OntoBroker
Mature and approved semantic middleware

Editor(s): Rinke Hoekstra, Universiteit van Amsterdam, The Netherlands
Solicited review(s): Jacopo Urbani, Vrije Universiteit Amsterdam, The Netherlands; one anonymous reviewer
Open review(s): Pascal Hitzler, Kno.e.sis Center, Wright State University; Georg Lausen, University of Freiburg, Germany

Jürgen Angele, ontoprise GmbH, angele@ontoprise.de, Germany

Abstract. OntoBroker provides a comprehensive, scalable and high-performance Semantic Web middleware. It supports all of the W3C Semantic Web recommendations for ontology languages and query languages. It is an ontology repository that includes a high performance deductive reasoning engine. Especially reasoning with rules is a major unique selling point for ontoprise. OntoBroker integrates a connector framework which makes it easy to connect a multitude of data sources like databases, web services etc. Thus it combines structured and unstructured data in one framework, OntoBroker is easy to extend and to integrate into existing IT landscapes and applications as it offers a variety of open interfaces. OntoBroker is also closely connected to ontoprise's ontology modeling environment OntoStudio which is the development environment for handling ontologies, mappings to information sources, rules, generating queries, creating business intelligence reports etc. At many customers OntoBroker serves as a common semantic layer which is accessed by various applications and integrates different information sources. OntoBroker is the run-time environment for industrial solutions like SemanticGuide, SemanticXpress, and SemanticIntegrator. As part of those meanwhile thousands of installations are in productive use.

Keywords: Ontobroker, semantic middleware, F-Logic, ObjectLogic, rule, semantic information integration, triple store

1. Introduction

OntoBroker provides a comprehensive, scalable and high-performance SemanticWeb middleware. It supports RDF(S) [16], OWL 2 RL [16], SPARQL[1] [16], and most of RIF-BLD [16]. It also supports F-Logic [12,3], and an extension of F-Logic which we call ObjectLogic.[2]

Based on OntoBroker, ontology-based applications can be developed which offer the following advantages (amongst others):

- The shared meaning (semantics) of information in a knowledge model
- The capturing of complex relationships
- The integration of heterogeneous data sources

Hence, the know-how and the business logic can be modeled separately from the execution logic. Users can flexibly adapt and extend the logic. When used systematically, ontologies form a conceptual semantic layer. This contains the relevant know-how for a particular business area or company, and can be accessed from all applications.

OntoBroker is also closely connected to ontoprise's ontology modeling environment OntoStudio [1]. It is either built into the OntoStudio single user application or serves as the core of the collaboration server, allowing multiple OntoStudios to collaborate online.

Various well-known customers currently use OntoBroker as semantic middleware. It serves as an ontology repository and provides a variety of ontology and integration services for different applications. OntoBroker also lies at the core of solutions such as the SemanticGuide and SemanticXpress, which have several thousands of installations in productive environments at customer sites.

OntoBroker is designed as an open architecture with various APIs. This allows partners and custom-

[1] http://www.w3.org/TR/rdf-sparql-query/

[2] See Section 2.6 for more details on ontobroker's coverage of different languages.

ers to closely integrate OntoBroker into their own solutions or own IT landscape.

In the following paper we describe OntoBroker's architecture, OntoBroker's features, its integration features, and sketch briefly industrial applications.

2. Onto Broker's Architecture

OntoBroker is an ontology repository that includes a high performance deductive reasoning engine. It provides a rich functionality for managing ontologies. It offers various import/export filters for ontologies or other structures such as file system hierarchies, UML, and so on.

OntoBroker supports connectivity of other information sources such as relational database systems, Web services, Excel sheets, and search engines to ontologies, and hence provides a sophisticated integration framework. OntoBroker has its own search index integrated for efficiently searching in ontologies, and for searching in documents. OntoBroker includes a Web-server for supporting Web-based applications and Web services. Furthermore, the UIMA framework[3] is integrated, supporting a huge amount of text analysis plug-ins from different providers. OntoBroker thus provides a broad basis for semantic applications.

OntoBroker has been designed to be used as a runtime system within semantic applications, or as a core part of ontology-based services in an Intranet set-up. Hence, it is available with a lot of different well-documented interfaces and extension possibilities. For example, OntoBroker makes up the core of the collaboration server, which is used by several OntoStudio clients to model the same ontology at the same time in a collaborative way. In the following chapters we will give more details on all of the components mentioned (see fig.1).

The core of OntoBroker contains a storage layer where the ontologies are stored. It is possible to configure this storage option as a persistent layer or as being main memory-based. The persistent layer is implemented as an embedded relational database which provides standard relational database functionality.

The deductive reasoning engine, (i.e. the inference kernel) processes normal logic programs (Horn logic extended by non-monotonic negation) [14] with well-founded negation [11] using various reasoning methods. The reasoning does not integrate equality reasoning. RDF(S) and OWL are translated into ObjectLogic primitives and SPARQL queries are translated into ObjectLogic queries. Equality reasoning for OWL 2 is axiomized, i.e. OntoBroker adds additional equality axioms to the logic program.

One of the major design goals for OntoBroker was a semantic middleware which is easy to extend and easy to integrate into existing IT landscapes and applications. For this reason, a variety of interfaces are available for this purpose. At the bottom, a connector framework enables various data sources to be attached to an ontology. The product is shipped with connectors for relational databases, for LOD (linked open data) [15], for other OntoBrokers or for Web services. As the connector API is well-documented, customers and partners can easily develop their own connectors. OntoBroker can also be extended using built-ins which define procedural attachments that are seamlessly embedded into ObjectLogic. Again it is easy for customers to develop their own built-ins. The left hand side of fig. 1 shows the option of:

- Extending OntoBroker with application-specific commands,
- Customizing it with startup scripts,
- Customizing the full-text index.

The top of the graphic shows the standard Web service API that allows the adding and removal of facts, plus the placing of queries in ObjectLogic or SPARQL. The Java API is a powerful API for developing arbitrary applications on top of OntoBroker. This API is a remote API which supports clients who are running on a different computer to the OntoBroker server. Finally, OntoBroker comes with "dynamic Web services". A query that was modeled in the OntoStudio query builder can be deployed as a Web service by simply pressing a button. This Web service is deployed to OntoBroker which executes the Web service in the final application. OntoBroker can expose its contents as an LOD (linked open data) endpoint.

All of the ontoprise solutions such as the SemanticGuide, Semantics for Sharepoint, SemanticIntegrator, or SemanticXpress use these interfaces and are hence layered on top of OntoBroker. Hence, OntoBroker integrates different legacy information sources, provides ontologies for different purposes and fulfills all of the semantic needs of applications sitting on top of OntoBroker. OntoBroker is therefore the a very powerful middleware for companies and fulfills most of the information requirements needed for applications on top.

[3] http://uima.apache.org/

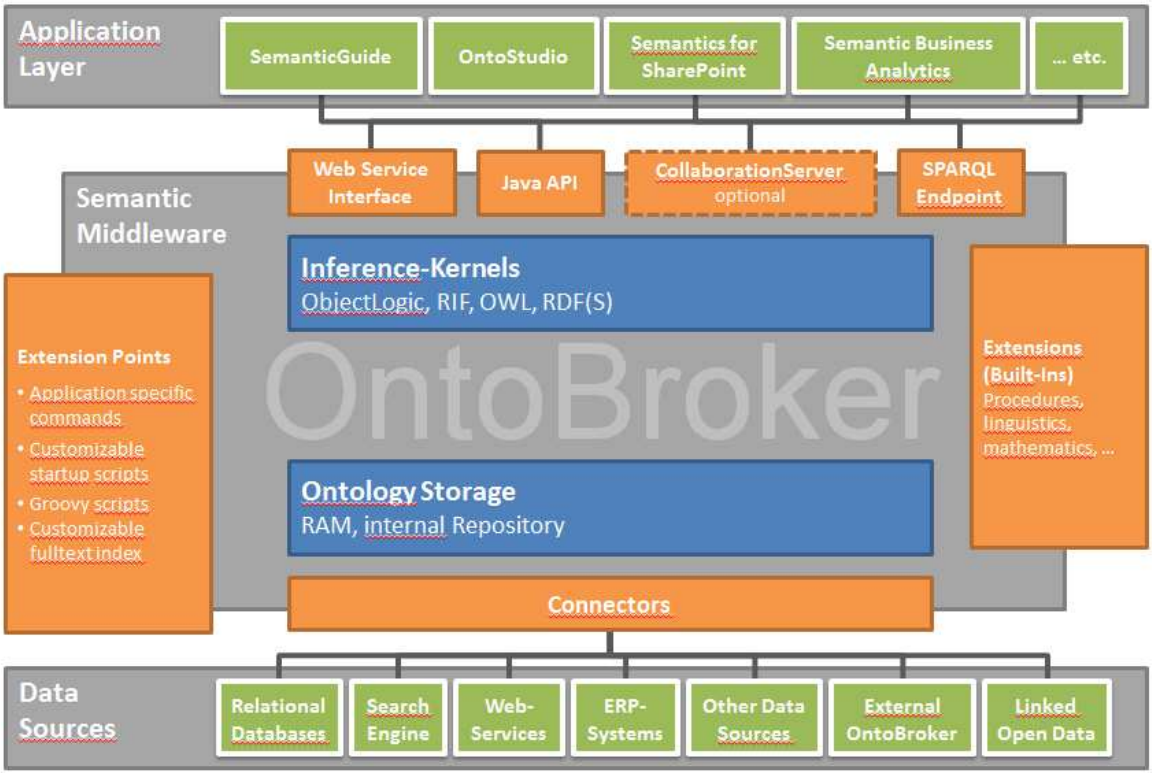

Figure 1. Onto Broker's Architecture

2.1 Inference Algorithms

In the kernel of OntoBroker, there is not one single but several reasoning methods available which can either be selected automatically or on demand. They may be classified into three different classes: Bottom-up reasoners (also called semi-naive evaluations) [18], simulated top-down reasoners and top-down reasoners. There are 3 different bottom-up reasoners: db-bottomup, bottomup2, bottomup3, 3 reasoners for simulating top-down reasoning: magic set [5], supplementary magic set [6] and dynamic filtering [13], and there is one purely top-down reasoner: QSQ [19].

In the following we briefly describe bottom-up reasoning, magic sets, dynamic filtering and finally QSQ. Let's introduce a small example for this purpose. We describe facts and rules about relatives, like parent, child, and sibling in ObjectLogic (the question marks indicate variables):

```
Leon[hasParent->Anja].
Luca[hasParent->Anja].
Kevin[hasParent->Petra].
Jan[hasParent->Petra].

?X[hasSibling->?Y]
:-?X[hasParent->?Z],?Y[hasParent->?Z].

// ?X is sibling to ?Y if both have the same
parent ?Y

?- Leon[hasSibling->?Y].
// who are the siblings of Leon
```

2.1.1. Bottom-up Reasoning

A bottom up reasoner takes the given facts, applies the rules and hence creates derived facts. Once again the rules are applied, and so on. This continues until no new facts can be derived. In our example this looks as follows.

First of all the rule is applied to the facts by substituting the variables by the facts:

```
Luca[hasSibling->Luca]
:-Luca[hasParent->Anja,
Luca[hasParent->Anja].

Leon[hasSibling->Leon]
:- Leon[hasParent->Anja],
Leon[hasParent->Anja].
```

```
Luca[hasSibling->Leon]
:- Luca[hasParent->Anja],
Leon[hasParent->Anja].

Leon[hasSibling->Luca]
:- Leon[hasParent->Anja],
Luca[hasParent->Anja].

Kevin[hasSibling->Kevin]
:- Kevin[hasParent->Anja],
Kevin[hasParent->Anja].

Jan[hasSibling->Jan]
:- Kevin[hasParent->Petra],
Jan[hasParent->Petra].

Kevin[hasSibling->Jan]
:- Kevin[hasParent->Petra],
Jan[hasParent->Petra].

Jan[hasSibling->Kevin]
:- jan[hasParent->Petra],
Kevin[hasParent->Petra].
```

This leads to the following derived facts:

```
Luca[hasSibling->Luca].
Leon[hasSibling->Leon].
Luca[hasSibling->Leon].
Leon[hasSibling->Luca].
Kevin[hasSibling->Kevin].
Jan[hasSibling->Jan].
Kevin[hasSibling->Jan].
Jan[hasSibling->Kevin].
```

A query is simply a rule without a head. Therefore, the query is applied in the same way to facts by substituting the variables in the query. This leads to the following instantiations of the query:

```
?- Leon[hasSibling->Leon].
?- Leon[hasSibling->Luca].
```

Hence OntoBroker finally answers with the substitutions for the variables in the query:

```
?Y = Leon
?Y = Luca
```

We see that bottom-up reasoning is a simple reasoning method. It has the disadvantage that a lot of (intermediate) facts are generated which are generally not necessary for answering the query. In our case, all of the facts which were derived by the rule but do not instantiate the query were derived needlessly. On the other hand, top-down reasoning sometimes provides so much overhead that this simple reasoning strategy performs best of all.

2.1.2. Dynamic Filtering

Dynamic filtering tries to avoid the mentioned disadvantages of purely bottom-up reasoning. It propagates ground terms in the query, and in the rules top-down to use it as early as possible as a filtering mechanism. Hence, in many cases, it is possible to avoid deriving needless (intermediate) facts. Let's take another look at our example.

We start with the query. `Leon` is a ground term in that query.

```
?- Leon[hasSibling->?Y].
```

The query matches the head of our rule.

```
?X[hasSibling->?Y]
:- ?X[hasParent->?Z,
?Y[hasParent->?Z].
```

This means that we can substitute the variables in the rule head with terms so that it equals our query body. This leads to the following (partially) instantiated rule. Notice that if a variable is substituted in the head it is also substituted in the same way in the body:

```
Leon[hasSibling->?Y]
:- Leon[hasParent->?Z],
?Y[hasParent->?Z].
```

Now we will consider the first rule body and search for possible instantiations of the variable ?Z by searching in our fact base (extensional database, EDB). We find only one fact which matches with this body leading to an instantiation of the variable `?Z` to `?Z = Anja`.

```
Leon[hasParent->Anja].
```

This instantiation of variable `?Z` is clearly also propagated to the second rule body leading to:

```
?Y[hasParent->Anja]
```

We again match this rule body with all facts of the EDB and thus find the substitution of variable `?Z` by `Leon` and `Luca`. That way we get two different instantiations of our rule:

```
Leon[hasSibling->Leon]
:- Leon[hasParent->Anja],
Leon[hasParent->Anja].
Leon[hasSibling->Luca]
:- Leon[hasParent->Anja],
Luca[hasParent->Anja].
```

And thus two derived facts:

```
Leon[hasSibling->Leon].
Leon[hasSibling->Luca].
```

Which again lead to the final answer by matching with the query:

```
?Y = Leon
?Y = Luca
```

In this case, we clearly see that only those facts and intermediate facts have been derived which were necessary to correctly answer the query. On the other hand, there are cases where huge amounts of top-down propagations vanish the performance win.

2.1.3. Magic Set Evaluation

Magic set reasoning simulates similar behavior to dynamic filtering by modifying the rules and then processing them using a bottom-up reasoner. Let's take another look at our example to motivate the rule transformation process.

In our query Leon is a ground term. For dynamic filtering we saw that this ground term is propagated top-down to the rule. The rule transformation process does something similar. It creates a magic fact and then creates a new rule from our rule:

```
magic1(Leon).
?X[hasSibling->?Y] :-
magic1(?X), ?X[hasParent->?Z],
?Y[hasParent->?Z].
```

We immediately see that this transformation brings the restricting ground term Leon directly to our rule, hence restricting the intermediate results just at the bottom of the rule graph. Hence the results of the and operation of the first two rule literals is:

```
Leon[hasParent->Anja].
```

The result of the `and` operation with the last rule literal then leads to the following intermediate result (instantiations of the three rule variables):

```
Leon, Luca, Anja
Leon, Leon, Anja
```

Hence we get two different rule results, which are also the final results of the query:

```
Leon[hasSibling->Luca]
Leon[hasSibling->Leon]
```

We again see that transforming the rules in that way and processing the resulting rule set in a bottom-up way reduces the number of intermediate results which do not contribute to the answer. We also see that this results in a similar data flow (at least during bottom-up reasoning) as can be observed for dynamic filtering. For magic set reasoning we observe a similar trade-off between this reduction effect and the additional performance loss by creating these additional rules. We have seen queries which are better evaluated in a purely bottom-up fashion and others which are better evaluated by dynamic filtering or magic sets. In general, it can be said that dynamic filtering beats magic sets in all those cases which are better for simulated top-down reasoning.

2.1.4. QSQ

QSQ is pure top-down reasoning based on resolution. For Horn rules resolution can be seen as continuously unfolding rules. Unfolding means that a body literal is substituted by a rule body where the rule head unifies with the body literal. Additionally, variables must be renamed and substituted correspondingly. Hence, we get sub-queries which are again handled in the same way. Let's take another look at our example.

We again start with the query.

```
?- Leon[hasSibling->?Y].
```

The query matches (unifies) the head of our rule.

```
?X[hasSibling->?Y]
:- ?X[hasParent->?Z], ?Y[hasParent->?Z].
```

Now this rule is unfolded into our query, which means that the single query literal is substituted by the rule body and the variable ?X is substituted by the constant `Leon`;

```
:- Leon[hasParent->?Z],?Y[hasParent->?Z].
```

Now we will consider the first body literal in the above query and search for possible instantiations of the variable ?Z by searching in our fact base (EDB). We find only one fact which matches this body leading to an instantiation of the variable `?Z` to `?Z = Anja`.

```
Leon[hasParent->Anja].
```

This instantiation of variable ?Z is clearly also propagated to the second rule body leading to:

```
?Y[hasParent->Anja]
```

We again match this rule body with all facts of the EDB and thus find the substitution of variable `?Z` by `Leon` and `Luca`. That way we get two different instantiations of our query:

```
:- Leon[hasParent->Anja],
Leon[hasParent->Anja].
:- Leon[hasParent->Anja],
 Luca[hasParent->Anja].
```

We know from the first step that the answers are all possible substitutions of the variable `?Y` in our sub-query. Hence we again get two possible answers:

```
?Y = Leon
?Y = Luca
```

In general, it turned out that bottom-up algorithms (including simulating top-down by bottom-up) are better as top-down reasoning tends to handle smaller sets of tuples which is less efficient. Nevertheless there are rare cases where QSQ performs better than all of the other reasoning methods. We should mention that the QSQ implementation in OntoBroker has some restrictions. Because it has not yet been implemented it cannot handle left-recursive rules and cycles in data relations.

2.2 The Reasoning Process

A query (an object logic query like we have seen in section 2.1 or a SPARQL query) is processed in several successive steps (see Fig. 2). First of all, the query is parsed and compiled into an internal data structure. The IDB (intensional database) contains all of the rules. From this set of rules, our query selects all of the rules that could contribute to the query. The resulting set of rules is then optimized by so called rewriters. Rewriters modify the rules but ensure that the modified rules evaluate to the same answers. For example, our magic set rewriter is such a rewriter. There are much more of these, simple ones like eliminating duplicate literals in rules or more complex ones like rewriting rules into SQL statements when integrating SQL databases.

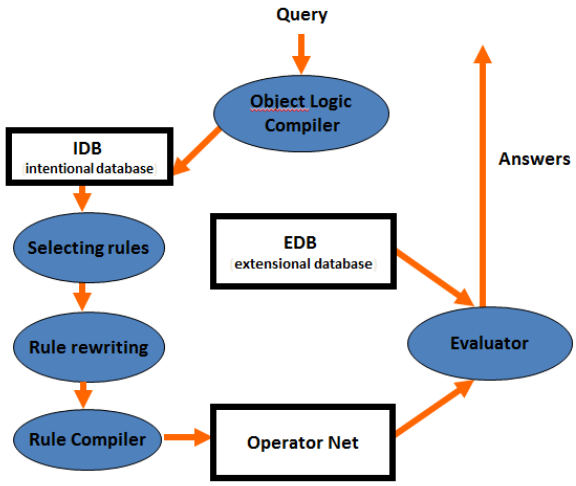

Figure 2. Processing of a Query in OntoBroker

Then a rule compiler creates a so-called operator net. Basically an operator net is a low level representation of all of the operations needed for processing the set of rules. Such an operator net contains operations like join, match, access to the EDB, projection, operations for built-ins, operations for connectors, and so on. Fig. 3 shows such an operator net. It contains move operations (MV), join operators (&), collectors, distributors (V), and rule out operators (R). Move operations just move tuples to another node, collectors store intermediate results, and distributors distribute tuples to several other nodes.

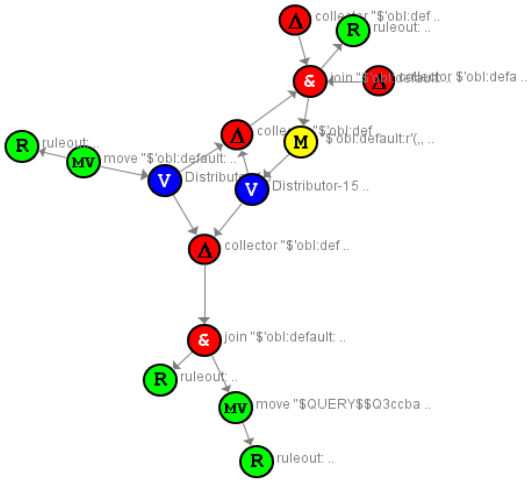

Figure 3. Operator Net as a Result of Rule Compilation

Such an operator net is purely data flow-oriented. This means that the data flow from data sources (accessors for EDB facts) through the operator net. Every operator performs its operation and sends the results to the successor nodes. Every reasoning algorithm has a separate rule compiler. Our example operator net in fig. 3 has been produced by a bottom-up compiler. The operator net for the same set of rules, but for dynamic filtering as a reasoning algorithm is shown in fig. 4.

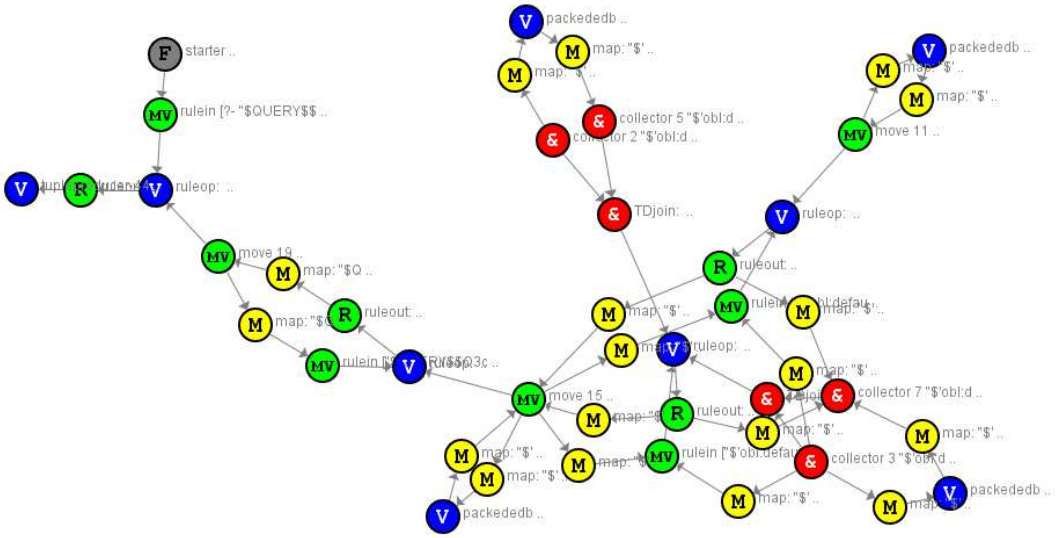

Figure 4. Operator Net as a Result of the Dynamic Filtering Compiler

We see that, on the one hand, the operator net becomes much more complex. On the other hand, in this case the reasoning time is drastically reduced because of the top down propagation which is expressed herein.

This operator net provides a way of parallelizing reasoning as the operators work independently from each other. This means that the processing of every operator can be done in a separate Java thread. Different active parts, for instance different join operations which do not influence each other in the operator net immediately can be processed in such different threads in parallel.

Together with Silicon Graphics (SGI) [4] we have shown how this architecture scales very well with the appropriate high performance hardware on a high end computer with 128 processors and 512 gigabyte main memory. In the open rule benchmark [5] OntoBroker outperforms competitor systems by far as shown in Fig. 5. For instance the above mentioned operator net parallelization reduced the answering time for the wine benchmark from 3.2 s to 1.5 s. In addition to that single join operation may run in several parallel threads. For instance for the cyclic transitive closure test with 500000 tuples in the open rule benchmark parallelizing single joins reduced the answer time from 39 s (single threaded) to 12 s on a quad core machine (4 threads). In another case where a large chemical process has been validated by OntoBroker the answering time has been reduced from 8 hours to under a minute.

The whole query processing step is multi user-capable. This means that multiple users can send queries to OntoBroker at the same time. All of those queries are processed in parallel. So there are three kinds of parallelisms involved. Several queries from several users can be evaluated in parallel, and each query can in turn be evaluated in several parallel threads and finally in some cases a single join operation runs in several parallel threads. This supports multi-core / multi-processor hardware very well.

We run all our reasoning algorithms every day through the open rule benchmark and a lot of addi-

4 http://rulebench.projects.semwebcentral.org/

5 http://www.sgi.com/global/de/newsroom/2008/2008-04-08-2.html

tional own performance benchmarks. This shows us performance regressions immediately and makes it easy to compare the different algorithms to each other. The statements in section 2.1 about the different reasoning algorithms are based on these benchmarks.

We have seen models which contain thousands of rules but limited number of facts. On the other hand we are processing models with large amounts of data which contain hundreds of millions of facts but only few and simple rules. Huge and complex rule sets in combination with huge amounts of facts clearly bring Ontobroker to its performance limits. We are continuously working to extend these limits.

Figure 5. Open rule Benchmark Comparing OntoBroker with a Number of Other Rule Engines[6][20]

2.3 Materializing Models

In the previous section we described how query evaluations take place inside OntoBroker. The same reasoning methods can also be used to materialize inferences. This means that rules are evaluated directly after loading and the results are stored in the internal ontology/ triple store which has the effect that, during query evaluation, these facts have to be accessed and retrieved only so that no evaluation takes place during query time. Clearly this may drastically improve response times but may also blow up the amount of data to be stored. OntoBroker even provides the functions for incrementally materializing models. This means that if a new triple is added to the model or deleted from the model, this materialization process only has to be repeated for a small subset of the model and not for the entire model.

2.4 The Storage Layer

All facts are separated from the rules and are stored in the extensional database (EDB). This EDB may either be configured to reside in the main memory or it may reside in a relational persistency layer. It is clear that for performance reasons the first configuration is preferable. Ontobroker stores a property in a quad: the object, the property, the value of the property and the module. As a rule of thumb, 10 million quads need 1 GB main memory on a 32 bit operating system. A 64 bit operating system requires roughly twice the size. This scales in a linear way.

[6] http://xsb.sourceforge.net/;http://cracs.fc.up.pt/node/3547

Currently this relational layer is based on the Java embedded relational database H2[7] and is used to store smaller ontologies in a persistent way.

Currently there is ongoing development work to substitute the persistency layer entirely with an in-house development based on B+ trees [7]. This implementation is targeted at sizes of up to 10 billion quads. This layer will be seamlessly integrated into our reasoning algorithms and should hence increase performance dramatically compared to the current persistency implementation. This persistent layer ensures rapid imports, rollback, parallel transactions and backups. A first prototype shows very promising results for the loading performance. 1 billion quads are loaded in roughly 3.5 hours.

2.5 Procedural Attachments

Some aspects cannot be easily described using logic. For example, complex mathematical algorithms should be described in a procedural way instead. OntoBroker can easily be extended with such procedural algorithms. Within ObjectLogic, these procedural attachments are called built-ins. They may be used inside rules or queries using predicate logic literals. For example, all of the mathematical built-ins are internally given in the same way. If we want to multiply numbers we could use the multiply built-in:

```
?X[hasDollarPrice->?Y] :-
?X:Product[hasEuroPrice->?Z],
_multiply(?Z,1.4,?Y).
```

The last body literal is a built-in literal. The extension of such a built-in is not a given set of facts. Instead the extension is computed by an algorithm.

Such a built-in may be easily developed by partners or customers. A well-documented API is available for such a built-in and the Java code must use this API. This Java code is compiled against an OntoBroker and then the class file is thrown into the extensions folder of OntoBroker. During the launch of OntoBroker this built-in is loaded and is subsequently available within queries and rules. Ontoprise provides a special workspace for the Eclipse Java programming environment which makes it easy to develop such built-ins. Connectors, i.e. software accessing external systems are developed in a similar way (see the section 3 information integration).

[7] http://www.h2database.com/html/main.html

2.6 Language Support

In the mean time, the W3C has issued several recommendations for languages for ontologies like RDF(S) [16], OWL 2 [16], SPARQL [16], and RIF [16]. OntoBroker supports all of these, with some reasonable restrictions.

The simplest ontology language is RDF(S). It provides classes, class hierarchies, properties and property hierarchies. OntoBroker fully supports RDF with minor restrictions for some data types. Hence, some numeric data types are mapped to a restricted set of data types. Full RDFS reasoning is supported . All of the common triple file formats such as RDF/XML, Turtle, N3, and N-Triples are offered for import and export purposes.

SPARQL is the semantic query language which can be used for all models (ObjectLogic, RDF, OWL) in OntoBroker. OntoBroker supports SPARQL 1.1 Query and Update[8].

OWL is another language used to express ontologies. Unlike RDF(S) it provides more abstract and expressive modeling primitives such as cardinalities, equality, complex class expressions, and so on. OWL is a decidable subset of predicate logic. As OntoBroker uses the OWL 2.0 API developed at the University of Manchester, it can handle and store full OWL 2 ontologies. The reasoning in OntoBroker is restricted to the OWL 2 RL subset which is a standardized subset that is well suited for large ontologies. Equality is axiomized, i.e. expressed by special axioms which are added by OntoBroker to each model. This reasoning is sound and complete for the OWL 2 RL subset of OWL 2.

Ontobroker implements RIF [16] by a dialect covering ObjectLogic. The intersection with other dialects, e.g. BLD, include among others frames, memberships, subclasses, conjunctions, disjunctions and rules. Some BLD features are not supported like derived equality and imports of RDF or OWL documents, i.e. combined semantics of RIF and RDF respectively OWL.

F-Logic (Frame Logic) [12] combines the advantages of conceptual modeling that come from object-oriented frame-based languages with the declarative style, compact and simple syntax, and the well-defined semantics of a logic-based language. F-logic supports typing, meta-reasoning, complex objects, methods, classes, inheritance, rules, queries, modularization, and scoped inference. In the meantime, F-

[8] http://www.w3.org/Submission/SPARQL-Update/

Logic has been enhanced by a lot of primitives borrowed from OWL while preserving its scalability [3]. F-Logic can be translated to Horn logic with non-monotonic negation, which is a very subset of predicate logic with highly efficient reasoning algorithms. Hence, the reasoning scales very well with large ontologies. It is 'Turing complete' (computationally universal) which means that everything that can be expressed on a computer can also be expressed in F-Logic. This is very important for ontoprise's applications. For example, a normalization of concurrencies in an integration scenario such as: converting prices from dollar values to euro values cannot be directly expressed in OWL. Ontobroker in fact covers a light extension of F-Logic which we call ObjectLogic. It additionally provides support for complex formulas in rule bodies, an extended set of built-ins, and property hierarchies. Techniques for realizing these simple extensions are well-known and have been suggested in the context of SWSL [22], FLORA-2 [23], FLORID [21], and on the F-logic discussion group[9]. Our many commercial applications show that ObjectLogic is an excellent language for applications.

3. OntoBroker as an Integration Engine

A very important application of OntoBroker is run-time ontology-based information integration. This means that information sources such as databases, Web services, linked open data (LOD) [15] sources, search engines, and so on, are attached to an ontology. The data in these data sources usually remain in these data sources and are not copied to the internal ontology store. At query-time, a query to the OntoBroker server is then translated into the query language for the attached data source, for example, SQL for relational data bases. These queries are then sent to the information sources, evaluated, and the results are integrated back into the reasoning process. As OntoBroker also provides rule materialization, i.e. executing all of the rules and storing the data in the internal ontology store, this also includes extraction of the data from the data sources and storing them locally in the own store. On the other hand, usually ontoprise's customers prefer query-time integration as they want to continue to maintain their own data stores.

Conceptually, information integration requires at least four different layers (cf. fig. 6):

- The bottom layer represents different data sources which contain or deliver the raw information which is semantically reinterpreted on an
- upper layer viz. ontologies. OntoBroker comes with connectors for Oracle DB, IBM DB2, Microsoft SQL server, Web services, other OntoBroker servers and a connector for SPARQL endpoints. Various other connectors have also been developed during projects.
- The second layer assigns a so called "data-source ontology" to each of the data sources. These "data-source ontologies" reflect only database or WSDL schemas of the data sources in terms of ontologies and can be created automatically. Hence, they are not real ontologies as they do not represent a shared conceptualization of a domain.
- The third layer represents the business ontology using terminology relevant to business users. This ontology is a real ontology, i.e. it describes the shared conceptualization of the domain at hand. It is a reinterpretation of the data described in the data-source ontologies and hence gives these data shared semantics. As a result, a mental effort is necessary for the reengineering of the data source contents, which cannot be done automatically.
- On a fourth layer, views to the business ontologies are defined. Basically, these views query the integration ontology for the information required. Exposed as Web services they can be consumed by portals, composite applications, business processes or other SOA components.

The elements of the different layers are connected via so-called mappings. The mappings between the data-sources and the source ontologies are created automatically, the mappings between the ontologies are engineered manually and the views are manually defined queries. Mappings define how source structures are mapped to destination structures. Hence, mappings provide ways of restructuring information, renaming information or transforming values. Until now, we have not considered and do not plan to consider approaches that try to derive such mappings automatically [17].

[9] http://forum.projects.semwebcentral.org/forum-syntax.html

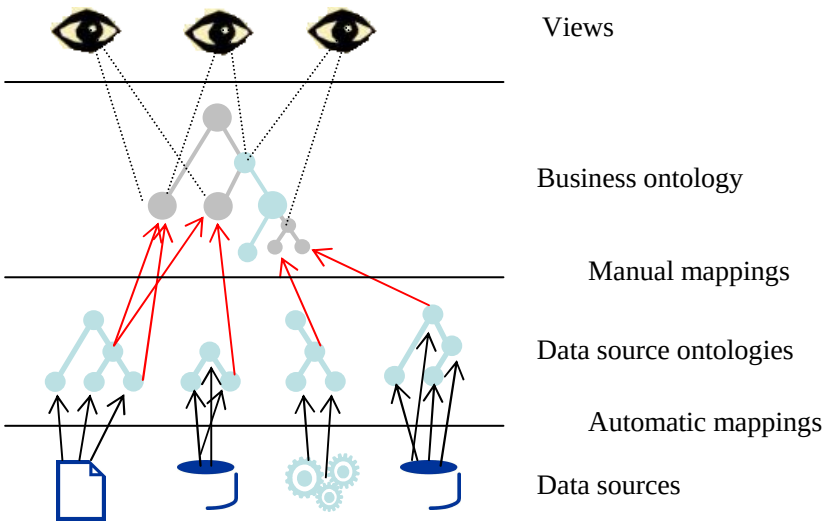

Figure 6. The Conceptual Layering of Ontologies in an Integration Scenario

The arrangement of information on different layers and the conceptual representation in ontologies, plus the mediation between the different models via mappings bring the following advantages:

- The reengineered information in the business ontology is a value of its own. It represents a documentation of the data source contents. Representation as an ontology is a medium that can be easily discussed by non-IT experts. By aggregating data from multiple systems, this business ontology provides a single view of the relevant information in the user's terminology. More than one business ontology enables different perspectives on the same information.
- It is easy to integrate a new data source with a new data schema into the system. It is sufficient to create a mapping between the corresponding source ontology and the integration ontology and hence does not require any programming know-how; pure modeling is sufficient.
- The mediation of information between data sources and applications, via ontologies, clearly separates both. In this way, changes in the data source schemas do not affect changes in the applications but only affect changes in the mediation layer, i.e. in the mappings.
- This conceptual structure strongly increases business agility. It makes it very easy to restructure information and hence react to changing requirements. It is not necessary to change either the data sources or the applications. It is only necessary to modify the business ontology and the mappings. Once again no programming skills are required, all the steps are made at model level. In this way it minimizes the impact of change, eases maintenance and allows for the rapid implementation of new strategies
- Ontologies have the powerful means to represent additional knowledge on an abstract level [9]. So, for example, the business ontology may be extended by additional knowledge about the domain using rules. Hence, the business ontology is a reinterpretation of the data, plus a way of representing complex knowledge correlating this data. In this way, business rules are directly captured in the information model, they determine the optimal system access and bring every user to the same level of effectiveness and productivity.

Sometimes projects at customers integrate a fourth layer between the data-source layer and the business ontology. This is called the normalization layer. For example, different currencies may be normalized in this layer.

The logics such as renaming, restructuring, normalization, and filtering in mappings are represented by rules. For example, to map a database table to a class in the data source ontology we use a rule such as:

```
?O:Person[hasName->?N] :-
databaseAccess(database access info,
database table name, C(column name, ?N)).
```

In a similar way, ontology elements of one layer are connected to the next layer via rules. Using OntoStudio [1], databases may be integrated in a comfortable way. The database schema can be imported and the data source ontologies (with the mappings to the databases) are created automatically. A mapping

editor is then used to map the data source ontologies to the business ontology. Fig. 8 shows the mapping of a Microsoft SQL schema to a simple business ontology speaking about employees.

Finally the query builder in OntoStudio is used to query for interesting information from the business ontology. These queries can then be deployed as a Web-service by simply pressing a button. These deployed queries then represent the fourth layer, the views layer.

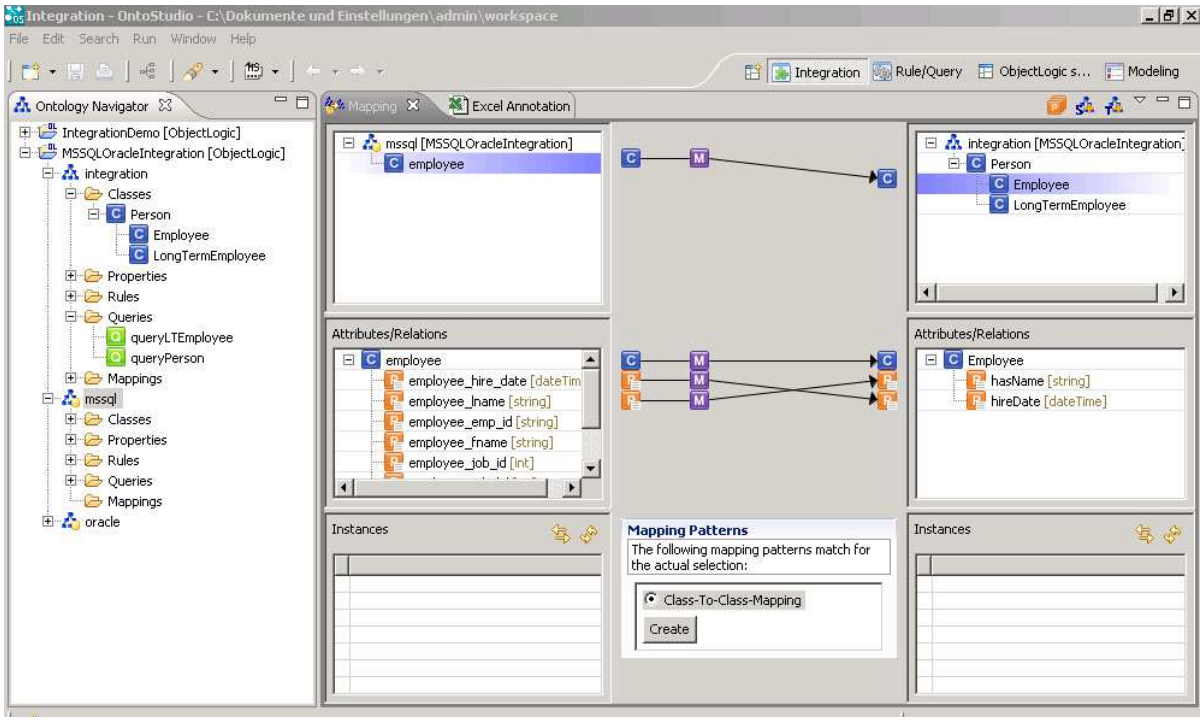

Figure 7. Mapping Editor in OntoStudio

4. OntoBroker as an LOD Endpoint

While the traditional Web links a huge amount of human-readable Web pages with each other, the Semantic Web creates, as the name suggests, a web of data. Linked open data (LOD) [15] sources are a light-weight version of this web of data. They collect structured information and provide this information on LOD servers. These LOD servers are accessible using the http-protocol and can thus be accessed by ordinary browsers. Objects are identified by URIs. Unfortunately the URIs for same objects are different in the different LOD servers. In most cases these data are structured using RDF and SPARQL is used as a query language for these servers. The data are interlinked with other data in other LOD servers and thus

create a huge graph of interlinked data. For example, DBpedia[10] [8] is a system that extracts information from Wikipedia and provides this information in a structured way as an LOD server. Freebase[11] [10] is another example for an LOD server. Freebase does not use SPARQL as query language. There are many links in Freebase which refer to DBpedia and vice versa.

OntoBroker can be exposed as being such an LOD endpoint, because it supports the appropriate protocol. Together with its integration capabilities this provides a large number of possibilities for bringing data into the linked open data cloud.

5. Industrial Applications of OntoBroker

The following sections briefly describe some of the industrial usages of OntoBroker.

6.1 SemanticGuide

For the producers of complex long living industrial products, the service and maintenance of these products is an important issue. Vertical areas are, for example, engineering, automotive, aerospace, and so on. For example, at a large robot provider ontoprise's SemanticGuide is used to support the customer service department [4].

The number of variations of robot configurations is extremely high. On average, one new system configuration – for instance, in the form of a new software update – comes to market every month. This represents a major challenge, in particular for younger and less experienced service technicians. With the help of a knowledge-based advisory system, the entire service crew can access the wealth of expertise from senior technicians.

OntoBroker is used as the semantic backend within the SemanticGuide to record the symptoms observed and use an ontology about robots and an ontology about symptoms, failures, and solutions. Additionally, a decision tree is used for the guided search, in order to find the best solution to the issue at hand. Amongst others, the SemanticGuide is also in use at Heidelberger Druck GmbH, at General Electric, and at SMA, and others for the same purpose. The fact that there is an offline version installed on the laptops of the maintenance engineers, in the meantime, means there are thousands of installations of OntoBroker in dozens of countries.

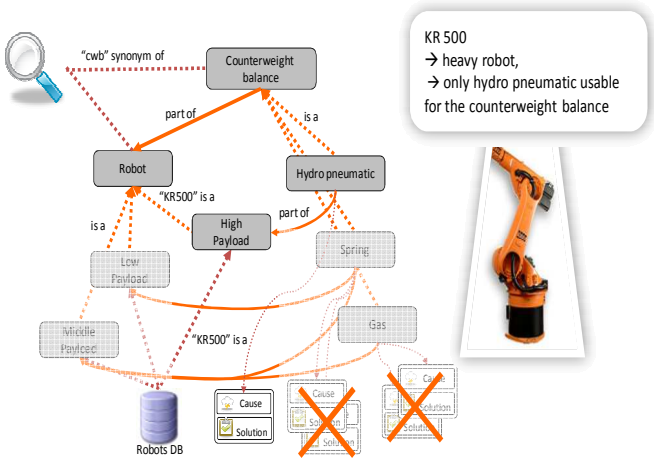

Figure 8. Sample Robot Example: As the specific KR 500 robot is a heavy weight robot and all heavy weight robots must be equipped with a hydro-pneumatic counterweight balancing system, the search for "cwb" in the context of "KR 500" can exclude all cases affecting gas or spring counterweight balancing systems.

6.2 Condition-Based Monitoring

In this application, Alstom is commissioning a hydro power plant in Malaysia, a project which is called the "Bakun Hydroelectric Project". The customer of the Bakun Hydroelectric Project requested an expert system delivered together with the plant that focuses on supporting the plant operators in predicting and handling issues in difficult situations. The role of the monitors is to pre-detect upcoming undesired operating conditions and faults and therefore suggest maintenance actions.

OntoBroker is fed with sensor values, like temperature, pressure, and so on, from the different parts of the power plant. Rules are used to analyze these values and, if necessary, to fire alarms. For example, let's consider the circumstances when an oil pump runs stable: The oil pump has a stabilization time of 5 minutes after start. The start of the pump is indicated by signal value 1 from signal CL105. This could be represented in a rule like: "if the signal CL105 has the value *1* and the time stamp *T*, then the pump runs stable at time *T+5* min.

Monitors are developed, maintained and deployed in the ontology development environment OntoStudio. OntoStudio has been extended by a component for the development and administration of such

[10] http://de.dbpedia.org/
[11] http://www.freebase.com

monitors. A specialized editor (rule wizard) has been developed for the easier definition of these monitors (cf. fig. 9).

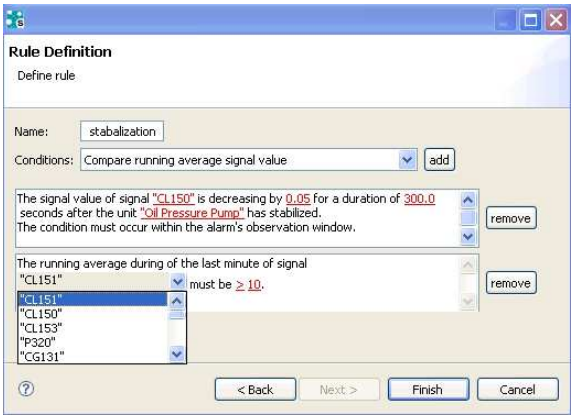

Figure 9: Rule Wizard for Monitor Rules

6.3 Semantic Information Integration

OntoBroker is part of the integration suite CrossVision from Software AG [2]. Within Software AG, the Information Integrator was used for a first project whose mission was to integrate data that, on the one hand, resides in a support information system and, on the other hand, is stored in a customer information system. The support information system, for example, maintains information about customers, their contact information and active or closed support requests. The customer information system stores information about clients, contracts, and so on. While the customer system stores its data in an Adabas database, the support system uses an SQL server. The integrated data view is exposed to various parties inside the company in a browser-based application. For example, instead of only seeing reported support requests, support engineers should get a more complete picture of a customer whilst talking to them.

In another information integration application at a healthcare company in USA, production data has been integrated. The production of this healthcare provider is fully automated. In several production lines an impressive amount (hundreds) of specialized machines and robots produce a huge amount of contact lenses. These products have to fulfill a high quality standard which has to be approved by the FDA. All of the production data and all of the sensory data measuring the product parameters are integrated into a 5-layer ontology model. Reports on top of this joint ontology provide vital information to quality management and for FDA approvals. Before this integration, the process of creating such a QM report or a report for FDA took about 10 days to retrieve the necessary information, combine it and create the reports. Thanks to the ontology-based semantic information integration, this process now takes less than a second. It is clear that this degree of acceleration drastically reduces junk and speeds up the ramp-up process for product variants. Particularly the last point drastically speeds up the time to market.

6.4 Vocabulary Management

Handling large vocabularies is an important task in some areas like pharmaceuticals, media, etc. For example, at Roche they use taxonomies like ICD[12], Mesh[13], NCI[14] and others. They describe diseases, organs, drugs, and so on. In addition, Roche has its own internal vocabulary. All of these vocabularies are linked to each other and subsequently used for information integration purposes, and to ensure a consistent communication. The different users maintain the vocabularies collaboratively, and a workflow and access rights define which users can suggest new terms and which users can integrate them into production.

OntoBroker forms the semantic backbone of this application. Inside OntoBroker, the different vocabularies are managed as light weight ontologies. Additional information from the LOD cloud is linked to these vocabularies. On the one hand, OntoBroker runs as a vocabulary and as a collaboration server. On the other hand, OntoBroker serves as an information provider for different applications inside Roche which need these vocabularies. In the future, these vocabularies will be used for a semantic/facetted search, and within applications such as MS Office.

6. Summary

OntoBroker is a powerful semantic middleware that hosts ontologies and reasoning on top of these ontologies. All of the W3C standards for ontology languages are supported. The integrated rule engine is at the top of the open rule benchmark. The integration capabilities allow the dynamic integration of a variety of information sources which are queried during

[12] http://www.who.int/classifications/icd/en/
[13] http://www.ncbi.nlm.nih.gov/mesh/
[14] http://www.cancer.gov/

query-time. In turn, OntoBroker can be used as a source of information either within companies or also as an LOD source.

OntoBroker is used as a semantic middleware in a variety of solutions, all being used in productive applications. Hence, in the meantime, OntoBroker has thousands of installations. Based on OntoBroker's powerful features, ontoprise provides several solutions, for example, the SemanticGuide, SemanticXpress, and SemanticIntegrator. These solutions bring additional value to business processes at the customer site.

References

[1] J. Angele, M. Erdmann, D. Wenke: *Ontology-based Knowledge Management in Automotive Engineering Scenarios*. In (Martin Hepp, Pieter De Leenheer, Aldo de Moor, York Sure. Eds.): Ontology Management: Semantic Web, Semantic Web Services, and Business Applications, ISBN 978-0-387-69899-1, Springer, 2007.

[2] J. Angele, M. Gesmann: *Semantic Information Integration with Software AGs Information Integrator*. In (Thomas Eiter; Knowledge Web.; et al Eds.): Proceedings of the 2nd RuleML conference, Las Alamitos, Calif. : IEEE Computer Society, 2006.

[3] J. Angele, M. Kifer, G. Lausen: *Ontologies in F-Logic*. In (S. Staab, R. Studer eds.): Handbook on Ontologies. International Handbooks on Information Systems, Springer Verlag, Second Edition, 2009, 45 -

[4] J. Angele, E. Mönch, H. Oppermann, H. Rudat, H.-P. Schnurr: *Customer Service accelerated by Semantics*. In: Proceedings of the industrial track of the Fifth International Semantic Web Conference (ISWC2006), Athens, USA, November 7-9, 2005.

[5] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman: *Magic sets and other strange ways to implement logic programs.* In Proceedings Fifth ACM Symposium on Principles of Database Systems, 1986, 1-15.

[6] Beeri C., Ramaksishnan R. *On the power of magic*. The Journal of Logic Programming, 10:255-300, January 1991.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MITPress and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 18:B-Trees, 434–454.

[8] Bizer, Christian; Lehmann, Jens; Kobilarov, Georgi; Auer, Soren; Becker, Christian; Cyganiak, Richard; Hellmann, Sebastian (September 2009). DBPedia – A crystallization point for the Web of Data, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3): 154–165. ISSN 1570-8268

[9] Friedland N., Allen P., Matthews G., Witbrock M., Baxter D., Curtis J., Shepard B., Miraglia P., Angele J., Staab S., Moench E., Oppermann H., Wenke D., Israel D., Chaudhri V., Porter B., Barker K., Fan J., Chaw S., Yeh P., Tecuci D., Clark P.. Project Halo: *Towards a Digital Aristotle*. In AI Magazine, vol. 25, no. 4, winter 2004, 29-4.

[10] Segaran T., Evans C., Taylor J.. Programming the Semantic Web, O'Reilly, 2009, ISBN:0596153813 9780596153816

[11] A. Van Gelder, K.A. Ross and J.S. Schlipf. *The Well-Founded Semantics for General Logic Programs*. Journal of the ACM 38(3) pp. 620—650, 1991.

[12] Kifer M., Lausen, and Wu (1995). *Logical foundations of object-oriented and framebased languages*. Journal of the ACM, 42; (1995), 741–843.

[13] Kifer and Lozinskii (1986). *A framework for an efficient implementation of deductive databases*. In Proceedings of the 6th Advanced Database Symposium, Tokyo, August (1986), 109–116.

[14] J.W. Lloyd: *Foundations of Logic Programming*, 2nd Editon. Springer-Verlag, Berlin, 1987.

[15] Tom Heath and Christian Bizer (2011) *Linked Data: Evolving the Web into a Global Data Space* (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool.

[16] Hitzler P., Krötzsch M., Rudolph S. Foundations of Semantic Web Technologies, Chapman & Hall, 2009, ISBN: 9781420090505

[17] Rahm E., Bernstein P. (2001). *A survey of approaches to automatic schema matching*, VLDB Journal 10(4):334-350, 2001

[18] J. D. Ullman: *Principles of Database and Knowledge-Base Systems*, vol II. Computer Sciences Press, Rockville, Maryland, 1989.

[19] L. Vieille: *Recursive Axioms in Deductive Databases: The Query/Subquery Approach*. In Proceedingsof First International Conference on Expert Database Systems, Charleston, 1986, 179-194.

[20] Senlin Liang, Paul Fodor, Hui Wan, Michael Kifer: *OpenRuleBench: An Analysis of the Performance of Rule Engines.* In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, Wolfgang Nejdl (Eds.): Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009. ACM 2009, ISBN 978-1-60558-487-4.

[21] J. Frohn, R. Himmer, G. Lausen, W. May, C. Schlepphorst: *Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective*. In Information Systems, vol. 23, no. 8, 1998, p. 589-613.

[22] D. Berardi, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, J. Su, S. Tabet: *SWSL: Semantic Web Services Language*. Semantic Web Services Initiative, April 2005.

[23] G. Yang, M. Kifer, C. Zhao: *FLORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web*. In International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)", Springer LNCS, vol. 2888, p. 671-688.