KAON SERVER - A Semantic Web Management System

Raphael Volz
Institute AIFB
University of Karlsruhe
Karlsruhe, Germany

volz@fzi.de

Daniel Oberle
Institute AIFB
University of Karlsruhe
Karlsruhe, Germany

oberle@aifb.uni-karlsruhe.de

Steffen Staab
Institute AIFB
University of Karlsruhe
Karlsruhe, Germany

staab@ontoprise.de

Boris Motik
WIM
FZI
Karlsruhe, Germany

motik@fzi.de

ABSTRACT

The growing use of ontologies in applications creates the need for an infrastructure that allows developers to more easily combine different software modules like ontology stores, editors, or inference engines towards comprehensive ontology-based solutions. We call such an infrastructure Ontology Software Environment. The papers discusses requirements and design issues of such an Ontology Software Environment. In particular, we present this discussion in light of the ontology and (meta)data standards that exist in the Semantic Web and present our corresponding implementation, the KAON SERVER.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering

General Terms

KAON Server

Keywords

Semantic Web, Ontology infrastructure

1. INTRODUCTION

Ontologies are increasingly being applied in complex applications, e.g. for Knowledge Management, E-Commerce, eLearning, or information integration. In such systems ontologies serve various needs, like storage or exchange of data corresponding to an ontology, ontology-based reasoning or ontology-based navigation. Building a complex ontology-based system, one may not rely on a single software module to deliver all these different services. The developer of such a system would rather want to easily combine different — preferably existing — software modules. So far, however, such integration of ontology-based modules had to be done ad-hoc, generating an one-off endeavour, with little possibilities for re-use and future extensibility of individual modules or the overall system.

This paper is about an infrastructure that facilitates plug'n'play engineering of ontology-based modules and, thus, the development

and maintenance of comprehensive ontology-based systems, an infrastructure which we call an *Ontology Software Environment*. The Ontology Software Environment facilitates re-use of existing ontology stores, editors, and inference engines. It provides the basic technical infrastructure to coordinate the information flow between such modules, to define dependencies, to broadcast events between different modules and to transform between ontology-based data formats.

Each application area, like E-Commerce or eLearning, uses its own, usually proprietary, ontology language and format. In the following we limit ourselves to the languages defined in the Semantic Web - an augmentation of the current WWW that adds machine understandable content to web resources by ontological descriptions. The ontology languages are currently becoming de jure standards specified by the World Wide Web Consortium (W3C) and thus will be of importance in the future.

The paper is structured as follows: First, we provide a brief overview about the Semantic Web in Section 2 and motivate the need for an Ontology Software Environment via a common usage scenario in Section 3. We derive requirements for such a system in Section 4. Sections 5 and 6 describe the design decisions that we can derive from important requirements, namely extensibility and lookup. The conceptual architecture is then provided in Section 7. Section 8 presents the KAON SERVER, a particular Ontology Software Environment for the Semantic Web which has been implemented. Before we conclude related work is discussed in Section 9.
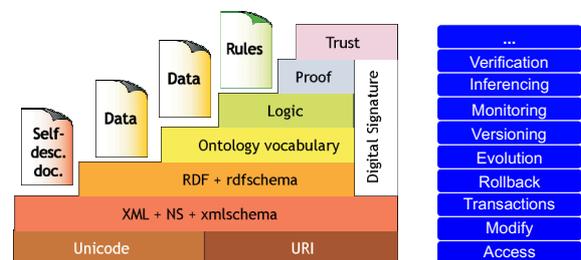

Figure 1: Static and dynamic aspects of the Semantic Web layer cake.

2. THE SEMANTIC WEB

The Semantic Web augments the current WWW by adding machine understandable content to web resources. Such contents are called metadata whose semantics can be specified by making use of ontologies. Ontologies play a key-role in the Semantic Web as they provide consensual and formal conceptualizations of a particular domain, enabling knowledge sharing and reuse.

Figure 1 shows both the static and dynamic parts of the Semantic Web layers. On the static side, 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 any kind of data exchange. XML Schema 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 paper has a title property with value "An extensible ontology software environment". RDF Schema extends RDF with the concepts of class and property hierarchies that enable the creation of simple ontologies.

The Ontology layer features OWL (Ontology Web Language) which is a family of richer ontology languages that intend to replace RDF Schema. The Logic, Proof and Trust layers aren't standardized yet.

The dynamic aspects apply to data across all layers. It is obvious that there have to be means for access and modification of Semantic Web data. Apparently, transactions and rollbacks of Semantic Web data operations should also be possible and meet the well-known ACID (atomicity, consistency, independence, durability) properties known from DBMS. Evolution and versioning of ontologies are also an important aspect, as domain formalizations usually have to cope with change [13]. Like in all distributed environments, monitoring of data operations is needed, in particular for confidential data. Finally, reasoning engines are to be applied for the deduction of implicit information as well as for semantic validation.
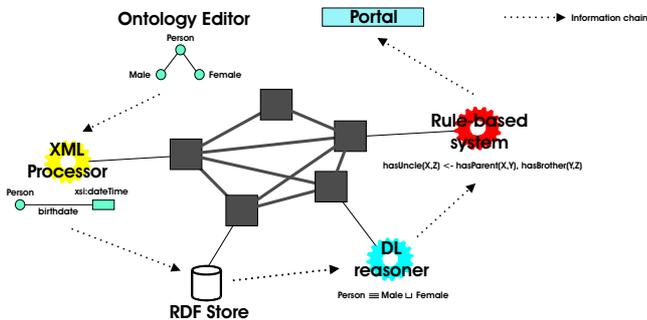

Figure 2: Motivating scenario

3. A MOTIVATING SCENARIO

This Section motivates the needs for the cooperation and integration of different software modules by a scenario depicted in Figure 2. The reader may note that some real-world problems have been abstracted away for the sake of simplicity.

Imagine a simple genealogy application. Apparently, the domain description, viz. the ontology, will include concepts like Person and make a distinction between Male and Female. There are several relations between Persons, e.g. hasParent or hasSister. This domain description can be easily expressed with standard description logic ontologies. However, many important facts that could be inferred automatically have to be added explicitly. A rule-based

system is needed to capture such facts automatically. Persons will have properties that require structured data types, such as dates of birth, which should be syntactically validated. Such an ontology could serve as the conceptual backbone for the information base of a genealogy portal. It would simplify the data maintenance and offer machine understandability. To implement the system, all the required modules, i.e. a rule-based inference engine, a DL reasoner, a XML Schema data type verifier, would have to be combined by the client applications themselves. While this is a doable effort, possibilities for re-use and future extensibility hardly exist.

The demands on an Ontology Software Environment from applications is to hook up to all the software modules and to offer management of data flow between them. This also involves propagation of updates and rollback behavior, if any module in the information chain breaks. In principle, an Ontology Software Environment responds to this need by bringing all the modules into one generic infrastructure. In the following Section, we will discuss requirements for this system.

4. REQUIREMENTS

Before we derive requirements from the scenario in Section 3, we introduce the term of a *Semantic Web Management System (SW-MS)*, which is a particular type of Ontology Software Environment, especially designed for aiding the development of Semantic Web applications. Certain web communities may have dedicated Semantic Web Management Systems that feature all the software modules needed in that particular community. E.g., the bioinformatics community may have its ontologies hosted by a SWMS, accessible to all its portals and application developers.

Basically, the scenario establishes four groups of requirements. Clients may want to connect remotely to the SWMS and must be properly authorized. Hence, it is obvious that in a distributed system like the Semantic Web there is the need for connectivity and security.

On the other hand, a SWMS should respond to the static aspects of the Semantic Web layer cake. In particular it should offer support for all its languages. A desirable property is also to provide the ability to translate between the different languages, thereby increasing interoperability between existing software modules that mostly focus on one language only.

The dynamic aspects result in another group of requirements, viz. finding, accessing and storing of data, consistency, concurrency, durability and reasoning.

Finally, the system is expected to facilitate a plug'n'play infrastructure in order to be extensible. The last group of requirements therefore deals with flexible handling of modules. In the following paragraphs we will elaborate on the requirements in more detail.

Connectivity and Security

- *Connectivity:* A Semantic Web Management System should enable loose coupling, allowing access through standard web protocols, as well as close coupling by embedding it into other applications. In other words, a client should be able to use the system locally and connect to it remotely via web services, for instance.

- *Security:* Guaranteeing information security means protecting information 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.

Semantic Web Languages

- *Language support:* A trivial requirement is the support of all the Semantic Web's ontology and metadata standards. The SWMS has to provide support for current SW standards such as RDF, RDFS and OWL while being flexible enough to support future languages which will result from the specification of the logic, proof and trust layers.

- *Semantic Interoperability:* We use the term semantic interoperability in the sense of translating between different ontology languages with different semantics. At the moment, several ontology languages populate the Semantic Web. Besides proprietary ones, we already mentioned RDFS, OWL Lite, OWL DL and OWL Full before. Usually, ontology editors and stores focus on one particular language and are not able to work with others. Hence, a Semantic Web Management System should allow to translate between different languages and semantics. An RDFS editor may not be able to load an OWL Full ontology for example. It is clear, that most often a translation cannot be achieved without loss of information.

- *Ontology Mapping:* In contrast to Semantic Interoperability, mapping deals with translating between different ontologies of the same language. Certain communities usually have their own ontology and could use Ontology Mapping to ease data exchange.

Dynamic Data

- *Ontology Storage:* Typically Semantic Web applications like editors or portals have to access and finally store ontological data. In addition, development of domain ontologies often builds on other ontologies as starting point. Examples are Wordnet or top-level ontologies for the Semantic Web [11]. Hence, the SWMS could store ontologies and offer them to applications.

- *Consistency:* Consistency of information is a requirement in any application. Each update of a consistent ontology must result in an ontology that is also consistent. In order to achieve that goal, precise rules must be defined for ontology evolution. Modules updating ontologies must implement and adhere to these rules. Also, all updates to the ontology must be done within transactions assuring the common properties of atomicity, consistency, isolation and durability (ACID).

- *Durability:* Like consistency, durability is a requirement that holds in any data-intense application area. It may be accomplished by reusing existing database technology.

- *Concurrency:* It must be possible to concurrently access and modify Semantic Web data. This may be achieved using transactional processing, where objects can be modified at most by one transaction at the time.

- *Reasoning:* Reasoning engines are central components of Semantic Web applications and can be used for several tasks like semantic validation and deduction of implicit information. A SWMS should provide access to such engines, which can deliver the reasoning services required.

Flexibility

- *Extensibility:* The need for extensibility applies to most software systems. Principles of software engineering try to avoid system changes when additional functionality is needed in the future. Hence, extensibility is also desirable for a SWMS. In addition, a SWMS has to cope with the multitude of layers and data models in the Semantic Web that lead to a multitude of software modules, e.g. 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 system. First, several data APIs will follow in the future to support each layer and leading to new software modules that might have to be integrated. Second, particular applications may demand proprietary behavior. This requires that given solutions can be reused and extended with custom features.

- *Lookup of software modules:* For a client, there should be the possibility to state precisely what it wants to work with, e.g. an RDF store that holds a certain RDF model and allows for transactions. Hence, means for intelligent lookup of software modules are required. Based on a semantic description of the search target, the system should be able to discover what a client is looking for.

- *Dependencies:* The system should allow to express dependencies between different software modules. For instance, this could be the setting up of event listeners between modules. Another example is the management of dependencies like "module A is needed for module B".

In the following sections 5 to 7, we develop an architecture that is a result from the requirements put forward in this Section. After that we present the implementation details of our Semantic Web Management System called KAON SERVER.

5. COMPONENT MANAGEMENT

Due to the requirement of extensibility, the Microkernel pattern is the fundamental paradigma of our design. The Microkernel pattern allows to adapt to changing system requirements. It separates a minimal functional core from extended functionality and application-specific parts. The Microkernel itself serves as a socket for plugging in these extensions and coordinates the collaboration of extensions [4].

In our setting, the Microkernel's minimal functionality must take the form of simple management operations, i.e. starting, initializing, monitoring, combining and stopping of software modules. This approach requires software modules to be uniform so that they can be treated equally by the kernel. Hence, in order to use the Microkernel, all extensions have to be brought into a certain form. We paraphrase this process as *making existing software deployable*, i.e. bringing existing software into the particular infrastructure of the Semantic Web Management System, that means wrapping it so that it can be operated by the Microkernel. In our terminology, a software module becomes a *deployed component* by this process.

Hence, we refer to the process of registering, possibly initializing and starting a component as *deployment*.

Apart from the cost of making existing software deployable, another drawback of this approach is that performance will suffer slightly in comparison to stand alone use, as a request has to pass through the kernel first (and possibly the network). A client that

wants to make use of a deployed component's functionality talks to the Microkernel, which in turn passes requests on to the providers of functionality.

On the other hand, the Microkernel architecture offers several benefits. By making existing functionality, like RDF stores, inference engines etc., deployable, one is able to treat everything equally. As a result, we are able to deploy and undeploy components ad lib, reconfigure, monitor and possibly distribute them dynamically. Proxy components can be developed for software that cannot be made deployable for whatever reasons. Throughout the paper, we will show further advantages, among them

- easy realization of security, auditing, trust etc. as interceptors (further discussed in Section 7)

- enabling a client to perform a lookup for the component it is in need of (cf. Section 6)

- incorporation of information quality in a uniform way along the registry (cf. Section 9)

- definition of dependencies between components (cf. Section 7)

- instrumentation of object trading techniques to provide clients with references to components they want to use (cf. Section 6)

6. DESCRIPTION OF COMPONENTS

This Section presents our support for the requirement "lookup of software modules" stated in Section 4. A client typically wants to find the components, which provides the functionality needed at hand. Therefore a dedicated component, called *registry* is required, which stores descriptions of all deployed components, and allows to query this information. Application developers can distinguish between several types of components:

Component All software entities which can be deployed to the kernel.

System Component A component that provides a part of the functionality of the Semantic Web Management System itself, e.g. the registry or a connector.

Functional Component Component that typcially is of interest to the client and provides specific functionality. Typically, ontology-related software modules become functional components by making them deployable, e.g. RDF stores.

External Service An external service cannot be deployed directly as it may be programmed in a different language, live on a different computing platform, etc.

Proxy Component Special type of functional component that manages the communication to an external service. Examples are proxy components for inference engines, like FaCT [5].

Each component can have *attributes* like the interface it implements, its name, connection parameters as well as several other low-level properties. Besides, we can express *associations* between components, which can be put in action to support dependency management and event notification systems. This allows us to express that an ontology store component can rely on an RDF store for actual storage. The definitions above contained a *taxonomy*, e.g. it was stated that "functional component" is a specialization of "component" etc.
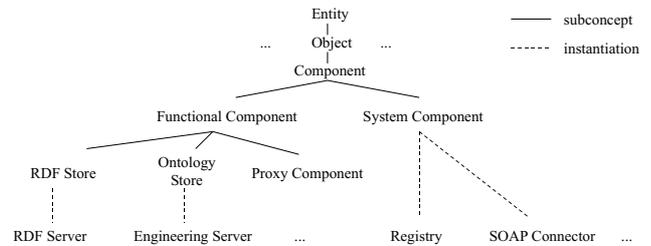

Figure 3: Management ontology

Thus, it is obvious to formalize taxonomy, attributes and associations in a management ontology as outlined in Figure 3 and Table 6[1]. The ontology formally defines which attributes a certain component may possess and categorizes components into a taxonomy. Hence, it can be considered as conceptual agreement and consensual understanding of the management world between a client and the SWMS. In the end, concrete and specific functional components, for example KAON's RDF Server or the Engineering Server (cf. subsection 8.5) would be instantiations of corresponding concepts.

Concept	Property	Range
Component	Name	String
	Interface	String
	...	...
	receivesEvent	Component
	sendsEvent	Component
	dependsOn	Component
	...	...

Table 1: Attributes and associations of Component

The support to important requirements, viz. extensibility and lookup of software components establishes the basis for the architecture of the SWMS. The next Section gives a brief overview about the conceptual architecture of our implementation.

7. CONCEPTUAL ARCHITECTURE

When a client first connects to the SWMS it will typically perform a *lookup* for some functional components it wants to interact with. The system will try to locate a deployed functional component in the registry that fulfills to the user query.

The lookup phase is followed by *usage* of the component. Here, the client can seamlessly work with the located functional component. Similar to CORBA, a surrogate for the functional component on the client side is responsible the communication over the network. The counterpart to that surrogate on the SWMS side is a connector component. It maps requests to the kernel's methods. All requests finally pass the management kernel which routes them to the actual functional component. In between, the properness of a request can be checked by security interceptors that may deal with authentication, authorization or auditing. Finally, the response passes the kernel again and finds its way to the client via the connector.

After this brief procedural overview, the following paragraphs will explain the architecture depicted in Figure 4. Note that in prin-

---

[1] The table shows some exemplary properties of the concept "Component". We use the term property as generalization for attribute and association. An attribute's range always is a string, whereas associations relate concepts.

ciple, there will be only three types of software entities: components, interceptors and the kernel.
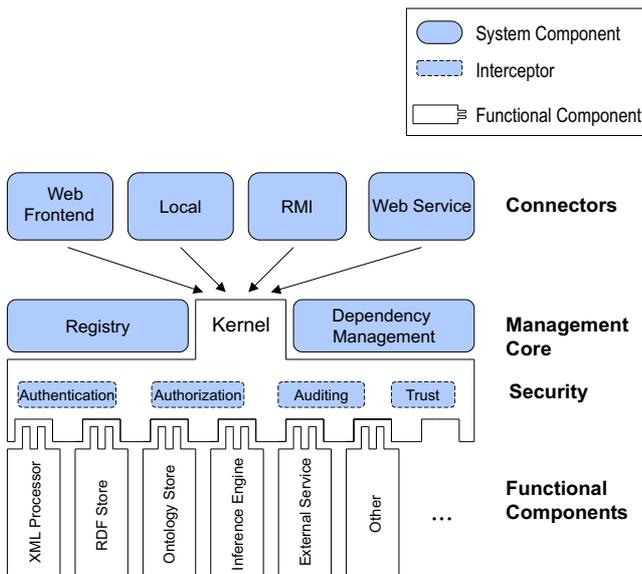

Figure 4: KAON SERVER Architecture

Connectors

A connector handles network communication with the system via some networking protocol. On the client side, surrogates for functional components are possible that relieve the application developer of the communication details. Besides having the option to connect locally, further connectors are possible to offer remote connection: e.g. via Java's Remote Method Invocation (RMI) protocol. Another example embodied in our implementation is a Web-frontend that renders the system interface into a HTML user interface and allows to interact with the system via the HTTP protocol.

Management Core

The Microkernel described in Section 5 is provided by the management kernel and deals with the discovery, allocation and loading of components, that are eventually able to execute a request. The registry hierarchically categorizes the descriptions of the components. It thus simplifies the lookup of a functional component for a client (cf. Section 6). As the name suggests, dependency management allows to express and manage relations between components. Besides enforcing classical dependencies, e.g. that a component may not be undeployed if others still rely on it, this also offers a basis for implementing event listeners, which allow messaging as an additional form of inter-component communication.

Security

Security is realized by several interceptors which guarantee that operations offered by functional components (including data update and query operations) in the SWMS are only available to appropriately authenticated and authorized clients. Each component can be registered along several interceptors which act in front and check incoming requests. Sharing generic functionality such as security, logging, or concurrency control lessens the work required to develop individual component implementations when realized via interceptors.

Functional Components

The software modules, e.g. XML processor, RDF store, ontology store etc., reside within the management kernel as functional components (cf. Section 5). In combination with the registry, the kernel can start functional components dynamically on client requests. Table 7 shows how the requirements established in Section 4 are supported by the architecture. Note that some requirements are eventually implemented by concrete functional components. Leaving those requirements aside in the conceptual architecture, e.g. language standards, increases the generality of the architecture, i.e. we could make almost any existing software deployable and use the system in any domain, not just in the Semantic Web. In the following Section we discuss a particular implementation, KAON SERVER, that realizes functional components specific for the Semantic Web.

Requirement \ Design Element	Connectors	Kernel	Registry	Interceptors	Dep. Man't	Func. Comp.
Connectivity	×					
Security				×		×
Language Support						×
Semantic Interoperability						×
Ontology Mapping						×
Ontology Storage						×
Consistency						×
Concurrency		×				×
Durability						×
Reasoning						×
Extensibility		×				×
Lookup			×			
Dependencies					×	

Table 2: Architectural support of requirements

8. IMPLEMENTATION

Our Karlsruhe Ontology and Semantic Web Tool suite (KAON, cf. http://kaon.semanticweb.org) provides a multitude of software modules especially designed for the Semantic Web. Among them are a persistent RDF store, an ontology store which is both optimized for concurrent engineering and direct access, ontology editors and many more. The tool suite relies on Java and open technologies throughout the implementation. KAON is a joint effort by the Institute AIFB, University of Karlsruhe as well as the Research Center for Information Technologies (FZI). KAON is a result of the efforts of several EU-funded research projects and implements requirements imposed by industry projects.

This Section presents the KAON SERVER which brings all those so far disjoint software modules plus optionally third party modules in a uniform infrastructure. KAON SERVER can thus be considered as a particular type of Semantic Web Management System optimized for and part of the KAON Tool suite. It follows the conceptual architecture presented in Section 7. The development of the KAON SERVER is carried out in the context of the EU IST funded WonderWeb[2] where it serves as main organizational unit and infrastructure kernel.

8.1 Kernel

---

[2]EU IST 2001-33052, http://wonderweb.semanticweb.org

In the case of the KAON SERVER, we use the Java Management Extensions (JMX[3]) as it is an open standard and currently the state-of-the-art for component management.

Java Management Extensions represent a universal, open technology for management and monitoring. Basically, JMX defines interfaces of managed beans, or *MBeans* for short, which are Java objects that represent JMX manageable resources. MBeans conceptually follow the JavaBeans components model, thus providing a direct mapping between JavaBeans components and manageability. MBeans are hosted by an *MBeanServer* which provides the services allowing their manipulation. All management operations performed on the MBeans are done through interfaces on the MBeanServer. Thus, in our setting, the MBeanServer realizes the Microkernel and components are realized by MBeans.

Apart from the interceptors, all functionality takes the form of MBeans, be it a software module which a client wants to use or functionality that realizes SWMS logic like a connector, for instance. In other words, the MBeanServer is not aware of those differences. For a Semantic Web Management System such as KAON SERVER it is therefore important to make the difference explicit in the registry.

8.2 Interceptors

Each MBean can be registered with an invoker and a stack of interceptors that the request is passed through. The invoker object is responsible for managing the interceptors and sending the requests down the chain of interceptors towards the actual MBean. For example, a logging interceptor could be inserted to implement auditing of operation requests. A security interceptor could be put in place to check that the requesting client has sufficient access rights for the MBean or one of its attributes or operations. The invoker itself may additionally support the component lifecycle by controlling the entrance to the interceptor stack. When a component is being restarted, an invoker could block and queue incoming requests until the component is once again available (or the received requests time out), or redirect the incoming requests to another component that is able to service the request.

8.3 Registry

We realized the registry as MBean and re-used one of the KAON modules which have all been made deployable (cf. subsection 8.5). The main-memory implementation of the KAON API holds the management ontology. When a component is deployed, its description (usually stored in an XML file) is properly placed in the ontology. A client can use the KAON API's query methods to lookup the component it is in need of.

8.4 Dependency Management

Dependency Management is another MBean that manages relations between any other MBeans. An example would be the registration of an event listener between two MBeans.

8.5 Data APIs

The functionality described so far, i.e. the JMX Microkernel, the interceptors, the registry and the dependency management could be used in any domain not just the Semantic Web. In the remaining subsections we want to highlight the specialties which make the KAON SERVER suited for the Semantic Web.

First, there is our KAON Tool suite which has been made deployable. Furthermore, we envision a functional component that enables semantic interoperability of Semantic Web ontologies as well as an ontology repository. Several external services (inference

---

[3]http://java.sun.com/products/JavaManagement/

engines in particular) are also deployable, as we have developed proxy components for them. All of them are discussed in the following subsections and, besides the KAON tools, can be considered as outlook.

Two Semantic Web Data APIs for updates and queries are defined in the KAON framework - an RDF API and an ontological interface called KAON API. An OWL interface will follow in the future. The different tools implement those APIs in different ways like depicted in Figure 5 and will become functional components. They are discussed in subsection 8.6.

RDF API. The RDF API consists of interfaces for the transactional manipulation of RDF models with the possibility of modularization, a streaming-mode RDF parser and an RDF serializer for writing RDF models. The API features the object oriented pendants to the entities defined in [7] as interfaces. A so-called RDF model consists of a set of statements. In turn, each statement is represented as a triple (subject, predicate, object) with the elements either being resources or literals. The corresponding interfaces feature methods for querying and updating those entities, respectively.

Ontology API. Our ontological API, also known as KAON API, currently realizes the ontology language described in [9]. We have integrated means for ontology evolution and a transaction mechanism. The interfaces offer access to KAON ontologies and contain classes such as Concept, Property and Instance. The API decouples the ontology user from actual ontology persistence mechanisms. There are different implementations for accessing RDF-based ontologies accessible through the RDF API or ontologies stored in relational databases using the Engineering Server (cf. Figure 5).

Figure 5: KAON API Implementations

8.6 Functional Components

RDF Mainmemory Implementation. This implementation of the RDF API is primarily useful for accessing in-memory RDF models. That means, an RDF model is loaded into memory from an XML serialization on startup. After that, statements can be added, changed and deleted, all encapsulated in a transaction if preferred. Finally, the in-memory RDF model has to be serialized again.

RDF Server. The RDF Server is an implementation of the RDF API that enables persistent storage and management of RDF models. This solution relies on a physical structure that corresponds to the RDF model. Data is represented using four tables, one represents models and the other one represents statements contained in the model. The RDF Server uses a relational DBMS and relies on the JBoss Application Server[4] that handles the communication between client and DBMS.

KAON API on RDF API. As depicted in Figure 5, implementations of the ontological KAON API may use implementations of the RDF API.

---

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

Engineering Server. A separate implementation of the KAON API can be used for ontology engineering. This implementation provides efficient implementation of operations that are common during ontology engineering, such as concept adding and removal by applying transactions. A storage structure that is based on storing information on a metamodel level is applied here. A fixed set of relations is used, which corresponds to the structure of the used ontology language. Then individual concepts and properties are represented via tuples in the appropriate relation created for the respective meta-model element. This structure was not chosen before by any other RDF database, however it appears to be ideal for ontology engineering, where the number of instances (all represented in one table) is rather small, but the number of classes and properties dominate. Here, creation and deletion of classes and properties can be realized within transactional boundaries.

Ontology Repository. One optional component we envision is a Ontology Repository, allowing access and reuse of ontologies that are used throughout the Semantic Web, such as WordNet for example. Within the Wonderweb project several of them have been developed [11].

Semantic Interoperability. In the more distant future we plan to include a component that allows Semantic Interoperability between different types of ontology languages as a response to the requirement put forward in Section 4. In the introduction, we already mentioned RDFS, OWL Lite, OWL DL and OWL Full. Besides, there are older formats, like DAML+OIL and also proprietary ones like KAON ontologies [9]. This component should allow the loading of KAON ontologies in other editors, like OILEd [2], for instance. It is clear that such an ontology transformation is a costly task, i.e. information will probably be lost as the semantic expressiveness of the respective ontology languages differ.

8.7 External Services

The definitions in Section 6 already clarified that external services live outside the KAON SERVER. So-called proxy components have to be developed that are deployed and take care of the communication. Thus, from a client perspective, an external service cannot be distinguished from an actual functional component. At the moment we are adapting several inference engines: Sesame [6], Ontobroker [12] as well as a proxy component for description logic classifiers that conform to the DIG interface[5], like FaCT [5].

9. RELATED WORK

We consider two distinct topic areas as related work. One topic area is the work done for RDF data management systems. The other is middleware.

9.1 RDF data management systems

All of the following data management systems focus on RDF language only. Hence, they are not build with the aspect of extensibility in mind and provide more specialized components than our implementation does. Hence, they offer more extensive functionality wrt. RDF.

Sesame [6] is a scalable, modular, architecture for persistent storage and querying of RDF and RDF Schema.Sesame supports two query languages (RQL and RDQL), and can use main memory or PostgreSQL, MySQL and Oracle 9i databases for storage. The Sesame system has been successfully deployed as a functional component for RDF support in KAON SERVER.

RDFSuite [1] is a suite of tools for RDF management provided by the ICS-Forth institute, Greece. Among those tools is RDF Schema specific Database (RSSDB) that allows to query 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 the KAON RDF SERVER.

Developed by the Hewlett-Packard Research, UK, Jena [8] is a collection of Semantic Web tools including a persistent storage component, a 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.

9.2 Middleware

Much research on middleware recently circles around so-called service oriented architectures (SOA) [6], which are similar to our architecture, since functionality is broken into components - so-called Web Services - and their localization is realized via a centralized replicating registry (UDDI)[7]. However, here all components are stand-alone processes and are not manageable by a centralized kernel. The statements for SOAs also holds for previously proposed distributed object architectures with registries such as CORBA Trading Services [3] or JINI[8].

Several of today's application servers share our design of constructing a server instance via separately manageable components, e.g. the HP-AS [9] or JBoss[10]. However, they do not allow to manage the relations between components and their dependencies, as well as dynamic instantiation of deployed components due to client requests - rather all components have to be started explicitly via configuration files or a management interface.

10. CONCLUSION

We have presented the requirements for an Ontology Software Environment and the design of a Semantic Web Management System. Our prototype implementation - the KAON SERVER - has already illustrated its utility in the WonderWeb project. Individual components of the system are successfully used within the several other projects both in academia and industry.

In the future, we will address two research aspects: First, we envision to incorporate information quality criteria into the registry. Users will then be able to query information based on criteria like "fitness for use", "meets information consumers needs", or "previous user satisfaction" [10]. We will also support aggregated quality values, which can be composed of multiple criteria. Second, we plan to implement dedicated components that allow to inter-operate between different Semantic Web languages. For example, a dedicated component may be able to explicate all implicit facts in an ontology base for RDF clients.

---

[5]Description Logic Implementation Group, http://dl.kr.org/dig/

[6]http://archive.devx.com/xml/articles/sm100901/sidebar1.asp
[7]http://www.uddi.org/
[8]http://www.jini.org
[9]http://www.bluestone.com
[10]http://www.jboss.org

11. REFERENCES

[1] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ics-forth rdfsuite: Managing voluminous rdf description bases. In *2nd International Workshop on the Semantic Web (SemWeb'01), in conjunction with Tenth International World Wide Web Conference (WWW10), Hongkong, May 1, 2001*, pages 1–13, 2001.

[2] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. Oiled: a reasonable ontology editor for the semantic web. In *Proc. of the Joint German Austrian Conference on AI, number 2174 in Lecture Notes In Artificial Intelligence, pages 396-408*. Springer, 2001.

[3] Juergen Boldt. Corbaservices specification, 3 1997.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, volume 1. John Wiley and Son Ltd, 1996.

[5] I. Horrocks. The fact system. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*. Springer, 1998.

[6] Frank van Harmelen Jeen Broekstra, Arjohn Kampman. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings International Semantic Web Conference 2002*. Springer, 2002.

[7] O. Lassila and R. Swick. Resource description framework (rdf) model and syntax specification.

[8] Brian McBride. 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*, 2001.

[9] B. Motik, A. Maedche, and R. Volz. A conceptual modeling approach for building semantics-driven enterprise applications. In *Proceedings of the First International Conference on Ontologies, Databases and Application of Semantics (ODBASE-2002)*, November 2002.

[10] Felix Naumann. *Quality-driven query answering for integrated information systems*, volume 2261 of *Lecture Notes in Computer Science*. Springer, 02 2002.

[11] Gangemi A. Guarino N. Oltramari, A. and C. Masolo. Dolce: a descriptive ontology for linguistic and cognitive engineering (preliminary report).

[12] Dieter Fensel Rudi Studer Stefan Decker, Michael Erdmann. Ontobroker: Ontology based access to distributed and semi-structured information. In *DS-8*, pages 351–369, 1999.

[13] L. Stojanovic, N. Stojanovic, and S. Handschuh. Evolution of metadata in ontology-based knowledge management systems. In *1st German Workshop on Experience Management: Sharing Experiences about the Sharing of Experience, Berlin, March 7-8, 2002, Proceedings*, 2002.