Preface

Here come the golden words.

Place, Month Year *First Name, Surname*

Table of Contents

1. An extensible ontology software environment

Daniel Oberle[1], Raphael Volz[1,2], Steffen Staab[1], and Boris Motik[2]

[1] University of Karlsruhe, Institute AIFB
D-76128 Karlsruhe, Germany
email: {lastname}@aifb.uni-karlsruhe.de
[2] FZI - Research Center for Information Technologies
D-76131 Karlsruhe, Germany
email: {lastname}@fzi.de

Summary.

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 article 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.

1.1 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 a 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 combines means 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.

Communication between modules requires ontology languages and formats. The Ontology Software Environment presented in this paper supports

all languages defined in the Semantic Web because they are currently becoming standards specified by the World Wide Web Consortium (W3C) and thus will be of importance in the future. The Semantic Web is an augmentation of the current WWW that adds machine understandable content to web resources by ontological descriptions.

The article is structured as follows: First, we provide a brief overview about the Semantic Web in section 1.2 and motivate the need for an Ontology Software Environment by a scenario in section 1.3. We derive requirements for such a system in section 1.4. Sections 1.5 and 1.6 describe the design decisions that immediately react to important requirements, namely extensibility and lookup. The conceptual architecture is then provided in section 1.7. Section 1.8 presents the KAON SERVER, a particular Ontology Software Environment for the Semantic Web which has been implemented. Related work and conclusion are given in sections 1.9 and 1.10, respectively.

1.2 The Semantic Web

In this section we want to introduce the reader to the architecture and languages of the Semantic Web that we use in our Ontology Software Environment. The Semantic Web augments the current WWW by adding machine understandable content to web resources. Such added 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.

The left hand side of figure 1.1 shows the static and part of the Semantic Web, i.e. its 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 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 article 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 augment RDF Schema (cf. Part A of this book for a detailed discussion). OWL Lite is the simplest of these. It is a limited version of OWL Full that enables simple and efficient implementation. OWL DL is a richer subset 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 in theory undecidable, but in practise useful reasoners can be constructed.

The Logic layer will provide an interoperable language for describing the sets of deductions one can make from a collection of data – how, given a domain ontology, we can make connections and derive new facts about it.

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 a system in which we can place trust that the data we are seeing, the 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[1]. The standardization process has currently reached the Ontology layer, i.e. Logic, Proof and Trust layers aren't specified yet.

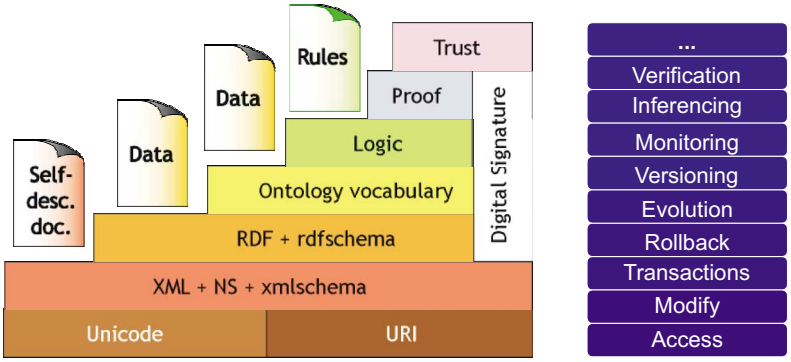

Fig. 1.1. Static and dynamic aspects of the Semantic Web layer cake

The right hand side of figure 1.1 depicts the Semantic Web's dynamic aspects that apply to data across all layers. Often, the dynamic aspects are disregarded by the Semantic Web community, however, from our point of view, they are an inevitable part for putting the Semantic Web into practise. It is obvious that there have to be means for access and modification of Semantic Web data. According to the the well-known ACID (atomicity, consistency, independence, durability) of DBMS, transactions and rollbacks of Semantic Web data operations should also be possible. Evolution and versioning of ontologies are an important aspect, because ontologies usually are subject to change [1.25]. Like in all distributed environments, monitoring of data operations becomes necessary for security reasons. Finally, reasoning

[1] Building the Semantic Web, http://www.xml.com/pub/a/2001/03/07/-buildingsw.html

engines are to be applied for the deduction of implicit information[2] as well as for semantic validation.

1.3 A Motivating Scenario

This section motivates the needs for the cooperation and integration of different software modules by a scenario depicted in figure 1.2. 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 and 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.

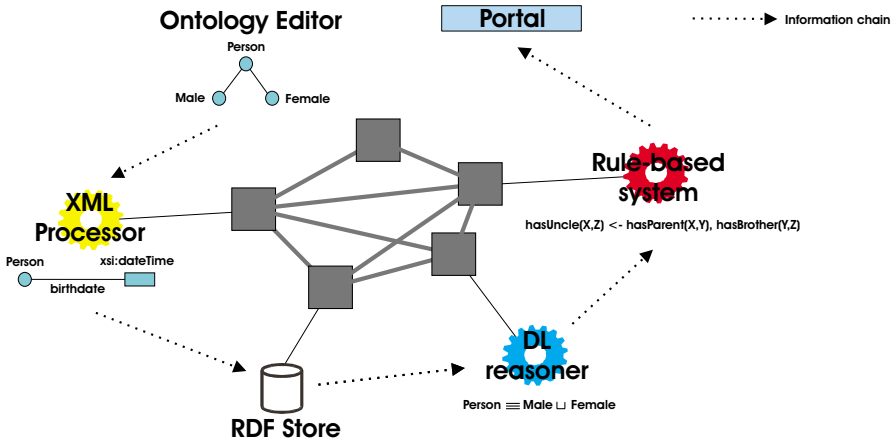

Fig. 1.2. Motivating scenario

[2] E.g., if "cooperatesWith" is defined as a symmetric property in OWL DL between persons A and B, a reasoner can deduce that B cooperatesWith A, too.

The demands on an Ontology Software Environment from this application 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 an Ontology Software Environment.

1.4 Requirements

Before we derive requirements from the scenario in section 1.3, we introduce the term of a *Semantic Web Management System (SWMS)*, which is a particular type of Ontology Software Environment, especially designed for aiding the development of a single Semantic Web application. We want to stress that a SWMS is not intended to manage the Semantic Web as a whole. Even within a community there will exist several SWMS instances supporting different applications.

Basically, the scenario establishes four groups of requirements. First, a SWMS should respond to the static aspects of the Semantic Web layer cake. In particular, it has to be aware of all its languages. Also belonging to the static aspects is the desire to translate between the different languages and thus increasing interoperability between existing software modules that mostly focus on one language only. Second, the dynamic aspects result in another group of requirements, viz. finding, accessing and storing of data, consistency, concurrency, durability and reasoning. Third, 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. 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.

- **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. The SWMS has to be aware of RDF, RDFS, OWL plus 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 by itself.
  - *Ontology Mapping* In contrast to Semantic Interoperability, 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, for instance.
- **Requirements stemming from the Semantic Web's dynamic part**
  - *Finding, accessing and storing of ontologies* 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 [1.2]. Those could be stored and offered by the SWMS to editors.
  - *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).
  - *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.
  - *Durability* Like consistency, durability is a requirement that holds in any data-intense application area. It may be accomplished by reusing existing database technology.
  - *Reasoning* Reasoning engines are core components of semantics-based 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.
- **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.

- **Flexible handling of modules**
  - *Extensibility* The need for extensibility applies to most software systems. Principles of software engineering 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 deal 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.
  - *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, that could be 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".

In the following sections 1.5 to 1.7, 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.

1.5 Component Management

Due to the requirement of extensibility, we decided for the Microkernel design pattern. This pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and application-specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration [1.14].

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 plus dispatching of messages between them. 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, software modules that shall be managed have to be brought into a certain form. We call this process *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 plugged into the Microkernel. Thus, a software module becomes a *deployed component*. The word *deployment* stems from research on service management and service oriented architectures where it is a terminus technicus. We adopt this meaning and applied in our setting we refine it as the process of registering, possibly initializing and starting a component to the Microkernel.

Apart from the cost of making existing software deployable, the only 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 dispatches requests.

But besides the drawbacks mentioned above, the Microkernel and component approach delivers several benefits. By making existing functionality, like RDF stores, inference engines etc., deployable, one is able to treat everything the same. 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 whatever reasons. Throughout the paper, we will show further advantages, among them

- enabling a client to perform a lookup for the component it is in need of (cf. section 1.6)
- definition of dependencies between components (cf. section 1.7)
- easy realization of security, auditing, trust etc. as interceptors (further discussed in section 1.7)
- incorporation of quality criteria as attributes of a component in registry (cf. section 1.10)

This section responded to the requirement of extensibility which led to a basic design decision. In the following, we will discuss how the lookup of software modules can be achieved.

1.6 Description of Components

This section responds to the requirement "lookup of software modules". As pointed out in the section 1.5, all components are equal as seen from the kernel's perspective. In order to allow a client performing a lookup for the components it is in need of, we have to make their differences explicit. Thus, there is a need of a registry that stores descriptions of all deployed components. In this section we show how a description of a component may look like. We start with the definition of a component and then specialize. The definitions result in a taxonomy that is primarily used to facilitate component lookup for the application developer.

Component The entity of management which is deployed to the kernel.
System Component Component providing functionality for the Semantic Web Management System, e.g. the registry or a connector.
Functional Component Component that is of interest to the client and can be looked up. 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, uses interfaces unknown, etc. It equals a functional component from a client perspective. This is achieved by having a proxy component deployed as surrogate for the external service.
Proxy Component Special type of functional component that manages the communication to an external service. Examples are proxy components for inference engines, like FaCT [1.15].

Each component can have *attributes* like the name of the interface it implements, connection parameters or other low-level properties. Besides, we want to be able to express *associations* between components. Associations can be dependencies between components, e.g. an ontology store component can rely on an RDF store for actual storage, or event listeners etc. Associations will later be put in action by dependency management (cf. section 1.7).

We formalize taxonomy, attributes and associations in a management ontology like outlined in figure 1.3 and table 1.1[3]. The ontology formally defines which attributes a certain component may have and places components into a taxonomy. In the end, actual functional components like KAON's RDF Server or the Engineering Server (cf. subsection 1.8.4) would be instantiations of components.

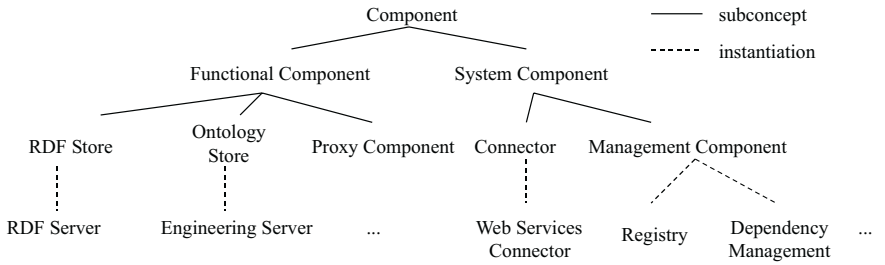

Fig. 1.3. Taxonomy of components

[3] 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.

Concept	Property	Range
Component	Name	String
	Interface	String
	...	...
	receivingEventsFrom	Component
	sendingEventsTo	Component
	dependsOn	Component
	...	...

Table 1.1. Attributes and associations of Component

Registry. Our approach allows us to realize the registry itself as a component. As explained in section 1.5, the Microkernel manages any functionality as long as it conforms to the contract. The registry is not of direct interest to the client - it is only used to facilitate the lookup of functional components. Hence, we can declare it as an instance of system component.

So far we have discussed two requirements, viz. extensibility and lookup of software components, which led to fundamental design decisions. The next section focuses on the conceptual architecture.

1.7 Conceptual Architecture

When a client connects to the SWMS it will perform a *lookup* for the functional components it is in need of. That could be an RDF store or an inference engine etc. The system tries to find a deployed functional component in the registry fulfilling the stated requirements and returns a reference.

From then on, the client can seamlessly work with the functional component. The lookup is followed by *usage* of the component. Similar to CORBA, a surrogate for the functional component on the client side handles the communication over the network. The counterpart to the 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 dispatches them to the actual functional component. While dispatching, the properness of a request can be checked by interceptors that may deal with authentication, authorization or auditing. An interceptor is a software entity which is allowed to look at a request and modify it before the request is sent the component. Finally, the response passes the kernel again and finds its way to the client through the connector.

After this brief procedural overview, the following paragraphs will explain the architecture depicted in figure 1.4. Note that in principle, there will be only three types of software entities: components, interceptors and the kernel. The first are distinguished into functional and system components only to facilitate the lookup for the application developer.

Connectors Connectors are system components. They send and receive requests and responses over the network by using some protocol. Apart from the

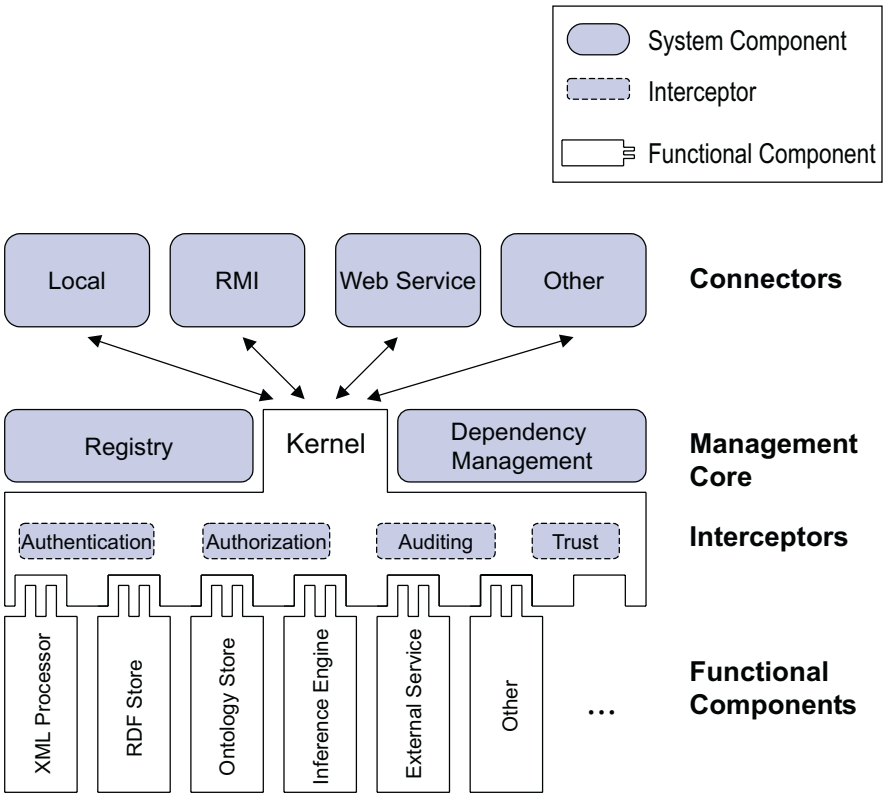

Fig. 1.4. Conceptual Architecture

option to connect locally, further connectors are possible for remote connection: e.g. ones that offer access per Java Remote Method Invocation (RMI), or ones that offer access per Web Service. Counterparts to a connector on the client side are surrogates for functional components that relieve the application developer of the communication details similar to stubs in CORBA.

Management Core The Management Core comprises the Microkernel (also called management kernel or simply kernel in the following) as well as several system components. The Microkernel described in section 1.5 is required to deal with the discovery, allocation and loading of components, that are eventually able to fulfill a request. The registry is a system component and hierarchically orders descriptions of the components. It thus facilitates the lookup of a functional component for a client (cf. section 1.6). Another system component called dependency management allows to express and manage relations between components. E.g., 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. The Management Core is ex-

tensible with additional functionality in the future by deploying extra system components.

Interceptors Interceptors are software entities that are allowed to look at a request and modify it before the request is sent the component. Security is realized by 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. Sharing generic functionality such as security, logging, or concurrency control lessens the work required to develop individual component implementations.

Functional Components The software modules, e.g. an RDF or ontology store, are deployed to management kernel as functional components (cf. section 1.5). In combination with the registry, the kernel can start functional components dynamically on client requests.

Table 1.2 shows where the requirements put forward in section 1.4 reflect themselves in the architecture. Note that most requirements are responded to by functional components. That is because the conceptual architecture presented here is generic, 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 Semantic Web standards.

Requirement \ Design Element	Connectors	Kernel	Registry	Interceptors	Dependency Management	Functional Components
Connectivity	×					
Security				×		×
Language Support						×
Semantic Interoperability						×
Ontology Mapping						×
Finding, accessing, storing of ontologies			×			×
Consistency						×
Concurrency		×				×
Durability						×
Reasoning						×
Extensibility		×				×
Lookup			×			
Dependencies					×	

Table 1.2. Requirements' reflections

1.8 Implementation

This section presents the KAON SERVER which is part of our Karlsruhe Ontology and Semantic Web Tool suite (KAON[4]) providing a multitude of software modules especially designed for the Semantic Web. The KAON SERVER brings all those so far disjoint software modules plus optionally third party modules into a uniform infrastructure. KAON SERVER can thus be considered as a particular kind of Semantic Web Management System optimized for and part of the KAON Tool suite. It follows the conceptual architecture presented in section 1.7 and relies on Java and open technologies throughout the implementation[5].

In the following, we will describe how the conceptual architecture discussed in section 1.7 has been implemented in the KAON SERVER. We will start with the Management Core in 1.8.1 as it is necessary to understand Connectors in 1.8.2, Interceptors in 1.8.3 and Functional Components in 1.8.5. Several of the latter are implementations of the two Data APIs defined in the KAON Toolsuite which are discussed before in 1.8.4.

1.8.1 Management Core

The Management Core of a Semantic Web Management System consists of the management kernel, the registry and dependency management system components. We will highlight all of their implementations in the subsections below.

Kernel. In the case of the KAON SERVER, we use the Java Management Extensions (JMX[6]) 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[7] that represent JMX manageable resources. MBeans are hosted by an *MBeanServer* which allows their manipulation. All management operations performed on the MBeans are done through interfaces on the MBeanServer.

In our setting, the MBeanServer realizes the kernel and components are realized by MBeans. Speaking in terms of JMX, there is no difference between

[4] KAON is a joint effort by the Institute AIFB, University of Karlsruhe as well as the Research Center for Information Technologies (FZI). The Tool suite is the result of ongoing development of several European Union IST projects and requirements of the industry, cf. http://kaon.semanticweb.org.

[5] The development of the KAON SERVER is carried out in the context of the EU IST funded WonderWeb, where it serves as main organizational unit and infrastructure kernel. EU IST 2001-33052, http://wonderweb.semanticweb.org

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

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

a system component and a functional component. Both are MBeans that are only distinguished by the registry.

Registry. We realized the registry as MBean and re-used one of the KAON modules which have all been made deployable (cf. subsection 1.8.4). 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.

Dependency Management. The JMX specification does not define any type of dependency management aspect for MBeans. Therefore, we realized dependency management by another MBean that manages relations between any other MBeans. An example would be the registration and management of an event listener between two MBeans A and B, so that B is notified whenever A issues an event. Other aspects are dependencies like "MBean A needs MBean B in order to work" or "MBean C must not be unloaded before MBean D is undeployed".

1.8.2 Connectors

The KAON SERVER comes with four system components, i.e. MBeans, that handle communication. First, there is an HTTP Adaptor from Sun that exposes all of the kernel's methods to a Web frontend. Additionally, 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, we have developed a local "connector" that fits smoothly into this infrastructure to embed the KAON SERVER into the client application.

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

1.8.3 Interceptors

Like explained in section 1.7, interceptors are software entities that are allowed to look at a request and modify it before the request is sent 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.

Apart from security, invokers and interceptors are useful to achieve other goals. E.g., 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 MBean that is able to fulfill it.

1.8.4 Data APIs

The functionality described so far, i.e. the Management Core, Connectors and Interceptors 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 suitable for ontologies that follow Semantic Web language standards and the Semantic Web, in particular.

First, the KAON Tool suite has been made deployable. Two Semantic Web Data APIs for updates and queries are defined in the KAON framework - an RDF API and an ontology data-oriented called KAON API. Their implementations result in functional components that are discussed in subsection 1.8.5. Before, the following paragraphs describe the APIs briefly.

Furthermore, we are currently developing functional components that enable semantic interoperability of Semantic Web ontologies (cf. section 1.4) as well as an ontology repository. Several external services (inference engines in particular) are also deployable, as we have developed proxy components for them. All of them are discussed in the remaining subsections.

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 object oriented pendants to the entities defined in [1.7] 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 ontology data-oriented API, also known as KAON API, currently realizes the ontology language described in [1.22]. We have integrated means for ontology evolution and a transaction mechanism. The interface offers access to KAON ontologies and contains classes such as Concept, Property and Instance. The API decouples a client from actual ontology storage mechanisms.

1.8.5 Functional Components

The KAON tools implement the APIs in different ways like depicted in figure 1.5 and have been made deployable. The paragraphs below talk about

the implementations in more detail. We also included paragraphs that describe additional functional components, i.e. ontology repository, OntoLiFT, semantic interoperability and finally external services.

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 in order to make changes persistent.

RDF Server. The RDF Server is an implementation of the RDF API that enables persistent storage and management of RDF models. It uses a relational database whose physical structure 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[8] that handles the communication between client and DBMS.

KAON API on RDF API. As depicted in figure 1.5, implementations of the ontological KAON API may use implementations of the RDF API. E.g., the KAON API can be realized using the mainmemory implementation of the RDF API for transient access and modification of a KAON ontology.

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 transactions.

Integration Engine. Another implementation of the KAON API is currently in the works which lifts existing databases to the ontology level. To achieve this, one must specify a set of mappings from some relational schema to the chosen ontology, according to principles described in [1.24]. E.g. it is possible to say that tuples of some relation make up a set of instances of some concept, and to map foreign key relationships into instance relationships.

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

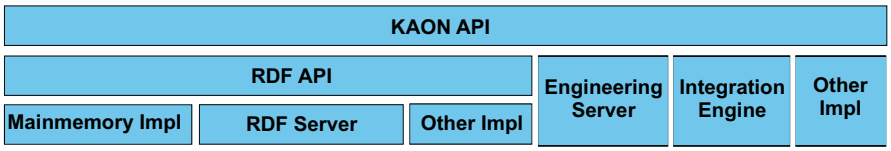

Fig. 1.5. KAON API Implementations

Ontology Repository. One optional component currently developed 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 [1.2].

OntoLiFT. Another component which is currently developed within WonderWeb is OntoLiFT aiming at leveraging existing schema structures as a starting point for developing ontologies for the Semantic Web. Effort has been invested in the development of schema 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. We are restricting our attention to the most important ones, namely the W3C schema languages for XML: Document Type Definitions (DTDs), XML Schema and relational database schemata. At the moment, we also provide a preliminary translation from UML-based software specifications to ontologies.

Semantic Interoperability. A functional component already developed, realizes the OWL Lite language on a SQL-99 compliant database system [1.1]. In addition, several others will later allow Semantic Interoperability between different types of ontology languages as a response to the requirement put forward in section 1.4. 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 [1.22]. It should be possible to load KAON ontologies in other editors, like OILEd [1.10], for instance. Information will be lost during ontology transformation as the semantic expressiveness of the respective ontology languages differ.

External Services. External services live outside the KAON SERVER. Proxy components are deployed and relay 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 [1.16], Ontobroker [1.5] as well as a proxy component for description logic classifiers that conform to the DIG interface[9], like FaCT [1.15] or Racer[1.6].

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

1.9 Related Work

Several systems approach the idea of a Semantic Web Management System. However, all of them focus on RDF(S) and cannot be extended very easily.

RDFSuite [1.9] is provided by ICS-Forth, Greece with a suite of tools for RDF management, among those is the so-call RDF Schema specific Database (RSSDB) that allows storing and querying RDF using RQL. For the implementation of persistence an object-relational DBMS is exploited. It uses a storage scheme that has been optimized for querying instances of RDFS-based ontologies. The database structure is tuned towards a particular ontology structure.

Sesame [1.16] is a RDF Schema-based repository and querying facility developed by Aidministrator Nederland bv as part of the European IST project On-To-Knowledge. The system provides a repository and query engine for RDF data and RDFS-based ontologies. It uses a variant of the RDF Query Language (RQL) that captures further functionality from RDFSchema specification when compared to the RDFSuite RQL language. Sesame shares its fundamental storage design with RDFSuite.

Developed by the Hewlett-Packard Research, UK, Jena [1.18] is a collection of RDF tools including a persistent storage component and a RDF query language (RDQL). 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.

Research on middleware circles around so-called service oriented architectures (SOA) [10], 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)[11]. 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 [1.12] or JINI[12].

Several of today's application servers share our design of constructing a server instance via separately manageable components, e.g. the HP Application Server[13] or JBoss[14]. Both have the Microkernel in common but follow their own architecture which is different from the one presented in our paper. JBoss wraps services like databases, Servlet and Enterprise JavaBeans containers or Java Messaging as components. HP applies its CSF (Core Services Framework) that provides registry, logging, security, loader, configuration facilities. However, whether JBoss nor HP AS deliver ontology-based registries,

[10] http://archive.devx.com/xml/articles/sm100901/sidebar1.asp
[11] http://www.uddi.org/
[12] http://www.jini.org
[13] http://www.bluestone.com
[14] http://www.jboss.org

dependency management nor are they suitable for the Semantic Web, in particular.

1.10 Conclusion

This article presented the requirements and design of a Semantic Web Management System as well as an implementation - the KAON SERVER. It is part of the open-source Karlsruhe Semantic Web and Ontology Tool suite (KAON). From our perspective, the KAON SERVER will be an important step in putting the Semantic Web into practice. Based on our experiences with building Semantic Web applications we conclude that such a management system will be a crucial cornerstone to bring together so far disjoint components.

KAON SERVER still is work in progress. We are currently developing aforementioned functional components like the ontology repository, semantic interoperability, OntoLiFT. The Web Service connector will be enhanced by semantic descriptions whose source is the registry. In the future, we envision to integrate means for information quality - a field of research that deals with the specification and computation of quality criteria. Users will then be able to query information based on criteria like "fitness for use", "meets information consumers needs", or "previous user satisfaction" [1.3]. We will also support aggregated quality values, which can be composed of multiple criteria.

References

1.1 Grosof, B., Horrocks, I., Volz, R., Decker, S., *Description Logic Programs: Combining Logic Programs with Description Logic*, Proceedings 12th International World Wide Web Conference (WWW12), Semantic Web Track, 2003, Budapest, Hungary.

1.2 Oltramari, A., Gangemi, A., Guarino, N., and Masolo, C., *DOLCE: a Descriptive Ontology for Linguistic and Cognitive Engineering (preliminary report)*, Proceedings 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW2002), Siguenza, Spain (in press).

1.3 Felix Naumann, *Quality-driven query answering for integrated information systems*, Lecture Notes in Computer Science, vol. 2261, Springer, 02 2002.

1.4 Yang-Hua Chu, Joan Feigenbaum, Brian A. LaMacchia, Paul Resnick, and Martin Strauss, *Referee: Trust management for web applications*, Proceedings of WWW6, 1997 (Ian F. Akyildiz and Harry Rudin, eds.), vol. 29, Computer Networks and ISDN Systems, no. 8-13, Elsevier, 1997, pp. 953–964.

1.5 Decker, S., Erdmann, M., Fensel, D., & Studer, R. (1999). Ontobroker: Ontology based access to distributed and semi-structured information. In et al., R. M. (Ed.), *Semantic Issues in Multimedia Systems*. Kluwer.

1.6 Haarslev, V., Moeller, R., *RACER System Description.*, Proceedings of Automated Reasoning, First International Joint Conference, IJCAR (Rajeev Goré

and Alexander Leitsch and Tobias Nipkow, eds.), vol. 2083, Lecture Notes in Computer Science, Springer, 2001, pp. 701-706.
1.7 O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification. Internet: http://www.w3.org/TR/REC-rdf-syntax/, 1999.
1.8 Dan Brickley and R. V. Guha. Resource description framework (RDF) schema specification 1.0. Internet: http://www.w3.org/TR/2000/CR-rdf-schema-20000372/, 2000.
1.9 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.
1.10 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.
1.11 Dave J. Beckett. The design and implementation of the redland rdf application framework. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 449–456, 2001.
1.12 Juergen Boldt. Corbaservices specification, 3 1997.
1.13 E. Bozsak, M. Ehrig, S. Handschuh, A. Hotho, A. Maedche, B. Motik, D. Oberle, C. Schmitz, S. Staab, L. Stojanovic, N. Stojanovic, R. Studer, G. Stumme, Y. Sure, J. Tane, R. Volz, and V. Zacharias. Kaon - towards a large scale semantic web. In *Proceedings of EC-Web 2002*. Springer, 2002.
1.14 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.
1.15 I. Horrocks. The fact system. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*. Springer, 1998.
1.16 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.
1.17 Michel Klein, Atanas Kiryakov, Damyan Ognyanov, and Dieter Fensel. Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02), Sigenza, Spain, October 1-4, 2002*, 2002.
1.18 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.
1.19 Deborah L. McGuinness. Proposed compliance level 1 for webont's ontology language owl. Knowledge Systems Laboratory, Stanford University.
1.20 Sergej Melnik. Rdf api. Current revision 2001-01-19.
1.21 Stefan Decker Michael Sintek. Triple - an rdf query, inference, and transformation language. In *In proceedings ISWC'2002*. Springer, 2002.
1.22 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.
1.23 Boris Motik, Daniel Oberle, Steffen Staab, Rudi Studer, and Raphael Volz. Kaon server architecture. Technical Report 421, University of Karlsruhe, Institute AIFB, 2002. http://www.aifb.uni-karlsruhe.de/WBS/dob/pubs/D5.pdf.

1.24 L. Stojanovic N. Stojanovic, R. Volz. A reverse engineering approach for migrating data-intensive web sites to the semantic web. In *IIP-2002, August 25-30, 2002, Montreal, Canada (Part of the IFIP World Computer Congress WCC2002)*, 2002.

1.25 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.

1.26 L. Stojanovic, N. Stojanovic, and R. Volz. Migrating data-intensive web sites into the semantic web. In *ACM Symposium on Applied Computing SAC 2002*, 2002.

1.27 Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. Ontoedit: Collaborative ontology development for the semantic web. In *Proceedings of the 1st International Semantic Web Conference (ISWC2002), June 9-12th, 2002, Sardinia, Italia*. Springer, 2002.

1.28 Raphael Volz, Daniel Oberle, and Rudi Studer. Views for light-weight web ontologies. In *Proceedings of the ACM Symposium on Applied Computing SAC 2003, March 9-12, 2003, Melbourne, Florida, USA*, 2003.