

# OntoAgent: A Platform for the Declarative Specification of Agents

Andreas Eberhart

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

**Abstract.** This paper presents a detailed description of our OntoAgent framework. It allows a software agent to be specified entirely using standard mark-up languages from the Semantic Web community, namely RDF, RDF Schema, and RuleML [2]. The basic agent components are identified and their implementation using relational databases and Java technology is described. The agents communicate via standard Internet protocols. Furthermore, we suggest action rules that allow agents to become active spontaneously, as well as a framework for interfacing the agents with intelligent applications that can perform learning algorithms or other complex tasks.

## 1 Introduction

In the recent years we have witnessed the tremendous success of browser-based web applications. A vast array of goods and services can be bought or reserved online. Most of these offerings require manual interaction though. Therefore, a lot of work has been done in trying to automate processes across the Internet. The B2B community had some success in defining message and API standards for information exchange. However, the general consensus seems to be that agent technology will ultimately allow previously unseen levels of flexibility in the interaction and ad-hoc configuration of a swarm of agents. A lot of research has been done in this area; however, agent technology has yet to achieve a major impact on today's Internet. Recently, the Semantic Web initiative has gotten a lot of attention. The idea is that with the definition of a stack of standard mark-up languages such as RDF, RDF Schema, DAML+OIL, and RuleML, software agents will be able to collaborate on a large scale without having to be dependent on a set of predefined interfaces. Common shared ontologies are thought to be the crucial centerpiece of this vision by formally conceptualizing the agents' application domain [1]. Boley et al. extend this idea and suggest that an agent can be constructed entirely using mark-up [2], allowing a declarative specification of software agents. We believe that this approach would simplify the development and adoption of agents tremendously. In this paper, we present OntoAgent, a platform for the specification of agents using RDF, RDF Schema, and RuleML.

Our initial implementation of OntoAgent described in [3] is extended with a detailed classification of rule types and their implementation, an extensive overview of the architecture and our design choices, as well as a description of possible future work on integrity constraints and intelligent agents.

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. Section 3 describes the individual modules of the OntoAgent framework in detail. This section ends by introducing a possible extension allowing the system to learn by having an intelligent application interface with the agent and adapt the agent rules as a result of learning progress. The paper concludes with an outline of future work and a summary.

## 2 Background

In the recent years, several definitions for the term agent have been given. Franklin and Graesser provide a nice overview of different definitions in [6]. They define an autonomous agent as "... a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future".

### 2.1 A Generic Agent Architecture

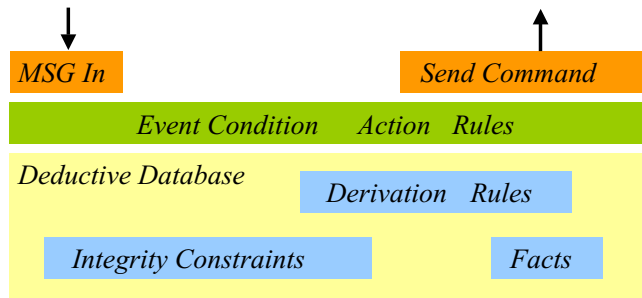
Boley et al.'s definition refines the statement above by identifying the five major components illustrated in figure 1. The following paragraphs explain each component's function as well as how it can be represented using Semantic Web mark-up languages.<sup>1</sup>

*Mental State* Every agent has a mental state, which is a set of facts it believes to be true. While there are a large variety of knowledge representation techniques available, simple directed labeled graphs have become the method of choice in the Semantic Web community. Objects or resources, represented by graph nodes, are interconnected via labeled relationships. The Resource Description Framework (RDF) can be used to serialize such graphs.

*Schema for the Mental State* Shared Ontologies are thought to be the key enabling technology for agent interoperation. A class taxonomy along with properties defined for the classes are the basic components of such an ontology. Therefore, ontology representation languages like RDF Schema and DAML+OIL provide syntax for the definition of classes and properties and they can be used to model a schema of an agent's mental state. Before agents can successfully exchange and understand each other's messages, they should agree on a common ontology to use. Otherwise it is not guaranteed that the data is interpreted according to the shared domain conceptualization.

---

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



**Fig. 1.** Components of a generic agent.

*Integrity constraints* Rules play a pivotal role as they appear in the following three agent components. Integrity constraints such as **IF condition-not-fulfilled THEN error** can be viewed as logic statements used to exclude illegal mental states. The RuleML initiative is currently planning to include integrity constraints in one of the next versions. Currently, they appear for example in SQL (check, create assertion, referential integrity, etc.) or UML's Object Constraint Language (OCL).

*Derivation rules* The classical form of rules are derivation rules such as **IF condition THEN conclusion**. They specify the agent's terminological and heuristic knowledge and allow deriving new information from the basic set of facts known to the agent.

*Reaction rules* Reaction rules define the agent's behavior in response to events and messages. Reaction rules are often referred to as Event Condition Action (ECA) rules and have the following form: **UPON message RECEIVED: IF condition THEN action**. While derivation rules influence the agent's reasoning by establishing conclusions, reaction rules can trigger actions such as sending email, printing a message, or sending a message to another agent.

## 2.2 Rationale for Rule Extensions

In addition to these three rule types that are used to model these basic agent building blocks, we introduce the following two categories.

*Queries* In many applications it is desirable for some sort of intelligent application to query an agent's mental state. We therefore consider queries, which can be viewed as a derivation rule without rule head. Rather than deriving conclusions, a query yields the variable assignments for which the query's condition is true: **IF condition THEN yield-result**. These variable assignments are then returned to the caller.

*Action Rules* Finally, we consider the situation where an agent not only reacts to external events, but can also act spontaneously. Therefore, we define action rules to be a special case of reaction rules where only the condition is necessary to trigger the action part. Table 1 summarizes the three basic rule types and our two additional rule classes.

The main argument for our extension comes from the question of how the message flow between agents is initiated. Our standpoint is that an agent should also be able to actively examine, i.e. query, its environment. In certain situations, these queries should be activated without an external stimulus. We think that agents will often try to obtain information from a legacy system via an RDF wrapper interface. The alternative standpoint is a more workflow-oriented view. Such a scenario might have sensors outside the agent that actively inform agents via the proposed event interface. Even though it makes the implementation more complicated, we choose the first approach in order to solve the message initiation problem.

Rule Type	Description	Invokation	Example
Derivation Rules	Define derived concepts on top of base concepts	Other rules	IF condition THEN conclusion
Action Rules	Like derivation rules but can contain commands like send email, print message, or assert new fact in the rule head	After update	IF condition THEN action
Reaction Rules (ECA)	Like action rules but in addition to a condition, an external event or message is needed for the rule to fire	Incoming message	UPON message RECEIVED: IF condition THEN action
Queries	Obtain base data and derived data from derivation rules	Application	IF condition THEN yield-result
Integrity Constraints	Make sure that the agent's internal state is legal with respect to the application domain	After update	IF condition-not-fulfilled THEN error

**Table 1.** OntoAgent rule types

### 2.3 Rule Execution

The rule classes expose important differences with respect to their execution behavior. Queries are initiated by an external component like an application or another agent. The local agent then services this request. The search condition is evaluated against the base facts and the derived facts. Therefore, derivation rules will be triggered if the search condition tests predicates that appear in

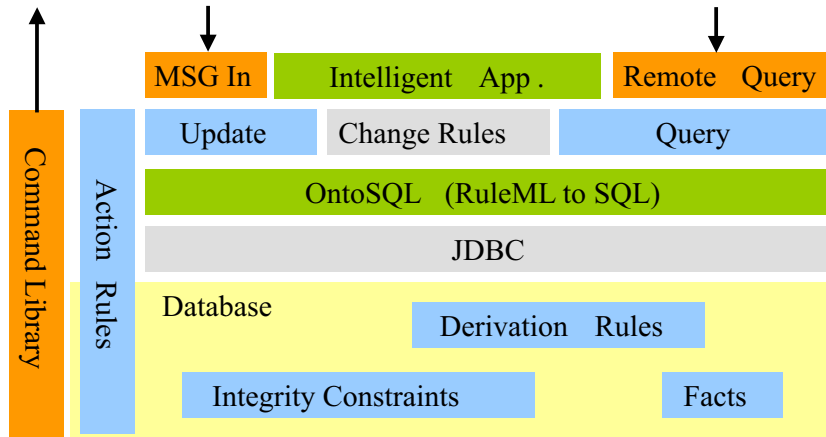


Fig. 2. Components of an agent running on the OntoAgent platform.

the rule heads of derivation rules. All these operations are read only due to the backward-chaining nature of our system. Action rules and integrity constraints, however, react to changes to the base facts. Integrity constraints need to be checked after each update operation in order to make sure the new state is consistent. Similarly, action rules can only be activated after an update. If this were not the case, they would fire permanently. Thus, both types of rules can be activated by a one-time event of a fact being inserted. ECA rules also react to such an event. Rather than an update, an incoming message or event is the deciding trigger for these kinds of rules.

These observations play an important role for the implementation of our OntoAgent platform.

### 3 Implementation of the Agent Framework

There are several rule engines available today, many of them even supporting RuleML. SweetRules [7] and the TRIPLE system [8] are arguably the most prominent among those. We chose to base the OntoAgent platform mainly on our OntoSQL platform. OntoSQL's main advantage is the fact that it bases on mainstream relational database technology. 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. This section will pick up the generic agent components mentioned in the previous section and describe the underlying major design decisions as well as the implementation basing on the OntoSQL [4] system.<sup>2</sup>

<sup>2</sup> The OntoAgent and OntoSQL software is available at <http://www.iu.de/schools/eberhart/>

### 3.1 Deductive Database

As illustrated in Figure 2, the fact base along with the derivation rules build the agent's foundation. Derivation rule engines are often referred to as deductive databases. This section briefly describes how a relational database server can function like a deductive database.

*Storing the Facts* We chose a very straightforward approach for storing the RDF graph representing the agent's mental state. A table is created for every predicate or graph label found. OntoSQL can read this information directly from the RDF Schema file specifying the ontology to be mapped.

```
create table [PredicateName]Fact
(
    subject varchar(255),
    object  varchar(255),
                                primary key(subject, object)
)
```

A tuple (S,O) in the table PFact then represents the RDF triple (S,P,O). Note that the choice of the composite primary key avoids duplicate entries in the respective predicate tables. Every subject and object refers to a resource. The resource's type is stored in the `typeFact` table, which is implicitly created. We do not require referential integrity of subjects and objects to subjects in the `typeFact` table since a resource with no type information is treated as the most general type RDF-Resource.

*Derivation Rules* Using relational databases as deduction engines is a well-established practice in the database community. OntoSQL implements the ideas presented in [5]. Every predicate is converted into an SQL view as follows. Assume `rule-1` through `rule-n` contain the predicate in their rule heads:

```
create view [PredicateName] as
    select subject, '[PredicateName]', object
        from [PredicateName]Fact
    union
    select clause for {\tt rule-1}
    union
    ...
    union
    select clause for {\tt rule-n}
```

The first query selects the known base facts from the respective table. These definitely must be included in the result. The remaining components of the overall union query can be obtained from the derivation rules. The individual rules are translated to SQL queries as outlined in the following example. Consider the rule  $a(X, Y) \leftarrow b(X, Y) \wedge c(Y, \text{"const"})$ . The conjunction is translated into

the equi-join of  $b$  and  $c$  on the variable  $Y$ . In turn, the columns to be selected are determined by the position of the variables in the rule head and body. Further conditions such as  $c$ 's object having to be equal to “*const*” appear as additional conditions in the query's where clause:

```
select b.subject, '[PredicateName]', b.object
from b, c
where b.object = c.subject and
      c.object = "const"
```

Note that  $b$  and  $c$  are again views, not the fact tables. This causes them to be executed if the select statement above is evaluated. Obviously recursive rule definitions will clash with a relational database's evaluation algorithm and an error will be raised when the view is to be created. OntoSQL provides two solutions for this. IBM's new version of DB2 supports SQL99 recursive query definitions allowing a view to appear in its own definition. OntoSQL also supports other databases by simply computing the union of a series of self joins on the table. This works fine if the recursion depth is smaller than the maximum number of times the table is self-joined. Unfortunately, this number depends on the current state of the fact base and might therefore exceed the fixed number of self-joins during runtime. Consequently, this workaround should be avoided if possible.

*Integrity Constraints* An important means to keep the database clean and in a correct state are the so-called integrity constraints. It is desirable to perform as many checks directly inside the database as possible, rather than pushing this duty up to an application. The SQL check mechanism allows restricting the range of a certain attribute. Referential integrity and uniqueness constraints are usually used for clean modeling of database schemas. However, the traditional database integrity constraint mechanisms are not really applicable in our case since the schema used is very generic. It can essentially be reduced to a single subject, predicate, object table. This approach has clear advantages in dealing with semi-structured data; however, it requires more effort to be put on the definition of constraints.

The create assertion construct would solve this problem, since it allows a constraint to be defined as a condition including several tables. In contrast, the check construct only allows referring to other attributes of the same table. Unfortunately none of the major database vendors currently implements this feature.

We propose to use SQL triggers in this case. Consider the following example of a trigger ensuring that the domain of the predicate *fatherOf* is the class of all males.

```
create trigger fatherOfSignature
on fatherOfFact, typeFact
for insert, update, delete
```

```

as
  if exists (
    select * from fatherOf where subject not in
      (select subject from type where object = 'Male')
  )
  begin
    raiserror ('Fathers must be Male', 16, 1)
    rollback transaction
  end

```

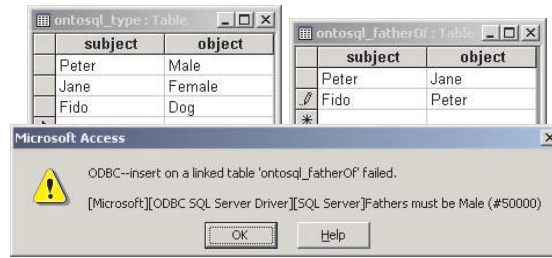
Triggers basically consist of an SQL statement that is executed upon an update. A trigger must be defined for a certain table. It seems natural to define this trigger on the `fatherOfFact` and `typeFact` tables since their views appear in the condition's SQL statement. However, other predicates such as `isParentOf` can imply fatherhood via a derivation rule. The respective fact tables, `isParentOfFact` in this case, would therefore also need to have this trigger defined on them. The trigger syntax allows using the pseudo tables deleted and inserted in the body. These tables have exactly the same structure as the base table, but only contain the deleted or the changed and inserted tuples of the table. This feature can be used to do incremental checks only on the tuples affected in the current transaction.

The question arises, whether we can use these pseudotables rather than re-computing the entire views again. While this is possible for simple cases, for example if the predicate does not appear in any derivation rule, the general case seems to get quite complex. One would have to rewrite the queries described in the previous section by replacing `[PredicateName]` with `inserted`. Furthermore, the delete case would require different logic. If the triple  $(Pat, type, Male)$  is deleted from the type base facts, the integrity constraint is only violated if Pat is a father and there are no other facts, such as  $(Pat, type, TallMale)$ , that allow us to conclude that Pat is male.

In case of a constraint violation, the trigger causes the current transaction to be rolled back, undoing all changes that lead to the violation. Figure 3 shows an example of the above trigger in action. It shows Microsoft Access which is used as a graphical front-end for SQL Server. The "Fathers must be Male" error shows up since Fido the dog cannot be Peter's father.

The RuleML initiative is currently working on extending their rule language to include syntax for constraints as well.

Currently, OntoAgent provides no support for the specification of integrity constraints. This quite non-trivial task is entirely left up to the application designer. At least applying the brute-force strategy of defining the trigger for all tables can solve the problem of deciding which tables need to be associated with a certain trigger. Another complication is that constraints often require existence and forall quantification. The example above could be read as: raise an error if there exists a father who is not in the set of male persons.



**Fig. 3.** Triggers can enforce integrity constraints at the database level. The dog Fido cannot be Peter’s father and the transaction is rolled back.

### 3.2 Agent Actions

The previous section laid the agent’s foundation by providing the fact store, inferencing, and the capability to exclude illegal mental states. This section will now explain how agent’s can perform actions in order to interact with their environment.

*Command Library* We chose to implement the command library in Java. Java offers a rich array of build-in functionality such as threading capabilities and a large selection of abstract data types. Furthermore, an extensive variety of external libraries are available as Java archives for sending email or SOAP RPC functionality. Table 2 shows the most important commands that can be triggered from rules.

<code>print(X)</code>	prints X to the console
<code>assert(triple)</code>	permanently asserts new fact
<code>delete(triple)</code>	permanently deletes fact
<code>email(X,Y)</code>	sends X the email with text Y
<code>load(URL)</code>	loads RDF from URL and asserts the triples
<code>send(X, triple-1, . . . , triple-n)</code>	sends X a message consisting of triples

**Table 2.** OntoAgent command library

The send, email and load commands are executed within their own thread, since these operations can potentially take a long time and could stall the engine’s execution. The load command allows the agent to interface with any RDF-enabled information source. This could be an enterprise information system as well as another agent’s remote query interface.

Asserting new facts has to be used with care, since a potentially only temporarily valid agent state can cause facts to be asserted permanently. Nevertheless, we believe that the assert and delete functionality is very important.

Assume an agent wants to maintain a history of important events in order to use them for making further decisions. Data about the events could be stored and deleted by action rules. The decision-making rules could be designed with the temporary nature of the data in mind. For example, these rules could have a heuristic character.

*Action Rules* As shown in table 1, action rules, like integrity constraints, get evaluated upon updates to the fact base. Again, triggers seem to be the natural choice, especially since both the Oracle and IBM database servers support Java stored procedures. This would make it extremely convenient to combine Java's flexibility with the reuse of an existing trigger mechanism. However, there is a complication. Integrity constraints must hold all the time. Therefore, it does not matter if they are evaluated when it is not really necessary, i.e. due to the problems we described with the triggers' incremental strategy.

Action rules, however, must only fire, if a variable binding makes the condition true when it was not beforehand. The following example illustrates the issue: If all parents receive a congratulation email, we only want them to get the email if their child was just born. If we'd simply check the condition, every parent would get the mail after any child was born.

Due to this problem we opted for a solution where action rules are triggered from outside the database. After each update, the action rules' conditions are checked via the JDBC interface. The resulting tables containing the variable assignments are stored in a Java abstract data type, along with the command they trigger. This information is then compared to the previous state and calling the appropriate methods in the command library performs all new actions:

```
set currentState = EMPTY
forever
  upon update:
    check action rule conditions
    store results in variable newState
    for each action in newState and action not in currentState
      perform action
    currentState = newstate
next
```

### 3.3 Communication Subsystem

In our system, we distinguish between two basic types of messages: queries and information messages. Figure 2 shows that the message input and remote query components handle the respective incoming events and messages, whereas outgoing messages and queries originate from the command library. This section describes the structure of the messages, their effect on the agent, as well as the implementation basing on the modules previously introduced.

*Queries from Remote Agents* An agent sends queries in a synchronous manner,<sup>3</sup> 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. This simple mechanism can easily be replaced by using SOAP middleware. We are working on an implementation using the axis web service engine<sup>4</sup> in conjunction with the tomcat web server. The answer obtained is then an RDF/XML document. 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 2 shows that messages or queries from the agent are initiated from the command library which is in turn activated by the action rule component. The following action rule causes the agent to query some information `host` for a customer's preferences, which are then also asserted into the local database:

```
queryAndAssert("http://infohost/servlet/", Cust, "hasPreference", "?")
← isCustomerOf(Cust, Comp)
```

The implementation of the query servlet only requires reading the requested predicate view and formatting the result in RDF. Future versions of OntoAgent might incorporate an RDF query language or the recently published RuleML query specification, in order to allow for more flexibility in the queries.

*Reaction (ECA) Rules* Obviously reaction rules are quite similar to action rules. Consider the following example:<sup>5</sup>

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

The first challenge is to correctly associate an incoming message with a certain ECA rule. The condition will again be computed via an SQL view. Therefore, the second task is to pass the incoming values into the view. One approach would be to use triggers again. Triggers themselves are ECA rules. A table `MsgIn` would be created for incoming messages. For the example above, the message handler stores a tuple (`requestReservation, CarGrp, Period, Customer`). A trigger reacts on inserts into this table. We can pass the tuple values into the SQL condition by joining the message table with the other predicates:

<sup>3</sup> Synchronous meaning that the calling thread is blocked. Note that, as described in section 3.2 the caller specifically starts a new thread to be able to resume its operation.

<sup>4</sup> <http://xml.apache.org/axis/>

<sup>5</sup> The example is taken from <http://tmitwww.tn.tue.nl/staff/gwagner/AORML/>

```

create trigger HandleRequestReservation
on MsgIn
for insert
as
  for all tuples in (
    select hc.sender from hasCapacity hc, inserted i
    where i.msgType = 'requestReservation'
    and hc.subject = i.par1
    and hc.object = i.par2
  )
  call send(askIfBlacklisted, hc.sender, Headquarter)
  remove message

```

Compared to the action rule case, this trigger only needs to be defined to react upon inserts in the message table, since only an incoming message can trigger an action. Nevertheless, we decided against this approach for the following reasons. First and foremost, this approach requires quite a lot of additional functionality in OntoSQL. Secondly, triggers and trigger actions tend to be fairly dependent on the implementation of the database server. It would be quite hard to support the major vendors. Finally the following alternative turns out to nicely leverage our action rule functionality. We treat the predicates that appear in the message as regular RDF Schema predicates. The handler for the incoming messages temporarily inserts the contents of the message into the database. Note that we use the intermediate resource *Reservation* to represent the ternary relationship between *Customer*, *CarGrp*, and *Period*:

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

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. After the insert, which triggers the ECA rules (which actually become action rules), the message facts are deleted again:

```

receive message M(p1, p2, ..., pn)
insert parameters (p1, p2, ..., pn) into fact base
  (this can trigger certain actions of ECA rules)
remove parameters (p1, p2, ..., pn)

```

### 3.4 Intelligent Application

Agents become intelligent agents if they are able to learn. Since machine learning techniques usually base on a wide variety of computational algorithms, it seems awkward to try and implement rule-based learning algorithms. We propose a different setting. Figure 2 shows an intelligent application component that interfaces with the basic agent framework. In such as layered architecture,

the learning algorithms can be implemented in any language or system using traditional techniques. The base data, however, can come from the deductive database via the query interface. The intelligent application can influence the basic framework in two ways. It can assert and delete base facts altering the inference results and ultimately the agent's behavior. More importantly, though, the rules themselves can be modified over time. Since we operate on top of a relational database, this would only require drop/create trigger/view statements that can even be performed while the agent is running.

We believe that the declarative specification is a great tool to let researchers and developers focus on the learning and behavioral aspects of agent technology located in the top layer of the architecture shown in figure 2. We currently using OntoAgent in the development of a collaborative agent system for document retrieval [3]. The idea is that a community of users all shares their personal collection of links to relevant online reading material. Rules are used to determine whom to ask in a given situation. The idea of an intelligent application might even be something as simple as a feedback system, where the user, over time, tells its document retrieval agent who provided the best links.

## 4 Future Work

Besides the application mentioned above, we are working on some of the aspects that are currently not implemented. We follow the RuleML developments with respect to queries, reaction rules, and integrity constraints.

Other important aspects are security and trust amongst the agents. Currently, a malicious agent can query the entire deductive databases of all other agents. To solve this issue, we are thinking about dropping the remote query mechanism and require an agent to send a regular message instead. This would allow the other agent to decide what to reply. We are also thinking about certain fact metadata fields such as who asserted a fact and possibly where it came from.

Last but not least we need to gain more experience on performance issues as well as the handling of different approaches, for example regarding integrity constraints. It will make sense to revisit the Java vs. SQL design choices, once the database vendors support more functionality.

## 5 Summary

This paper bases on Boley et. al's [2] idea of specifying an agent entirely using the Semantic Web mark-up languages RDF, RDF Schema, and RuleML. We described our implementation of OntoAgent, which bases on the OntoSQL tool that allows using a relational database as an inference engine. With a set of add-ons, written in Java, we were able to augment the system with the necessary components, i.e., reaction rules, a command library, and a messaging subsystem based on HTTP. Several design choices and trade-offs were discussed. In several cases we opted against the pure SQL variant with assertions and triggers due

to varying database implementations, unimplemented features, and engineering simplicity.

We believe that 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.

#### **Acknowledgements**

We want to thank Gerd Wagner at the Eindhoven University of Technology as well as the reviewers for their valuable comments and suggestions.

## **References**

1. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 28–37, May 2001.
2. H. Boley, S. Tabet, and G. Wagner. Design rationale of RuleML: A markup language for Semantic Web rules. In *Semantic Web Working Symposium*, 2001.
3. A. Eberhart. An agent infrastructure based on semantic web standards. In *Proc. of the AI-2002 Workshop on Business Agents and the Semantic Web*, May 2002.
4. 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*, February 2002.
5. R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, chapter 24, pages 729–760. Addison-Wesley, second edition, 1992.
6. S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Agent Theories, Architectures, and Languages*, pages 21–35, 1996.
7. B. Grosz. Representing e-business rules for the semantic web: Situated courteous logic programs in ruleml. In *Proc. of the Workshop on Information Technologies and Systems (WITS '01)*, 2001. Also refer to the extended version of the paper at <http://ebusiness.mit.edu/bgrosz/paps/wits01-extended-working-paper-12-01.pdf>.
8. 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.