Ontobroker and OntoEdit Adaptation

Daniel Oberle[1], Dirk Wenke[2], Raphael Volz[1], Steffen Staab[1]

[1]University of Karlsruhe
Institute AIFB
D-76128 Karlsruhe
email: {lastname}@aifb.uni-karlsruhe.de

[2]Ontoprise GmbH
D-76227 Karlsruhe
email: {lastname}@ontoprise.de

Identifier	Del 9
Class	Deliverable
Version	1.0
Date	07-29-2003
Status	Final
Distribution	Public
Lead Partner	AIFB

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Contents

Executive Summary

This deliverable describes the adaptation of Ontobroker [2] and OntoEdit [12] to KAON SERVER (also known as OntoServer) — the project's main organisational and infrastructure kernel.

OntoEdit is a graphical ontology editor and as such a typical client of the KAON SERVER. It uses Ontobroker both as knowledge base with inferencing capabilities and for semantic validation. Our adaptation allows to work directly on a KAON [1] ontology store residing within the KAON SERVER.

Ontobroker is a well-known inference engine that is based on frame-logic [4]. Our adaptation allows to load it into KAON SERVER, thus facilitating reuse and development of possible clients. Both Ontobroker and OntoEdit are commercial products courtesy of Ontoprise GmbH, Karlsruhe, Germany.

We changed the original title ("SilRI and OntoEdit clients") for several reasons. First of all, SiLRI (Simple Logic-based RDF Interpreter) is not to become a client of the KAON SERVER but adapted by a proxy component. Second of all, we decided to adapt Ontobroker instead of SiLRI because it was an intermediate version of Ontobroker with RDF read capabilities. SiLRI has been superseeded by the development of new versions of Ontobroker taken place since the start of WonderWeb

1 Introduction

Building a complex Semantic Web application typically requires more than a single software module. Ideally the developer of such a system wants to easily combine different — preferably existing — software modules. So far, however, such integration had to be done ad-hoc, generating a one-off endeavour, with little possibilities for reuse and future extensibility of individual modules or the overall system.

The way out of this shortcoming is WonderWeb's main infrastructural kernel which we call *Application Server for the Semantic Web (ASSW)*. We already presented its architecture in deliverables D6 and D12. An Application Server for the Semantic Web facilitates reuse of existing modules, e.g. ontology stores, editors, and inference engines and, thus, the development and maintenance of comprehensive Semantic Web applications. It combines means to coordinate the information flow between modules, to define dependencies, to broadcast events between different modules and to translate between Semantic Web data formats. For WonderWeb, we are currently building one particular Application Server for the Semantic Web, called *KAON SERVER* which is part of the Karlsruhe Ontology and Semantic Web Toolsuite (KAON, cf. http://kaon.semanticweb.org).

This deliverable describes the adaptation of Ontobroker [2] and OntoEdit [12] to KAON SERVER. Both are written in Java and are commercial products courtesy of Ontoprise GmbH, Karlsruhe, Germany.

Ontobroker is a well-known inference engine that is based on frame-logic [4]. It is a typical software module that a developer might want to reuse when building a Semantic Web application. Hence, the aim of our adaptation is to make it deployable to the KAON SERVER, thus facilitating reuse and development of possible clients. We decided to adapt Ontobroker instead of SiLRI (Simple Logic-based RDF Interpreter) as it is outdated and technically just a wrapper around Ontobroker itself.

OntoEdit is a graphical ontology editor and as such a typical client of the KAON SERVER. It uses Ontobroker both as knowledge base with inferencing capabilities and for semantic validation. Hence, its datamodel is strongly connected to frame logic as well. Our adaptation allows to work directly on a KAON [1] ontology store residing within the KAON SERVER.

The document is basically divided in two parts: First, Section 2 discusses Ontobroker and its adaptation. Second, Ontoedit's adaptation is discussed in Section 3. In both cases we provide a short overview of the tools, discuss integration on a design level and talk about implementation details. Finally, we conclude and give an outlook in Section 4.

2 Adaptation of Ontobroker

This section discusses the adaptation of the Ontobroker inference engine. First, we will have a look at Ontobroker's client/server architecture. After the analysis, we will discuss the integration scenario that basically consists of a client-side surrogate for an Ontobroker component, i.e. Ontobroker wrapped such that it can be deployed to the KAON SERVER. Finally, we give some implementation details.

2.1 Ontobroker's Client/Server Architecture

Ontobroker may be seen as a middleware run time system to provide an information delivering base for intranet and extranet applications, for knowledge management systems for e-commerce systems and in general for intelligent applications. Ontobroker integrates the access to different information sources like databases, keyword based search engines etc. It reads various input formats like XML, OXML, RDF(S), F-Logic, Prolog. Thus it provides a homogeneous access to an inhomogeneous set of information sources and input formats. It provides compilers for different languages to describe ontologies, rules and facts. Ontobroker may be used as a server which reads input files containing facts and rules and which then evaluates queries sent to the server and sends the results back [9].

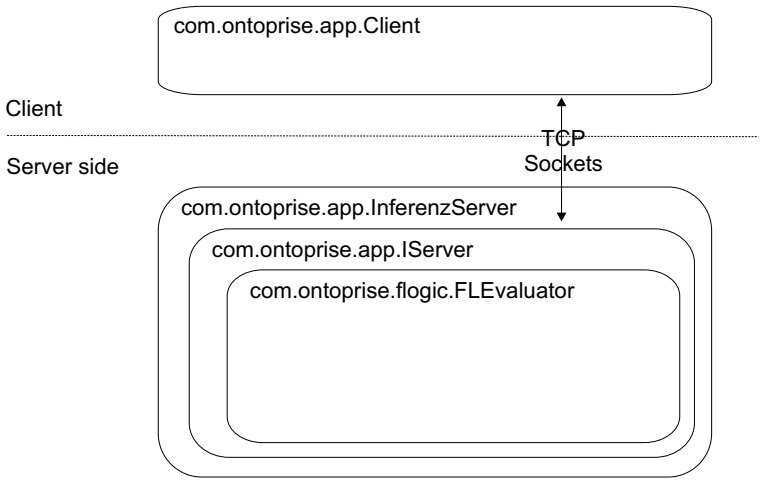

Figure 1: Client/Server architecture of Ontobroker

Ontobroker's client/server architecture is shown in Figure 1. The existence of a client-side surrogate class is of importance for the integration in KAON SERVER (cf. next subsection). We will explain the four depicted classes below:

com.ontoprise.flogic.FLEvaluator This class represents the actual inference engine. It is able to parse and load F-Logic facts, rules and allows for querying.

com.ontoprise.app.IServer Convenience class for the usage of FLEvaluator. Encapsulates an instance of FLEvaluator in order to avoid creating new instances every time. The class is able to receive and send messages via TCP sockets.

com.ontoprise.app.InferenzServer This rather thin class encapsulates an instance of I-Server and starts it.

com.ontoprise.app.Client Client-side surrogate for the inference engine. Basically, it sends commands and queries to an instance of IServer via TCP sockets. Commands are strings that conform to a certain syntax (cf. [9]).

2.2 Integration

KAON SERVER's Microkernel approach (cf. [13]) requires software modules[1] to be uniform in order to be loadable. Hence, software modules that shall be managed by the Microkernel have to be brought into a certain form. We call this process *making existing software modules deployable*, i.e. bringing existing software modules into the particular infrastructure of the KAON SERVER, that means wrapping it so that it can be plugged into the Microkernel. Thus, a software module becomes a *Component* which we define as software module that is deployable to the Microkernel. We use the word *deployment* as the process of registering, possibly initializing and starting a component to the Microkernel.

Like depicted in Figure 2, a software module either becomes a component (1) or proxy components have to be developed (2). A *Proxy Component* relays communication to a software module that cannot be made deployable, e.g. because it is written in another programming language. In that case, the software module remains unchanged but is called an *External Module* in our terminology to emphasize the fact that it lives outside the KAON SERVER.

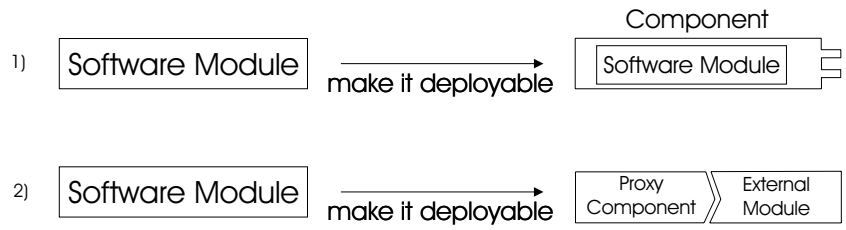

Figure 2: Software modules become components by making them deployable

In order to distinguish between components that are of direct interest to the developer and components providing functionality for KAON SERVER itself (e.g. a connector component or the registry), we call the first *Functional Components* and the latter *System Components*.

For the integration here, Ontobroker remains an external module and we make it deployable by a proxy component. We exchange the client-side surrogate com.ontoprise.-app.Client by our own. However, our replacement resembles the original one by offering the same methods to make life easier for the developer. The difference is that it communicates with any connector system component that is deployed to the KAON SERVER. By that, we are independent of the actual protocol used. The proxy component receives messages from a connector component and translates them to socket messages sent to the external module. Thus, we can leave existing code unchanged but have an additional indirection. The ideas are depicted in Figure 3. For more information on KAON SERVER's architecture and the benefits gained by such an adaptation please refer to [13].

[1] We use the term *software module* as software entity that fulfills a certain task. Examples for software modules in the Semantic Web are ontology and RDF stores or inference engines. A *software entity* is any executable code on a computer regardless of its state, i.e. we subsume both programs and processes (running programs) under this concept.

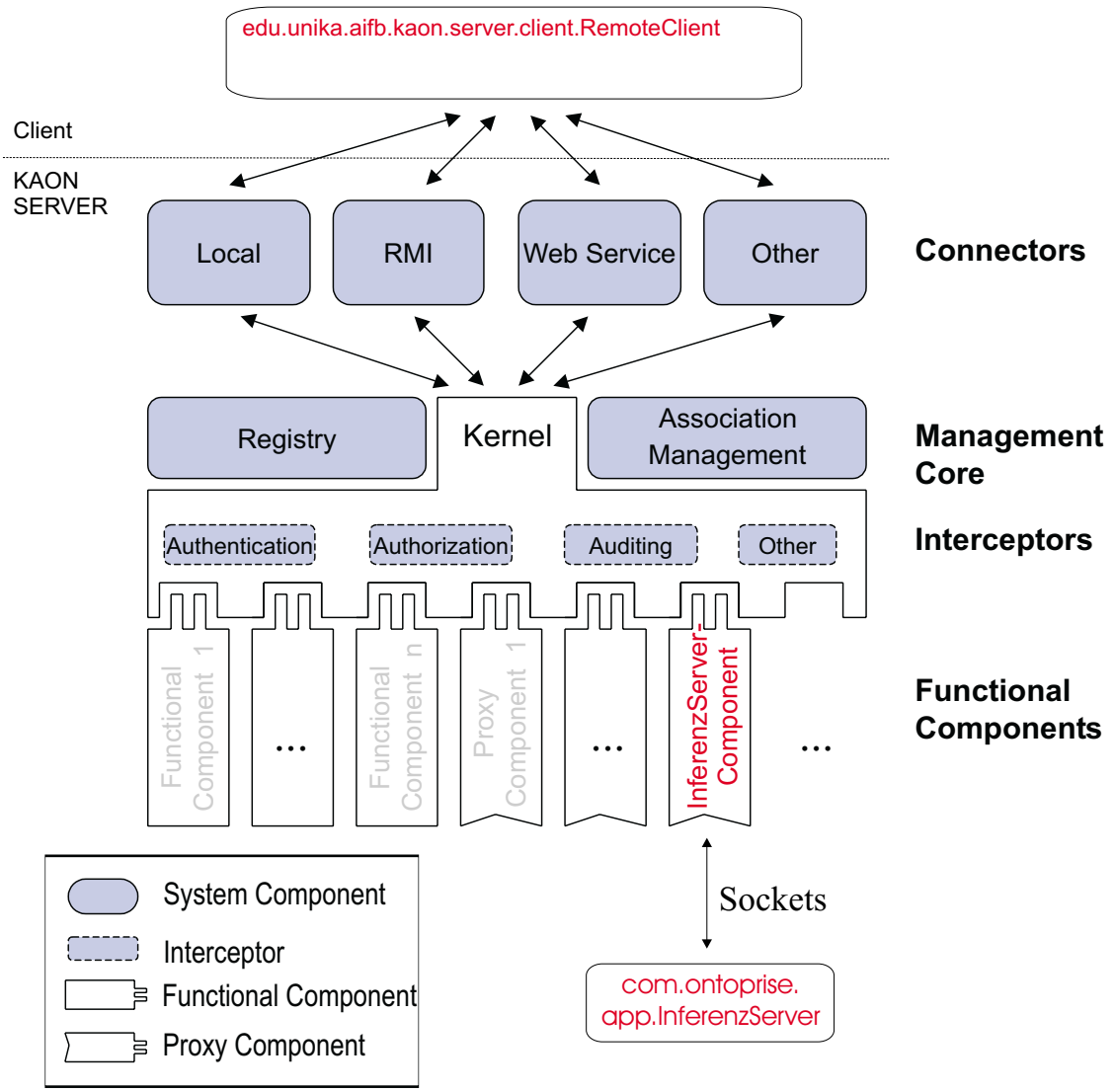

Figure 3: Adaptation of Ontobroker

2.3 Implementation

Proxy component and client-side surrogate are part of KAON SERVER's distribution and are freely available at http://sourceforge.net/projects/kaon-ext/. Both are implemented such that they do not require Ontobroker's class-files to compile.

The proxy component edu.unika.aifb.kaon.server.components.InferenzServerComponent is a regular MBean and offers three management operations:

- Vector query(query, host, port, netEncoding)
- String command(query, host, port)
- Boolean isStillEvaluating()

The management operations are called by the client-side surrogate (see paragraph below) and handle the communication to the actual InferenzServer via sockets. All three

operations are very much alike the operations of com.ontoprise.app.Client. In order to be independent of Ontobroker's classes, we copied the class com.ontoprise.app.fetch which is needed in the query operation. It became an internal class of InferenzServerComponent.

Our client-side surrogate edu.unika.aifb.kaon.server.client.RemoteClient offers the exact same methods like com.ontoprise.app.client[2]. Unfortunately, we could not let it implement the same interface com.ontoprise.app.clienttype as there would be a dependence to Ontobroker's class files. For the same reason, we copied com.ontoprise.util.Logfile to an internal class as it is needed for auditing.

Basically, we copied the com.ontoprise.app.client and just exchanged all the socket dependent communication parts. We had to change the three methods mentioned above which now talk to any connector component residing within the KAON SERVER.

A client would typically discover an InferenzServerComponent by querying KAON SERVER's registry. The result consists of all matching component IDs, i.e. MBean names. Our client-side surrogate has to be provided such an ID. Similar to hostname and port in sockets-based communication, such an ID is needed to specify which component should receive the method calls.

[2]By convention, class names of surrogate start with *Remote* and are followed by the original class name or the component name.

3 Adaptation of OntoEdit

This section discusses the adaptation of OntoEdit. In contrast to Ontobroker, the editor acts as a client to the KAON SERVER, i.e. it might want to discover components it is in need of. What we want to achieve is interoperation with ontology stores that conform to the KAON API, which is a Java application programmer's interface allowing access to ontological knowledge bases. OntoEdit should be capable of loading, editing and storing contents from a KAON ontology store. In the following subsections we provide the architectural details of OntoEdit as well as the KAON API. We demonstrate two possibilities of achieving our goal, i.e. interoperation of both. Finally, we decide for one possibility and give some implementation details.

3.1 Architectural details of OntoEdit

OntoEdit is an Ontology Engineering Environment supporting the development and maintenance of ontologies by using graphical means. OntoEdit is built on top of an internal ontology model. This paradigm supports representation-language neutral modelling as much as possible for concepts, relations and axioms. Several graphical views onto the structures contained in the ontology support modelling the different phases of the ontology engineering cycle [10].

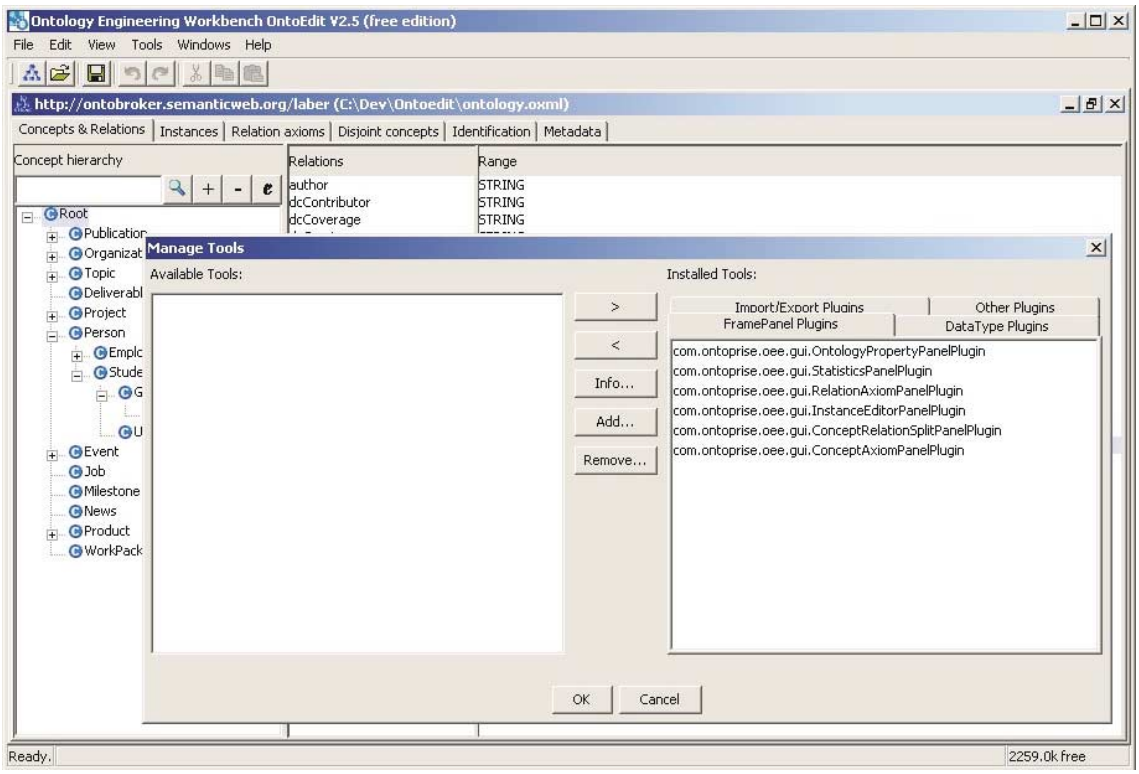

Figure 4: Screenshot of OntoEdit

Plug-in Framework The tool is based on a flexible plug-in framework that allows extending functionality in a modularized way. The plug-in interface is open to third parties which enables users to extend OntoEdit easily by additionally needed functionalities. Having a set of plug-ins available, like e.g. a domain lexicon, an inferencing plug-in and several export and import plug-ins, this allows for user-friendly customization to adapt the tool to different usage scenarios.

Each installed plug-in is notified when others are registered. By means of the service mechanism, each plug-in can discover and use services offered by other plug-ins. In order to build a plug-in, either the OntoPlugin interface must be implemented directly or one of its descendants (OntoPluginServiceConsumer or OntoPluginServiceProvider).

Each plug-in may provide menu entries within the menu bar and an icon within the toolbar which are connected to corresponding actions. Apart from its own menu actions, a plug-in can react to an OntoEdit standard event, if it has been registered appropriately. Further, it can share an event model with another plug-in based on the service mechanism. A plug-in can be in the role of a service-provider or a service-consumer, according to the Java interface it implements. A plug-in that implements the service-consumer interface registers it and intends to be notified of new services. A plug-in that implements the service-provider interface registers a new service within the framework. The framework notifies all currently registered service-users that a new service-provider plug-in has been added.

Datamodel OntoEdit's datamodel is influenced by F-Logic. Besides concepts, relations and instances, there exist predicates, predicate instances, axioms and modules as additional ontological entities. Predicates are n-ary associations and can be interpreted as a typed variant of predicates in the sense of first order logic (FOL). Predicates can be instantiated just as concepts and relations. Axioms in an ontology can model complex dependencies between concepts or instances that are beyond the general ontological primitives such as *is-a* or *instance-of*. In essence, they represent logical formulae, e.g. in FOL, Prolog, Datalog or F-logic. Modules are used in OntoEdit for convenience. They contain additional information that a plug-in wants to store along an ontology [3].

Speaking in technical terms, OntoEdit defines an API for accessing ontologies with each of the aforementioned entities being reflected by a corresponding object-oriented representation. There are several implementations of this API - the default implementation stores ontological entities in main-memory. Another applies Ontobroker for persistent storage. It is important to note that implementations of the datamodel are plug-ins themselves.

3.2 The KAON API

The KAON API is a set of interfaces offering access to KAON ontologies [7]. Basically, it is a client-side API that contains classes such as Concept, Property and Instance. The API reflects the capabilities of the KAON ontology language described in [6]. It decouples the user from actual ontology persistence mechanisms. Management of ontology changes is realized through pluggable evolution strategies [11], that the user can adapt according to his needs. To facilitate ontology reuse, KAON API supports inclusion. Each unit of

information in the API is called an OI-model, and it may contain concepts, properties and instances. Further, each OI-model may include any number of other OI-models and thus reuse ontology and instance definitions.

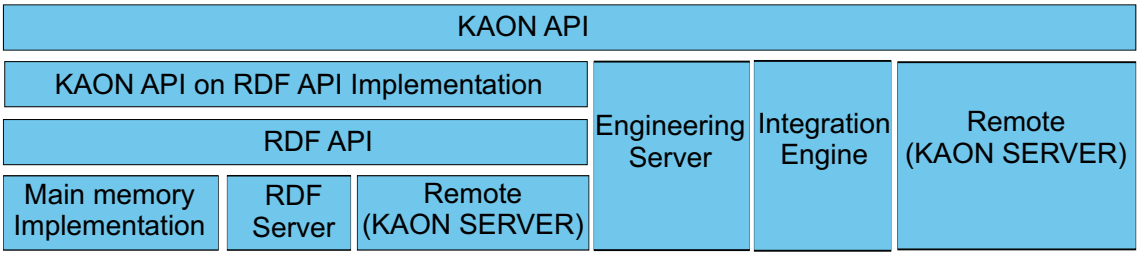

Figure 5: The KAON API and its implementations

Like depicted in Figure 5, there are several implementations of the KAON API, some of them for accessing RDF-based ontologies accessible through the KAON RDF API or ontologies stored in relational databases. We would like to discuss the different implementations in the following:

KAON API on RDF API Makes use of implementations of the KAON RDF API. The latter is similar to Sergej Melnik's original RDF API [5], however, it includes means for modularization and transactions. The KAON RDF API itself offers several implementations:

Main memory implementation This implementation comes closest to Sergej Melnik's implementation as it stores all statements in main memory and, hence, offers no persistence.

RDF Server Here, a relational database system is used for actual storage. Calls to methods are translated into respective SQL statements. Transactions and persistence are supported.

Remote (KAON SERVER) Implementations of the KAON RDF API have been made deployable to the KAON SERVER. This implementation can be considered as client-side surrogate that just connects to the respective functional component residing within the KAON SERVER.

Engineering Server The Engineering Server is the implementation of the KAON API that uses a scalable database representation for storing KAON ontologies. It is optimized for ontology engineering what is reflected in the physical database structure as well as in the implementation of API operations.

Integration Engine In contrast to the Engineering Server, whose schema is optimized for ontology engineering, this implementation lifts existing relational schemas to the ontology level.

Remote (KAON SERVER) Implementations of the KAON API have already been made deployable. This implementation can be considered as client-side surrogate that communicates with the corresponding functional component in the KAON SERVER.

3.3 Integration scenarios

Basically, there are two possibilities for interoperation of OntoEdit and KAON ontology stores. We will discuss both on a design level and decide for one in the next subsection.

3.3.1 RDF based integration

This scenario involves the development of so-called Import/Export plug-ins. OntoEdit comes with several of those, e.g. for the import of RDF files (RdfFileReader) or the export of an ontology into the DAML+OIL format (DamlFileWriter).

For the purpose of interoperation with KAON ontologies, a KaonRdfReader and a KaonRdfWriter are needed. Note, that both would not act on a file but on any implementation of the KAON RDF API (cf. Figure 5). Hence, we would have both integration with (KAON RDF stores deployed to) KAON SERVER and with existing KAON RDF stores. For example, KaonRdfReader could read from an KAON RDF API implementation residing within the KAON SERVER and KaonRdfWriter could save in another implementation.

The biggest drawback of this scenario is its inability to dynamically access the contents of an KAON RDF implementation. That means, the contents would be loaded into OntoEdit, changed by the user and finally stored back in a batch-manner. Hence, there are no possibilities for concurrency and during the implementation we might encounter parsing problems due to the different XML/RDF formats.

The basic steps of interoperation are shown in Figure 6. We explain the single steps in the enumeration below.

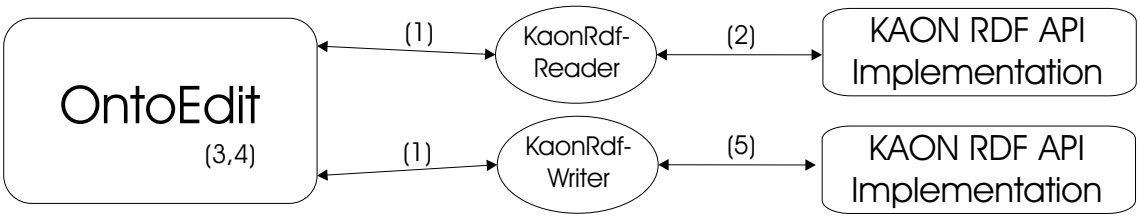

Figure 6: RDF based integration

1. In a first step, both KaonRdfReader and KaonRdfWriter plug-ins would have to be loaded into OntoEdit.

2. A KAON ontology could be loaded into OntoEdit by using the KaonRdfReader. For that, the user specifies the ontology to be loaded by providing a physical URI of the model (cf. [7]) as well as other parameters. The contents of the RDF model are serialized to XML within the plug-in by the KAON RDF Serializer.

3. OntoEdit receives an XML stream of RDF and converts it into its own datastructures.

4. The user is now able to edit the ontology as usual.

5. Finally, when editing is finished, the altered ontology can be stored in the same or another KAON RDF store. The user might have to provide the physical URI and other parameters in that step. OntoEdit serializes the ontology in XML and KaonRdfWriter uses a parser and eventually saves the ontology.

3.3.2 Datamodel API integration

Like discussed in subsection 3.1, implementations of com.ontoprise.datamodel are plug-ins themselves. Hence, the second possibility of interoperation is to create a new implementation of the datamodel whose methods translate to KAON API calls.

This possibility is more of an effort compared to the RDF based integration, however it allows dynamic access to KAON ontologies rather than batch-loading and -storing. In principle we are able to edit KAON ontologies concurrently and to preserve as much of the language's expressivity as possible. Also, the plug-in mechanism allows us to develop and distribute the new datamodel independent of OntoEdit.

Note that this type of integration also is independent of the actual KAON API implementation. That means, we can connect and edit a KAON ontology deployed in the KAON SERVER or any other existing implementation — most prominently this will be the RDF main memory implementation for simple file-based editing (cf. Figure 5).

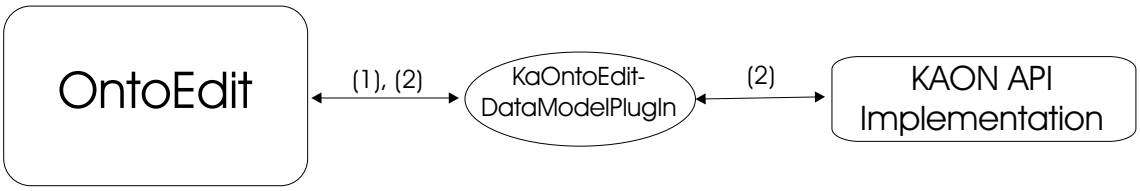

Figure 7: Datamodel API integration

The idea of the Datamodel API integration is shown in Figure 7.

1. The first step would be to load the new KaOntoEditDataModelPlugIn into OntoEdit.

2. After that, the user is able to load a KAON ontology by specifying its physicalURI (cf. [7]), which also determines the API implementation to be connected to, as well as other parameters. The KaOntoEditDataModelPlugIn cares for communication and translates every call to the KAON API. The user can start editing instantly and changes are processed dynamically.

3.4 Implementation

We decided to implement the Datamodel API integration as it allows dynamic access to KAON ontologies and thus to interact with the Engineering Server implementations of the KAON API. This decision involved the implementation of a new datamodel plug-in for OntoEdit which became part of the KAON Extensions project. The plug-in can be obtained from http://sourceforge.net/projects/kaon-ext/ or from http://www.ontoprise.de. Installation instructions are given in the KAON Extensions Developer's Guide [8].

The plug-in's functionality can be accessed in the "Tools" menu like shown in Figure 8. Similar to KAON's own graphical ontology editor, OIModeler, the user can open, create and save OIModels, i.e. KAON ontologies. The dialog shown in the picture allows to enter all required parameters for the chosen KAON API implementation. After a successful create or open, the user is able to edit the ontology in the usual way. Eventually, KAON ontologies may be saved or exported into OXML, DAML+OIL or F-Logic (depending on the installed export plug-ins).

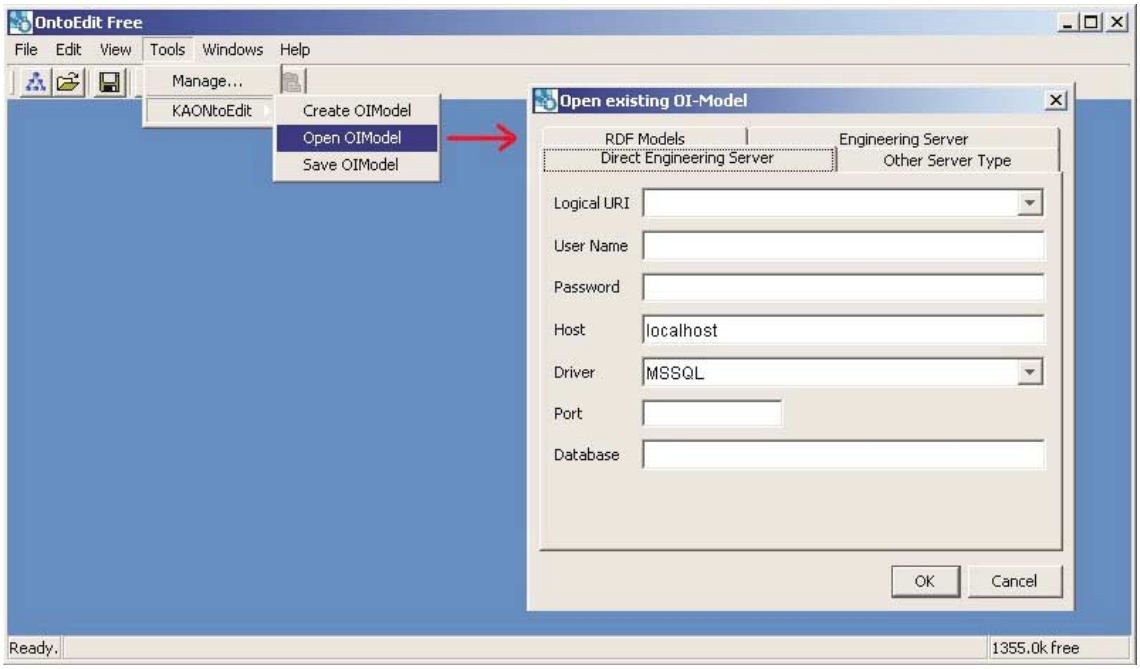

Figure 8: Screenshot of the Plug-in

The table below lists typical features of an ontology datamodel and compares them in OntoEdit's datamodel, KAON and the plug-in. In contrast to KAON's graphical user interface, the OIModeler, OntoEdit is able to open and edit only one ontology at a time.

Another difference affects properties and relations, respectively. Like in RDFS, KAON assumes a property-centric view, i.e. properties aren't attached to their domain and are treated as stand-alone entities. Hence, in KAON, a property can be linked to arbitrary domains and ranges. OntoEdit, however, assumes the object-oriented view and always links a relation to its domain concept. In addition, OntoEdit allows only one range - it is not possible to link a relation with the same domain and name to another range. Two relations have to created in this case. The Plug-in always treats the first range (as returned by methods of the KAON API) as *the* range in OntoEdit's datamodel.

The intersection of both Lexical Layers' capabilities results in the support of "documentation" and "external representation" (called "label" in KAON). For the latter, language identifiers are automatically transformed.

Implementations of the KAON API feature inherent inferencing regarding the concept and property-hierarchies as well as reasoning over symmetric, transitive and inverse properties. Inferencing in OntoEdit is possible only when Ontobroker is plugged in.

Feature Feature	OntoEdit's datamodel	KAON	Plug-in
Concepts	x	x	x (no disjoint concepts)
Properties	x (called relations)	x	x (symmetric, transitive, inverse)
Property Ranges	only 1 Range per property	property-centric, several ranges	1 range (first is displayed)
Attributes	several datatypes	string	string
Min/Max cardinalities	x	x (depending on API impl.)	x (depending on KAON API impl.)
Instances	x	x	x
Axioms	x	-	-
Predicates	x	-	-
Lexical Layer	x	x	x (only label and documentation)
Modularization	-	x	- (entities of included OIModel are shown)
Evolution	-	x	-
Meta-modelling	-	x	-
Ontology metadata	x	-	-

4 Conclusion

We have presented technical details of the adaptation of two wide-spread, commercially available tools to the project's main infrastructural kernel, the KAON SERVER. We are sure that the interoperation of OntoEdit and KAON ontology stores is an important merge of two decisive tools in the Semantic Web.

The adaptation of Ontobroker will facilitate the development and reuse of Semantic Web applications that are in need of the inference engine. In general, one gains a lot of benefits by making existing software modules deployable, i.e. bringing them in the particular infrastructure of an Application Server for the Semantic Web. First of all, the components can be deployed, undeployed, initialized, monitored etc. at runtime. Second of all, the registry stores ontological descriptions of all deployed components and allows a client to discover the ones it is in need of (when the client is aware of their interface, cf. next paragraph). Third, associations between components can be expressed within ontological descriptions and put in action by the Association Management component. E.g., event listeners between components can be applied. Furthermore, interceptors are a powerful means that provide a multitude of possibilities. For instance, transaction interceptors can be applied on several ontology stores, similar to what a transaction monitor does with several database systems, or semantic interoperation interceptors might seamlessly translate between Semantic Web languages. Finally, different connectors allow to reveal a component's functionality in different ways. Not only the low-level protocols may change (e.g. access via Java's Remote Method Invocation or Messaging Service), but also the paradigm. E.g. the methods of an ontology store might be offered as web service or even as peer services by using a JXTA connector.

However, making a software module deployable does not solve every problem. Most prominently, there still remains a close coupling between client and the component residing within the Application Server for the Semantic Web. That means, the client still has to be aware of the original software module's API which is revealed in the client-side surrogate. We claim that a loose coupling with automatic discovery and interaction will not be feasible. Nevertheless, we will investigate the semantic description of APIs and how such descriptions can be of help for the developer. We envision that API descriptions will be stored in the registry along other information and that the developer might manually discover a suitable component and then start programming against it. E.g., the developer might be in need of an ontology store with a functionality "store RDF". He/she could discover the functionality on a semantic level first and then look at the actual signature of the respective method which will probably differ from store to store.

An Application Server for the Semantic Web should also offer means for semantic interoperation, i.e. translating between different ontology languages. In general, interceptors are the first choice for the technical realization of semantic interoperation. E.g., an ontology store which is only aware of OWL could be deployed with a corresponding interceptor that seamlessly translates to F-Logic. The registry would then attribute the store as both capable of F-Logic and OWL. By doing so, the OWL store could be used by clients that speak F-Logic — OntoEdit would be the most prominent example. The translation of one language into another with as little information loss as possible is a research field on its own and it remains to be evaluated if interceptors can be developed

independent of client and component or if they have to be developed specifically for every client/component combination. In this deliverable we documented how the semantic interoperation of OntoEdit's datamodel and that used by the KAON tool suite is achieved. We came up with a solution that does not use interceptors but a plug-in especially designed for OntoEdit. The reason for this is that semantic interoperation should also be possible without the KAON SERVER.

To increase flexibility and interoperation in the OntoEdit/Ontobroker scenario, OntoEdit could dynamically discover instances of Ontobroker in the KAON SERVER by querying the registry in a next step. E.g., by specifying a query for Ontobroker already holding a certain ontology. In order to achieve this functionality, the existing datamodel implementation would have to be extended by a corresponding discovery procedure. In addition, OntoEdit could use a deployed Ontobroker also for semantic validation of its ontologies. We did not realize this functionality as the original, commercially available source-code would have to be adapted.

References

[1] 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*, E-Commerce and Web Technologies, Third International Conference, EC-Web 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings (Kurt Bauknecht, A. Min Tjoa, and Gerald Quirchmayr, eds.), Lecture Notes in Computer Science, vol. 2455, Springer, 2002.

[2] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer, *Ontobroker: Ontology based access to distributed and semi-structured information*, Database Semantics - Semantic Issues in Multimedia Systems, IFIP Conference Proceedings, vol. 138, Kluwer, 1998, pp. 351–369.

[3] Michael Erdmann, *OXML 2.0 — reference manual for users and developers of ontoedit's xml-based ontology representation language*, http://www.ontoprise.de, 04 2002.

[4] Michael Kifer, Georg Lausen, and James Wu, *Logical foundations of object-oriented and frame-based languages*, Journal of the ACM **42** (1995), no. 1, 741–843.

[5] Sergej Melnik, *RDF API*, Current revision 2001-01-19.

[6] B. Motik, A. Maedche, and R. Volz, *A conceptual modeling approach for building semantics-driven enterprise applications*, On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30 - November 1, 2002, Proceedings (Robert Meersman and Zahir Tari, eds.), Lecture Notes in Computer Science, vol. 2519, Springer, 2002.

[7] Boris Motik, *KAON — the Karlsruhe Ontology and Semantic Web framework — Developer's guide for KAON 1.2.5*, http://kaon.semanticweb.org, Feb 2003.

[8] Daniel Oberle, *KAON Extensions — extensions to the Karlsruhe Ontology and Semantic Web framework — Developer's guide for KAON Extensions 0.5*, http://kaon.semanticweb.org, Jul 2003.

[9] ontoprise GmbH, *How to use Ontobroker - Users and developers guide for the Ontobroker system version 3.6*, http://www.ontoprise.de, Apr 2003.

[10] _____, *How to work with OntoEdit - Users guide for OntoEdit version 2.6*, http://www.ontoprise.de, Apr 2003.

[11] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic, *User-driven ontology evolution management*, On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30 - November 1, 2002, Proceedings (Robert Meersman and Zahir Tari, eds.), Lecture Notes in Computer Science, vol. 2519, Springer, 2002.

[12] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke, *OntoEdit: Collaborative ontology development for the Semantic Web*, The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings (Ian Horrocks and James A. Hendler, eds.), Lecture Notes in Computer Science, vol. 2342, Springer, 2002.

[13] R. Volz, D. Oberle, S. Staab, and B. Motik, *KAON SERVER - a Semantic Web Management System*, Proceedings of the Twelfth International World Wide Web Conference WWW12, Alternate Tracks, Practice and Experience, 20-24 May 2003, Budapest, Hungary, 2003.