

An Agent Infrastructure based on Semantic Web Standards

Andreas Eberhart

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

Abstract

This paper picks up the idea of completely specifying an agent using standard mark-up languages from the Semantic Web community, namely RDF, RDF Schema, and RuleML [1]. We describe an implementation using a cross-compiler that translates the mark-up to an SQL-based inference engine and knowledge base which is augmented with a set of Java modules. The agents communicate via standard Internet protocols. A sample application in area of document retrieval shows, that the agent framework supports collaborative behavior of an agent population basing on heterogeneous sets of rules.

Key words: Agent, Agent Infrastructure, Semantic Web, RuleML, RDF, RDF Schema, Document Retrieval

1 Introduction

Apart from its amazing growth, the Internet also evolves on a functional level. After being primarily a medium to publish static pages, we have experienced a phase, which can be characterized by the term Web Applications. Just about any service or good can be purchased or booked online. However, these sophisticated e-Commerce applications all require manual user interaction. The B2B and EDI communities have had some success in automating the exchange of data, but the orchestration of complex services amongst heterogeneous participants still remains a big problem. Collaborative agent techniques are thought to be a key technology in realizing the web's third phase, enabling more flexible and independent software interaction. Obviously this is a challenging task. A lot of systems have been developed in the AI community, but having these systems scale and interoperate on the web poses significant integration and software engineering tasks. The Semantic Web community addresses these issues by defining standard mark-up languages like RDF, RDF

Schema, DAML+OIL, and RuleML.¹ This stack of languages is a point of crystallization for the development of interoperable agent systems [2]. Generally, designing and implementing agent systems is a complicated task. Boley et al. address this issue and suggest that an agent can be constructed entirely using mark-up [1]. We pick up this idea, describe the implementation of an agent infrastructure basing on our OntoSQL framework, and finally apply the infrastructure in a small example that uses collaborative agents for document retrieval.

The rest of this paper is organized as follows. The next section describes the generic components of an agent and how these can be modeled using Semantic Web mark-up techniques. Sections 3 and 4 describe the framework and the application that uses it. The paper concludes with an outline of future work and a summary.

2 Background

We first want to restate the observations from [1] on how important components of an agent can be expressed. A class taxonomy represented in RDF Schema, along with properties defined for the classes can model a schema of an agent's mental state. The state itself is represented by an RDF graph. It comprises facts defined a priori as well as knowledge acquired through the perception system.

RuleML plays a pivotal role since it appears in three flavors. First, it can define a set of integrity constraints for excluding illegal mental states. Second, derivation rules specify the agent's terminological and heuristic knowledge. Last but not least, reaction rules define the agent's behavior in response to events and messages. Here, we introduce a further partitioning into action and reaction rules. An action rule has the following form: **IF condition THEN action**, with action being an operation like sending email, displaying a message, or sending a data to another agent. Reaction rules, define how to react upon incoming events or messages. Here, **condition** may include something like **reservation message received**.

Executing such an agent specification requires a data store, an inference engine operating on top of it, as well as a messaging system for incoming and outgoing communication.

¹ Refer to <http://www.w3.org/RDF/>, <http://www.daml.org/>, and <http://www.dfki.uni-kl.de/ruleml/> for the detailed specifications.

3 Applying the OntoSQL Framework

This section explains how the necessary components mentioned in the previous section are implemented.

Database and Inference Engine

We approach the implementation of this agent infrastructure by extending our previous work with the OntoSQL cross-compiler [3]. OntoSQL implements ideas from the field of deductive databases [4] by automatically compiling the Datalog-like RuleML version 0.8 derivation rules into SQL views on a set of fact base tables. These tables simply have two columns holding (subject, object). Since there is one fact table for each predicate, each tuple within a certain table reflects an RDF triple. Note that by leveraging IBM DB2's implementation of recursive SQL statements according to the SQL-99 standard, we are also able to cope with recursive rule definitions. Consider the following rule stating that a father is also a parent: $isParentOf(A, B) \leftarrow isFatherOf(A, B)$. This rule gets converted to the following view:

```
create view isParentOf as
  select subject, 'isParentOf', object from isParentOfFact
 union
  select subject, 'isParentOf', object from isFatherOfFact
```

For our agent infrastructure implementation we chose OntoSQL due to the ability of a database engine to perform backward chaining inference. This simplifies the system since derived facts are not explicitly stored, but rather computed on demand.

Integrity Constraints and Action Rules

The database with its SQL interface is augmented by Java components handling the integrity constraints and the action rules. These components again use OntoSQL's RuleML to SQL features. Figure 1 illustrates the system architecture. The integrity constraint component (IC), which is not implemented yet, and the action rule component (AR) are located in the update functionality. Upon changes to the database, first the constraints will be checked, rolling back the update transaction if a violation is detected. Consider the constraint that information on the mother must be present if the father is given: $\forall F, P \exists M : isMotherOf(M, P) \leftarrow isFatherOf(F, P)$. This constraint is violated if the following relation is not empty:

```
select * from isMotherOf where object not in
  (select * from isFatherOf)
```

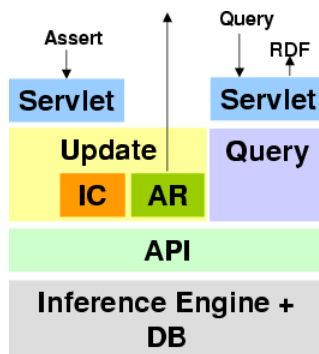


Fig. 1. The agent architecture. The deductive database exposes an API, which is used by the agent’s communication subsystem. Incoming HTTP messages (queries and information messages) are handled by servlets. Outgoing traffic is generated by action rules.

The SQL standard defines the CREATE ASSERTION command, which would be ideal for implementing integrity constraints, however, none of the major database vendors currently supports this feature. Therefore, we will check the constraints from the Java portion of our framework.

Once the update is completed successfully, the action rules are checked via the SQL interface by joining the views corresponding to the right side’s predicates. The resulting relation yields the variable assignments to be used for executing the Java statements in the rule heads.

Repeated Firing of Rules

One problem we faced in the action rule component was repeated firing of the rules. Every time the database is updated, the rules’ conditions are checked again, possibly re-triggering the action part. It is not obvious to determine, whether a rule fires because of the newly inserted data. The naive approach of re-executing all the actions leads to very undesirable behavior, for instance if the actions involve queries or messages to be sent out. We solved this problem by storing a history of which rule executed with what parameters. If a rule’s head $email(X, Y)$ was just activated with $X = joe@yahoo.com$ and $Y = text$, then the rule will only fire again for different assignments of X or Y .

This mechanism seems to be quite complicated, especially considering that SQL has a trigger functionality built-in. Triggers solve the problem of repeated rule firing by using the pseudo-tables inserted and deleted, which contain only the tuples changed by the current transaction. We opted against this approach, since it is not trivial to ensure the correct rule execution. Basically, the pseudo-tables would take the place of the fact tables in the views. However, syntactically a view cannot be defined on these pseudo-tables. For a more complicated rule set, it is therefore impossible to evaluate a trigger condition containing several predicates.

Reaction Rules and Message Handling

In our system, we distinguish between two basic types of messages: queries and information messages. An agent sends queries in a synchronous manner, in order to obtain data from another agent. Since most RDF parsers support reading data from URLs, it seemed natural to package the query inside an HTTP GET request. The answer obtained is then an RDF/XML document. An extension to SOAP and web services is planned and will be straightforward. We support very simple queries retrieving all outgoing arcs from an RDF resource (*subject, ?, ?*) or all outgoing arcs from an RDF resource with a specified label (*subject, predicate, ?*). A query sent to the agent at *host* could look as follows:

```
http://host/servlet/Query?subject=...&predicate=...&object=?
```

The RDF result is then added to the querying agent's fact base via the update interface. Figure 1 indicates that queries are usually launched from action rules. The following code shows an example of a Java action rule head (the corresponding body might be $\leftarrow isCustomerOf(Cust, Comp)$).

```
<java>
    runtime.Loader.load("http://infohost/servlet/Query?subject="
        + <var>Cust</var> + "&predicate=hasPreference&object=?");
</java>
```

The implementation of the query servlet only required reading the requested predicate view and formatting the result in RDF.

Information messages are used when an agent wants to tell another agent something. The request

```
http://host/servlet/Update?subject=...&predicate=...&object=...
```

causes the triple contained in the request to be inserted in the database of the agent at *host*. We use this mechanism as a workaround to implement reaction rules. Consider the following example:²

```
ON RECEIVE requestReservation(?CarGrp, ?Period) FROM ?Customer
IF hasCapacity(?CarGrp, ?Period)
THEN SEND askIf( blacklisted(?Customer)) TO Headquarter
```

Rather than implementing special handling for reaction rules, we temporarily insert the following facts into the database. Note that we use the intermediate resource *Reservation* to represent the ternary relationship between *Customer*,

² The example is taken from <http://tmitwww.tm.tue.nl/staff/gwagner/AORML/>

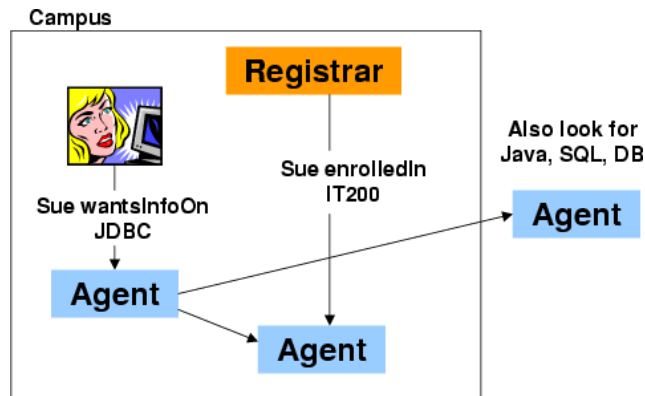


Fig. 2. Agents using different rule bases collaborating on a document retrieval task.

CarGrp, and *Period*:

$$\begin{aligned}
 &requestReservation(Customer, Reservation) \\
 &\quad hasCarGrp(Reservation, CarGrp) \\
 &\quad hasPeriod(Reservation, Period)
 \end{aligned}$$

The rule is rewritten by treating the ON RECEIVE part as a normal condition. If the entire condition is met, the rule fires which mimics the desired reaction rule behavior. Facts asserted by other agents are deleted again, once the respective action and reaction rules fired.

4 Sample Implementation

This section applies the agent infrastructure to the problem of document retrieval. An important aspect of how humans find information can be characterized by the term search context. If I know that Jim is the database guru in my company, he could probably point me to a good tutorial on JDBC. What is important here is that Jim also knows me. Thus he knows, for example, which level of difficulty would be appropriate. This context is lost, if an external hotline is called for help. However, an external service might have a larger knowledge base to work with.

This observation can be mapped onto an agent system. Every user would have his or her personal document retrieval agent. In addition, there are larger scale agents, similar to today's search engines. The agents share a basic ontology on users, user interests, documents, keywords, and synonyms.

The agents expose intelligence in the following ways: given a search query by the user, they decide which other agents to involve in the search. Agents also know where to obtain further information on the user that might be helpful to determine the search context. Finally, agents can reason about which

documents might be helpful.

Figure 2 illustrates a sample session. Sue works on the homework for the course IT200. In order to obtain information for solving a problem that came up, Sue provides her agent with a search string via the *wantsInfoOn* predicate. The corresponding triple is inserted into the personal agent's database, triggering action rules to send out information messages to other agents containing this data. A friend's agent is involved in the hope that the friend encountered the same problem and found a good reference. Positive feedback from Sue's friend might have cause her agent to adapt the rule base in order to recommend the reference next time around. An external agent at JavaGuru.com is also invoked. The friend's agent queries the university's registrar to obtain information on which courses Sue is enrolled in, hoping this will help in the decision making on which documents to recommend. The JavaGuru.com agent does not query the contextual knowledge, but it has a large body of domain knowledge built-in, allowing it to compute terms related to the original query about JDBC. Further action rules cause any recommendation to be sent back to Sue's agent, which displays the result.

This sample illustrates how queries and information messages triggered by action rules bring this agent ecosystem to life. The agents work in different environments, have different heuristics, and work with varying degrees of domain knowledge.

5 Future Work

We are currently refining issues revolving around the automatic conversion of integrity constraints and action rules in the framework. Apart from these engineering tasks we plan on implementing a layer on top of the current infrastructure that would allow for the rules to be fine tuned according to information gathered from user feedback and a reward-based system for instance. We found such features virtually impossible to implement using basic rules. However, a layered approach with upper layers adapting the rules according to higher-level decisions seems to be much cleaner and more natural anyway.

Other important aspects are security and trust amongst the agents. Currently, a malicious agent can do a lot of harm. An agent must decide whether it wants to disclose possibly sensitive information to other agents through the query interface.

6 Summary

We presented the implementation of an agent framework that allows to completely specify agents via the RDF, RDF Schema, and RuleML mark-up languages. This is an extremely promising approach since it relieves programmers from many of the burdens that are usually inherent with the implementation of agent systems. It also makes it easier to integrate agents written by different teams. Agreeing on a common RDF Schema and proper action and reaction rules that enable collaboration would be prerequisites for that. The agent middleware bases on a simple HTTP requests. Therefore, it should be a simple task to interface this solution to KQML-based agent platforms for instance.

Acknowledgements

We want to thank Gerd Wagner at the Eindhoven University of Technology for his valuable comments and suggestions.

References

- [1] H. Boley, S. Tabet, G. Wagner, Design rationale of RuleML: A markup language for Semantic Web rules, in: *Semantic Web Working Symposium*, 2001.
- [2] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, *Scientific American* (2001) 28–37.
- [3] A. Eberhart, Automatic generation of Java/SQL based inference engines from RDF Schema and RuleML, in: *Proc. of the International Semantic Web Conference 2002, Sardinia, 2002*.
- [4] R. A. Elmasri, S. B. Navathe, *Fundamentals of Database Systems*, 2nd Edition, Addison-Wesley, 1992, Ch. 24, pp. 729–760.