

# **KAON Extensions**

.

## **Extensions to the Karlsruhe Ontology and Semantic Web Framework**

**Developer's Guide for KAON Extensions 0.6**

**August 2003**

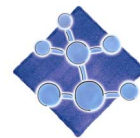
Knowledge Management Group (WBS),  
Institute AIFB,  
University of Karlsruhe,  
Germany

<http://www.aifb.uni-karlsruhe.de/WBS>

Research Group Knowledge Management (WIM)  
Research Center for Information Technologies  
at the University of Karlsruhe (FZI)

Germany

<http://www.fzi.de/wim>



---

<b>1</b>	<b>RELATIONSHIP TO KAON</b> .....	<b>3</b>
<b>2</b>	<b>KAON SERVER</b> .....	<b>5</b>
2.1	TERMINOLOGY.....	6
2.2	ARCHITECTURE.....	7
2.3	JAVA PACKAGES.....	9
2.4	THE REGISTRY.....	10
2.5	DEPLOYING MBEANS.....	11
2.6	CLIENT-SIDE SURROGATES.....	12
2.6.1	General information.....	12
2.6.2	KAON Surrogates.....	13
2.7	STARTING THE SERVER.....	15
<b>3</b>	<b>DESCRIPTION LOGIC PROGRAMS</b> .....	<b>16</b>
<b>4</b>	<b>KAONTOEDIT</b> .....	<b>17</b>
4.1	CAPABILITIES.....	17
4.2	INSTALLATION.....	18
<b>5</b>	<b>MAINTENANCE AND RESPONSIBILITY</b> .....	<b>20</b>

# 1 Relationship to KAON

KAON is an open-source ontology management infrastructure targeted for semantics-driven business applications. It includes a comprehensive tool suite allowing easy ontology management and application. Important focus of KAON is on integrating traditional technologies for ontology management and application with those used typically in business applications, such as relational databases. KAON is developed by [FZI](#) and [AIFB](#), at the University of Karlsruhe. For a detailed technical description please refer to the KAON Developer's Guide available at <http://kaon.semanticweb.org>

As the name suggests, KAON Extensions is a set of software modules that is optional to the KAON tool suite but relies on it. In contrast to KAON, the modules in KAON Extensions are disjoint, i.e. they do not rely on other modules within the project. However, they all rely at least on the kaonapi module from the KAON project. Hence, in order to work with a KAON Extensions module, the whole KAON project is needed (please cf. KAON Developer's Guide). Both projects should be checked out in parallel resulting a file-structure like follows:

```
<any dir>
| - build
|   | - kaon_build_root
|   |   | - apionrdf
|   |   | - ...
|   | - kaon-ext_build_root
|   |   | - dlp
|   |   | - ...
|
| ...
| - kaon
|   | - 3rdparty
|   | - apionrdf
|   | - apiproxy
|   | - ...
| - kaon-ext
|   | - 3rdparty
|   | - dlp
|   | - kaonserver
|   | - ...
```

Each module in KAON Extensions has its own `build.xml` in the corresponding directory (all of them include `common.xml` where some global constants are defined). The build files may reference some libraries from `<any dir>/kaon/3rdparty` or from `<any dir>/build/kaon_build_root/<module-dir>/lib`. The remaining libraries of all KAON Extensions modules are stored in `<any dir>/kaon-ext/3rdparty`.

A successful build results in a corresponding directory located at `<any dir>/build/kaon-ext_build_root/<module-name>`. Each module usually features a source and binary distribution zip-archive, javadocs, a copy of each required library and optionally some generated scripts.

KAON Extensions work with Sun's JDK 1.4.1\_03 (available at <http://java.sun.com/j2se/>). KAON Extensions' source code, as well as released binaries can be obtained from <http://sourceforge.net/projects/kaon-ext>.

## 2 KAON SERVER

KAON SERVER can be considered as an *Application Server for the Semantic Web (ASSW)* facilitating reuse of existing software 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.

In <http://www.aifb.uni-karlsruhe.de/WBS/pubs/dob/ACM.pdf> we describe analysis, design and implementation in detail. The following subsections will recap the terminology, the conceptual architecture and talk about the existing java packages, the registry, how to deploy an MBean, the client-side surrogates and about starting the server in more detail.

Note that KAON SERVER still is a prototype and under constant development. It is currently becoming WonderWeb's (an EU IST project, cf. <http://wonderweb.semanticweb.org>) main organisational and infrastructural kernel. The project runs until mid 2004 and so will probably the development of the KAON SERVER. Current versions can be checked out by anonymous CVS or one can download source and binary distributions from <http://sourceforge.net/projects/kaon-ext>.

KAON SERVER relies on *Java Management Extensions (JMX)*, cf. <http://java.sun.com/products/JavaManagement/>) 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 suited for management purposes. MBeans are hosted by an *MBeanServer* which allows their manipulation. All management operations performed on the MBeans are done through interfaces on the MBeanServer. We would like to point out two important methods of the MBeanServer:

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

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

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

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

An MBean must be a concrete and public Java object with at least one public constructor. An MBean must have a statically typed Java interface that explicitly declares the management attributes and operations. The naming conventions used in the MBean interface closely follow the rules set by the JavaBeans component model. To expose the management attributes, one has to declare get and set methods, similar to JavaBean component properties. The MBeanServer uses introspection on the MBean class to determine which interfaces the class implements. In order to be recognized as a Standard MBean, a class `x` has to implement an interface `xMBean`. Defining the methods `getAttr()` and `setAttr()` will automatically make `Attr` a management attribute, in this case with read and write access. Only management attributes can be accessed and modified by a client. All the other public methods will be exposed as management operations. Each MBean is accessible by an `javax.management.ObjectName` that takes the syntax `domain:attribute1=value1, ..., attributen=valuen`. Here, the domain should reveal the type of component and attribute/value pairs are limited to name. E.g., a proxy component for Ontobroker would be labelled by "Proxy Component:name=Ontobroker"

JMX only provides a specification. There are several implementations available. For the KAON SERVER, we have chosen *JBossMX*, a JMX implementation which is part of the comprehensive application server JBoss (cf. <http://www.jboss.org>). The reason for this decision is (a) it perfectly fits KAON SERVER requirements and (b) JBoss is open-source software.

## 2.1 Terminology

For our setting, we came up with a concise and comprehensive terminology which is crucial for the overall understanding and also important when working with the KAON SERVER. In the end, the server will use a management ontology for several tasks (description, deployment, discovery, implementation) in which most of the concepts below will be reflected:

**Software Entity:** Any executable code on a computer regardless of its state, i.e. we subsume both programs and processes (running programs) under this concept

**Software Module:** Software Entity that fulfills a certain task, e.g. storing, inferencing and/or implements an Application Programmer's Interface (API). An example for a software entity that is not a module: a program that consists of an endless loop.

**Component:** Software Module which is deployable to the Microkernel

**System Component** - Component providing functionality for the Application Server for the Semantic Web itself, e.g. 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 Module** - An external module 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 that relays communication to the external module.

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

**Surrogate**: Client-side object that reveals the API of the component residing within the ASSW and relays communication to it. Similar to stubs in CORBA.

**Microkernel**: Core of an Application Server for the Semantic Web offering a minimal set of functionality in the form of simple component management operations.

**Deployment**: Process of registering, possibly initializing and starting a component to the Microkernel.

**Interceptor**: Software entity that monitors a request and modifies it.

The following table depicts that some concepts are used on a design level and some are used only in the specific case of the KAON SERVER, i.e. on an implementation level:

Design	Implementation
Application Server for the Semantic Web	KAON SERVER
Component	MBean
Microkernel	MBeanServer of JBossMX

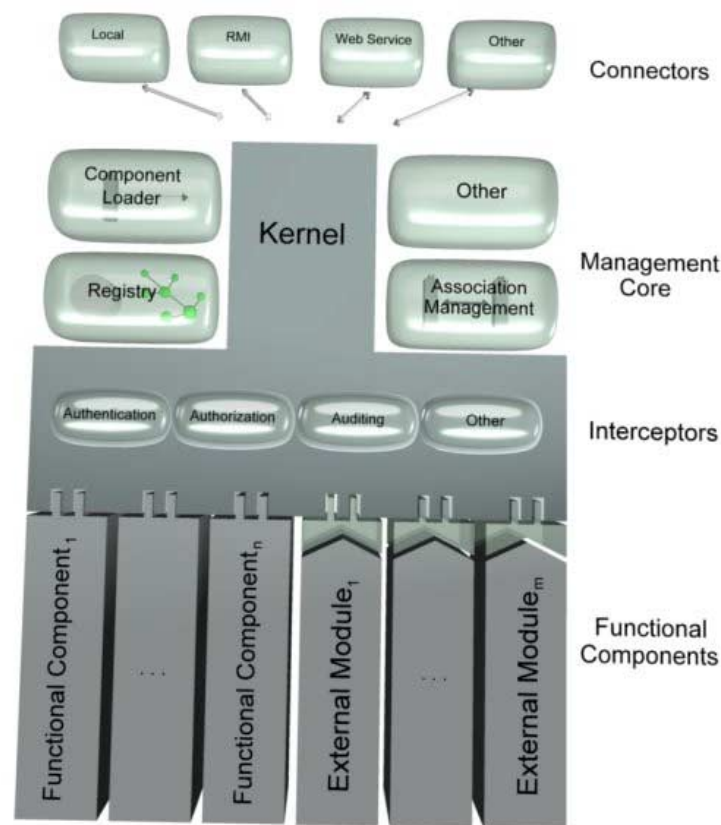
## 2.2 Architecture

The following paragraphs detail the KAON SERVER's conceptual architecture depicted in Figure 1. Note that not all of the described functionality has been implemented yet.

**Connectors** Connectors are system components. They send and receive requests and responses over the network. Aside from the option to connect locally, further possibilities exist for remote connection: e.g. ones that offer access via Java Remote Method Invocation (RMI), ones that offer access via Web Service protocols or ones that offer asynchronous communication. 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 Management Core is required to deal with the discovery, allocation and loading of components. The registry, a system component, manages descriptions of the components and facilitates the discovery of a

functional component for a client. Another system component called association management allows to express and manage relations between components. 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 component loader facilitates the deployment process for a client, e.g. by adding a description of the newly created component to the registry. System components can be deployed and undeployed ad hoc, so extensibility is also given for the Management Core. Further components are possible, e.g. ones that replicate and coordinate requests between a set of components, if the required functionality can only be provided jointly, or a cascading component that offers seamless access to the components deployed in another instantiation of an Application Server for the Semantic Web.



**Figure 1: KAON SERVER's Architecture**

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

**Functional Components** RDF stores, ontology stores etc., are finally deployed to the management kernel as functional components. In combination with the component loader, the registry can start functional components dynamically on client requests.

## 2.3 Java Packages

The Java packages of the project are organised akin to the conceptual architecture depicted above, i.e. they are divided in components, management, connectors and client (which holds all the client-side surrogates). We will explain the packages below:

`edu.unika.aifb.kaon.server` – only holds one interface `Constants` which, as the name suggests, contains all the constants needed throughout the project. If a class needs some constants, it just has to implement the interface.

`edu.unika.aifb.kaon.server.client` – contains all the client-side surrogates for components written so far (hence the package name “client”). All surrogates are entitled `Remote<original class name or component name>`. See section 2.6 for a description of the surrogates.

`edu.unika.aifb.kaon.connectors` – holds all the code needed for the connector MBeans.

`edu.unika.aifb.kaon.management` – holds system components that belong to the Management Core. At the moment this is only the Component Loader, as the registry is a KAON ontology store, whose component is situated in the components package. The Microkernel is implemented by JBoss' JMX (Java Management Implementations) MBeanServer, located in `javax.management` in `jboss-jmx.jar`.

`edu.unika.aifb.kaon.components` – contains classes and respective MBean interfaces for all functional and proxy components written so far.

`edu.unika.aifb.kaon.server.test` – here are all the clients for test purposes.

## 2.4 The Registry

The registry and its ontology play a central role in KAON SERVER. Please do note that the ontology is still under heavy construction and researched upon. In essence, the registry is a KAON ontology store. More precisely, we are using the apionrdf implementation of the KAON API, i.e. the main memory transient version. Components can be described according to the management ontology and those descriptions can be given as argument to the Component Loader (see section 2.5) which in turn enters them in the registry. One can use the OIModeler GUI to view the registry's current contents. In the "Open OIModel" dialog choose "Other" and enter as physicalURI (the last part is the UTF-8 encoded originalphysicalURI, see section 2.6.2.1): `mbean://localhost:1099?http%3A%2F%2Fkaon.semanticweb.org%2Fkaon%2Fserver%2Fregistry.kaon`. Additionally, the connection parameters `SERVER_URI` and `KAON_CONNECTION` must be provided (see also section 2.6.2.1).

The registry's surrogate (RemoteRegistry) basically wraps a RemoteKAONConnection (cf. section 2.6.2.1) to the main-memory based KAON API component deployed to the kernel and offers some convenience methods for interaction. The logical URI of the registry is <http://kaon.semanticweb.org/kaon/server/registry> which is basically an empty OIModel that includes the actual ontology/scheme <http://kaon.semanticweb.org/kaon/server/registryscheme>. Component descriptions are RDF files that in turn include the scheme above. The namespace to be used is always the scheme above. PhysicalURIs are usually logicalURI + ".kaon" The RemoteRegistry enters descriptions on the registry system component on the server side, i.e. it includes the description OIModel into <http://kaon.semanticweb.org/kaon/server/registry> (on server side!). You may have a look at the ontology with OIModeler by opening <http://kaon.semanticweb.org/kaon/server/registryscheme.kaon> (choose "RDF Models" in the "Open OIModel" dialog).

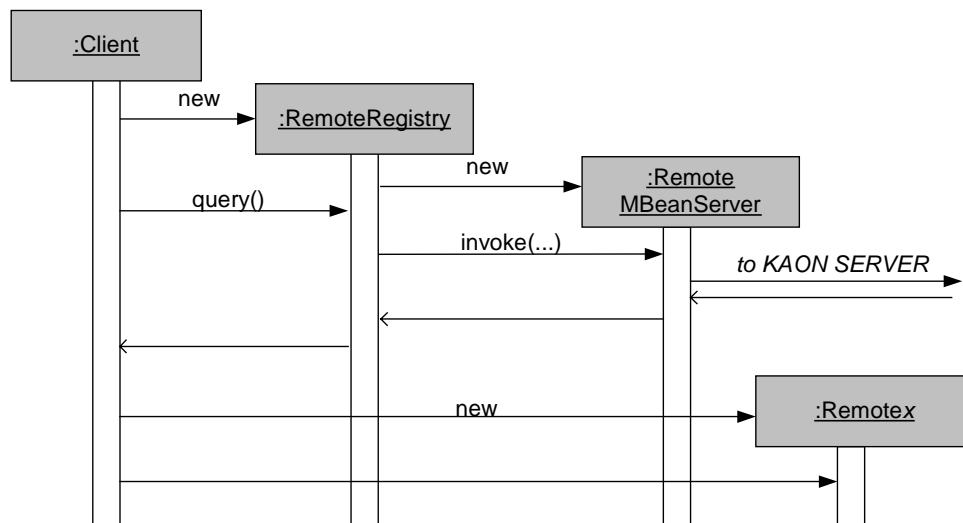


Figure 2: Sequence diagram - Registry interaction

Note, that there always is a close coupling, i.e. the client has to be aware of the desired component's surrogate (its libraries and classes). However, if there are several components deployed that are of the same type or the client is not aware of the component's name, it can use the registry to discover the components it is in need of. Like shown in Figure 2, the first step would be to instantiate `RemoteRegistry` which has to be provided with a connection property list (see 2.6). Its query method takes a KAON Query expression as argument (cf. KAON Developer's Guide). `RemoteRegistry` uses `RemoteMBeanServer` to translate to the kernel's `invoke` method and returns the results. Out of the results the client retrieves an `ObjectName`, identifying the MBean that matched the query. Together with a property list containing the connection parameters, the client is now able to construct the surrogate for the component it wants to use.

## 2.5 Deploying MBeans

Basically, there are two ways of deploying an MBean to `MBeanServer`. First of all, there exists the method of hard-coding the deployment in your code. Like shown in the example below, a new `javax.management.ObjectName` has to be created and given as argument to `registerMBean` together with the actual MBean. The MBean in our case is `remote` KAON `OIModel` (see section about surrogates). Note that this method of deployment does whether enter the MBean in the registry nor apply the association management.

```
ObjectName name = new ObjectName("Functional Component:name=KAONComponent1");
server.registerMBean(oimodel, name);
```

A second option is to use the component loader system component for convenience. Your client application has to create a surrogate for it in a first step. After that, the `deploy()` method takes an ontological description of the component which is automatically inserted into the registry. Also, associations to other components will be detected in future versions and automatically applied by using the association management system component.

```
Properties props = new Properties();
props.put(CONNECTION, RMI);
props.put(RMI_HOST, "localhost");
props.put(RMI_PORT, "1099");
props.put(RMI_NAME, "/jmx/RMIConnector");
RemoteComponentLoader rcl = new RemoteComponentLoader(props);
rcl.deploy("file:///C:/Dev/KAON-ext/config-rdfish.kaon");
```

The exemplary description `file:///C:/Dev/KAON-ext/config-rdfish.kaon` may look like below and conforms to the registry's ontology which is located at <http://kaon.semanticweb.org/kaon/server/registryscheme.kaon>. In essence, a component's description is made up of instances

```

<Component rdf:ID="Queue">
  <hasClassName>sample.standard.Queue</hasClassName>
  <versionTag>1.0</versionTag>
  <hasComponentName>
    <ComponentName>
      <domainName>Functional Component</domainName>
      <hasKeyValuePairs>
        <KeyValuePair>
          <key>name</key>
          <value>Queue</value>
        </KeyValuePair>
      </hasKeyValuePairs>
    </ComponentName>
  </hasComponentName>
  <requiresArchives>
    <JARArchive>
      <physicalURL>
        http://kaon.semanticweb.org/Members/
        rvo/KAONSERVER/testcomponents.jar
      </physicalURL>
      <versionTag>1.0</versionTag>
    </JARArchive>
  </requiresArchives>
</Component>

```

## 2.6 Client-side surrogates

### 2.6.1 General information

Like mentioned in section 2.1, surrogates are client-side objects that reveal the API of particular components residing within the KAON SERVER and relay communication to them similar to stubs in CORBA. The idea of a surrogate is to relieve the developer of tunneling all method-calls to an MBean via `MBeanServer.invoke()`. Instead, the developer should be put in a position equal to working with the software module directly. All surrogates are in `edu.unika.aifb.kaon.server.client`, they are labelled `Remote<original class name or component name>`.

Surrogates work with any connector component which has to be specified typically to the constructor in the form of `java.util.properties`. The following table gives an overview about possible connectors and their properties (all of them defined in `edu.unika.aifb.kaon.server.Constants`). At the moment, there are only three connectors: a local connector, a SOAP and a RMI connector. Besides the connection parameters, every surrogate has to be provided the name of the MBean to talk to. This name is typically discovered by querying the registry.

CONNECTION	Parameters	Possible values
LOCAL	No other parameter needed	-
SOAP	SOAP_HOST	Hostname of SOAP connector, default:localhost
	SOAP_PORT	Portnumber of SOAP connector's, i.e. KAON SERVER's host, default:8085
	SOAP_NAME	Name of the WSDL-description, e.g. soapconnector
	SOAP_PATH	Path of the WSDL-description, e.g. jmx
RMI	RMI_HOST	Hostname of RMI connector's, i.e. KAON SERVER's host, default:localhost
	RMI_NAME	Name of the RMI connector, default:jmx/RMConnector
	RMI_PORT	Portnumber of RMI host, default: 1099

We already gave an example of constructing a surrogate using the RMI connector in section 2.5. Additionally, we will demonstrate how to construct the properties needed for the surrogate "RemoteClient" which relays communication to the Ontobroker proxy component by using a SOAP connector. Its constructor takes the property list as well as the name of the MBean as argument. After instantiation the client is able to work with Ontobroker by using the surrogate object.

```

Properties props = new Properties();
props.put(CONNECTION, SOAP);
props.put(TYPE, MGMT);
props.put(SOAP_HOST, "localhost");
props.put(SOAP_PORT, "8085");
props.put(SOAP_PATH, "jmx");
props.put(SOAP_NAME, "soapconnector");
RemoteClient ontobroker=new RemoteClient(props, "Proxy
Component:name=Ontobroker");

```

Note, that there also is a surrogate for the MBeanServer itself to interact directly. Typically, all the other surrogates, like RemoteClient above, use RemoteMBeanServer to translate their calls to the MBeanServer's invoke method.

## 2.6.2 KAON Surrogates

KAON API and KAON RDF API implementations have been made deployable and surrogates have been developed, too. The surrogates themselves implement the KAON API and KAON RDF API respectively and relay communications to KAON API and KAON RDF API implementations

deployed to the KAON SERVER. So, there is an additional indirection with KAON SERVER in between. Relevant prefix for the API implementations' physicalURIs is `mbean://`

### 2.6.2.1 RemoteKAONConnection

The `RemoteKAONConnection` implements a KAON connection for KAON SERVER. On the server side, the actual `KAONConnection` is deployed as an `MBean`. PhysicalURIs for this kind of connection take the following syntax:

```
mbean://HOST[:PORT]?originalphysicalURI
```

`PORT` is optional and can be used to specify the RMI port for instance. The `originalphysicalURI` is the `physicalURI` of the `KAONConnection` wrapped by the `MBean` (`file`, `jboss`, `direct` etc.). Note, that the `originalphysicalURI` has to be UTF-8 encoded. This is achieved by the following code `new URI("mbean://<host>[:<port>]" + URLEncoder.encode( <original-physical-URI> , "UTF-8" ));`. `RemoteKAONConnection` needs a `SERVER_URI` as additional parameter that holds the information needed to connect to the `MBeanServer`. It takes the following syntax:

```
mbean://CONNECTION@HOST:PORT/PATH/JMX-DOMAIN?JMX-ATTR-LIST
```

where

<code>CONNECTION</code>	is either <code>LOCAL</code> , <code>RMI</code> or <code>SOAP</code>
<code>HOST</code>	is either <code>localhost</code> , <code>RMI_HOST</code> or <code>SOAP_HOST</code>
<code>PORT</code>	is either <code>0</code> , <code>RMI_PORT</code> or <code>SOAP_PORT</code>
<code>PATH</code>	is either <code>RMI_NAME</code> or <code>SOAP_NAME</code>
<code>JMX-DOMAIN</code>	is the domain of the <code>MBean</code>
<code>JMX-ATTR-LIST</code>	are the attributes of the <code>MBean</code> 's JMX name

For example, a typical `physicalURI` with `SOAP` access would look like the following `mbean://SOAP@localhost:8085/jmx/soapconnector/example?Functional%20Component=KAONComponent1`. In addition to `SERVER_URI`, the `KAON_CONNECTION` parameter must be set to `"edu.unika.aifb.kaon.server.client.RemoteKAONConnection"`.

### 2.6.2.2 RemoteRDFFactory

An implementation of the RDF factory for remote models residing in the KAON SERVER. Relevant prefix for this kind of physicalURI is "mbean". All information needed to address the remote `MBeanServer` is encoded in the `physicalURI`:

```
mbean://CONNECTION@HOST:PORT/PATH/JMX-DOMAIN?JMX-ATTR-LIST
```

where

---

CONNECTION	is either LOCAL, RMI OR SOAP
HOST	is either localhost,RMI_HOST OR SOAP_HOST/tr>
PORT	Is either 0,RMI_PORT OR SOAP_PORT
PATH	Is either RMI_NAME OR SOAP_NAME
JMX-DOMAIN	Is the domain of the MBean
JMX-ATTR-LIST	are the attributes of the MBean's JMX name

For example, a typical physicalURI with SOAP access would look like the following  
mbean://SOAP@localhost:8085/jmx/soapconnector/example?Functional%20  
Component=RDFComponent1

Within RemoteRDFFactory's methods, this URI would be parsed into a valid JMX-name  
Functional Component:name=RDFComponent1 to address the MBean within the server. All the  
other information is needed to instantiate the SOAPConnector. Before using this surrogate, the  
factory should be registered with the RDFManager:

```
RDFManager.registerFactory( "edu.unika.aifb.kaon.  
server.client.RemoteRDFFactory");
```

## 2.7 Starting the server

At the moment, the best method to start up the server with all connectors and the management  
core's system components is the class `edu.unika.aifb.kaon.server.test.Prototype`. In the  
build or binary release, there is a script automatically starting this class in `<any dir>/build/  
kaon-ext_build_root/kaonserver/release/bin/startserver.bat`. Note that this script  
includes the invocation of the RMI registry which is needed for the RMI connector. If you don't  
use the script but starting the class `Prototype` by hand, you have to start the RMI registry manually  
with a proper classpath set to KAON SERVER's classes. `startserver.bat` takes as argument a  
KAON ontology identified by a file physicalURI. If it is provided and found that script deploys a  
RDF and KAON component to play around with. The script also starts a HTTP Adaptor GUI, i.e.  
a web-accessible management console, which shows all deployed Mbeans, their management  
attributes and methods. You can access the web frontend at `localhost:8082`

## **3 Description Logic Programs**

## 4 KAONtoEdit

KAONtoEdit allows to work directly on implementations of the KAON API with the graphical user interface *OntoEdit* from *ontoprise* (cf. [www.ontoprise.de](http://www.ontoprise.de)). *OntoEdit* builds on a plug-in architecture and defines its own datamodel which is closely related to F-Logic. Implementations of *OntoEdit*'s datamodel API are plug-ins themselves. E.g., there is a main memory based implementation as well as an implementation that uses *Ontobroker* for persistence. Likewise, *KAONtoEdit* is a plug-in and implements *OntoEdit*'s datamodel on top of the KAON API. In essence, methods are mapped from one API to the other, letting the plug-in act like a wrapper.

There are several ways to obtain the plug-in (cf. also installation guide in 4.2):

1. CVS. Use the anonymous CVS account to KAON-Ext Sourceforge and check out the project
2. Source Distribution. Download the source distribution archive from Sourceforge <http://prdownloads.sourceforge.net/kaon-ext/kaontoedit-src-07-24-2003.zip?download> and execute the build-file with Jakarta Ant
3. Binary Distribution. Download the binary distribution from Sourceforge <http://prdownloads.sourceforge.net/kaon-ext/kaontoedit-bin-07-24-2003.zip?download> or from [www.ontoprise.de](http://www.ontoprise.de)

The plug-in has been successfully tested with JDK 1.4.1\_03, *OntoEdit* 2.6.4 (free and professional versions) and KAON's binary distribution (15<sup>th</sup> July 2003), in particular with the implementation on the RDF API and the Direct Engineering Server.

### 4.1 Capabilities

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 *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 be 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, e.g. "en" is transformed into "<kaon-lexical-ns>#en".

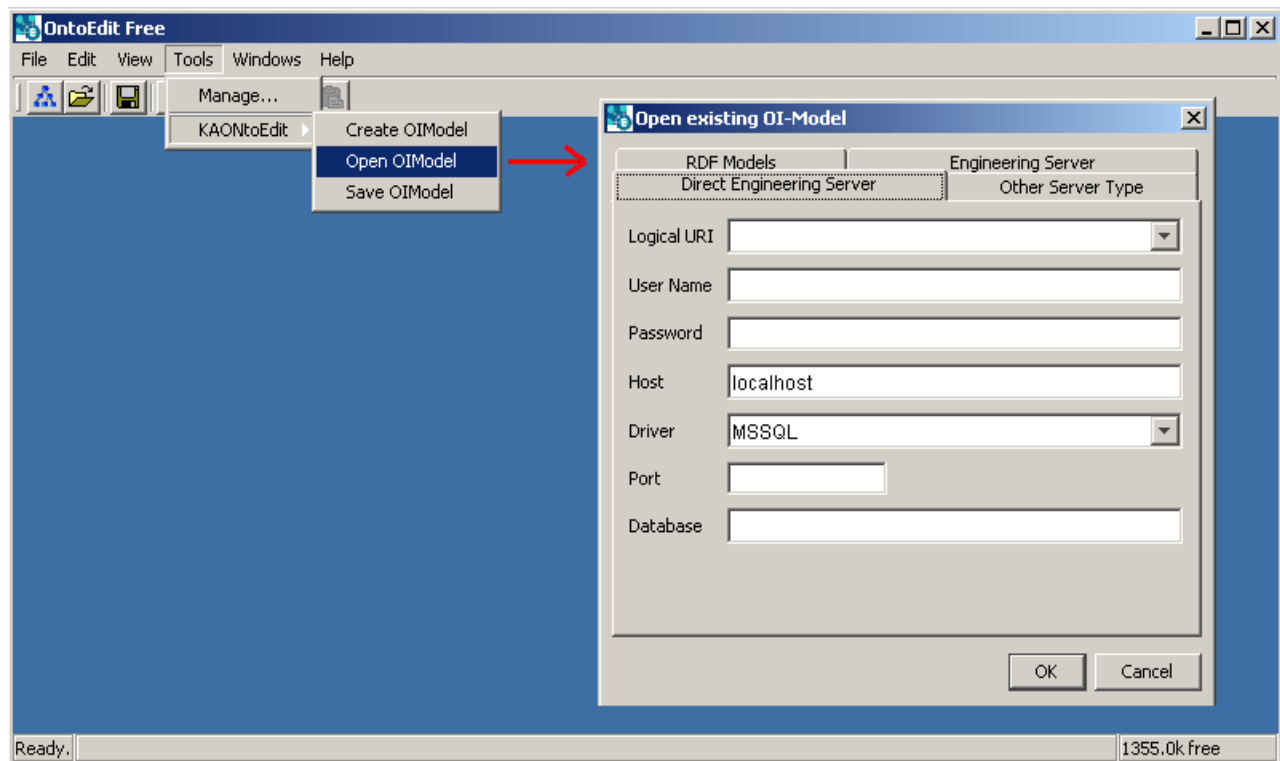
Implementations of the KAON API feature inherent inferencing regarding the concept and property-hierarchies as well as reasoning over symmetric, transitive and inverse properties. Inferencing in OntoEdit is possible only when Ontobroker is plugged in.

Feature	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 Relation	Property-centric, several ranges	1 Range (first one is displayed when more)
Attributes	Several Datatypes	String	String
Min-/Max Cardinalities	X	X (depending on API implementation)	X (depending on KAON API implementation)
Instances	X	X	X
Axioms	X	-	-
Predicates	X	-	-
Lexical Layer	X	X	X (only documentation and label)
Modularization	-	X	- (entities of included OIModels are shown)
Evolution	-	X	-
Meta-Modelling	-	X	-
Ontology-Metadata	X	-	-

## 4.2 Installation

1. Download the current binary distribution (usually named kaontoedit-bin-<date>.zip) at Sourceforge
2. Unzip the included jar-files in your OntoEdit-directory. The jar-files will be placed into <Ontoedit-dir>/kaontoedit
3. Edit your batch-file. Open <Ontoedit-dir>/ontoedit.bat in an editor and add all the jar-files of step 2 to the classpath. E.g. "java -cp kaontoedit/apionrdf.jar;kaontoedit/apiproxy.jar; ..."
4. Start OntoEdit by invoking the batch-file.

5. Uninstall the mainmemory datamodel. Click on Tools/Manage, then on the tab "Other PlugIns". Choose com.ontoprise.datamodel.cache.CachedOntologyPlugIn and click on "<".
6. Install the KAONtoEdit PlugIn. In the manage-dialog click on "Add" and type "edu.unika.aifb.kaon.ontoedit.K2OPlugin". If all goes well, the plugin will be listed in available tools. Choose the entry and click ">".
7. Restart OntoEdit (to be 100% sure, should work without restarting however).
8. In the Manage-Menu there should now be an entry "KAONtoEdit" with the typical options "Create OIModel", "Open OIModel", "Save OIModel". The subsequent dialogs are similar to those from KAON's OIModeler. File -> Open Ontology and File -> Import cannot be used.
9. Opened or created OIModels can easily be stored as OXML, DAML+OIL, Flogic or other data-formats (depending on installed Import/Export Plug-Ins) by using File -> Save and File -> Export, respectively.



## 5 Maintenance and responsibility

Module	Since	Current	Maintainer(s)
DLP	4/2003	5-30-03	Raphael Volz <a href="mailto:rvo@aifb.uni-karlsruhe.de">rvo@aifb.uni-karlsruhe.de</a> Boris Motik <a href="mailto:motik@fzi.de">motik@fzi.de</a>
KAON SERVER	1/2003	7-02-03	Daniel Oberle <a href="mailto:dob@aifb.uni-karlsruhe.de">dob@aifb.uni-karlsruhe.de</a> Raphael Volz <a href="mailto:rvo@aifb.uni-karlsruhe.de">rvo@aifb.uni-karlsruhe.de</a>
KAONtoEdit	7/2003	7-24-03	Daniel Oberle <a href="mailto:dob@aifb.uni-karlsruhe.de">dob@aifb.uni-karlsruhe.de</a> Dirk Wenke <a href="mailto:wenke@ontoprise.de">wenke@ontoprise.de</a>