Supporting Application Development in the Semantic Web

DANIEL OBERLE, STEFFEN STAAB, RUDI STUDER, and RAPHAEL VOLZ
University of Karlsruhe, Germany
Institute for Applied Informatics and Formal Description Methods (AIFB)

The Semantic Web augments the current WWW by giving information a well-defined meaning, better enabling computers and people to work in cooperation. This is done by adding machine understandable content to web resources. Such added content is called metadata, whose semantics is provided by referring to an ontology—a domain's conceptualization agreed upon by a community. The Semantic Web relies on the complex interaction of several technologies involving ontologies. Therefore, sophisticated Semantic Web applications typically comprise more than one software module. Instead of coming up with proprietary solutions, developers should be able to rely on a generic infrastructure for application development in this context. We call such an infrastructure Application Server for the Semantic Web whose design and development are based on existing Application Servers. However, we apply and augment their underlying concepts for use in the Semantic Web and integrate semantic technology within the server itself. The article discusses requirements and design issues of such a server, presents our implementation KAON SERVER and demonstrates its usefulness by a detailed scenario.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Extensibility*; D.2.11 [**Software Engineering**]: Software Architectures; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms: Design, Languages, Management

Additional Key Words and Phrases: Application server, extensibility, interoperation, KAON, KAON SERVER, middleware, ontology, reuse, semantic middleware, Semantic Web, Wonder-Web

1. INTRODUCTION

The Internet was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines

The KAON project is a meta-project carried out at the Institute AIFB, University of Karlsruhe and the Research Center for Information Technologies (FZI), Karlsruhe (http://kaon.semanticweb.org). The KAON SERVER is developed in the context of WonderWeb, an EU Information Society Technologies (FZI) funded project (EU IST 2001 33052, http://wonderweb.semanticweb.org).
Authors' address: University of Karlsruhe, Institute AIFB, 76128 Karlsruhe, Germany; email: {oberle,staab,studer,volz}@aifb.uni-karlsruhe.de; http://www.aifb.uni-karlsruhe.de/WBS.

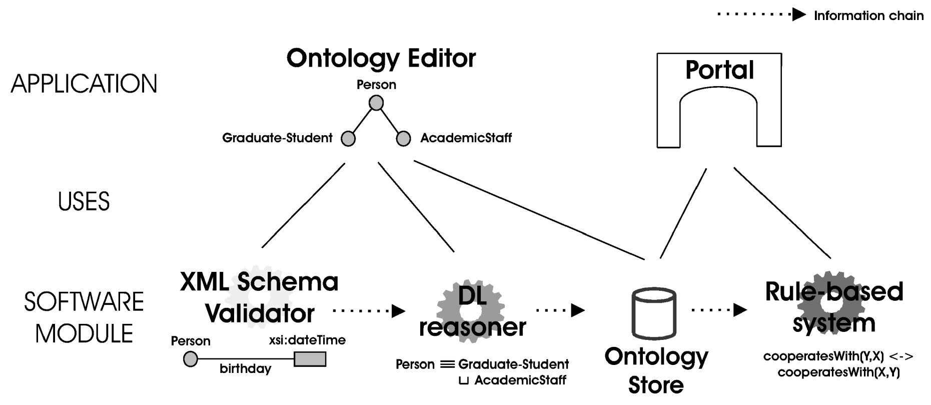


Fig. 1. Information flow in the research and academia example.

would be able to participate and help. One of the major obstacles to this is the fact that most information on the WWW is designed for human consumption, and even if it were derived from a database with well defined meanings (in at least some terms), the meaning of the data would not be evident to a web application system.[1]

The way out of this shortcoming is the Semantic Web, which augments the current WWW by giving information a well-defined meaning, better enabling computers and people to work in cooperation [Fensel et al. 2003]. This is done by adding machine understandable content to Web resources. The results of this process are metadata, usually circumscribed as data about data, that can be a simple statement like “site x’s author is Daniel Oberle.” Descriptions like this are given their semantics by referring to an ontology, a domain’s conceptualization agreed upon by a community [Gruber 1993]. In the statement above, we could express that “Daniel Oberle” is a “PhD-Student” and “PhD-Student” is a specialization of “Graduate-Student.”

Ontologies serve various needs in the Semantic Web, like storage or exchange of data corresponding to an ontology, ontology-based reasoning or ontology-based navigation. Building a complex Semantic Web application, one may not rely on a single software module to deliver all of these different services. The developer of such a system would rather want to easily combine different—preferably existing—software modules.

An example would be the domain ontology for an application supporting research and academia. Such an application, manages information about a university’s staff, their publications, students, and courses. Its ontology can be easily expressed by Semantic Web languages and constructed by a corresponding editor (cf. Figure 1). There will be properties of concepts that require structured XML Schema data types [Biron and Malhotra 2001] whose correctness can be checked by a validator. A description logic reasoner is usually applied for semantic validation of the ontology. An ontology store saves the ontology and can be reused by a research and academia portal. The latter

[1]Semantic Web Roadmap, Tim Berners-Lee, http://www.w3.org/DesignIssues/Semantic.html.

may exploit a rule-based inference engine that is capable of handling large amounts of instances and deduction of additional information by rules.

So far, such integration of ontology-based modules has had to be done in an ad hoc manner, generating a one-off endeavour, with little possibilities for reuse and future extensibility of individual modules or the overall system.

This article is about an infrastructure that facilitates plug'n'play engineering of ontology-based modules and thus, the development and maintenance of comprehensive Semantic Web applications, an infrastructure that we call *Application Server for the Semantic Web* (*ASSW*). It facilitates reuse of existing modules, for example, ontology stores, editors, and inference engines. It combines means to coordinate the information flow between such modules, to define dependencies, to broadcast events between different modules and to translate between ontology-based data formats.

Existing Application Servers [Mohan et al. 2001] are frequently viewed as part of a three-tier application, consisting of a front-end, for example, web browser-based graphical user interface, a middle-tier business logic application or set of applications, and a third-tier, back-end, database, and transaction server. The Application Servers are the middleware between browser-based front-ends and back-end databases and legacy systems.[2] Typically their functionality comprises connectivity and security, flexible handling of software modules, monitoring, transaction processing and so on.

The Application Server for the Semantic Web will help to put the Semantic Web into practice because it adopts this idea for easier development of Semantic Web applications. In our scenario (cf. Figure 1), such a server would facilitate the building of the portal, for example, by enabling the reuse of required software modules. The portal as a whole would become part of the Semantic Web because it publishes and consumes ontology-based metadata. The fact that an Application Server for the Semantic Web is used is merely a relief for the developer. The greatest benefit is in the building of complex applications. We do not envision every Semantic Web application being built by such a server, analogous to applications in general that are not all developed on the basis of application servers. Apart from facilitating application development, our server uses semantic technology, which allows us to achieve an even greater functionality than existing Application Servers.

The article is structured as follows: First, we provide a brief overview of the Semantic Web, in particular of its languages, in Section 2. We list requirements for an Application Server for the Semantic Web in Section 3. Sections 4 and 5 describe the design decisions that directly meet important requirements, namely extensibility, and ones that call for the semantic enhancement of the server. The conceptual architecture is then provided in Section 6. Section 7 presents the KAON SERVER, a particular Application Server for the Semantic Web, which was developed as part of the EU-funded WonderWeb project. In Section 8 we elaborate on the scenario depicted in Figure 1: how it can be implemented by making use of the KAON SERVER. Related work and conclusions are given in Sections 9 and 10, respectively.

[2]cf. searchDatabase.com, http://searchdatabase.techtarget.com.

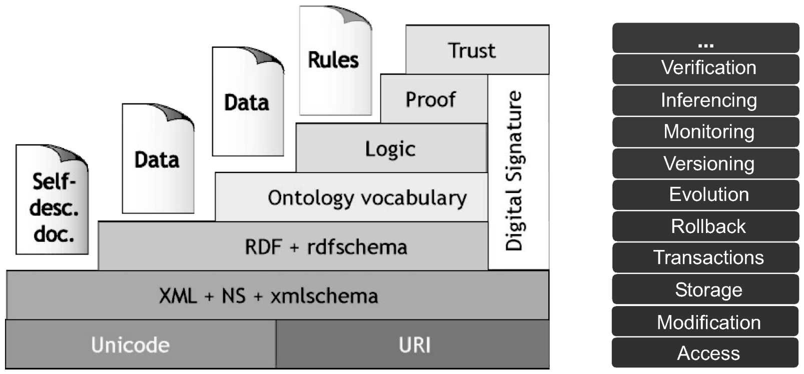


Fig. 2. Static and dynamic aspects of the Semantic Web layer cake.

2. THE SEMANTIC WEB

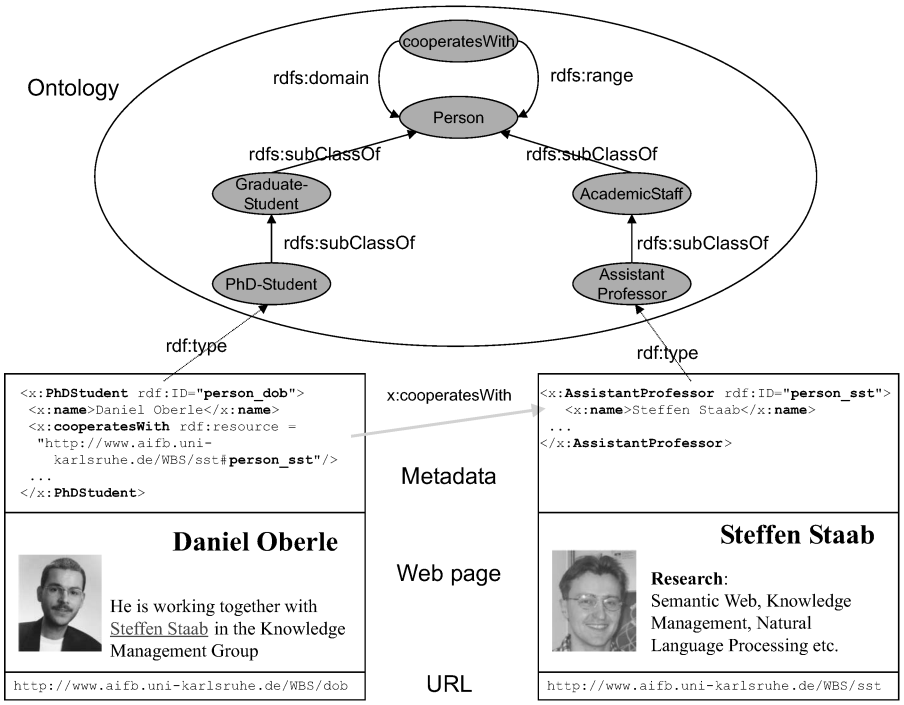
In this section we want to introduce the reader to the architecture and languages of the Semantic Web that we support in our Application Server. The left hand side of Figure 2 shows the static part of the Semantic Web,[3]: its language layers. Unicode, the URI and namespaces (NS) syntax and XML are used as a basis. XML's role is limited to that of a syntax carrier for data exchange. XML Schema [Biron and Malhotra 2001] defines simple data types like string, date, or integer.

The Resource Description Framework (RDF) may be used to make simple assertions about Web resources or any other entity that can be named. A simple assertion is a statement that an entity has a property with a particular value, for example, that this article has a title property with value “Supporting application development in the Semantic Web.” RDF Schema extends RDF by class and property hierarchies that enable the creation of simple ontologies.

RDF and RDFS are already standardized by the World Wide Web Consortium (W3C) [Lassila and Swick 1999]. Figure 3 depicts an example for ontology-based metadata in the domain of research and academia. The ontology features a concept Person, with specializations such as Graduate-Student, PhD-Student as well as AcademicStaff and AssistantProfessor. RDFS' modelling primitives formalize the domain description as RDF statements, for example, “PhD-Student rdfs:subClassOf Graduate-Student.” CooperatesWith is a symmetric property defined on Person by using the rdfs:domain and rdfs:range primitives.

XML serializations of RDF statements can be added to Web resources like the homepages of PhD-Student “Daniel Oberle,” and AssistantProfessor “Steffen Staab.” The metadata formally define both as instances of the ontology's concepts through the rdf:type primitive. Relationships are provided with formal semantics by referring to the ontology. A search engine could later infer that “Steffen Staab” also cooperates with “Daniel Oberle”, because the property is defined to be symmetric.

[3]Semantic Web—XML 2000, Tim Berners-Lee, http://www.w3.org/2000/Talks/1206-xml2k-tbl/Overview.html.



```
<x:PhDStudent rdf:ID="person_dob">
 <x:name>Daniel Oberle</x:name>
 <x:cooperatesWith rdf:resource =
  "http://www.aifb.uni-
    karlsruhe.de/WBS/sst#person_sst"/>
 ...
</x:PhDStudent>
```

```
<x:AssistantProfessor rdf:ID="person_sst">
   <x:name>Steffen Staab</x:name>
 ...
</x:AssistantProfessor>
```

Daniel Oberle

He is working together with Steffen Staab in the Knowledge Management Group

http://www.aifb.uni-karlsruhe.de/WBS/dob

Steffen Staab

Research: Semantic Web, Knowledge Management, Natural Language Processing etc.

http://www.aifb.uni-karlsruhe.de/WBS/sst

Fig. 3. Semantic Web example.

The Ontology layer features the Web Ontology Language (OWL [van Harmelen et al. 2003]). OWL is a family of richer ontology languages consisting of OWL Lite, DL and Full. They augment RDF Schema and are based on the descriptions logics (DL) paradigm [Baader et al. 2003]. OWL Lite is the simplest of these. It is a limited version of OWL DL, enabling simple and efficient implementation. OWL DL is a richer subset of OWL Full for which reasoning is known to be decidable so complete reasoners may be constructed, though they will be less efficient than an OWL Lite reasoner. OWL Full is the full ontology language, which is however undecidable.

The Logic layer[4] will provide an interoperable language for describing the sets of deductions one can make from a collection of data—how, given an ontology-based information base, one can derive new information from existing data.

The Proof language will provide a way of describing the steps taken to reach a conclusion from the facts. These proofs can then be passed around and verified, providing short cuts to new facts in the system without having each node conduct the deductions themselves.

The Semantic Web's vision is that once all these layers are in place, we will have an environment in which we can trust that the data we are seeing, the

[4]A better description of this layer would be "Rule layer," as the Ontology layer already features a logic calculus with reasoning capabilities. We here use the naming given by Tim Berners-Lee in his roadmap.

deductions we are making, and the claims we are receiving have some value. The goal is to make a user's life easier by the aggregation and creation of new, trusted information over the Web [Dumbill 2001]. The standardization process has currently reached the Ontology layer—Logic, Proof and Trust layers aren't specified yet.

The right hand side of Figure 2 depicts the Semantic Web's dynamic aspects that apply to data across all layers. Often, the dynamic aspects are neglected by the Semantic Web community, however, from our point of view, they are essential for putting the Semantic Web into practice. Transactions and rollbacks of Semantic Web data operations should be possible, following the well-known ACID properties (atomicity, consistency, independence, durability) of Database Management Systems (DBMS). Evolution and versioning of ontologies are important aspects, because ontologies usually are subject to change (cf. Stojanovic at al. [2002a]). As in all distributed environments, monitoring of data operations becomes necessary for security reasons. Finally, reasoning engines must be applied for the deduction of additional facts[5] as well as for semantic validation.

3. REQUIREMENTS

The requirements for an Application Server for the Semantic Web can be divided into five groups. First, such a server should respond to the static aspects of the Semantic Web layer cake. Second, the dynamic aspects result in another group of requirements: finding, accessing, modifying and storing of data, transactions and rollbacks, evolution and versioning, monitoring as well as inferencing and verification. Third, clients may want to remotely connect to the server by different protocols and must be properly authorized. Hence, another group deals with connectivity and security. Fourth, the system is expected to facilitate an extensible and reconfigurable infrastructure. This set of requirements therefore deals with flexible handling of modules. The last group subsumes requirements that are associated with semantic descriptions of software modules. In the following subsections we will investigate the groups organized in common requirements that hold for every Application Server (Subsection 3.1), requirements that are specific to the Semantic Web (Subsection 3.2) and requirements that call for the semantic enhancement of the server itself (Subsection 3.3).

3.1 Common Requirements

3.1.1 *Connectivity and Security.*

—*Connectivity.* An Application Server for the Semantic Web should enable loose coupling, allowing access through standard protocols, as well as close coupling by embedding it into an application. In other words, a client should be able to use the system locally and connect to it remotely.

[5]E.g. if "cooperatesWith" is defined as a symmetric property in OWL DL between persons, a reasoner should be able to deduce that B cooperatesWith A, given the fact that A cooperatesWith B.

—*Ease of use.* A developer does not want to expend extra effort in connecting to and using a software module when an Application Server for the Semantic Web is applied. A software module ought to be accessed seamlessly.

—*Offering functionality via different communication protocols.* There might be the need to offer a software module's functionality via another communication protocol. The Application Server for the Semantic Web should be able to offer its methods via separate web services enhanced by automatically generated DAML-S [Burstein et al. 2002] descriptions, for instance via peer or agent protocols.

—*Security.* Guaranteeing information security means protection against unauthorized disclosure, transfer, modification, or destruction, whether accidental or intentional. To realize it, any operation should only be accessible by properly authorized clients. Proper identity must be reliably established by employing authentication techniques. Confidential data must be encrypted for network communication and persistent storage. Finally, means for monitoring (logging) of confidential operations should be present.

3.1.2 *Flexible Handling of Modules.*

—*Extensibility.* The need for extensibility applies to most software systems. It is a principle of software engineering to avoid system changes when additional functionality is needed in the future. Hence, extensibility is also desirable for an Application Server for the Semantic Web. In addition, such a server has to deal with the multitude of layers and data models in the Semantic Web that lead to a multitude of software modules, for example, XML parsers or validators that support the XML Schema datatypes, RDF stores, tools that map relational databases to RDFS ontologies, ontology stores, and OWL reasoners. Therefore, extensibility regarding new data APIs and corresponding software modules is an important requirement for such a server.

—*Integrating existing functionality via different communication protocols.* The Semantic Web will be populated by different kinds of software entities, for example, web services, peers, or agents. A developer might want to integrate the ones needed to build an application. That would lift the responsibility of handling different protocols from the developer and enable them to be included in a transaction, for example.

—*Dependencies.* The server should enable the expression of dependencies between different software modules; for instance, the setting up of event listeners between modules. Another example would be the management of a dependency like "module A is needed for module B."

3.2 Semantic Web Specific Requirements

3.2.1 *Requirements Stemming from the Semantic Web's Static Part.*

—*Language Support.* A trivial requirement is the support of all the Semantic Web's ontology and metadata standards. An Application Server for the Semantic Web has to be aware of RDF, RDFS, OWL as well as future languages that will be used to specify the logic, proof, and trust layers.

—*Semantic Interoperation.* We use the term semantic interoperation in the sense of translating between different ontology languages with different semantics. At the moment, several ontology languages populate the Semantic Web. We have already mentioned RDFS, OWL Lite, OWL DL and OWL Full apart from proprietary ones. Usually, ontology editors and stores focus on one particular language and are not able to work with others. Hence, an Application Server for the Semantic Web should enable translation between different languages and semantics [Grosof et al. 2003; Bennett et al. 2002].

—*Ontology Mapping.* In contrast to semantic interoperation, ontology mapping translates between different ontologies of the same language. Mapping may become necessary as web communities usually have their own ontology and could use ontology mapping to facilitate data exchange [Noy and Musen 2000; Handschuh et al. 2003].

—*Ontology Modularization.* Modularization is an established principle in software engineering. It has to be considered also for ontology engineering as the development of large domain ontologies often includes the reuse of several existing ontologies. E.g. top-level ontologies might be used as a starting point. Hence, an Application Server for the Semantic Web should provide means to fulfill that requirement [Borgida and Serafini 2002; Bozsak et al. 2002].

3.2.2 *Requirements Stemming from the Semantic Web's Dynamic Part.*

—*Finding, Accessing, Modifying and Storing of Ontologies.* Semantic Web applications like editors or portals have to access, modify and finally store ontological data. In addition, the development of domain ontologies often requires other ontologies as starting points. Examples are Wordnet [Miller et al. 1990] or top-level ontologies for the Semantic Web [Oltramari et al. 2002]. These could be stored and offered by the server to editors.

—*Transactions and Rollback.* The dynamic aspects, transactions, and rollbacks (cf. Figure 2) lead to further requirements. All updates to the Semantic Web data must be done within transactions assuring the properties of atomicity, consistency, isolation (concurrency) and durability (ACID) [Ullman 1988]. Although in general transactions can be considered as a common requirement, they can become specific as the Semantic Web languages require special handling.

—*Evolution and Versioning.* Ontologies are applied in dynamic environments with changing application requirements (cf. [Stojanovic et al. 2002a]). To fulfill the changes, the underlying ontology must often be evolved as well. Database researchers distinguish between schema evolution and schema versioning [Noy and Klein 2002]. Schema evolution is the ability to change a schema of a populated database without loss of data (i.e. providing access to both old and new data through the new schema). Schema versioning is the ability to access all the data (both old and new) through different version interfaces. For ontologies, however, a distinction between evolution, which allows access to all data only through the newest schema, and versioning, which allows access to data through different versions of the schema, cannot be made. This is due to the fact that multiple versions of the same ontology are bound to exist and must be supported. Hence, ontology evolution and versioning can be combined into a single concept defined as the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology.

—*Monitoring.* See "Connectivity and Security" in Subsection 3.1.

—*Inferencing and Verification.* Reasoning engines are core components of semantics-based applications and can be used for several tasks like semantic validation and deduction. An Application Server for the Semantic Web should provide access to such engines, which can deliver the reasoning services required.

3.3 Requirements for Smantic Enhancement of the Application Server

This last group contains requirements that call for the semantic enhancement of the Application Server for the Semantic Web.

—*Discovery of Software Modules.* For a client, there should be the possibility of stating precisely what it wants to work with, for example, an RDF store that holds a certain RDF model and offers a transaction concept. Hence, means for intelligent discovery of software modules are required.

—*API Discovery.* The developer may want to find a certain API (application progammer's interface) independent of a concrete software module that implements it. Preferably, the developer wants to specify high level details and get a comprehensive list of existing APIs that perform desired tasks. As there can be several related and overlapping APIs, the system should recommend the one that fits best.

—*Classification of Software Modules.* The conceptualization used by the descriptions should also facilitate the classification of new software modules. Mostly, their APIs fulfill overlapping tasks, for example, an ontology store offers both inferencing and storing.

—*Implementation Tasks.* The Application Server for the Semantic Web itself should take advantage of semantic descriptions. For example, when loading a module, its description could express required libraries and other modules as properties. Libraries might rely on others, the same holds for components. By defining corresponding properties transitively, the task of automatically infering all necessary libraries and components is simplified.

While the common requirements are met by most of the existing Application Servers, Semantic Web specific requirements and the ones that call for the semantic enhancement of the server itself are clearly beyond state-of-the-art. In the following Sections, 4 to 6, we develop an architecture that is a result of the requirements put forward in this section. Thereafter we present the implementation details of our Application Server for the Semantic Web called KAON SERVER.

4. COMPONENT MANAGEMENT

Due to the requirement for Extensibility, we decided to use the Microkernel design pattern. The pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core, the Microkernel, from extended functionality and application-specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration [Buschmann et al. 1996].

In our setting, the Microkernel's minimal functionality offers simple management operations: starting, initializing, monitoring, combining and stopping of software modules as well as dispatching of messages between them. This approach requires software modules to conform to a management interface so that they can be managed by the Microkernel. Conformity is accomplished by *making existing software deployable*: bringing existing software into the particular infrastructure of the Application Server for the Semantic Web. This means that existing software is wrapped such that it can be managed by the Microkernel. Thus, a software module becomes a *deployed component*. The word *deployment* stems from service management and service oriented architectures where it is a technical term [Bishop 2002]. We adopt and apply it in our setting. It describes the process of registering a component to the Microkernel with possible initialization and start.

Apart from the cost of making existing software deployable, a drawback of this approach is that performance will suffer slightly in comparison to stand-alone use, since a request has to first pass through the Microkernel (and possibly the network). A client that wants to make use of a deployed component's functionality talks to the Microkernel, which in turn dispatches requests.

The Microkernel and component approach delivers several benefits. By making existing functionality, like RDF stores, inference engines and so on deployable, one is able to manage them in a centralized infrastructure. As a result, we are able to deploy and undeploy components ad hoc, reconfigure, monitor and possibly distribute them dynamically. Proxy components can be developed for software that cannot be made deployable, for example, because it has been developed for a particular operating system. Throughout the article, we will show further advantages; among them:

—Enabling a client to discover the component it is in need of (cf. Section 5).
—Definition of dependencies between components (cf. Section 6).
—Easy integration of modularization, transactions, evolution and semantic interoperation by interceptors (cf. Section 6).

Thus, we have responded to the requirement for extensibility. In the following, we discuss how the requirements that call for the semantic enhancement of the server are met.

5. COMPONENT DESCRIPTION

This section responds to the group of "Requirements for semantic enhancement of the Application Server." All of them implicitly call for a formal

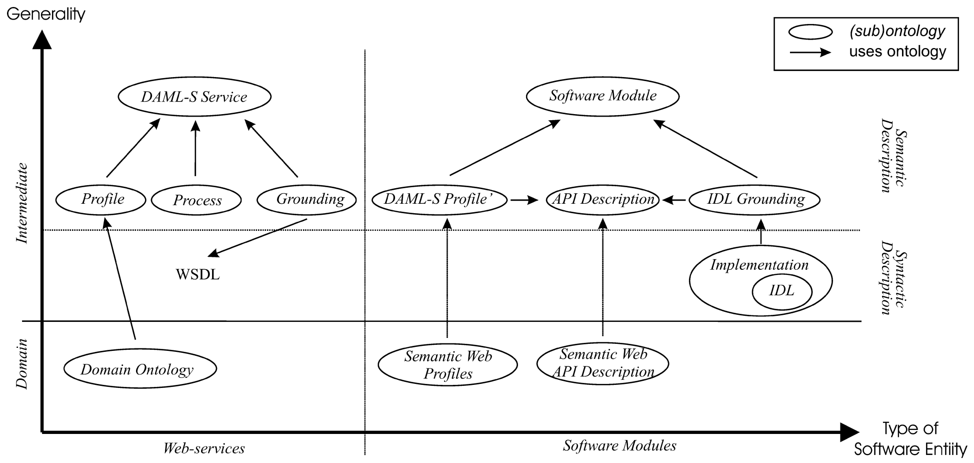


Fig. 4. Overview of ontologies.

conceptualization providing the means for component description. Hence, the definition and usage of an ontology is the first design choice derived here.

As pointed out in Section 4, all components are managed in a central infrastructure, namely the Microkernel. In order to allow a client to discover the components it needs, we have to distinguish between them. Thus, the second design choice is a registry that stores descriptions of all deployed components. The Microkernel and component approach allows us to implement the registry itself as a component.

For the ontology's design we tried to stay as close as possible to DAML-S [Burstein et al. 2002] for it is an accepted standard that has been investigated for a long time and has a sound basis. DAML-S is an initiative of the Semantic Web community to facilitate automatic discovery, invocation, composition, interoperation and monitoring of web services through their semantic description. Expressed in DAML+OIL, it is conceptually divided into three subontologies for specifying *what a service does* (Profile), *how the service works* (Process) and *how the service is implemented* (Grounding). The existing grounding enables aligning the semantic specification with implementation details described using WSDL [Christensen et al. 2003], the industry standard for web service description.

Although DAML-S serves as a good starting point for our ontology, the main difficulty was in the type of software entities to be described. While DAML-S describes web services, our goal is to describe components and their APIs. As a result some parts of DAML-S were not reusable. Figure 4 presents all the subontologies in DAML-S in comparison to ours. The following discussion is organized using our design principles (cf. also Oberle at al. [2003]).

Modularity. Modularity enables easy reuse of specifications and extensibility of the ontology. An important issue is the size of the reusable parts. For example, because a *Profile* instance contains a lot of information, which is often very specific, such as the contact information of the providers, it is less likely that this instance will be reused by any other description (except if it is provided by the same company). Therefore a coarser granularity (less information per concept) increases the chance of reusability. We have applied this principle by making an effort to centralize related content to a certain concept whose instance can be reused at description time. We decided to group together chunks of information that are most likely to be reused. Also, we have grouped this information in small ontologies that are used by other subontologies.

Semantic vs. Syntactic Descriptions. We have adopted the separation between semantic and syntactic descriptions. Therefore we are able to map a certain semantic description to several syntactic descriptions if the same semantic functionality is accessible in different ways, and vice versa. However, we modified some of the DAML-S subontologies as follows:

—We have kept the DAML-S *Profile* ontology for specifying semantic information about the described modules and extended it with a few concepts for describing APIs at the conceptual level that are grouped in a small subontology called *API Description*.
—We did not use the *Process* ontology since we have not been interested in the internal working of the modules in the first step.
—We formalized a subset of *IDL*[6] terms and use them to describe the syntactic aspects of APIs in an *Implementation* ontology, which is further discussed below.
—As a consequence of the above changes, we could not reuse the existing DAML-S *Grounding* and wrote an *IDL Grounding* ontology.

Generic vs. Domain Knowledge. Our approach for building the ontology can be described in terms of the ONIONS [Gangemi et al. 1999] ontology development methodology, which advises grouping knowledge with different generality in generic, intermediate and domain ontologies. We have built two domain ontologies: *Semantic Web Profiles* and *Semantic Web API Description*, which specify the type of existing Semantic Web software modules at coarse and fined grained level, respectively. As depicted in Figure 4, all the other subontologies are categorized in the intermediate-level. For the top- or generic-level, which is not shown in the Figure, we applied DOLCE [Oltramari et al. 2002].

We would like to conclude this section with a closer look at the *Implementation* subontology. It is primarily used to facilitate component discovery for the client and it is important for further understanding. Its core taxonomy is shown below. We start with the definition of a component and then refine it.

Component. Software module that is deployable to the Microkernel.
System Component. Component providing functionality for the Application Server for the Semantic Web itself, for example, the registry.
Functional Component. Component that is of interest to the client and can be discovered. Ontology-related software modules become functional components by making them deployable, for example, RDF stores.

[6]Interface Description Language, cf. http://www.omg.org.

External Module. An external module cannot be deployed directly as it may be programmed in a different language, live on a different computing platform, and so forth. It equals a functional component from a client perspective by having a proxy component deployed that relays communication to the external module.

Proxy Component. Special type of functional component that manages the communication to an external module. Examples are proxy components for inference engines, like FaCT [Horrocks 1998].

Each component can have attributes like the name of the interface it implements, connection parameters or several other low-level properties. Besides, we express associations between components. Associations can be dependencies between components, for example, an ontology store component can rely on an RDF store for actual storage, or event listeners. Associations will later be put in action by an association management system component (cf. Section 6).

So far we have discussed the requirement for Extensibility and Requirements for semantic enhancement of the Application Server, which led to fundamental design decisions. The next section continues with an overall view of the conceptual architecture.

6. CONCEPTUAL ARCHITECTURE

When a client connects to the Application Server for the Semantic Web it either needs to discover the required functional components or to deploy them itself. In the first case, the client uses the registry to find a deployed functional component fulfilling its prescriptions and is returned a reference. From then on, the client can seamlessly work with the functional component by surrogates that handle the communication over the network. On the server side, the counterpart to the surrogate is a connector component. It maps requests to the Microkernel's methods. All requests pass the Microkernel, which dispatches them to the appropriate functional component. While dispatching, a request can be modified by interceptors that may deal with auditing, for instance. Finally, the response passes the Microkernel again and finds its way to the client through the connector and the surrogate. The following paragraphs will explain the architecture depicted in Figure 5.

Surrogates. Surrogates (not shown in Figure 5) are objects embedded in the client application that relieve the developer of the communication details similar to stubs in CORBA[7] (cf. requirement "Ease of use"). They offer the same API as a particular component and relay communication to any connector, which in turn passes the request to the respective functional component through the Microkernel.

Connectors. Connectors are system components. They send and receive requests and responses over the network. Aside from the option to connect locally,

[7]Common Object Request Broker Architecture: Core Specification, http://www.omg.org/technology/documents/formal/corbaiiop.htm.

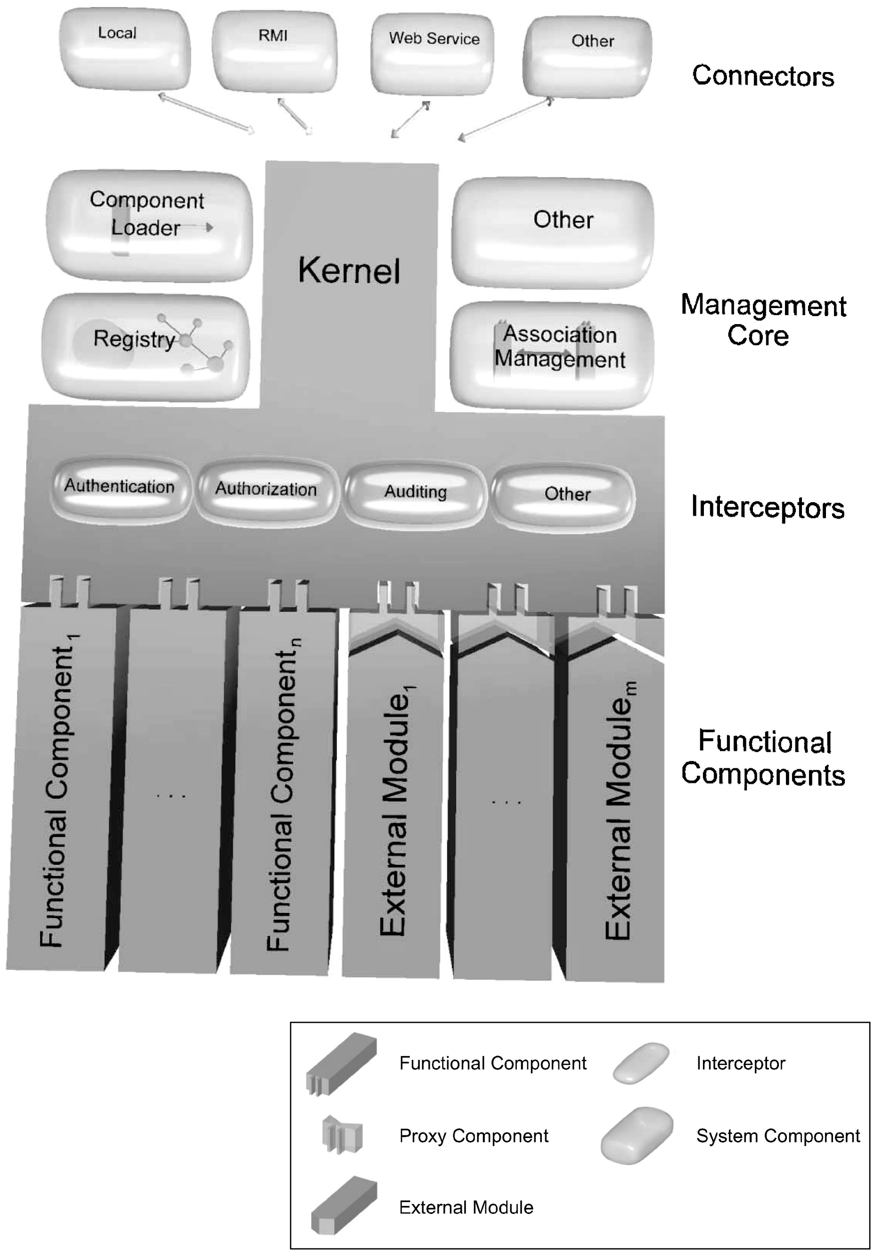


Fig. 5. Conceptual architecture.

further possibilities exist for remote connection: for example, ones that offer access via Java Remote Method Invocation (RMI), or ones that offer asynchronous communication. Connectors also enable publishing components' methods as separate web services with automatically generated DAML-S descriptions out of the registry. Offering the functionality with peer or agent protocols is also possible (cf. requirement "Offering functionality via different communication protocols").

Management Core. The Management Core comprises the Microkernel (also called management kernel or simply kernel in the following) as well as several system components. It is necessary to deal with the discovery, allocation and loading of components. The registry, a system component, manages descriptions of the components and facilitates the discovery of a functional component for a client, and the component loader facilitates the deployment process. It takes a component description as argument, handles the deployment, enters the description in the registry and applies the association management if necessary. The latter is another system component that puts ontological associations between components into action (cf. requirement "Implementation tasks"). For example, event listeners can be put in charge so that a component A is notified when B issues an event, or a component may only be undeployed if others don't rely on it. System components can be deployed and undeployed ad hoc, so extensibility is also given for the Management Core. Further components are possible, for example, a cascading component that offers seamless access to the components deployed in another Application Server.

Interceptors. Interceptors are software entities that monitor a request and modify it before the request is sent to the component. A component can be deployed with a stack of arbitrary interceptors. Security aspects are met by interceptors that guarantee that operations offered by functional components in the server are only available to appropriately authenticated and authorized clients. Sharing generic functionality such as security, logging, or concurrency control requires less work than developing individual component implementations. For example, when a component is being restarted, an interceptor can block and queue incoming requests until the component is available again. Transactions, modularization and evolution spanning several ontology stores may also be realized by interceptors.

Functional Components. RDF stores, ontology stores and so on, are finally deployed to the management kernel as functional components (cf. Section 4). In combination with the component loader, the registry can start functional components dynamically in response to client requests. Proxy components (which are conceptually subsumed by functional components, cf. Section 5) can be developed for external modules, but also for web services, peers or agents. That allows a developer to access them conveniently by surrogates instead of handling several other protocols. In addition, interceptors can be applied on top, so that, for example, a web service might be part of a transaction along operations of a deployed ontology store.

Table I shows where the requirements put forward in Section 3 are reflected in the architecture. Due to the Microkernel design pattern the architecture basically consists of the Microkernel itself, components, interceptors and surrogates. Components are conceptually specialized into system, functional, and proxy components to facilitate the discovery for the application developer. Table I only shows connectors as a subconcept of system component as well

Table I. Dependencies between Requirements (cf. Section 3) and Architecture

		Components							
Requirement\Design Element	Kernel	Connectors	Registry	Component Loader	Association Management	Functional Components	Proxy Components	Interceptors	Surrogates
Connectivity		×							
Ease of use									×
Offering functionality via protocols		×							
Security								×	
Extensibility	×	×		×		×	×	×	
Integrating functionality via protocols							×		
Dependencies					×				
Language Support						×			
Semantic Interoperation						×		×	
Ontology Mapping						×			
Ontology Modularization						×		×	
Finding, Accessing, Storing of ontologies						×			
Transactions and Rollback						×		×	
Evolution and Versioning						×		×	
Monitoring						×		×	
Inferencing and Verification						×			
Discovery of software modules			×						
API discovery			×						
Classification of software modules			×						
Implementation tasks			×	×	×				

as registry, component loader and association management, which are particular system components. Functional and proxy components are represented in one column each.

For Semantic Interoperation, interceptors could translate between Semantic Web ontology languages. For example, if a client wants to talk in DAML+OIL to an OWL ontology store, an interceptor could be registered that automatically translates the request. However, Semantic Interoperation could also be realized by dedicated functional components. Ontology Modularization, Transactions and Rollback, Evolution and Versioning as well as Monitoring are different in that they can all be implemented within one functional component. A comprehensive ontology store might offer means for transactions, for instance. Interceptors, on the other hand, could realize those mechanisms on top of several components. Akin to what a transaction monitor does with several database systems, an interceptor would be capable of realizing transactions spanning several ontology stores.

7. IMPLEMENTATION

This section presents our implementation of an Application Server for the Semantic Web, called KAON SERVER, which is part of the KAON[8] Tool suite

[8]Karlsruhe Ontology and Semantic Web Tool suite, http://kaon.semanticweb.org.

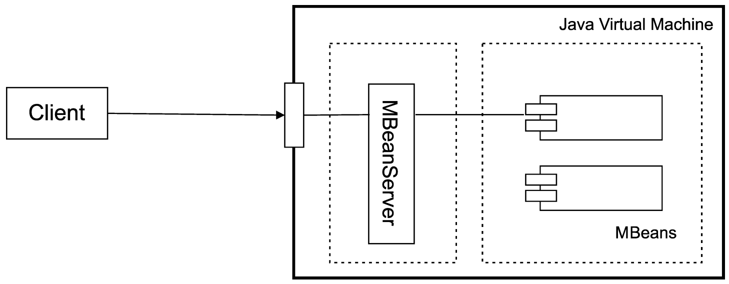


Fig. 6. JMX Management architecture.

[Bozsak et al. 2002]. The latter includes software modules allowing easy ontology creation and management, as well as building ontology-based applications in Java. All of the modules have been made deployable.

The KAON SERVER architecture reflects the conceptual architecture presented in the previous section. In the following, an in-depth description is given. We will start with the Management Core in 7.1 as it is necessary to understand Connectors in 7.2, Interceptors in 7.3 and Functional Components in 7.4.

7.1 Management Core

The Management Core of an Application Server for the Semantic Web consists of the management kernel, the component loader, registry and association management system components. We will outline all of their implementations in the subsections below.

7.1.1 *Kernel.* In the case of the KAON SERVER, we use the Java Management Extensions (JMX [Lindfors and Fleury 2002]) as it is an open technology and currently the state-of-the-art for component management.

Java Management Extensions represent a universal, open technology for management and monitoring. By design, it is suitable for adapting legacy systems and implementing management solutions. Basically, JMX defines interfaces of managed beans, or *MBeans* for short, which are JavaBeans[9] suited for management purposes. MBeans are hosted by an *MBeanServer*, which allows their manipulation. All management operations performed on the MBeans are done through interfaces on the MBeanServer as depicted in Figure 6. We would like to point out two important methods of the MBeanServer:

```
registerMBean(Object object, ObjectName name),
```

which, as the name suggests, registers an object as MBean to the MBeanServer; the object has to fulfill a certain contract implementing a prescribed interface, and

```
Object invoke(ObjectName name, String operationName,
              Object[]params, String[] signature).
```

[9]http://java.sun.com/products/javabeans/.

All method invocations are tunnelled through the MBeanServer to the actual MBean by this method. The corresponding MBean is specified by `name`, whereas `operation-Name`, `params` and `signature` provide the rest of the information needed. Type checking has to be done by the developer and method calls are centralized. Hence, the architecture responds flexibly to changing requirements and evolving interfaces. Due to this technique, it becomes easy to incorporate the mechanism of interceptors (cf. Subsection 7.3).

An MBean must be a concrete and public Java object with at least one public constructor. An MBean must have a statically typed Java interface that explicitly declares the management attributes and operations. The naming conventions used in the MBean interface closely follow the rules set by the JavaBeans component model. To expose the management attributes, one has to declare get and set methods, similar to JavaBean component properties. The MBeanServer uses introspection on the MBean class to determine which interfaces the class implements. In order to be recognized as a Standard MBean, a class `x` has to implement an interface `xMBean`. Defining the methods `getAttr()` and `setAttr()` will automatically make `Attr` a management attribute, in this case with read and write access. Only management attributes can be accessed and modified by a client. All the other public methods will be exposed as management operations. Each MBean is accessible by its identifying name, which follows a special syntax.

JMX only provides a specification. There are several implementations available. For the KAON SERVER, we have chosen *JBossMX*, a JMX implementation that is part of the comprehensive Application Server JBoss.[10] The reason for this decision is that it perfectly fits KAON SERVER requirements and that JBoss is open-source software.

In our setting, the MBeanServer implements the kernel and MBeans implement components. Speaking in terms of JMX, there is no difference between a system component and a functional component. Both are MBeans that are only distinguished by the registry.

7.1.2 *Registry*. The registry is simple main-memory based ontology store containing component descriptions and constituting a database of components that can be instantiated by the server. This information source is built around a management ontology, which specifies the functional aspects of a component, for example, the libraries required by a component, its name, the class that implements the component itself and so forth (cf. Section 5).

Building on an ontology instead of a fixed data schema, allows us to retain flexibility and extensibility. Component providers can locally extend the ontology, for example by introducing a new subcategory "InferenceEngine" to functional components. The use of expressive ontology languages allows one to restrict globally defined component associations. For example, an RDFStore may restrict a sendingEventsTo association to RDFEventListeners. The inference services offered by engines capable of dealing with expressive ontology languages additionally allow subsumption reasoning. Thus, it is possible to

[10]http://www.jboss.org.

integrate multiple local extensions to the management ontology into a concise taxonomy.

We implemented the registry as MBean and reused one of the KAON modules, which have all been made deployable (cf. Subsection 7.4). The main-memory implementation of the KAON Application Programmer's Interface (API) holds the management ontology. When a component is deployed, its description (usually stored in an XML file) is represented as an instance of a concept. A client can use the registry's surrogate to discover the component it is in need of.

7.1.3 *Association Management.* The management ontology also allows one to express associations among components. For example, dependencies that state that a given component requires the existence of another component. Therefore, the server has to load all required components and be aware of the dependencies when unloading components. This essentially requires maintaining the number of clients to a component. A component can only be unloaded if it does not have any further clients.

The JMX specification does not define any type of association management aspect for MBeans. That is the reason why we had to implement this functionality separately as another MBean. Apart from dependencies, it is able to register and manage event listeners between two MBeans A and B, so that B is notified whenever A issues an event.

7.1.4 *Component Loader.* The MBeanServer offers methods to deploy any MBean at runtime. However, the client application of an MBeanServer must explicitly create the MBeans it needs, it must maintain the list of required libraries, and it must add newly created MBeans to the registry by itself.

To lift these responsibilities from the individual client, we have developed a special component loader MBean that facilitates the deployment process. MBeans are described by XML documents that contain RDF(S) serializations according to the management ontology mentioned in Section 5. The component loader uses this description to deploy the MBean, to add the MBean in the registry and to put associations into action by applying the association management. For example, it deals with the transitive loading of required components. The component loader is able to deploy an MBean from arbitrary URLs, hence users of the server are not required to install any libraries on the server machine before instantiating a component. The component loader also ensures that shared libraries that are part of the component implementation are only loaded once if multiple components share the same library.

7.2 Connectors

The KAON SERVER comes with four MBeans that handle communication. First, there is the HTTP Adapter from Sun, which exposes all of the kernel's methods to a Web frontend. Second and third, we have developed Web Service (using the Simple Object Access Protocol) and RMI (Java Remote Method Invocation) connector MBeans. Both export the kernel's methods for remote access.

Finally, the Local connector embeds the KAON SERVER locally into the client application.

For the client there is a surrogate object called RemoteMBeanServer that implements the MBeanServer interface. It is the counterpart to one of the four connector MBeans mentioned above. Similar to stubs in CORBA, the application uses this object to interact with the MBeanServer and is relieved of all communication details. The developer can choose which of the four options (HTTP, RMI, Web Service, Local) shall be used by RemoteMBeanServer.

To facilitate all of the above for the client, we have built a ConnectorFactory, the methods of which return surrogate objects for the registry, association management, and component loader. In addition, we have also developed surrogate objects for functional components. For example, there exists a RemoteRDFServer, relaying communication to one of the KAON tools (cf. Subsection 7.4). Every surrogate has to be provided with the MBean's identifying name, which can be discovered in the registry.

7.3 Interceptors

As explained in Section 6, interceptors are software entities that monitor a request and modify it before the request is sent to the component.

In the kernel, each MBean can be registered with an invoker and a stack of interceptors. A request received from the client is then delegated to the invoker first, before it is relayed to the MBean. The invoker object is responsible for managing the interceptors and sending the requests down the chain of interceptors towards the MBean. For example, a logging interceptor can be activated to implement auditing of operation requests. An authorization interceptor can be used to check that the requesting client has sufficient access rights for the MBean.

Invokers and interceptors are useful to achieve other goals apart from security. For example, when a component is being restarted, an invoker could block and queue incoming requests until the component is available again or the received requests time out. Alternatively, it could redirect the incoming requests to another MBean, which is able to fulfill them. Interceptors may also be used to meet the requirement of Semantic Interoperation. Client requests in a particular Semantic Web language can be translated such that they can be understood by a component that might talk another language.

7.4 Functional Components

KAON Tools. The KAON Tool suite defines two Semantic Web Data APIs for updates and queries—an RDF API and an API for querying and updating ontologies and instances (KAON API). There are different implementations that have been made deployable, among them main-memory based and persistent RDF stores as well as main-memory based and persistent KAON ontology stores.

Ontology Repository. One optional component currently developed is an Ontology Repository [Maedche et al. 2003], allowing access and reuse of ontologies that are used throughout the Semantic Web, such as WordNet [Miller et al. 1990] or foundational ontologies [Oltramari et al. 2002].

OntoLiFT. The OntoLiFT component leverages existing schema structures as a starting point for developing ontologies for the Semantic Web [Stojanovic et al. 2002b]. Methods have been developed for deriving ontology structures for existing information systems, such as XML-DTD, XML-Schema, relational database schemata, or UML specifications of object-oriented software systems. The LiFT tool semi-automatically extracts ontologies from such legacy resources.

External Modules. We have developed several proxy components in order to adapt external modules: Sesame [Broekstra et al. 2002], Ontobroker [Decker et al. 1998] as well as a proxy component for DL classifiers that conform to the DIG interface,[11] like FaCT [Horrocks 1998] or Racer [Haarslev and Moeller 2001]. Many more will follow in the future.

8. BUILDING A PORTAL FOR ACADEMIA AND RESEARCH

This section shows the usefulness of an Application Server for the Semantic Web by a detailed scenario (cf. also Staab et al. [2000]). The scenario shows the reader how the different parts of the Application Server, which so far have only been described in isolation from each other, interact.

We refer to the scenario depicted in Figure 1, which involved concise modelling of the research and academia domain in description logics. The ontology thus created can be used in several research and academia applications. In our scenario, we want to set up a comprehensive portal, which exploits a rule-based system capable of handling large amounts of instances and deduction of additional information by rules. Basically, there are three types of rules:

(1) **Schema integration** Rules that put concepts into a taxonomy, like *Graduate and AcademicStaff are specializations of Person*. Such constraints are resolved by subsumption reasoning in descriptions logics.
(2) **Rules involving instances and one concept** For example, *If Person A cooperatesWith Person B then B also cooperatesWith A*. Description logics are capable of handling such rules in theory. However, no performant reasoner exists that can handle large amounts of instances.
(3) **Rules that involve instances and several concepts** For example, *If a Person A works in Project X and X's topic is Z, then Person A is familiar with the topic Z*. Decidable description logics are not able to express such rules [Schmidt-Schauß 1989].[12]

In the following subsections, we want to show how the scenario can be solved with the KAON SERVER using existing clients and several components. The modeling of a domain ontology is initially independent of a particular application. It has to be as concise as possible and agreed upon by the community. The

[11]Description Logic Implementation Group, http://dl.kr.org/dig/.
[12]Also they don't intend to, as they mostly focus on reasoning at the schema level.

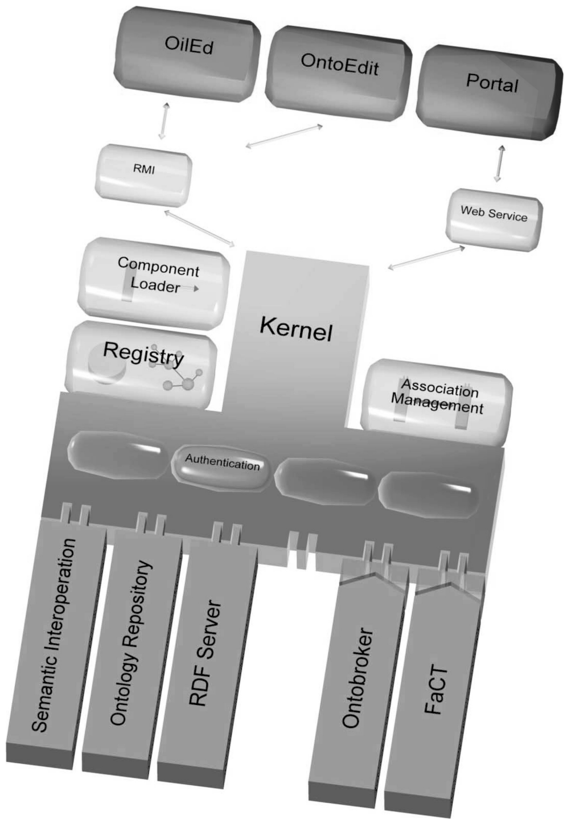


Fig. 7. An instance of KAON SERVER.

preferred choice for concise domain modeling are description logics, in our case OWL DL (cf. Section 2). Therefore one may use OilEd [Bechhofer et al. 2001] for the construction of a domain ontology that takes a DOLCE [Oltramari et al. 2002] top-level ontology as a starting point.

For the portal application, OntoEdit [Sure et al. 2002] and its corresponding ontology store Ontobroker [Decker et al. 1998] are well-suited because they are based on frame logics [Kifer et al. 1995] that, in contrast to OWL DL, allow the definition of, and reasoning with, rules of type (2) and (3).

We assume that an instance of the KAON SERVER is up and running, deployed with RMI and Web Service connectors, component loader, registry, association management as well as semantic interoperation, ontology repository functional components and proxy components for Ontobroker and FaCT [Horrocks 1998] (cf. Figure 7). The RDF Server will later be deployed by one of the editors.

OilEd's and OntoEdit's interactions with the server are discussed by UML-like sequence diagrams [Booch et al. 1998] in the following. Note that these diagrams do not show the exact Java method calls for the sake of brevity. For the same reason, we have omitted all the details involving connectors.

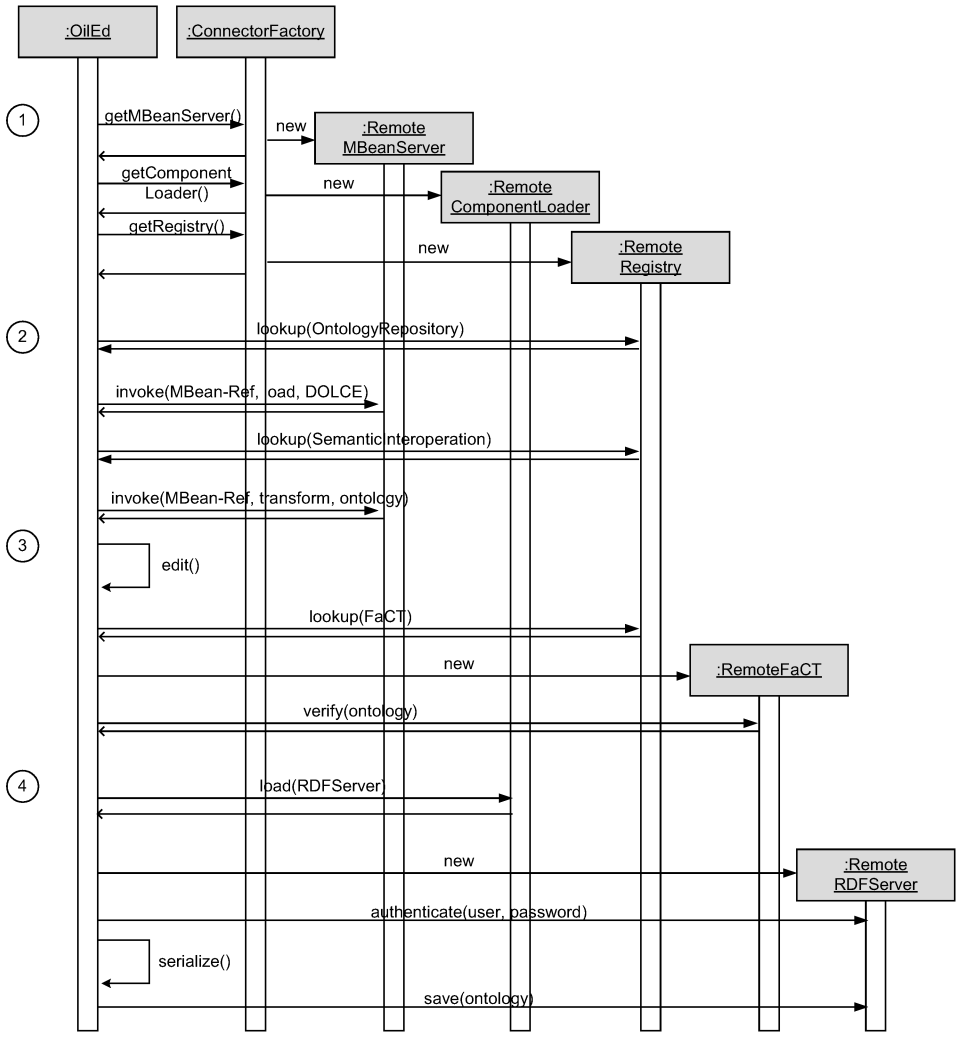


Fig. 8. Sequence diagram—OilEd with KAON SERVER.

8.1 Modelling the Ontology

For ontology engineering we use OilEd, an editor that supports the OWL DL language among others. It connects to the KAON SERVER through Java Remote Method Invocation (RMI). As depicted in Figure 8, OilEd uses the ConnectorFactory to get surrogate objects for the MBeanServer itself, the component loader, and the registry in the acquisition phase (1). What follows in step (2) is a successful discovery of the ontology repository functional component.

Interactions from surrogate objects (Remote*x*) to the KAON SERVER are not shown in the diagrams. Each surrogate has to be created on the client-side and relay its method calls over the network to a connector's invoke() method, which eventually calls the MBeanServer's invoke().

A reference to the repository MBean is returned to OilEd, which in turn loads a DOLCE top-level ontology as the starting point for the domain ontology. This method invocation is directly routed through the MBeanServer without using a surrogate object. This is achieved by the invoke() method, which takes an MBean reference, the name of the operation, and its parameters as arguments (cf. Section 7.1).

After that, the editor looks up the MBean reference for the semantic interoperation functional component. OilEd uses it to transform the DOLCE ontology into the OWL DL language. This method invocation is also routed through the MBeanServer without any surrogate objects. At this point, the user is able to start editing the research and academia ontology (3). When finished, a verification on the ontology is usually done by applying the FaCT reasoner [Horrocks 1998]. OilEd tries to find such an inference engine in the registry. In our scenario, there is a proxy component deployed, and thus a reference is returned. The editor creates a RemoteFaCT object, which hides the communication details. In our case, since the ontology is consistent, the user proceeds with saving. For storing the ontology, an instance of KAON's RDF Server along an authentication interceptor is created by using the component loader (4). OilEd is relieved from starting and initializing. It retrieves a reference to the newly created MBean from the component loader. Only then is it able to create an instance of RemoteRDFServer, which, like all other surrogates, hides the communication details as well as handling possible interceptors. For the latter, RemoteRDFServer has to be first provided with the credentials. After serializing the ontology into RDF, it is finally saved by the persistent RDF Server.

8.2 Definition of Rules

In the envisioned portal we want to apply reasoning based on logic programming [Das 1992] in order to deduce additional information. OWL DL does not allow the definition of rules, but we want to reuse the domain ontology. The semantic interoperation functional component allows the translation from OWL DL into frame logic and thus the usage of OntoEdit, which provides a graphical user interface for editing ontologies and rules.

Figure 9 depicts the sequence diagram for OntoEdit's communication with the server. RemoteMBeanServer and RemoteRegistry objects are created in phase (1), similar to OilEd's interactions. We assume that the user is aware of the RDF Server and the ontology just created. He/she can provide enough information to perform a successful discovery for the store as well as the required credentials (2). An instance of RemoteRDFServer is responsible for communication and handling the authentication interceptor on the server side. Invocation of getOntology(...) on RemoteRDFServer delivers an RDF-stream that is to be transformed into frame logic, OntoEdit's ontology language, by the semantic interoperation functional component. OntoEdit discovers the latter and calls the respective method directly, without creating any special surrogate object, through RemoteMBeanServer. The user is now able to add rules, instances and to perform adaptations on the ontology, as some information might have been lost during the translation from OWL DL (3).

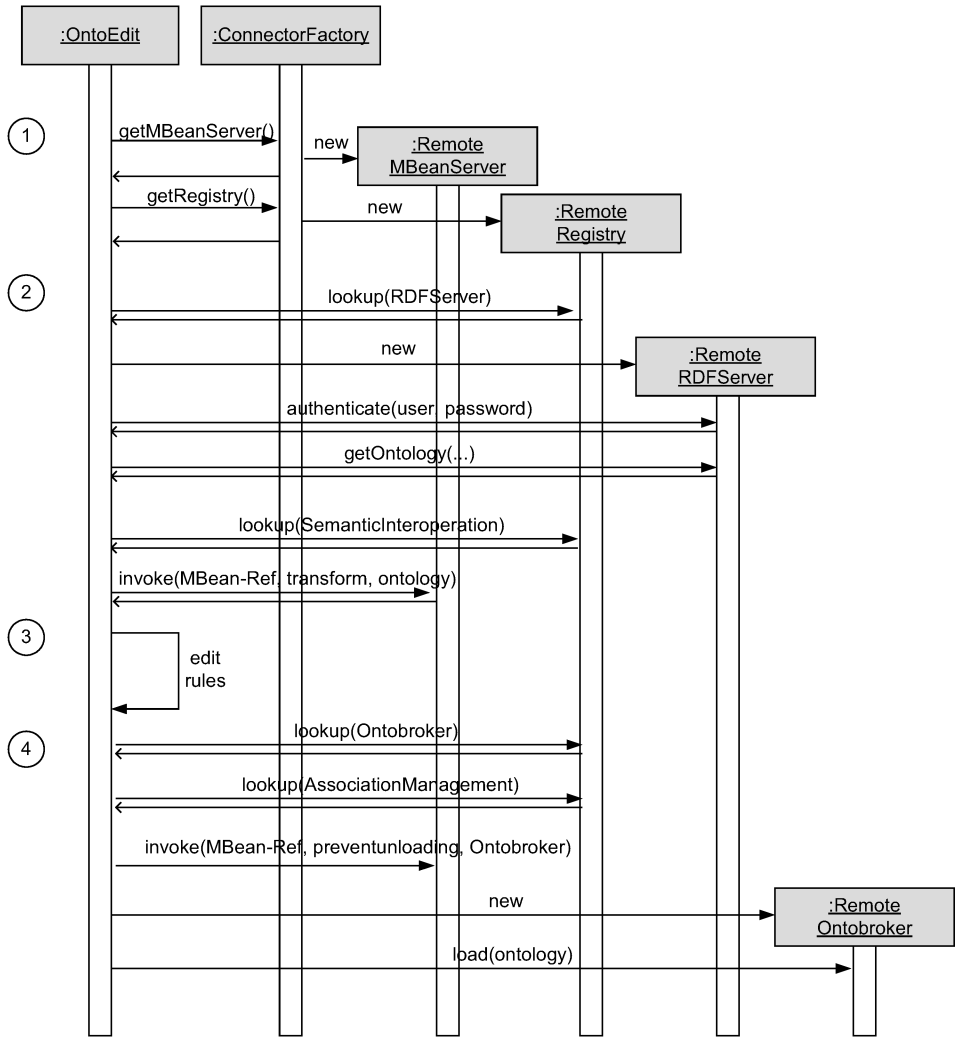


Fig. 9. Sequence diagram—OntoEdit with KAON SERVER.

OntoEdit uses Ontobroker for ontology storage and reasoning as well as semantic validation of the ontology (analogous to OilEd and FaCT). Ontobroker exploits a relational database system for persistence. We have already assumed that a proxy component for Ontobroker is deployed to the KAON SERVER. Instead of loading a new one, OntoEdit tries to discover such a component and retrieves a reference to the respective MBean (4). Before loading the frame logic ontology into Ontobroker, the editor ensures that the proxy component is not unloaded by other clients, or due to server performance reasons. It therefore retrieves a reference to the association management via the registry and invokes a corresponding method. Frame logic ontology, instances and rules can now be loaded into Ontobroker.

8.3 Setting up the Portal

After translation into frame logic, possible adaptations and addition of rules with OntoEdit, the portal application just needs to reuse the deployed Ontobroker residing within the KAON SERVER. It already holds the required ontology together with the rules. The application has to connect to the KAON SERVER, in this scenario by a Web Service connector, discover Ontobroker and start displaying and changing the ontology's instances by a web front-end. Without the KAON SERVER, all of the above would lead to a one-off effort of combining software modules without the possibility for much reuse and extensibility.

9. RELATED WORK

We consider four distinct topic areas as related work. First, RDF Data Management Systems approach some ideas relevant to an Application Server for the Semantic Web. Other than that, Ontology Development Environments as well as Knowledge Base Interoperation approaches share some similarities to our approach. Finally, the huge field of middleware and also classical software reuse systems are closely related to the Application Server for the Semantic Web.

9.1 RDF Data Management Systems

All of the following data management systems focus on RDF(S) only. Hence, they are not built with the aspect of extensibility in mind. However, they provide more specialized components than our implementation does and offer more extensive functionality with respect to RDF.

Sesame [Broekstra et al. 2002] is a scalable, modular architecture for persistent storage and querying of RDF and RDF Schema. It supports two query languages (RQL and SeRQL), and can use main memory or PostgreSQL, MySQL and Oracle 9i databases for storage. The Sesame system has been successfully made deployable by a proxy component for RDF support in KAON SERVER.

RDFSuite [Alexaki et al. 2001] is a suite of tools for RDF management provided by the ICS-Forth institute, Greece. Among those tools is an RDF Schema specific Database (RSSDB) that allows querying RDF using the RQL query language. The implementation of the system exploits the PostgreSQL object-relational DBMS. It uses a storage scheme that has been optimized for querying instances of RDFS-based ontologies. The database content itself can only be updated in a batch manner (dropping a database and uploading a file), hence it cannot cope with transactional updates, such as KAON's RDF Server.

Developed by Hewlett-Packard Research, UK, Jena [McBride 2001] is a collection of Semantic Web tools including a persistent storage component, an RDF query language (RDQL) and a DAML+OIL API. For persistence, the Berkley DB embedded database or any JDBC-compliant database may be used. Jena abstracts from storage in a similar way as the KAON APIs. However, no transactional updating facilities are provided so far.

9.2 Ontology Development Environments and Knowledge Base Interoperation

The Ontolingua ontology development environment [Fikes et al. 1997] provides a suite of ontology authoring tools and a library of modular, reusable ontologies. The tools in Ontolingua are oriented towards the authoring of ontologies by assembling and extending ontologies obtained from a library. However, Ontolingua's tools do not support the Semantic Web languages.

The same limitation holds for the Stanford Research Institute's OKBC (Open Knowledge Base Connectivity), which is a protocol for accessing knowledges bases (KBs) stored in Knowledge Representation Systems (KRSs) [Chaudhri et al. 1998]. The goal of OKBC is to serve as an interface to many different KRSs, for example, an object-oriented database. OKBC provides a set of operations for a generic interface to underlying KRSs. The interface layer separates an application from the idiosyncrasies of specific KRS software and enables the development of generic tools (e.g. graphical browsers and editors) that operate on many KRSs.

9.3 Middleware

Middleware is a broadly used term nowadays and comes into play as soon as one divides an application into several tiers. Generally speaking, *middleware* (i.e. the middle-tier) is any software that is located between client front-end and database back-end. Because of its generality, the term comprises simple servlet engines, more comprehensive Application Servers, service-oriented architectures, messaging products and even transaction monitors. Their common feature is that they all attempt to facilitate multi-tier application development in distributed infrastructures. We limit ourselves to Application Servers since they are most related.

An *Application Server* is a component-based product that resides in the middle-tier of a server centric architecture. It provides middleware services for security and state maintenance, along with data access and persistence.[13] A multitude of commercially available Application Servers exist that either conform to J2EE (Java 2 Platform, Enterprise Edition) or Microsoft's .NET architecture. Both comprise means for handling components (Enterprise JavaBeans, COM), remote access (RMI, DCOM), location services (JNDI, ActiveDirectory), database connectivity (JDBC, DAO), transaction services (JTS, MTS), asynchronous communication (JMS, MSMQ) and many more. The most popular Application Servers are BEA WebLogic, IBM WebSphere, Oracle 9i, JBoss and Hewlett Packard's Core Services Framework (CSF).

Our Application Server for the Semantic Web adopts the ideas and technologies of state-of-the-art Application Servers, but offers special features that are of particular importance in the Semantic Web (as visible in the Semantic Web specific requirements, cf. Subsection 3.2). In fact, we are reusing the very core of the JBoss Application Server—its JMX implementation JBossMX—and have built our own functionality around it. Connectors allow publishing components' methods as different Semantic Web software entities, for example,

[13]cf. http://www.service-architecture.com.

peers or agents, and proxy components enable easy integration of existing ones. Additionally, interceptors offer means for semantic interoperation between Semantic Web data formats. Anther novelty is the usage of semantic technology within the server itself (cf. requirements in Subsection 3.3). We offer an ontology-based registry that is used for discovery, classification, connectivity and implementation tasks. Common Application Servers use common directory-services for discovery only. The association management component puts ontological dependencies into action in conjunction with the component loader and the registry. As a result, we consider our Application Server for the Semantic Web as *semantic middleware*.

9.4 Software Reuse Systems

Classical Software Reuse Systems are comparable with our work in that they also need to describe software modules appropriately for efficient and precise retrieval. Techniques like faceted classification [Diaz 1991] represent features of the providers rather than the goals they achieve. Techniques such as analogical software reuse [Massonet and van Lamsweerde 1997] share a representation of modules that is based on goals achieved by the software, roles and conditions. Zaremski and Wing [1997] describe a specification language and matching mechanism for software modules. They allow for multiple degrees of matching but consider only syntactic information. UPML, the Unified Problem-solving Method Development Language [Fensel et al. 1999], has been developed to describe and implement intelligent broker architectures and components to facilitate semiautomatic reuse and adaptation. It is a framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components that are represented by inputs, outputs, preconditions and effects of tasks. However, none of these approaches provides means for semantic module and API discovery, semantic classification of modules or facilitation of implementation tasks.

10. CONCLUSION

The article discussed the requirements, design and conceptual architecture of an Application Server for the Semantic Web, and also presented a particular implementation—the KAON SERVER.

The KAON SERVER is based on the design and development of existing Application Servers, applying and augmenting their underlying concepts for use in the Semantic Web. From our perspective, it is an important step in putting the Semantic Web into practice. Based on our experiences with building Semantic Web applications we conclude that such a server will be crucial to achieve reuse and extensibility. This conclusion is substantiated by a detailed scenario that shows how the KAON SERVER can be used to facilitate the development of portal applications.

In contrast to existing Application Servers we also apply semantic technology—an ontology is used for component and API discovery, classification, connectivity and implementation tasks. All this makes life easier for the developer and adds to the capabilities of the Application Server going beyond state-of-the-art design and development.

In the future we will investigate the automatic generation of web service and peer descriptions out of the registry. The yet to be standardized upper layers of the Semantic Web, for example, the Trust layer, will result in additional requirements for our Application Server, which have to be met.

ACKNOWLEDGMENTS

We would like to thank our colleagues, in particular Boris Motik, Alexander Maedche, Marta Sabou, Debbie Richards, Brian McBride and all of the Wonder-Web researchers for fruitful discussions that helped to shape this contribution.

REFERENCES

ALEXAKI, S., CHRISTOPHIDES, V., KARVOUNARAKIS, G., AND PLEXOUSAKIS, D. 2001. On storing voluminous RDF descriptions: The case of web portal catalogs. In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB'01)—In conjunction with ACM SIGMOD/PODS, May 24–25 2001, Santa Barbara, CA, pages 43–48*. Informal Proceedings.

BAADER, F., HORROCKS, I., AND SATTLER, U. 2003. *Description Logics*. International Handbooks on Information Systems, vol. Handbook on Ontologies in Information Systems. Steffen Staab and Rudi Studer, Eds., Springer, Chapter I: Ontology Representation and Reasoning, 3–31.

BECHHOFER, S., HORROCKS, I., GOBLE, C., AND STEVENS, R. 2001. OilEd: A reason-able ontology editor for the Semantic Web. In *Proceedings of the Joint German Austrian Conference on Artificial Intelligence*. Lecture Notes in Artificial Intelligence, vol. 2174. Springer, 396–408.

BENNETT, B., DIXON, C., FISHER, M., HUSTADT, U., FRANCONI, E., HORROCKS, I., AND DE RIJKE, M. 2002. Combinations of modal logics. *Artif. Intel. Rev. 17*, 1 (Mar.), 1–20.

BIRON, P. V. AND MALHOTRA, A. 2001. XML Schema part 2: Datatypes. W3C Recommendation. http://www.w3.org/TR/xmlschema-2/.

BISHOP, J. M., Ed. 2002. *Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20–21, 2002, Proceedings*. Lecture Notes in Computer Science, vol. 2370. Springer.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 1998. *The Unified Modeling Language User Guide*. Vol. 1. Addison-Wesley.

BORGIDA, A. AND SERAFINI, L. 2002. Distributed description logics: Directed domain correspondences in federated information sources. In *On the Move to Meaningful Internet Systems, 2002—DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30 - November 1, 2002, Proceedings*, R. Meersman and Z. Tari, Eds. Lecture Notes in Computer Science, vol. 2519. Springer, 36–53.

BOZSAK, E., EHRIG, M., HANDSCHUH, S., HOTHO, A., MAEDCHE, A., MOTIK, B., OBERLE, D., SCHMITZ, C., STAAB, S., STOJANOVIC, L., STOJANOVIC, N., STUDER, R., STUMME, G., SURE, Y., TANE, J., VOLZ, R., AND ZACHARIAS, V. 2002. KAON—towards a large scale Semantic Web. In *E-Commerce and Web Technologies, Third International Conference, EC-Web 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, Eds. Lecture Notes in Computer Science, vol. 2455. Springer.

BROEKSTRA, J., KAMPMAN, A., AND VAN HARMELEN, F. 2002. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *The Semantic Web—ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9–12, 2002, Proceedings*, I. Horrocks and J. A. Hendler, Eds. Lecture Notes in Computer Science, vol. 2342. Springer.

BURSTEIN, M. H., HOBBS, J. R., LASSILA, O., MARTIN, D., MCDERMOTT, D. V., MCILRAITH, S. A., NARAYANAN, S., PAOLUCCI, M., PAYNE, T. R., AND SYCARA, K. P. 2002. DAML-S: Web service description for the Semantic Web. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9–12, 2002, Proceedings*, I. Horrocks and J. A. Hendler, Eds. Lecture Notes in Computer Science, vol. 2342. Springer, 348–363.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Vol. 1. John Wiley and Son Ltd.

CHAUDHRI, V., FARQUHAR, A., FIKES, R., KARP, P., AND RICE, J. 1998. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the Fifteenth National Conference* *on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26–30, 1998, Madison, Wisconsin, USA*. AAAI Press/The MIT Press, 600–607.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. 2003. Web services description language (WSDL). Working Draft. http://www.w3.org/TR/wsdl.

Das, S. K. 1992. *Deductive Databases and Logic Programming*. Addison Wesley.

Decker, S., Erdmann, M., Fensel, D., and Studer, R. 1998. Ontobroker: Ontology based access to distributed and semi-structured information. In *Database Semantics—Semantic Issues in Multimedia Systems*. IFIP Conference Proceedings, vol. 138. Kluwer, 351–369.

Diaz, R. P. 1991. Implementing faceted classification for software reuse. *Comm. ACM 34*, 5 (May), 88–97.

Dumbill, E. 2001. Building the Semantic Web. http://www.xml.com/pub/a/2001/03/07/-buildingsw.html. Knowledge Technologies Conference 2001, March 4–7, Austin Convention Center, Austin, TX, USA, Keynote presentation.

Fensel, D., Benjamins, R., Motta, E., and Wielinga, B. J. 1999. UPML: A framework for knowledge system reuse. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31–August 6, 1999. 2 Volumes, 1450 pages*, T. Dean, Ed. Morgan Kaufmann, 16–23.

Fensel, D., Hendler, J. A., Lieberman, H., and Wahlster, W., Eds. 2003. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press.

Fikes, R., Farquhar, A., and Rice, J. 1997. Tools for assembling modular ontologies in ontolingua. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27–31, 1997, Providence, Rhode Island.* AAAI Press / The MIT Press, 436–441.

Gangemi, A., Pisanelli, D. M., and Steve, G. 1999. An overview of the ONIONS project: Applying ontologies to the integration of medical terminologies. *Data and Knowledge Engineering 31*, 2 (Sept.), 183–220.

Grosof, B., Horrocks, I., Volz, R., and Decker, S. 2003. Description logic programs: Combining logic programs with description logic. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20–24 May 2003*. ACM, 48–57.

Gruber, T. R. 1993. Towards principles for the design of ontologies used for knowledge sharing. In *Formal Ontology in Conceptual Analysis and Knowledge Representation*, N. Guarino and R. Poli, Eds. Kluwer Academic Publishers, Deventer, The Netherlands.

Haarslev, V. and Moeller, R. 2001. Racer system description. In *Proceedings of Automated Reasoning, First International Joint Conference, IJCAR*. Lecture Notes in Computer Science, vol. 2083. Springer, 701–706.

Handschuh, S., Staab, S., and Volz, R. 2003. On deep annotation. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20–24 May 2003*. ACM, 431–438.

Horrocks, I. 1998. The FaCT system. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '98, Oisterwijk, The Netherlands, May 5–8, 1998, Proceedings*, H. C. M. de Swart, Ed. Lecture Notes in Computer Science, vol. 1397. Springer.

Kifer, M., Lausen, G., and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *J. ACM 42*, 1, 741–843.

Lassila, O. and Swick, R. 1999. Resource description framework (RDF) model and syntax specification. W3C Recommendation. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

Lindfors, J. and Fleury, M. 2002. *JMX—Managing J2EE with Java Management Extensions*. Sams. The JBoss Group.

Maedche, A., Motik, B., Stojanovic, L., Studer, R., and Volz, R. 2003. An infrastructure for searching, reusing and evolving distributed ontologies. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20–24 May 2003*. ACM, 439–448.

Massonet, P. and van Lamsweerde, A. 1997. Analogical reuse of requirements frameworks. In *3rd IEEE International Symposium on Requirements Engineering (RE'97), January 5–8, 1997, Annapolis, MD, USA*. IEEE Computer Society, 26–39.

McBride, B. 2001. Jena: Implementing the RDF model and syntax specification. In *Proceedings of the Second International Workshop on the Semantic Web—SemWeb'2001, Hongkong, China, May 1, 2001*, S. Decker, D. Fensel, A. P. Seth, and S. Staab, Eds.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. A. 1990. Introduction to WordNet: An on-line lexical database. *Int. J. Lexicog. 3*, 4, 235–244.

Mohan, C., Cable, L., Devin, M., Dietzen, S., Helland, P., and Wolfson, D. 2001. Application servers (panel session): born-again TP monitors for the web. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM Press, 622.

Noy, N. F. and Klein, M. 2002. Ontology evolution: Not the same as schema evolution. Tech. Rep. SMI-2002-0926, Stanford University.

Noy, N. F. and Musen, M. A. 2000. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30–August 3, 2000, Austin, Texas, USA*. AAAI-2000 Technical Papers. AAAI Press / The MIT Press, 450–455.

Oberle, D., Sabou, M., Richards, D., and Volz, R. 2003. An ontology for semantic middleware: Extending DAML-S beyond web-services. In *On the Move to Meaningful Internet Systems and Ubiquitous Computing, 2003—DOA / CoopIS / ODBASE 2003 Confederated International Conferences DOA, CoopIS and ODBASE 2003 Catania, Sicily, Italy, November 3–7, 2003, Workshops*. Lecture Notes in Computer Science. Springer. In press.

Oltramari, A., Gangemi, A., Guarino, N., and Masolo, C. 2002. Sweetening ontologies with DOLCE. In *Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th International Conference, EKAW 2002, Siguenza, Spain, October 1–4, 2002, Proceedings*, A. Gómez-Pérez and V. R. Benjamins, Eds. Lecture Notes in Computer Science, vol. 2473. Springer.

Schmidt-Schauß, M. 1989. Subsumption in KL-ONE is undecidable. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada, May 15–18 1989.*, R. J. Brachman, H. J. Levesque, and R. Reiter, Eds. Morgan Kaufmann, 421–431.

Staab, S., Angele, J., Decker, S., Erdmann, M., Hotho, A., Maedche, A., Studer, R., and Sure, Y. 2000. Semantic community web portals. In *Proceedings of the 9th World Wide Web Conference (WWW-9), Amsterdam, Netherlands*. ACM, 473–491.

Stojanovic, L., Maedche, A., Motik, B., and Stojanovic, N. 2002a. User-driven ontology evolution management. In *On the Move to Meaningful Internet Systems, 2002—DOA / CoopIS / ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30–November 1, 2002, Proceedings*, R. Meersman and Z. Tari, Eds. Lecture Notes in Computer Science, vol. 2519. Springer.

Stojanovic, N., Volz, R., and Stojanovic, L. 2002b. A reverse engineering approach for migrating data-intensive web sites to the Semantic Web. In *Intelligent Information Processing, IFIP 17th World Computer Congress—TC12 Stream on Intelligent Information Processing, August 25–30, 2002, Montréal, Québec, Canada*, M. A. Musen, B. Neumann, and R. Studer, Eds. IFIP Conference Proceedings, vol. 221. Kluwer.

Sure, Y., Erdmann, M., Angele, J., Staab, S., Studer, R., and Wenke, D. 2002. OntoEdit: Collaborative ontology development for the Semantic Web. In *The Semantic Web—ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9–12, 2002, Proceedings*, I. Horrocks and J. A. Hendler, Eds. Lecture Notes in Computer Science, vol. 2342. Springer.

Ullman, J. D. 1988. *Principles of Database and Knowledge-base systems*. Principles of Comupter Science Series, vol. 14. Computer Science Press.

van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. 2003. Web ontology language (OWL) reference version 1.0. W3C Working Draft. http://www.w3.org/TR/owl-ref/.

Zaremski, A. M. and Wing, J. M. 1997. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol. 6*, 4 (Oct), 333–369.

Received April 2003; revised June 2003; accepted October 2003