of the operators). This should be enough to support a user in most of the cases when creating a query template. It should be also sufficient because query templates are assumed to be created by IT-experts, who have an understanding of WSML. If these experts further test their query templates on appropriate ontologies, they should be able to find the remaining meaningless expressions in the WSLET.

Now, we already saw that the semantical analysis follows the syntactical checking of a WSLET. The combination of the two activities can be summarized in the term *validation*. A valid WSLET is an expected condition for various user-interface supported functionalities. E.g. when creating or editing a query-template it's impossible to save it, as long as the associated WSLET is not valid. Also, the GUI that interprets a WSLET requires it to be valid. Only after this is guaranteed, the user will see a nice graphical representation of the query-template. This later scenario is depicted in figure 4.1. One can also see from it how a valid WSLET serves as a starting point for the creation of the data-model, that will be one part of the input for the template processor.

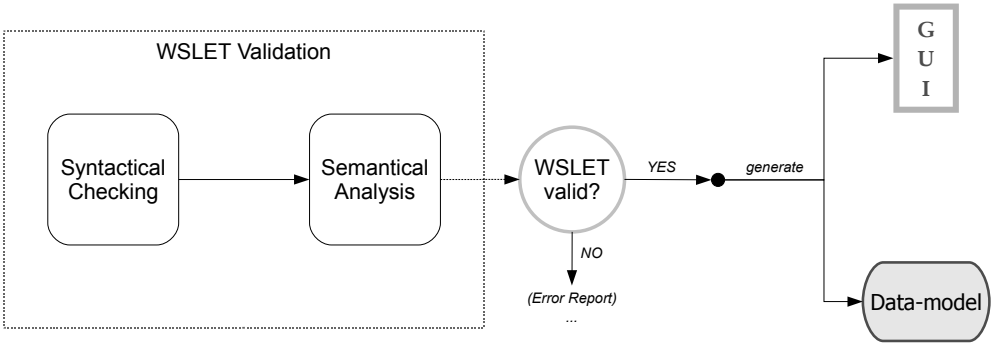

Figure 4.1 WSLET validation and it's role as pre-condition for other operations

At this point it may be appropriate to loose one short word about the used data-model and about the used data-structure to store it. From what we know of WSLET, it's quite obvious that the input data for the template processor needs to be a collection of template-variables along with their user-assigned values. However, one also has to take care of WSML variables in case a template-variable gets assigned "any value(s)" - which will cause it to become a real WSML variable

in the template processor. The template processor needs to know which variable names are still free, in order to avoid equal names, that may result in an unwanted query. It therefore suggest itself to create a list of WSML variables during parsing. In the provided implementation this list is saved along with the template-variables in the same data-structure.

4.1.3 Template processing

The template processor may seem to be the core functionality of the system, though it's implementation is one of the easiest parts. Under the assumption that the WSLET along with the input data is valid, everything comes down to a simple substitution mechanism. Actually it's a bit more difficult, because the template-variables that were assigned "any value(s)", become real WSML variables. The template processor has to take care that the generated variable names are different or the same, depending on whether there are multiple occurrences of a named template-variable. Everything else is straightforward, regarding the current state of the WSLET language extent, that is at the moment limited to typed and named template-variables.

4.2 WSLET Tool Support

During the past millenniums of cultural evolution mankind proved to be not only a tool making, but to be a tool obsessed species. The obvious reason for this obsession, and for the invention of tools in general, is that they make our lives easier. Actually, tools are so important to mankind that their creation and evolution influences our evolution in almost every aspect of our being. They are our self made aid – and some of these useful objects are even referred to as art (apart from the fact that almost all of them are artificial).

The term 'tool' is so widely used in our modern society that it's easy to think about different categories of tools. Here, we want to distinguish them by a particular aspect. This aspect has nothing to do with whether the tools are real or virtual, but with who uses a tool and what is achieved by its use.

We already mentioned that things, which are referred to by the term 'tools', can bring an immediate benefit to individuals and make their lives easier and more convenient. However, tools, in a more traditional sense, can also be used to

make and build other things, which can again be tools. In computer science such tools are appropriately called *developer tools*.

The intention of developer tools is to help programmers and other specialist, that use the computer as a work aid, to produce computational artifacts (e.g. programs, configuration files, ...) and to support them with associated tasks (e.g. refactoring of code, deployment of components, ...). On the other hand, the so called 'end-user tools' should help normal users with the 'make-life-easier-and-more-convenient' part.

The tool support for WSLET, presented in this section, tries to serve both sides. It helps the normal user to create a WSML query, and the expert to create a query template. The context for the use of these tools are WSMO ontologies. WSMO ontologies in turn have a very close relation to semantical annotated web services, called *Semantic Web Services* (SWS). To support experts in various aspects of SWS, including the creation and querying of WSMO ontologies, integrated development environments (IDE) have been developed. IDEs can basically be seen as a collection of developer tools.

The Web Service Modeling Toolkit [2](WSMT) and WSMO Studio are such IDEs for WSMO-based SWSs. Both of the tools are based on the Eclipse framework, which allows to create applications that are modular and can be easily extended. The additional functionalities are provided as a component that can be almost literally plugged-in to such an application. That's why such components are called *plug-ins*. It therefore suggest itself to provide the WSLET tools as such a plugin that can be integrated into the mentioned SWS-IDEs.

To make such an integration easy the dependencies of the plugin have to be kept to a minimum. In fact, the WSLET plugin only needs an appropriate version of wsmo4j (a Java object-model of WSMO) to create a query over an ontology. Of course a pragmatic use of the plugin can only be achieved when it's used together with other components, the most important of which may be a reasoner, to which the generated query can be sent.

In the following we will see how to use the tools to create and execute a query template. The basic functionalities will be shown on the basis of easy, tutorial-style examples and by the use of WSMO Studio.

[2]SourceForge.net: The Web Service Modeling Toolkit (WSMT). Accessed on 11 April 2008 <http://sourceforge.net/projects/wsmt>.

4.2.1 Executing query templates

To execute a query template in WSMO Studio a WSML-file containing an ontology has to be opened. After the file opens the ontology should be displayed in the WSMO Navigator. A right click on the ontology element brings up a context-menu from which we can choose ***Execute Query Template***. This will open a dialog, resembling the one depicted in Figure 4.2.

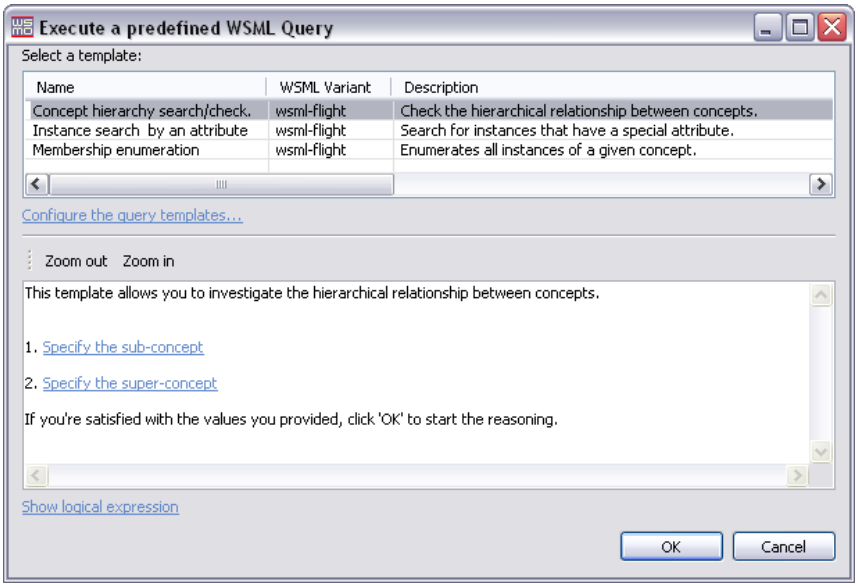

Figure 4.2 Executing a query template

The upper half of the dialog contains a table with all active query templates. It shows us the name of a template, as well as a short description, that tells what it does. The table can be sorted alphabetically and column-wise by clicking on the appropriate column-header. The link ***Configure the query templates...*** that is placed below the table, brings up the preference page that allows us to manage our query templates and to create new ones.

In the lower part of the dialog we see a textbox containing the textual description and/or instruction of the template selected in the table. We can change the size of the text by using the ***Zoom in*** and ***Zoom out*** buttons in the toolbar. Some of the content may be represented as links. If you think of the template as a form, the links are the fields that can be filled in.

Now, selecting different templates from the table will change the content of the lower textbox. Figure 4.3 shows us an example where the textbox has the following content:

Searches all instances where this attribute has some assigned value.

This template does what it says – we only have to tell it what attribute we're interested in. To do so, we have to click on the link. This will bring up a little shell just underneath the link, that allows us to select a value from a list. As soon as this little shell becomes inactive (e.g. by clicking somewhere in the textbox), the new value is assigned to this placeholder and the text of the link changes. If we wish to see the original text again, we just have to place our mouse-cursor over the link and wait till the tooltip-text is shown (see Figure 4.3).

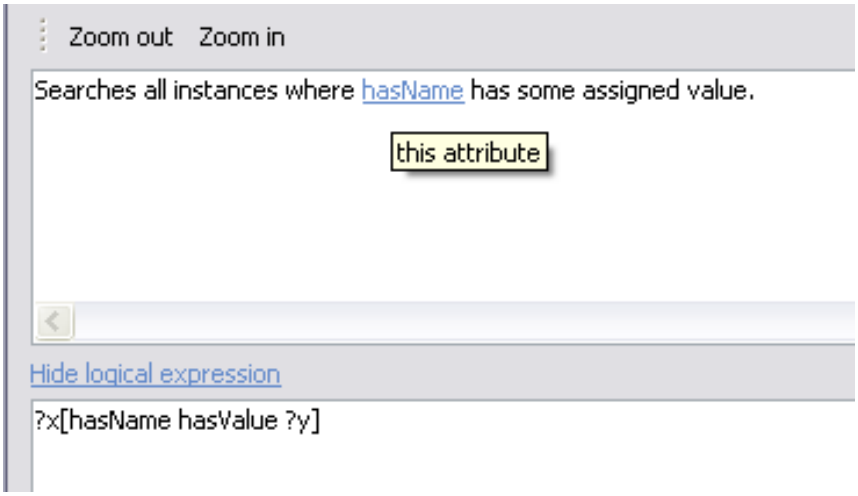

Figure 4.3 Showing a generated WSML Logical Expression

To get a better imagination of what is happening behind the scenes, we can click on the link ***Show logical expression***, that can be found on the bottom of the dialog. This will bring up an editor, that shows us the actual WSML Logical Expression that will be executed when we click ***OK***(see Figure 4.3). If we would click on the link for the attribute in the upper textbox, and assign a different value, we could see how the logcial expression changes. Everytime a value is changed, the query template and the set of user-specified values (input data), will be processed into a WSML logical expression.

In case one is familiar with logical expressions, and think that the generated query needs a little treatment from his side, he can also use the editor to make any changes of the query he desires. As long as the editor is visible, its content is the query that will be send to the reasoner, when we click ***OK***.

4.2.2 Managing query templates

The *Query Templates Preference Page* is the entry point for managing templates and for creating new ones. There are two ways to reach this page. The first one is via the main menu. A click on ***Window ➲ Preferences...*** brings up the *Preferences Dialog*. In the tree shown on the left side we have to choose ***Query Templates*** to show the appropriate preference page (see Figure 4.4). The other method is to use the ***Configure the query templates...*** link that we can find in the query template execution dialog (see Section 4.2.1). This will bring us to a preferences dialog containing only the *Query Templates Preference Page*.

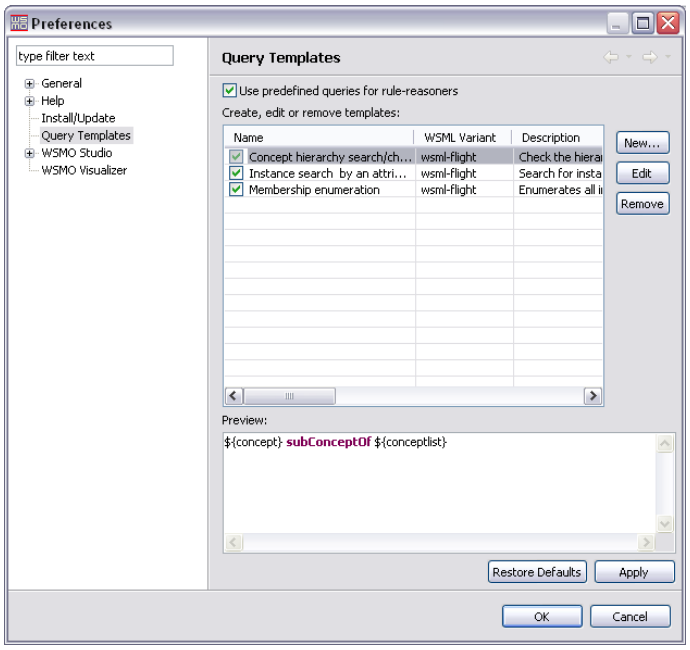

Figure 4.4 The Query Templates Preference Page

Now, managing query templates is very straightforward. All of our query templates are displayed in a table. Selecting a template will show a preview of the WSLET that makes up the template. A selected template can also be *removed* or *edited* by clicking the appropriate buttons. The checkbox on the left of the template name, allows us to *activate* or *deactivate* a template. Only activated query templates will be shown to the user when he opens the dialog to execute a query template.

Clicking ***Apply*** or ***OK*** will save any changes made to the templates. If we want to dismiss the changes we made, we can do so by clicking ***Cancel***. In case we want to restore the build-in templates we can click on ***Restore Defaults***. Cau-

tion, restoring these templates actually removes the current set of templates as soon as it's confirmed with ***OK*** or ***Apply***.

If we click on one of the ***New...*** or ***Edit*** buttons, or double click on a template in the table, the dialog for editing the properties of a query template will be opened. We will discuss the functionalities of this dialog in the following.

4.2.3 Creating query templates

In this section we will discuss the features of the dialog, that allows the editing or creating of a template, by following a simple example. Let's bring up the *Query Templates Preference Page* by following the steps explained in section 4.2.2.

Next, we click on ***New...*** to create a new query template. The dialog we see now resembles the one in figure 4.5. The first thing to notice, is that the ***OK*** button of the query template is disabled. That's because every query template requires to have at least a WSML-Variant assigned, and more importantly, a WSLET (referred in the dialog by *Logical Expression Template*).

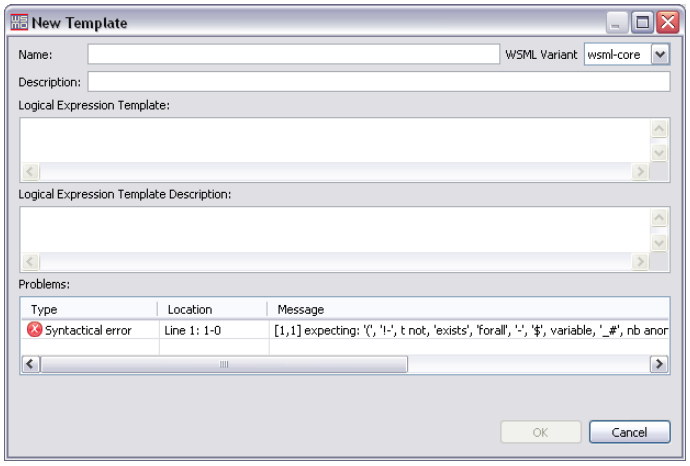

Figure 4.5 Dialog for creating a new query template

But let's start with giving the template a *name* and a *description*. We want to create a template that gives us all members of a concept, that has some special super-concept. Let's call this template "*Members of sub-concept search*". There are better names, for sure, that may be even more intuitive for an end-user. As a description we provide some meaningful short sentence of what the template does like "*Searches all members of a concept that has some special super-concept(s)*".

The next thing, we want to do, is to create the actual WSLET. For this we use the upper editor. The lower editor will be used to tell the user what the template does, but let's create the WSLET first. To do so we type in the following text:

> ?x **memberOf** **$**{*concept*[alpha]} and **$**{*concept*[alpha]} **subConceptOf** **$**{*conceptlist*}

While typing, one may see error indicators at the bottom of the dialog and highlighted error ranges in the editor – that's because the WSLET is validated whenever it changes. If we typed in the string correctly, the dialog will inform us that the query template is valid. We also notice, that the content of the second editor changed. This editor is used to provide the user with a meaningful description of what the template does, and possibly instructions on how to use it. Note, that the number of generated fields (delimited by '{{' and '}}' respectively) corresponds with the number of template-variables we are using in the WSLET. These are actually the fields, that will become links in the query template execution dialog, and the text between the double quotes, will be the initial text of such a link. At this place a picture may say more than thousand words, so take a look at figure 4.6.

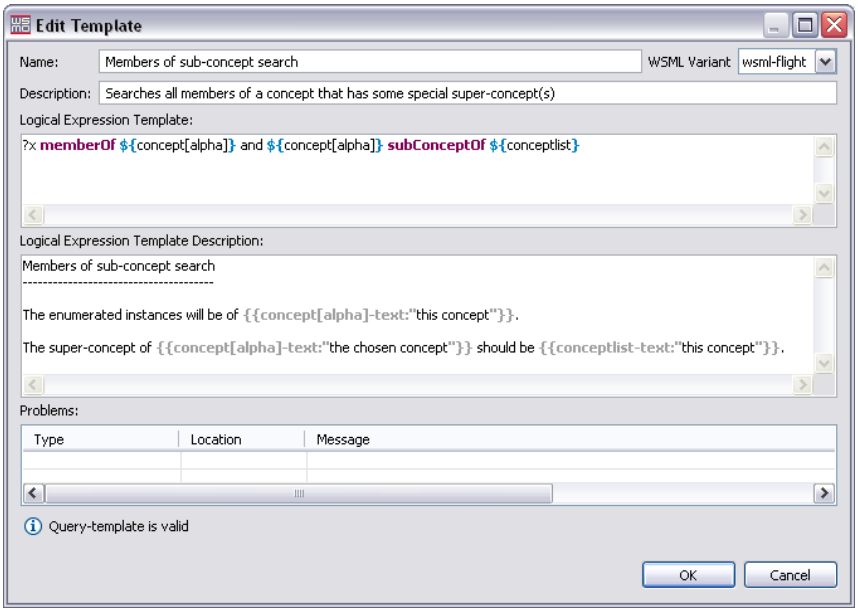

Figure 4.6 A query template example

What we did not yet discuss is the error reporting mechanism of the dialog. We already mentioned that each time we change a WSLET, it gets validated. The

validation process is done in two steps. First syntactical, then semantical. In case there is an error a new entry will be shown in the problems table. The entry will tell us what *type* of error occured (syntactical or semantical), the *location* of the erroneous character sequence and a *message* that gives us additional information. Clicking on an entry in the table will select the erroneous range in the editor, which can be convenient in case you want to replace it.

Let's make some errors to see how this works. We replace the original WSLET, by insert the following invalid version:

> ?x **memberOf** **$**{*instance*[alpha]} and **$**{*concept*[alpha]} **subConceptof** **$**{*instancelist*}

As you may have already noticed, there are a couple of errors in this query. First of all there is a typo – the user entered '*subConceptof*' instead of '*subConceptOf*'. The parser found this error, and stopped the further validation of the expression. To resolve it, we select the error entry in the table. This will in turn select the error in the editor and we can start to type. Let's type in a small '*s*' and hit *Ctrl+Space*. This will bring up the content assistent and in our case, we just have to hit *Enter* to insert the correctly spelled keyword.

Since our expression is now syntactically correct, the validator starts with the semantical analysis. As we can see there are two errors (see Figure 4.7). It may be convenient to maximize the dialog, so you can better see the error message. Semantical error messages are interesting, because they give us some suggestions, on how to resolve the error. In our example wrong types are used for the template-variables. We click on the first error entry, and replace the editor-selection with '*concept*'. With only one error left, let's click on it and replace the editor-selection with '*conceptlist*'. We now have a perfectly valid query template.

It's advisable to double check, if the *Logical Expression Template Description* is still what one wanted it to be, because it may be, that the texts are no longer at the right position, in case template-variables are added to or deleted from the WSLET. If everything meets our requirements, let's click ***OK***. The dialog will close and the new template will be displayed in the list. Important – don't forget to click ***Apply*** or ***OK*** in case you are sure you want to safe the templates. If you're not satisfied with the changes you made, click ***Cancel*** to dismiss them.

As we just saw, the dialog for creating and editing templates provides some handy functionalities that mainly aim at creating a valid query template. How-

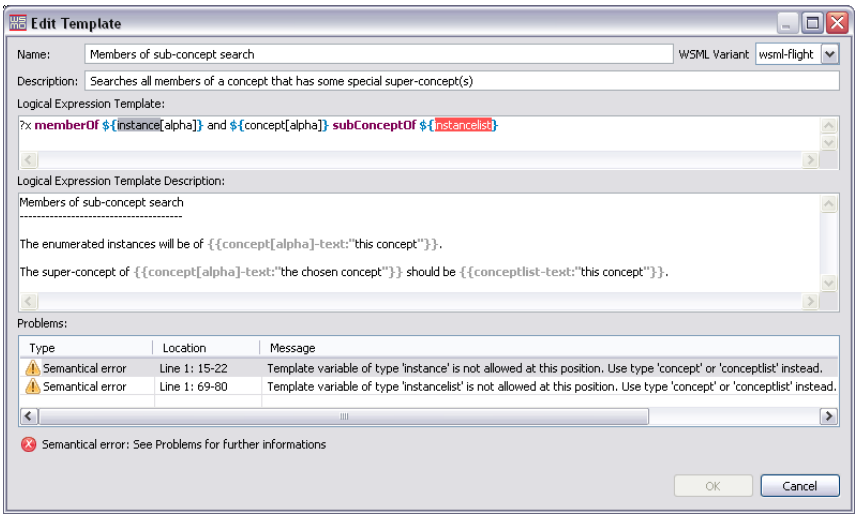

Figure 4.7 Reporting of semantical errors

ever, the possibility to provide a meaningful natural language description may be the biggest benefit for the users of WSML query templates.

Chapter 5

Conclusions

The goal of this thesis was to provide a mechanism and tools, to make the querying of WSMO ontologies, specified in WSML, easier for non-IT-experts. A template-based approach has been taken in order to achieve this aim. The approach has been motivated and adapted to the concrete situation. In order to meet all identified requirements a template language – WSLET – has been specified. WSLET in turn serves as the basis of a query template system that is intended to be used with graphical user interfaces – the tool support for WSLET. The following sections will enumerate the contributions that have been made and link the current achievements with possible future extensions.

5.1 Summary of Contributions

In the course of this thesis the following contributions have been made:

1. A template-based approach, to make the querying of WSMO-ontologies easier, has been provided. This appraoch has been depicted in an abstract view that shows the main parts of the template-system and how users are intended to interact with it.

2. A template language called WSML Logical Expression Templates (WSLET) has been specified. Further a parser, based on the grammar specification of WSLET, was created by using the SableCC parser generator. Also, relevant semantical aspects of WSLET were reflected and a minimal suggestion for a semantic analyzer was given. This suggestion was followed to implement a

semantical analyzer that has been realized by doing a depth-first traversal on the abstract syntax tree (AST) of a WSLET.

3. Tools, that support the management, creation and usage of query templates have been implemented as an Eclipse-plugin. This plugin can be integrated into the Web Service Modeling Toolkit (WSMT) and WSMO Studio. The management of query templates can be done via a preference page, that allows to add, remove and edit query templates. The main features of the interface, that allows to create a query template, comprise:

   - an editor for WSLET that provides syntax- and error-highlighting along with a simple content assistent
   - an editor to provide a (natural language) description of a WSLET
   - an error reporting mechanism that provides suggestions on how to solve semantical errors

   The interface that helps using/executing a query template provides the following features:

   - an intuitive representation of query templates that is shown to the user
   - an easy mechanism that allows users to fill in values for the template-variables

5.2 Future work

Even though the overall goal of making the query task easier for non-IT-experts has been achieved, many things of the presented solution can be improved and extended. The following list includes some ideas on how to make WSLET and its tools better.

1. *Possible extensions of WSLET*

   (a) One fundamental idea of WSLET is that the interface, that interprets it, provides the user an intuitive representation of what the query tempalte is good for, and to help him filling in the right values. For the later task, the interface provides all elements of a particular type in

the queried ontology. For example if a query template has a template-variable of type attribute, all attributes, defined in every concept of the ontology (and every imported ontology) are represented as a possible value to the user. This current mechanism can be seen as the least restrictive way to provide values. However, there may be situations, when the user has the desire to be more concrete about the values, he wants to use. He may be for example only interested in the attributes of a specific concept, or even only in the attributes of a specific instance.

Therefore, a useful extension would be to define an optional data-provider part for WSLET template-variables. The appropriate place for this optional part might be after the type definition. To give an easy example of how this could look like consider the following expression:

?x[**$**{*attribute* **valuesFrom** *concept*[c]} **hasValue** ?x]
memberOf ${*concept*[c]}

This WSLET would allow an user to only choose attributes that are really defined in the chosen concept. He therefore needs to know less about the ontology and would still be able to choose reasonable values.

(b) Of course changes and/or extensions to the WSML specification itself would have to be reflected in the WSLET specification to stay consistent with WSML.

2. *An XML Schema for query templates*

Since, at the current version of the tools, query templates are stored in a preference file within the application, the query templates a user posses are the ones that are hard-coded in the plugin as defaults and the ones that have been explicitly typed into his application. This means, that at the current version, there is no way a user can open a file with query templates and add them to his collection fo templates. Since this lack in functionality is not very user-friendly, it would be good to create an XML Schema for storing query templates and to extend the tools with functionalities that allow to open and to save such schema-based XML-files. It would allow the user to easily add new query templates, an expert wrote, to his current collection of templates.

5.3 Final words

Bringing the original problem back to the language level was the basic idea that led to the presented solution. Having a template language for WSML Logical Expressions should make the querying task easier. However, an evaluation of the implemented tools has been omitted, since their development is work in progress. Especially the lack of the just mentioned functionalities prevents the tools of being as user-friendly as intended. It may therefore be subject of future investigations to find out in how far this solution can bring real benefit to the problem of creating semantic queries.

Finally, I want to thank Michal Zaremba for guiding me through the process of writing this thesis and for always giving me quick and valuable feedback, as well as Mick Kerrigan for his inspiring ideas.

Appendix A

WSLET Grammar Specification

This appendix contains the grammar specification of WSML Logical Expression Templates (WSLET). It's intended for the reader that posseses a basic understanding of formal languages and who's interested in WSLET for technical reasons. The grammar is represented by the content of the SableCC specification file that was used to create a first parser. It's a simple text file containing the lexical definitions along with the grammar of the language. The specification file is divided into different sections:

Helpers Contains regular expressions that are used to define tokens.

Tokens Specifies the lexical definitions (tokens), also using regular expressions.

Ignored Tokens Lists all the tokens that should be ingnored by the parser.

Productions Specifies the production rules of the grammar, written in Backus-Naur Form(BNF).

A.1 Helpers

```
Helpers
all   =    [ 0x0 .. 0xffff  ];
escape_char   =    '\';
basechar   =    [ 0x0041 .. 0x005A  ] | [ 0x0061 .. 0x007A  ];
ideographic  =  [ 0x4E00 .. 0x9FA5 ] | 0x3007 | [ 0x3021 .. 0x3029 ];
letter   =    basechar | ideographic;
digit   =    [ 0x0030 .. 0x0039  ];
combiningchar   =    [ 0x0300 .. 0x0345  ] | [ 0x0360 .. 0x0361  ]
                     | [ 0x0483 .. 0x0486  ];
extender   =    0x00B7  | 0x02D0  | 0x02D1  | 0x0387  | 0x0640  | 0x0E46  | 0x0EC6
                 | 0x3005  | [ 0x3031 .. 0x3035  ] | [ 0x309D .. 0x309E  ]
                 | [ 0x30FC .. 0x30FE  ];
alphanum   =    digit | letter;
hexdigit   =    [ '0' .. '9'  ] | [ 'A' .. 'F'  ];
```

```
not_escaped_namechar   =    letter | digit | '_'  | combiningchar | extender;
escaped_namechar   =    '.'  | '-'  | not_escaped_namechar;
namechar   =    ( escape_char escaped_namechar ) | not_escaped_namechar;
mark   =    '-'  | '_'  | '.'  | '!'  | '~'  | '*'  | '''  | '('  | ')';
escaped   =    '%'  hexdigit hexdigit;
unreserved   =    letter | digit | mark;
scheme   =    letter ( letter | digit | '+'  | '-'  | '.'  )*;
port   =    digit*;
idomainlabel   =    alphanum ( ( alphanum | '-'  )*  alphanum )?;
dec_octet   =    digit | ( [ 0x31 .. 0x39  ] digit ) | ( '1'  digit digit )
                 | ( '2'  [ 0x30 .. 0x34  ] digit ) | ( '25'  [ 0x30 .. 0x35  ] );
ipv4address   =    dec_octet '.'  dec_octet '.'  dec_octet '.'  dec_octet;
h4   =    hexdigit hexdigit?  hexdigit?  hexdigit?;
ls32   =    ( h4 ':'  h4 ) | ipv4address;
ipv6address   =    ( ( h4 ':'  )*  h4 )?  '::'  ( h4 ':'  )*  ls32
                   | ( ( h4 ':'  )*  h4 )?  '::'  h4 | ( ( h4 ':'  )*  h4 )?  '::';
ipv6reference   =    '['  ipv6address ']';
ucschar   =    [ 0xA0 .. 0xD7FF  ] | [ 0xF900 .. 0xFDCF  ] | [ 0xFDF0 .. 0xFFEF  ];
iunreserved   =    unreserved | ucschar;
ipchar   =    iunreserved | escaped | ';'  | ':'  | '@'  | '&'  | '='  | '+'  | '$'
               | ',';
isegment   =    ipchar*;
ipath_segments   =    isegment ( '/'  isegment )*;
iuserinfo   =    ( iunreserved | escaped | ';'  | ':'  | '&'  | '='  | '+'  | '$'
                 | ','  )*;
iqualified   =    ( '.'  idomainlabel )*  '.' ?;
ihostname   =    idomainlabel iqualified;
ihost   =    ( ipv6reference | ipv4address | ihostname )?;
iauthority   =    ( iuserinfo '@'  )?  ihost ( ':'  port )?;
iabs_path   =    '/'  ipath_segments;
inet_path   =    '//'  iauthority ( iabs_path )?;
irel_path   =    ipath_segments;
ihier_part   =    inet_path | iabs_path | irel_path;
iprivate   =    [ 0xE000 .. 0xF8FF  ];
iquery   =    ( ipchar | iprivate | '/'  | '?'  )*;
ifragment   =    ( ipchar | '/'  | '?'  )*;
iri_f   =    scheme ':'  ihier_part ( '?'  iquery )?  ( '#'  ifragment )?;
tab   =    9;
cr   =    13;
lf  =  10;
eol   =    cr lf | cr | lf;
dquote   =    '"';
not_cr_lf   =    [ all - [ cr+ lf ] ];
escaped_char   =    escape_char all;
not_escape_char_not_dquote   =    [ all - [ '"' + escape_char ] ];
string_content   =    escaped_char | not_escape_char_not_dquote;
long_comment_content   =    [ all - '/'  ] | [ all - '*'  ] '/';
long_comment   =    '/*'  long_comment_content*  '*/';
begin_comment   =    '//'  | 'comment ';
short_comment   =    begin_comment not_cr_lf*  eol;
comment   =    short_comment | long_comment;
blank   =    ( ' '  | tab | eol )+;
qmark   =    '?';
luridel   =    '_"';
ruridel   =    '"';
```

A.2 Tokens

```
Tokens
t_blank   =    blank;
t_comment   =    comment;
comma   =    ',';
  endpoint   =    '.'  blank;
lpar   =    '(';
```

```
rpar  =  ')';
lbracket   =    '[';
rbracket  =  ']';
lbrace   =    '{';
rbrace  =  '}';
hash   =    '#';
t_and   =    'and';
t_or   =    'or';
t_implies   =    'implies'  | '->';
t_implied_by  =  'impliedBy' | '<-';
t_equivalent  =  'equivalent' | '<->';
  t_implied_by_lp   =    ':-';
t_constraint   =    '!-';
t_not   =    'neg'  | 'naf';
t_exists   =    'exists';
t_forall  =  'forall';
gt   =    '>';
lt  =  '<';
gte  =  '>=';
lte  =  '=<';
equal  =  '=';
strong_equal  =  ':=:';
unequal  =  '!=';
add_op   =    '+';
sub_op   =    '-';
star   =    '*';
div_op  =  '/';

// WSLET Tokens
dollar = '$';
t_concept = 'concept';
t_conceptlist = 'conceptlist';
t_attribute = 'attribute';
t_instance = 'instance';
t_instancelist = 'instancelist';
t_attributevalue = 'attributevalue';
t_attributevaluelist = 'attributevaluelist';


t_hasvalue   =    'hasValue';
t_impliestype   =    'impliesType';
t_memberof   =    'memberOf';
t_oftype   =    'ofType';
t_subconcept   =    'subConceptOf';
variable   =    qmark alphanum+;
anonymous   =    '_#';
nb_anonymous   =    '_#'  digit+;
pos_integer   =    digit+;
pos_decimal  =  digit+ '.' digit+;
full_iri   =    luridel iri_f ruridel;
string   =    dquote string_content*  dquote;
name   =    ( letter | '_'  ) namechar*;
```

A.3 Ignored Tokens

```
Ignored Tokens
t_blank,
t_comment;
```

A.4 Productions

```
Productions
// Original rule changed: 'enpoint' token was not optional;
// All alternatives changed from 'endpoint' to 'endpoint?'
log_expr =
{lp_rule} [head]: expr t_implied_by_lp [body]: expr endpoint? |
{constraint} t_constraint expr endpoint? |
{other_expression} expr endpoint?;

expr =
{implication} expr imply_op disjunction |
{disjunction} disjunction;

disjunction =
{conjunction} conjunction |
disjunction t_or conjunction;

conjunction =
{subexpr} subexpr |
conjunction t_and subexpr;

subexpr =
{negated} t_not subexpr |
{simple} simple |
{complex} lpar expr rpar |
{quantified} quantified;

quantified =
quantifier_key variablelist lpar expr rpar;

simple =
{molecule} molecule |
{comparison} comparison |
{atom} term;

molecule =
{concept_molecule_preferred} term attr_specification? cpt_op termlist |
{concept_molecule_nonpreferred} term cpt_op termlist attr_specification |
{attribute_molecule} term attr_specification;

attr_specification =
lbracket attr_rel_list rbracket;

attr_rel_list =
{attr_relation} attr_relation |
attr_rel_list comma attr_relation;

attr_relation =
{attr_def} term attr_def_op termlist |
{attr_val} term t_hasvalue termlist;


comparison =
[left]: term comp_op [right]: term;

comp_op =
{gt} gt |
{lt} lt |
{gte} gte |
{lte} lte |
{equal} equal |
{strong_equal} strong_equal |
{unequal} unequal;

cpt_op =
{memberof} t_memberof |
```

```
{subconceptof} t_subconcept;

quantifier_key =
{forall} t_forall |
{exists} t_exists;

imply_op =
{implies} t_implies |
{impliedby} t_implied_by |
{equivalent} t_equivalent;

attr_def_op =
{oftype} t_oftype |
{impliestype} t_impliestype;

termlist =
{term} term |
lbrace terms rbrace;

terms =
{term} term |
terms comma term;

term =
{data} value |
{var} variable |
{nb_anonymous} nb_anonymous |
{tvar} tvar; //added

// WSLET Production Rule
tvar =
dollar lbrace tvar_type tvar_name_decl? rbrace;

// WSLET Production Rule
tvar_name_decl =
lbracket name rbracket;

// WSLET Production Rules
tvar_type =
{concept_type} t_concept |
{conceptlist_type} t_conceptlist |
{attribute_type} t_attribute |
{instance_type} t_instance |
{instancelist_type} t_instancelist |
{attributevalue_type} t_attributevalue |
{attributevaluelist_type} t_attributevaluelist;

variablelist =
{variable} variable |
{variable_list} lbrace variables rbrace;

variables =
{variable} variable |
variables comma variable;

value =
{datatype} functionsymbol |
{term} id |
{numeric} number |
{string} string;


functionsymbol =
{parametrized} id lpar terms? rpar |
{math} lpar arith_val rpar;

arith_val =
mult_val |
{addition} arith_val arith_op mult_val |
```

```
{semisimple1_addition} term arith_op mult_val |
{semisimple2_addition} arith_val arith_op term |
{simple_addition} [a]: term arith_op [b]: term;

mult_val =
[a]: term mul_op [b]: term |
{multiplication} mult_val mul_op term;

arith_op =
{add} add_op |
{sub} sub_op;

mul_op =
{mul} star |
{div} div_op;

id =
{iri} iri |
{anonymous} anonymous;

iri =
{iri} full_iri |
{sqname} sqname;


// Original rule changed: second alternative omitted
// '{localkeyword} prefix anykeyword'
sqname =
{any} prefix? name;

prefix =
name hash;

number =
{integer} integer |
{decimal} decimal;

integer =
sub_op? pos_integer;

decimal =
sub_op? pos_decimal;
```

Bibliography

[1] IST Project Fact Sheet – Semantics Utilised for Process management within and between EnterPrises (SUPER), 14 Jan. 2008. The Office for Official Publications of the European Communities (Publications Office). 2 April 2008 <http://cordis.europa.eu/ist/projects/projects.htm>.

[2] SUPER Integrated Project – Scope and content of the objectives. SAP AG. 2 April 2008 <http://www.ip-super.org/content/view/25/42/>.

[3] SUPER Integrated Project – Integrated Project SUPER. SAP AG. 2 April 2008 <http://www.ip-super.org/content/view/27/44/>.

[4] Jos de Bruijn. D16.1v0.21 The Web Service Modeling Language WSML, 2005. 16 June 2007 <http://www.wsmo.org/TR/d16/d16.1/v0.21/>.

[5] Dumitru Roman, Holger Lausen, and Uwe Keller. D2v1.4. Web Service Modeling Ontology (WSMO), 2007. 16 June 2007 <http://www.wsmo.org/TR/d2/v1.4/>.

[6] Ioan Toma and Nathalie Steinmetz. D16.1v0.3 WSML Language Reference, 2008. 4 April 2008 <http://www.wsmo.org/TR/d16/d16.1/v0.3/>.