Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML

Andreas Eberhart

International University in Germany
Campus 2, 76646 Bruchsal, Germany
eberhart@i-u.de
http://www.i-u.de/schools/eberhart/

Abstract. This paper describes two approaches for automatically converting RDF Schema and RuleML sources into an inference engine and storage repository. Rather than using traditional inference systems, our solution bases on mainstream technologies like Java and relational database systems. While this necessarily imposes some restrictions, the ease of integration into an existing IT landscape is a major advantage. We present the conversion tools and their limitations. Furthermore, an extension to RuleML is proposed, that allows Java-enabled reaction rules, where calls to Java libraries can be performed upon a rule firing. This requires hosts to be Java-enabled when rules and code are moved across the web. However, the solution allows for great engineering flexibility.

1 Introduction

The Semantic Web is about to open up exciting new possibilities and enable a wide array of new applications on the web, that are able to process large amounts of machine readable data. Starting with the basic RDF and RDF Schema markup languages, more powerful and higher level languages like DAML+OIL and RuleML are being specified and tools supporting them being developed. Nevertheless, we feel that the adoption of these new technologies with a wide user community is still lagging behind. In a recent study [5] we found that it is extremely hard to find RDF data on the web today, unless considerable effort is invested in the search process. One can only speculate about the reasons for this, but we believe that it is crucial for the success of the Semantic Web to provide tools that integrate with proven and widely used technologies, namely object oriented languages like Java or C#, database management systems (DBMSs), and Web Services. It is clear that such systems cannot offer the full range of features provided by an XSB-based solution for instance. However, the example of the CiteSeer research index [2] shows that it is at least as important to scale as it is to provide complex reasoning and inference functionality.

In this paper we present the OntoJava [4] and OntoSQL cross compilers. OntoJava automatically converts RDF Schema and RuleML sources into a set of Java classes that comprise a main memory object database with a built in forward chaining rule engine. OntoSQL maps both recursive and non-recursive rules onto SQL-99 running on an IBM DB2 database server. Both software packages are open source and can be obtained from `http://www.i-u.de/schools/eberhart/ontojava/` and `http://www.i-u.de/schools/eberhart/ontosql/`.

The rest of the paper is organized as follows. The next section provides some background on RDF Schema, RuleML, and RDF query languages. Sections 3 and 4 then present the OntoJava and OntoSQL cross-compilers along with restrictions imposed by the respective mapping approach used. The remaining sections 5 and 6 describe future work, evaluate the approach, and summarize the paper.

2 Background

One of the main goals of the Semantic Web initiative is to promote sharing and reuse of semi-structured, machine-readable information on the web. A key to this strategy is the stack of mark-up languages consisting of RDF, RDF Schema,[1] and DAML+OIL.[2]. The recently started RuleML [1] initiative builds on this work, allowing rules to be encoded in a standard way. In the future it might even be possible to obtain the latest set of tax rules directly from the government.

Crucial pieces are engines that are able to store, handle, and process data encoded using these mark-up languages. A lot of work has been done in the area of storing and querying RDF data. The first RDF query languages like RDFDB mainly base on graph matching algorithms. This approach is augmented by features to also query the schema, as in RQL, and support for non-recursive rules in RDFQL. SquishQL is another variant being used in the RDQL and Inkling projects. Its syntax bases on SQL. The XSB deductive database is often used by Semantic Web related projects [3]. Apart from its query capabilities, XSB is also able to load facts from a relational database via an ODBC interface, allowing to easily access large amounts of data stored in commercial enterprise information systems.

Engines supporting RuleML are also available. A frequently used approach is to use stylesheets to transform the XML-based rules into the language of an inference system such as JESS. The TRIPLE language provides great flexibility by allowing the language semantics of RDF Schema, DAML+OIL, or others to be modeled in horn logic. Together with the query, these logic statements are then run against XSB [12]. A mapping of RuleML to the TRIPLE language is also possible. The RuleML initiative also joined forces with the Java Specification Request JSR-000094 to develop a Java rule engine API.[3]

[1] http://www.w3.org/RDF
[2] http://www.daml.org
[3] http://jcp.org/jsr/detail/94.jsp

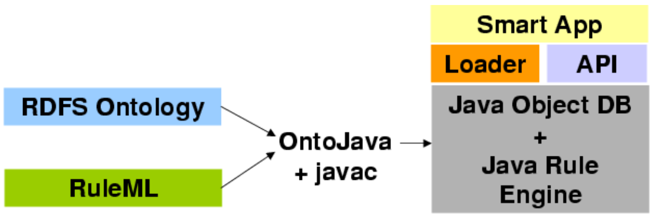


Fig. 1. The object database/inference engine generated by OntoJava can be loaded with RDF data and accessed by an application

3 OntoJava

The core idea behind OntoJava is that a directed, labeled RDF graph lends itself to being modeled using objects and object references of an object oriented programming language like Java. An object located in a main memory database represents every resource with its URI. We will examine to what extent the restrictions on the graph's arcs imposed by RDF Schema can be enforced by the language's type system and which restrictions are necessary. Apart form this discussion, we introduce the OntoJava cross compiler, that automatically converts RDF Schema, and RuleML into a set of Java Classes that act as a combined main memory object database with a built in forward chaining rule engine. Figure 1 illustrates this process and how a smart application can interface with the object database. An object-oriented language like Java offers many advantages and therefore the section concludes by proposing Java-enabled reaction rules and ways to customize the generated classes.

3.1 Mapping the Class Taxonomy

A class taxonomy, which can be expressed in RDF Schema, is the backbone of an ontology. Domain concepts are represented as classes with a hierarchy being imposed by the `subClassOf` property. Recent advances in the area of ontology mark-up languages resulted in the popular DAML+OIL approach [9], which bases on RDF Schema but is way more expressive. OntoJava works with RDF Schema, due to the availability of powerful editing tools such as Protégé [11]. Section 3.5 briefly covers how certain aspects beyond RDF Schema could be implemented by OntoJava. Consider the following example:

```
<rdfs:Class rdf:about="&pre;Person">
    <rdfs:subClassOf rdf:resource="&pre;Animal"/>
</rdfs:Class>
```

OntoJava maps every RDFS class into a Java class. The `subClassOf` property is similar to inheritance in object oriented systems with respect to both being transitive relationships defining the class hierarchy. However, the RDF Schema version is more flexible than its Java or C# counterpart since it allows multiple inheritance. OntoJava is therefore not able to handle multiple inheritance in the RDFS source at the moment. Section 3.6 outlines a simple solution for this limitation where RDF Schema classes are converted to interfaces which are then implemented. The RDF Schema example above is mapped to a class `Person` that inherits from `Animal`.

```
public class Person extends Animal
```

3.2 Mapping Properties

RDF Schema properties are defined with a domain and a range. Using the directed labeled graph metaphor, an arc's label identifies which property it refers to. The types of the resources connected by the arc must be the same class that's listed as the property's range or domain, or a subclass of it. The range is decisive for the type of the arc's source, the domain for its destination's type. Besides a class, the range of a property can also be a literal. Properties of classes can be defined as follows:

```
<rdf:Property rdf:about="&pre;isParentOf"
    <rdfs:domain rdf:resource="&pre;Person"/>
    <rdfs:range rdf:resource="&pre;Person"/>
</rdf:Property>

<rdf:Property rdf:about="&pre;name"
    <rdfs:domain rdf:resource="&pre;Person"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
```

Properties with a literal as the range are mapped directly into instance variables, which is quite straightforward. Note that it is easy to change this to get/set access methods. This becomes necessary if changes need to be tracked to support an undo operation or if multiple inheritance basing on interfaces is implemented. Relations to other instances are represented by a collection of references to other Java objects. This is the natural representation of a directed labeled graph in Java. The RDF Schema example above causes to following methods and variables to be defined in the class `Person`:

```
public class Person extends Animal {
    public String name;

    private HashSet isParentOf = new HashSet();
    public void putIsParentOf(Person p) {
        has.add(p);
    }
    public boolean getIsParentOf(Person p) {
        return has.contains(p);
    }
    public HashSet getAllIsParentOf() {
        return isParentOf;
    }
}
```

As sections 3.3 and 3.5 will explain further, the get/put access methods play an important role for the rule mechanism or when restrictions on the variables are to be checked. The corresponding get and put methods ensure that the appropriate data types, as defined in the RDF Schema property definition, are used. Note that the Java compiler enforces this. RDF Schema allows a property to have multiple domains, but only a single range. Multiple domains would result in several Java classes having variables and methods with the same name, which is no problem.

In terms of logic, the assertion $isParentOf(a, b)$ corresponds to the invocation of `a.putIsParentOf(b)`. The query $isParentOf(a, b)$ can be answered by calling `a.getIsParentOf(b)` whereas `a.getAllIsParentOf()` yields the answer to $isParentOf(a, X)$. Finally, $isParentOf(X, Y)$ is answered by iterating over all person instances and calling `getAllIsParentOf` again.

Ternary relations are not part of RDF and are handled by a workaround of using an intermediate pseudo resource. A similar scheme can be applied to OntoJava by promoting a relation from a simple object reference to having its own object, which can then point to more than two constituents.

3.3 Mapping Rules

The two basic rule evaluation strategies are forward and backward chaining. Obviously a imperative language environment lends itself to forward chaining, where the rules are executed in an if-then fashion. Rule implications are simply asserted into the fact base as new facts. The popular Java Expert System Shell (JESS)[4] is another representative for this evaluation style, implementing the RETE [7] algorithm to ensure efficient rule execution.

Restrictions on the rules' expressiveness, namely disallowing negation, make sure, that exactly one minimal model exists for the set of rules [6]. A minimal model describes the smallest possible fact base, where for any variable assignment on any rule body yielding true, the rule's head is also true. Therefore, if the minimal model is contained in the current fact base, no more rules will fire and assert new facts. The order in which the rules are evaluated does not matter, since the fact base cannot shrink. If stratified negation is allowed, rules containing the not operator must be executed last. This feature is not yet implemented in OntoJava.

In OntoJava, each rule from the RuleML base is converted into a static method. The brute force approach would be to check the right side for all possible bindings of the free variables for each rule until no new assertion occurs, which would be quite inefficient. OntoJava implements the following optimizations. Every time an update takes place on a specific property of an

[4] http://herzberg.ca.sandia.gov/jess/

object, all rules are evaluated, that contain that property in their right side, i.e. the rules that are potentially affected by the change. This means that the rules are checked incrementally. Consider the rule $isUncleOf(A, C) \leftarrow isBrotherOf(A, B) \wedge isParentOf(B, C)$ which has some label, say rule5. Further assume that this is the only rule, which has the `isBrotherOf` predicate on the right side. This causes the following call to be placed in the `putIsBrotherOf` access method in the person class:

```
public void putIsBrotherOf(Person p) {
    if (!(isBrotherOf.add(p)))
        return;
    Rule.rule5(this, p, null);
}
```

First, the new relationship is stored by inserting it into the set datastructure. If the object was already in there, the add method returns false and the call returns. Otherwise, the rule is activated. Since the rule has three variables, three parameters are passed. We also know that the added relation is the only change to the database. Therefore, A and B, the first and second variable, must be bound to `this`, the current object, and `p`, the object inserted into the datastructure. Only C must be bound to all persons. Here, we can use the free variables' type information. A free variable appearing with a `isParentOf` predicate can only be bound to persons. Thus, instead of iterating over all objects, we only need to iterate over persons. The following shows the rule's code:

```
public static void rule5(Person a, Person b, Person c) {
    if a==null, iterate a over all persons
    if b==null, iterate b over all persons
    if c==null, iterate c over all persons
    with combinations of (a, b, c) {
        if (a.getIsBrotherOf(b) && b.getIsParentOf(c))
            a.putIsUncleOf(c);
    }
}
```

Note that the call to `putIsUncleOf` can again trigger other rules that are activated in the respective put method.

Obviously this approach will not perform efficiently for rules with many free variables, since all but two of the combinations of free variables need to be tested by nested loops. We are planning another optimization here by performing short circuit evaluation of parts of the condition at the outer loops. This should drastically reduce the number of combinations that need to be checked.

Rules with large amounts of free variables being checked in nested loops resemble a relational join very much. This leads to section 4 where we examine how rules can be evaluated top-down by an SQL engine.

3.4 Property Inheritance

RDF Schema defines the core property `subPropertyOf` with the following semantics:

$$parent(a, b) \leftarrow subPropertyOf(father, parent) \wedge father(a, b) \tag{1}$$

This rule can be rewritten into a set of rules:

$$\{p(a, b) \leftarrow c(a, b) \mid subPropertyOf(c, p))\} \tag{2}$$

Rather than implementing special handling for subproperties in the generated code of the access methods, we decided to reuse the built-in rule mechanism and include a rule for each subproperty relation defined in the RDF Schema source according to equation 2.

The transitivity of `subPropertyOf` is handled by the fact that asserting the parent property through the rule will in turn trigger another rule asserting the grandparent property, and so on.

3.5 Constraints

RDF Schema currently has no mechanism for defining further constraints. However, it is clear that this would be a valuable addition. DAML+OIL, for instance, offers a construct like `daml:maxCardinality` to restrict the number of outgoing arcs from an object. Dealing with a constraint violation includes detecting the violation, notifying the user, and finally reversing changes made to the data repository. Constraints cannot be checked incrementally, since partially inserted data might reflect an inconsistent state. Once the user issues some sort of commit, the conditions must be checked. It is definitely possible to generate constraint checks into each class. Violations could be flagged using Java exception mechanism. To make sure the operations of a transaction can be reversed, each object might clone the internal datastructures and retain a copy of the old values until a successful commit is issued.

3.6 Multiple Inheritance

Unlike C++, Java does not allow a class to have more than one superclass. The reason lies in ambiguities in which implementation of an inherited method `m` is to be called, if both super classes implement `m`. Java solves this ambiguity by disallowing multiple inheritance for classes and offering multiple inheritance for interfaces only. Since an interface only contains the method signatures and not the implementations, it is always clear which method a caller refers to. Consequently, an Java interface instead of a class will be generated for each RDF Schema class. A class `StudentWorker` that is derived from both `Student` and `Worker`, results in the following Java interfaces:

```
public interface Student extends Person { ... }
public interface Worker extends Person { ... }
public interface StudentWorker extends Student, Worker { ... }
```

An implementation class is generated for each interface. Note that the complete implementations of `StudentImpl` and `WorkerImpl` need to be repeated in `StudentWorkerImpl`. This is not really a problem since the implementation is generated automatically anyway.

```
public class StudentWorkerImpl extends PersonImpl
                               implements StudentWorker {
    ... implement Student, Worker, and StudentWorker methods
}
```

Finally, the following code then creates an instance of the class:

```
    StudentWorker sw = new StudentWorkerImpl();
```

3.7 Defining Instances

RDF Schema defines the `type` property. A resource can be declared to be an instance of a class via a (ResourceURI, type, ClassName) triple. Unlike regular triples that are represented by storing a reference to the object in the respective data structure of the subject, the OntoJava framework handles this triple by instantiating an object of type ClassName:

```
ClassName obj = DB.createClassName(ResourceURI);
```

Thus, the triple is represented by the Java expression `obj instanceof ClassName` being true. Using a factory method offers further flexibility when custom behavior is to be added to the generated classes. Section 3.8 describes this mechanism in more detail. OntoJava also maintains global data structures allowing convenient access to the objects stored in the main memory database. Every object that is created by a factory method is immediately inserted in a global hash set using the resource's URI as the key allowing for efficient retrieval. Furthermore, a set data structure is maintained for each RDF Schema class to be able to quickly answer a query asking for all Person instances in the database.

Since the type predicate is treated in a special way, consequently, checking if a resource is of a specific type is not preformed by searching a type-predicate data structure but via the Java `instanceof` operator. The respective special handling for type predicates is built into the OntoJava software.

This behavior reveals a restriction of the OntoJava system. While in RDF, a class can be defined to be an instance of several classes, this is not possible in an object oriented programming environment. An instance can be viewed or cast to the interfaces and super classes related to the instance's class, but never to a completely unrelated class.

A solution for this limitation would be to implement the typing mechanism with own code instead of having it (partly) handled by the programming language environment. However, this would greatly complicate the code and make the application programming interfaces less descriptive, reversing major advantages of the solution. Furthermore, the Java compiler would no longer be able to catch RDF Schema violations at compile time. Since these points are important advantages of OntoJava, it seems reasonable to not opt for such a workaround and accept this limitation.

A workaround suggested for handling multi-class membership in classes $C_1, ..., C_n$ with Protégé is to create a new class C which inherits from $C_1, ..., C_n$.[5] An instance of this new class has the multi-class property, however, this would require creating, compiling, and loading new classes into the virtual machine at runtime.

3.8 Extending the Generated Classes

Extending the generated classes with own functionality can customize the system. As was mentioned in section 3.7, we use factory methods to create new objects in the database. Using the abstract factory design pattern [8], the user can replace the object factory with an own version that creates the customized classes instead of the original ones. The inference functionality remains since the new classes only extend the existing ones.

3.9 Namespaces

OntoJava's handling of namespaces is fairly rudimentary at the moment. One option, which was used in the examples shown so far, is to omit a certain namespace prefix from the RDF Schema source. Alternatively, the entire URI is used in class an method names. Here, we replace characters that cannot appear in variable names with an underscore. This results in very lengthy class and method names and makes the API quite unreadable. Nevertheless, this allows different RDF Schema sources to be combined into a single application.

3.10 Reaction Rules

Rather than asserting new facts, reaction rules perform an operation like sending mail or printing a message to the console. Usually the inference system defines a set of commands like `print` that can be used in reaction rules. Since we are dealing with a Java environment, it seems quite natural to allow Java statements to be embedded in reaction rule heads. The following example calls a web application, using a free variable as a parameter:

```
<_head><java>
    runtime.Loader.load("http://host/servlet/SearchGate?flight="
                        + <var>F</var>.name);
</java></_head>
```

The web application's RDF output is then loaded into the database using a runtime library. The implementation is quite simple. Instead of generating the call to the assertion method as shown in section 3.3, the code from the rule is printed. Only the variable references need to be replaced.

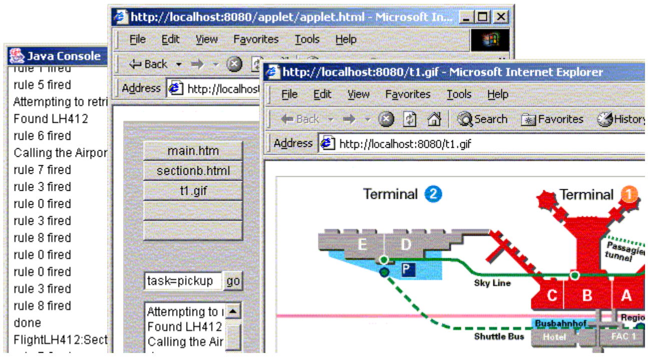
[5] http://smi-web.stanford.edu/projects/protege/protege-rdf/protege-rdf.html

This solution is flexible, but it also seems fairly proprietary. After all, rules are supposed to be exchanged across any platform and system. However, any kind of reaction rule command is proprietary. It seems natural to reuse an existing platform. Furthermore, applets apply the same concept. With Java being a cross-platform language and a virtual machine being installed on a large fraction of hosts, it seems fairly reasonable to load not just RuleML, but also some accompanying Java libraries for sending email etc.

3.11 A Sample Application

As shown in figure 1, any application can interface with the generated code. After an initialization call, the runtime data loader can be used to insert RDF triples from any URI into the database. The data loader is part of the runtime libraries that are provided. It is independent of the generated classes since it uses the Java reflection API. OntoJava does not implement any RDF query facility at the moment. Data can be obtained via global datastructures that can return all instances of a certain class. A reference to a stored object can also be obtained from its URI. The object graph can then be traversed using the get methods described in section 3.2. The put methods allow to application to assert further facts.

Figure 2 outlines a big advantage of OntoJava. The application as well as the generated engine is deployed as an applet guiding the user through the Frankfurt airport. In this example, the engine queries the flight information service via an HTTP request from a Java reaction rule. The data, which is returned in RDF format, is then loaded into the database triggering web pages being recommended. In our case, a map of terminal one is being displayed, because the user's flight is boarding there.


Java Console
http://localhost:8080/applet/applet.html - Microsoft In...
File Edit View Favorites Tools Help
Back
Address http://localhost
main.htm
sectionb.html
t1.gif
task=pickup
go
http://localhost:8080/t1.gif - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/t1.gif
Terminal 2
Terminal 1
E
D
P
Sky Line
C
B
A
Passagiertunnel
Busbahnhof
Regio
Shuttle Bus
Hotel
FAC 1

```
rule 5 fired
Attempting to retri
Found LH412
rule 6 fired
Calling the Airpor
rule 7 fired
rule 3 fired
rule 0 fired
rule 3 fired
rule 8 fired
rule 0 fired
rule 0 fired
rule 3 fired
rule 8 fired
done
FlightLH412:Sect
```

```
Attempting to r
Found LH412
Calling the Air
```

Fig. 2. An OntoJava enabled application. Rules trigger calls to the airport information system and suggest web pages to the user based on collected information.

The benefit of this application is quite clear. The development effort is minimal since a standard ontology about airports and the OntoJava generator can be used. The application is available at `http://www.i-u.de/schools/eberhart/smartguide/`.

4 OntoSQL

This section first compares the similarities and differences between the datalog-oriented RuleML version 0.8 and SQL. We pick up and elaborate ideas presented in [6] and the Edutella white paper [10], where a mapping of the RDF-QEL query language to SQL is discussed briefly.

The second part introduces the OntoSQL system, which is able to automatically generate the necessary tables and views in a relational database system, enabling it to act as a RuleML engine.

4.1 Mapping Datalog Queries to SQL

For this discussion we assume that the information is stored in a single table containing fact triples. We have three string columns, subject, predicate, and object, which are all part of the composite primary key, preventing the application to insert the same triple twice.

This storage schema is obviously very simplistic, yet sufficient for our analysis. Possible alternatives deal with models and optimize space requirements by introducing namespaces and URIs as entities in the schema.[6]

We distinguish between implication rules and queries. A rule $A \leftarrow B$ states that A_{var} is true if B_{var} is found to be true for a certain variable assignment. A query finds all variable assignments for which the search condition is true. We can generalize a query to be an implication rule with an empty left side: $\leftarrow B$.

We examine the mapping of queries first. Handling rules is complicated by the fact that the right side can depend on other rules. Section 4.2 explains how this is handled using SQL views.

Queries on a single predicate are mapped as follows:

```
isFatherOf(X, Y)

select * from fact where predicate = 'isFatherOf'
```

Conjunctions are translated to self-joins on the fact table. The join condition is determined by the occurrences of the variables in the query. Here, the object of the `isParentOf` triple must be the same resource as the subject of the `isBrotherOf` triple.

```
isParentOf(X, Y) and isBrotherOf(Y, Z)
```

[6] Sergey Melnik collected a list of proposals about ways of storing RDF data in relational databases at http://www-db.stanford.edu/~melnik/rdf/db.html

```
select * from fact a, fact b where
    a.object = b.subject and
    a.predicate = 'isParentOf' and
    b.predicate = 'isBrotherOf'
```

The SQL union operator can handle disjunctions. Note that generally, a disjunction like $A \leftarrow B\ or\ C$ can be written as two rules or queries $A \leftarrow B$ and $A \leftarrow C$, and vice versa.

```
isBrotherOf(X, Y) or isSisterOf(X, Y)

select * from fact where predicate = 'isBrotherOf'
union
select * from fact where predicate = 'isSisterOf'
```

4.2 Mapping Datalog Rules to SQL

As mentioned before, rules are similar to queries, except for the fact that implication results can influence other rules. Consider the following simple example consisting of two rules:

```
isSiblingOf(X, Y) <- isSisterOf(X, Y)
isRelatedTo(X, Y) <- isSiblingOf(X, Y)
```

The second rule depends on the first rule via the `isSiblingOf` property. To answer a query asking for all siblings, we could use the following SQL query that is embedded in a view:

```
create view isSiblingOf as
    select * from fact where predicate = 'isSiblingOf'
    union
    select subject, 'isSiblingOf', object from fact
        where predicate = 'isSisterOf'
```

The first subquery gets all `isSiblingOf` facts from the database. This is necessary since the user might assert a sibling relationship, if the gender of the sibling is not known. The second subquery processes the first rule. Except for the select clause, that explicitly states the predicate used on the rule's left side, the query corresponds directly to the datalog query `isSisterOf(Y, Z)`. Wrapping the query in a view allows us to treat the view's name `isSiblingOf` as a table. The DBMS internally resolves the underlying SQL statement.

If a query for all related resources is posed, the same mechanism can be applied. Note that the SQL query references the view defined above:

```
create view isRelatedTo as
    select * from fact where predicate = 'isRelatedTo'
    union
    select subject, 'isRelatedTo', object from isSiblingOf
```

Since even this simple example results in quite elaborate queries, it makes a lot of sense to encapsulate the queries retrieving every triple with a given predicate in a separate SQL view. These views can then be reused in other views or queries, as demonstrated in the last example.

Now the user can run the SQL query `select * from isRelatedTo`. The SQL engine of the DBMS handles all rules. While this is very convenient, we must rely on the DBMS's optimizer to efficiently handle the range of joins and union operations triggered by a simple query like the one above.

4.3 Recursive Rules

Recursive rules, i.e. rules where the predicate on the left side also appears on the right side, cannot be handled with the methodology presented above. In order to make sure that the rule set can be converted, a predicate dependency graph is established. It contains a node for each predicate. Whenever a predicate A appears in the body of a rule which has the predicate B in its head, we define B to be dependent on A and we draw an arc from A to B. The rule set can be converted if the dependency graph is free of cycles, with the exception of a cycle caused by a linear recursion of a predicate with itself. These cases can be handled by recursive queries, which are defined in the SQL-99 standard:

```
create view isAncestorOf as
    with rec(subject, 'isAncestorOf', object, level) AS (
        select * from fact where predicate = 'isAncestorOf'
        union all
        select subject, 'isAncestorOf', object from Parent
        union all
        select a.subject, 'isAncestorOf', b.object, level+1
            from rec a, Parent b
            where a.object = b.subject and level < 9
    )
    select * from ancestor;
```

This view corresponds to $isAncestorOf(A, C) \leftarrow isAncestorOf(A, B) \wedge isParentOf(B, C)$. The first and the second subquery get all existing ancestor information and combine it with the parent information. This forms the basis for the recursive third subquery. DB2 does not terminate the query, if the data contains a cycle. The `level` parameter registers the depth of the recursion and prevents an endless loop by restricting the search to a certain depth.

We model every predicate as an SQL view. From the dependency graph we can conclude, that, with the exception of the case above, no view definition will be recursive. This allows us to run SQL queries similar to the ones shown in section 4.1 from arbitrary database clients. The rules will then be executed transparently by the system.

Note that even the fact table might actually be a view of a regular ER schema. This way, up to date information can be used without any need to upload data from an enterprise information system.

4.4 Building Applications with OntoSQL

The OntoSQL system bases on the parsers used in OntoJava. Rather than generating Java classes, SQL scripts are created according to the methodology presented in the previous sections. An application program can then simply query the views using the generic DBMS SQL interfaces.

Since we found DB2[7] to be the only DBMS currently supporting SQL 99 recursive queries, OntoSQL contains the following workaround in order to support other DBMSs as well. Rather than a recursive query, a sequence of self-joins on the fact table is preformed and the results combined by the union operator. This yields the correct results if the longest transitive closure sequence is smaller than the maximum number of self joins performed on the fact table.

5 Future Work

In this paper we illustrated certain features such as multiple inheritance and constraints. We are currently working on their implementation. An interesting issue would be to exploit object relational features such as SQL 99 types and inheritance and replace the simple fact table. This would allow more efficient data access and checking of RDF Schema types within the database engine. The next step from there would be a closer integration of OntoJava and OntoSQL. Security and portability aspects of Java reaction rules are other important topics. Finally, we are planning on running performance tests with the solutions to test their scalability. We expect to hit a maximum rulebase complexity with OntoSQL, which is imposed by the DBMS's limitation on the maximum query complexity.

6 Summary

OntoJava focuses on the ease of use and integration. This brings some inherent shortcomings such as an object not being able to have several types or the type information being absolutely necessary to store an object at all. Compared to logic-based approaches, OntoJava is restricted to simple forward reasoning with all derived facts having to be stored. Finally, it is not possible to ask ad-hoc queries.

We get many benefits from using a mainstream technology like Java. There is an abundance of tools like IDEs and debuggers available. The javadoc tools for instance can automatically create a browsable documentation of the ontology. Java's reflection interface also enables us to easily inspect the ontology from an application or browse the state of the object database. Many applications today are written in Java. OntoJava allows for easy extension and integration of the ontology into the existing IT landscape. Finally, it is possible to customize the inference mechanism. We are experimenting with probabilistic reasoning for document retrieval.

[7] The personal developer edition of DB2 version 7.2 is available at the IBM website.

OntoSQL addresses the forward chaining limitation of the Java approach. It is also not limited to main memory and provides persistence. We currently do not support updating data through a convenient API. The facts must be inserted manually since the views are not updateable due to the frequent use of the union operator.

We feel that OntoJava and OntoSQL are useful tools when developing applications using Semantic Web technology. Their strengths are the ease of use and the great tools support by basing it on mainstream technology.

References

1. H. Boley, S. Tabet, and G. Wagner. Design rationale of RuleML: A markup language for Semantic Web rules. In *Semantic Web Working Symposium*, 2001.
2. K. Bollacker, S. Lawrence, and C. L. Giles. CiteSeer: An autonomous web agent for automatic retrieval and identification of interesting publications. In K. P. Sycara and M. Wooldridge, editors, *Proceedings of the Second International Conference on Autonomous Agents*, pages 116–123, New York, 1998. ACM Press.
3. D. Brickley and L. Miller. RDF, SQL and the Semantic Web - a case study. http://ilrt.org/discovery/2000/10/swsql/, November 2000.
4. A. Eberhart. OntoJava - applying mainstream technology to the Semantic Web. International Conference on Electronic Commerce, Vienna, Austria, Workshop on Semantic Web-based E-Commerce and Rules Markup Languages, 2001.
5. A. Eberhart. Survey of RDF data on the web. Technical report, International University in Germany, 2001. http://www.i-u.de/schools/eberhart/rdf/.
6. R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, chapter 24, pages 729–760. Addison-Wesley, second edition, 1992.
7. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37, 1982.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
9. J. Hendler and D. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):72–73, Nov./Dec. 2000.
10. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P networking infrastructure based on RDF, November 2001. http://edutella.jxta.org/reports/edutella-whitepaper.pdf.
11. N. F. Noy, M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, and M. A. Musen. Creating Semantic Web contents with Protege-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.
12. M. Sintek and S. Decker. TRIPLE - an RDF query, inference, and transformation language. In *Proceedings of the International Conference on Applications of Prolog*, Tokyo, Japan, October 2001.