CS 4997

Honours Thesis Project Report

OO jDREW: Design and Implementation of a Reasoning Engine for the Semantic Web

Marcel A. Ball
3066379

April 6, 2005

Abstract

With a growing interest in Semantic Web technologies there has been an increased interest in reasoning engines and rule encodings for the Semantic Web. One specification for the markup of rules for the Semantic Web is the RuleML Initiative[3], which combines positional logic with object-oriented knowledge representation.

The primary purpose of this report is to describe the design and implementation of OO jDREW, a Java based reasoning engine designed to support the features that are available in RuleML and the Object-Oriented extensions for RuleML. In order to implement the features present in OO RuleML a unification algorithm has to be implemented that is capable of dealing with a mixture of positional and keyed parameters as well as order-sorted types; object identity and built-ins are also taken dealt with. Moreover, underlying data structures to represent knowledge bases and operations on those data structures have to be implemented.

The engine that was developed as part of the project implements the major features of OO RuleML and is available online at `http://www.jdrew.org/oojdrew`. The current implementation is suitable for testing and executing small to medium size knowledge bases in RuleML. Plans for an advanced indexing system, lacking in the current implementation, are outlined to improve scaleability of the engine for large knowledge bases.

Contents

List of Figures

Chapter 1

Project Goals

The goal of this honours thesis project is to develop the core components of a reasoning engine that is suited to the needs of the Semantic Web community. The reasoning engine's design is based on the jDREW [12] reasoning engine and Object-Oriented RuleML (OO RuleML) [4], with the ultimate goal of becoming the reference implementation for the derivation rules subset of the RuleML specification. To achieve this goal three major features, somewhat inspired by features in other logic systems, but not present in standard Prolog systems, have been added to the engine.

The first major new feature, non-positional named ("keyed") slots, similar to the slots in F-logic or object instance variables in Java, has been added to the system. This allows an object-centered knowledge representation with named attributes, instead of a sequence of positional arguments. Accomplishing this required the design and implementation of a unification algorithm that works with a combination of both positional arguments and non-positional named slots.

The second major feature is order-sorted types for terms. This allows users of the system to give user-defined types to variables, individual constants and complex terms, which can then be used to restrict the application of rules so that they are only applied when the data type restrictions are met. Additionally, hierarchical typing provides the link to the Semantic Web's light-weight RDF Schema (RDFS) ontologies.

The final major feature is object identifiers (oids), which allow users to provide identification terms to rules and facts. The object identifiers are also intended to allow URI anchoring and referencing, but this has not been implemented as there are unresolved W3C issues with normalizing URIs; therefore object identifiers are currently restricted to symbolic names.

In addition to these major features this reasoning engine supports a set of built-in relations based upon the SWRL built-ins proposal [7]. Many useful relations, for example a numeric *greater than*, cannot be expressed practically as a finite set of rules and facts; therefore it is a necessity to have a system for built-in relations. The SWRL built-ins proposal covers a large subset of the useful relations that are candidates for being implemented as built-ins and has been gaining support within the Semantic Web Rules community; it was therefore chosen as the basis for the set of built-ins included with the reasoning engine.

Chapter 2

Background Information

The Semantic Web is an extension of the current World Wide Web technology with the goal of allowing improved machine interpretation of web information and better enabling computers and people to work together[2]. By adding well-defined semantics to web objects, inference rules can be used to process the formal meaning of information on the Web.

Typically the meaning of information on the Semantic Web is expressed in RDF (Resource Description Framework) using sets of subject-property-object triples to encode metadata. The RDF system has several advantages that makes it ideal for describing Semantic Web information: first, it has an open-ended non-constrained data model making it easy to expand the semantic description of resources; second, its simple data model and W3C-recommended formal semantics provides a basis for reasoning about the meaning of RDF encoded information[10].

One common issue encountered when reasoning about semantic information is that different knowledge bases may use different terminology to describe concepts that are similar or the same[2]. One approach to this issue is to describe relationships between terms in the form of ontologies, typically taxonomies with associated inference rules; the encoding of these relationships allows computers to correctly interpret and reason about information when different terminologies are used. One common encoding of the relationships between terms is RDF Schema, which allows the modeling, again in standard RDF, of taxonomic relationships between classes or properties as well as domain and range restrictions of properties. Other more expressive systems, such as the OWL Web Ontology Language, have been proposed for describing ontologies.

2.1 Existing Rule Systems

While there are many logic programming systems already in existence, none of these systems are able to use all of the capabilities described in the RuleML specification. Several of these logic programming systems served as inspiration for OO RuleML and the design of the OO jDREW reasoning engine.

- Prolog

  One of the earliest logic programming systems is Prolog (Programmation en logique), developed in Marseilles, France, in the 1970s. Standard Prolog systems use a positional knowledge representation; this positional representation makes it difficult to use Prolog for reasoning about Semantic Web information in the object-centered RDF. While standard Prolog systems do not include support for order-sorted types, order-sorted types have been added to Prolog like systems in previous work such as Huber et al. 1987 [8] and Furbach et al. [6].

  The positional components of the POSL syntax, an ASCII representation of RuleML rules that is supported by OO jDREW, is inspired by Prolog's syntax for Hornrules[5]. Additionally, standard Prolog rules can be represented in RuleML (or POSL) and can be used in the reasoning engine that was developed.

- F-logic

  F-logic (Frame Logic) is a declarative programming formalism developed by Michael Kifer, Georg Lausen and James Wu starting in 1989. Unlike Prolog systems, F-logic is a declarative object-oriented language with features that allow for typing and non-positional knowledge representation [9].

  F-logic's frames are similar to the slots from RuleML, and served as the basis for the slotted components of the POSL syntax[5] that is supported by OO jDREW. While F-logic does support object-oriented knowledge representation, it is not suitable for RuleML knowledge, as non-positional arguments can only be used in object definitions and not in relations.

- jDREW

  jDREW (Java Deductive Reasoning Engine for the Web) is a highly configurable reasoning engine written in the Java programming language. The system includes both goal-driven backward chaining of rules, similar to most Prolog systems, and a data-driven forward-chaining engine[12]. By utilizing the Java programming language the jDREW engine is portable across platforms and easily integrated into Java based web applications. These design decisions make jDREW an excellent basis for the design of a reasoning engine for the Semantic Web, as portability and easy integration with existing web technologies is critical; therefore jDREW was chosen as inspiration for the design of OO jDREW, in particular the top-down search strategy.

2.2 Rules and the Semantic Web

In order for the Semantic Web to be able to function and computers to be able to interpret the meaning of information on the Semantic Web, information must be encoded in a structured format, primarily RDF, and sets of inference rules must be constructed that allow for automated reasoning [2]. This need to reason about structured semantic data has created an interest within the Semantic Web community in reasoning systems and the encoding of rules for use within Semantic Web systems. The Rule Markup Initiative[1] is working towards the definition of an XML-based rule language for Web-based rule storage, exchange and execution[3].

A set of extensions to RuleML, described in "Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms"[4], have been proposed; these extensions expand RuleML from a primarily positional knowledge representation to an object-oriented knowledge representation system. Unfortunately, reasoning engines, such as jDREW[2] and Mandarax[3], which were capable of utilizing positional knowledge that was represented in RuleML, are unable to use the newer object-oriented version of RuleML. This created an interest in designing a reasoning engine that was capable of dealing with both the positional and object-oriented components of RuleML. The reasoning engine that was developed, OO jDREW, is intended to be the reference implementation for the derivation-rule subset of the RuleML specification.

An overview of the RuleML implementation is given in Ball et al. 2005[1]. An application to Semantic Match-Making in the live portal Teclantic[4] can be found in Yang et al. 2005[13].

[1]http://www.ruleml.org/
[2]http://www.jdrew.org/
[3]http://mandarax.sourceforge.net/
[4]http://www.teclantic.ca/

Chapter 3

Keyed Parameters (Slots)

The first major feature provided in the Object-Oriented extensions to RuleML is the inclusion of user level roles, which allow the use of unordered, keyed arguments for atoms and complex terms[4]. These user level roles (`<slot>` in RuleML) are comparable to frames in F-logic, or instance variables in Object-Oriented languages such as Java or C++. The introduction of slots eases the creation and maintenance of knowledge bases, in particular those with larger, multi-argument structures.

In a positional system, such as Prolog or the non-Object-Oriented RuleML sub-languages, the left-to-right order of the arguments of atoms and complex terms is significant. This positional nature creates an extra burden on users of the system, as they must remember and interpret the meaning of these positional arguments. This also creates problems with the maintenance of knowledge bases, as the maintainer must know and remember the conventions used in the ordering of the arguments. Additional problems are encountered when adding arguments to atoms and structures as the positional signature capturing the order of arguments must be adapted and existing rules must be updated to use this new ordering.

For example, consider the positional fact in Figure 3.1; this could either be interpreted as Henry is the father of George *or* George is the father of Henry, depending on implicitly underlying positional signature with respect to the order of the arguments.

```
<Atom>
  <Rel>father</Rel>
  <Ind>Henry</Ind>
  <Ind>George</Ind>
</Atom>
```

Figure 3.1: Positional Father Fact

This fact could be augmented with the slots from RuleML to remove the ambiguity, as in Figure 3.2. In this case the relationship is no longer ambiguous, as the user level role names in the slots clarify the meaning of the relation, making the knowledge base easier to interpret.

```
<Atom>
   <Rel>father</Rel>
   <slot><Ind>parent</Ind><Ind>Henry</Ind></slot>
   <slot><Ind>child</Ind><Ind>George</Ind></slot>
</Atom>
```

Figure 3.2: Slotted Father Fact

Additionally, the slotted fact in Figure 3.2 can unambiguously be matched with either of the two queries in Figure 3.3 and Figure 3.4, as the introduction of named slots removes the need to maintain a single order for the arguments.

```
<Query>
   <Atom>
      <Rel>father</Rel>
      <slot><Ind>parent</Ind><Ind>Henry</Ind></slot>
      <slot><Ind>child</Ind><Var>who</Var></slot>
   </Atom>
</Query>
```

Figure 3.3: Example Slotted Query #1

In the following we regard both atoms and complex terms as just terms, as is done in the implementation. To provide an efficient unification algorithm, for combined positional-slotted terms, we assume that the arguments are internally ordered in some canonical manner. Since the named slots do not require any particular order, an artificial order (such as a lexicographical order), can be imposed internally upon the slots. In OO jDREW the ordering of slots is the encounter order, where slots are ordered based upon the order in which they are first encountered; this ordering is performed when creating the internal data structures that represent the terms. In OO jDREW, when an internal representation of a term with arguments (Atom, Cterm, or Plex) is created, the term is normalized so that the arguments are in the following order: the object identifier (more details in Chapter 5), positional parameters (in their original order relative to other positional parameters), the positional rest parameter (more details in Section 3.1) if it exists, the slotted parameters (in the encounter order), and finally, the slotted rest parameter (more details in Section 3.1), if it exists.

```
<Query>
   <Atom>
      <Rel>father</Rel>
      <slot><Ind>child</Ind><Var>who</Var></slot>
      <slot><Ind>parent</Ind><Ind>Henry</Ind></slot>
   </Atom>
</Query>
```

Figure 3.4: Example Slotted Query #2

By having the terms in this normalized form it is possible to implement a unification algorithm that runs in $O(m+n)$ time (where m and n are the number of arguments in the terms). This ordering allows the algorithm to scan the two lists of arguments, matching roles (and position in case of positional parameters) and unifying the fillers in each of the argument lists. If there is a role present in one term, but not in the other, then either the unification will fail, or it will be matched by one of the rest variables, if they exist, as detailed in Section 3.1.

3.1 Rest Variables

One of the main advantages of the keyed arguments (slots) comes with the introduction of rest variables. A slotted rest variable, indicated by the `<resl>` tag in RuleML (to be introduced in version 0.89, although it is already supported by OO jDREW), will be matched with any named parameters that are not matched with regular named parameters. When a term t with a rest variable X is unified with another term t‘, any slots that are in t‘ that are not in t will be matched with the rest variable X and used to generate a `Plex` which will be assigned to X. This allows the user of the system to specify that they want certain slots to be present, but there can be other arbitrary slots which are not required.

```
<Cterm>
   <Ctor>person</Ctor>
   <slot><Ind>name</Ind><Ind>John Doe</Ind></slot>
   <slot><Ind>age</Ind><Ind>28</Ind></slot>
   <slot><Ind>sex</Ind><Ind>male</Ind></slot>
</Cterm>
```

Figure 3.5: Slotted example: Complex Term (Cterm) #1

For example, consider the unification of the complex terms in Figures 3.5 and 3.6. The complex term in Figure 3.6 only has an *age* slot and does not have a rest variable;

therefore it must match exactly with the complex term in Figure 3.5; since there are *name* and *sex* slots in the first Cterm, they will not unify.

```
<Cterm>
   <Ctor>person</Ctor>
   <slot><Ind>age</Ind><Var>age</Var></slot>
</Cterm>
```

Figure 3.6: Slotted example: Complex Term (Cterm) #2

Instead, if you consider the complex terms in Figures 3.5 and 3.7, the second complex term has a rest variable, and will thus unify with the first complex term. In this case the rest variable will be bound to a `Plex` containing the remaining slots that are not matched. This is very useful when working with complex structures in your knowledge base, as your rules only need to specify the slots that are relevant, instead of requiring the presence of extra slots with "null values" or anonymous variables. Additionally, you have the ability to add new slots to your data structures without affecting your already existing rules.

```
<Cterm>
   <Ctor>person</Ctor>
   <slot><Ind>age</Ind><Var>age</Var></slot>
   <resl><Var>X</Var></resl>
</Cterm>
```

Figure 3.7: Slotted example: Complex Term (Cterm) #3

In the unification of two terms with arguments (Atom, Cterm or Plex) if a user role is present in only one of the two terms the following will occur: if the term without the user role in question contains a slotted rest variable then the slot is added to a list of unmatched roles; this is used to create a Plex (`<Plex><slot> ... </slot> ... </Plex>`) which will be unified with the rest argument. If the term does not have a slotted rest variable then unification will fail.

The slotted rest term (signified by the `<resl>` role tag) has a positional analogue. The positional rest term (signified by the `<repo>` role tag) will match an arbitrary number of positional terms that are not matched during the regular unification; this is very similar to Prolog's "`|`" operator, although it works in all terms with arguments (Atoms, Complex terms and Plexes), not just in lists (a special kind of plex). For example, consider the two Cterms in Figures 3.8 and 3.9; in this case the first positional parameter in each will be matched, with *who* being bound to *John Doe*, and the remaining parameters in the first term will be matched with the anonymous positional rest variable (`<Var />`), after becoming a `Plex` (`<Plex>...</Plex>`) containing the three additional parameters.

```
<Cterm>
   <Ctor>person</Ctor>
   <Ind>John Doe</Ind>
   <Ind>22</Ind>
   <Ind>male</Ind>
   <Ind>Fredericton</Ind>
</Cterm>
```

Figure 3.8: Positional example: Complex Term (Cterm) #1

```
<Cterm>
   <Ctor>person</Ctor>
   <Var>who</Var>
   <repo><Var /></repo>
</Cterm>
```

Figure 3.9: Positional example: Complex Term (Cterm) #2

Chapter 4

Order-Sorted Types

The second major feature introduced in the Object-Oriented version of the RuleML (OO RuleML) specification and supported by OO jDREW is the inclusion of an order-sorted typing system for terms[4]. The typing system gives users the ability to define a partial order for a set of types to be used by the reasoning system. By having an order-sorted type system built into the reasoning engine users can easily provide types to terms, and the system will automatically restrict the clause search space to only those clauses that are appropriate based upon the specified types.

Currently, users of the system define the type sorts in the RDFS XML syntax, thus allowing the reasoning engine to take advantage of the lightweight taxonomies of the Semantic Web. The system could easily be expanded to support other representations for type sorts, such as the OWL Web Ontology Language.

The engine includes a small set of predefined types, based upon the basic data types defined for XSD (XML Schema Definition, Part 2). Currently this is a minimal subset consisting of equivalents for `xsd:string`, `xsd:integer`, and `xsd:float`, as well as a non-XSD type, `numeric`, which is a base type for all numeric types. These built-in types are utilized by the built-in relations that are provided, as detailed in chapter 6. As the set of built-in relations provided by the system is expanded, the built-in types provided would be expanded to include a larger subset of the XSD types. In addition to the XSD types that are defined, two system types are defined: `Thing` which is the root or top of the type lattice: all types inherit from this type; and `Nothing` which is the bottom of the type lattice: it inherits from all types in the system.

The current type system in OO jDREW only allows the user to model the taxonomic relationships between the types, and does not give the ability to model the domain and range restrictions of properties. While the system can model that a *Car* is a *Motor Vehicle*, it is unable to model that the property *make* has the domain *Car* or even that a *Car* must have properties for *year*, *colour*, *model*, etc. While such an object-centered signature does not change the operational semantics, it is useful in assisting in the creation of knowledge bases that are correct, as mistakes in the modeling can be detected during parsing instead of during reasoning. Extensions to the current type system are being considered; these are discussed further in Section 8.2.

4.1 Representation of Types in OO jDREW

In order to implement the order-sorted type system in an efficient manner, the types are internally represented as a directed acyclic graph(dag), in which the types are represented as nodes within the graph, and edges represent the "subtype to supertype" relationships of the types. By utilizing a dag to represent the types and their subtype/supertype relationships, we are able to implement all type operations that are needed using simple graph algorithms [6].

The type graph is defined as the set of vertices $V = \{t \mid t\ is\ a\ type\}$ and the set of directed edges $E = \{(u,\ v) \mid u\ \in\ V,\ v\ \in\ V\ ,\ u\ is\ the\ subtype\ of\ v\}$.

There are two main operations that must be done by the type system: the first is determining if one type inherits from another type, and the other is determining type intersection, i.e. the most general type that inherits from two specified types. Determining if a type t‘ inherits from type t can be accomplished by determining if there is a directed path from t‘ to t in the type graph. To determine the intersection of two types t and t‘ the system finds the greatest lower bound of the two type nodes within the type graph [6]. If t and t‘ have a non-empty intersection defined in the type graph the result of finding the greatest lower bound will be the intersection type; otherwise it will be *Nothing* indicating that there is no user defined type that is the intersection of t and t‘. While "t‘ subtype t" is formally equivalent to "t‘ intersection t equals t‘", in practice a path finding algorithm is used.

For example, consider the type graph in Figure 4.1; if the system needed to check if *ToyotaCorolla* inherits from *Car* we would use a graph path finding algorithm to look for a path from the *ToyotaCorolla* node to the *Car* node; since this directed path exists we would know that a *ToyotaCorolla* is a *Car*. If the system were to check for a directed path from the *ToyotaCorolla* node to the *Van* node no path would be found; therefore the system could conclude that a *ToyotaCorolla* is not a *Van*. If the system needed to find the type intersection of *Sedan* and *Van* it would find the greatest lower bounds of those nodes; the result of this operation would be *Nothing* which indicates that there is nothing that is both a *Sedan* and a *Van*. On the other hand, if the system found the intersection of the *PassengerVehicle* and *Van* types it would find *MiniVan*, indicating that a *MiniVan* is both a *PassengerVehicle* and a *Van*.

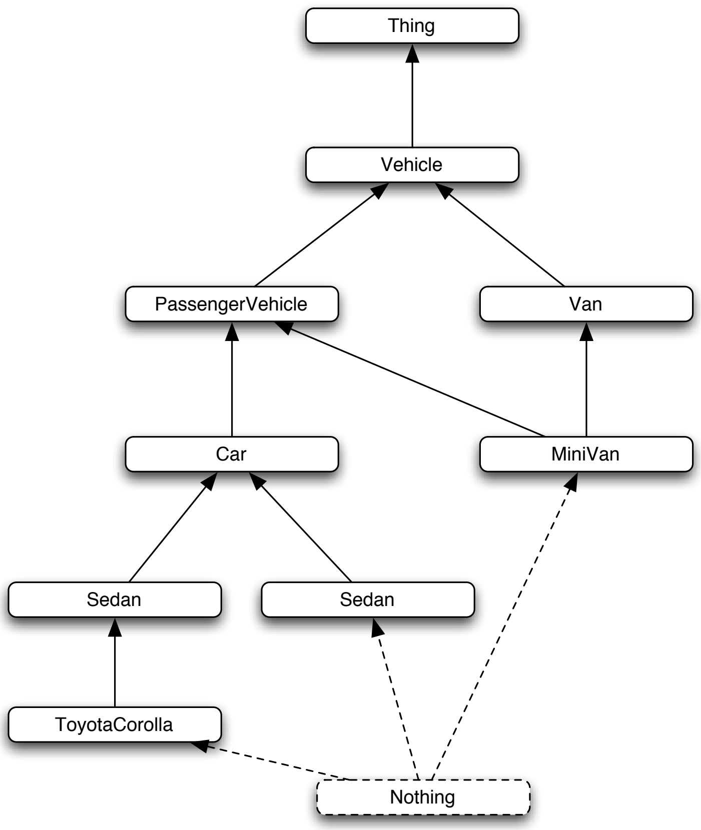

Figure 4.1: Example type graph

4.2 Typed Term Unification

When unifying two terms that are defined to have types from the type sorts, the unification can be broken into three different cases. The first case is the unification of two typed variables; the second is the unification of a typed variable and a typed constant or complex term; finally, the third is the unification of two typed constants or complex terms.

In the first case, the unification of two typed variables, the result of the unification is a variable, which in a later resolution step can be unified with another term. In a typed system, any term that is going to be unified with the resulting variable must be able to unify with both of the variables in the original unification. In order for this to be enforced, the type of the resulting variable must be the intersection of the two original types.

For example, consider the following two unifications, with respect to the types defined in Figure 4.1:

1. The unification of the variables `<Var type="Van">Var1</Var>` and `<Var type= "PassengerVehicle">Var2</Var>`: the variable resulting from the unification will have a type of `MiniVan`.
2. The unification of the variable `<Var type="MiniVan">Var1</Var>` and `<Var type="Sedan">Var2</Var>`: the variable resulting from the unification will have a type of `Nothing`, as there is no type that inherits from both `MiniVan` and `Sedan`.

In the second case, the unification of a typed variable with a typed non-variable term (Individual constant, Complex term, or Plex), in order for the types to be satisfied and the unification to succeed the type of the constant must either be the same as the type of the variable, or inherit from the type of the variable.

For example, consider the following unifications, with respect to the types defined in Figure 4.1:

1. The unification of `<Var type="Van">Var</Var>` with `<Ind type="Van">1998 Ford Econoline Van</Ind>`: since the types are the same, the unification will succeed and `<Var type="Van">Var</Var>` will be bound to `<Ind type="Van"> 1998 Ford Econoline Van</Ind>`.
2. The unification of `<Var type="Van">Var</Var>` with `<Ind type="MiniVan"> 1999 Dodge Caravan</Ind>`: since the type of the constant (`MiniVan`) inherits from the type of the variable (`Van`) the unification will succeed and `<Var type="Van">Var</Var>` will be bound to `<Ind type="MiniVan">1999 Dodge Caravan </Ind>`.
3. The unification of `<Var type="Van">Var</Var>` with `<Ind type="Sedan"> 2003 Toyota Corolla</Ind>`: since the type of the constant (`Sedan`) does not inherit from the type of the variable (`Van`) the unification will fail.

In the third case, the unification of two typed non-variable terms, in order for the types to be satisfied and the unification to succeed the type of the constant from the fact or head of a rule must be the same as or inherit from the type of the constant from the body of a rule or query, and the regular symbolic unification must succeed.

For example consider the following unifications, with respect to the types defined in Figure 4.1(in all cases the term from the fact or head of a rule is first and the term from the body of a rule or query is second):

1. The unification of `<Ind type="MiniVan">1999 Dodge Caravan</Ind>` and `<Ind type="Van">1999 Dodge Caravan</Ind>`: since the type of the constant from the fact (or head of a rule) inherits from the type of the constant from the query (or body of a rule) the types are compatible and unification will succeed.

2. The unification of `<Ind type="Van">1999 Dodge Caravan</Ind>` and `<Ind type ="MiniVan">1999 Dodge Caravan</Ind>` (order is reversed): since the type of the constant from the fact (or head of a rule) does not inherit from the type of the constant from the query (or body of a rule) the types are not compatible and unification will fail.

3. The unification of `<Ind type="MiniVan">1999 Dodge Caravan</Ind>` and `<Ind type="Van">1995 Toyota Sienna</Ind>`: while the types are compatible as in example 1 the regular symbolic unification will fail, causing the overall unification to fail.

4.3 Typed Clause Example

The example facts and query in this section are meant to illustrate how the inclusion of order-sorted types limits the search space. The two facts in Figures 4.2 and 4.3 represent the base prices for insurance for a *Car* and a *Van* respectively. The query in Figure 4.4 has a goal of `base_price`, where the vehicle is defined as having a type of *ToyotaCorolla*. In these examples the first fact (Figure 4.2) will match, as according to our type graph (Figure 4.1) *ToyotaCorolla* inherits from *Car*, therefore the variable of type *Car* will unify with the *ToyotaCorolla* complex term. In these examples the second fact will not match, as *ToyotaCorolla* does not inherit from *Van*, causing the unification of the typed variable and the typed complex term to fail.

If atoms, such as those of the two facts (Figures 4.2 and 4.3), were the heads of rules only the rule body of the first rule would even be entered by a resolution step, thus reducing the search space.

```
<Atom>
   <Rel>base_price</Rel>
   <slot>
      <Ind>customer</Ind>
      <Plex>
         <slot><Ind>sex</Ind><Ind>male</Ind></slot>
         <resl><Var /></resl>
      </Plex>
   </slot>
   <slot><Ind>vehicle</Ind><Var type="Car" /></slot>
   <slot><Ind>price</Ind><Ind type="Float">650.00</Ind></slot>
</Atom>
```

Figure 4.2: Example fact with slots and types #1

```
<Atom>
   <Rel>base_price</Rel>
   <slot>
      <Ind>customer</Ind>
      <Plex>
         <slot><Ind>sex</Ind><Ind>male</Ind></slot>
         <resl><Var /></resl>
      </Plex>
   </slot>
   <slot><Ind>vehicle</Ind><Var type="Van" /></slot>
   <slot><ind>price</Ind><Ind type="Float">725.00</Ind></slot>
</Atom>
```

Figure 4.3: Example fact with slots and types #2

```
<Query>
   <Atom>
      <Rel>base_price</Rel>
      <slot>
         <Ind>customer</Ind>
         <Plex>
            <slot><Ind>sex</Ind><Ind>male</Ind></slot>
            <slot><Ind>name</Ind><Ind>John Doe</Ind></slot>
            <slot><Ind>age</Ind><Ind>28</Ind></slot>
         </Plex>
      </slot>
      <slot>
         <Ind>vehicle</Ind>
         <Cterm type="ToyotaCorolla">
            <Ctor>vehicle</Ctor>
            <slot><Ind>make</Ind><Ind>Toyota</Ind></slot>
            <slot><Ind>model</Ind><Ind>Corolla</Ind></slot>
            <slot><Ind>year</Ind><Ind>2001</Ind></slot>
            <slot><Ind>colour</Ind><Ind>Green</Ind></slot>
         </Cterm>
      </slot>
      <slot><Ind>price</Ind><Ind type="Float">800.00</Ind></slot>
   </Atom>
</Query>
```

Figure 4.4: Example Query with slots and types

4.4 Types as Unary Datalog Predicates

In Datalog and Hornlog systems (such as Prolog) variable types in the head can be reduced to a set of unary predicates that are called as extra goals in the body. While this allows you to implement types in a system that has no native support for term typing, these types are more limited and less efficient than a built-in type system.

The biggest limitation of types as a set of unary predicates is the speed difference caused by extra resolution steps. For example, consider the rule (and supporting fact) in Figure 4.5 and the query (and supporting fact) in Figure 4.6 (these are the fact and query in Figures 4.2 and 4.4 respectively, after conversion to be positional and to use unary predicates for typing; the rules to define the types are included in Appendix A); to prove this query using unary predicate typing requires five resolution steps (the main resolution step and four resolution steps to prove the typing constraints) compared to the single resolution step using the built-in type sorts system. Even in this simple example, with only two facts and a small set of types, the differences are significant, and the differences will increase dramatically as the knowledge base and/or type sorts increase in complexity.

As a comparison to illustrate the benefits of having an order-sorted type system built into the reasoning engine, the unary predicate version given in Figures 4.5 and 4.6 was run and compared with the version that uses the built-in types. The time differences are fairly significant, cutting the deduction time almost in half, and the differences will only increase as the complexity of the types grows.

	Built-in types	Unary Predicates
Time to issue query 150 times	265 ms	473 ms
Avg. time for single query	1.7666 ms	3.15 ms

Table 4.1: Results from built-in types and unary predicates comparison

Moreover, using a set of unary predicates for term typing reduces the ability to provide generic built-in relations. In a system with built-in term typing you can define a generic built-in relation, for example *greater than*, that will perform the correct comparison based upon the types of the terms. In a system employing unary predicates for term typing using the types in built-in relations is not easily accomplished, as the types are not available at the time the built-ins are invoked. This will typically lead to one of two situations, neither of which is ideal: either multiple type-specific built-in relations will have to be defined where a single generic built-in would have sufficed (for example, our generic *greater than* may become *numeric greater than*, *date greater than* and *string greater than*), or the built-ins are forced to rely on naive assumptions about the terms, such as doing a numerical comparison if they could be interpreted as numbers, and otherwise defaulting to a string comparison.

```
<Atom>
   <Rel>Float</Rel>
   <Ind>800.00</Ind>
</Atom> <!-- This defines 800.00 as being of type Float -->

<Implies>
   <Atom>
      <Rel>Car</Rel>
      <Var>vehicle</Var>
   </Atom>
   <Atom>
      <Rel>base_price</Rel>
      <Cterm>
         <Rel>customer</Rel>
         <Var /> <!-- name -->
         <Var /> <!-- age -->
         <Ind>male</Ind> <!-- sex -->
      </Cterm> <!-- customer -->
      <Var>vehicle</Var> <!-- vehicle -->
      <Ind>800.00</Ind> <!-- price -->
   </Atom>
</Implies>
```

Figure 4.5: Example typed “fact” using unary predicates

```
<Atom>
   <Rel>ToyotaCorolla</Rel>
   <Cterm>
      <Ctor>vehicle</Ctor>
      <Ind>Toyota</Ind> <!-- make -->
      <Ind>Corolla</Ind> <!-- model -->
      <Ind>2001</Ind> <!-- year -->
      <Ind>green</Ind> <!-- colour -->
   </Cterm>
</Atom>
<!-- This fact is required to define the complex
        term as being of type ‘‘ToyotaCorolla’’ -->

<Query>
   <And>
      <Atom>
         <Rel>base_price</Rel>
         <Cterm>
            <Rel>customer</Rel>
            <Ind>John Doe</Ind> <!-- name -->
            <Ind>28</Ind> <!-- age -->
            <Ind>male</Ind> <!-- sex -->
         </Cterm> <!-- customer -->
         <Cterm>
            <Ctor>vehicle</Ctor>
            <Ind>Toyota</Ind> <!-- make -->
            <Ind>Corolla</Ind> <!-- model -->
            <Ind>2001</Ind> <!-- year -->
            <Ind>green</Ind> <!-- colour -->
         </Cterm> <!-- vehicle -->
         <Var>price</Var> <!-- price -->
      </Atom>
      <Atom>
         <Rel>Float</Rel>
         <Var>price</Var>
      </Atom>
   </And>
</Query>
```

Figure 4.6: Example typed query using unary predicates

Chapter 5

Object Identifiers (oids)

The final major feature in the Object-Oriented RuleML extensions is object identifiers (oids). An oid allows for the unique identification of a clause within a knowledge base, e.g. through the atoms of a rule. If the creator of the knowledge base knows the identification of the clause that will contain the relevant information, this can be specified as the oid of the goal within the query or rule body; this is comparable to sending a message (calling a method) to an object in a language like Java or SmallTalk.

If the oid for a goal is known and specified, the system will look up the clause by oid; since this process is quicker than a regular search, this can lead to faster reasoning when appropriate oids are known. In the case where an oid is not known a regular search will take place. While the system supports the use of complex terms as oids, fast searching is not available for clauses that use complex terms as oids.

The implementation of object identifiers (oids) within OO jDREW did not require any changes to the unification algorithm; the actual unification of oids is handled properly in the same manner as any other role. To support the quick look-up clauses by oid, and the enforcement of the uniqueness constraint, an hash-index was created on the oid of the clause. This allows for the quick retrieval of clauses when the oid of your goal is known, and for efficient enforcement of the uniqueness constraint.

The RuleML specification allows for URI anchoring and referencing via oids[4]; because of unresolved W3C issues in normalizing URIs this feature is currently not implemented in OO jDREW. As multiple URIs can reference the same web object, it is difficult to determine if two URIs should be considered to be the same for the purpose of unification. For example, on many http servers `index.html` is the default document for a directory; therefore `http://example.org/test`, `http://example.org/test/` and `http://example.org/test/index.html` may all reference the same resource, and should be considered the same. Unfortunately this is not always the case, as most http servers have the ability to redefine the default documents.

Chapter 6

Built-in Relations

In many cases a relation cannot be effectively and efficiently defined using a finite set of rules and facts; because of this situation many useful relations must be implemented as built-ins that are integrated into the reasoning engine. For example, it is not possible to efficiently express a numerical *greater than* relation using a finite set of rules and facts; therefore many declarative languages, including Prolog, have a *greater than* relation built into the reasoning engine.

There are two primary challenges when creating a system for built-in relations for a reasoning engine: first, the system should be designed in a way that the set of built-in relations is easily expandable without significant changes to the engine; second, a default set of built-ins, that is a subset of the SWRL built-ins, is provided with the system. The approach used for the first challenge takes advantage of the Object-Oriented nature of the Java programming language; this is detailed further in section 6.2. The approach used for the second challenge is based upon the SWRL built-ins proposal; this is detailed further in section 6.1.

6.1 SWRL Built-ins

The Semantic Web Rule Language (SWRL) proposal, which is a combination of the OWL DL and OWL Lite sub-languages with the Unary/Binary Datalog RuleML sub-languages, defines a modular set of built-ins for use within rule infrastructures for Semantic Web languages, services, and applications. As the SWRL proposal covers a large number of the relations that are likely to be implemented as built-in relations, and is gaining momentum within the Semantic Web community, it was chosen as the basis for a set of built-ins for the OO jDREW reasoning engine. All built-ins must be deterministic relations (currently Datalog), and can only be called either completely ground, in the case of comparison built-ins, or with a variable or a constant in the first position and all other arguments ground, as is the case for the math and string operations.

The SWRL built-ins document is broken into seven sections: comparisons; math; boolean values; strings; date, time and duration; URIs; and Lists [7]. The comparisons, math and strings sections of the SWRL built-ins proposal have been implemented in the current version of the OO jDREW engine. These sections were chosen for implementation and inclusion with the engine as it was felt that these would be the most commonly used, and are thus the most critical built-ins. While the comparison built-ins are implemented, they do not currently support the date or URI data types that are present in the SWRL proposal. A more complete subset of the SWRL built-ins is planned for future releases of the OO jDREW reasoning engine.

6.2 Built-ins Design and Implementation

One of the main goals in the design of a built-in system is to create an easily extensible system that is adaptable for both the bottom-up data driven and top-down goal driven versions of the OO jDREW system. To implement a new built-in relation the user must define a Java class that implements the `jdrew.oo.builtins.Builtin` interface. Defining a class to implement a built-in requires only basic knowledge about the data structures that are used in the engine, making it possible for users to implement new built-in relations without having detailed knowledge about the internals of the reasoning engine.

The `Builtin` interface contains two methods that must be implement for the built-in to function. The `int getSymbol()` method should return the integer code for the predicate symbol to be used for the built-in relation. The implementation class should ensure that the symbol is included in the symbol table by calling the `int internSymbol(String symbol)` method of the `SymbolTable` class, passing the predicate symbol; the result of the call to `internSymbol` is what should be returned by the `getSymbol()` method. The `performBuiltin(Term t)` method actually performs the steps required to interpret the built-in call, which is represented as a `Term` object, and create a clause (in the form of a `DefiniteClause` object) to unify with the built-in call goal; if the built-in call should succeed, the implementation should generate and return a clause to use for the resolution process; otherwise `null` should be returned.

Once the Java class defining the built-in has been created, the built-in must be registered with the reasoning engine; this registration is done in one of two ways. The first method to register a built-in is by calling the `registerBuiltin(Builtin)` method of the `ForwardReasoner` or `BackwardReasoner` object, passing an instance of the object that implements the built-in. The second method involves invoking the `registerBuiltin (Classname)` built-in in an instance of the top-down engine;

this method will dynamically load the built-in class and register it with the engine. This goal will succeed if the system is able to dynamically load and register the built-in, or if this built-in has already been successfully loaded; if the system is not able to successfully load or register the built-in, then this goal will fail. This second method is not supported by the bottom-up data driven version as the order of clause application is not guaranteed; therefore the system may need to invoke the built-in before the registration takes place.

One thing that should be noted about the design of the built-in system is that if a built-in relation with predicate symbol p is defined, no clauses where the head relation has a predicate symbol p will be invoked. Therefore you cannot define a built-in relation and a set of clauses with the same predicate symbol and have both of these used in the reasoning engine.

Chapter 7

Unification Algorithm

The key component of a Prolog-like reasoning engine is the unification algorithm; i.e. the algorithm that creates variable bindings and determines if two possibly non-ground terms match, and thus can be used for a complete resolution step.

The unification algorithm for a standard positional logic language is described with the following steps. First, the predicate symbols and arities must match; second, for each pair of predicate arguments one of three conditions must be met: if both are constants then the constants must be the same, if one is a constant and one is a variable then the variable is bound to the constant, or if both are variables then one variable is bound to the other. This unification algorithm must be changed considerably for unifying RuleML terms for three reasons: first, with the introduction of rest variables, terms with different arities can still unify; second, this algorithm does not consider the types of terms, and therefore cannot enforce type restrictions when matching; finally, it does not account for the non-positional nature of RuleML keyed arguments (slots).

Section 7.1 describes the unification algorithm used in OO jDREW that allows the use of these features of OO RuleML. The term objects in this chapter are assumed to have the following properties and operations:

symbol - The symbol code of the term. For a complex term this is the constructor symbol, for an atom this is the predicate symbol. Variables have a symbol code that is less than 0 while constants have a symbol code that is non-negative.

role - The role code of the term. Positional arguments have a role code of POSITIONAL, slotted rest arguments have a role code of SLOTREST, positional rest arguments have a role code of POSREST; all other arguments have a role code based upon the slot name.

type - The type of the term.

arity - The number of arguments in the argument list.

posrestindex - The index of the positional rest argument in the argument list.

slotrestindex - The index of the slotted rest argument in the argument list.

isSimple() - This returns true if the term is a simple term (individual constant or variable), false otherwise (atom, complex term or plex).

hasPosRest() - This returns true if the term has a positional rest argument.

hasSlotRest() - This returns true if the term has a slotted rest argument.

operator [**n**] - This returns the nth argument of the term after normalization.

The following operations are also available for use in the algorithm:

dereference(term) - Dereferences a term if possible; i.e. it resolves any variable bindings in the current environment.

bind(varterm, term) - Binds the variable represented by varterm to term.

Types.glb(type1, type2) - Finds the greatest lower bound (type intersection) of the two types; this should be unique in the current implementation.

Types.isa(type1, type2) - Determines if type1 is a type2; i.e. if type1 is equal to type2 or inherits from type2.

new Plex(arguments) - Creates a new plex term using arguments as the positional-slotted arguments of the plex.

7.1 OO jDREW Unification Algorithm

The following pseudo-code reflects the open-source java implementation of OO jDREW available at `http://www.jdrew.org/oojdrew/`.

```
Algorithm unify(t1, t2)
  Input: t1 - term object to unify - from head of clause
         t2 - term object to unify - from body of rule or query
  Output: returns true if the two terms unify

  t1 := dereference(t1)
  t2 := dereference(t2)
```

The first thing that must be done in the unification algorithm is to dereference the terms to resolve any variable bindings that have already taken place. Once this is completed the dereferenced terms are unified. This unification process can be one of three cases: two simple terms, one simple term and one non-simple term, and two non-simple terms. The first case is two simple terms.

```
if t1.isSimple() and t2.isSimple()
    if t1.symbol < 0 and t2.symbol < 0
        if t1 = t2
            { These are the same variables. Do not
                bind in order to avoid an infinite
                dereference loop }
            return true
        else
            type := Types.glb(t1.type, t2.type)
            newVar := copy(t2)
            newVar.type := type
            bind(t1, newVar)
            return true
```

This first block handles the case when both of the simple terms are variables. If these are the same variable no action takes place and *true* is returned; a binding is not created as this would cause an infinite dereference loop. If the two variables are different then the intersection of the types is calculated using the `Types.glb()` operation and the variable represented by *t1* is bound to a copy of *t2* with its type set to the result of the glb operation, then *true* is returned.

```
    else if t1.symbol < 0 and t2.symbol >= 0
        if Types.isa(t2.type, t1.type)
            bind(t1, t2)
            return true
        else
            return false
    else if t1.symbol >= 0 and t2.symbol < 0
        if Types.isa(t1.type, t2.type)
            bind(t2, t1)
            return true
        else
            return false
```

These two blocks handle the case where one of the simple terms is a variable and the other is a constant. First a type check is done using the `Types.isa()` operation; if the types are compatible, i.e. the type of the constant is or inherits from the type of the variable, then the variable is bound to the constant and *true* is returned; otherwise *false* is returned.

```
    else if t1.symbol >= 0 and t2.symbol >= 0
        if t1.symbol = t2.symbol
```

```
            and Types.isa(t1.type, t2.type)
               return true
         else
            return false
```

This block handles the case where both simple terms are constants. If the symbols match and the types are compatible then *true* is returned; otherwise *false* is returned.

```
   else if t1.isSimple() and not t2.isSimple()
      if t1.symbol < 0 and Types.isa(t2.type, t1.type)
         bind(t1, t2)
         return true
      else
         return false

   else if not t1.isSimple() and t2.isSimple()
      if t2.symbol < 0 and Types.isa(t1.type, t2.type)
         bind(t2, t1)
         return true
      else
         return false
```

These two blocks handle the case where one term is a simple term and one term is not simple. In order for unification to succeed in this case the simple term must be a variable and the types must be compatible. If the simple term is a variable, and the type of the non-simple term is the same as or inherits from the type of the variable then the variable is bound to the non-simple term and *true* is returned; otherwise *false* is returned.

The remaining sections of code deal with the case where both terms are not simple.

```
   else if not t1.isSimple() and not t2.isSimple()
       if t1.symbol != t2.symbol
          or not Types.isa(t1.type, t2.type)
             return false
```

When both terms are not simple the first steps are to ensure that the constructor/predicate symbols are equal and that the types are compatible; if this is not the case then *false* is returned.

```
       else
          t1slotrest := new empty list
          t2slotrest := new empty list
          t1posrest := new empty list
          t2posrest := new empty list

          i := 0
          j := 0
```

If the symbols are equal and the types are compatible, steps must be taken to ensure that the argument lists of each term unifies. The first thing in the process is to initialize variables that will be used throughout the remainder of the code. The lists (*t1slotrest, t2slotrest, t1posrest*, and *t2posrest*) are used to create the rest lists if appropriate; *i* and *j* are indices into the normalized argument lists, as described in Chapter 3 of *t1* and *t2* respectively.

The process below is repeated while i and j are less than the number of arguments for terms t1 and t2 respectively.

```
while i < t1.arity and j < t2.arity
    if t1[i].role = SLOTREST or t1[i].role = POSREST
        i := i + 1
        continue
    if t2[j].role = SLOTREST or t2[j].role = POSREST
        j := j + 1
        continue
```

Rest variables are handled separately at the end of this process; therefore in the two preceding blocks the current argument of *t1* or *t2* is skipped if it is a rest argument.

```
    if t1[i].role < t2[j].role
        if t1[i].role = POSITIONAL
           and t2.hasPosRest()
               t2posrest.append(t1[i])
               i := i + 1
        else if t1[i].role != POSITIONAL
           and t2.hasSlotRest()
               t2slotrest.append(t1[i])
               i := i + 1
        else
           return false

    else if t1[i].role > t2[j].role
        if t2[j].role = POSITIONAL
           and t1.hasPosRest()
               t1posrest.append(t2[j])
               j := j + 1
        else if t2[j].role != POSITIONAL
           and t1.hasSlotRest()
               t1slotrest.append(t2[j])
               j := j + 1
        else
           return false
```

The two preceding blocks deal with the case where a slot is in one term but not the other, or the case where one term has more positional arguments. If an appropriate rest term exists in the other term then the argument is added to the rest list and processing continues to the next argument, otherwise the unification process fails.

```
    else if t1[i].role = t2[j].role
        if not unify(t1[i], t2[j])
            return false
        else
            i := i + 1
            j := j + 1
```

This block handles the case where the slot, or positional argument, exists in both terms; if these two arguments do not unify then the overall unification fails and *false* is returned. If they do unify then we continue to the next argument in each term.

```
while i < t1.arity
    if t1[i].role = POSREST or t1[i].role = SLOTREST
        i := i + 1
    else if t1[i].role = POSITIONAL
        and t2.hasPosRest()
            t2posrest.append(t1[i])
            i := i + 1
    else if t1[i].role != POSITIONAL
        and t2.hasSlotRest()
            t2slotrest.append(t1[i])
            i := i + 1
    else
        return false

while j < t2.arity
    if t2[j].role = POSREST or t2[j].role = SLOTREST
        j := j + 1
    else if t2[j].role = POSITIONAL
        and t1.hasPosRest()
            t1posrest.append(t2[j])
            j := j + 1
    else if t2[j].role != POSITIONAL
        and t1.hasSlotRest()
            t1slotrest.append(t2[j])
            j := j + 1
    else
        return false
```

The two preceding blocks deal with the case where one term has more arguments than the other. The remaining arguments from the larger term are added to the appropriate rest lists; if no appropriate rest argument is present then unification fails.

```
if t1.hasPosRest()
```

```
    t1prestterm := new Plex( t1posrest )
    if not unify(t1[t1.posrestindex], t1prestterm)
       return false

 if t2.hasPosRest()
    t2prestterm := new Plex( t2posrest )
    if not unify(t2[t2.posrestindex], t2prestterm)
       return false
```

These blocks handle the positional rest arguments. A plex is generated from the positional arguments of the other term that were not used in previous steps of the unification process; this generated plex must unify with the positional rest argument for unification to succeed; if it does not, then *false* is returned.

```
 if t1.hasSlotRest()
    t1srestterm := new Plex( t1slotrest )
    if not unify(t1[t1.slotrestindex], t1srestterm)
       return false

 if t2.hasSlotRest()
    t2srestterm := new Plex( t2slotrest )
    if not unify(t2[t2.slotrestindex], t2srestterm)
       return false
```

These blocks handle the slotted rest arguments. A plex is generated using the slotted arguments of the other term that were not used in the previous steps of the unification process; this generated plex must unify with the slotted rest argument for unification to succeed; if it does not, then *false* is returned.

If the unification process did not fail at any of the previous steps then the unification of the two non-simple terms succeeds; therefore *true* is returned to the caller.

```
 {Unification of the two non-simple terms
  succeeds, return true}
 return true
```

Chapter 8

Future Work

In completing this project, two areas for improvement were identified. The first issue that was identified deals with the scalability of the reasoning engine to large knowledge bases. In the current system clauses are only indexed by the predicate symbol of one of the atomic formulas: the head atom in the case of the top-down version, and the head of facts and the first body atom for rules in the bottom-up version. By introducing an improved indexing system it should be possible to greatly increase the performance of the engine when dealing with large knowledge bases; this is discussed further in section 8.1. The second area of improvement that was identified was the term typing system. As discussed in section 4 the current type system only allows the modeling of taxonomic relationships between the types; it would be advantageous to be able to define domain and range restrictions for properties; this is discussed further in section 8.2.

8.1 Indexing of Clauses

In the current indexing system clauses are only indexed based upon the predicate symbol. Because of this the reasoning engine will attempt to unify its current goal with all clauses with the same predicate symbol; this situation leads to large amounts of wasted processing time when running the reasoning engine, as many of these clauses could be eliminated quickly using a more advanced indexing system.

Most relational database systems have the ability to index relational tuples in one or more predefined indexes, usually B+ trees. While an approach similar to this would provide an efficient indexing system for the indexing of clauses, it has one major disadvantage that makes it impractical for use in a reasoning engine for RuleML. In a relational database system the indexes are defined as part of the Data Definition Language (DDL) when declaring the structure of the database; RuleML knowledge bases do not have any predefined structure; therefore choosing which parameters to perform indexing on proves to be problematic, and it would be too costly to provide indexes for all parameters.

One indexing structure that has been used for Prolog-like reasoning engines, such as jDREW [12], is discrimination trees [11]. In Prolog, a predicate with an arity of n will never unify with a predicate with an arity of $m \neq n$. This property gives the ability to omit large sub-trees of the discrimination tree when looking for clauses that will unify with a goal. Unfortunately this structure cannot be used, as is, for indexing RuleML clauses: because of the ability to use rest variables (section 3.1) in RuleML clauses, a sub-tree of the discrimination tree cannot be omitted during the search because of a different arity or because a slot is missing, as clauses in those sub-trees may still unify with the goal being considered. By dividing the clauses to be indexed into four groups based upon the presence of rest variables (no rest variables, positional rest variable only, slotted rest variable only, and both slotted and positional rest variables), each with a specialized search algorithm, it may be possible to create an indexing system that will work efficiently in most common situations utilizing rest variables in RuleML.

8.2 Improvements to the Type System

The second area identified for possible improvement is the term typing system. In the current implementation users are able to model the taxonomic relationships between classes, but they are unable to model the domain and range restrictions of properties. While having the types included in the system is a significant advantage over using unary predicates for term typing, it would be advantageous to be able to define the domain and range restrictions for properties. Such an expanded type system would assist in creating knowledge bases that are consistent with the type definitions.

The proposed expansion to the type system would still use RDFS for the definition of the types, allowing the continued use of the lightweight taxonomies of the Semantic Web. In the enhanced version, RDFS properties would be used to define the slots, and their associated types, which would be required for a typing to be valid. The domain and range of simple types should be able to be defined based upon restrictions to the predefined base types, such as string, integer, decimal, etc.; the syntax for defining simple types could be based upon the XSD Part 2 type definition syntax. This would amount to introducing positional-slotted signature declarations to RuleML.

Chapter 9

Conclusions and Recommendations

The engine that was created as part of this project implements the features required to utilize the combine positional and object-oriented features of the RuleML specification, making it suitable for use in designing and running RuleML derivation rules. The developed engine is able to work in a data-driven bottom-up manner as well as a goal driven top-down manner, similar to most Prolog style systems. The system includes features that allow the use of combined positional and named arguments, user-defined order-sorted types and object identifiers; additionally, a set of built-in relations based upon the SWRL built-ins proposal is included with the engine. While the system includes support for symbolic object identifiers; URIs as object identifiers are not supported, primarily because of difficulties with normalization.

The current system only indexes clauses based upon the relation name of the indexed atom; the lack of an advanced indexing system may limit the scalability of the engine for very large knowledge bases. There is work in progress, separate from this honours project, on developing an advanced indexing system based upon discrimination trees[11]; once completed and integrated into OO jDREW this advanced indexing system should greatly increase the scalability of the engine when working with large knowledge bases.

Bibliography

[1] Marcel Ball, Harold Boley, David Hirtle, Jing Mei, and Bruce Spencer. Implementing RuleML Using Schemas, Translators, and Bidirectional Interpreters. W3C Workshop on Rule Languages for Interoperability Position Paper.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. The Scientific American, 2001.

[3] Harold Boley. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Semantic Web Working Symposium (SWWS'01)*, pages 381–401, July/August 2001.

[4] Harold Boley. Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms. In *Rules and Rule Markup Languages for the Semantic Web*, volume LNCS 2876, pages 1–16, October 2003.

[5] Harold Boley. POSL: An Integrated Positional-Slotted Language for Semantic Web Knowledge. http://www.ruleml.org/submission/ruleml-shortation.html, May 2004.

[6] Ulrich Furbach and Steffen Hölldobler. Equations, Order-Sortedness, and Inheritance in Logic Programming.

[7] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.daml.org/2004/04/swrl/, 30 April 2004.

[8] H. Huber and L. Varsek. Extended Prolog for order-sorted unification. In *The 4th IEEE Symposium on Logic Programming*, pages 34–45, 1987.

[9] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM*, 42:741–843, July 1995.

[10] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax (W3C Recommendation). http://www.w3.org/TR/rdf-concepts/, 10 February 2004.

[11] William McCune. Experiments with Discrimination-Tree Indexing and path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[12] Bruce Spencer. The Design of j-DREW: A Deductive Reasoning Engine for the Web. In *First CologNET Workshop on Component-Based Software Development and Implementation Technology for Computational Logic Systems*, volume CBD ITCLS 2002, pages 155–166, Madrid, Spain, September 2002.

[13] Lu Yang, Marcel Ball, Virendrakumar C. Bhavsar, and Harold Boley. Weighted Partonomy-Taxonomy Trees with Local Similarity Measures for Semantic Buyer-Seller Match-Making. In *Business Agents and the Semantic Web (BASeWEB) Workshop*, 2005.

Appendix A

Unary predicate type sort clauses

The unary-predicate rules that define the types from Figure 4.1 are given here.

```
<Implies>
   <Atom>
      <Rel>Van</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Vehicle</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
<Implies>
   <Atom>
      <Rel>PassengerVehicle</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Vehicle</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
```

```
<Implies>
   <Atom>
      <Rel>MiniVan</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Van</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
<Implies>
   <Atom>
      <Rel>MiniVan</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>PassengerVehicle</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
<Implies>
   <Atom>
      <Rel>Car</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>PassengerVehicle</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
<Implies>
   <Atom>
      <Rel>Sedan</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Car</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
```

```
<Implies>
   <Atom>
      <Rel>SportsCoupe</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Car</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
<Implies>
   <Atom>
      <Rel>StationWagon</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Car</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
<Implies>
   <Atom>
      <Rel>ToyotaCorolla</Rel>
      <Var>x</Var>
   </Atom>
   <Atom>
      <Rel>Sedan</Rel>
      <Var>x</Var>
   </Atom>
</Implies>
```