Towards a Visual Editing Environment For the Languages of the Semantic Web

Johan Lövdahl johlo805@student.liu.se

July 8, 2002

Abstract

The content on the Web is primarily tailored for human readers. The use of HTML gives a simple way of describing how information should be displayed, but it makes the automation of tasks very hard. The *semantic* Web aims to alleviate this problem by adding machine-processable semantics to the information available on it, thus enabling software agents to reason about the information. The central concept for achieving this semantics is the *ontology*. An ontology provides a formal specification of the concepts that exists in a domain and how they are related to each other.

This thesis is focusing on the visual representation and editing of ontologies using UML. A mapping from UML class diagrams to a ontology language (DAML+OIL) is defined. We also investigate a visual notation called *constraint diagrams* that can be used to complement the UML ontologies with more expressive logic rules. Furthermore a translator from (a subset of) constraint diagrams to *Frame-Logic* has been implemented. The generated logic rules have been used to check context-dependent aspects of e.g. Java. The possibility of translating DAML+OIL to Frame-Logic is also discussed.

```
<RDF xmlns    = "http://w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:DC ="http://purl.org/dc/elements/1.1/">

   <Description about = "#">
     <DC:Title>Towards a Visual Editing Environment
               For the Languages of the Semantic Web</DC:Title>
     <DC:Creator>Johan Lövdahl</DC:Creator>
     <DC:Type>Masters Thesis</DC:Type>
     <DC:Date>2001-12-11</DC:Date>
     <DC:Description>This work explores the possibility of
                     using UML class diagrams complemented
                     with constraint diagrams to create
                     ontologies for the semantic web.
                     The constraint diagrams can be used to
                     create binary, typed Horn clauses.
                     It also discusses how to map DAML+OIL
                     to F-Logic.
     </DC:Description>
     <DC:Language>en</DC:Language>
   </Description>
</RDF>
```

Contents

1 Introduction

The content on the Web is mainly intended to be viewed by *humans*, the use of HTML gives us a simple way of defining how the information supplied should be displayed in the browser. A problem with this approach is that it makes *automation* of tasks very difficult, it is virtually impossible for a software agent to distinguish between a page that contains information about the programming language Java, the island Java or the coffee-shop Java.

The *semantic* Web aims to alleviate this problem by adding explicit, machine-available semantics to Web resources. This will be done by creating *ontologies*, i.e. vocabularies defining domain-specific concepts and the relations between them. The semantics emerges when agents share ontologies.

Currently a language called DAML+OIL exists for writing these ontologies, it is based on description logics and the syntax is defined as an application of XML. DAML+OIL uses a subset of predicate logic for creating concepts. Editing of ontologies is done in text editors (which is cumbersome) or special ontology editors (which people have to learn to use).

A goal of this thesis was to create a visual notation for editing ontologies. We will show how UML class diagrams can be used as a visual notation for editing DAML+OIL ontologies. The UML is a very well-known notation with a large user base and several tools exists, thus making it a good candidate. Furthermore UML and DAML+OIL share the basic modelling primitives, classes (concepts) and attributes (properties), but there are also diffrences such as the DAML+OIL class constructors.

DAML+OIL is based on description logic (DL), DL systems are usually focused on classification of class definitions and instances. But an ontology is really a schema defining the allowed structure, this means that we can use it to check instance documents for *compliance* with the schema. This was the other major goal of the thesis, to see how DAML+OIL could be used to specify the *static semantics* (context-dependent syntax) of programming languages (using the UML notation). It is well known that the static semantics can be described using first-order logic.

Although DAML+OIL is expressive enough to handle a number of context-dependent constraints it is not enough to describe everything. For example derived properties (rules) can not be stated[1]. Therefore we have experimented with two visual notations for adding binary (typed) Horn clauses to the UML notation for DAML+OIL. The Horn clauses can be used to create derived properties as well as asserting invariants.

First we experimented with rules in the UML class diagrams, which gives us a uniform visual notation for both DAML+OIL and the rules extension. The UML rules would be simple to translate to Horn clauses. But the focus has been on the second notation, the *constraint diagrams*. This is a notation developed to be a visual equivalent to UML's constraint language OCL. The constraint diagrams only supports the assertion of invariants, not definition of rules. When extending the notation with rule definitions and negation we can use it to define the binary Horn clauses. The meaning of constraint diagrams is not straightforward to define, the notation is quite different from the corresponding e.g. prolog text representation. The constraint diagrams have not been extensively studied, it is was proposed quite recently and no description of the semantics of the notation has been published.

The combination of UML class diagrams for the DAML+OIL language and constraint diagrams for the logic extension is called the *Semantic Web Editing Environment* (SWEDE). Figure 1 shows the architecture.

The **Constraint Diagram Editor** is an existing tool that can export diagrams to XML format.

The architecture is independent of which UML editor that is used as long as it can export diagrams to the standard XMI format. Currently no inference

[1] A very restricted form can actuallly be stated, but it's not enough

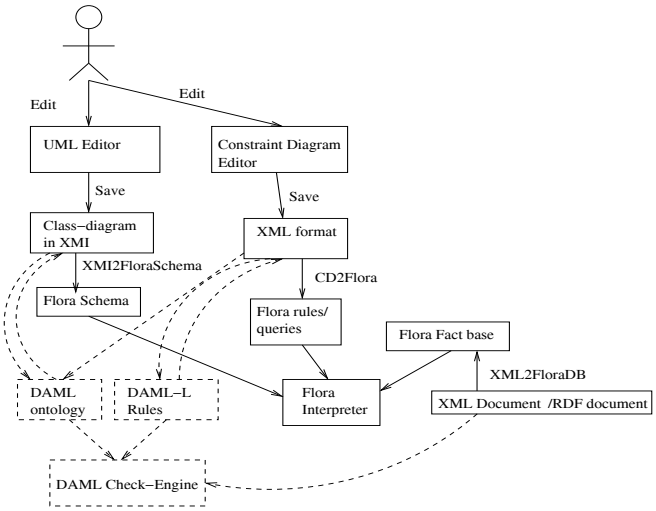

Figure 1: Architectural overview of the semantic web editing environment (SWEDE).

engine for DAML+OIL exists[2], so to make it possible to evaluate invariants a set of translation programs were implemented. The SWEDE uses *Frame-Logic* (F-Logic) system **Flora** as the inference engine for both constraint diagrams and DAML+OIL.

The **XMI2FloraSchema** translates a small subset of the DAML+OIL UML notation to F-Logic schemas. The **CD2Flora** program translates a subset of the constraint diagrams (plus the extensions) to F-Logic code. And the **XML2FloraDB** translates XML documents (i.e. instances) to F-Logic facts.

The SWEDE is demonstrated by specifying the static semantics of XHTML extended with some tags. An initial experiment with the static semantics for Java is briefly described as well.

The rest of the thesis is structured as follows:
Section 2 describes the concepts and techniques used in the rest of the sections.
Section 3 describes the architecture of the SWEDE framework.
Section 4 describes the mapping from DAML+OIL to UML class diagrams.
Section 5 examines how RDF(S) and DAML+OIL can be translated to F-Logic.
Section 6 examines what DAML+OIL can and can't do, and introduces the two different Horn rule notations.
Section 7 shows how the SWEDE environment is used to specify the static semantics.
Section 8 discusses how the Compost/Recoder program can be used in conjunction with the SWEDE to provide a static semantics description of Java.
Section 9 sums up the work.

[2] But it should be possible to map it to a DL system

2 Background

This section introduces techniques and concepts that have been the basis for this thesis.

2.1 Motivation and Scope of the thesis

In the beginning of the work the following goal was formulated:

> Create a visual environment for the specification of DAML+OIL ontologies and use it to specify the static semantics for C.

The static semantics specification should be used in a system called COMPOST-C. COMPOST-C eats c-programs and creates a abstract syntax tree (AST) on which meta-programs can perform changes (see section 2.2 for a more detailed introduction to COMPOST). The specification should be used to check if the resulting AST is syntactically correct (see section 2.3 for definition of syntax).

The specification was intended to be written in DAML+OIL, an emerging ontology language for the semantic web (see section 2.4.2). Furthermore we wanted a visual notation for doing the specification, so we decided to investigate UML.

The work was carried out in roughly three phases. In the initial phase I spent most of the time reading in on various topics. The second phase concerned design/implementation. The last phase was spent writing this document. Figure 2 gives a rough overview of how the time was spent between the phases.

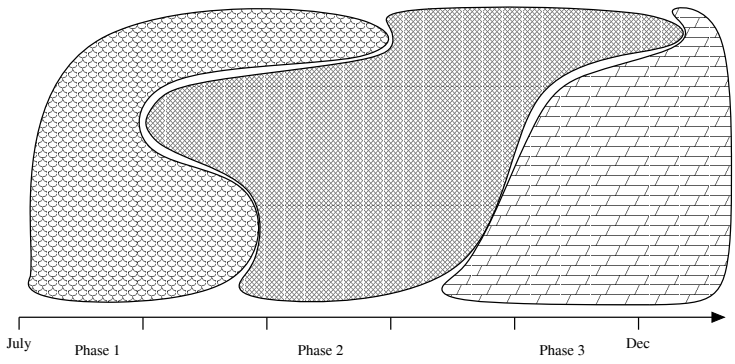

Figure 2: *The approximate workload distribution between the three phases.*

2.1.1 Initial Phase: Existing Approaches and Technologies

The first months of the thesis was largely spent reading the necessary literature. This work has been summarized in the section 2.

The possible use of UML to visualize DAML+OIL was recognized early in the work, and it has formed the backbone of this thesis. I also spent time reading parts of the ISO C specification [11] as well as studying an implementation of a C interpreter in the MAX system [22].

2.1.2 Middle Phase: Evaluation, Design and Implementation

The work done in this phase is documented in the second part of this thesis.

When I intended to start on the second phase we had to change the original goal in two aspects.

1. The intention was to use a system called C-COMPOST as the basis to read C programs and create ASTs, in the first phase I had familiarized myself with the Java-COMPOST system which does the same thing but for Java programs. But still in late September this system was not in a usable state. So we decided to work with the Java-COMPOST instead, meaning that a static semantics specification for Java should be written.

2. We couldn't use DAML+OIL as the specification language. The DAML-L language is not finished. An engine for evaluating DAML+OIL(-L) specifications was lacking, the plan was to get another masters student to write such an engine but this was not done. Instead I used a F-Logic system.

   The future COMPOST will use DAML+OIL as its specification language, but this doesn't make this thesis less relevant since rules are specified visually independent of the underlying language. Section 5 also points out the similarity and discrepancies between DAML+OIL and F-Logic.

It was during this phase I tested different tools, techniques etc. and also designed and implemented the editing environment called *SWEDE* 3.1.

2.1.3 End Phase: Write-Up

As shown in figure 2 the major part of the last two months was spent on writing this thesis. However some implementation was done, I added more rules to the static semantics specification. The figure shows the write-up phase as one solid block, but I also wrote shorter reports on my work during the two first phases. These reports have partly been reused in this final write-up.

2.2 Digging into the COMPOST: A System for Invasive Software Composition

Anyone who has written an application, in any programming language, recognizes the difficulty of getting it right from the beginning. To create systems that are easily understood, maintainable, reusable, efficient, etc. it is common to use an iterative approach to the implementation.

This means that the software system is synthesized incrementally, the first step might be to create a quick and dirty version of the intended system. Then the code is improved in an iterative manor, these improvements are called *refactorings*. A refactoring can be something as simple as replacing a variable name with another name, or it can be something more complex such as introducing a design pattern into a system.

Another important concept in software engineering is *component-based system development*. The ultimate goal of this approach is to enable programmers to create complex systems just by *composing* off-the-shelf components.

COMPOST [18] is a Java library that

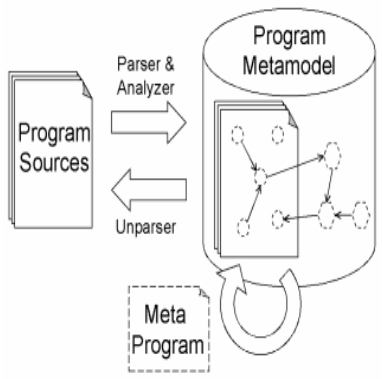

Figure 3: *The central dogma of COMPOST. The source code is parsed and the program model is created. Next the analysis phase determines the type of expressions etc. The metaprogram is then applied, and the result is printed back to source by the pretty printer. Figure taken from [18]*

2.3 Ways of Defining Syntax

In natural language only certain combinations of words are meaningful, this is also the case for programming languages. In contradiction to natural languages the allowed combinations are well defined by the *syntax* of the language. This means that the *syntax* of a language is described by a set of rules determining which strings are members of the language, these rules form the *grammar* for the defined language.

The definition of programming-language syntax is usually divided into two aspects, *context-free* and *context-dependent* syntax.

2.3.1 Defining the Context-Free Aspects

The BNF (Backus-Naur Form) notation [25] was introduced in the late 1950s and has since then been used almost universally as the specification language for context-free syntax. Usually contemporary programming language specifications use BNF, or Extended BNF (EBNF), as the metalanguage[3] to define the valid programs.

BNF and *context-free* grammars are the same thing, they both characterize the class of context-free languages. I will use the terms interchangeably in this text.

A BNF grammar consists of a set of *production rules*, each rule says that the symbol on the left-hand side of the arrow, '::=' is used to denote an arrow, can be replaced by the sequence of symbols on the right-hand side. The left-hand side can only contain one symbol.

Symbols enclosed by brackets ('<' and '>') are called *nonterminals* and can be seen as abstractions of syntatic structures (in grammar 1 the nonterminals are *<program>*, *<stmt list>*, *<var>* etc).

[3] A metalanguage is a language used to describe a language

The other symbols, *begin, end, :=* etc., are called *terminals.* A nonterminal can always be rewritten to a sequence of terminals.

A program (a sequence of strings) is syntactically correct if it can be reduced to a single, specific rule, with no input left, by repeated application of the rules.

The BNF language has a very limited vocabulary. It consists of the production arrow ('::='), the + operator indicating that the symbol should occur *atleast* once, * means the symbol should occur one or more times. Grammar 1 contains the '|' relation which indicates an alternative, but it could be rewritten as an additional production rule so it is just syntactic sugar.

*Grammar 1.*The following BNF grammar defines the context-free syntax for a simple language

<program> ::= **begin** <stmt list> **end**
<stmt list> ::= <stmt>
| <stmt> ; <stmt list>
<stmt> ::= <var> := <expression>
<var> ::= (a..z|A..Z)$^+$
<expression> ::= <var> + <var>

We can think of the BNF as consisting of three relations for describing the structure of strings(as noted in [20]):

1. The *child* relation, all the symbols on the right-hand side of a rule are children to the symbol on the left-hand side.
2. The *ordering* relation, since the right-hand side is a *sequence* there is an ordering between the symbols (this is a total order).
3. The *ISA* relation, all nonterminal symbols have a label.

We should notice that these are also the relations needed to create ordered, labelled trees. The root of the tree is corresponds to the *start symbol* of the grammar, *<program>*.

A string that conforms to the grammar restrictions can be mapped to a *derivation tree*, it is a tree were each internal node is labelled with a nonterminal symbol the grammar and the leafs with terminal symbols. The root of the tree is labelled with the start symbol.

The concept of a derivation tree is important for defining the context-dependent syntax, this is shown in the next subsection.

Below is an example of a program conforming to grammar 1, and its derivation tree.

When you have a grammar for a language it can be used to generate language recognizers, also known as *parsers.* A parser takes a string (a sequence of tokens) as input and determines if it belongs to the defined language. There are many tools available that take BNF grammars as input and generates parsers, for example Bison and JavaCC.

2.3.2 Defining the Context-Dependent Aspects

Some aspects of programming-language syntax are difficult to capture with context-free grammars, and others that are impossible. In other words, programming languages are context-sensitive languages.

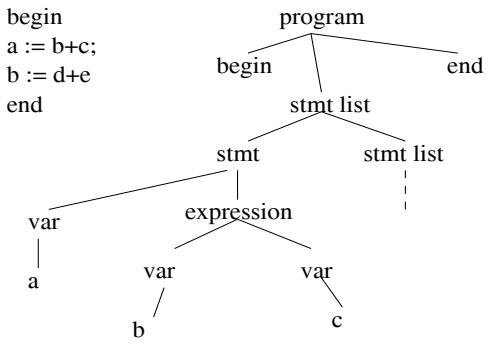

Figure 4: *Example program and its derivation tree.*

As an example of a language rule that is difficult to express with BNF, consider type compatability rules. For example, in Java a value of type 'float' cannot be assigned to a variable of type 'int' and a 'break' statement can only occur inside 'switch' or loop-statements. To express this in BNF you would have to add a lot of new production rules and nonterminals, and the result would be less readable (and of course very big).

Then there are those rules that are not possible to express with BNF at all[4], for example the rule that variables must be declared before they are referenced.

These two categories of problems makes up the part of syntax specification called *context-dependent rules*. Another name is *static semantics*, but I choose not to use it since it brings to mind *dynamic semantics* which this thesis is *not* about.

So, BNF itself is not powerful enough to characterize the intended language, $\mathcal{L}$, described by *Grammar 2*. It describes a *superset* of $\mathcal{L}$. The grammar allows you to reference a variable that has not been declared, however this was not the intention.

Grammar 2. The following BNF grammar defines a simple language with declarations and definitions of identifiers.

<program> ::= <var decl> **begin** <stmt list> **end**
<var decl> ::= **declare** <var> (, <var>)*
<stmt list> ::=<stmt>
| <stmt> ; <stmt list>
<stmt> ::= <var> := <expression>
<var> ::= (a..z|A..Z)$^+$
<expression> ::= <var> + <var>

The obvious idea would be to use *context-sensitive grammars* (CSG's) to

[4]this is a general truth for all context-free grammars

characterize the intended language, however there are two problems with this approach

- Context-sensitive grammars are difficult to understand.
- No known algorithm for generating efficient parsers from CSG's is known.

The second bullet is the most troublesome, we of course want a way of generating the parsers.

The most common approach to overcome the limitations of context-free grammars is to complement the BNF specification with rules imposing constraints on the derivation trees. Attribute grammars (AG's) and functional methods such as VDM are two example formalisms that takes this approach. The AG's seems to be the most widely used [20], they are e.g. popular in compiler generation systems (e.g. the MAX system[22]).

The basic idea behind attribute grammars is to add a few features to the BNF grammar:

- Each symbol (terminals and nonterminals) in the BNF grammar is associated with a set of attributes (i.e a set of variables).
- Each production rule in the BNF grammar is associated with a set of *semantic rules*. These rules uses the attributes of the symbols in the associated production rule, either adding a value to the attribute or checking if the attribute contains a certain value.

We can create an AG from *Grammar 2* to the problem of undeclared variables. If we add an attribute $A$to the $<$*var decl*$>$ production we can define a semantic rule that adds all declared variables to the attribute. In the $<$*stmt*$>$ production we would then check if attribute A contains the referenced variables.

The Bison/JavaCC parser-generators mentioned in the previous subsection also implements the concept of semantic rules. They could be seen as aiding the construction of AGs, they are actually more general than AGs since you can write arbitrary C (Bison) or Java (JavaCC) programs inside the semantic rules.

CADET *CAlculus on DErivation Trees*[20] is a less well-known formalism for defining the context-dependent aspects. The idea is to take the derivation tree generated by a (E)BNF grammar (such as shown in *Grammar 2*) for the language and use logic formulas (called *context-rules*) to check if the form of derivation tree is legal.

CADET allows formulas to be written in full first-order predicate logic, this means you can use the usual logical connectives ($\neg$,$\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$) as well as universal ($\forall$) and existential ($\exists$) quantification. The *domain* of the formulas consists of the set of nodes in the derivation tree. Functions can also be defined to help simplify the context-rules. Furthermore each node in the tree has a *type*. The type of a node is the same as the label of the node.

A string belongs to the defined language if and only if the following conditions hold

- It belongs to the context-free language described by the BNF grammar.

- All context-rules evaluate to true when applied to the string's derivation tree[5]

To constrain the language accepted by *Grammar 2* we could add the following CADET context-rule

Rule definedBeforeUse =
$\forall\ v1 : var\ [\exists\ v2 : var,\ d : declaration,\ childOf(v2, d),\ varEquals(v1, v2)]$

Here $\forall\ v1 : var$ is a universal quantification over all nodes in the domain with type *var*.

In [20] a complete specification of Oberon using CADET is presented. The conclusions drawn in that work is that logic as a specification language yields comparatively small specifications, and it is easier to read and understand than e.g. attribute grammars.

2.4 The Languages of the Semantic Web

Currently the Web is relying on the use of HTML to structure documents, however HTML is only defining how the content should be displayed and doesn't say anything about the role of the content. This gives us *semi-structured* data which is difficult, almost impossible, for machines to extract information from.

To alleviate this problem *XML* was introduced. XML allows people to create fully structured documents, i.e. add markup (tags) to the document that indicates the role of the content.

But XML alone is not enough to create truly machine-accesible documents, computers are still limited to transmitting and presenting information on it without being able to process the information. The problem is that the tags used to markup documents are arbitrary strings, and doesn't carry any machine-available semantics.

The *Semantic Web* initiative aims to solve this problem. This initiative will make use of distributed *ontologies* to bring semantics to web resources.

This section first gives a short description of XML, and then introduces the languages used to bring semantics to XML.

2.4.1 Extensible Markup Language XML: The Universal Syntax

The *eXtensible Markup Language* (XML) (see e.g. [34]) is a format for representing data in a machine-processable way. The XML defines a standard way of adding markup of document contents, so a stadandardized syntax is achieved.

HTML specifies a fixed set of tags, and the intended semantics for those tags. For example $< h1 >$ is always interpreted as a first level heading and $< item >$ is meaningless. In contrary to this approach XML only specifies the general form on XML documents (called well-formedness), no predefined set of tags and consequently no semantics. A set of defined tags for some use is called an *application* of XML.

An example of a well-formed XML document is:

[5] From a model theoretic viewpoint a derivation tree is an interpretation of the context-rules. A string belongs to the specified language iff it's derivation tree is a model of the context-rules.

$$
\begin{array}{l}
< items > \\
\quad < car \underbrace{manufacturer = "Volvo"}_{attribute} / > \\
\quad < hat > \\
\quad\quad \underbrace{\underbrace{< size >}_{start\ tag} 12 \underbrace{< /size >}_{end\ tag}}_{Element} \\
\quad < /hat > \\
< /items >
\end{array}
$$

The XML datamodel is a ordered, labelled tree structure[6].

A DTD *(Document Type Definition)* is the equivalent of a BNF notation for XML applications. A DTD defines the valid structure of a well-formed XML document.

Recently the **XML Schema** [33] definition language has been proposed by the web consortium as a new metalanguage for describing the grammar of XML documents[7], it is more expressive than DTDs. But XML Schemas still only allows context-free aspects to be constrained. Furthermore XML Schema has been critized for being to big and complex and other languages such as DSD [8] that is both cleaner and more powerful (some context-aspects are handled) than XML Schema have been proposed. But currently it seems that XML Schema is gaining followers, in spite of its complexity.

XSL Transformations [30] is a language for transforming XML documents to some other format. A XSLT program is also called a *stylesheet*. A stylesheet is made up of a set of *templates* that matches nodes in the input document (in the form of a tree) and transforms them to produce the output.

XSLT is not a general-purpose language, it is a domain language specialized for handling the tree model of XML documents. It lacks things like defining your own functions or types, and there are e.g. no global variables. It is interesting to note that XSLT is defined as an application of XML. This means that XSLT easily can be used for meta-programming.

XML is by many considered ugly and hard to read, however I feel this is misguided criticism since XML is a format for machine-to-machine exchange. No-one should really have to edit XML-documents by hand, instead tools should enable the user to edit content in a suitable way and then export the content as XML. Currently however there is not a great deal of tool support, but this is improving. For example this thesis develops a tool to edit ontologies which will be exported to XML syntax.

2.4.2 Adding Meaning to the Web: RDF(S), DAML+OIL and Above

Currently the World Wide Web only facilitates human consumption of information, although everything on it is *machine-readable* the data is not *machine-*

[6] It is really a graph because you can use attributes to refer to other elements
[7] The XML Schema language is itself an application of XML, DTDs are not.
[8] http://www.brics.dk/DSD

understandable. This entails that it is very hard to automate tasks such as finding the author of a certain web page or determining the topic of a web page, even if this information is available in the page. This basically has to be done using heuristics, with poor results (just think about the search engines).

If we had some way of making the implicit information explicit, it would be possible for software agents to reason about the content (making inferences and such) and perform complex operations.Currently all we can do is use the HTML *meta* tag to add keywords that describe the content of the page, this is just a list of words and thus doesn't help a lot.

The solution proposed by, among others, the World Wide Web Consortium (W3C) is the use of *metadata* to describe the data on the web. Metadata is data about data, or in the context of the web: "data about web resources". The HTML meta tag is a very simple example of metadata, we need more expressive ways of defining metadata.

We should note that the *semantic* in "semantic web" and the use of *machine-understandable* doesn't mean that the software agents will truly "understand" the information, but it will be able to manipulate the information more effectively.

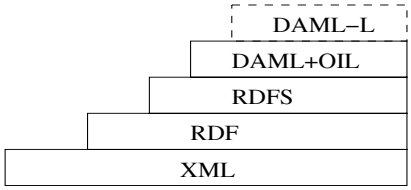

Figure 5: The layering of languages for the semantic web. The language at level l is implemented by the language at level $l-1$. Figure adopted from [7].

RDF (Resource Description Framework) [29] is a simple language defined by the W3C and is used to represent metadata about *resources*. A *resource* is anything that can be uniquely identifiable with a *Uniform Resource Identifier* (URI), an example URI is *http://www.w3c.org* also known as a URL[9]. A URI doesn't have to identify a web resource it could be any kind of object, for example we could assign URI's to cars, atoms, abstract concepts or this document.

Assume we want to add some metadata about this thesis, informally described as:

> The author of the thesis is Johan Lövdahl and it is written in english.

The RDF representation of this metadata would look like:

```
<RDF xmlns     = "http://w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:myOntology =  "http://www.ida.liu.se/\~{}x01johlo/myOntology#">
   <Description about=''#''>
```

[9]The set of URL's is a subset of all URI's

```
        <type resource=''myOntology:Thesis''/>
        <myOntology:author>Johan Lövdahl</myOntology:author>
        <myOntology:language>english</myOntology:language>
    </Description>
</RDF>
```

Here I have used the XML format for RDF descriptions, other formats exists but I will only use the XML format. The initial **RDF** element marks where the RDF description starts and ends in an XML document, it also sets up the namespace. RDF descriptions can be standalone documents (with there own URI) or can be incoroporated into the objects (e.g. webpages) that they describe.

Then the a resource description element comes, the attribute

```
about=''#''
```

states that the description applies to *this* document (the same URI as the description itself). If we want to describe another resource we give the URI for it.

The **description** is the fundamental concept in RDF. A description consists of one or several **statements**, each statement makes an assertion about the described resource. In the RDF example above we have three statements, asserting that the resource is an instance of "Thesis", the author is "Johan Lövdahl" and that it is written in "english".

A statement can be seen has having three parts:

- A *subject*, the described resource (*this* document in the example above).
- An *object*, for example "Johan Lövdahl" and "english".
- A *property* (also called predicate), *author* and *language* are example properties. A property relates the subject to an object. The objects in the example are both string literals, but they can also be URI's.

So these statements are the actual metadata, the **description** element is just a convenient way of grouping statements that has the same subject. Properties are binary relations.

This is all you can do in RDF, assert facts about resources. RDF defines the primitive property called **type**[10] which can be used in a statement to say that the subject is of type object, RDF doesn't define any properties such as *author* or *language*. Those kinds of properties are domain-specific and they are defined with the RDF Schema language.

XML Namespaces are used heavily in RDF(S)/DAML+OIL, here I will give a short introduction since the namespace concept should be familiar to the readers.

The XML namespaces recommendation defines a way of creating universally unique names for XML elements and attributes. A namespace is a collection of tags and attributes. The namespace is referenced by its URI, any element can be referenced by its namespace URI combined with the local name (local in the namespace). We can use a namespace by writing e.g. *xmlns:foo="www.foo.org/" xmlns:bar="www.bar.org/"*, to reference an element in the *foo* namespace we prepend *foo:* to the element tag e.g. <**foo:a**><**bar:a**>...</**bar:a**></**foo:a**>.

[10] A few properties used for *reification* is also defined, but these will not be discussed.

RDFS (RDF Schema) [31] is a language for defining domain specific properties (such as *author* and *language*) as well as classes. The classes can be used to assert that a particular resoure is an instance of this class. This is basically the same thing as providing a *object-oriented type system* to be used with RDF describing resources. RDFS defines resources like e.g. **Class** and properties such as **subClassOf** (which can be applied to resources with **type Class**).

The classes and properties that are defined in the schema is called an *ontology*. An ontology describes the static structure of a domain, i.e. the concepts (classes) found in the domain and how they are related (via properties). An ontology doesn't describe any behavioural aspects of the domain. Unlike a DTD description, which gives specific constraints on the structure of of an XML document, an RDF Schema defines how RDF statements should be interpreted. RDFS can be seen as a set of ontological modelling primitives defined using RDF. RDF is used to describe the specific instance of a domain.

An example ontology (called *myOntology* in the RDF example above) defining the properties **author** and **language** and the class **Thesis** used in the instance example above would look like this:

```
<rdf:RDF xmlns:rdf = "http://w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs = "http://w3.org/2000/01/rdf-schema#">
    <rdf:Description rdf:ID=''author''>
       <rdf:type rdf:resource=''rdf:Property''/>
       <rdfs:domain rdf:resource=''#Thesis''/>
       <rdfs:range rdf:resource=''String''/>
    </rdf:Description>
    <rdf:Description rdf:ID=''language''>
       <rdf:type rdf:resource=''rdf:Property''/>
       <rdfs:domain rdf:resource=''#Thesis''/>
       <rdfs:range rdf:resource=''String''/>
    </rdf:Description>

    <rdf:Description rdf:ID=''Thesis''>
       <rdf:type rdf:resource=''rdf:Class''/>
    </rdf:Description>
</rdf:RDF>
```

As we can see a RDFS schema is itself written as a set of RDF statements, it defines some new resources (*author, language* are asserted to be instances of properties and *Thesis* is a class). These are the resources that are referenced in the RDF description of this document above (we assume that the ontology is located at the URL given in the RDF description). The use of **rdf:ID="XX"** means that a new resource is created that has the URI "baseURI"+"#"+"XX", where baseURI is the URI of the document containing the **ID** statement.

The most important constructs defined by the RDFS language are: *Class, Property, subClassOf, subPropertyOf, domain* and *range*. The *domain*, *subPropertyOf* and *range* properties applies to resources of type *Property*. The *subClassOf* property applies to resources of type *Property*. The properties are also called *axioms*.

The *subPropertyOf* property allows to define a hiearchy for properties, this is different from ordinary type-systems for programming languages. Another difference from type-systems is that RDFS properties are defined in terms of

the classes of resources they apply to[11], properties can have several *domains* and one *range*, denoted by URI's. Furthermore properties are first-class entities, this means that we can make statements about properties. The possibility for anyone to say anything about any resource is one of the central ideas of the web.

The success of the semantic web depends on the emergence of *shared* ontologies for understanding. Only when two or more agents use the same base ontology we can achieve the interoperability that we desire. Notice that two agents do not need to refer to the same ontology as long as the concepts defined in those ontologies are related (through subClassOf/subPropertyOf etc.) with some common upper level ontology. There will of course be many ontologies on different levels of granularity, some ontologies will describe very particular domains such as postal codes, programming languages etc. and others will describe very general concepts such as *abstract,concrete,animate, inanimate*. When the specialized ontologies define their concepts in terms of these general ontologies we can achieve atleast partial understanding between agents.

DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer) [6] is another ontology description language. The reason for introducing a new language is the limited expressiveness of RDFS. For example in RDFS you could state that the class *Person* is a subclass of the class *Animal*, but this is a very weak assertion. DAML+OIL extends RDFS by adding more powerful modelling primitives such as axioms stating equivalence between classes or properties, disjointness between classes etc.

The DAML+OIL language can be divided in two parts, the *class constructors* (which I will call operators) (see table 2) and the *axioms* (table 1. Both the operators and the axioms are resources defined as properties. These kinds of operators/axioms allow inference engines to deduce more complex information from an ontology.

Section 4.2 gives the abstract syntax of DAML+OIL ontologies, as well as more discussion on all the language constructs. Here I will only introduce the modelling primitives, other auxillary constructs exists.

The axioms are basically the same as in RDFS, with some additions. The *sameClassAs/samePropertyAs* properties asserts that two classes/properties are the same, i.e. that they must have the same extension. The *disjointWith* asserts that two classes can not have any instances in common.

The *inverseOf* property asserts that a property is the inverse of another property. If we assert that *parent* is the inverse of *child*, and given an RDF instance asserting that i_1 is the *child* of i_2 an inference engine can deduce that i_2 is the *parent* of i_1. An example DAML+OIL only defining the property *parent*:

```
<rdf:RDF xmlns:rdf  = "http://w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs = "http://w3.org/2000/01/rdf-schema#"
         xmlns      = "http://www.daml.org/2001/03/daml+oil#">

     <Ontology rdf:about=''#''>...</Ontology>
```

[11] This is different from the usual way in OO-languages to define classes in terms of the properties it has.

Axiom	Meaning
subClassOf	$C1 \sqsubseteq C2$
subPropertyOf	$P1 \sqsubseteq P2$
sameClassAs	$C1 \doteq C2$
samePropertyAs	$P1 \doteq P2$
disjointWith	$C1 \sqsubseteq \neg C2$
inverseOf	$P1 \doteq P2^{-}$
sameIndividualAs	$\{x1\} \doteq \{x2\}$
differentIndividualFrom	$\{x1\} \sqsubseteq \neg \{x2\}$

Table 1: *The class constructors for DAML+OIL.*

```
      <rdf:Property rdf:ID=''father''>
        <inverseOf rdf:resource=''child''/>
        <rdfs:domain .../>
        <rdfs:range .../>
      </rdf:Property>
      ... Introduction of the 'child' property ...
<rdf:RDF>
```

Notice that the DAML+OIL is itself an RDF description introducing new resources that are instances of RDFS/DAML+OIL types. The notation <**rdf:Property rdf:ID="..."**> is a shorthand for the equivalent <**rdf:Description ...**> <**rdf:type rdf:resource="rdf:Property"**/>.

The first thing we do is to assert that this document is a *Ontology* (this is a class defined in the DAML+OIL language), this is not necessary but it is good practice. Then we introduce a new resource called *father*, and assert that it is the inverse property of *child*. The DAML+OIL language defines its own *domain/range* properties (asserted to be *samePropertyAs* the RDFS range/domain), it doesn't seem to matter wether you use the RDFS or DAML+OIL version of these properties, but it seems to be common practice to use RDFS/RDF when possible instead of the equivalent DAML+OIL properties/classes.

The axioms part of DAML+OIL is a modest extension of RDFS, the real difference is the *class constructors* (table 2). Class constructors are *operators* that create *anonymous* classes from other classes and/or property restrictions, the axioms can then be used to assert equivalence/subclassing to define named classes. The class constructors basically gives a limited form of predicate logic to create the classes.

The *complementOf* operator creates the class which is the complement of the referenced class. The *unionOf/intersectionOf* operator creates the union/intersection of a list of classes. The *oneOf* is used to define a class by enumerating its individuals.

The operators *toClass/hasClass* are used to restrict a given property's range to the given class. *hasValue* creates the class of all objects having the given value for the given property. We can also use cardinality restrictions on properties to

create classes, if we use the *cardinality* operator on a property, P, we get the class of all resources for which P has exactly N distinct values. The qualified cardinality restrictions not only restrict the cardinality but also demand that the values must be of the supplied type.

Furthermore a few subclasses to the *Property* class is defined. *Transitive-Property* means that the declared property is transitive. *UniqueProperty* means that the property only has one object for each subject. *UnambigousProperty* indicates that an object can only be the value of one subject. These operators

Constructor	Meaning
complementOf	$\neg$C1
unionOf	C1 $\cup \ldots \cup$ CN
intersectionOf	C1 $\cap \ldots \cap$ CN
oneOf	{x1, ..., xn}
toClass	$\forall$P.C
hasClass	$\exists$P.C
hasValue	$\exists$P.{x}
minCardinalityQ	$\geq$nP.C
maxCardinalityQ	$\leq$nP.C
cardinalityQ	$= nP.C$
minCardinality	$\geq$nP
maxCardinality	$\leq$nP
cardinality	$= nP$

Table 2: *The class constructors for DAML+OIL.*

can be combined in arbitrary ways using the intersection and union operators. DAML+OIL is based on research in *description logics*, the operators added in this layer are a subset of predicate logic that provides decidable reasoning (see [10] for the relation between DL and DAML+OIL). DAML+OIL uses this subset of predicate logic because there exists efficient inference engines that can be used. If a more expressive logic layer was built on top of RDFS we would immediately get computational problems.

In section 6.1 I discuss what can be expressed with DAML+OIL and what can't be.

DAML-Logic/Rules will be the next layer in the 'cake', it will provide logic rules so one can make logical assertions about the concepts and relations in the ontology. These rules extend the decidable logic layer of DAML+OIL and thereby extend the possibilities to reason about individuals in domains. The DAML-L language has not yet been defined, there are still open questions about e.g. the expressivity. It will most likely be based on the Horn clauses (which is a subset of predicate logic) used in Prolog systems since efficient engines exists for Horn clauses. An example XML application for rules is RuleML [3].

2.5 UML: A Visual Modelling Language for Object-Oriented Systems

The synthesis of complex artifacts usually benefits from the use of models. For example construction of large buildings cannot be undertaken without first creating a blueprint for the building.
The use of models enables people to discuss and think about artifacts that might otherwise be too complex to understand entirely, it also gives us a way of checking that the produced artifact is correct.

The software engineering activity is no exception when it comes to need for modeling.

The first part of this section describes the basics of UML, a standardized[12] visual modeling language used for building models of object oriented systems. UML offers a few different views called *diagrams* to the user. Here I will only touch on *class* and *object* diagrams, for a complete overview of the different diagrams see [19].

After that OCL is very briefly described. OCL is a predicate logic language used to express invariant constraints on the UML models. Although OCL itself isn't really used any further in the thesis it is needed to understand the use of UML in the thesis.

And in the end the standard XML format for UML models, XMI, is described.

2.5.1 UML Class Diagrams: A View of the Classes and their Relations

In object-oriented programming the concept of a *class* is central, so naturally UML offers class diagrams to the user.

Class diagrams show classes and their relations to each other in the modelled system. Figure 6 shows a class diagram that uses the basic UML modelling primitives.

- *Classes* are represented by boxes with three parts[13]: the name of the class, the attributes of the class and the operations of the class. The UML meaning of a class is the same as in an oo-language such as Java.

- *Generalization* is represented by solid lines with large hollow arrow heads pointing to the super class, e.g. classes *Car* and *Vehicle* in figure 6. Generalization has the same meaning as *inheritance* in Java or C++, UML allows multiple inheritance.

- *Association* is represented by solid lines between two classes, e.g. *Person* and *Car*. The ends may be named, as in the sample diagram: *driver* and *drives*, these names are called *roles*. Associations can also have names, this is not shown in the example.

- *Aggregation* is represented as an association with a diamond at one end of the line. Aggregations are specialized associations.

[12] Standardized by the Object Management Group (OMG). http://www.omg.org
[13] A box with two parts represent *interfaces*.

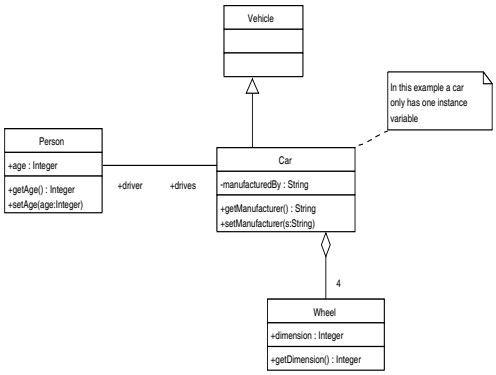

Figure 6: *A sample UML class diagram.*

The ends of association and aggregation relationships can be annotated with multiplicity constraints, e.g. in figure 6 the aggregation states that *Car* has exactly four *Wheel*'s and that each *Wheel* can only be contained in one *Car*. If no multiplicity is given "1" is assumed as default. The rectangle with a folded corner is a *note*, it contains comments (in plain text).

There are several other constructs in UML, but these are perhaps the most basic. When I use class-diagrams further on in the thesis I will introduce any new constructs that are used.

Object diagrams are graphs where the nodes are instances (objects) of the classes in the corresponding UML class diagram, the edges are instances of the associations in the class diagram. An object diagram shows the detailed state of a system at one point in time, the diagram is made up of *objects* and *links*. We can refer to an object diagram as a *snapshot*. Class diagrams define the valid forms of object diagrams.

2.5.2 OCL: A Textual Constraint Language for UML Models

There are many invariant constraints[14] for classes and their relationships that can't be expressed with this notation. For example simple multiplicity constraints can be expressed, but more complex constraints like *"The driver of a car has to be atleast 18 years old"* cannot be expressed only using the basic vocabulary of UML. To achieve this an additional language was adopted, the *Object Constraint Language* (OCL) (chapter 7 in [19]). OCL is part of the UML standard and is intended to be used in conjunction with UML class diagrams. OCL constraints express more complex invariants on the object diagrams.

OCL is a typed first-order predicate logic language. The typedness means that all OCL expressions have a type. OCL can only be used to specify constraints on classes, it can't be used as a ordinary programming language.

[14] A *invariant constraint* is an expression that must hold for all instances of the

The constraint *"The driver of a car has to be atleast 18 years old"* would in OCL be written as:

context Car **inv:**

$self.driver.age >= 18$

This invariant would then be evaluated for each object of type Car in the object diagram.

Each OCL expression is written in the *context* of an instance of a specific type. The reserved word *self* is used to refer to the contextual instance. The *inv* reserved word indicates that the expression is an invariant, *pre* and *post* could be used to write pre/post conditions on methods.

2.5.3 XML Metadata Interchange (XMI): Standard Serialization Format for UML Models

When engineering large complex software systems it is common to use a number of different tools, Computer Aided Software Engineering (CASE) tools, for instance design (UML editors), browsers, report generators and reengineering tools. Defining standard saving formats for these tools enables them to work together. XMI is such a standard.

Basically XMI is a standardized XML application[15] for UML models.

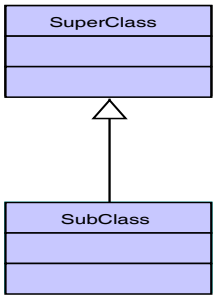

Figure 7: *A simple class diagram showing inheritance between the SuperClass and the SubClass.*

The XMI representation of a UML model is very verbose (hence the file size is large), this is because XMI format is designed for any meta model which can be described in MOF. An XML format specifically designed for UML models could have been less verbose. Given the UML diagram in figure 7 the Poseidon UML tool (see appendix[REF]) will produce the XMI representation in figure 8. Note that the XMI format only contains "logical" information about the model, e.g. the classes and their relationships, tool-specific information such as the visual representation of the classes is specified in another file.

During my work I used three different UML editors, and one of the findings is that XMI is not very well supported. XMI is a recent standard so it will probably take a while before it is implemented by all tools (if ever!).

To read this thesis it is only important to know that XMI is a standard XML format for UML models, the details are not further considered.

[15] *application* in the sense that XMI defines a set of tags

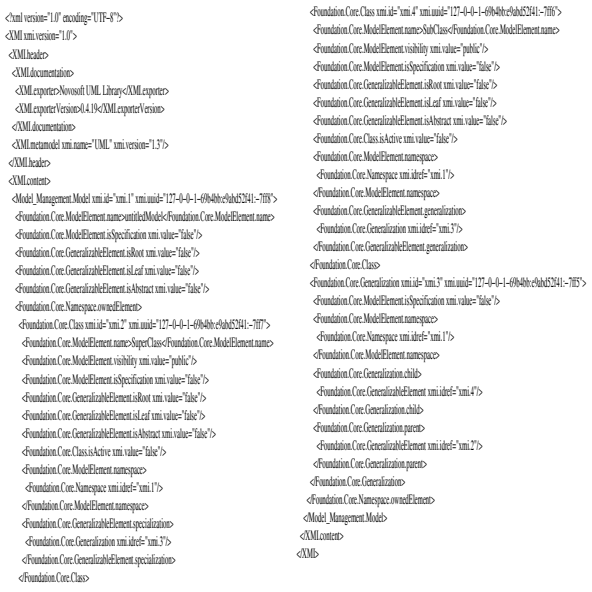

Figure 8: XMI representation for the class diagram in figure 7.

2.6 Constraint Diagrams: Visualizing Constraints

Constraint diagrams is a visual notation for expressing constraints on UML models[16]. The notation was first introduced in [12] and has been explored further in [9],[8] and [14]. Pairs of constraint diagrams have also been used to express pre/post conditions for methods in a visual form [13]. The UML already has a standard language to express constraints on models, OCL, so why do we need constraint diagrams? The number one reason is that OCL can be difficult to understand if you do not have the proper background in logic and mathematics, constraint diagrams let you draw circles and arrows which is potentially more intuitive.

The constraint diagram notation is still being explored, and it seems to change slightly from article to article. Open questions are e.g. the expressibility of CD's compared to OCL/Predicate logic, defining the semantics of the diagrams,how intuitive are these diagrams to the users, etc. Currently one tool exists for editing CDs: the **CDEditor** (see appendix C).

In this introduction I will use the notation supported by the CDEditor, and it is also this notation I have used as a basis in my work. The constraint diagram notation can be divided into two parts, the *spider diagrams* (SDs) and the full blown *constraint diagrams*.

Spider diagrams is a sub-notation of the CDs, they are based on *Venn diagrams* and *Euler circles*. Figure 9 shows the basic concepts in spider diagrams.

[16] Actually constraint diagrams can be used in conjunction with other OO modelling notations, not only UML

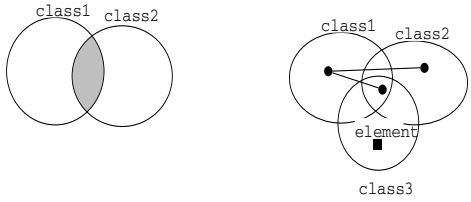

Figure 9: Two examples of spider diagrams. The first shows that the intersection between the two sets is empy. The second demonstrates two spiders.

- *Sets* are represented by circles called *contours*. The set label (e.g. *class1, class2* in figure 9) determines the type of the elements in the set. A set consists of all the objects in a object diagram with the specified type.
- *Spiders* are represented by the small "dots" shown in figure 9 they are called *motes*. Spiders are used to denote elements of sets. A spider can be a single mote (as the one labelled *element* in the example), or a set of motes connected by solid lines (creating a tree) these are called *articulated spiders*. Square motes denote a particular element in the set, round motes asserts the existence of atleast one element in the set.
- *Projections* (not shown in the example) are represented by circles with dashed outlines. Projections are used to show the intersection of more than three sets. This is otherwise difficult to do.

The topology of the contours is used to express relationships between sets (intersection, containment and disjointness). The spiders are used to express statements about the elements in the contours.

The left diagram in figure 9 shows the intersection between two sets, the shading indicates that the intersection must be empty. If seen as an invariant on UML class diagrams it would mean that there cannot be any objects with type $class1 \wedge class2$.

The diagram on the right shows:

1. One named spider, labelled *element*. The intended meaning is $element \in (class3 - class1 - class2)$. Named spiders denote a particular element in the set it occurs.
2. An articulated spider, these are used to denote disjunctions. The intended meaning of the multi-set spider is: $\exists x \; x \in ((class1 \cup class2) - class3 - (class1 \cap class2)) \cup (class1 \cap class2 \cap class3$

The intended meaning of the entire right hand diagram in figure 9 is the conjunction between 1 and 2 above.

For spider diagrams (without projections) a formal semantics has been defined in [9].

Constraint diagrams extend the spider diagram notation with two concepts: *arrows* and *universal motes*.

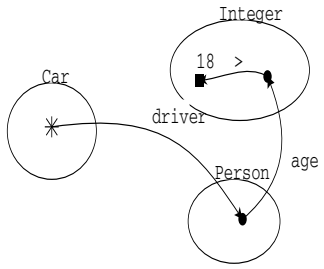

Figure 10: Constraint diagram expressing the age invariant on the UML diagram in figure 6, see section 2.5.2 for the equivalent OCL invariant.

The example diagram (figure 10) shows how the example OCL constraint in section 2.5.2 can look in CD notation. The '*' in the *Car* contour represents universal quantification over the members of the set.

- Arrows have a label, a source and a target. The source can be a contour or a spider. The target can be a contour or a spider[17].
- Universal quantification motes, represented by wildcards. Universal motes can't be the target of an arrow.

Problems with constraint diagrams will be discussed in section 6.3.

2.7 Frame Logic: An Object-Oriented Logic

Frame logic [15], abbreviated *F-Logic*, is a formalism that integrates the *deductive programming paradigm* with the *object-oriented data representation paradigm*. A sound and complete resolution-based proof algorithm has been developed for F-Logic, and an interpreter, *Flora-2* (see appendix C, exists.

So basically F-Logic is a logic programming system like *Prolog* (see [28] for introduction), but with additional object oriented features. F-Logic doesn't use the ordinary prolog syntax, but the F-Logic rules are formulated as Horn clauses. A Horn clause always looks as:

$$A_0 \leftarrow A_1, A_2, \ldots, A_n$$

Where each A_n is a n-ary predicate with constants or variables as parameters, predicates may be negated. Horn clauses with $n \geq 1$ is called a *rule*, if the predicates on the right-hand side of the arrow are true the left-hand side is asserted to be true. A Horn clause that only consists of A_0 is called a *fact*, a

[17] In this work I will only allow spiders as the source and target, not contours

fact is always true. In the case of F-Logic each A_n is also refered to as a *F-Logic atom* and it basically looks like

$$O[M_1 -> V_1, \ldots, M_n -> V_n]$$

This means that the *methods* M_i are applied to the object O and yields the values V_i.

An interesting extension to the ordinary prolog expresiveness is the addition of *higher-order logic*. This means e.g. that variables can have predicates as values. This is very convenient for the programmer, but perhaps it doesn't add any expressiveness above the horn-clauses. The Flora system translates all F-Logic statements to ordinary prolog and uses the XSB prolog system to do the computations.

F-logic supports among other features: types (classes), objects, (multiple) inheritance etc. An example of F-Logic syntax:

```
%% subclass relation
person :: animal.
male :: person, female :: person.
%% Method declarations. father is a single-valued method, sons is multi-valued
person[father => person, sons =>> person, children =>> person].
%% Derivation rule
X[sons->> Y] :- X[children->> Y], Y : male.
%% Declare some objects (i.e. facts)
bob:male[children-»rita, fred.
rita:female, fred:male.
%% Query: get bob's sons
?- bob[sons-»X].
X = fred
```

The order of the methods are irrelevant when applied to an object (they are not positional like parameters in Java).

The class declarations and inheritance specifications are collectively called the *schema*.

The XSB system also introduces the concept of *modules*, this is not usually part of prolog systems. A module is a namespace in which the programmer can put schemas and rules. As an example, if you want to print a string on the console you would evaluate the relation *write* as follows:

```
?- write('this is a string')@prolog().
```

The '@' sign is used to select the module. Since Flora is built on top of XSB this feature is also part of Flora.

3 Visual Ontology Editing for the Semantic Web: The Architecture

3.1 Architecture of the SWEDE

The current tools for creating ontologies are text-based. The most well-known tools such as Protege[18] and OntoEdit[19] provide a graphical user interface with tree structures to navigate among the concepts and relations.

These kinds of tools (in particular Protege) have been around for quite some time in the Artificial Intelligence field, used by e.g. the description logics community, to create ontologies. However with the arrival of the semantic web ontology editing will done not only by people who have experience from AI communities. It would therefore be desirable to reuse tools and notations from various domains so experts in those domains can create domain ontologies without a too steep learning curve.

In software engineering UML is perhaps the most well-known notation (see section 2.5). It has a large user base in industry and available tools. Moreover there is a rather strong analogy between ontologies and UML class diagrams (see 4).

In my work I have developed an environment for editing ontologies represented in DAML+OIL, it is based on UML class diagrams complemented by constraint diagrams. I have choosen the acronym *SWEDE* (Semantic Web EDiting Environment), figure 11.

The UML class diagrams are used to represent ontologies in the DAML+OIL language and the constraint diagrams have been used as a visual notation for (typed) Horn clauses with binary predicates. The SWEDE allows a subset of the UML notation for DAML+OIL to be translated to the F-Logic language (XMI2FloraSchema in figure 11, it just translates the subclass property), section 5 discusses how full DAML+OIL can be translated. A translator from constraint diagrams to F-Logic rules and invariants has also been implemented (CD2Flora in figure 11), section 6.3 discusses this. A translator from XML documents to F-Logic facts has also been implemented (XML2FloraDB in figure 11. The generated F-Logic rules/invariants can then be loaded in the Flora-2 F-Logic engine and use the generated facts (instance document) to make inferences.

The SWEDE architecture tries to reuse as much existing notations and tooling as possible and no actual *tools* have been implemented, only the programs translating between the different saving formats of the tools.

The UML language has many existing editors, I have tried three different: Rational Rose, Argo UML and Poseidon UML (see C). An important feature that I required from the editor was the possibility to export class diagrams in XMI format. Argo and Poseidon UML could to this, but Rational Rose had poor support[20]. The use of XMI as the format for the class diagrams ensures that

[18] http://protege.stanford.edu/index.shtml

[19] http://ontoserver.aifb.uni-karlsruhe.de/ontoedit/

[20] A plugin for XMI exists but can only be run on windows platforms. A opensource project (http://crazybeans.sourceforge.net) can take Rose format and translate to XMI, but it is still buggy.

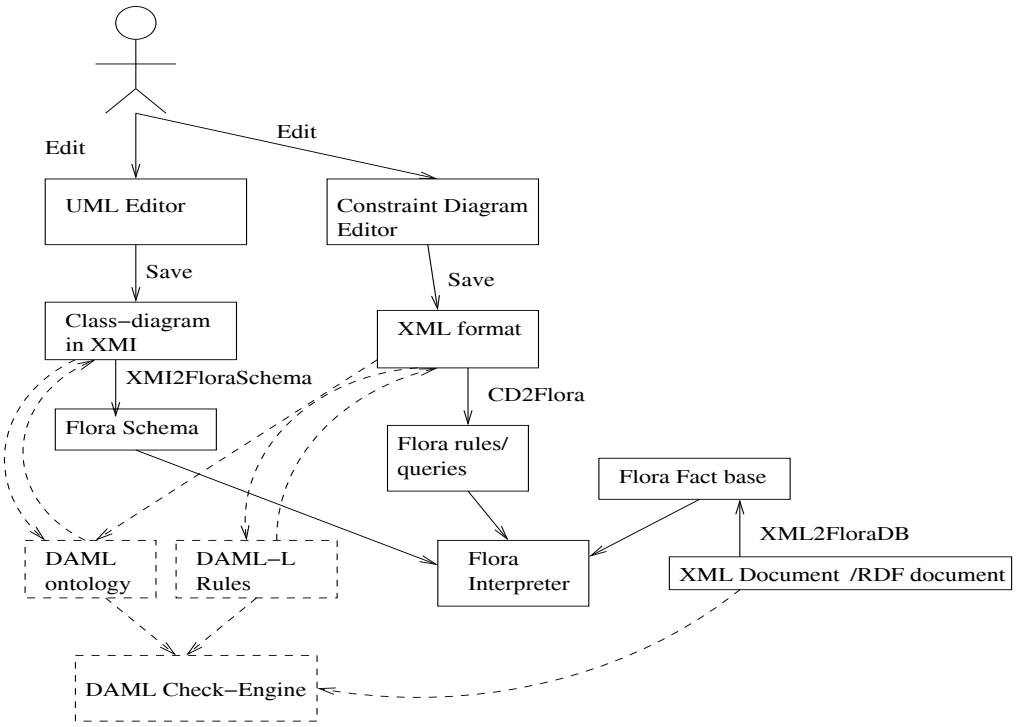

Figure 11: Architectural overview of the semantic web editing environment (SWEDE).

the SWEDE isn't coupled to any particular UML editor, any editor supporting XMI will work (I have used Poseidon due to its stability).

The Constraint Diagram Editor is also a existing tool that is capable of saving the graphical diagrams in an XML format. This cannot be considered as a *standard* way of writing invariants/rules (I have to make extensions to the notation to facilitate rules), but it is a nice graphical representation compared to the OCL.

XML has been used as an intermediate format both for class and constraint diagrams. This allows us to use standard tools (parsers etc.) for manipulating these representations. It also enables the SWEDE architecture to check the static semantics of the SWEDE (or atleast the class and constraint diagrams), this has however not been done in this thesis.

The Flora-2 F-Logic engine is used as the underlying inference engine. The reason for this is the straightforward mapping of many DAML+OIL concepts to F-Logic. Another important reason is the possibility to extend DAML+OIL with Horn clause rules. If we would use a description logics engine to perform DAML+OIL reasoning it would be difficult to integrate Horn clause rules into

the system. Of course the use of a prolog implementation for DAML+OIL doesn't give us the nice computational properties we would have if a special purpose algorithm had been used.

Furthermore the purpose of the thesis was not to implement a full-fledged inference engine for DAML+OIL, it was to develop an graphical ontology editing environment and use it to evaluate static semantics rules.

The dotted lines shown in the overview figure (11) indicates that when a 'real' inference engine for DAML+OIL emerges, we only have to exchange the translators and not the notation.

As described in section 2.4.2 the instance data is described by RDF statements, this is necessary when we can use arbitrary properties and classes to describe the data. But when we are using the ontologies to describe static semantics rules for abstract syntax/derivation tree we can actually use ordinary XML representations. Section 7.1 describes why this is so. This is why I have developed the *XML2FloraDB*, the translation from RDF statements to F-Logic facts is simpler than the XML translation so this could be done.

It is important to note that even though the example applications use the SWEDE as a environment for describing the static semantics of languages, the architecture is intended to be used in a general setting.

4 Visual Editing of Ontologies using UML

4.1 Existing Approaches

The analogy between ontology languages and UML has been noticed by other people as well. Since DAML is quite a new language there are not any ready-to-use UML profiles for it, but there will be in the future. There are a few different groups working on a mapping between DAML and UML, there also exists a Visio profile for DAML.

In articles by Cranefield et. al. [24, 23] a mapping from UML class-diagrams to RDFS is defined.

The mapping has been implemented by processing the XMI representation with a XSLT stylesheet. The articles just show a tiny example ontology and the mapping is not explicitly specified in any article, but by looking at the XSLT stylesheet one can notice the following problems:

- It is not possible to express the *subPropertyOf* relation in the UML diagram. It seems that the RDFS properties are not treated as first-class entities in the UML notation. This simplifies the problem greatly since the major mismatch between UML and RDFS (and DAML) is the status of the properties. Properties are modelled as UML *associations*. This is problematic e.g. when you just want to declare a property without defining the range and domain. And only a subset of the RDFS constructs are handled.

- The distributed nature of RDFS (and DAML) is ignored. Since properties and classes used in an ontology definition can reside in another ontology

somewhere on the web it is necessary to specify the URI for the property/class, this is not handled in the mapping.

If the goal of Cranefields work is to map legacy UML diagrams to RDFS this might be sufficient. Of course this approach doesn't give us a tool for general ontology construction.

Although Cranefield's articles provided a starting point it is not sufficient for my goal. DAML+OIL is more difficult to map to UML since it has more complex ways of constructing classes and other complex constructs.

The work described in [1], by the UBOT project, is aimed at developing a UML profile for DAML+OIL. The article investigates the differences between UML and DAML and they reach the conclusion that there are two problematic mismatches between UML and DAML.

- Properties are first-class entities.
- Property restrictions.

The reason for this mismatch is of course that UML was designed to model object-oriented programming languages such as Java/C++/SmallTalk, and the property features above are not part of those languages.

This approach assumes that the UML class-diagrams have been created for the purpose of being treated as DAML ontologies. Legacy diagrams are not considered. The UBOT mapping defines almost the whole DAML+OIL language in terms of existing UML constructs (and UML's extension mechanisms). The missing pieces are the property and restriction constructs in DAML, due to the mismatches explained above. The recommendation from the UBOT report is that the UML metamodel should be extended with new constructs to overcome these obstacles. Furthermore the namespace problem doesn't seem to be handled.

This article was published after I had finished my own mapping and I therefore didn't benefit from the experiences put forward in it.

4.2 The Mapping from DAML+OIL to UML

4.2.1 General Approach

The goal was to create a mapping between DAML and UML that complied to the current UML specification and available UML tools. To do this I used the UML extension mechanims, *stereotypes, tagged values and constraints*. One of the original goals was to create icons for the new DAML+OIL stereotypes, but the lack of tool-support for stereotype icons made this impossible. Therefore the UML notation might seem unecessary ugly in some places, but I have tried to point out where icons could alleviate these problems.

When defining a mapping between two languages, one has to think about the trade-off between the complexity of the program that will do the translation and the "naturalness" of the mapping. If one wants a simple translator the structure of the UML constructs must be very close to the structure of the DAML+OIL constructs. With a more complex translator we can make the UML profile more natural. Since I always want the tools to do as much as possible for me I have chosen the latter alternative.

A potential problem with this approach is that people aware of the DAML+OIL language in some other notation might find it confusing when there is no one-to-one mapping between the notations. But this is a minor problem.

Note that the goal in this work is to enable ontology developers to use UML tools to develop ontologies from scratch, we are not interested in taking legacy UML class-diagrams and translate them to DAML.

In the following sub-sections I will present the abstract syntax of DAML+OIL as a collection of UML diagrams, and show how each DAML+OIL construct can be mapped to a corresponding UML construct or idiom. To show that this mapping is more or less complete it will be necessary to give a detailed explanation of DAML+OIL, this will be done in the sub-sections. It is worth noting that the mapping that is defined can also be used to create RDF Schemas, since DAML+OIL is a superset of RDFS.

The grouping of language constructs has been based on the grouping done in the DAML+OIL reference description.

Further on in this document I will refer to the UML profile for DAML+0IL as *DAML+OIL/UML*, the notation DAML+OIL/XML indicates that the DAML+OIL ontology is in XML format.

4.2.2 The Ontology Declaration

The diagram in figure 12 shows the starting point for the DAML abstract syntax. Abstract classes (the class name is in italics) denote the non-terminals of the grammar. The "normal" classes denote terminal elements of the grammar, i.e. the name of the class is an actual keyword in the DAML+OIL language. The subclass and aggregation arrows have the ordinary UML meaning. The abstract syntax tree corresponds to the latest release of DAML+OIL as described in [6].

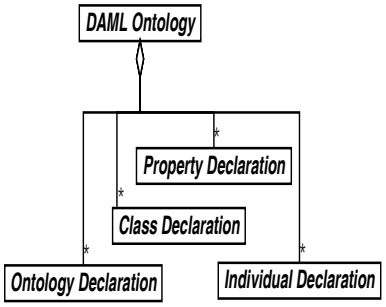

Figure 12: *Abstract syntax for DAML ontologies*

A DAML+OIL ontology is represented by a class-diagram. The property and class declarations are described in the next sub-sections.

The DAML+OIL specification allows *individual declarations* to be part of an ontology, but it is not commonly agreed that indiviudals should *not* be included in the ontology definition. I intended to ignore the individuals in this mapping, but one can notice the strong analogy between DAML+OIL instances and UML object-diagrams. Object diagrams represent the instances of class-diagrams, so

this could probably be a nice way of defining individuals visually. See the section on *Class Definitions* for the use of object diagrams.

The *ontology declaration* non-terminal is expanded in figure 13.

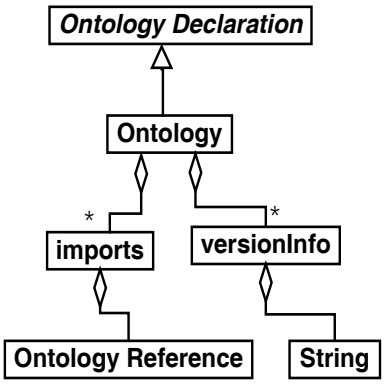

Figure 13: *Abstract syntax for DAML ontology declarations*

A ontology declaration consists of a assertion that the document has type **ontology**. In XML this would look like:

```
<daml:ontology rdf:about=''#''>
   <daml:versionInfo>This is the third revision of this ontology</daml:versionInfo>
   <daml:imports rdf:resource=''http://www.daml.org/2001/03/daml+oil.daml''/>
</daml:ontology>
```

The UML contains a construct that is similar to the DAML+OIL ontology, namely the *package*. So we'll use the UML package construct stereotyped with **Ontology**. To justify the stereotype it is necessary to dissect the example above. The RDF specification defines that the above definition is a shorthand of the following definition:

```
<rdf:Description rdf:about=''#''>
<rdf:type rdf:resource=''daml:Ontology''/>
    ⋮
</rdf:Description>
```

The difference is that the **type** property is made explicit, the two examples have the same meaning. As we will see in the sections on property and class declarations this shorthand is usually used. Since most resources only have one type, we can define a general notation rule:

> *the DAML+OIL type of the resource is used as the stereotype of the UML element for that resource.*

Resources can have more than one type, see next sections how this is handled. However ontologies can only have one type (the **ontology** type) so this is not a problem here.

When we use the ontology stereotype it is possible to use ordinary packages without stereotypes to structure the content of the ontology. If we e.g. have a large ontology we can split it in several UML packages, but they are still logically in the same ontology.

As shown in the DAML/XML example above and the abstract syntax in figure 13 the ontology type has two predefined properties **versionInfo** and **imports** statements.

The **versionInfo** property is modelled as a UML tagged value for the ontology, this is possible since the range of the property is a string.

The **imports** property is an example of a more complex property. The domain is a reference to another DAML+OIL ontology. The use of these complex DAML+OIL properties are modelled as associations with the name of the property as the stereotype. We can define this as another notational rule:

> *The use of complex DAML+OIL properties are denoted by directed UML associations where the name of the property is the stereotype of the association.*

The **imports** property is used to include definitions specified in other ontologies. The example ontology declaration above would be shown in DAML/UML as:

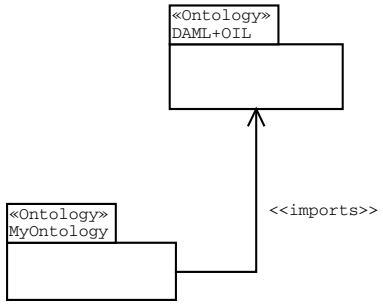

Figure 14: *An ontology declaration and the DAML+OIL* ***imports*** *property.*

4.2.3 Class Definitions

When we have created a ontology package we can start adding definitions to it. The defined DAML+OIL properties for class definitions are showed in figure 15.

A simple DAML+OIL/XML example of class definition is:

```
<daml:Class rdf:ID=''Female''>
   <rdfs:subClassOf rdf:resource=''Person''/>
</daml:Class>
```

This will define a new class called *Female* and defines it to be a subclass of the class *Person*. Obviously the DAML+OIL **Class** type has a corresponding concept in UML, the UML Class. If we follow the rule that was defined in the previous section we should use a **Class** stereotype for the UML class, but I think the analogy between the concepts is so strong it is not necessary with any sterotype at all, so I'll deviate from the notation rule in this case.

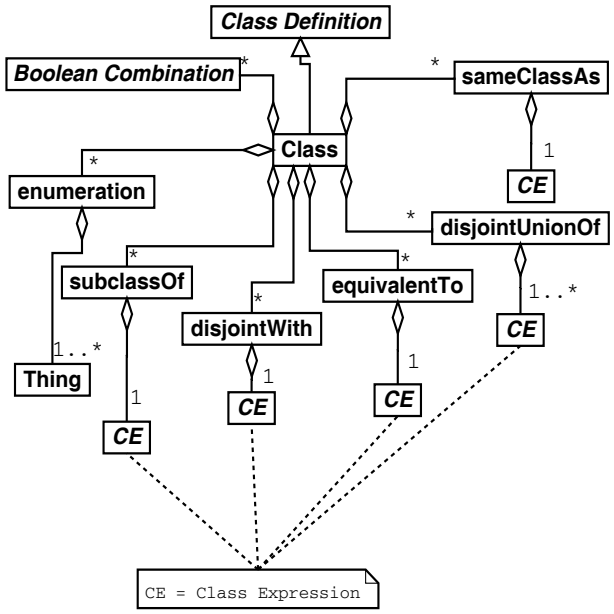

Figure 15: *Abstract syntax for class definitions*

The subClassOf property used in the example is a complex property and thereby covered by the rule defined in the previous section (directed association stereotyped with **subClassOf**. The DAML+OIL/UML for the example:

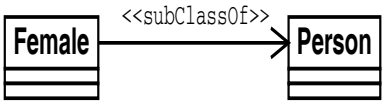

Figure 16: *The DAML+OIL* ***subClassOf*** *property.*

Of course UML has a standard icon for the subclass relationship and this can be used as a stereotype icon for this property.

One feature of DAML+OIL is, as I have mentioned before, the decentralized nature of definitions. This means that a definition (like the class definition above) not necessarily contains the whole definition, we can add more information by referencing an existing definition. This would be done as follows in DAML/XML (I'll assume that the *Male* class is defined).

```
<daml:Class rdf:about=''Female''>
   <daml:disjointWith rdf:resource=''Male''/>
</daml:Class>
```

Here we have used the **rdf:about** identification mechanism which means that an existing resource is being referenced. The first definition of *Female* used **rdf:ID** which introduces a new resource identifier. The logical interpretation of

this decentralized definition is the same as if we had collected all the properties in one definition. In the example I have given a decentralized definition of a class where both the original definition and the "extension" resides in the same ontology. This might seem pointless, the real usefulness shows when the original definition resides in another ontology which is imported and then the extended.

This decentralized definition means that we have to add something to the UML classes so we can distinguish between the definition of a new resource and the extension of an existing resource. If, for example, the *Female* class definition is given in ontology A which is imported by ontology B, and in B a class named *Female* is introduced (by a UML class) we can't tell what the intended meaning is. Either it means that the property in A should be extended or that a new property with the same name should be defined. To separate between these two cases I chose to introduce a new tagged value, **extends**, which takes the namespace (as a string) of the property that should be extended. If we want to define a new class, or if we want to extend a property in the same ontology we simply ignore the tagged value.

The **disjointWith** property is a complex property so we use the general notation rule.

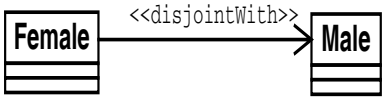

Figure 17: *The DAML+OIL* ***disjointWith*** *property.*

The **subClassOf, disjointWith, equivalentTo** and **sameClassAs** properties all follow the general notation rule for complex properties. Each of these properties have *one* **Class Expression** as their range (see next section). The intended meaning of these properties should be obvious, the **sameClassAs** and **equivalentTo** properties are basically the same.

It gets more complicated with the remaining properties **enumeration** and **disjointUnionOf**, these properties can have arbitrary many values.

The **enumeration** property asserts that the defined class contains exactly the set of individuals that is enumerated, no more and no less. The DAML+OIL specification allows more than one enumeration per class definition, but this seems pointless since each enumeration must be the same or an error will occur. DAML/XML example of **enumerations**

```
<daml:Class rdf:ID=''Boolean_Values''>
  <daml:oneOf parseType=''daml:Collection''>
    <daml:Thing rdf:resource=''True''/>
    <daml:Thing rdf:resource=''False''/>
  </daml:oneOf>
</daml:Class>
```

Where *True* and *False* are individuals (of type emphBool perhaps), all individuals automatically have the type **Thing** (This is analogous to the Java *Object* class, instances of any class are always Objects). **Thing** is a class defined in

the DAML+OIL specification together with the **Nothing** class (no individuals can have the type **Nothing**).

The **disjointUnionOf** property asserts that the defined class has the same extension as the disjoint union of the given set of **Class Expression**s.

These properties are not binary, causing some trouble since the UML associations we have used for properties indeed are binary. The UML specification [19] defines *n-ary associations*, so the obvious choice would be to use use that. But none of the UML editors I have used support n-ary associations, so we have to find an alternative.

I decided to use a new stereotype called **collection**, this is applied to a UML class. The **enumeration** and **disjointUnionOf** are then drawn between the defined class and the **collection** class (which doesn't need a name, see example below). The **collection** class has an arbitrary number (>= 1) of associations to enumerate the elements, no stereotype is necessary.

The word *collection should* imply that the order of the elements is irrelevant but the DAML+OIL specification states that these collections will be translated to lists (where the order relevant). This would indicate that we would loose some information in the DAML/UML notation (in DAML/XML the element already have an implicit order), this could be remedied by using the UML **constraint** mechanism on each of the element associations. The constraint would be the number (1, 2, 4, etc.) indicating the elements position. However this explication of the order for the elements is only necessary if the evaluator of the language uses it in some way, the DAML+OIL language doesn't appear to need ordered elements.

The **disjointUnionOf** property can take class expressions as values, this DAML/UML example shows the simplest case of class expressions: a class reference. Class expressions can be complex and the UML notation for them is discussed in 4.2.4

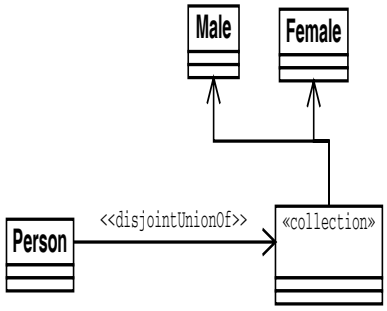

Figure 18: *The DAML+OIL enumeration.*

In this construct it would be useful to introduce a stereotype icon for the «**collection**» class.

Another problem is that the **enumeration** property contains a set of *individuals*, In the overview I claimed that I would ignore individuals in the ontology but we let's solve this problem anyway. So how should we denote individuals? As pointed out in a previous section the UML concept of *objects* is the same as DAML+OIL's individuals, and the UML standard allows you to draw objects

in class-diagrams. The DAML/UML visualization then becomes a variant on the previous example, see example 19.

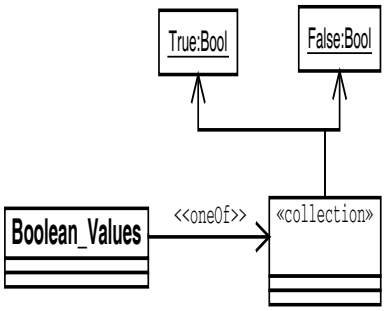

Figure 19: *The DAML+OIL enumeration. The class* Person *is the disjoint union of all the individuals of the* Male *and* Female *classes.*

The last way of defining classes is through **Class Expression Combinations**, a class definition can contain any number of such combinations. See next section for these properties.

4.2.4 Class Expressions

A class expression either refers to a named class (**Class Reference**) or defines an anonymous class. The anonymous class is equivalent to either an **enumeration** that contains all individuals of the class, or to all individuals that satisfy the **Property Restriction**, or to the **Class Expression Combination**. As

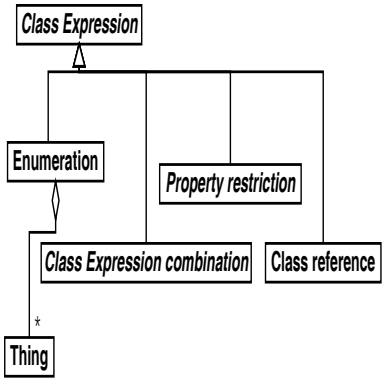

Figure 20: *Abstract syntax for class expressions*

we have seen in the previous sections, class expressions are used as the value of some properties and can't stand alone. We have already seen how the **enumeration** construct is mapped to DAML/UML, the **Property Restriction** is described in section 4.2.7, and **Class Expression Combination** in the next section.

4.2.5 Combinations of Class Expressions

There are three defined properties for combining class expressions (this is called *boolean combination* in the DAML+OIL specification). The result of these properties are anonymous classes, and can consequently occur in positions where classes are expected (see the syntax diagrams). The **complementOf** defines

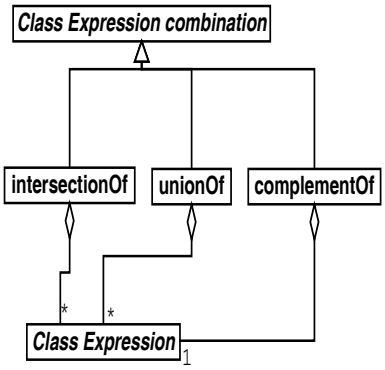

Figure 21: *Abstract syntax for combinations of class expressions*

the class that contains all objects that doesn't belong to the referenced class expression (this is analogous to logical negation). A DAML/XML example is:

```
<daml:Class rdf:ID="Color">
   <rdfs:subClassOf>
    <daml:class>
      <daml:complementOf>
        <daml:unionOf rdf:parseType="daml:collection">
          <daml:Thing rdf:resource="White"/>
          <daml:Thing rdf:resource="Black"/>
        </daml:unionOf>
      </daml:complementOf>
    </daml:class>
   </rdfs:subClassOf>
</daml:Class>
```

This example is a bit more complex than the ones we have seen before, it asserts that the class *Color* contains a subset of all individuals that do *not* have type *White* or *Black*. Here we have to use an explicit **Class** element, stereotyped with **Anonymous Class**, since we can't subclass a property (**complementOf**).

In DAML/UML this would be as shown in figure 22.

The two properties **intersectionOf** and **unionOf** are analogous to logical conjunction and logical disjunction respectively. are similar to the **disjointUnionOf** property described in the previous section. Both properties have a list of class expressions as the range. The visualization is equivalent to the **disjointUnionOf**.

Since these properties takes class expressions as values (and a **class expression combination** is a class expression, see previous section) arbitrarily complex expressions can be created.

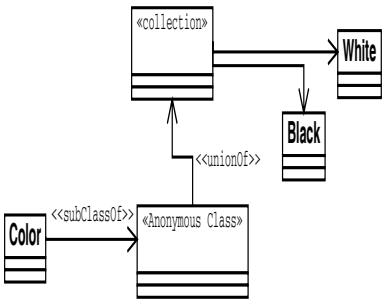

Figure 22: *The DAML+OIL enumeration. The class* Person *is the disjoint union of all the individuals of the* Male *and* Female *classes.*

4.2.6 Property Definitions

Sofar the mapping between DAML+OIL and UML has been more or less obvious, but now we come to the property definitions and as noticed before these deviate from the UML way of defining properties of classes.

We can start by dividing the DAML+OIL properties into two disjoint classes, the **DatatypeProperty** class and and the **ObjectProperty** (which I have referred to as *complex* properties in previous sections), the other types of properties shown in figure 23 are more specialized than these.

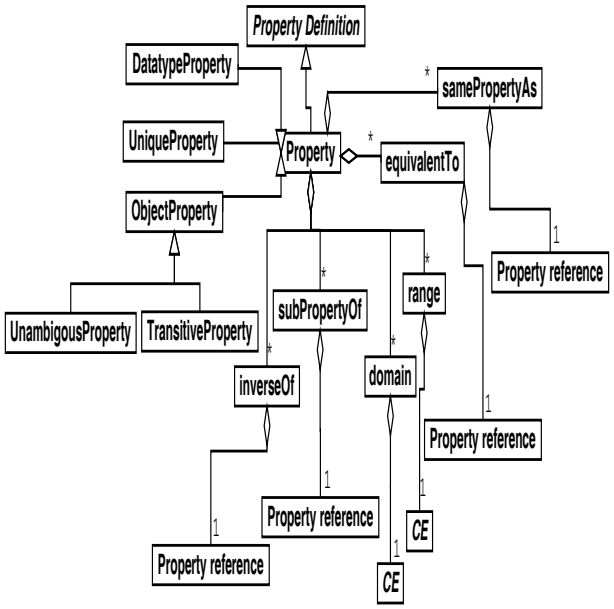

Figure 23: *Abstract syntax for property definitions*

The **DatatypeProperty** relates objects to datatype values, so this is similar to the UML concept of attributes. The DAML+OIL specification allows the use of the predefined XML Schema datatypes, such as integer, decimal, string, etc.

It is also possible to use XML Schema definitions to create user defined types. To allow this in the DAML/UML mapping we would have to map the XML Schema constructs to UML which I will not attempt, see [4] for an example of this kind of mapping. A **DatatypeProperty** is defined as follows.

```
<daml:DatatypeProperty rdf:ID="shoeSize">
   <rdfs:domain rdf:resource="Person"/>
   <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#decimal"\>
</daml:DatatypeProperty>
```

This means that the class *Person* has the property *shoeSize* that can take decimal values (since the range is decimal).

The DAML/UML mapping might seem obvious, if we assume that all allowed datatypes are defined in the XML Schema specification the above DAML/XML example could be visualized as:

Figure 24: *A simple approach to the datatype property definition.*

This defines the property *shoeSize*, the domain is the containing class and the range is given after the colon. This is what attribute definitions looks like in UML and it would nice to reuse this notation. However this approach has a few problems.

1. The DAML+OIL properties are first-class objects, so we must be able to use e.g. **subPropertyOf** with the *shoeSize* property, this is not possible in the example above.

2. DAML+OIL allows you to declare a property without defining the domain or range. In UML (as in the DAML/UML example in figure 24) a property can't exist without a domain and range.

3. The namespace of the domain as well as of the range. The above DAML/UML doesn't say anything about this. If we allow other datatypes than the ones in the XML schema specification, we can define our one, we need to reference the correct resource.

To solve these issues I have choosen to introduce a new stereotype, **DatatypeProperty**, which can be applied to a UML class. This follows the general notation rule defined in a previous chapter, the type of the resource is indeed **DatatypeProperty**. When we define a new datatype property we create a UML class stereotyped with **Property**, the name of this class becomes the name of the defined property. If we don't want to define neither the range or domain this is all we need to do. This allows us to use the properties defined for properties, **subClassOf, inverseOf** etc. see figure 23.

But now we have a problem. Apart from the option of not specifying the domain and range at all or specifying both domain and range (in which case

the DAML/UML above works), DAML+OIL allows you to specify the domain without any range and the range without any domain. The case domain without range is simple to achieve, just take the diagram in figure 24 and remove the ":decimal" part.

The last case with a range but no domain is trickier, UML doesn't seem to support this in any way. But we can make use of the fact that all properties have the type **Thing**. Create a new UML class named **Thing** (and set the tagged value *NS/URI*), and add the property name together with the range. In this way nothing has really been said about the domain. Figure 25 shows the correct way of defining properties.

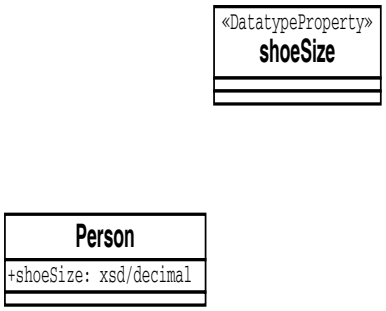

Figure 25: *The DAML/UML for the* marriedTo *property.*

Note that we have to make same distinction between a property definition and a property extension as we did with class definitions, this problem is handled the same way they were for classes. All classes stereotyped with **DatatypeProperty** has a tagged value **extends**.

The **ObjectProperty** relates objects to other objects, this is not completely compatible with the UML concept of attributes since the domain and range can be class expressions. As we have seen the **DatatypeProperty** can be, with some trouble, be represented with existing UML notations, but for the **ObjectProperty** I had to leave the UML attributes behind.

We can reuse some of the notation from the **DatatypeProperty** solution, namely the idea of representing a property as a UML class with a stereotype. Now we of course use a **ObjectProperty** stereotype on the class. Again this stereotyped class is used for the **subPropertyOf** etc. relations. An example of a **ObjectProperty** in DAML/XML is:

```
<daml:ObjectProperty rdf:ID=''marriedTo''>
   <rdfs:domain rdf:resource=''Person''/>
   <rdfs:range rdf:resource=''Person''/>
</daml:ObjectProperty>
```

For this simple example it would be possible to completely reuse the **DatatypeProperty** notation, since both the domain and range are class references, but more complex class expressions can't be visualized that way. For example:

```
<daml:ObjectProperty rdf:ID=''marriedTo''>
```

```
    <rdfs:domain>
      <daml:unionOf rdf:parseType=''daml:collection''>
         <daml:Thing rdf:resource=''Male''/>
         <daml:Thing rdf:resource=''Female''/>
      </daml:unionOf>
    </rdfs:domain>
    <rdfs:range rdf:resource=''Person''/>
  </daml:ObjectProperty>
```

To visualize this we have (atleast?) three choices:

1. Use the defined UML notation for class expressions, but instead of the resulting anonymous class, give the class a name. Then we can use the defined notation for the **DatatypeProperty** properties.

2. Use the UML concept of *association classes*. Association classes allows UML associations to have both class and association properties.

3. Use a directed association (with a suitable stereotype) drawn between the domain class and the range class.

I dislike the first option becuase it means we have to create names for classes that really wasn't intended to have names. This is frustrating for the user who will start using non-descriptive names, thus making the ontology more difficult to understand.

The second alternative looks compelling, association classes allow you to draw associations between associations so this would be a natural way of representing these complex object properties. Unfortunately the UML editor that is being used doesn't support association classes, so this option is not viable.

So let's choose the third alternative, using a standalone class stereotyped with **ObjectProperty** and a stereotyped association between the domain and range classes. I have choosen the association stereotype **domain/range**, the UML language contains a construct called *aggregate* which could be used as a sterotype icon. We also have to now the name (and of course namespace) of the property who's domain/range is being defined, this is given as the associations name. The above **ObjectProperty** would then be represented as shown in figure 26

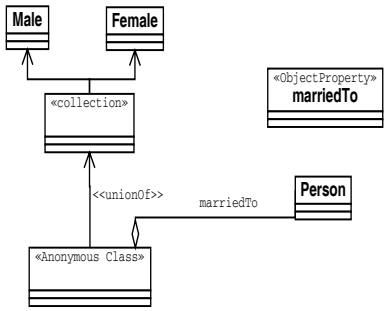

Figure 26: *The DAML+OIL/UML for the* marriedTo *property, showing the use of the aggregate association for declaring domain and range.*

Here we can note a discrepancy between the DAML/XML and DAML/UML, in the UML notation we need a **Anonymous Class** definition since we can't draw the **domain/range** with origin in the **collection** class. This is ugly but it might be alleviated with stereotype icons.

I suggest using two complementary notations for **ObjectProperty** definitions. If the domain and range are class references we can use the same notation as for **DatatypeProperties** (UML attributes), otherwise the more clumsy notation with **domain/range** associations has to be used. Fortunately the second case might not be very frequently used, it might be more ontology might easier understood if the domain and range are named classes (class reference). The association class concept would be useful, but as mentioned the used UML editor doesn't support this.

The use of UML attributes makes it possible for the ontology developer to immediately see the properties associated with the class, but we still have properties as first class citizens. I think this is an advantage over the DAML/XML notation, there it is difficult to get an overview of which properties a class has. The negative side-effect in the DAML/UML notation is that the domain/range is visually decoupled from the rest of the resource definition, by looking at a class stereotyped as a property it is not possible to tell the domain/range for it. But I think that ontology developers will spend most of the time defining classes and relations between them rather than defining relations between properties. This assumption is certainly true for the collection of example ontologies found on the DAML+OIL website[21].

We can notice that the use of association classes would actually give us the opportunity to see the properties of a class, as well as seeing the domain/range for a property. However the UML standard doesn't allow an association class to occur more than once in the diagram which can be problematic when we want to specify multiple domains and/or ranges for a property. So even if the used UML editor had supported association classes it would have been a better idea to use the current approach with normal associations.

Sofar I have only talked about object and datatype properties, but DAML+OIL also contains other types of properties. As shown in figure 23 **UnambigousProperty** and **TransitiveProperty** are subclasses of the object property so the same visualization rules goes for these properties, just exchange the **ObjectProperty** stereotype for the new property type. A unambigous property is a property whose object (value) uniquely identifies its subject and a transitive property obviously states that the property is transitive.

The **UniqueProperty** is a subtype of **Property** and can therefore be applied to any kind of property. For a unique property the subject uniquely determines the object. Usually you wouldn't define a property to be just a unique property, you define it to be a object or datatype property and then add the **UniqueProperty** assertion (it is a global cardinality constraint). For example:

```
<daml:UnambigousProperty rdf:ID=''marriedTo''>
```

[21] This is an interesting fact that may indicate several things. First, people are not used to this status of the properties since they are used to OO-languages, it will take a while before realize how to use this concept. Second many of the sample ontologies have been translated from other KR languages that may not have properties as first-class citizens. Third, the concept of first-class properties isn't very useful.

```
    <rdf:type rdf:resource=''daml:UniqueProperty''/>
    <daml:domain rdf:resource=''Person''/>
    <daml:range rdf:resource=''Person''/>
</daml:UnambigousProperty>
```

This states that a person can be married to exactly one person. Here I have used the *rdf:type* construct to make sure that the *marriedTo* property is also a **UniqueProperty**. The DAML/UML notation for the unique property uses the UML concept of cardinality constraints. If a property definition contains a UML constraint with the value '1' it will become a unique property. Notice that only the value '1' would be allowed, DAML+OIL doesn't allow arbitrary global cardinality constraints. See figure 27 for this definitition of *marriedTo*.

Figure 27: *The definition of the marriedTo property as a unique and unambigous property*

Again we have the problem that the uniqueness of the property is not apparent just by looking at the DAML/UML property classes, one has to look at the class definitions.

The **Property** class is the top class for all the property types and usually you don't define a property to be of this type, you use one of the subtypes. But sometimes when you are refering to an already defined property you can use the **property** sterotype since it is the superclass of all property types. This is convenient when refering to properties in other ontologies, you don't have to look up the exact type.

A property definition can make use of a few different properties between properties, **subPropertyOf, equivalentTo, inverseOf** etc. These are visualized as usual, an stereotyped association between the properties (i.e. the UML classes sterotyped with **Property** or a subtype of property).

For the **subPropertyOf** we can reuse the UML inheritance arrow. See figure 28.

4.2.7 Property Restrictions

A property restriction is a special kind of class expression. It defines an anonymous class consisting of the objects satisfying the restriction. There are two kinds of restrictions, **ObjectRestriction**s that is applied to object properties and **DatatypeRestriction**s that works on datatype properties. The syntax for these two types of restrictions is the same.

An example class definition that uses some restrictions is:

```
<daml:Class rdf:ID=''Person''>
   <rdfs:subClassOf>
     <daml:ObjectRestriction>
```

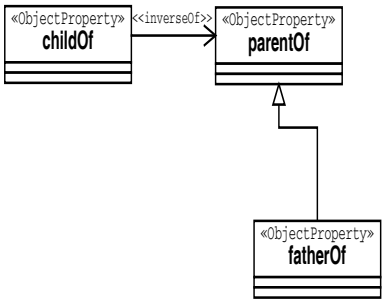

Figure 28: *The sonOf is the inverse property of parentOf and fatherOf is a subproperty to parentOf.*

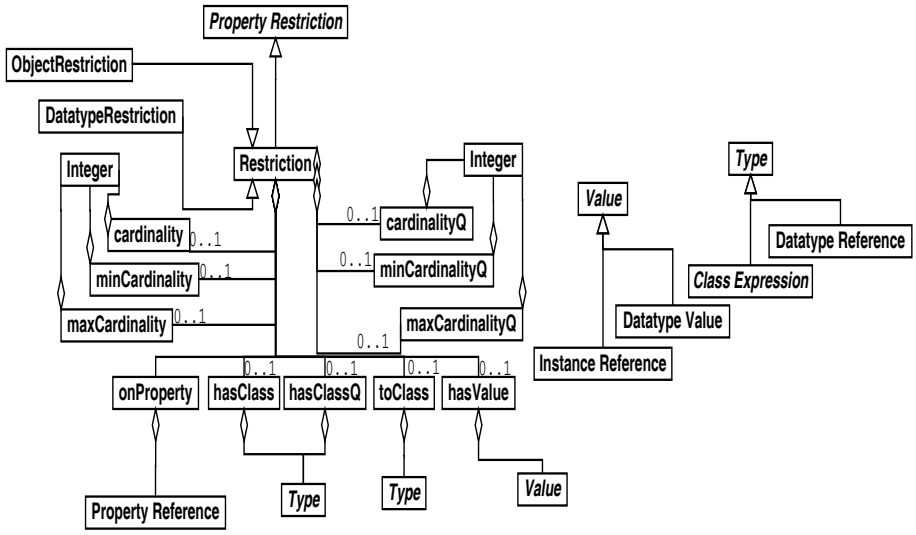

Figure 29: *Abstract syntax for property restrictions*

```
        <daml:onProperty rdf:resource=''marriedTo''/>
        <daml:maxCardinality>1</daml:maxCardinality>
      </daml:ObjectRestriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <daml:DatatypeRestriction>
        <daml:onProperty rdf:resource=''shoeSize''/>
        <daml:cardinality>1</daml:cardinality>
      </daml:DatatypeRestriction>
    <rdfs:subClassOf>
</daml:Class>
```

This asserts that all *Person*s can only be *marriedTo* exactly one Person and that Person must have exactly one shoesize. In example ontologies found on the web the **Restriction** keyword is usually used instead of the more specialized

DatatypeRestriction/ObjectRestriction, this is bad practice so I will use the more specialized variants.

The DAML+0IL language lacks global cardinality constraints on properties (apart from the unique/unambigous property types), the philosophy is that this should be done through property restrictions.

First of all a property restriction always creates an anonymous class which can be used as an ordinary class (i.e it can be subclassed etc.). To represent this in DAML/UML we can reuse the class stereotype, which was introduced in a previous section, *«Anonymous Class»* which gives us an anonymous class.

Secondly a property restriction is created by putting a restriction on a property. This means that we always need a reference to the restricted property, this is given by the **onProperty** property.

So, a DAML+OIL restriction always has exactly one **onProperty** property. This property is simple to visualize in DAML/UML, just use the general notation rule, draw the stereotyped relation between the anonymous class and the property reference. But since we are not actually mapping the DAML+OIL element **Restriction** to a corresponding element in DAML+OIL/UML we have to change the name of **onProperty** to **restrictOnProperty** to make it more intuitive.

But we also have to say in which way this property should be restricted. There are basically three ways of restricting a property

- Cardinality restrictions.
- Domain and range restrictions.
- Combinations of the two restriction types above.

I will step through these restriction types one-by-one.

Cardinality restrictions on properties defines the anonymous class to be the set of objects that has the cardinality for the restricted property. For example, the **ObjectRestriction** on the property *marriedTo* above defines the anonymous class to be the set of objects that has *atmost* one value for the *marriedTo* property.

The **cardinality** restriction defines the class of all objects that have exactly N distinct values for the restricted property. The **maxCardinality/minCardinality** defines the class of objects that have atmost/atleast N distinct values for the restricted property.

The UML notation allows multiplicity constraints to be asserted for associations. This notation should be reused for the DAML+OIL cardinality constraints. We can associate the cardinality with the **restrictOnProperty** association. A **cardinality** constraint is given by **maxCardinality minCardinality**

See figure 30 for example of constraints.

In the DAML+OIL/XML example we explicitly wrote that the restriction was on a object property (**ObjectRestriction**) or a datatype property (**DatatypeRestriction**), but in the UML notation we don't see any distinction.

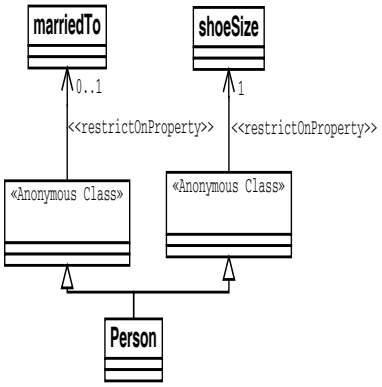

Figure 30: *DAML+OIL/UML representation for the DAML+OIL/XML example above.*

We don't need this explicit information in UML since the translator from UML to XML can determine the type of the constrained property and hence generate the correct restriction. If, for some reason, the translator can't determine the type of the property it works to generate a **Restriction** element.

Note that DAML+OIL doesn't allow *global* cardinality constraints, the constraints given in the restriction above are *local* and doesn't affect the definition of the restrained properties. The only way to assert global constraints is to use the **UniqueProperty/UnambigousProperty** types when defining a property.

Type restrictions can be used to create anonymous classes by restricting the range of the constrained property to be of a certain type. DAML+OIL defines three operators for this kind of restriction, **toClass, hasClass** and **hasValue**.

The **toClass** property defines the class of all objects for which the values of the constrained property belong to the class expression, e.g. it is a universal restriction on the range of the property. For example:

```
<daml:Class rdf:ID=''Person''>
   <rdfs:subClassOf>
      <daml:ObjectRestriction>
         <daml:onProperty rdf:resource=''hasFather''/>
         <daml:toClass rdf:resource=''Person''/>
      </daml:ObjectRestriction>
   </rdfs:subClassOf>
</daml:Class>
```

Let us assume that the *hasFather* property has domain/range *Animal*. The restriction then means that a *Person* must always have an object of type *Person* as the value of the *hasFather* property. The **hasClass** property creates the class of objects for which there exists atleast one value, for the constrained property, belonging to the class expression/datatype. In other words, an existential restriction.

These restriction are visualized with an association stereotyped with **toClass** or **hasClass**. The DAML+OIL **toClass** example above is visualized in figure 31.

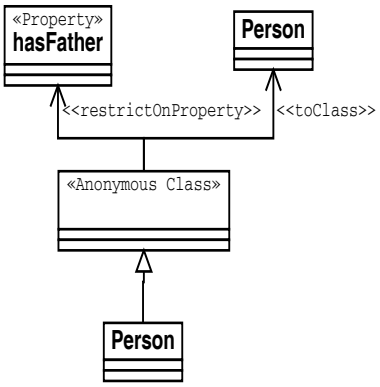

Figure 31: *DAML+OIL/UML representation for the DAML+OIL/XML* ***toClass*** *example above.*

The **hasValue** property is used to create the class of objects whose value is the supplied individual value. For example:

```
<daml:Class rdf:ID=''PersonsWithAge24''>
   <daml:subclassOf>
      <daml:DatatypeRestriction>
        <daml:onProperty rdf:resource=''hasAge''/>
        <daml:hasValue>24</daml:hasValue>
      </daml:DatatypeRestriction>
   </daml:subClassOf>
</daml:Class>
```

This restriction defines the class of all objects that has the value "24" for the property *hasAge*. An **ObjectRestriction** would be written analogously. The DAML+OIL/UML simply uses a new association stereotyped with *hasValue*. See figure 32. The range of **hasValue** is always an individual, the UML ob-

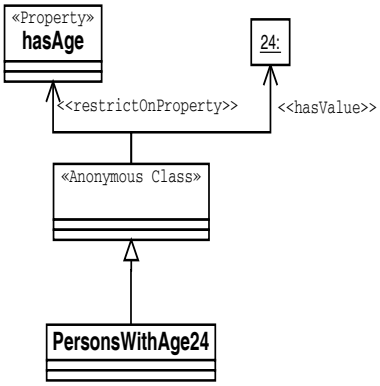

Figure 32: *DAML+OIL/UML representation for the* ***hasValue*** *example above defining the class PersonsWithAge24.*

ject notation is used to denote DAML+OIL individuals. The visualization of restrictions on object properties is the same.

Combined restrictions are also called *qualified constraints*. These restrictions impose both cardinality constraints as well as type constraints. Combined restrictions are created with the **hasClassQ** operator and one of the **cardinalityQ**, **minCardinalityQ** and **maxCardinalityQ** properties, the "Q" stands for *qualified*. In the cardinality constraint shown previously a person could be married to atmost one **Thing**, e.g. anything at all. Now we can constrain this even more and assert that a person is married to atleast one person.

```
<daml:Class rdf:ID=''Person''>
   <rdfs:subClassOf>
     <daml:ObjectRestriction>
       <daml:onProperty rdf:resource=''marriedTo''/>
       <daml:maxCardinalityQ>1</daml:maxCardinalityQ>
       <daml:hasClassQ rdf:resource=''#Person''/>
     </daml:ObjectRestriction>
   </rdfs:subClassOf>
</daml:Class>
```

This is simply visualized by modifiying the notation in figure 30, we'll add an association for the **hasClassQ** property. This shown in figure 33.

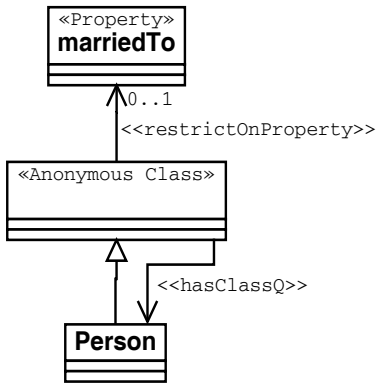

Figure 33: *DAML+OIL/UML representation for the* ***hasClassQ*** *example above defining Person.*

4.2.8 Notes on the Ubiqitous Namespace in DAML+OIL/UML

In the description of the mapping I have defined a number of stereotypes as well as tagged values for these stereotypes, some tagged values are general to *all* the stereotypes e.g. URI, Namespace. This reflects the fact that everything in DAML+OIL (properties, classes, values) are instances of *resources*. We could take advantage of this in our DAML/UML profile by introducing a new class in the UML metamodel, the **Resource** class that has these properties. All the UML elements used in the mapping could then be instances of this metaclass, thus giving us a nice conceptual solution to the tagged value problem.

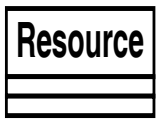

Figure 34: *The **Resource** meta-class.*

Of course the ontology developer must always supply the namespace for the resource, this could be alleviated by tool support. When a property/class name is used the tool could automatically try to find the correct namespace by looking at which ontologies that are imported and which properties/classes they define. If a resource with the same name is defined in several namespaces the user could get a list of namespaces to chose from.

4.2.9 How to Translate the UML Notation to DAML+OIL

A XSLT stylesheet (XMI2DAML+OIL) has been developed to translate the UML classes (in XMI format) to DAML+OIL classes, the **subclassOf** property (displayed in the UML diagram with the UML subclass arrow) is also translated. All the other constructs are ignored. A equivalent stylesheet has been developed to translate the XMI to F-Logic schemas (XMI2FloraSchema). It should be straightforward to implement the rest of the mapping, and the use of XSLT is very convenient.

I also experimented with the UML editor *Ration Rose*'s scripting language and wrote a script that outputs classes and inheritance from the Rose editor in F-Logic schemas. But using this approach would bind the ontology editor to a particular UML editor.

In the description of the mapping I have used sterotyped UML associations for most DAML+OIL properties, the stereotype name is the name of the corresponding DAML+OIL property. This has the advantage of simplifying the translation from UML to DAML+OIL since we just have to select the stereotype name. But the goal was to create a graphical notation that was close to the 'ordinary' UML, therefore it is necessary to reuse things like UML inheritance arrows for **subclassOf/subPropertyOf** and aggregation arrows/attributes. This will make the translation program slightly larger since it has to handle different cases.

The best solution would be if the UML editor supports stereotype icons in a way that the XMI file contains a stereotyped association and the editor displays a stereotype icon (e.g. inheritance arrow for **subclassOf/subPropertyOf**). None of the UML editors I have used supports this.

5 Notes on the Correspondence between DAML+OIL, RDF(S) and F-Logic

Here I shortly discuss the possibilities/problems of translating RDF(S) and DAML+OIL to F-Logic. If we find a translation we could use the Flora F-Logic system as an inference engine for RDFS and/or DAML+OIL.

5.1 Translating RDF descriptions to F-Logic Facts

As I explained in 2.4.2 the RDF metadata descriptions are assertions that a resource has certain values for certain properties. This maps nicely to F-Logic facts.

An RDF statement always has a subject, predicate and object. This is also true for F-Logic molecules, the following generic RDF statement:

```
<Description ID=''subject''>}
   <foo:property1>object1</foo:property1>
   .
   .
   .

   <bar:propertyN resource=''objectN''/>
</Description>
```

would be translated to the F-Logic fact:

$$subject[foo\_property1 -> object1, \ldots, bar\_propertyN -> objectN].$$

This is all we have to do, we could load it into the Flora-2 system and query this database.

The reification properties available in RDF can be reprsented through the higher-order capability of F-Logic.

5.2 Translating RDFS Schemas to F-Logic Schema and Rules

In the RDF Schema language the basic modelling primitives are classes and properties, such languages are also *Frame-systems*. Since F-Logic is an abbreviation for *Frame*-Logic we might expect that these two languages should be similar. An indeed they are, there is a very strong analogy between RDF/F-Logic facts and RDFS/F-Logic Schema.

The basic modelling primitives in F-Logic are classes and properties, and the axioms specified in the RDFS language (subClassOf, subPropertyOf etc.) can all be mapped easily to F-Logic. Table 3 shows how RDFS constructs can be mapped directly to equivalent F-Logic constructs.

RDFS construct	F-Logic construct
Class (C) declaration	C[]
Property (P) declaration	DomainClass[P=>RangeClass]
subClassOf	C1::C2
(rdf:)type	i:C
subPropertyOf (P2 sub of P1)	Subject[P1->obj] :- Subject[P2->obj]

Table 3: *The mapping between RDFS and F-Logic constructs.*

The mapping is slightly more complicated since RDFS properties can be defined on their own, without having any domain or range. The mapping shown in the table for property declarations only works when we have domain/range specified. If either domain or range is missing we will have to have a dummy

object of type *Resource* in which we could add the property. The *Resource* class should be introduced anyway, and all classes should inherit from this. Furthermore we should make the subPropertyOf axiom transitive.

Several engines for handling RDFS inferences exists, e.g. the *Triple* system, described in [26] and DAMLJessKB[22].

5.3 Translating DAML+OIL Ontologies to F-Logic Schema and Rules

Currently no inference engine for the DAML+OIL language exists. The DAML-JessKB system claims to handle it but it can only handle a subset, the lacking features are basically the class constructors (which is basically the difference between RDFS and DAML+OIL).

The TRIPLE system interprets the RDF(S) language by translating the RDFS to a F-Logic like language. TRIPLE handles (or will in the future, it is not clear) DAML+OIL through an interface to a Description Logics system. The translation from RDFS to F-Logic should be straightforward, since both languages are *frame-based*.

The problem with mapping DAML+OIL to F-Logic is the class constructors, these operators have no direct counterpart in F-Logic. It doesn't mean it is impossible to map DAML+OIL to F-Logic but it is tricky.

For example, the *complementOf* operator applied to a class c could be implemented as follows (pseude F-Logic code):

```
resource[]. %% Top class, all classes must inherit from this.
class[]::resource[]. %% The rdfs:class.
c[]:class[].      %% The class we want the complement of.
c_comp[]:class[]. %% Generated anonymous class to store result.
class_operators[create_complement=>Class, ...]. %% wrapper class for operators.

%% If a given object is not of type 'c' then it will get the type 'c_comp'
class_operators[create_complement->Class:c_comp[]] :- tnot Class:c.
```

When a fact base (translated from RDF statements) is loaded together with the above definition the Flora-2 engine will automatically exectue the rule `class_operators[create_complement...` which will assign the type `c_comp` to those objects not of type `c`. It is most likely that this approach can be used to implement all the DAML+OIL class constructors.

A description logic engine would be able to reason with the ontology itself without any instances, such reasoning is important in the design of large ontologies. It would allow us to get error messages if the ontology is inconsistent e.g. if A is declared as equivalent to B and B is declared to be disjoint with A, a DL engine would be able to detect this. To achieve this in F-Logic we would have to complement the reprsentation sketched above with a additional representation of how classes are defined and then evaluate rules on that. Furthermore a DL system would be able to show answers to queries *intensionally* i.e. the definition in terms of class constructors and axioms instead of *extensionally* which is the set of objects, this might also be possible to do in F-Logic if we use the complementary representation.

[22] http://plan.mcs.drexel.edu/projects/legorobots/design/software/DAMLJessKB/

When using a system for general Horn clause reasoning would probably be very inefficient compared to a description logics engine which uses specially tailored algorithms.

DAML+OIL is a description logic language[23], in that research area most work seems to be about investigating the tradeoff between expressivity of the language and algorithmic complexity. It is surprisingly difficult to find any paper discussing the expresiveness of DL languages compared to Horn logic or predicate logic.

The advantage with the F-Logic approach is the simple combination of Horn rules together with the DAML+OIL. Since F-Logic handles Horn logic we can represent both DAML+OIL constructs and horn-rules in one unified format (F-Logic formulas), it is probably more difficult to integrate Horn logic reasoning with a description logic engine ([17] describes how this is done for a DL, but no recursive Horn rules).

6 Visual Editing of Rules and Invariants

This section describes why whe need a more expressive language on top of DAML+OIL, in this thesis I have assumed that this language will be Horn clauses. Different approaches to the visual specification of Horn clauses (rules and invariants) is investigated.

6.1 Why DAML+OIL is not Enough, the need for Rules

The DAML+OIL language is a representation language that has been designed to model complex concept (class) hierarchies, it supports complex class constructors that can be used to create complex concepts from primitive ones and restrictions. But the DAML+OIL has no support for *property* constructors so it is not simple to define complex/derived properties (which I will refer to as *rules*), this can basically just be done with the **subPropertyOf** axiom.

This is very restrictive, e.g. a property can't be declared as a subproperty of itself or any of its subproperties. This disables us from declaring properties that are recursively defined. Another problem is the composition of properties. Assume that we want to define a derived property **happilyMarriedTo**, this could be expressed like follows:

$$happilyMarriedTo(X, Y) \leftarrow hasSpouse(X, Y) \wedge bestFriend(X, Y)$$

This states that X is happily married to Y if Y is a spouse of X and also is the best friend of X. This means we have a intersection of the properties on the right hand side of the arrow.

If we try to define a equivalent DAML+OIL property we have to use the **subPropertyOf**, consider the following definition:

```
<daml:Property rdf:ID=''happilyMarriedTo''>
   <rdfs:subPropertyOf rdf:resource=''hasSpouse''/>
   <rdfs:subPropertyOf rdf:resource=''bestFriend''/>
</daml:Property>
```

[23] more precisely: it is a member of the $\mathcal{SHIQ}$ family of description logics

This does not have the same semantics as the prolog definition above. The DAML+OIL definition implies that **happilyMarriedTo** would become:

$$happilyMarriedTo(X,Y) \leftarrow hasSpouse(X,Y) \vee bestFriend(X,Y)$$

Which was not intended. In fact there is one way[24], we of defining this derived property indirectly by using the fact that the domain and range of properties can be *class expressions*. To achieve this we begin by defining a property **spouseOrBestFriend**, this is actually what the DAML+OIL definition of **happilyMarriedTo** defines so we just change the name on it. Then we define a new class called **HappilyMarriedPerson** by intersecting the textbfhasSpouse, **bestFriend** and **spouseOrBestFriend** properties, this definition imposes cardinality restrictions on the three properties. By restricting the exact cardinality of these properties to be *one* we get the class of persons who are married to their best friend. Then we can define the domain of the **happilyMarriedTo** property to be the class **HappilyMarriedPerson**, finally we say that **happilyMarriedTo** is a subproperty of **hasSpouse** and now we have the intended definition of the derived property **happilyMarried**. So we avoided the need for intersecting by introducing a new class that used property restrictions.

Clearly this is very complicated and Horn rules seem to offer us a more convenient way of defining derived properties.

Although we managed to express the above rule in DAML+OIL there are other rules that simply cannot be stated because of the lack of property constructors. For example:

$$hasSibling(X,Y) \leftarrow hasFather(X,Z), hasFather(Y,Z)$$

can't be expressed. We need a more expressive language for that[25].

The DAML+OIL language has good support for class definitions but poor support for property definitions. For Horn clauses the opposite is true, it has good support for defining complex properties but usually no support for class hierachies. If we could combine the expressiveness of these two languages we would get a language that supports typed horn-logic (order-sorted logic). Currently work is ongoing in the semantic web community to define such a language (called DAML-L) and it seems like Horn clauses will be used, but nothing has been published about it yet.

So basically the DAML+OIL has the following limitations in expresiveness

- No composition of properties to create rules/complex properties.
- No recursive properties (this is really subsumed by the above item).

The binary nature of properties in DAML+OIL isn't really restricting the expressiveness since we can rewrite an n-ary property to a binary if we have rules.

[24] Perhaps there are several other ways of doing it, but I doubt that.

[25] In the article [5] it is proved that a particular DL is equivalent in expresiveness to the L_3 fragment of FOPC. I think the constructions DAML+OIL are also part of that DL, this would mean that DAML+OIL has the same limitation

The simplest way to achieve a rule language would be to define a serial syntax (XML format perhaps) and write the rules in that language. However a goal of the thesis was to experiment with graphical ways of defining the rules so they would fit nicely with the UML diagrams.

I have experimented with two graphical notations for binary Horn clauses. The first one defines a notation for using UML class diagrams to write rules and invariants, the second uses the constraint diagram notation.

We will also show how invariants can be stated using these two notations. Here I have used the word *invariant* to denote a *general condition*, i.e. not necessarily a condition that has to be true for all objects of a certain class but rather that the condition always is fulfilled. For example we might want to express that atleast one object of a class has to fulfill the condition.

6.2 Rules and Invariants in UML Class-Diagrams

Since we have mapped the DAML+OIL language to class diagrams it would be useful if we could define our rules and invariants in the class diagram as well. This would enable us to edit both DAML+OIL and the rules in the same tool.

We will use the UML's extension mechanisms stereotypes and constraints to achieve this.

6.2.1 Invariant Specifications

Invariants are just a set of properties that all must evaluate to true. Figure 35 shows the general form for invariants in our UML approach.

The properties $p_1 \ldots P_n$ are DAML+OIL properties defined as described in the UML mapping. The UML class stereotyped with **Invariant** can have a name but it is not necessary. The associations between the **Invariant** class and the properties have a number of UML constraints. The $domainvar_1$ and $rangevar_1$ in the schema denote the name of the variables used for the domain and range of property P_1, the default value is X_1 and Y_1 (generally X_n and Y_n for property P_n). The point with naming the domain/range variables of each property is that we can use a variable name of property P_k as the domain and/or range of any other property P_l.

The constraint placed in the middle of the asscociation contains a number and possibly the keyword **not**. The number is used to determine in which order the properties should be evaluated. The **not** keyword is used to negate the property. No disjunction is possible between the properties.

The **Invariant** class also have some other associations (zero or more), targeted at classes. These associations introduce new variables that are quantified according to the stereotype of the association, **forall** means that the variable is universally quantified and **exists** denotes existential quantification. The name of the new variable is given in the $quantificationvar_n$ constraint. Variables that are not explicitly quantified are considered to be existentially quantified.

The type of the variable is determined by the class at the end of the association. The quantifed variable can then be used in the domain/range for the properties in the constraint.

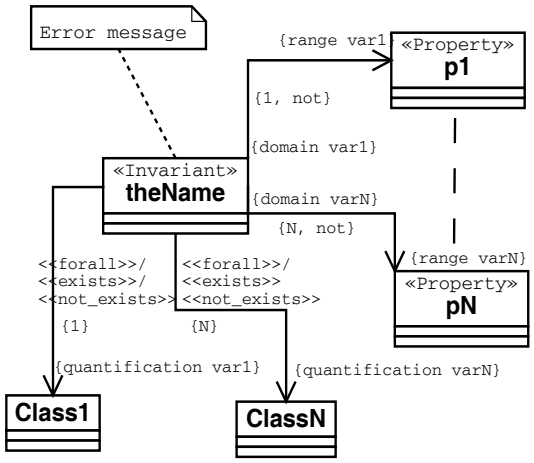

Figure 35: *The template for invariants in UML.*

The note attached to the **Invariant** class defines the error message that should be printed if the constraint evaluates to false.

To make the specification less cluttered we assume that if no explicit variable is given as the domain for a property P_n the range variable of P_{n-1} will be bound to the default domain variable X_n. This makes it very convenient to write so called *chain* expressions, i.e. expressions on the form $P_1(X_1, Y_1), P_2(Y_1, Y_2), \ldots, P_n(Y_{n-1}, Y_n)$ An example constraint is shown in figure 36.

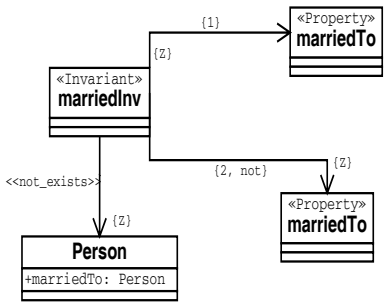

Figure 36: *An example UML invariant. The marriedTo property must be reflexiv.*

The intended meaning is: $\neg\exists\ Z\ marriedTo(Z, Y_1)$, $\neg marriedTo(Y_1, Z)$.

6.2.2 Rule Specifications

The rule specification is similar to the invariant specification. We use a stereotype **Rule** to assert that we're defining a rule, see figure 37. Notice that **Rule** is declared as a subclass of the DAML+OIL **Property** class, this means that it can be used anywhere a **Property** can be used. For example in the definition of

rules and invariants. The **Invariant** was not declared to be a **Property** since it shouldn't be used in definitions.

Apart from this the difference is that we have removed the introduction of quantified variables. The example rule in figure 38 defines the ancestor property.

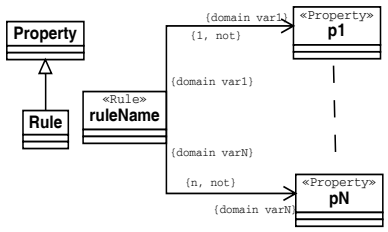

Figure 37: *The template for rules in UML.*

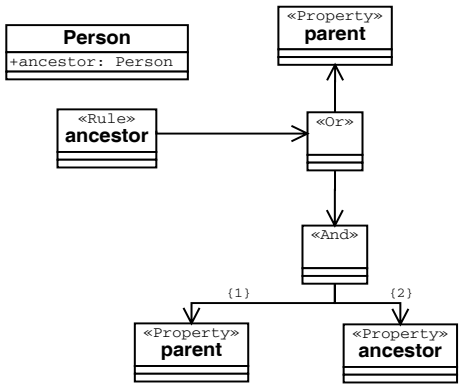

Figure 38: *The recursive* ***ancestor*** *rule in UML.*

This example shows the addition of the stereotyped classes **Or** and **And**. These meaning of these is the usual. The translation to prolog code would be:
$ancestor(X, Y) :- parent(X, Y).$
$ancestor(X, Y) :- parent(X, Y_1), ancestor(Y_1, Y)$

6.2.3 Discussion on UML Invariants/Rules

The use of UML gives us a very nice integration with the UML notation for DAML+OIL, we can reuse the mechanims for namespace handling etc. However I feel that the notation is a bit clumpsy and to similar to the corresponding F-Logic/prolog syntax. This is of course an advantage when translating the visual rules to F-Logic, but the key-point is that they should be intuitive to the user.

Translating the UML rules to F-Logic is very simple, they are just translated to F-Logic rules. The structure of the proposed UML notation is basically equivalent to the text-based syntax.

The invariants is a bit more trickier. Intuitively an invariant should be translated to a F-Logic query, i.e. we query the database and if the result is **yes** the invariant is fulfilled, otherwise an error message is issued. I have used

three different quantifiers in the invariant specification **exists, forall** and **not exists**. An ordinary query is equivalent to the **exists** invariant. The other types of quantifiers can however easily be represented by adding wrapper predicates.

No translation from UML to F-Logic rules has been implemented, instead I focused on the constraint diagrams.

6.3 Rules and Invariants with Constraint Diagrams

Since the UML notation for rules and invariants is a bit verbose it could be interesting to experiment with the constraint diagram notation that seems to give small and intuitive specifications.

6.3.1 Invariant Specifications

The specification of invariants has been briefly described in section 2.6 and I will only describe an extension that I have made to the basic notation.

In the UML notation for invariants I used the **not exists** even though this can be expressed with the $\forall$ and $\exists$ quantifiers I found it useful becuase it simplifies the invariants. So I decided to add it to the constraint diagrams as well. Since I don't have access to the source code for the editor, any notational changes had to use existing constructs. In figure 39 we can see the **not exists** quantifier, it is represented as a hollow circle (this has another meaning in the usual constraint diagrams).

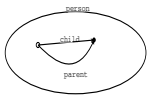

Figure 39: *Example constraint using the* not exists *quantifier.*

6.3.2 Rule Specifications

Constraint diagrams are (as the name indicates) intended to be used for constraints, but we also want to specify *rules*. This is achieved by extending the notation slightly. Another important concept missing in the original constraint diagrams is *negation*.

Rules are created simply by drawing an arrow between two motes. The arrow name (i.e. the property name) must be prepended with two underscores so the translator can see which property is being defined. The rule in figure 40 also uses the second extension, *negation*. Negation is visualized by appending _**not**_ to the name of the arrow.

The translation to prolog/F-Logic would give us: $definedRule(X,Y) :- X : T1, Y : T3, r(X,Z), Z : T2, \neg q(Z,W), W : T3, p(Y,W), \neg W :=: Y.$

6.3.3 Discussion on CD Invariants/Rules

The constraint diagrams seems to be quite simple to understand, and the notation is not resembling the text syntax for logic which to me seems like a

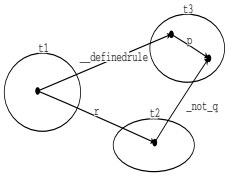

Figure 40: *Example definition of a rule.*

advantage over the UML approach.

The CD approach also has some nice visual features. For example the assertion of equivalence/inequality of variables. In the prolog code for figure 40 the last term is: $\neg W :=: Y$. this may appear strange since we don't seem to assert this anywhere in our diagram. But the use of different motes in the class $T3$ means exactly that. If the two arrows had been pointing to the same mote the inequality assertion would not have been generated and the variables W and Y would be replaced by only one common variable. To state this in the UML approach we would have had to use a equality property.

The use of Venn-diagrams to express intersection of classes is also convenient.

The major disadvantage compared to the UML approach is that we need UML for DAML+OIL and the CDEditor for constraint diagrams. It is of course better if we had an integrated environment.

There are also some problems with the constraint diagram notation. One is shown in figure 41. There are two ways of reading this invariant. There

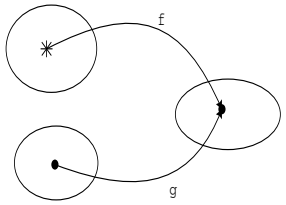

Figure 41: *An ambiguous diagram. It is not clear if it should be read as:* $\forall x \exists y \exists z$ $f(x,z) \wedge g(y,z)$ *or as:* $\exists y \forall x \exists z$ $f(x,z) \wedge g(y,z)$.

is no way of determining from the diagram which quantifier should be placed first. This is resolved demanding that all nodes that doesn't have any incoming arrows must have the same quantification type (universal/existential/negated existential). In UML-rules we could have used constraints giving the order for the quantified variables.

6.3.4 CD2Flora: Implementation of the Translation From E-CD's to F-Logic

Now that we have determined *what* the extended constraint diagrams should be translated to, it is a small matter of programming to actually make it happen! As noted in the previous section there exists a tool for editing (extended) constraint diagrams, the CDEditor. The CDEditor uses a binary saving format for diagrams (perhaps object serialization), but it is also possible to export the diagrams to XML. Clearly the XML format had to be used by the translator, lets call the XML application for constraint-diagrams *CD-XML*.

The figure in 42 shows the different programs involved in translating the diagrams to F-Logic.

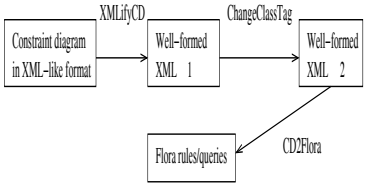

Figure 42: *The dataflow for translating constraint diagrams to Flora. The arrows denote the application of the named program.*

The first step in the figure shows that the program `XMLifyCD` is applied to the CD-XML document, this has to be done because the CDEditor doesn't create well-formed XML documents, more of XML-*like* documents, but the discrepancies are easy to fix. The `XMLifyCD` program is implemented in perl (~ 25 lines of code) and produces well-formed XML documents. Since the source code for the CDEditor isn't available I couldn't fix this, which is obviously a bug in the program.

The second step applies the program `ChangeClassTag` to the XML document, it replaces all occurences of the tag *<class>* for the tag *<myclass>*. This seems to be a pointless activity, but it is necessary to do because of a bug in the **Castor** Schema compiler used to implement the last step in the translation.

The final step in the figure **CD2Flora** does all the work, translating the constraint diagrams to F-Logic.

I considered three alternative ways to implement the translation from the XML format to Flora:

- Since the diagrams are in XML format a obvious implementation would use an *XSLT stylesheet*.
- Use an ordinary XML parser to read the diagrams.
- Use an XML Schema compiler.

Since I didn't have much experience with XSLT and I wasn't sure that it would be convenient to write the translator in this language. The second alternative, to write the translator in a general purpose language and use a XML parser (e.g. Java), seemed to be the safest way to go about. However recently several *Schema compilers* (a.k.a *XML databinding tools*) have emerged, they automate the mapping between XML documents and Java objects. These tools take an XML Schema/DTD as input and generates Java classes. The generated classes takes care of the parsing/unparsing of XML documents conforming to the XML schema. The constraints expressed in the XML schema are checked by these classes, so a non-conforming document will render an error message. When the document has been deserialized you have the XML document as an object structure in-memory.

When comparing the two last approaches it seemed that the Schema compiler would save some work compared to the rather low-level parser alternative (SAX or DOM). *If* the parser approach would have been used I felt that I would need to implement classes for the abstract syntax of constraint diagrams and also write the code for creating the in-memory objects. Since this is exactly what the Schema compilers give us, I decided to use this kind of tool.

The key to using a Schema compiler is to have a schema for the XML application you want to work with. The creator(s) of the CDEditor hasn't defined one (not surprising since it doesn't output *real* XML), but a BNF description (that was almost correct) existed. So the first step was to create a XML Schema for the CD-XML application.

Now that we have overcome the obstacle of getting the diagrams to in-memory objects, we can start the translation. We have to separate between to different kinds of diagrams, *rule* diagrams and *invariant* diagrams, they have to be treated slightly different (you can't mix rules and invariants in the same diagram).

But there is a common step that can be carried out before we start the diagram-type specific translations, e.g. the *type-analysis*.

The type-analysis traverses all the motes in the diagram and assigns a variable to it (which is used by the F-Logic generator) and also determines the type of the mote. This is the most important part of the translation. The current implementation doesn't handle intersection types, just union- and single types of motes.

If the diagram defines a *rule* the rule head (the defined rule) is generated and then the rest of the arrows are traversed to generate the body of the rule. The translator tries to find *starting motes*, i.e. motes that have outgoing arrows but no ingoing ones. These can be seen as starting *chains* (length $>= 1$) of properties. It then translates the chain for each starting mote.

If the diargram defines an invariant basically the same algorithm can be used, the difference is that we have quantifiers that must be considered. The F-Logic code generation is a bit more involved for invariants than for rules, but still quite straigthforward.

In retrospect I am not sure that using the Schema compiler and a Java program was the best solution, the resulting code is quite lengthy and hard to

read. It would probably be both more intelligble as well as *less* code if I had used an XSLT approach.

I think that using *one* monolithic stylesheet to do the translation would have been hard work and difficult to understand. However by piping the XML through a sequence of 2-3 stylesheets (where the first 1-2 stages takes XML and outputs new XML, and only the last outputs F-Logic code) would have yielded an elegant solution.

7 Using the SWEDE to Specify the Static Semantics of XHTML

In this section I show how the SWEDE environmen can be used to specify the static semantics of XHTML. This section was added to provide a complete example of how the SWEDE can be used to define the static semantics of a language, as explained in section 2.1 the original goal of specifying the semantics for Java was not completed.

I have choosen XHTML since most people have some knowledge about HTML and also because the definition of the tags is quite small and simple. Unfortunately it is a bit *too* simple to demonstrate the SWEDE, so I have added some additional tags that gives the opportunity to show some interesting details in the SWEDE architecture.

XHTML is defined by the World Wide Web Consortium (W3C)[26], and it is simply a XML application for ordinary HTML. This means that all the tags that are valid in HTML are also valid in XHTML. The reason for this redefinition of HTML as an XML application is the need for extensibility, the HTML tag set is fixed, and since XHTML documents are well-formed XML documents, tools developed for XML can be used for XHTML.

An example of an XHTML document is

```
<html>
   <head>
      <title>A webpage</title>
   </head>
   <body>
     <p>Moved to <a href=''http://www.foobar.com''>foobar</a></p>
   </body>
</html>
```

7.1 The Correspondence between XML and Abstract Syntax: The Tree Ontology

As hinted in section 3.1 we don't have to use RDF statements to describe instances of ontologies when we are checking integrity constraints for derivation trees. We can automatically derive the interesting relations and properties from the XML representation. In section 2.3 we noted that a derivation tree basically

[26] www.w3c.org

consists of two interesting properties: *containment (child)* and *order*. These are also the basic relations that we need when describing static semantics.

Consider the XHTML example above, it is a derivation tree w.r.t. a DTD defining the legal forms. It starts with the **html** element and it contains two *child* nodes, **head, body**, these in turn have their own children etc. So basically XML uses a tree model to represent data, this is why it fits nicely with the *derivation trees*. A part of the RDF representation for the XHTML document could be:

```
<Description about=''html1''>
  <type resource=''xhtml_ontology:html''/>
  <child resource=''head1''/>
  <child resource=''body1''/>
</Description>
```

The **html1, head1, body1** would be instances of the corresponding tag types (html, head, body). Instead of first translating the XHTML code to RDF statements (which could be done automatically). I will translate derivation trees in XML format direcly to F-Logic facts using the *XML2FloraDB* program. The result is exactly the same.

In order to use the properties and classes we first have to create an ontology for *ordered trees* (i.e. the XML data model). This is used to generate a F-Logic schema[27] for the domain.

Figure 43 shows the types of things found in a tree. Nodes have *parent*s and *child*ren, the *preceededBy* property gives us an ordering on sibling nodes[28]. The *text* property returns any value of a element (e.g. "A webpage" for **title**, shown above).

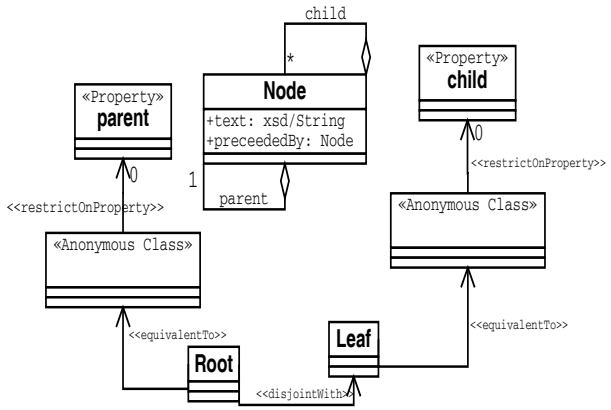

Figure 43: *The different concepts in the domain of ordered trees.*

[27] Currently this schema is written by hand, not generated from the UML.

[28] This gives a partial ordering of the nodes, if a total order is needed we can define one with constraint diagrams

The class *Root* is the set of all nodes that doesn't have any parents, any *Leaf*s don't have any children. Figure 44 declares the properties.

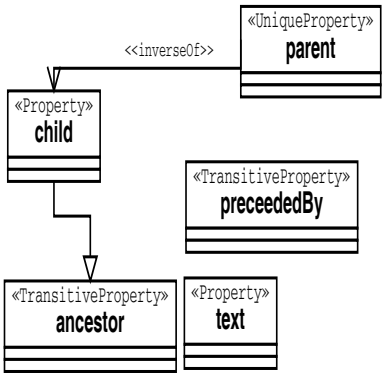

Figure 44: *The different properties that are fundamental for ordered trees.*

parent/*child*(X, Y) should be read as X has the parent/child Y. The **ancestor** property is a rule using the primitive **child** as a basis, but DAML+OIL can only partially define this, we need to complement the declaration in 44 with the recursive rule by the constraint diagram in figure 45. This schema can be

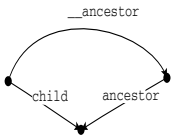

Figure 45: *Partial definition of the* ***ancestor*** *rule, the base case for the recursion is given in the DAML+OIL/UML definition (figure 44).*

used by all applications that needs to refer to the basic structure of XML documents, it can be used for providing fundamental properties from which more complex (and domain dependent properties) could be built. When describing static semantic conditions this is exactly what we want.

7.2 The Static Semantics Invariants of XHTML

The DTD for XHTML doesn't define all the constraints on the syntax, some constraints would make the size of the DTD a lot bigger so they were left out and instead specified informally in the XHTML specification. The static semantics invariants described in this section assumes that the XHTML document has been checked against a XML DTD.

The following elements have constraints on which elements they can contain

- **a** cannot contain other **a** elements.
- **pre** cannot contain the **img, object, big, small, sub** or **up** elements.
- **button** cannot contain the **input, select, textarea, label, button, form, fieldset, iframe** or **isindex** elements.
- **label** cannot contain other **label** elements.
- **form** cannot contain other **form** elements.

These are all instances of the same constraint type, so in the example I have choosen to use only the **a** constraint. The XHTML specification defines a number of tags, in the ontology description I will only include those tags that are needed to make the examples in the next section work. Of course it is no problem adding the rest of the defined tags, but in this demonstration it is not necessary.

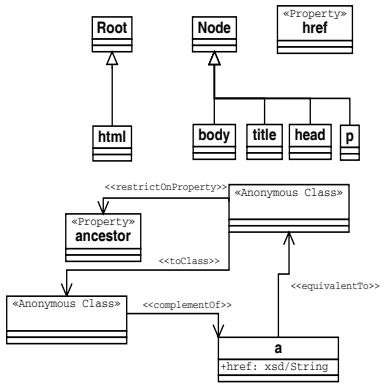

Figure 46: *A subset of the XHTML classes, they inherit from the Node/Root concepts in the tree ontology.*

Notice how the **a** class is defined, it imposes the static semantics constraint on the tag as described above. By using a **toClass** restriction on the property **ancestor**, it is restricted to allow anything except **a** individuals. Consequently individuals with the type **a** may not have an ancestor individual that is of type **a**, if there would exist one the individual would not conform to the DAML+OIL schema and an error message would be issued. This would work but since I haven't implemented a full translation from DAML+OIL to F-Logic we can't evaluate this constraint, instead we'll write it as a constaint diagram which can be evaluated.

The constraint diagram expressing this constraint is shown in figure 47

To demonstrate a bit more interesting constraints that can be stated I have extended the XHTML syntax with some tags. The extension looks like follows:

```
<Pair>
   <Person>
```

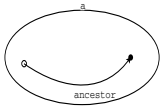

Figure 47: *Invariant condition prohibiting nested* ***a*** *elements.*

```
      <Name>the_name1</Name>
      <Skill>low/high</Skill>
   </Person>
   <Person>
      <Name>the_name2</Name>
      <SkillLevel>low/medium/high</SkillLevel>
   </Person>
</Pair>
```

The *Pair* elements might be used to show which students are going to work together during a project. The classes introduced for the **Pair** extension could be described in its own ontology/namespace, see figure 48. Assume that a XHTML document can contain any number of these **Pair** elements. Several syntax aspects could be characterized through a DTD, such as: a **Pair** element must always have two **Person** elements.

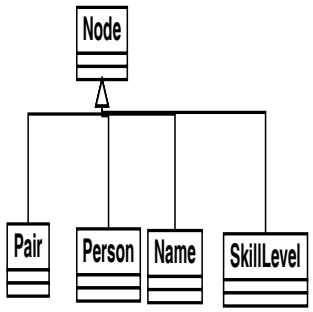

Figure 48: *The classes for the extensions to XHTML. The classes inherit from the Node concepts in the tree ontology.*

Now we can add some constraints that cannot be checked by context-free grammars. Three constraints will described here.

First constraint:

> A person can only occur in one pair. We can assume that the name of a person uniquely identifies him/her.

This invariant is shown in figure 49. It is formulated as "There exists no two different nodes of type **Name** that has the same value for the **text** property".

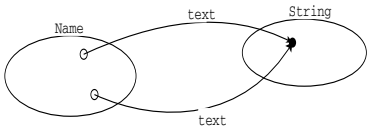

Figure 49: .

Second constraint:

A person may not occur twice in the same pair.

Here it is convenient to introduce two rules, **firstPerson** and **secondPerson**. These rules select the first resp. second **Person** element of a given **Pair** element. These rules are shown in figure 50 and 51. When we have defined these rules,

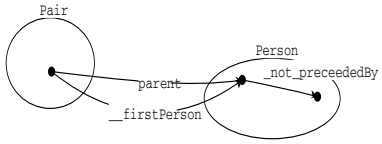

Figure 50: *Definition of the* ***firstPerson*** *property.*

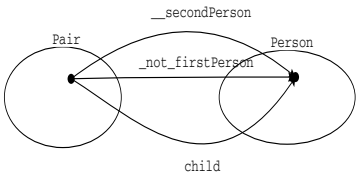

Figure 51: *Definition of the* ***secondPerson*** *property.*

we can define the invariant as shown in figure 52.

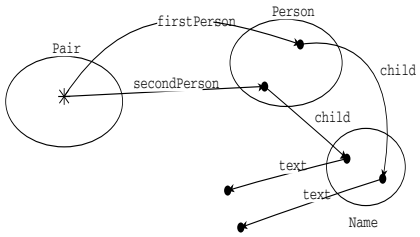

Figure 52: *The same person may only occur once in the same pair.*

The invariant is formulated as "All nodes of type **Pair** must have two distinct child nodes of type **Person**, these two nodes must in turn have **Name** nodes and the **text** property must evaluate to different values". As it is formulated the invariant also states that every **Pair** node must have (atleast) two child nodes of type **Person**, but this should always hold anyway since we assume that the instance document has been checked against the XML DTD.

We can also notice that this constraint is really not necessary since the first constraint also covers this case. One reason to provide a more specific constraint could be to get better error messages.

Third constraint:

> Two persons in the same pair must have different skill levels. In the prototypical example above I have assumed that there are only three skill levels, but this is not important.

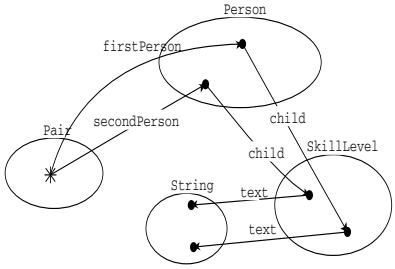

Figure 53: *Two persons in the same pair must have different skill levels.*

The constraint is formulated as "For all **Pair**s the **Person**s in the pair have different **SkillLevel**s".

7.3 Checking XHTML Documents

Now we can use the programs written for the SWEDE and translate the XHTML class hierarchy and the constraint diagrams to F-Logic and check the validity

of XHTML documents.

The class-diagrams from the previous section can be used to generate the classes, which are then used as the types in the constraint diagrams.

Let's check the static semantics of the following document:

```
<html>
  <body>
    <Pair>
      <Person>
        <Name>Johan</Name>
        <SkillLevel>High</SkillLevel>
      </Person>
      <Person>
        <Name>Johan</Name>
        <SkillLevel>High</SkillLevel>
      </Person>
    </Pair>

    <a href=''foo.org''><a href=''bar.org''>bar</a></a>
  </body>
</html>
```

As we can see the document is inconsistent with the constraints in the previous section. The XML document is translated to F-Logic facts with the **XML2FloraDB** program. So lets run the constraints defined in the diagrams on the XHTML document above. I have added appropriate error messages to the constraint diagrams by editing the XML output from the CDEditor. All you need to do check a instance document is to generate the F-Logic facts from the XML document that should be checked, generate the rules/constraints and add the error messages. Then you have to put the rules/constraints and the facts in the same file and then load it into Flora.

When loading the generated facts, schema and rules into the Flora engine we get the following result:

```
flora2 ?- [constraints_file].

[FLORA: compiling /home/johan/Compilers/XSB/person_once.flr]
      .
      .
      .
[FLORA: Done! CPU time used: 0.009998 seconds]

Yes.
ERROR: A person occurs more than once in a pair!
Yes.
ERROR: There is a nested 'a' element!
Yes.
Error: Persons within a pair have the same skill level!
Yes.
Yes.
```

More interesting error messages could be created if the translator from XML also adds a property giving the line number for each XHTML element. See appendix B.2 for the F-Logic representation of the XHTML document and constraints/rules.

8 Notes on Using the SWEDE to Define the Static Semantics of Java

In this almost final section I return to the goal that was formulated in the beginning of the thesis, namely how to use the visual environment to specify the static semantics of Java.

Much implementation and testing time was spent on this part of the project, but her I will only sketch the solution. The reason for this is that only a small set of the constraints for Java syntax was written down.

8.1 The Java Concepts

One of the intentions of this project was to see how the DAML+OIL language could be used to add static semantics descriptions to the Compost/Recoder system. Compost has already has a set of Java classes that implements the abstract syntax for Java, Compost can also read Java programs and create in-memory objects of those programs.

Figure 54 again shows the architecture of the SWEDE and it also shows how we can connect Compost to the SWEDE. First of all when creating the DAML+OIL ontology/schema for Java we can use the UML tool to reverse-engineer[29] the classes in Compost that implements the abstract syntax for Java. Reverse-engineering of code is something most UML-tools can do. This provides us with the first connection point to the SWEDE architecture, the concept (class) extraction.

Secondly the Compost system is used to read Java programs and the pretty-printing it to XML (this is done by the *RecoderAST2XML* program). This provides us with the second connection point to the SWEDE architecture, we have seen in the previous section how XML documents can be translated to F-Logic facts through the generic *XML2FloraDB* program.

Once we have achieved these connection points we can use the primitive relations/properties given by the tree ontology (section 7.1) to write down the static semantics for Java.

To implement the full semantics from the simple **child/order** relations require great care in the layering of relations, i.e. relations in level m is built with relations from level n where n *leq* m. In the bottom we have **child/order** in the second layer we might have general (but derived) properties such as ancestor. Then comes properties for selecting sub-nodes with a particular properties, e.g. if you have a *swith*-statement node you want to have the first *case*-statement. The subsequent rule-layers would include type-analysis of variables using the scope rules of Java etc. And the final layer would contain the static semantics

[29] Read the source code and produce UML class-diagrams, classes and relations, from it.

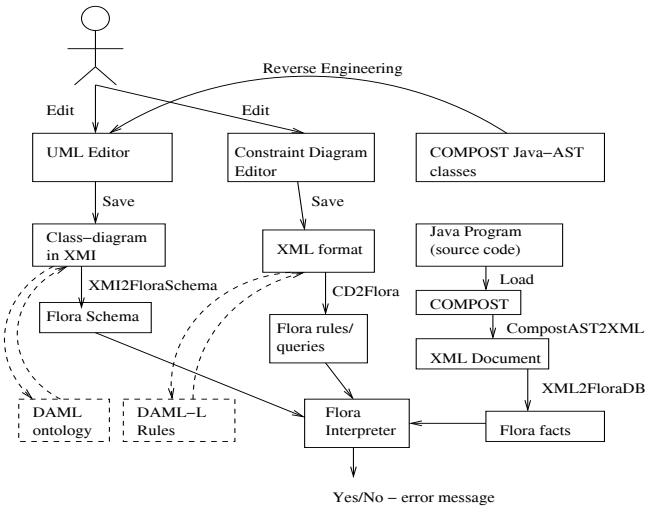

Figure 54: The connection between Compost and SWEDE. Compost AST classes are reverse-engineered to produce the DAML+OIL classes and Compost is used to create the instance documents by reading the Java program and outputting it as a XML document.

invariants such as definition-use checks (a variable must be declared before it is used) etc.

The Compost system does however already do a lot of this work. It performs type-analysis and also supports the evaluation of constant expressions. This allowed me to prettyprint much of the information I needed for writing static semantics constraints, i.e. I didn't have to start from the bottom with the **child/order** relations and build from there.

Since Compost checks some context conditions I had to modify the Compost error handling slightly. Instead of issuing a error message when reading a incorrect Java program I override the errorhandling so it continues and prints the program in XML format.

Figures 55, 56 and 57 shows some of the constraints that were tested.

8.2 Checking Statements

9 Conclusions and Summary

In this thesis we have looked at how the ontology language DAML+OIL can be represented and edited visually as UML class diagrams. We also looked at how to use the UML notation for DAML+OIL to describe the context-dependent aspects of XML documents. DAML+OIL can express a fragment of first-order predicate logic. FOPL has been used before to define the context-dependent aspects of programming languages, by charaterizing the legal form of the derivation trees. It is interesting to see how DAML+OIL can be used as to the same thing for XML trees.

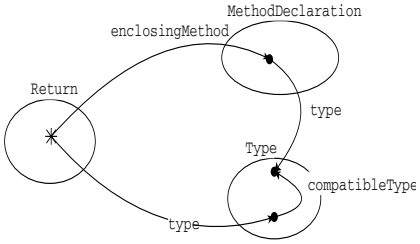

Figure 55: Invariant: The type of the return expression must be compatible with the declared return type of the method.

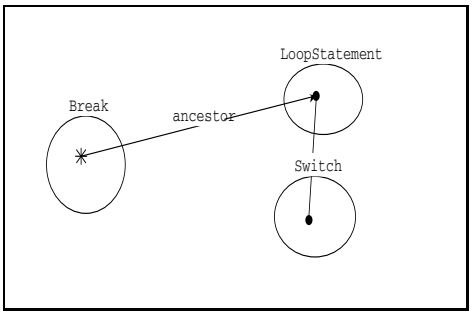

Figure 56: Invariant: A break statement can only occur in Switch and Loop statements.

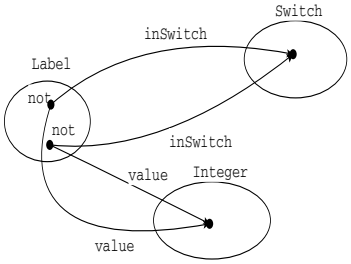

Figure 57: Invariant: All labels in the same Switch statement must evaluate to different values.

The UML notation for DAML+OIL is described in detail and we notice that the major differences between these two languages are:

- DAML+OIL supports *class constructors* (also called class expressions) which create anonymous classes by combining other classes and restricting properties.
- DAML+OIL treats properties as first-class citizens. We also define proper-

ties in terms of which classes they apply to, which is contrary to the UML concept of properties (i.e. attributes). Properties can even be declared without any domain and/or range.

- The namespaces. DAML+OIL makes heavy use of namespaces this has to be modelled with tagged values in UML.

These problems have been solved in the described mapping. The mapping tries to reuse as much as possible of existing UML concepts, one point where this fails is the need for anonymous classes in the diagram.

It became apparent that the core DAML+OIL language wasn't expressive enough to describe all kinds of context conditions. While DAML+OIL provides a rich set of operators to create classes (also called concepts) it lacks the possibility to create derived properties/rules from primitive properties, i.e. no property constructors. To overcome this problem we looked at two different visual notations to represent Horn clauses. Horn clauses are good at creating properties and one can therefore benefit from this merging. The Horn clauses are used both to create rules as well as invariants.

First, the possibility of editing rules and invariants in UML class diagrams was considered. This would give an integrated environment since the DAML+OIL classes and properties can be defined in the same diagram. This approach was however abandoned because it was difficult to create intuitive invariant specifications (rules were simpler), it became very complex to state which classes that the invariant should range over.

The second approach was the constraint diagrams. This notation has been developed as a visual alternative to UML's constraint language OCL. It uses circles and arrows to denote classes and properties between the classes. I felt that this notation was very simple to understand, so I decided to develop it further by adding the possibilty of defining rules. An editor for constraint diagrams already existed so I could use it to create diagrams. A program was developed that translates a subset of the extended constraint diagrams to F-Logic.

The *Semantic Web Editing Environment* (SWEDE) is the framework in which ontologies can be edited wtih UML/constraint diagrams and then be translated to an underlying inference engine.

The SWEDE uses the XMI format for translating ontologies to the underlying inference engine, thus making it independent of which UML editor being used. The use of constraint diagrams limits the freedom of choice since only one editor exists. But we have also presented a way of defining Horn rules in UML, so this problem could be alleviated.

Although the main purpose of the semantic Web movement is to bring machine-processable metadata to Web resources we have used the developed techniques as a uniform language for checking static semantics.

An example application of the SWEDE is demonstrated when we use it to impose context-constraints on XHTML code. We also sketch how the Compost system could use the SWEDE to check the static semantics of Java programs.

A Program Listings

A short explanation of the more important programs written during the course of the project.

A.1 XMI2FloraSchema.xsl

An XSL stylesheet that takes a UML class diagram in XMI format and maps the inheritance hierarchy (classes and the subclass axiom) to a F-Logic schema. A similar stylesheet for producing the equivalent DAML+OIL code has also been developed.

A.2 XML2FloraDB.xsl

An XSL stylesheet that takes an arbitrary XML document and produces F-Logic facts. The tag of an element is considered to be a class, so the generated F-Logic fact is an object of type tag-name. The object gets a dummy name. XML attributes are mapped to F-Logic methods. The properties/relations produced correspond to the primitive properties declared in *Tree Ontology* defined in 7.1.

A.3 RecoderAST2XML

A Java program that uses the Recoder/Compost system to read Java source code and then outputs the abstract syntax tree in XML format.

A.4 ConstraintDiagram2ClausalForm

A Java program that takes the XML-format for constraint diagrams and produces F-Logic formulas. Currently the Venn-diagrams for intersection of types hasn't been implemented.

A.5 Rose2FloraSchema

An extension script (a BASIC variant) for the Rational Rose UML editor that translates the internal logical representation of a class diagram to F-Logic. Only classes and inheritance is translated. This program was superseeded by the XMI2FloraSchema translator.

B Constraint Diagrams

B.1 XML Schema for Constraint Diagrams

An XML Schema definition of the XML serialization of constraint diagrams as given by the CDEditor. The output from the CDEditor has to be rewritten in order to be well-formed XML.

B.2 Generated F-Logic Code for The XHTML Example

The *XML2FloraDB* and *ConstraintDiagram2ClausalForm* generates prolog-style predicates on the form $P(X, Y)$ this is equivalent to the F-Logic syntax $X[P->Y]$. The Flora engine can handle both.

```
%%%%%%%%%%%%%% The ancestor relation has been manually added
%%%%%%%%%%%%%% since we don't have the XMI2FloraSchema finished.
ancestor(X, Y) :- child(X, Y).
ancestor(X, Y) :- child(X, Z), ancestor(Y, Z).

%%%%%%%%%%%%%%%%%% INDIVIDUALS generated from the XHTML file.
nodeN400001:chtml.
nodeN400002:cbody.
nodeN400003:cPair.
parent(nodeN400003, nodeN400002).
child(nodeN400002, nodeN400003).
nodeN400004:cPerson.
parent(nodeN400004, nodeN400003).
child(nodeN400003, nodeN400004).
nodeN400005:cName.
parent(nodeN400005, nodeN400004).
child(nodeN400004, nodeN400005).
text(nodeN400005, 'Johan').
nodeN400007:cSkillLevel.
parent(nodeN400007, nodeN400004).
child(nodeN400004, nodeN400007).
preceededBy(nodeN400007, nodeN400005).
text(nodeN400007, 'High').
nodeN400009:cPerson.
parent(nodeN400009, nodeN400003).
child(nodeN400003, nodeN400009).
preceededBy(nodeN400009, nodeN400004).
nodeN40000A:cName.
parent(nodeN40000A, nodeN400009).
child(nodeN400009, nodeN40000A).
text(nodeN40000A, 'Johan').
nodeN40000C:cSkillLevel.
parent(nodeN40000C, nodeN400009).
child(nodeN400009, nodeN40000C).
preceededBy(nodeN40000C, nodeN40000A).
text(nodeN40000C, 'High').
nodeN40000E:ca.
parent(nodeN40000E, nodeN400002).
child(nodeN400002, nodeN40000E).
preceededBy(nodeN40000E, nodeN400003).
href(nodeN40000E, 'foo.org').
nodeN400010:ca.
parent(nodeN400010, nodeN40000E).
child(nodeN40000E, nodeN400010).
href(nodeN400010, 'bar.org').
text(nodeN400010, 'bar').
%%%%%%%%%%%%%%%%%%%% RULES generated from constraint diagrams
firstPerson(Z1, Z2) :- Z2:cPerson, Z3:cPerson, Z1:cPair, child(Z1, Z2),
                       tnot preceededBy(Z2, Z3), child(Z1, Z3), tnot Z2:=:Z3.

secondPerson(Z1, Z2) :- Z1:cPair, Z2:cPerson, tnot firstPerson(Z1, Z2), child(Z1, Z2).

%%%%%%%%%%%%%%%%%%%% CONSTRAINTS generated from constraint diagrams
?- t.
```

```
t :- tnot r1, !.
t :- writeln('ERROR: A person occurs more than once in a pair!')@prolog().
r1 :- _Z1:cPair, tnot r2(_Z1).
r2(_Z1) :- _Z2:cPerson, secondPerson(_Z1, _Z2), _Z4:cName, child(_Z2, _Z4),
           text(_Z4, _Z7), _Z3:cPerson, firstPerson(_Z1, _Z3), _Z5:cName, child(_Z3, _Z5),
           text(_Z5, _Z6), tnot _Z4:=:_Z5, tnot _Z7:=:_Z6, tnot _Z2:=:_Z3.

?- t2.
t2 :- tnot r3, !.
t2 :- writeln('ERROR: There is a nested ''a'' element!')@prolog().
r3 :- Z1:ca, r4(Z1).
r4(Z1) :- Z2:ca, ancestor(Z1, Z2).

?- t3.
t3 :- tnot r5, !.
t3 :- writeln('Error: The persons within a pair have the same skill level!')@prolog().
r5 :- Z1:cPair, tnot r6(Z1).
r6(Z1) :- Z2:cPerson, secondPerson(Z1, Z2), Z5:cSkillLevel, child(Z2, Z5), text(Z5, Z6),
          Z3:cPerson, firstPerson(Z1, Z3), Z4:cSkillLevel, child(Z3, Z4), text(Z4, Z7),
          tnot Z6:=:Z7, tnot Z2:=:Z3, tnot Z5:=:Z4.
```

C Tools used

In this project I have used a number of software tools (and investigated a few more). Here I list the major tools that have been used, together with the version number and where (if at all) they can be downloaded. This is to make it easier for potential users of the written programs to find the correct tools.

Tool	Version	Web site
ArgoUML	0.9.3	http://argouml.tigris.org
Castor	0.9.3	http://castor.exolab.org
CDEditor	0.29	http://www.cs.technion.ac.il/Labs/ssdl/research/cdeditor/
COMPOST	0.63	http://i44w3.info.uni-karlsruhe.de/~compost
Dia	http://www.lysator.liu.se/~alla/dia/	http://www.lysator.liu.se/~alla/dia/
Flora/XSB	2.4	http://xsb.sourceforge.net
LaTeX	3.14159	http://www.tug.org
MAX	8.0.0	http://www.informatik.fernuni-hagen.de/import/pi5/download/
Poseidon UML	1.0	http://www.gentleware.com
Rational Rose	2001 Unix	www.rational.com
Wine	FIXME	http://www.winehq.org
Xalan-Java	2.2	http://xml.apache.org/xalan
Xfig	3.2.3	http://www.xfig.org

References

[1] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes III, J. Letkowski, and M. Aronson, *Extending UML to Support Ontology Engineering for the Semantic Web*, 2001.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila, *The Semantic Web*, 2001, Scientific American, http://www.sciam.com/2001/0501issue/0501berners-lee.html.

[3] H. Boley, S. Tabet, and G. Wagner, *Design Rationale of RuleML: A Markup Language for Semantic Web Rules*, 2001, http://www.dfki.uni-kl.de/ruleml/.

[4] G. Booch, M. Christerson, M. Fuchs, and J. Koistinen, *UML for XML Schema Mapping Specification*, August 1999, http://www.rational.com/media/uml/resources/media/uml_xmlschema33.pdf.

[5] Alexander Borgida, *On the Relative Expressiveness of Description Logics and Predicate Logics*, Artificial Intelligence **82** (1996), no. 1-2, 353–367.

[6] *DAML+OIL Specification*, Mar 2001, http://www.daml.org/2001/03/daml+oil-index.html.

[7] D. Fensel, *The Semantic Web and its Languages*, November/December 2000, IEEE Intelligent Systems, FIXME: URL.

[8] J Gil, J Howse, and S Kent, *Constraint Diagrams: A Step Beyond UML*, Proceedings of TOOLS USA'99, IEEE Computer Society Press, December 1999.

[9] ———, *Formalizing Spider Diagrams*, Proceedings of IEEE Symposium on Visual Languages (VL99), IEEE Computer Society Press, December 1999.

[10] Ian Horrocks, *Reasoning with DAML+OIL: What can it do for YOU?*, Lecture Notes, http://www.cs.man.ac.uk/~horrocks/Slides.

[11] ISO, *ISO C Standard*, 1999, http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/n843.htm.

[12] S. Kent, *Constraint Diagrams: Visualizing Invariants in OO Modelling*, Proceedings of OOPSLA97, ACM Press, October 1997, pp. 327–341.

[13] S. Kent and Y. Gil, *Visualising Action Contracts in OO Modelling*, IEE Proceedings: Software, 145, no. 2-3, April 1998, pp. 70–78.

[14] S. Kent and J. Howse, *Mixing Visual and Textual Constraint Languages*, Proceedings of UML'99, IEEE Computer Society Press, October 1999.

[15] M. Kifer, G. Lausen, and J. Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages*, ??, http://FIXME.

[16] M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks, *The Relation Between Ontologies and XML Schemas*, FIXME, FIXME.

[17] Alon Y. Levy and Marie-Christine Rousset, *CARIN: A representation language combining horn rules and description logics*, European Conference on Artificial Intelligence, 1996, pp. 323–327.

[18] Andreas Ludwig, *COMPOST Technical Manual*, April 2001, http://i44w3.info.uni-karlsruhe.de/~compost.

[19] Object Management Group (OMG), *OMG Unified Modeling Language Specification*, June 1999, Version 1.3.

[20] Martin Odersky, *A new approach to formal language definition and its application to Oberon*, Ph.D. thesis, No. 18, Reihe Informatik Dissertationen der ETH Zuerich, 1989.

[21] ———, *Defining context-dependent syntax without using contexts*, ACM Transactions on Programming Languages and Systems **15** (1993), no. 3, 535–562.

[22] A. Poetzsch-Heffter, *Programming Language Specification and Prototyping Using the MAX System*, 1993.

[23] S. Cranefield, *UML and the Semantic Web*, Feb 2001, FIXME.

[24] ———, *Networked Knowledge Representation and Exchange using UML and RDF*, FIXME, FIXME.

[25] Robert W. Sebesta, *Concepts of Programming Languages*, Addison-Wesley, 1996.

[26] Michael Sintek and Stefan Decker, *TRIPLE - An RDF Query, Inference, and Transformation Language*.

[27] J.F. Sowa, *Building, Sharing and Merging Ontologies*, http://FIXME.

[28] Ulf Nilsson and Jan Maluszynski, *Logic, Programming and Prolog*, John Wiley and Sons Ltd., 1995, second edition.

[29] W3C, *Resource Description Framework (RDF) Model and Syntax Specification*, Feb 1999, http://www.w3.org/TR/REC-rdf-syntax.

[30] W3C, *XSL Transformations (XSLT) Version 1.0*, Nov 1999, http://www.w3.org/TR/1999/REC-xslt-19991116.

[31] W3C, *Resource Description Framework (RDF) Schema Specification 1.0*, Mar 2000, http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

[32] W3C, *XHTML 1.0: The Extensible HyperText Markup Language*, January 2000, http://www.w3.org/TR/xhtml1.

[33] W3C, *XML Schema Part 0: Primer*, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[34] N. Walsh, *A Technical Introduction to XML*, http://nwalsh.com/docs/articles/xml/.

[35] Guizhen Yang and Michael Kifer, *Flora-2: User's Manual*, July 2001, http://xsb.sourceforge.net.